

# Integration von Prolog-Modulen in eine Microservice-Architektur

Masterarbeit

von

Fabian Stolz

Studiengang: Informatik

Institut für Programmstrukturen und Datenorganisation (IPD)

KIT-Fakultät für Informatik

Prüfer:	Prof. Dr. Ralf Reussner
Zweiter Prüfer:	Prof. Dr. Andreas Oberweis
Betreuer:	Dipl.-Inform.Wirt Sascha Alpers
Eingereicht am:	2. November 2018

# Erklärung

*Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.*

Karlsruhe, 02. November 2018

Fabian Stolz

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Zielsetzung . . . . .	2
1.2	Aufbau der Arbeit . . . . .	2
<b>2</b>	<b>Grundlagen</b>	<b>5</b>
2.1	Geschäftsprozessmodellierung . . . . .	5
2.2	Petri Net Markup Language . . . . .	6
2.3	Prolog . . . . .	6
2.3.1	Terme . . . . .	7
2.3.2	Fakten und Regeln . . . . .	8
2.3.3	Abfragen, Unifikation und Backtracking . . . . .	9
2.3.4	Ausführbare und interaktive Programme . . . . .	11
2.3.5	Arity/Prolog . . . . .	13
2.4	PASIPP . . . . .	13
2.4.1	Beschreibung von Petri-Netzen . . . . .	14
2.4.2	Darstellung von Erreichbarkeitsbäumen . . . . .	18
2.4.3	Benutzung des Programms . . . . .	20
2.5	Microservices . . . . .	22
2.5.1	Shared Nothing . . . . .	24
2.5.2	Smart Endpoints, Dumb Pipes . . . . .	24
2.5.3	Team-Organisation . . . . .	24
2.5.4	DevOps . . . . .	25
2.5.5	Skalierbarkeit . . . . .	27
2.5.6	Polyglot Programming . . . . .	28
2.5.7	Dezentralisiertes Datenmanagement und Polyglot Persistence . . . . .	29
2.6	Vorgeschlagene Microservice-Architektur . . . . .	31
2.6.1	Architektur . . . . .	31
2.6.2	Implementierung . . . . .	32
2.7	REST . . . . .	32

<b>3</b>	<b>Konzept</b>	<b>35</b>
3.1	Architektur . . . . .	35
3.1.1	Simulator . . . . .	35
3.1.2	Server . . . . .	39
3.2	Verwendung von Prolog und PASIPP . . . . .	40
3.3	Ein- und Ausgabeformate . . . . .	42
3.4	Benutzung des Services . . . . .	42
3.4.1	Installation und Ausführung . . . . .	45
3.4.2	hasCycles . . . . .	47
3.4.3	fireableTransitions . . . . .	48
3.4.4	simulateStep . . . . .	49
3.4.5	reachabilityTree . . . . .	50
3.4.6	Komprimierte Rückgabe . . . . .	54
3.5	Fehlerbehandlung . . . . .	57
<b>4</b>	<b>Implementierung und mögliche Erweiterungen</b>	<b>59</b>
4.1	Anpassungen an PASIPP . . . . .	59
4.2	Wettlaufsituation . . . . .	60
4.3	Möglichkeiten zur zukünftigen Erweiterung des Services . . . . .	62
4.3.1	Erweiterungen der Simulator-Komponente . . . . .	62
4.3.2	Erweiterungen der Server-Komponente . . . . .	63
<b>5</b>	<b>Evaluierung</b>	<b>67</b>
5.1	hasCycles . . . . .	68
5.2	fireableTransitions und simulateStep . . . . .	68
5.3	reachabilityTree . . . . .	71
<b>6</b>	<b>Fazit und Ausblick</b>	<b>77</b>

# Abbildungsverzeichnis

1	Termtypen und ihre Beziehung in Prolog . . . . .	7
2	Beispiel-Petri-Netz . . . . .	16
3	Graphische Darstellung des Erreichbarkeitsbaums aus Codeausschnitt 13 .	20
4	Beispiel-Petri-Netz mit potenziell unendlich vielen Markierungen (links) und der entsprechende Erreichbarkeitsbaum (rechts) . . . . .	20
5	Startbildschirm von PASIPP (ausgeführt unter vDos) . . . . .	22
6	Hauptmenü von PASIPP (ausgeführt unter vDos) . . . . .	23
7	Beziehung zwischen Team-Organisation und Software-Architektur [2] . . .	25
8	Abbotts Skalierbarkeitswürfel ([18], vgl. [1]) . . . . .	28
9	Skalierung bei monolithischen und Microservice-Architekturen [9] . . . . .	29
10	Datenhaltung bei monolithischen und bei Microservice-Architekturen [9] .	30
11	Microservice-Architektur des BPM-Editors [2] . . . . .	33
12	Unterteilung des Microservices in eine <i>Server</i> - und eine <i>Simulator</i> -Kompo- nente . . . . .	35
13	Architektur der <i>Simulator</i> -Komponente (Klassendiagramm) . . . . .	36
14	Architektur der <i>Server</i> -Komponente (Klassendiagramm) . . . . .	39
15	Veranschaulichung der öffentlichen Methoden der Java-Klasse <i>PrologWrap- per</i> (Konstruktor, <i>void callPrologCommand(String command)</i> , <i>String get- PrologResponse()</i> , <i>void close()</i> ) als Sequenzdiagramm . . . . .	43
16	Beispiel-Petri-Netz . . . . .	45
17	Beispiel-Petri-Netz (links) mit entsprechendem Erreichbarkeitsbaum (rechts)	52
18	Screenshot des Programms yEd mit dem Erreichbarkeitsbaum für Abbil- dung 2 (Seite 16), ohne Label . . . . .	55
19	Screenshot des Programms yEd mit dem Erreichbarkeitsbaum für Abbil- dung 2 (Seite 16), mit Label . . . . .	55
20	Beispiel-Petri-Netz (Bild aus dem HORUS Business Modeler) . . . . .	67
21	Beispiel-Petri-Netz mit Zyklus (Bild aus dem HORUS Business Modeler) .	68
22	Erreichbarkeitsbaum für das Petri-Netz aus Abbildung 20 . . . . .	73
23	Erreichbarkeitsbaum für das Petri-Netz aus Abbildung 21 . . . . .	74

24	Erreichbarkeitsbaum für das Petri-Netz aus Abbildung 20, mit anfänglich 2 Tokens in der Stelle "Potenzielle Kunden" . . . . .	75
25	Erreichbarkeitsbaum für das Petri-Netz aus Abbildung 20, mit anfänglich 3 Tokens in der Stelle "Potenzielle Kunden" . . . . .	75

## Tabellenverzeichnis

1	Zuordnung von Namen und IDs der Stellen für die Netze in Abbildung 20 und 21 . . . . .	68
2	Zuordnung von Namen und IDs der Transitionen für die Netze in Abbildung 20 und 21 . . . . .	70





## Codeausschnittverzeichnis

1	Beispiel-Fakten . . . . .	8
2	Variablen in Fakten . . . . .	8
3	Regeln zur Definition von Eltern- und Großelternbeziehung . . . . .	9
4	Rekursive Definition der Vorfahren-Regel . . . . .	9
5	Definition von Eltern-Beziehungen . . . . .	10
6	Modul-Definition von PASIPP.ARI [16] . . . . .	12
7	Aufbau einer <i>marke</i> -Regel (vgl. [16]) . . . . .	15
8	Aufbau einer <i>fire</i> -Regel (vgl. [16]) . . . . .	15
9	Petri-Netz in PNML . . . . .	17
10	Struktur des Netzes in Prolog . . . . .	17
11	Markierung des Netzes in Prolog . . . . .	18
12	Pseudocode-Algorithmus zur Konvertierung von PNML- in Prolog-Notation	19
13	Erreichbarkeitsbaum in Prolog-Notation . . . . .	19
14	PASIPP-Darstellung des Erreichbarkeitsbaums aus Abbildung 4 . . . . .	21
15	Regeln <i>geben</i> und <i>godd</i> zur Generierung von geraden beziehungsweise un- geraden Ganzzahlen (vgl. [21]) . . . . .	41
16	Petri-Netz in PNML . . . . .	44
17	GraphML-Erreichbarkeitsbaum für Abbildung 17 . . . . .	53
18	Vergleich eines Netzmarkierungsknotens mit und ohne Labelinformation . .	56
19	Fehlerrückgabe . . . . .	58
20	Änderung der Methode <i>read_string</i> (Datei SPECIAL.ARI) . . . . .	59
21	Die Regel <i>weiter</i> und die Änderung der Regel <i>ausgabe_one</i> (beide in der Datei HELP1.ARI) . . . . .	61
22	Zeitabhängige Benennung der PASIPP-Eingabedateien (PnmlToPrologCon- verter.java) . . . . .	62
23	Struktur der öffentlichen Methoden der Klasse <i>Simulator.java</i> (mit erklä- renden Kommentaren) . . . . .	64
24	Struktur der HTTP Ressource-Klassen (mit erklärenden Kommentaren) . .	66
25	Rückgabe der Funktion <i>hasCycles</i> für das Petri-Netz aus Abbildung 20 . .	69
26	Rückgabe der Funktion <i>hasCycles</i> für das Petri-Netz aus Abbildung 21 . .	69
27	Rückgabe der Funktion <i>fireableTransitions</i> für das Petri-Netz aus Abbil- dung 20 . . . . .	69
28	Token Game für das Petri-Netz aus Abbildung 20. . . . .	72



# 1 Einleitung

Die Entwicklungszyklen von Softwaresystemen werden immer kürzer und kürzer, gleichzeitig werden immer mehr Anwendungen als verteilte Softwaresysteme realisiert. Traditionelle monolithische Programme haben sich als schlecht geeignet für diese Anforderungen erwiesen. Dies liegt zum einen daran, dass sie oft nur als Ganzes skalierbar sind, und man nicht gezielt den Flaschenhals eines Systems skalieren kann. Andererseits liegt es daran, dass Änderungen an einem Teil des Systems oft Änderungen im ganzen System nach sich ziehen. Auch ist es schwer, bei der Entwicklung neuer Systeme relevante Teile einer bestehenden Software zu übernehmen, da diese oft Abhängigkeiten zum Rest der Software haben.

In letzter Zeit findet das Software-Architekturmuster Microservices eine immer weitere Verbreitung. Bei dem Muster geht es darum, große Systeme als Sammlung vieler kleiner Dienste zu realisieren. Die einzelnen Dienste laufen unabhängig voneinander, haben einen definierten Verantwortungsbereich und kommunizieren über definierte Schnittstellen miteinander. Die Kommunikation findet meist über leichtgewichtige Technologien wie HTTP oder HTTPS statt.

Diese Aufteilung verbunden mit der unabhängigen Ausführung der Dienste hat den Vorteil, dass einzelne Teile der Software gezielt skaliert werden können, dass die einzelnen Teile - ohne den Rest der Software zu beeinflussen - geändert oder sogar neu geschrieben werden können, und dass es leichter ist, bestimmte Teile der Microservice-Architektur in zukünftigen Projekten weiter zu verwenden.

Außerdem können Microservices die Weiternutzung bestehender Software in neuen Softwareprojekten vereinfachen. Die bestehende Software kann dazu in einem Service gekapselt werden, der die Funktionalität dieser Software in einem feingranularen Interface nach außen hin anbietet.

Dies hat den Vorteil, dass die anderen Services unabhängig der Eigenschaften der bestehenden Software, wie zum Beispiel deren Implementierungssprache oder erwarteter Datenformate, sind. Bestehende Software hat sich unter Umständen über Jahre bewährt und war oft teuer in der Implementierung. Der Ansatz der Kapselung in einem Microservice hat gegenüber einer Neuimplementierung den Vorteil, Kosten zu sparen und die Qualitätseigenschaften der bewährten Software, wie zum Beispiel Korrektheit oder Sicherheit, zu bewahren.

## 1.1 Zielsetzung

Im Rahmen dieser Masterarbeit soll ein Weg aufgezeigt werden, ein altes Programm in einer modernen Software-Infrastruktur weiter zu benutzen. Es soll ein Weg gezeigt werden, wie alter Prolog-Code in anderen, heutzutage üblicheren Programmiersprachen, wieder zugänglich gemacht werden kann. Dazu ist es sinnvoll, das alte Programm zu isolieren und nur in einem Teil der neuen Software direkt zu benutzen. Dieser Teil bietet dann die Funktionalität der alten Software nach außen hin an. Das Architekturmuster Microservices bietet sich aufgrund seiner oben beschriebenen Eigenschaften besonders gut für diese Anforderungen an.

Um die allgemeine Möglichkeit aufzuzeigen, alte Programme und insbesondere alte Prolog-Module in neuen Software-Infrastrukturen zu integrieren, soll im Rahmen dieser Arbeit ein bestehendes Prolog-Programm in eine vorgeschlagene Microservice-Architektur integriert werden. Bei dem bestehenden Programm handelt es sich um PASIPP, ein Programm zur Analyse und Simulation von Petri-Netzen. PASIPP ist in Prolog geschrieben und die vorliegende Version 2.3 wurde 1991 an der Universität Mannheim und der Universität Karlsruhe entwickelt.

Bei der vorgeschlagenen Microservice-Architektur handelt es sich um eine Architektur für eine Software zum Bearbeiten von Geschäftsprozessmodellen. Um die Funktionalitäten von PASIPP in dem Editor verfügbar machen zu können, soll ein neuer Microservice erstellt werden. Dieser soll nach außen hin feingranulare Interfaces anbieten, die den einzelnen Funktionalitäten von PASIPP entsprechen. Intern soll das bestehende Programm PASIPP benutzt werden, was das Ausführen des Programms, die Transformation von Eingabe- und Ausgabedaten und die Kommunikation mit dem Programm nötig macht.

## 1.2 Aufbau der Arbeit

Die vorliegende Arbeit ist wie folgt aufgebaut:

Kapitel 2 behandelt die Grundlagen, auf die diese Arbeit aufbaut und die daher für das Verständnis der Arbeit nötig sind. Es behandelt die Modellierung von Geschäftsprozessen, die Petri Net Markup Language, die Programmiersprache Prolog, das Programm PASIPP, das Microservice-Architekturmuster, eine vorgeschlagene Microservice-Architektur zur Modellierung von Geschäftsprozessen und den Architekturstil REST.

Das Konzept, nach dem der Microservice erstellt wurde, wird in Kapitel 3 vorgestellt. Der Abschnitt beginnt mit einem Überblick über die Architektur des Systems, und erklärt daraufhin die Benutzung von Prolog und PASIPP und die Ein- und Ausgabeformate des Microservices. Danach wird darauf eingegangen, wie der Service benutzt werden kann, und wie mit Fehlern umgegangen wird.

Besonderheiten, die sich während der Implementierung des Services ergeben haben, werden in Kapitel 4 kurz vorgestellt. Hier werden auch Änderungen vorgestellt, die an dem ursprünglichen Code des Programmes PASIPP vorgenommen wurden. Außerdem wird darauf eingegangen, wie sich der Microservice um weitere Funktionalitäten erweitern lässt.

Kapitel 5 geht auf die Evaluierung des Microservices und die Integration mit weiteren Services ein. Dazu werden die einzelnen Funktionalitäten des Microservices mit einem Geschäftsprozessmodell getestet, das mit dem HORUS Business Modeler<sup>1</sup> erstellt wurde. Um dies zu ermöglichen, wird ein externer Microservice verwendet, der HORUS-PNML-Dateien in standardkonforme PNML-Dateien umwandelt.

Kapitel 6 fasst die Arbeit zusammen und gibt einen Ausblick über mögliche zukünftige, darauf aufbauende, Arbeiten.

---

<sup>1</sup><https://www.horus.biz/de/downloads/>



## 2 Grundlagen

Dieses Kapitel erklärt Konzepte und Begriffe, die für das Verständnis der Arbeit notwendig sind. Abschnitt 2.1 stellt Geschäftsprozessmodellierung vor. Abschnitt 2.2 führt die Petri Net Markup Language (PNML) ein. In Abschnitt 2.3 wird eine Übersicht über die Programmiersprache Prolog gegeben. Abschnitt 2.4 stellt PASIPP vor, ein Prolog-Programm, welches in dieser Arbeit verwendet wird. In Abschnitt 2.5 wird der Architekturstil Microservices erklärt. Abschnitt 2.6 stellt eine vorgeschlagene Microservice-Architektur vor, auf die in dieser Arbeit aufgebaut wird. Abschnitt 2.7 erklärt kurz das Architekturmuster Representational State Transfer (REST).

### 2.1 Geschäftsprozessmodellierung

Ein Geschäftsprozess (Business Process) ist eine definierte Tätigkeit innerhalb eines Unternehmens, deren Ausführung ein geschäftliches oder betriebliches Ziel verfolgt [20]. Ein Geschäftsprozess besteht aus Teilprozessen (Einzeltätigkeiten oder wiederum eigene Geschäftsprozesse), deren Ausführung durch Geschäftsregeln bestimmt werden [20]. Die Regeln bestimmen zum Beispiel die Reihenfolge (Ablauflogik) der einzelnen Teilprozesse. Ein Geschäftsprozess wird durch ein bestimmtes Ereignis ausgelöst, und transformiert Einsatzgüter (Input) unter Einsatz materieller und immaterieller Güter zu Arbeitsergebnissen (Output) [20]. Diese Transformation wird beeinflusst durch unternehmensinterne Faktoren, die vom Unternehmen beeinflusst werden können, und unternehmensexterne Faktoren, auf die das Unternehmen keinen Einfluss hat [20]. Ein Beispiel für unternehmensinterne Faktoren ist der Personalbestand eines Unternehmens; ein Beispiel für einen externen Faktor sind gesetzliche Bestimmungen. Input und Output können sowohl materielle als auch immaterielle Güter sein [20].

Die Modellierung von Geschäftsprozessen (Business Process Modeling (BPM)) dient unter anderem zur Definition neuer oder Verbesserung bestehender Geschäftsprozesse und zu deren Dokumentation und Verständnis [12]. Geschäftsprozessmodelle werden dabei oft für unterschiedliche Arten von Stakeholdern entworfen [12]. Viele Ansätze zur Geschäftsprozessmodellierung haben eine graphische Repräsentation [12]. Dies erhöht die Verständlichkeit bei den unterschiedlichen Stakeholdergruppen.

Will man die Eigenschaften eines Geschäftsprozessmodells systemgestützt analysieren, ist eine formale Semantik notwendig [12]. Während eine Syntax angibt, wie eine Sprache aufgebaut ist, gibt eine Semantik an, was sie bedeutet. Für natürliche Sprachen sagt eine Syntax beispielsweise aus, wie man Sätze aufbauen kann (Vokabular und Grammatik), während die Semantik die Bedeutung wiedergibt. Eine formale Semantik ist eine formale Beschreibungssprache für Semantiken von Sprachen [14]. Petri-Netze bieten sowohl eine

graphische Syntax als auch eine formale Semantik. Petri-Netze werden von den meisten Modellierungsansätzen verwendet, die über eine formale Semantik verfügen [12].

## 2.2 Petri Net Markup Language

Die Petri Net Markup Language (PNML) ist eine Sprache zur Definition von Petri-Netzen. Die Ziele hinter der Entwicklung von PNML waren *Universalität*, *Lesbarkeit* für Menschen und *Mutualität* [28]. Mit *Universalität* ist gemeint, dass PNML als universales Austauschformat zwischen beliebigen Programmen zur Modellierung von Petri-Netzen fungieren können soll. Da für unterschiedliche Petri-Netztypen oft sehr unterschiedliche Zusatzinformationen gespeichert werden müssen (beispielsweise Datenverlust in Netzwerken [10] oder unterschiedliche Ausführungsszenarien [24]), wurde eine Möglichkeit zur Erweiterung von PNML-Netzen um programmspezifische Informationen eingeführt. PNML basiert auf XML, das einerseits weit verbreitet ist und andererseits von Menschen gelesen und verstanden werden kann (*Lesbarkeit*). Als *Mutualität* bezeichnen die Entwickler von PNML die Fähigkeit, möglichst viele Informationen über ein Petri-Netz extrahieren zu können - selbst wenn man den eigentlichen Petri-Netz-Typ nicht kennt. Um dies zu erreichen, wurde versucht, die gemeinsamen Prinzipien und Notationen von Petri-Netzen in PNML zu extrahieren.

## 2.3 Prolog

An dieser Stelle erfolgt nur ein kleiner Überblick über jene Aspekte der Programmiersprache Prolog, die für das Verständnis der Arbeit wichtig sind. Mehr Details über Prolog und die Konzepte hinter der Programmiersprache gibt zum Beispiel [3].

Prolog, abgeleitet vom französischen ***P**rogrammation en **L**ogique* [6] (deutsch: “Programmieren in Logik”), ist eine logische Programmiersprache, die einem deklarativen Programmierstil folgt. Dies bedeutet, dass Prolog-Programmcode nichts darüber aussagt, *wie* ein Problem zu lösen ist (wie bei der prozeduralen Programmierung), sondern lediglich *was* man über das Problem weiß.

Prolog enthält drei grundsätzliche Konstrukte: Fakten, Regeln und Abfragen. Fakten bilden einfache Tatsachen ab, während Regeln Beziehungen zwischen Fakten aufstellen. Eine Sammlung von Fakten und Regeln wird “Wissensbasis” genannt. Ein Prolog-Programm besteht lediglich aus Fakten und Regeln und entspricht somit einer Wissensbasis. Abfragen werden vom Benutzer an das Prolog-System gestellt, welches deren Erfüllbarkeit anhand der gegebenen Wissensbasis überprüft und als Ergebnis ausgibt. Prolog folgt dabei einer Closed-World-Annahme [27]. Dies bedeutet, dass alles, was vom Programmierer



nicht explizit modelliert wurde, als nicht-existent beziehungsweise als logisch inkorrekt angesehen wird.

Der Rest dieses Abschnitts ist wie folgt aufgebaut: Unterabschnitt 2.3.1 erklärt Prolog-Terme, Unterabschnitt 2.3.2, erklärt die Definition von Fakten und Regeln in Prolog und Unterabschnitt 2.3.3 erklärt, wie Abfragen an ein Prologsystem gestellt werden können. Unterabschnitt 2.3.4 geht auf ausführbare Prolog-Programme ein, und Unterabschnitt 2.3.5 auf Arity/Prolog, eine Implementierung der Programmiersprache Prolog.

### 2.3.1 Terme

Fakten, Regeln und Abfragen bestehen aus *Termen*. Ein Term ist entweder ein *Atom*, eine *Zahl*, eine *Variable* oder ein *komplexer Term*. Atome und Zahlen werden als *Konstanten* bezeichnet. Konstanten und Variablen bilden die sogenannten *einfachen Terme*. Die Beziehung der Termtypen zueinander ist in Abbildung 1 dargestellt.

#### Atome

Prolog-Atome stellen Objekte der modellierten Welt dar, wie zum Beispiel *james*, *lilly*, *harry*, oder Beziehungen zwischen solchen Objekten, zum Beispiel *liebt*, *vater*. Einfache Atome bestehen aus Buchstaben, Zahlen und Unterstrichen, und beginnen mit einem Kleinbuchstaben, zum Beispiel *harryWizard*. Außerdem kann jeder beliebige String zwischen zwei einzelnen Anführungszeichen als Atom dienen, zum Beispiel *'Harry Wizard'* (mit Leerzeichen) oder *'%@'* (Sonderzeichen).

#### Variablen

Variablen kommen in komplexen Termen, Fakten, Regeln und Abfragen zum Einsatz. Sie können - anders als Atome und Zahlen - unterschiedliche Werte annehmen. Mehr dazu in Unterabschnitt 2.3.3. Variablennamen bestehen in Prolog aus Buchstaben, Zahlen und Unterstrichen. Variablen starten entweder mit einem Großbuchstaben oder Unterstrich, also zum Beispiel *X*, *Y*, *Head*, *\_tail*, *X3*, *List\_4*.

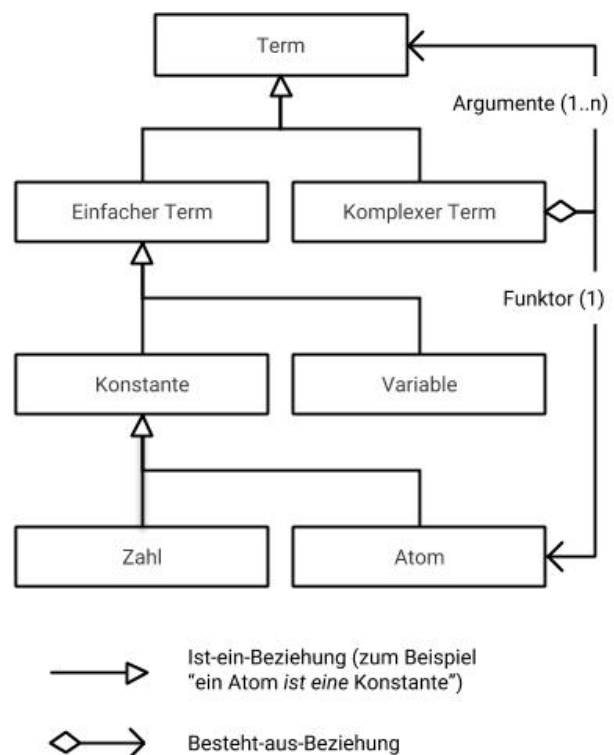


Abbildung 1: Termtypen und ihre Beziehung in Prolog

```

1  mensch(ginny). % bedeutet, dass die Aussage ``Ginny ist ein Mensch'' wahr
    ↪ ist.
2  mag(ginny, pizza). % bedeutet, dass die Aussage ``Ginny mag Pizza'' wahr
    ↪ ist, dass also Ginny Pizza mag.

```

#### Codeausschnitt 1: Beispiel-Fakten

```

1  mag(X, pizza). % Jeder mag Pizza.
2  mag(X, X). % Jeder mag sich selbst.
3  mag(X,Y). % Jeder mag alle(s).

```

#### Codeausschnitt 2: Variablen in Fakten

### Komplexe Terme

Komplexe Terme sagen etwas über die Eigenschaften oder Beziehungen anderer Terme aus. Ein komplexer Term besteht aus einem *Funktor* und einem oder mehreren *Argumenten*. Ein Funktor ist ein Atom, das eine Eigenschaft oder Beziehung der Argumente angibt, zum Beispiel *mag* oder *mensch*. Ein Argument kann ein beliebiger Term sein, sowohl einfach als auch komplex. Beispiele für komplexe Terme sind *mag(ginny,pizza)* oder *mensch(ginny)*.

### 2.3.2 Fakten und Regeln

Wie schon erwähnt, bestehen Prologprogramme aus Wissensbasen, die eine Ansammlung von Fakten und Regeln sind. Diese werden auf Anforderung des Nutzers vom Prolog-System überprüft. Im Folgenden werden Fakten und Regeln genauer erklärt.

#### Fakten

Ein Fakt besteht aus einem Prolog-Term, gefolgt von einem Punkt. Ein Fakt kommt innerhalb eines Prolog-Programms vor und sagt aus, dass der entsprechende Term wahr ist, wie in Codeausschnitt 1 erklärt.

Variablen innerhalb eines Fakts sind implizit universell instanziiert. Dies bedeutet, dass die Aussage für alle Objekte in der Wissensbasis als wahr gilt, siehe Codeausschnitt 2.

#### Regeln (Implikationen)

Regeln verknüpfen mehrere Terme miteinander über logische Implikationen. Eine Regel besteht aus dem Zeichen  $:-$ , einem Term links davon und mindestens einem Term rechts davon, getrennt durch Kommata. Das  $:-$  ist dabei eine logische Implikation von rechts

```
1 parent(A,B) :- father(A,B).
2 parent(A,B) :- mother(A,B).
3 grandparent(A,B) :- parent(A,C), parent(C,B).
```

Codeausschnitt 3: Regeln zur Definition von Eltern- und Großelternbeziehung

```
1 ancestor(A,B) :- parent(A,B).
2 ancestor(A,B) :- ancestor(A,C), ancestor(C,B).
```

Codeausschnitt 4: Rekursive Definition der Vorfahren-Regel

nach links ( $\leftarrow$ ), das bedeutet: Wenn alle rechten Terme *wahr* sind, dann ist auch der linke Term *wahr*.

Gibt es mehrere Regeln mit dem gleichen Regelkopf (der Term links des  $:-$ ), sind diese disjunktiv miteinander verknüpft. Das heißt, der linke Term ist *wahr* sobald eine der Regeln zu *wahr* ausgewertet wird. Ein Beispiel dafür ist in Codeausschnitt 3 zu sehen: die Regel  $parent(A,B)$  trifft zu, wenn eine ihrer beiden Definitionen - also entweder  $father(A,B)$  oder  $mother(A,B)$  - wahr ist. Außerdem ist  $A$  *grandparent* (Großelternteil) von  $B$ , falls ein  $C$  existiert, so dass gilt:  $A$  ist Elternteil von  $C$ , und  $C$  ist Elternteil von  $B$ . Dabei handelt es sich um eine konjunktive Verknüpfung der beiden Teilziele  $parent(A,C)$  und  $parent(C,B)$ .

Die einzige Möglichkeit, Schleifen in Prolog zu implementieren, ist über Rekursion. Codeausschnitt 4 enthält eine rekursive Definition des Prädikats “Vorfahre” (*ancestor*): Eine Person  $A$  ist Vorfahre von  $B$ , wenn sie entweder Elternteil von  $B$  ist, oder Vorfahre eines Vorfahren  $C$  von  $B$ .

### 2.3.3 Abfragen, Unifikation und Backtracking

Um Prolog zu nutzen, startet man die Prolog-Installation üblicherweise in einer Kommandozeile, lädt Programmdateien und stellt danach Abfragen an das Prolog-System. Das Prolog-System versucht dann anhand der geladenen Wissensbasen (also den Programmen), die Aussage der Abfrage zu beweisen.

Für Abfragen, die nur Atome enthalten, gibt das Prolog-System entweder *true* oder *false* zurück, je nachdem, ob sich die enthaltene Aussage aus den verfügbaren Wissensbasen ableiten lässt oder nicht. Dabei wird zuerst geprüft, ob das entsprechende Prädikat existiert. Dann werden die in der Abfrage vorkommenden Atome nacheinander mit den Regeln in der Wissensbasis verglichen. Ist beispielsweise die Wissensbasis aus Codeausschnitt 5 geladen, und man stellt die Anfrage `?- father(harry, lillyLuna).`, prüft das Sys-

```

1 father(fleamond, james).
2 father(james, harry).
3 father(harry, lillyLuna).
4 father(harry, albusSeverus).
5 father(harry, jamesSirius).
6
7 mother(euphemia, james).
8 mother(lilly, harry).
9 mother(ginny, lillyLuna).
10 mother(ginny, albusSeverus).
11 mother(ginny, jamesSirius).

```

### Codeausschnitt 5: Definition von Eltern-Beziehungen

tem zuerst ob ein Prädikat namens *father* existiert. Darauf vergleicht es nacheinander die Atome *harry* und *lillyLuna* mit den Atomen, die in den definierten *father*-Fakten vorkommen. Dabei vergleicht es (*harry, lillyLuna*) zuerst mit (*fleamond, james*), dann mit (*james, harry*) und dann mit (*harry, lillyLuna*). Dieser Fakt erfüllt die Abfrage, und das System gibt *true* zurück. Bei einer Abfrage, die nicht zutrifft, zum Beispiel *?- father(james, albusSeverus).*, geht das Prolog-System alle *father*-Fakten durch und gibt am Ende *false* zurück.

Enthält eine Abfrage eine Variable, gibt Prolog nicht nur aus, ob die Abfrage erfüllbar ist, sondern sucht auch nach Variablenbelegungen, die die Abfrage erfüllen, und gibt diese aus. Dieser Vorgang wird *Unifikation* genannt. Stellt man beispielsweise die Abfrage *father(harry, X)* (mit den Wissensbasen aus Codeausschnitt 3 und 5), sucht Prolog nach einem Atom, mit dem *X* ersetzt werden kann, sodass die Abfrage erfüllt ist. Der gleiche Mechanismus wird auch verwendet, wenn für das Finden einer Antwort nicht nur Fakten, sondern auch Regeln analysiert werden müssen. Stellt man beispielsweise die Abfrage *?- grandparent(james, X).*, so wird die enthaltene Regel zuerst mit *parent(james, C)*, *parent(C, X)* unifiziert. Daraufhin wird ein Term *\_C1* gesucht der *parent(james, \_C1)* erfüllt, und *C* wird damit unifiziert. Daraufhin wird wiederum ein Term *\_X1* gesucht, der *parent(\_C1, \_X1)* erfüllt, und *X* wird damit unifiziert.

Prolog führt dabei eine Tiefensuche aus, das heißt es sucht innerhalb der Wissensbasen immer von oben nach unten, und innerhalb eines Terms von links nach rechts. Das Prolog-System merkt sich dabei jeden Entscheidungspunkt, also jeden Punkt, an dem zwei Terme miteinander unifiziert wurden. Falls das Prolog-System feststellt, dass die Abfrage mit der aktuellen Variablenbelegung nicht erfüllbar ist, geht es zum letzten Entscheidungspunkt zurück, macht die Variablenbelegung rückgängig und wählt die nächste mögliche Unifi-

kation. Dieser Vorgang heißt *Backtracking*. Der gleiche Mechanismus wird angewendet, wenn eine gültige Variablenbelegung an den Nutzer zurückgegeben wurde und der Nutzer eine weitere Erfüllung der Abfrage anfordert. Ist die gesamte Wissensbasis durchsucht (das heißt, es gibt keine (weitere) Lösung), gibt Prolog *false* zurück.

### 2.3.4 Ausführbare und interaktive Programme

Das erste Prolog-System wurde 1972 implementiert [6]. Da in der Softwaretechnik erst zu dieser Zeit das Konzept der Modularität aufkam [13], enthielt Prolog zuerst kein Konzept für Modularität. Unterstützung von Modularität wurde später in unterschiedlichen Prolog-Systemen eingeführt, unter anderem in Arity/Prolog (siehe Unterabschnitt 2.3.5).

Ein Programm kann dabei in mehrere Dateien unterteilt werden, wobei in jeder Datei definiert wird, welche Regeln von außen benutzt werden und welche nach außen hin angeboten werden. Ein Beispiel dafür ist in Codeausschnitt 6 zu sehen, das die Modul-Definition der Datei PASIPP.ARI, der Start-Datei des Programms PASIPP (siehe Abschnitt 2.4, wiedergibt. In Zeile 1 wird der Modulname definiert. Zeile 3 definiert die Regel *main* als öffentlich. Die restlichen Zeilen definieren welche Regeln aus anderen Dateien benötigt werden. Die Zahlen hinter einem / geben dabei die Stelligkeit der Regel, also die Anzahl ihrer Argumente, an.

Dadurch, dass nur bestimmte Regeln als öffentlich deklariert werden, wird eine Trennung erreicht zwischen Regeln, die tatsächlich von außen benutzt werden sollen und Regeln, die nur Implementationsdetails beinhalten. Damit ist es möglich, in Prolog Interfaces zu definieren.

Eine Besonderheit ist die *main/0*-Regel. Hat man eine *main/0*-Methode definiert, kann man mithilfe des Prolog Compilers seinen Prolog-Code in ein ausführbares Programm konvertieren. Führt man das Programm aus, wird die *main*-Regel ausgewertet, und danach endet das Programm wieder.

Prolog verfügt über mehrere Standard-Regeln, um innerhalb einer eigenen Regel mit dem Benutzer zu interagieren. Die Methode *write* ermöglicht es, Text an den Benutzer oder in eine Datei auszugeben, während *cls* (kurz für "clear screen") die Bildschirmanzeige zurücksetzt. Mit diesen beiden Methoden ist es möglich, ein textuelles User Interface aufzubauen. Die Regeln *readString* und *readLine* ermöglichen es, sowohl Dateien als auch vom User eingegebenen Text zu lesen. Es lassen sich also über die *main*-Regel und über Eingabe-/Ausgabe-Regeln Programme aufbauen, die sowohl User- als auch Datei-Interaktion ermöglichen.

```
1  :- module pasipp.  
2  
3  :- public main/0.  
4  
5  :- extrn clear_screen/0.  
6  :- extrn clear_DB/2.  
7  :- extrn read_string/1.  
8  :- extrn exec_command/1.  
9  :- extrn ende_pasipp/0.  
10 :- extrn init/0.  
11 :- extrn dynamic_menuue/0:far.  
12 :- extrn static_menuue/1.  
13 :- extrn simulation_menuue/0.  
14 :- extrn reachtree_menuue/0:far.  
15 :- extrn option_menuue/1:far.  
16 :- extrn network_menuue/0.  
17 :- extrn weiter/0.  
18 :- extrn meldung/0.  
19 :- extrn laden/1.
```

Codeausschnitt 6: Modul-Definition von PASIPP.ARI [16]

### 2.3.5 Arity/Prolog

Arity/Prolog ist eine Implementierung der Prolog-Programmiersprache. Sie wurde von der Arity Corporation ursprünglich für 16-Bit-Systeme entwickelt. Später wurde mit Arity/Prolog32 eine Version für 32-Bit-Betriebssysteme herausgebracht.

Die Dokumentation von Arity/Prolog32 ist heutzutage zugänglich auf der Webseite von Peter Gabel [11], einem der ursprünglichen Entwickler von Arity/Prolog32. Sie ist lizenziert unter der *Creative Commons Attribution-NonCommercial-NoDerivs 3.0 United States* Lizenz<sup>2</sup> und darf somit kopiert und weiterverbreitet werden unter der Bedingung der Namensnennung von Autoren und Lizenz, der unkommerziellen Nutzung und keiner Bearbeitung des Materials. Die Dokumentation wurde von der Webseite kopiert und im digitalen Anhang dieser Arbeit [23] verfügbar gemacht.

Der Interpreter und Compiler für Arity/Prolog32 für Windows sind ebenso verfügbar auf der Webseite von Peter Gabel [11] unter der MIT-Lizenz<sup>3</sup>. Sie erlaubt die uneingeschränkte Nutzung und Weiterverbreitung der Software unter der Bedingung, bei der Weiterverbreitung eine Kopie der originalen Lizenz mitzuverbreiten. Der Interpreter und Compiler sind ebenfalls im digitalen Anhang dieser Arbeit verfügbar.

## 2.4 PASIPP

PASIPP ist ein Prolog-Programm zur Analyse und Simulation von Petri-Netzen [16]. Das Programm wird in der vorliegenden Arbeit innerhalb des Microservices genutzt, um nach außen hin Analyse und Simulation von Petri-Netzen zu ermöglichen. Prolog wurde in Arity/Prolog geschrieben und für PCs mit DOS-Betriebssystem entwickelt. Das für diese Arbeit vorliegende Programm hat die Version 2.3 und stammt aus dem April 1991. Es liegt sowohl als exe-Datei als auch in Form von Arity/Prolog-Quelldateien vor. Die exe-Datei lässt sich in einer virtuellen DOS-Umgebung wie zum Beispiel vDos<sup>4</sup> ausführen. Die Quelldateien lassen sich mit dem Arity/Prolog32-Interpreter (siehe Unterabschnitt 2.3.5) laden und nutzen.

PASIPP ermöglicht sowohl die Analyse als auch die Simulation von Petri-Netzen. Bei der Simulation wählt der Benutzer selbst welche Transitionen schalten sollen, während bei einer Analyse das Programm versucht, eine Lösung für ein gegebenes Problem zu finden [16]. Bei der Analyse kann es sich beispielsweise um eine Erreichbarkeitsanfrage handeln, die das Programm zu lösen versucht. Dabei wird vom Programm eine Folge von Transitionsschaltungen gesucht, die die Anfrage erfüllt. Dazu ist es notwendig, bereits geschaltete,

---

<sup>2</sup><https://creativecommons.org/licenses/by-nc-nd/3.0/us/>

<sup>3</sup><https://opensource.org/licenses/mit-license.php>

<sup>4</sup><https://vdos.info/>

nicht zum Ziel führende, Transitionen zurücksetzen zu können. Dieser Mechanismus wurde über Prolog-Backtracking (siehe Unterabschnitt 2.3.3) realisiert [15]. Prolog ist aufgrund der Unifikation und des Backtrackings gut zur Simulation von Pr/T-Netzen geeignet [26].

Im Folgenden wird in Unterabschnitt 2.4.1 zuerst auf die Beschreibung von Petri-Netzen in PASIPP eingegangen, und daraufhin in Unterabschnitt 2.4.2 auf die Beschreibung von Erreichbarkeitsbäumen. Unterabschnitt 2.4.3 beschreibt die Funktionen und die Nutzung des Programms. Eine detailliertere Erklärung findet man in [16].

### 2.4.1 Beschreibung von Petri-Netzen

Zur Benutzung von PASIPP zur Analyse oder Simulation von Petri-Netzen ist es zuerst nötig, die Netze in einer definierten Notation in Prolog zu speichern. Während der Ausführung von PASIPP kann man die Dateien laden, Simulationen und Analysen durchführen und bei Bedarf Ergebnisse wieder in der gleichen Notation speichern. Dabei werden für das Netz und für die Markierung des Netzes jeweils eigene Dateien benötigt.

Eine **Markierung** des Netzes ist definiert als die Zuweisung von Marken (engl. Tokens) an die Stellen (Places) des Netzes. Jede dieser Zuweisungen entspricht einem Prolog-Fakt. Bei dem Fakt handelt es sich um einen komplexen Term mit dem Funktor *marke* und zwei Argumenten: die Anzahl der Marken und der Name der Stelle, in der sie sich befinden. Definiert man Markierungen auf diese Weise, sind die einzelnen Marken *anonym*. Man kann einzelne Marken nicht voneinander unterscheiden. Neben dieser Art der Definition ist es auch möglich, *Strukturen* zu verwenden. Mithilfe von Strukturen ist es möglich, Eigenschaften von einzelnen Marken zu definieren. Dafür gibt man in der Definition der Stellen eine Struktur an, die dann von den Marken instanziiert wird. Codeausschnitt 7 zeigt in Zeile 1 und 2 den allgemeinen Aufbau von anonymen Markierungen und Markierungen mit Struktur. Zeile 4 zeigt eine beispielhafte anonyme Markierung aus dem Kontext eines Autoverleihs. Zeile 6 bis 8 zeigen entsprechende Markierungen mit Struktur. Die Stelle *verfuegbaresAuto* hat dabei die Struktur *verfuegbaresAuto(Typ,Baujahr,Kennzeichen)*.. Da diese Arbeit lediglich Marken ohne Struktur benutzt, wird darauf allerdings nicht weiter eingegangen.

Neben der initialen Markierung des Netzes ist es auch nötig, das Netz selbst zu definieren. Die **Netz**-Datei besteht aus *fire*-, *kapazitaet*- und *fact*-Regeln. Jede Regel wird jeweils als ein Prolog-Regel ausgedrückt (siehe Unterabschnitt 2.3.2). An dieser Stelle wird nur auf die *fire*-Regeln eingegangen, da die anderen beiden Regeltypen in dieser Arbeit nicht verwendet werden. Eine *fire*-Regel definiert den Schaltvorgang einer Transition. Für jede Transition existiert eine Regel, die wie in Codeausschnitt 8 zu sehen aufgebaut ist.

Die Regel besteht aus mehreren Blöcken:



```
1 fire([transitionsname, Inputvariablen, Outputvariablen]) :-  
2   entferne(anzahl [1], stellename [1] (marke [1])),  
3   ...  
4   entferne(anzahl [i], stellename [i] (marke [i])),  
5   transitionsbeschriftung,  
6   einfuege(anzahl [i+1], stellename [i+1] (marke [i+1])),  
7   ...  
8   einfuege(anzahl [n], stellename [n] (marke [n])),
```

Codeausschnitt 7: Aufbau einer *marke*-Regel (vgl. [16])

```
1 fire([transitionsname, Inputvariablen, Outputvariablen]) :-  
2   entferne(anzahl [1], stellename [1] (marke [1])),  
3   ...  
4   entferne(anzahl [i], stellename [i] (marke [i])),  
5   transitionsbeschriftung,  
6   einfuege(anzahl [i+1], stellename [i+1] (marke [i+1])),  
7   ...  
8   einfuege(anzahl [n], stellename [n] (marke [n])),
```

Codeausschnitt 8: Aufbau einer *fire*-Regel (vgl. [16])

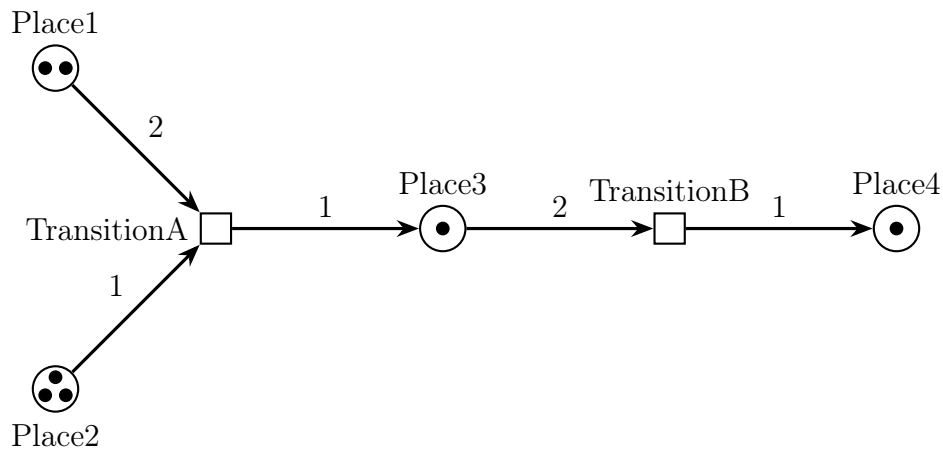


Abbildung 2: Beispiel-Petri-Netz

- *transitionsname*: gibt die ID der Transition an
- *Inputvariablen*, *Outputvariablen*: dienen zum Arbeiten mit der Struktur von Marken (in dieser Arbeit nicht benutzt)
- *entferne*-Block: jeder *entferne*-Fakt gibt für eine eingehende Stelle an, wie viele Tokens aus ihr entfernt werden sollen (ggf. mit der Struktur der Stellen)
- *transitionsbeschriftung*: Prolog-Ausdruck, der aus den Marken der Input-Stellen bestimmte Exemplare auswählt, und Exemplare für die Output-Stellen definiert.
- *ein fuege*-Block: jeder *ein fuege*-Fakt gibt für eine ausgehende Stelle an, wie viele Tokens zu ihr hinzugefügt werden sollen (ggf. mit der Struktur der Stellen)

Abbildung 2 zeigt ein beispielhaftes, einfaches Petri-Netz. Codeausschnitt 9 stellt das Netz in PNML dar. Jede Stelle und Transition hat dabei einen *Place*- beziehungsweise *Transition*-Knoten in der XML-Struktur. Markierungen werden mit *initialMarking* angegeben. Die Kanten des Netzes werden separat als *arc* mit ihren jeweiligen Start- und Endknoten angegeben.

Codeausschnitt 10 und 11 geben die Netz-Struktur beziehungsweise die Markierung des Netzes in Prolog-Notation wieder. Für jeden *Place*, der über ein *initialMarking* verfügt, existiert dabei eine *marke*-Regel, die diese Markierung wiedergibt. Außerdem existiert für jede Transition eine *fire*-Regel. Für jede eingehende Stelle gibt dabei eine *entferne*-Regel die Anzahl der zu entfernenden Input-Tokens an, während für jede ausgehende Stelle eine *ein fuege*-Regel die Anzahl der zu erzeugenden Output-Tokens angibt.

Um aus einer PNML-Transition eine *fire*-Regel zu erzeugen, erstellt man zuerst den Regelkopf, also `fire([Transitionsname]) :-`. Dann fügt man für jeden Kante (*arc*), die die Transition als *source* enthält, eine *entferne*-Regel mit Wert 1 hinzu, beziehungsweise

```

1 <pnml xmlns="http://www.pnml.org/version-2009/grammar/pnml">
2 <net id="net0" type="http://www.pnml.org/version-2009/grammar/ptnet">
3   <page id="page0">
4     <place id="place1">
5       <initialMarking><text>2</text></initialMarking>
6     </place>
7     <place id="place2">
8       <initialMarking><text>3</text></initialMarking>
9     </place>
10    <place id="place3">
11      <initialMarking><text>1</text></initialMarking>
12    </place>
13    <place id="place4"/>
14    <transition id="transitionA"/>
15    <transition id="transitionB"/>
16    <arc id="arc1" source="place1" target="transitionA"/>
17    <arc id="arc2" source="place1" target="transitionA"/>
18    <arc id="arc3" source="place2" target="transitionA"/>
19    <arc id="arc4" source="transitionA" target="place3"/>
20    <arc id="arc5" source="place3" target="transitionB"/>
21    <arc id="arc6" source="place3" target="transitionB"/>
22    <arc id="arc7" source="transitionB" target="place4"/>
23  </page>
24 </net>
25 </pnml>

```

Codeausschnitt 9: Petri-Netz in PNML

```

1 fire(['transitionB']) :-
2     entferne(2, 'place3'),
3     einfuege(1, 'place4').
4 fire(['transitionA']) :-
5     entferne(2, 'place1'),
6     entferne(1, 'place2'),
7     einfuege(1, 'place3').

```

Codeausschnitt 10: Struktur des Netzes in Prolog

```

1  marke(1, 'place3').
2  marke(2, 'place1').
3  marke(3, 'place2').

```

### Codeausschnitt 11: Markierung des Netzes in Prolog

erhöht den Wert der Regel um 1, wenn sie schon existiert. Dies macht man analog mit einfüge-Regeln, falls die Transition als *target* vorkommt.

Dieser Algorithmus ist in Pseudocode in Codeausschnitt 12 dargestellt. In dem Algorithmus beginnt jeder Klassenname entweder mit *pnml* oder mit *prolog*. Dies soll deutlich machen, in welchem Kontext die entsprechende Entität steht.

## 2.4.2 Darstellung von Erreichbarkeitsbäumen

Für die vorliegende Arbeit wurde unter anderem das Erzeugen eines Erreichbarkeitsbaums umgesetzt (siehe Unterabschnitt 2.4.3 und 3.4.5). Die Erreichbarkeitsbaumgenerierung von PASIPP gibt die Bäume wie in Codeausschnitt 13 zu sehen aus. Jede Zeile entspricht dabei einer Kante des Baumes. Die ersten äußeren eckigen Klammern  $/$   $/$  enthalten die Markierung des Elternknotens, die zweiten äußeren eckigen Klammern die Markierung eines Kindknotens. Dies bedeutet, dass man von dem Elternknoten mit dem Schalten einer Transition zum Kindknoten gelangt. Eine Netzmarkierung (die äußeren Klammern) besteht dabei aus der Menge der Markierungen der einzelnen Stellen. Diese sind in den inneren Klammern angegeben.  $/ (2, place2) /$  bedeutet dabei beispielsweise, dass die Stelle *place2* in der aktuellen Markierungen 2 Marken (Token) enthält. Sie entspricht also der in Unterabschnitt 2.4.1 vorgestellten Notation `marke(2, 'place2')`..

Eine leere innere eckige Klammer  $/$   $/$  bedeutet, dass die Stelle, die hier stehen würde, über keine Marken verfügt. Hat ein Elternknoten mehrere Kindknoten (das heißt, bei der entsprechenden Markierung können mehr als eine Transition schalten, deren resultierende Markierungen sich voneinander unterscheiden), werden jede Eltern-Kind-Beziehung in einer extra Zeile angegeben. Codeausschnitt 13 enthält den Erreichbarkeitsbaum für das Netz und die Startmarkierung, die in Abbildung 2 und Codeausschnitt 9 bis 11 vorgestellt wurden. Der entsprechende Erreichbarkeitsgraph ist in Abbildung 3 zu sehen.

Manche Petri-Netze haben Stellen, deren Markenanzahl unendlich groß werden kann. Dies kann zum Beispiel vorkommen, wenn ein Petri-Netz einen Kreis enthält, der mehr Marken produziert als er konsumiert, oder wenn es eine Transition enthält, die zwar Marken produziert, aber keine Marken konsumiert. PASIPP erkennt solche Situationen und gibt in einem solchen Fall als Markierung der Stelle den Wert *infini*t zurück. Abbildung 4 zeigt ein Beispiel für einen Kreis, der mehr Marken produziert als er konsumiert, und den

```

1  for all PnmlTransitions t {
2      create and store new PrologFireRule fr;
3      for all PnmlArcs a {
4          if (a.target == t) {
5              if (a.entferneRules already contains
6                  ↪ (a.source,X)) {
7                  X := X+1;
8              } else {
9                  a.entferneRules.add((a.source, 1));
10             }
11         }
12         if (a.source == t) {
13             if (a.einfuegeRules already contains
14                 ↪ (a.target,X)) {
15                 X := X+1;
16             } else {
17                 a.einfuegeRules.add((a.target, 1));
18             }
19         }
20     }
21     for all PnmlPlaces p {
22         if (p.marking > 0) {
23             create and store new PrologMarking(p.id, p.marking);
24         }
25     }

```

Codeausschnitt 12: Pseudocode-Algorithmus zur Konvertierung von PNML- in Prolog-Notation

```

1  [ [], [ (2 , place2) ], [ (2 , place3) ], [] ] , [ [], [ (2 , place2) ], [], [
    ↪ (1 , place4) ] ]
2  [ [ (2 , place1) ], [ (3 , place2) ], [ (1 , place3) ], [] ] , [ [], [ (2 ,
    ↪ place2) ], [ (2 , place3) ], [] ]

```

Codeausschnitt 13: Erreichbarkeitsbaum in Prolog-Notation

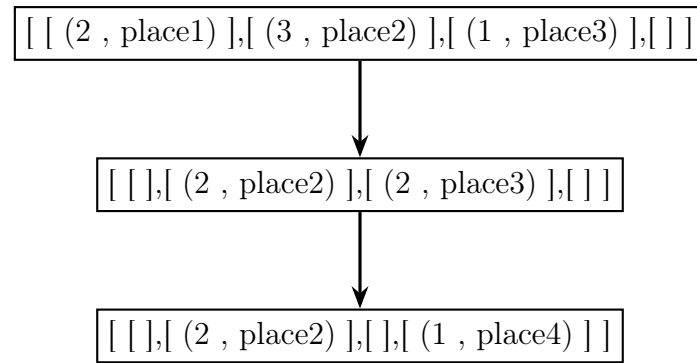


Abbildung 3: Graphische Darstellung des Erreichbarkeitsbaums aus Codeausschnitt 13

entsprechenden Erreichbarkeitsbaum. Codeausschnitt 14 zeigt die entsprechende PASIPP-Ausgabe.

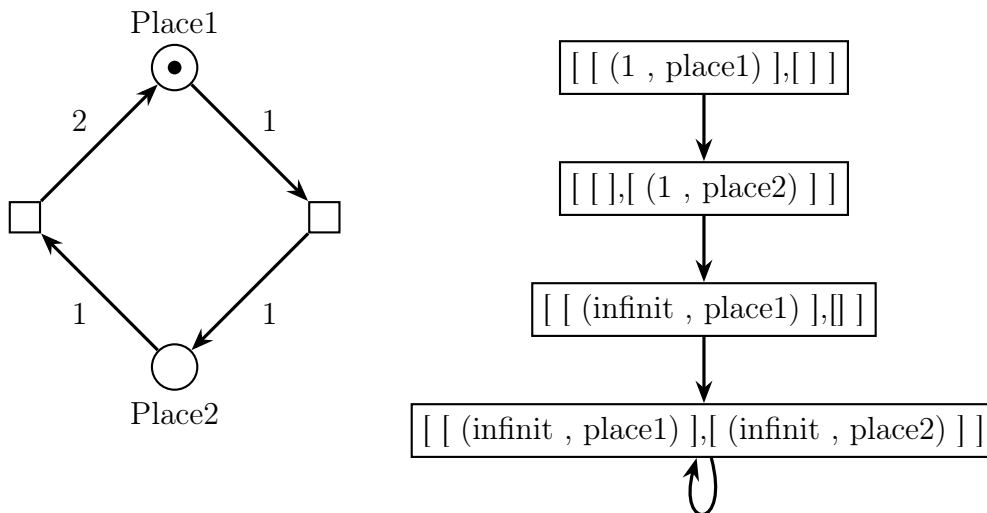


Abbildung 4: Beispiel-Petri-Netz mit potenziell unendlich vielen Markierungen (links) und der entsprechende Erreichbarkeitsbaum (rechts)

### 2.4.3 Benutzung des Programms

Beim Start von Prolog erscheint zunächst ein Startbildschirm (siehe Abbildung 5), gefolgt von der Aufforderung, den Pfad für eine Netzdatei und eine Markierungsdatei anzugeben. Dies kann man jeweils mit Drücken der *Enter*-Taste überspringen (oder alternativ die Dateien angeben, woraufhin diese geladen werden). Danach erscheint das Hauptmenü, das in Abbildung 6 zu sehen ist. Die Navigation durch das Programm erfolgt durch die Eingabe des jeweiligen Buchstabens vor einem Menüpunkt, bestätigt durch die *Enter*-Taste.

```

1  [ [ (infini t , place1) ], [ (infini t , place2) ] ] , [ [ (infini t ,
   ↪ place1) ], [ (infini t , place2) ] ]
2  [ [ (infini t , place1) ], [ (infini t , place2) ] ] , [ [ (infini t ,
   ↪ place1) ], [ (infini t , place2) ] ]
3  [ [ (infini t , place1) ], [] ] , [ [ (infini t , place1) ], [ (infini t ,
   ↪ place2) ] ]
4  [ [], [ (1 , place2) ] ] , [ [ (infini t , place1) ], [] ]
5  [ [ (1 , place1) ], [] ] , [ [], [ (1 , place2) ] ]

```

Codeausschnitt 14: PASIPP-Darstellung des Erreichbarkeitsbaums aus Abbildung 4

Unter *<e> Netz bearbeiten (einlesen, sichern oder ändern)* kann man neue Netz- und Markierungs-Dateien sowie Kapazitätsbeschränkungen laden und die aktuelle Markierung speichern. Außerdem kann man selbst definierte Prologregeln laden und somit zur Wissensbasis des Programms hinzufügen. Falls ein Netz Zyklen enthält, lädt das Programm die Datei und gibt dabei Netzdatei eine Warnmeldung zurück. Verstößt dagegen eine neue Markierung gegen bestehende Kapazitätsbedingungen, beziehungsweise machen neu geladene Kapazitätsregeln eine aktuelle Markierung ungültig, wird die entsprechende Datei nicht geladen, und es wird eine Fehlermeldung ausgegeben. In dieser Arbeit werden dabei nur Programm Meldungen berücksichtigt, die die Netzstruktur betreffen, nicht aber Meldungen, die Kapazitätsbeschränkungen betreffen. Dies hat den Grund, dass in der Arbeit PNML ohne Erweiterungen als Format für Petri-Netze verwendet wird, welches keine Unterstützung für Kapazitätsbeschränkungen bietet.

Unter *<b> Netzstrukturanfragen* kann man Informationen zum Petri-Netz erhalten, wie zum Beispiel die aktuelle Markierung, die Stellen des Netzes oder die Liste aller Transitionen. Unter *<c> Simulation* lassen sich das Schalten einer oder mehrerer Transitionen simulieren. Der Nutzer kann dabei bestimmen, welche Transitionen geschaltet werden und ob dies sequentiell oder parallel geschehen soll. Der Nutzer kann so lange Transitionen schalten, bis eine Markierung erreicht wird, in der keine schaltbare Transaktion existiert. Unter *<d> Erreichbarkeitsbaumgenerierung* lassen sich für eine gegebene Markierung Erreichbarkeitsbäume generieren. Unter dem Menüpunkt *<a>* lassen sich Erreichbarkeitsanfragen stellen, was in dieser Arbeit aus Zeitgründen allerdings nicht umgesetzt wurde.

Zur Nutzung der Funktionalitäten von PASIPP in dieser Arbeit wurde nicht das ausführbare Programm benutzt, sondern der Arity/Prolog32-Interpreter, der die PASIPP-Quelldateien liest und interpretiert. Dazu mussten kleine Änderungen an einzelnen Funktionen vorgenommen werden, unter anderem da sich die Spezifikation von Arity/Prolog und Arity/Prolog32 leicht unterscheiden. Mehr dazu in Abschnitt 4.1. Der Grund für die Nutzung der Quelldateien ist, dass PASIPP ursprünglich für eine Nutzung durch

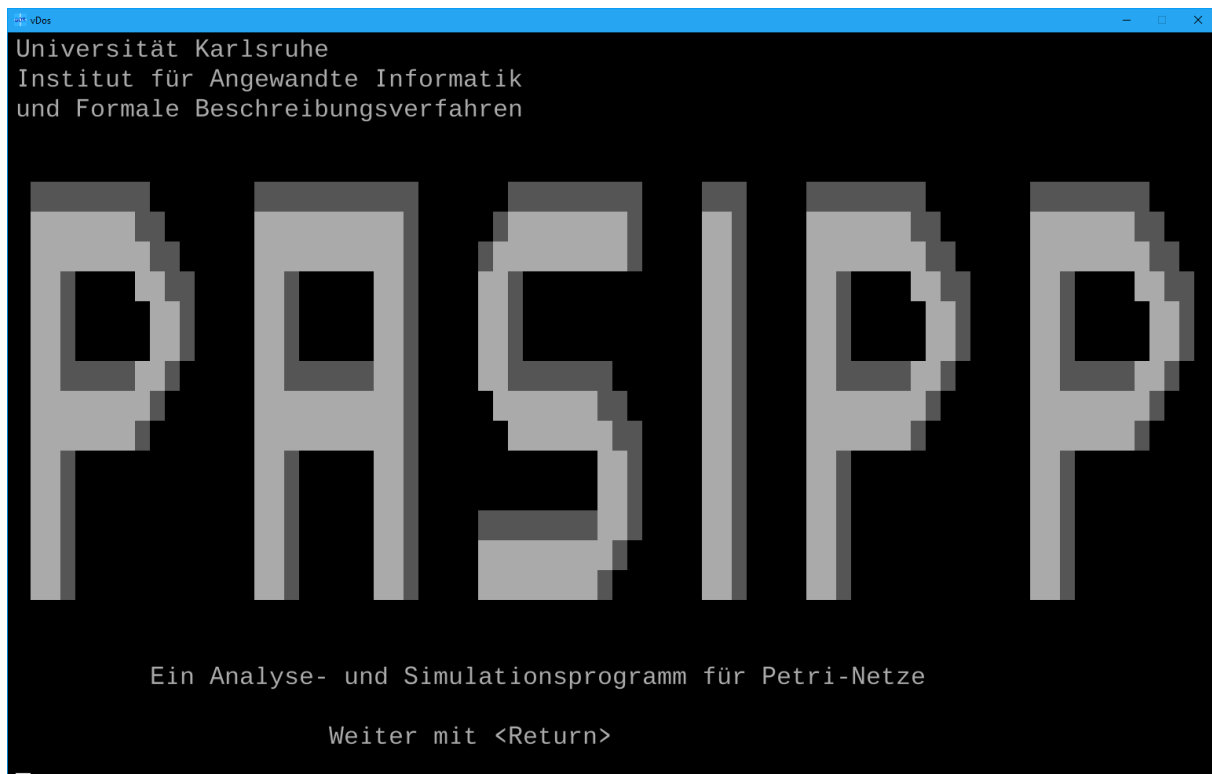


Abbildung 5: Startbildschirm von PASIPP (ausgeführt unter vDos)

Menschen, nicht durch andere Programme, geschrieben wurde. Die Navigation durch das Programm würde Overhead und dadurch zusätzliche potenzielle Fehlerquellen generieren. Außerdem ist die vorliegende ausführbare Datei nur unter DOS-Betriebssystemen oder unter virtuellen Umgebungen wie zum Beispiel vDos<sup>5</sup> lauffähig. Zur direkten Nutzung in modernen Betriebssystemen müsste sie also neu kompiliert werden, was dieselben Code-Anpassungen nötig machen würde wie die direkte Nutzung der Quelldateien. Abschnitt 3.2 gibt mehr Details zur Nutzung von Prolog und PASIPP in dieser Arbeit.

## 2.5 Microservices

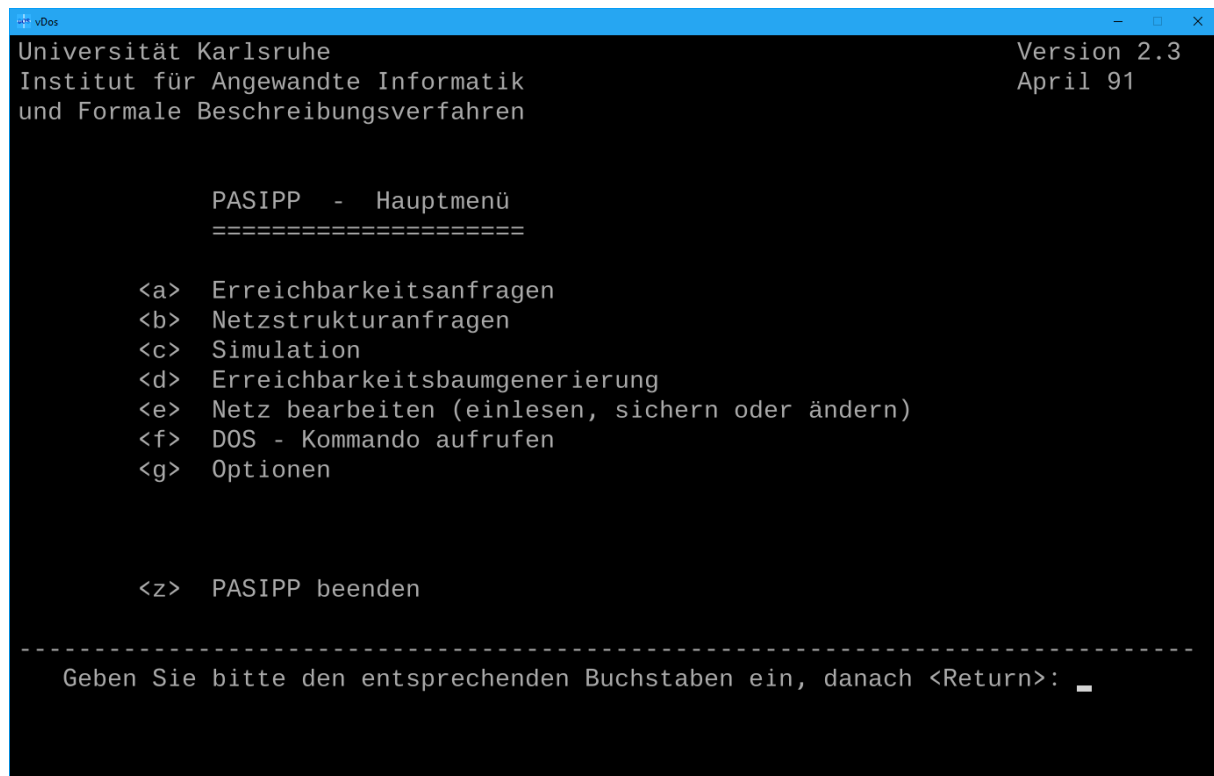
Microservices sind ein Architekturstil für verteilte Software-Systeme. Einzelne Anwendungen werden dabei als Gruppe kleiner Services entwickelt, die jeweils in einem eigenen Prozess laufen und über leichtgewichtige Mechanismen, zum Beispiel eine HTTP Resource API, miteinander kommunizieren [9].

Microservice-Architekturen werden eingesetzt um unterschiedliche Schwächen monolithischer Systeme zu umgehen. Ein Monolith ist ein Softwaresystem, das als einzelne Einheit entwickelt wird und aus einer einzelnen ausführbaren Datei besteht. Das System läuft in einem einzelnen Prozess. Intern ist ein monolithisches System oft in 3 Schichten aufgeteilt,

---

<sup>5</sup><https://vdos.info/>



The image shows a screenshot of a DOS-style window titled 'vDos'. Inside the window, the text is as follows: At the top left, 'Universität Karlsruhe' and 'Institut für Angewandte Informatik und Formale Beschreibungsverfahren'. At the top right, 'Version 2.3' and 'April 91'. In the center, 'PASIPP - Hauptmenü' followed by a line of equals signs. Below this is a list of options: '<a> Erreichbarkeitsanfragen', '<b> Netzstrukturanfragen', '<c> Simulation', '<d> Erreichbarkeitsbaumgenerierung', '<e> Netz bearbeiten (einlesen, sichern oder ändern)', '<f> DOS - Kommando aufrufen', and '<g> Optionen'. Further down is '<z> PASIPP beenden'. At the bottom, a dashed line separates the menu from a prompt: 'Geben Sie bitte den entsprechenden Buchstaben ein, danach <Return>: ' followed by a cursor. The window has a blue title bar and standard DOS window controls (minimize, maximize, close) in the top right corner.

```
Universität Karlsruhe                                     Version 2.3
Institut für Angewandte Informatik                         April 91
und Formale Beschreibungsverfahren

PASIPP - Hauptmenü
=====

<a> Erreichbarkeitsanfragen
<b> Netzstrukturanfragen
<c> Simulation
<d> Erreichbarkeitsbaumgenerierung
<e> Netz bearbeiten (einlesen, sichern oder ändern)
<f> DOS - Kommando aufrufen
<g> Optionen

<z> PASIPP beenden

-----
Geben Sie bitte den entsprechenden Buchstaben ein, danach <Return>: _
```

Abbildung 6: Hauptmenü von PASIPP (ausgeführt unter vDos)

die GUI, die Geschäftslogik und die Datenhaltung. Jede dieser Schichten deckt jeweils die gesamte Domäne des Systems ab. Bei Client-Server-Systemen kann die serverseitige Anwendung als Monolith umgesetzt werden.

Ein monolithisches System hat den Vorteil, dass keine Netzwerkkommunikation zwischen den einzelnen Systemteilen implementiert werden muss. Innerhalb des Systems im Netzwerk verlorene Daten oder Netzwerkausfälle müssen somit nicht berücksichtigt werden [22]. Allerdings hat ein monolithisches System auch Nachteile. So muss bei jeder Änderung des Codes das gesamte System neu erstellt und verteilt werden. Dies führt dazu, dass die Änderungszyklen einzelner Bestandteile des Systems voneinander abhängig sind [9]. Das System ist nur als Ganzes skalierbar, selbst wenn nur einzelne Teile skaliert werden müssen [9]. Außerdem ist es schwer, über die Zeit eine gute modulare Struktur beizubehalten [9]. Änderungen an einem Teil der Software können dadurch zu nötigen Änderungen in anderen Teilen führen. Monolithische Anwendungen sind daher schwer zu warten [22], was insbesondere für große Systeme gilt [19].

Im Gegensatz zu monolithischen Systemen sind Microservice-Architekturen nicht aus wenigen Schichten aufgebaut, die jeweils die gesamte Systemdomäne abdecken. Stattdessen deckt jeder Service einen Aspekt der Domäne ab. Jeder Service wird dabei unabhängig ausgeführt, verwaltet seine eigenen Daten und bietet leichtgewichtige Kommunikationsmechanismen für andere Services an [2, 9].

Der Rest dieses Abschnitts behandelt weitere Details zu unterschiedlichen Eigenschaften von Microservices.

### 2.5.1 Shared Nothing

Shared Nothing bedeutet, dass Services zwar Drittanbieterbibliotheken benutzen dürfen, aber weder Code noch einen gemeinsamen Zustand mit anderen Microservices teilen dürfen [2]. Geteilter Code würde die lose Kopplung von Services verletzen und Teams beeinträchtigen, da sie die Auswirkungen von Code-Änderungen auf andere Teams berücksichtigen müssten. Geteilter Zustand würde die Skalierbarkeit des Systems verschlechtern: Information müsste in mehreren Orten gleichzeitig gespeichert werden, und bei einer Änderung eines Zustandes müsste jeder betroffene Service umgehend synchronisiert werden.

### 2.5.2 Smart Endpoints, Dumb Pipes

In vielen Systemen, die den Architektur-Ansatz Service-Orientierte Architektur (SOA) anwenden, kommt ein sogenannter Enterprise Service Bus (ESB) zum Einsatz. Ein ESB dient als Kommunikationsverbindung zwischen einzelnen Services, bietet aber neben der reinen Datenübertragung üblicherweise auch anspruchsvolle Dienste an. Dazu gehören Routing und Transformation von Nachrichten, Service-Orchestrierung und -Choreographie und das Anwenden von Geschäftsregeln.

Im Gegensatz dazu folgen Microservices dem Prinzip von *Smart Endpoints, Dumb Pipes* (Smarte Endpunkte, dumme Verbindungen). Das Prinzip ist vergleichbar mit Unix-Services und dem Pipes-und-Filter-Architekturmuster [9, 18]. Eine Microservice-Architektur zielt darauf ab, so lose gekoppelt wie möglich zu sein. Jeder Service ist im vollen Besitz seiner eigenen Domänenlogik. Die komplette Logik des Systems befindet sich innerhalb der jeweiligen Microservices, nicht jedoch in den Kommunikationsverbindungen zwischen ihnen. Der zweite Teil des Prinzips - *Dumb Pipes* - sagt dementsprechend, dass Kommunikationsverbindungen zwischen Services “dumm” sein sollten. Damit ist gemeint, dass sie Informationen nur weiterleiten, nicht aber selbst bearbeiten sollen. Dazu werden häufig REST-Protokolle oder ein leichtgewichtiger Nachrichtenbus wie RabbitMQ benutzt [9], die oft nur geringe Quality-of-Service-Garantien geben.

### 2.5.3 Team-Organisation

Die Architektur großer Systeme ist oft in mehrere technologische Schichten unterteilt. Entwicklungsteams werden entlang dieser Schichten eingeteilt. Das Ergebnis sind spezialisierte Teams, die auf ihrer Schicht jeweils für die gesamte Domäne des Systems verant-

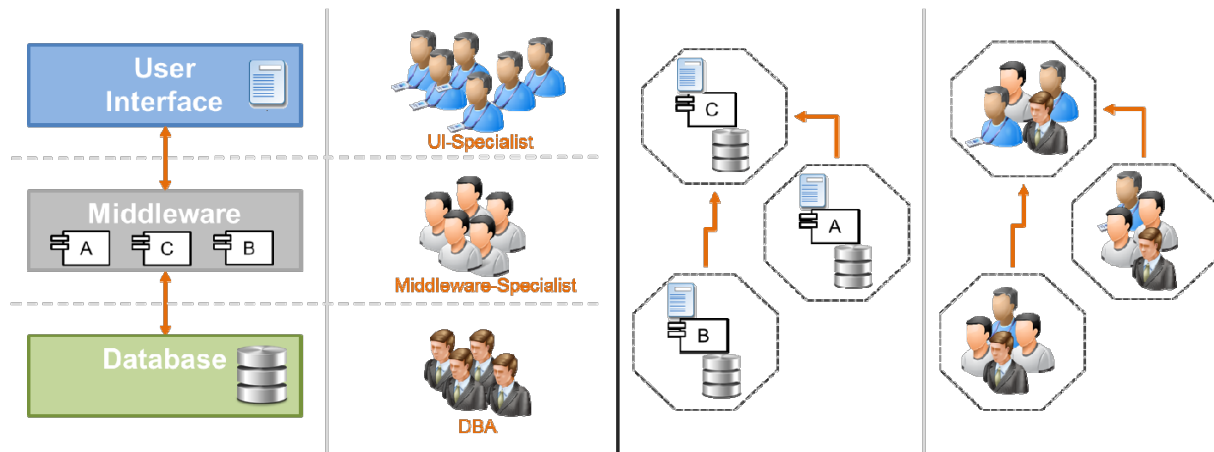


Abbildung 7: Beziehung zwischen Team-Organisation und Software-Architektur [2]

wortlich sind. Typisch sind zum Beispiel ein UI-Team, ein Geschäftslogikteam und ein Datenbankteam. Dies ist ein Fall des Gesetzes von Conway (siehe Abbildung 7):

*“Any organization that designs a system [...] will produce a design whose structure is a copy of the organization’s communication structure.” [7]*  
 (Deutsche Übersetzung: *“Jede Organisation, die ein System entwirft [...] wird einen Entwurf hervorbringen dessen Struktur eine Kopie der Kommunikationsstruktur der Organisation ist.”*)

Microservices unterteilen ein System nicht entlang technologischer Ebenen, sondern entlang der einzelnen funktionalen Features des Systems. Jeder Service setzt dabei alle technologischen Ebenen eines einzelnen Features um, also zum Beispiel User Interface, Geschäftslogik und Datenhaltung. Entwicklerteams werden daher funktionsübergreifend (*cross-functional*) zusammengestellt.

Dabei sollte ein einzelner Service eine einzelne oder wenige Verantwortlichkeiten haben, ähnlich dem *Single Responsibility Principle* aus der Objektorientierten Programmierung [18]. Eine Verantwortlichkeit wird dabei als ein *Grund zur Änderung* betrachtet. Die Aufteilung in Services sollte also so gewählt werden, dass eine Änderung zum Beispiel in den Systemanforderungen möglichst wenige Services beeinflusst. Dies bedeutet, dass Dinge, die sich häufig zusammen ändern, sich im gleichen Service befinden sollten [9].

#### 2.5.4 DevOps

Die klassische Organisation von Entwicklerteams führt dazu, dass selbst kleine Änderungen oft die Koordination mehrerer Teams erfordern. Dies liegt daran, dass sowohl zwischen den Modulen einer Schicht als auch zwischen den Schichten Abhängigkeiten bestehen. Aufgrund der Abhängigkeiten und mangelnden Entkopplung muss oft das gesamte System neu erstellt und verteilt werden. Wenn das System größer wird, wachsen auch die Anzahl

der Abhängigkeiten und somit die Anzahl der Konflikte bei Codeänderungen. Die Änderung monolithischer Systeme ist also mit hohem zeitlichem und koordinativem Aufwand verbunden.

Im Gegensatz dazu sind die einzelnen Services einer Microservice-Architektur tatsächlich lose gekoppelt, mit genau definierten Schnittstellen als einzige Interaktionspunkte zwischen Services. Änderungen an einem Service, die sich nicht auf die angebotenen Interfaces auswirken, bleiben somit innerhalb dieses Services. Da für jeden Service genau ein Team verantwortlich ist, ist von diesen Änderungen nur das jeweilige Team betroffen. Jedes Team ist dabei dafür verantwortlich, dass ihr Service die von außen erwartete Funktionalität aufrecht erhält, das heißt dass weder Interfaces noch sonstige Spezifikationen wie zum Beispiel QoS-Garantien ändern.

Dies führt zu einer Reihe von Vorteilen gegenüber der Entwicklung monolithischer Systeme mit traditioneller Teamorganisation ([5]):

- Verbesserte Verteilbarkeit aufgrund folgender Eigenschaften:
  - Unabhängiges Deployment: Jedes Team kann Änderungen direkt in die Software einbringen, ohne Änderungen von oder Auswirkungen auf andere Teams berücksichtigen zu müssen.
  - Kürzere Deploymentzeiten: Bei Änderungen an monolithischen Systemen muss üblicherweise die ganze Software neu verteilt werden, bei Microservices jedoch nur der betroffene Service. Da diese deutlich kleiner sind, ist auch ihr Deployment-Prozess deutlich schneller. Dies führt zu häufigeren Commits, da einzelne Commits die Produktivität weniger beeinträchtigen.
  - Einfachere Deployment-Prozeduren: Da einzelne Microservices deutlich kleiner als ein monolithisches System sind, ist es einfacher Tools zum automatischen Deployment von Microservices zu bauen. Außerdem wird es praktikabel, generische Tools zum Deployment unterschiedlicher Microservices zu erstellen. Dies erleichtert das Hinzufügen von Features in Form neuer Services deutlich.
  - Keine Downtime (Ausfallzeit): Services können einzeln verteilt werden. Datenbankschemata sind weniger komplex, da sie nur die Domäne einer einzelnen Systemfunktion abdecken. Daher ist es einfacher möglich, einzelne Systemteile zu aktualisieren, ohne das System vom Netz zu nehmen.
- Erhöhte Modifizierbarkeit aufgrund folgender Eigenschaften:
  - Kürzere Zykluszeiten: Da Änderungen in den meisten Fällen nur noch ein einzelnes Team betreffen, sind keine teamübergreifenden Diskussionen und Über-einkommen mehr nötig. Dadurch können Entscheidungen deutlich schneller gefällt werden.

- Inkrementelle Änderung von Qualitätsattributen: Wachsende Qualitätsanforderungen an eine Microservice-Architektur sind oft einfacher zu erfüllen als bei einer monolithischen Architektur: da jeder Microservice für genau eine Funktionalität zuständig ist, und die Services nur lose gekoppelt sind, ist es einfach, den Bottleneck in einem System zu finden. Dieser Service kann dann unabhängig von anderen Services verbessert oder skaliert werden (siehe Unterabschnitt 2.5.5)
- Einfachere Änderung der Technologieauswahl: siehe Unterabschnitt 2.5.6
- Einfachere Upgrades von Sprachen und Bibliotheken: siehe Unterabschnitt 2.5.6

Entwicklerteams von Microservices arbeiten also unabhängig voneinander und sind für alle Aspekte ihres Services verantwortlich, inklusive der Wahl des Technologie-Stacks, der Einhaltung von Interfaces und QoS-Garantien und dem (Re-)Deployment der Services. Dazu benutzen sie häufig automatisierte Deployment-Mechanismen. Da eine Änderung in einem Service Auswirkungen auf andere Services haben könnte, ist jedes Team für die Überwachung des korrekten Verhaltens seiner Services und damit für deren Operation verantwortlich. Diese Praxis wird als *DevOps* (*“Development and Operations”*, “Entwicklung und Betrieb”) bezeichnet.

### 2.5.5 Skalierbarkeit

Abbott stellt in seinem Buch *The Art of Scalability* [1] ein dreidimensionales Skalierbarkeitsmodell für Softwaresysteme vor, den *Skalierbarkeitswürfel*, zu sehen in Abbildung 8. Der Würfel besteht aus 3 Achsen, die jeweils einer Skalierungsdimension entsprechen. X-Achsen-Skalierung entspricht horizontaler Duplizierung. Dabei werden mehrere identische Kopien eines Systems parallel ausgeführt. Ein Beispiel hierfür wäre eine Anwendung, die von mehreren Servern ausgeführt, mit einem vorgeschalteten Load-Balancer-Server.

Bei Z-Achsen-Skalierung werden auch mehrere Kopien einer Anwendung parallel ausgeführt, allerdings ist jede Kopie nur für eine bestimmte Untermenge der Daten zuständig (Datenpartitionierung). Dies kann vor allem bei datenintensiven Anwendungen sinnvoll sein, da jeder Server nur den Teil der Daten speichern muss, der für ihn relevant ist.

Allerdings ist es bei Z-Achsen-Skalierung nötig, ein Partitionierungsmechanismus zu implementieren, was die Komplexität der Anwendung zusätzlich erhöht. Außerdem skalieren beide Ansätze immer die gesamte Anwendung, unabhängig davon, welcher Teil der Anwendung tatsächlich mehr Ressourcen benötigt. Auch löst keiner der beiden Ansätze das Problem steigender Entwicklungs- und Anwendungskomplexität von größer werdenden Systemen [18].

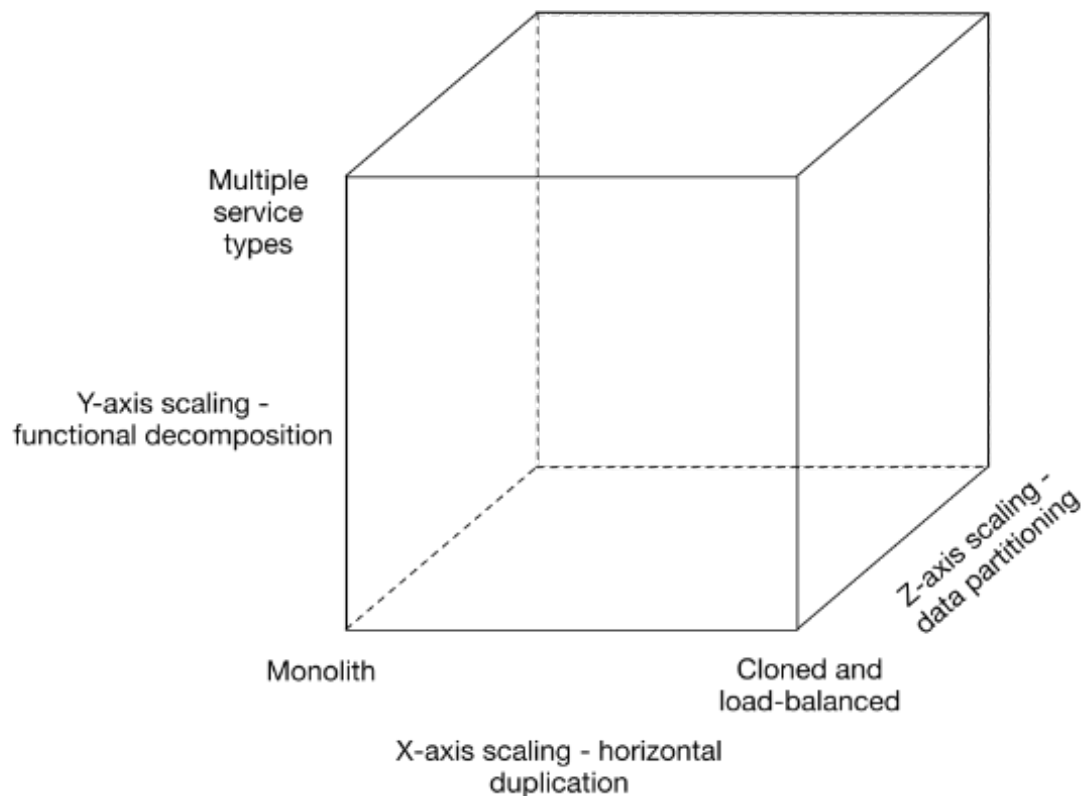


Abbildung 8: Abbotts Skalierbarkeitswürfel ([18], vgl. [1])

Monolithische Architekturen sind auf diese Skalierungsdimensionen beschränkt. Microservices können dagegen auch von der Y-Achsen-Skalierung Gebrauch machen. Bei Y-Achsen-Skalierung wird eine Anwendung in mehrere, unabhängig voneinander ausführbare, funktionale Einheiten aufgeteilt, auch funktionale Dekomposition genannt. Dies entspricht den einzelnen Services einer Microservice-Architektur. Auf diese Weise ist es möglich, genau die Teile eines Systems mehrfach auszuführen, die tatsächlich einen erhöhten Ressourcenbedarf haben, anstatt immer das ganze System duplizieren zu müssen (vgl. Abbildung 9)

### 2.5.6 Polyglot Programming

Die einzelnen Services in einer Microservice-Architektur kommunizieren typischerweise über leichtgewichtige Mechanismen, wie zum Beispiel eine HTTP(S)-API. Während jeder Service genau definierte Interfaces (Schnittstellen) nach außen hin anbietet, ist die Umsetzung des Services selbst nach außen hin eine Black Box. Dies bedeutet, dass die Entwickler frei entscheiden können, wie sie die Funktionalität des Services umsetzen, solange sie sich an das Interface halten. Da jeder Service in einem eigenen Prozess läuft und aufgrund der verwendeten, Programmiersprachen-unabhängigen Kommunikationsmechanismen gilt dies insbesondere auch für die Wahl der Programmiersprachen.

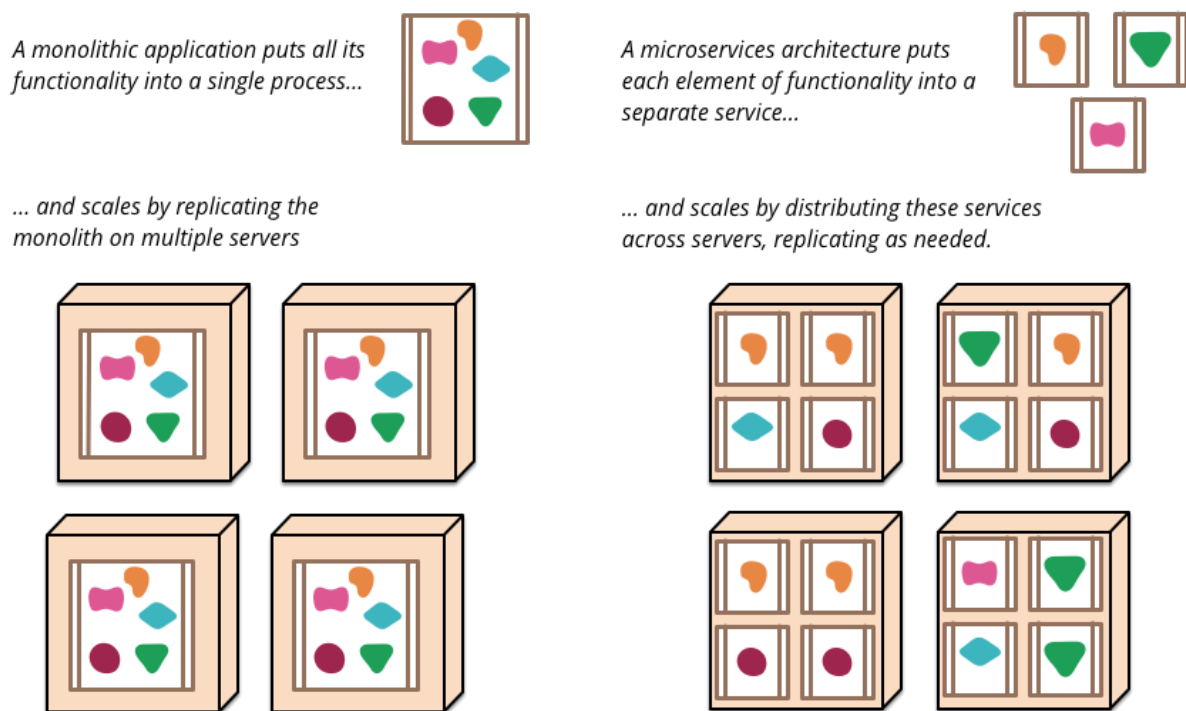


Abbildung 9: Skalierung bei monolithischen und Microservice-Architekturen [9]

Dies ermöglicht die Anwendung mehrerer Programmiersprachen in einer *Polyglot Programming* Strategie [29]. Dabei wird für jeden Service die Programmiersprache gewählt, die sich am besten für dessen Umsetzung eignet. So wäre es zum Beispiel möglich, den Hauptteil eines Programmes in Java zu schreiben, die Evaluierung von Analyseregeln in einem Prolog-Service zu schreiben und die Ausführung zeitintensiver Algorithmen in einen C++-Service auszulagern.

Obwohl auch monolithische teilweise Systeme zu einem gewissen Grad Ansätze für Polyglot Programming anbieten, werden diese Möglichkeiten oft nicht benutzt [9].

Die Trennung der einzelnen Services macht es außerdem einfacher, neue Programmiersprachen und Bibliotheken einzuführen oder auszutesten oder benutzte Versionen zu aktualisieren [5].

### 2.5.7 Dezentralisiertes Datenmanagement und Polyglot Persistence

Die einzelnen Teile eines Softwaresystems behandeln unterschiedliche Teile der Domäne des Systems. Sie haben eine eigene Sicht auf die Domäne und arbeiten daher mit sich voneinander unterscheidenden Daten und Datenstrukturen. Zwei Module eines Systems können dabei unterschiedliche Aspekte des gleichen Domänenobjektes modellieren, oder unterschiedliche Domänenkonzepte unter dem gleichen Namen abbilden (vgl. [9]).

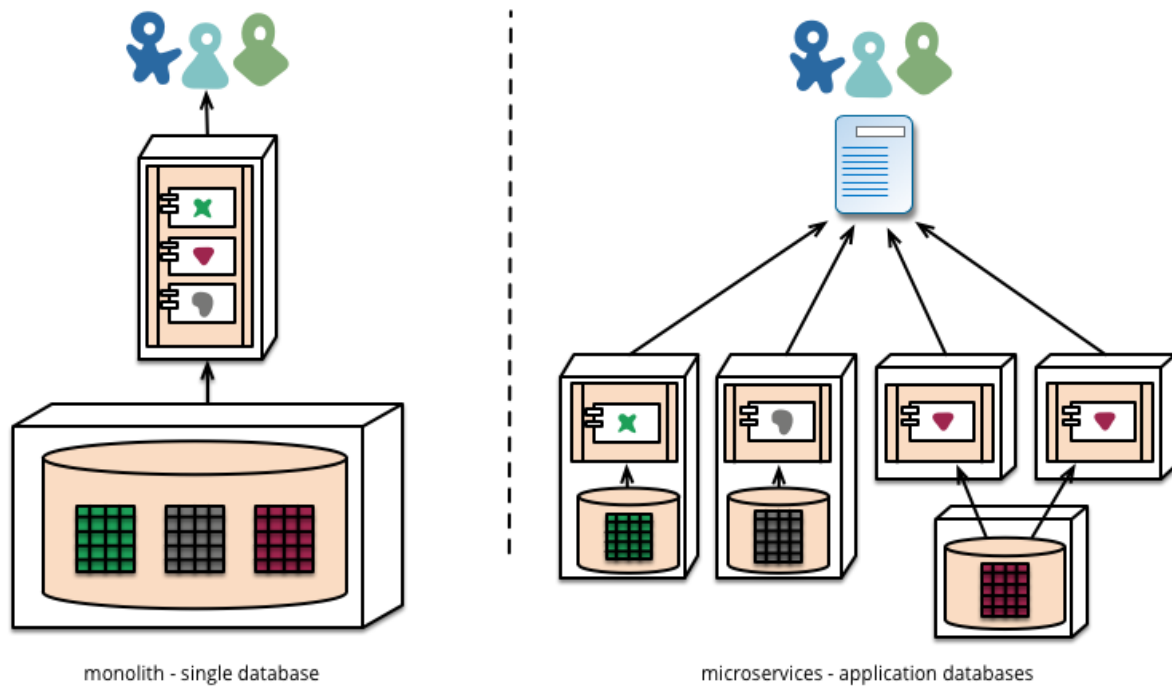


Abbildung 10: Datenhaltung bei monolithischen und bei Microservice-Architekturen [9]

Monolithische Architekturen bevorzugen meist eine einzelne Datenbank für persistente Daten für das gesamte System [9]. Alle Aspekte der Systemdomäne müssen somit von einem einzelnen Datenbankschema abgebildet werden. Das resultierende Schema wird leicht komplex und unübersichtlich. Die Datenbank kann auch große Mengen an Daten enthalten, die nicht zusammen benötigt werden. Dieser Ansatz verstößt außerdem gegen die Softwarearchitekturrichtlinie der losen Kopplung von Komponenten und Modulen: das Ändern des Datenmodells eines einzelnen Moduls kann zur Änderung des zentralen Datenbankschemas und somit zur nötigen Änderung weiterer Module führen. Der Einsatz eines einzelnen unterliegenden Datenbankschemas beschränkt außerdem alle Komponenten der Software auf das gleiche Datenbanksystem. Unterschiedliche Komponenten können allerdings unterschiedliche Anforderungen an die Datenhaltung haben, die von unterschiedlichen Datenbanksystemen erfüllt werden. Zum Beispiel könnte sich eine relationale Datenbank gut für die Zwecke einer Komponente eignen, während eine Graphdatenbank besser für eine andere Komponente geeignet wäre.

Aus diesen Gründen benutzen Microservices dezentralisiertes Datenmanagement: jeder Service verwaltet seine eigenen Daten. Dabei können mehrere Instanzen des gleichen Datenbanktyps zum Einsatz kommen, oder verschiedene Services benutzen unterschiedliche Datenbanksysteme - genannt *Polyglot Persistence* ("Polyglott-Persistenz"). Dieser Ansatz hilft bei der Behebung der oben genannten Probleme, erschwert allerdings die Umsetzung von Funktionalitäten, die Daten von mehreren Services benötigen (vgl. [18]). Die unterschiedlichen Ansätze zur Datenhaltung sind in Abbildung 10 zu sehen.



Eine einfache Lösung wäre das Nutzen von Remote Call Procedures (RPCs): Immer wenn ein Service Daten braucht, die er nicht selbst besitzt, startet er eine RPC an den besitzenden Service. Der aufrufende Service hat dadurch immer die korrekten Daten. Allerdings wirkt sich dieser Ansatz schlecht auf die Verfügbarkeit aus, da der aufgerufene Service selbst verfügbar sein muss. Außerdem erhöht der zusätzliche RPC die Antwortzeit. Eine andere Lösung ist daher, eine Kopie der benötigten Daten zu speichern.

Dies erfordert einen Mechanismus, um Daten, die von mehreren Services benutzt werden, in allen Services aktuell zu halten. Verteilte Transaktionen erfüllen diesen Zweck, allerdings beeinflussen sie die Verfügbarkeit negativ, da alle betroffenen Services verfügbar sein müssen damit die Transaktion abgeschlossen werden kann.

Ein anderer Ansatz sind Event-gesteuerte asynchrone Updates. Dabei löst ein Service ein Event aus, wenn er Daten aktualisiert. Andere Services können diese Events abonnieren, um ihre Daten aktuell halten zu können. Dieser Ansatz setzt lose Kopplung der Services um, und erhöht die Verfügbarkeit. Allerdings tauscht er Konsistenz gegen Verfügbarkeit und Reaktionszeit: für die redundant gespeicherten Daten kann nur noch *eventual consistency* ("letztendliche Konsistenz") garantiert werden. Die Services müssen so geschrieben werden, dass sie hiermit umgehen können. Gegebenenfalls muss das Ausführen von Rollbacks möglich gemacht werden.

## 2.6 Vorgeschlagene Microservice-Architektur

In [2] wird eine Microservice-basierte Architektur für Modellierungs- und Analysetools für Geschäftsprozesse vorgeschlagen. Außerdem wird eine mögliche Implementierung eines solchen Geschäftsprozesseditors auf Basis dieser Architektur erklärt. Dieser Abschnitt stellt in Unterabschnitt 2.6.1 zuerst die vorgeschlagene Architektur und dann in Unterabschnitt 2.6.2 die Implementierung vor.

### 2.6.1 Architektur

Die vorgeschlagene Microservice-Architektur ist vereinfacht in Abbildung 11 abgebildet. Die Anwendung ist entlang von Entitäten und Ressourcen in Microservices aufgeteilt. Jeder Service deckt dabei jede Operation ab, die eine bestimmte Ressource betrifft (z.B. ein Petri Netz).

Die Architektur besteht aus 4 Gruppen von Servicetypen: Editoren, Management, Analysefunktionalität und Präsentation. Sie ist so ausgelegt, dass zusätzliche Services jederzeit hinzugefügt werden können. Entsprechend der Microservice-Idee läuft jeder Service unabhängig in einem eigenen Prozess mit eigener Datenhaltung. Die Services sind dadurch lose gekoppelt, und für jeden Service kann ein eigener Technologie-Stack (sowohl

Programmiersprachen als auch Datenhaltung) benutzt werden, der den Anforderungen des Services am ehesten gerecht wird. Daten werden mitunter redundant in mehreren Services gespeichert, es gilt allerdings das Prinzip der *eventual consistency* (siehe Unterabschnitt 2.5.7). Die Kommunikation zwischen Services und mit einem Client geschieht über REST-Schnittstellen. Jeder Service kann einzeln angesprochen werden. Darüber hinaus gibt es außerdem einen API-Gateway zur einfacheren Benutzung der Services. Gemäß dem *Smart Endpoints, Dumb Pipes*-Prinzip (siehe Unterabschnitt 2.5.2) enthält der API-Gateway allerdings keine anspruchsvolle Logik; diese ist nur in den Services selbst enthalten.

Für die Anzeige ist es nötig, Ergebnisse unterschiedlicher Microservices zu kombinieren. Dafür sind die Präsentationsservices zuständig. Der Rendering-Service baut dabei die Grundstruktur der Seite auf, während die Frontend-Integration Services diese mit Informationen füllen. Analyse- und Simulationsergebnisse können von Reporting-Services aggregiert werden, bevor sie von einem Präsentationsservice angezeigt werden. Die verfügbaren Präsentationen können einfach an geänderte Anforderungen angepasst werden, da nur ein neuer GUI-Service geschrieben werden muss.

### 2.6.2 Implementierung

Nach der Besprechung der Architektur schlägt [2] eine Implementierung für Petri-Netzeditoren vor, die auf dieser beruht. Die Implementierung folgte einer iterativen Entwicklung. Zuerst wurden grundlegende Microservices implementiert, die BPM allgemein unterstützen. Dazu zählen zum Beispiel Services zum Speichern und Exportieren von Modellen oder ein Toolbar Services, der die Symbole der verfügbaren Editorwerkzeuge zurückgibt. Daraufhin wurden Services für Analyse und Reporting implementiert, zum Beispiel ein Reachability und ein Token Game Service. Da alle Analyse-Services den Standard PNML benutzen, sind die Services einfach wiederzuverwenden.

Um Endnutzern eine Funktionalität zur Verfügung zu stellen, wurden zum Schluss zwei GUI-Editoren erstellt, die von den restlichen Services Gebrauch machen. Einer der Editoren ist für iOS und Android optimiert, der andere ist web-basiert und für Browser auf Desktop-Systemen optimiert.

## 2.7 REST

Representational State Transfer (REST) ist ein Architekturstil für den Entwurf verteilter Anwendungen. REST dient hauptsächlich der Maschine-zu-Maschine-Kommunikation, also zum Beispiel zur Bereitstellung von APIs. Der Unterschied zu anderen Architekturstilen für verteilte Systeme ist vor allem die Anforderung nach einheitlichen Schnittstellen.

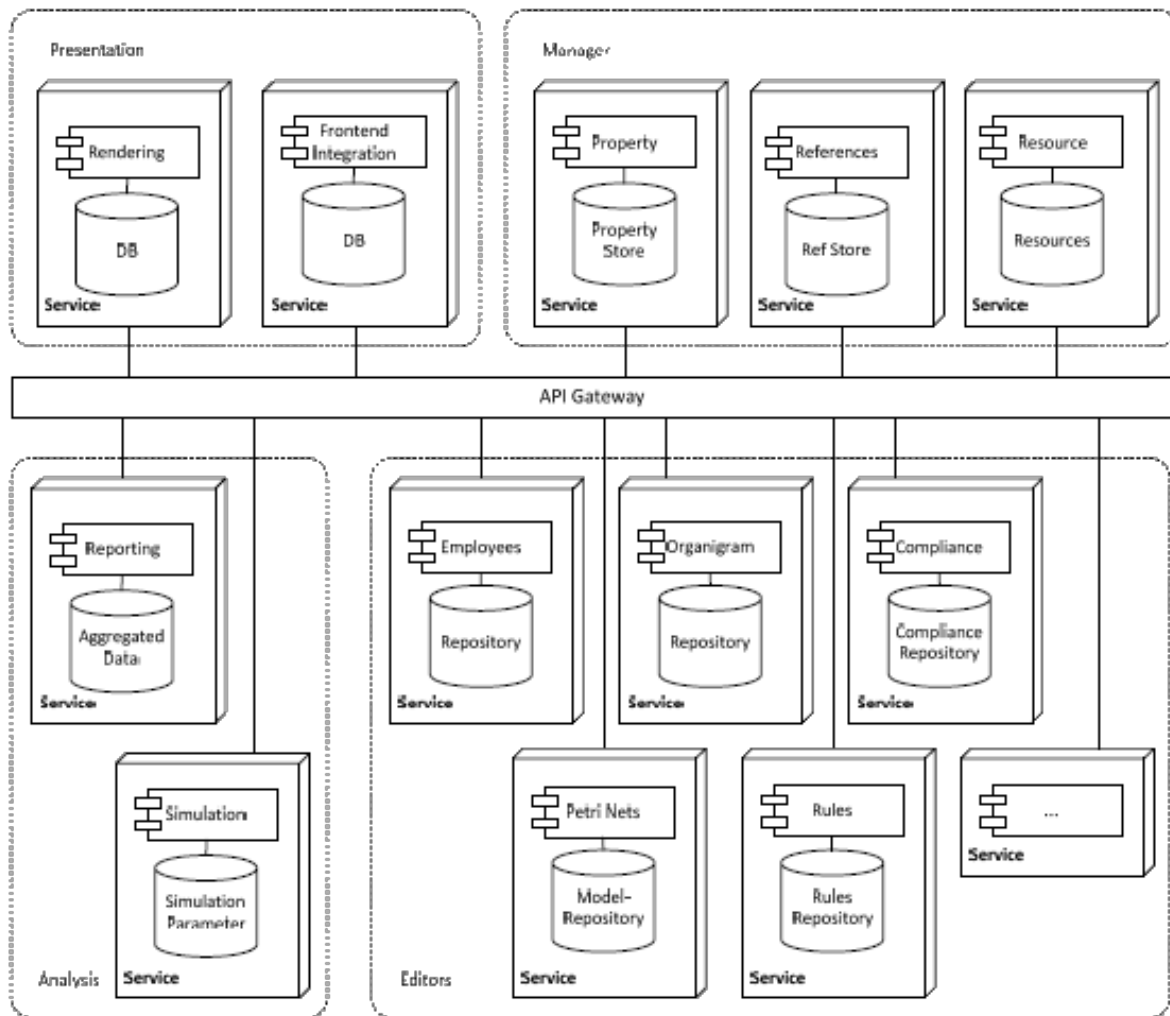


Abbildung 11: Microservice-Architektur des BPM-Editors [2]

REST stellt eine Abstraktion des Protokolls HTTP zu einem Architekturstil dar. Viele Online-Dienste sind bereits von sich aus REST-konform. REST-konforme Dienste werden auch als *RESTful web services* bezeichnet.

Die REST-Architektur wird durch sechs Einschränkungen (*constraints*, auch als Prinzipien bezeichnet) definiert, die REST-konforme Dienste erfüllen müssen [8]:

**Client-Server:** REST folgt der Client-Server-Architektur.

**Zustandslosigkeit:** Jede REST-Nachricht enthält alle Informationen, die zur Verarbeitung der Information nötig sind. Dies bedeutet, dass der Server keine Zustands- oder Sitzungsinformationen zwischenspeichern muss. Dies erleichtert die Skalierung von Anwendungen auf mehrere Server.

**Caching:** Clients können Caching verwenden, um mehrfache Anfragen zu vermeiden. Antworten des Servers müssen definieren, ob sie cachebar sind oder nicht.

**Einheitliche Schnittstellen:** Die Forderung nach einheitlichen Schnittstellen ist ein fundamentaler Teil eines REST-Services. Sie dient der losen Kopplung von (Teil-)Systemen und somit deren unabhängiger Weiterentwicklung. Die Einschränkung auf einheitliche Schnittstellen besteht selbst aus vier Einschränkungen. Die erste ist die *Adressierbarkeit von Ressourcen*: REST-Anfragen identifizieren individuelle Ressourcen, wie zum Beispiel einen Datenbankeintrag oder eine URI. Die Ressource selbst ist dabei unabhängig von ihrer Repräsentation. Ein Server kann zum Beispiel eine Ressource im JSON- oder im XML-Format zurückgeben, wobei keine der internen Repräsentation des Servers entsprechen muss. *Manipulation von Ressourcen durch Repräsentationen*: Der Client kann Ressourcen manipulieren, indem er eine Repräsentation der erwarteten Ressource an den Server sendet, der seine interne Repräsentation entsprechend anpasst.

*Selbstbeschreibende Nachrichten*: Jede Nachricht sollte alle Informationen enthalten, die für ihre Verarbeitung benötigt werden. Dies wird unter anderem durch Verwendung von Standardmethoden wie zum Beispiel HTTP GET, HTTP POST etc. erreicht. *Hypermedia as the Engine of Application State*: Wenn der Client eine initiale URI der REST-Anwendung aufruft - analog zur Startseite einer Internetseite - sollte er dynamisch (ohne Hard-Coding) aus der Serverantwort alle weiteren Ressourcen und Methoden der Anwendung entdecken können. Dazu enthält die Server-Antwort Hyperlinks zu anderen verfügbaren Aktionen und Ressourcen.

**Mehrschichtige Systeme:** Anwendungen können aus mehreren Schichten aufgebaut sein, sollen dem Client aber nur ein Interface zur Verfügung stellen.

**Code on Demand:** Diese Einschränkung ist optional. Ausführbarer Code, wie zum Beispiel JavaScript, wird bei Code on Demand erst im Bedarfsfall vom Server an den Client übertragen.

## 3 Konzept

Im Rahmen der Arbeit wurde ein Java-Microservice erstellt, der die Funktionen des Programms PASIPP (siehe Abschnitt 2.4) in modernen Software-Infrastrukturen wieder verfügbar macht. In diesem Kapitel wird das Konzept dargestellt, nach dem der Microservice entwickelt wurde.

Abschnitt 3.1 stellt die Softwarearchitektur vor, die dem Programm zugrunde liegt. Abschnitt 3.2 erklärt, wie die in Prolog implementierte Funktionalität innerhalb des Programmes genutzt wird. Abschnitt 3.3 erklärt die Ein- und Ausgabeformate des Microservices. Abschnitt 3.4 erklärt die serverseitige Ausführung des Microservices und geht auf die Funktionalität ein, die er an Clients anbietet. In Abschnitt 3.5 wird darauf eingegangen, wie das Programm mit Fehlern umgeht.

### 3.1 Architektur

Das erstellte Programm ist unterteilt in zwei Komponenten (siehe Abbildung 12): eine *Simulator*-Komponente und eine *Server*-Komponente. Die Simulator-Komponente erbringt die Geschäftslogik des Programms, Die Server-Komponente bietet die Funktionalität nach außen in einem REST-Interface an. Die Simulator-Komponente ist in Unterabschnitt 3.1.1 genauer erklärt, die Server-Komponente in Unterabschnitt 3.1.2.

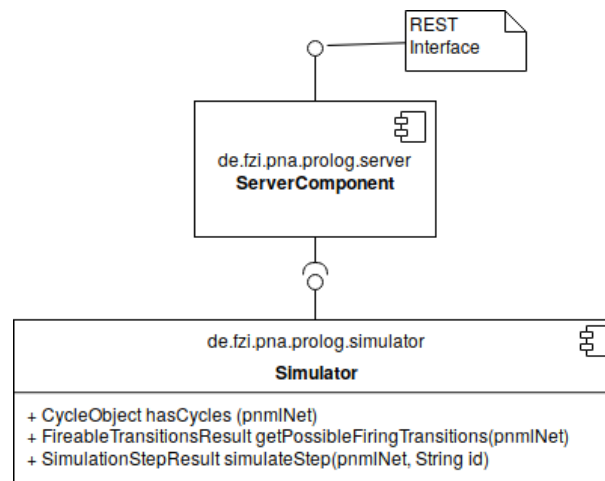


Abbildung 12: Unterteilung des Microservices in eine *Server*- und eine *Simulator*-Komponente

#### 3.1.1 Simulator

Der Entwurf der Simulator-Komponente kann in Abbildung 13 gesehen werden. Die Hauptklasse der Komponente trägt ebenfalls den Namen *Simulator*. Sie bietet nach au-

ßen hin vier Funktionalitäten an: *hasCycles(pnmlNet)*, *getPossibleFiringTransitions(pnmlNet)*, *simulateStep(pnmlNet, String id)* und *getReachabilityTree(pnmlNet)*.

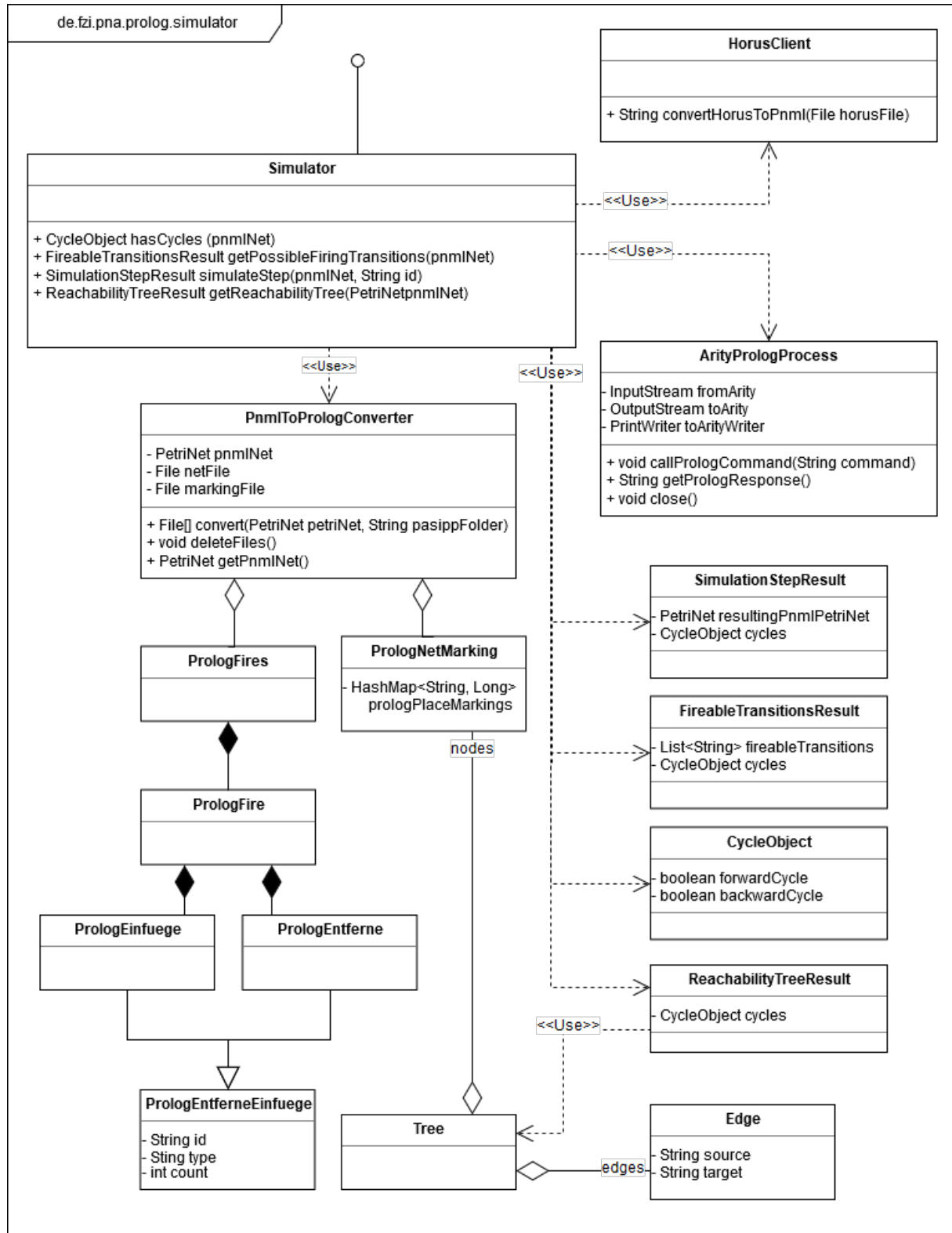


Abbildung 13: Architektur der *Simulator*-Komponente (Klassendiagramm)

Die Methoden realisieren folgende Funktionalitäten: *hasCycles* gibt für ein gegebenes Petri-Netz zurück, ob es Vorwärts- und/oder Rückwärtszyklen enthält; *getPossibleFiringTransitions* gibt alle Transitionen zurück, die bei der aktuellen Markierung schalten können; *simulateStep* nimmt ein Petri-Netz und eine Transition-ID entgegen, führt die Transition aus, und gibt das resultierende Petri-Netz zurück; und *getReachabilityTree* gibt für ein gegebenes Petri-Netz inklusive Startmarkierung den Erreichbarkeitsbaum aller von der Startmarkierung aus erreichbaren Markierungen aus. Die Funktionalität der Methoden wird in Abschnitt 3.4 genauer erklärt.

Der Parameter *pnmlNet* kann dabei eine PNML-Datei sein, ein PetriNet-Objekt<sup>6</sup>, oder der Inhalt einer PNML-Datei als String. Um dies zu ermöglichen, gibt es für jede Funktionalität drei Methoden. Jeweils zwei der drei Methoden konvertieren lediglich ihre Parameter und rufen dann eine der anderen Methoden auf. Die eigentliche Funktionalität ist also für alle Parametertypen dieselbe.

Die *Simulator*-Klasse erwartet, dass es sich bei dem übergebenen Petri-Netz um ein PNML Place-/Transition-Netz oder um ein HORUS Ablaufmodell handelt. Falls das übergebene PNML-Netz kein standardkonformes Place-/Transition-Netz ist, geht der Simulator davon aus, dass es sich um ein HORUS Ablaufmodell handelt und versucht, dieses in standardkonformes PNML umzuwandeln. Dies geschieht in der Klasse *HorusClient*. Die Klasse *HorusClient* benutzt einen externen Microservice, der HORUS-PNML in standardkonformes PNML umwandeln kann [25]. *HorusClient* geht bei der Konvertierung davon aus, dass der externe Microservice unter localhost auf dem Port 8090 läuft.

Für die Rückgabewerte von jeder der vier Funktionalitäten existiert eine Klasse, die die zurückzugebenden Werte bündelt. *hasCycles* gibt ein *CycleObject* zurück, das zwei Booleans, *hasForwardCycle* und *hasBackwardCycle*, enthält. *getPossibleFiringTransitions* gibt die Klasse *fireableTransitionsResult* zurück, das ein *CycleObject* enthält und eine String-Liste, die die IDs der schaltbaren Transitionen enthält. *simulateStep* gibt die Klasse *SimulationStepResult* zurück, die das resultierende Petri-Netz und ein *CycleObject* enthält. *getReachabilityTree* gibt ein Objekt der Klasse *ReachabilityTreeResult* zurück, das ein *CycleObject* und ein Objekt der Klasse *Tree* enthält. Die Klasse *Tree* enthält die Struktur des Erreichbarkeitsbaums. Die Knoten des Baumes sind in einer *Map* gespeichert, die jeweils einem PrologNetMarking eine String-ID zuweist. Die Kanten des Baumes werden als Liste von Objekten der Klasse *Edge* gespeichert. Objekte der Klasse *Edge* enthalten jeweils einen String *source* und *target*, die die ID der Quelle beziehungsweise des Ziels der Kante enthalten.

Um die PNML-Eingaben in Prolog-Notation umzuwandeln benutzt *Simulator* die Klasse *PnmlToPrologConverter*. Wie in Unterabschnitt 2.4.1 bereits erwähnt, arbeitet PASIPP

---

<sup>6</sup>ein Objekt der Klasse *fr.lip6.move.pnml.ptnet.PetriNet*

mit zwei Dateien: einer Datei, die die Struktur des Netzes speichert, und einer Datei, die die Markierung des Netzes speichert.

Die Netzdatei enthält für jede Transition eine Schaltregel (*fire*-Regel). Eine *fire*-Regel enthält für jede eingehende Stelle eine *entferne*-Regel, die angibt, wie viele Tokens von dieser Stelle entfernt werden, und eine *ein fuege*-Regel, die angibt, wie viele Tokens an diese Stelle ausgegeben werden. Um diese Datei einfach erstellen zu können, wurde ihre Struktur beim Entwurf der Klassen nachgeahmt: *PnmlToPrologConverter* erstellt zuerst ein *PrologFires*-Objekt. Diese Klasse entspricht der Gesamtheit der Informationen der Netzdatei. Sie enthält für jede Transition ein *PrologFire*-Objekt. Jedes *PrologFire*-Objekt verfügt wiederum über eine Liste von *PrologEin fuege*- und *PrologEntferne*-Objekten. Diese entsprechen den einzelnen *ein fuege*- und *entferne*-Regeln dieser Transition. Über das Aufrufen der *toString*-Methode der jeweiligen Klassen werden diese direkt in ihre entsprechende Prolog-Notation übertragen. *PrologEntferne* und *PrologEin fuege* wurden aufgrund ihrer Ähnlichkeit zueinander (gleiche Variablen, ähnliche *toString*-Methode) als Unterklasse derselben Elternklasse *PrologEntferneEin fuege* implementiert.

Die Markierungsdatei eines Netzes besteht aus den Markierungen der Netzstellen, und deren Anzahl. PASIPP bietet die Möglichkeit, Markierungen mit einer Struktur zu versehen und diese Struktur mit individuellen Werten zu belegen. Da das für diese Arbeit geschriebene Programm allerdings mit PNML ohne spezielle Erweiterungen arbeitet, welches nur anonyme Markierungen bietet, wurden in dem Programm nur anonyme Markierungen umgesetzt (mehr Details zu Markierungen enthält Unterabschnitt 2.4.1). Die Informationen der Markierungsdatei werden von einem *PrologNetMarking*-Objekt gespeichert. Dieses enthält eine *HashMap<String, Long> placeMarkings*, die die Markierung der einzelnen Stellen in Form einer Zuweisung von Markenanzahl an die Stellen-ID speichert. Wie bei den Klassen der Netzdatei ist es möglich, über die *toString*-Methode auch die Markierungsklasse direkt in ihre Prolog-Notation zu überführen.

Bisher erwähnt wurden die Klassen zur Transformation der PNML-Eingabe in eine Prolog-Repräsentation, und die Klassen zur Rückgabe von Ergebnissen. Außerdem benutzt Simulator die Klasse *PrologWrapper*. Diese erstellt bei Instanziierung einen externen Prozess, startet darin den Arity/Prolog32-Interpreter und verknüpft einen Input- und einen OutputStream damit. Nach außen hin bietet die Klasse die Methoden *void callPrologCommand(String command)* und *String getPrologResponse()* an, um mit dem Prozess zu interagieren, und *void close()*, um den Prozess zu beenden. Die Klasse kapselt somit die Interaktion mit dem Prolog-Prozess an einem einzelnen Ort.

Ein Vorschlag, wie die Server-Komponente um neue Funktionalitäten erweitert werden kann, findet sich in Unterabschnitt 4.3.1.



### 3.1.2 Server

Die Server-Komponente des Microservices ist im Package *de.fzi.pna.prolog.server* enthalten und wurde mit dem Jersey Framework für RESTful Web Services<sup>7</sup> umgesetzt. Sie enthält eine Klasse *Main*, die die *main*-Methode des Programms enthält. Diese Methode durchsucht das *server*-Package nach JAX-RS-Ressourcen und startet einen Server-Prozess. JAX-RS (*Java API for RESTful Web Services*) ist eine JAVA-API, die dem Jersey-Framework zugrunde liegt und das Erstellen von RESTful Web Services mithilfe von Annotationen ermöglicht.

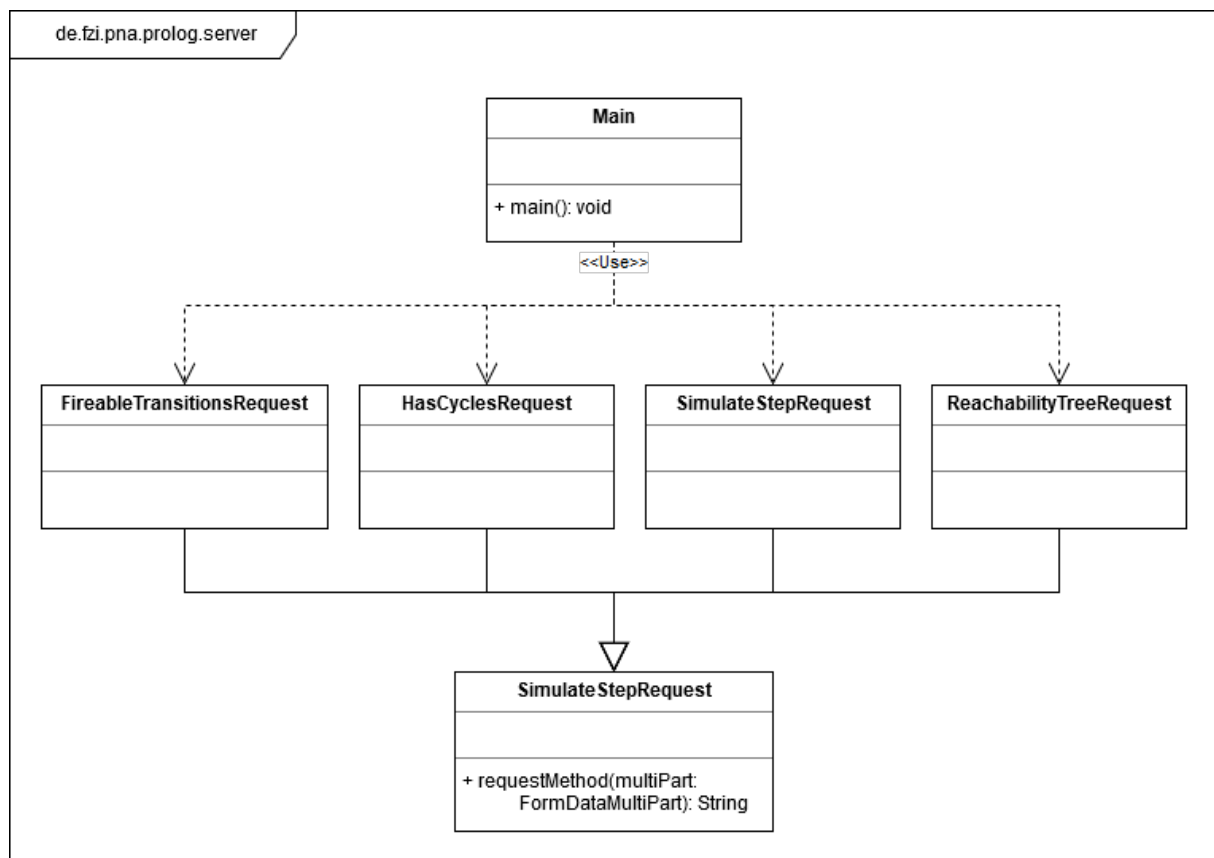


Abbildung 14: Architektur der *Server*-Komponente (Klassendiagramm)

Daneben enthält die Komponente für jede der vier Funktionalitäten der Simulator-Komponente (siehe Unterabschnitt 3.1.1) eine JAX-RS-Ressourcenklasse. Jede der vier Klassen nimmt den HTTP Request-Body entgegen, extrahiert daraus die Parameter, ruft die entsprechende Methode der *Simulator*-Komponente auf und übergibt ihr dabei die extrahierten Parameter. Um doppelten Code zu vermeiden, wurden die vier JAX-RS-Ressourcenklassen als Unterklasse einer gemeinsamen Oberklasse *SimulatorRequest* implementiert. Die Architektur der Server-Komponente ist in Abbildung 14 abgebildet. Die

<sup>7</sup><https://jersey.github.io/>

prinzipielle Implementierung und eine Empfehlung, wie neue JAX-RS-Ressourcenklassen zu dem Server hinzugefügt werden können, findet sich in Unterabschnitt 4.3.2.

### 3.2 Verwendung von Prolog und PASIPP

Für diese Arbeit lagen sowohl die Quelldateien des Programmes PASIPP vor, als auch eine kompilierte, ausführbare Version davon, die unter virtuellen DOS-Betriebssystemen wie zum Beispiel vDos<sup>8</sup> benutzt werden kann. Für den Microservice wurde nicht das ausführbare Programm, sondern die Quelldateien von PASIPP verwendet. Dies hat den Vorteil, dass die verwendete Funktionalität direkt genutzt werden kann und weniger Implementationsarbeit für die Navigation durch das Programm aufgebracht werden muss. Für die Nutzung von PASIPP wurden zwei kleine Änderungen an den Quelldateien vorgenommen, die in Abschnitt 4.1 erklärt werden.

Zur Nutzung von Prolog in Java sahen vor allem JPL<sup>9</sup> und Projog<sup>10</sup> vielversprechend aus. JPL ist ein Prolog/Java-Interface, das die Einbindung von Prolog in Java (und von Java in Prolog) ermöglicht. Prolog-Anfragen werden dabei im Hintergrund an ein auf dem Computer installiertes Prolog-System weitergegeben und dann als Java-Objekt zurückgegeben.

Bei dem von JPL im Hintergrund genutzten Prolog-System handelt es sich um SWI-Prolog<sup>11</sup>, eine freie Implementierung der Prolog-Programmiersprache. Beim Laden der PASIPP-Dateien in SWI-Prolog gibt das System Fehler- und Warnmeldungen aus, kann die Wissensbasis aber trotzdem laden. Beim Benutzen der Wissensbasis ergeben sich allerdings Probleme, da SWI-Prolog einige der von PASIPP benutzen, in Arity/Prolog eingebauten, Systemregeln nicht unterstützt. Dies ist unter anderem die Regel *directory*, die den Inhalt eines Ordners auflistet. Die Regel wird beim Laden von Netz- und Markierungsdateien benutzt und wird somit für die Funktionalität von PASIPP benötigt. Da das Ziel der Arbeit war, PASIPP mit möglichst wenig Anpassungen zu nutzen, schied SWI-Prolog und damit JPL zur Nutzung von Prolog aus.

Projog ist eine Java-Implementierung von Prolog, die eine API anbietet, um Prolog innerhalb von Java-Anwendungen zu nutzen. Beim Testen von Projog mit Übungsaufgaben aus einer Programmierparadigmen-Vorlesung [21] zeigte sich allerdings, dass Projog teilweise nicht vorgesehene Ergebnisse lieferte. Dies war der Fall für die in Codeausschnitt 15 gezeigten Regeln *geben* und *godd*. Die Regel *geben* dient zur Generierung gerader Zahlen größer oder gleich 0; *godd* dient zur Generierung ungerader Zahlen größer 0. Die Regel

---

<sup>8</sup><https://vdos.info/>

<sup>9</sup><http://www.swi-prolog.org/packages/jpl/>

<sup>10</sup><http://www.projog.org/index.html>

<sup>11</sup><http://www.swi-prolog.org/>

generiert dabei jedes Mal, wenn der Nutzer ; drückt, eine neue Zahl und endet, sobald der Nutzer *Enter* drückt. Sowohl SWI-Prolog als auch Arity/Prolog32 generierten bei einem Test beliebig viele gerade beziehungsweise ungerade Zahlen; Projog allerdings generierte nur jeweils die erste gerade beziehungsweise ungerade Zahl, also 0 beziehungsweise 1.

Außerdem zeigte sich später, dass Projog keine Definition von Modulen unterstützt. Beim Laden der PASIPP-Quelldateien tritt deshalb eine *ParserException* auf, und die Wissensbasis wird nicht geladen. Aus diesen beiden Gründen wurde auch JPL nicht verwendet.

```

1 geben(0) .
2 geben(X) :- godd(Y), X is Y+1, X>0.
3 godd(1) .
4 godd(X) :- geben(Y), X is Y+1, X>1.

```

Codeausschnitt 15: Regeln *geben* und *godd* zur Generierung von geraden beziehungsweise ungeraden Ganzzahlen (vgl. [21])

Stattdessen wird der Arity/Prolog32-Interpreter ohne Verwendung einer API direkt genutzt (siehe Abbildung 15). Dazu wird der Interpreter aus Java heraus als externer Prozess gestartet. Es wird ein *OutputStream* und ein *PrintWriter* genutzt, um Kommandozeileingaben an den Arity-Prozess zu schicken. Die Ausgaben des Prozesses werden über einen *InputStream* gelesen. Die Interaktion mit dem Interpreter-Prozess wurde in die Klasse *PrologWrapper* gekapselt. Bei der Erzeugung eines *PrologWrapper*-Objektes (*new PrologWrapper*) wird ein Prolog-Prozess gestartet (*:Process*), und ein *InputStream* (*fromArity*), ein *OutputStream* (*toArity*) und ein *PrintWriter* (*toArityWriter*) werden damit verknüpft.

Nach außen hin bietet die Klasse die Methoden *void callPrologCommand(String command)* und *String getPrologResponse()*, um mit dem Prozess zu interagieren. Außerdem bietet sie die Methode *void close*, um den Prozess zu beenden. Das Sequenzdiagramm in Abbildung 15 stellt den Ablauf dieser Funktionen und die Instanziierung eines *PrologWrapper*-Objektes schematisch dar.

Bei der Instanziierung eines Objektes dieser Klasse wird der Prolog-Interpreter-Prozess gestartet. Daraufhin wird ein *InputStream* und ein *OutputStream*, der seinerseits an einen *PrintWriter* gebunden ist, mit dem Prozess verbunden. *Input* und *Output* bezeichnen dabei die Situation aus Sicht des Microservices, nicht des Prolog-Prozesses: der *InputStream* repräsentiert den Input, das der Microservice von Prolog erhält und somit den Output des Prolog-Prozesses; analog repräsentiert der *OutputStream* die Daten, die der Microservice an den Prolog-Prozess ausgibt, also den Input aus der Sicht des Prolog-Prozesses. Dementsprechend wurden die Bezeichnungen *fromArity* für den *InputStream* und *toArity* und *toArityWriter* für den *OutputStream* beziehungsweise den *PrintWriter* vergeben.

Die Methode *void callPrologCommand(String command)* übergibt den entgegen genommenen String an den *toArityWriter*, der ihn in den *toArity* schreibt. *toArity* ist der Input des Prolog-Prozesses, der nach dem Schreibvorgang den Befehl ausführt. Alle Ausgaben des Prolog-Prozesses werden automatisch in *fromArity* geschrieben. Die Methode *getPrologResponse()* liest *fromArity* aus und gibt dessen Inhalt zurück. Sie liefert immer die gesamte Ausgabe des Prolog-Prozesses seit dem letzten Aufruf von *getPrologResponse()*. Die Methode *close()* beendet den Prolog-Prozess.

### 3.3 Ein- und Ausgabeformate

Um die Benutzung des Microservices für Clients möglichst einfach zu machen, erfolgt die Ein- und Ausgabe von Petri-Netzen in einem standardisierten Format. Die Wahl fiel dabei auf die Petri Net Markup Language (PNML) (siehe Abschnitt 2.2). PNML-Dokumente können mehrere Netze beinhalten, die wiederum jeweils aus mehreren *Pages* bestehen können. Enthält der Microservice ein PNML-Dokument als Teil eines HTTP-Requests, berücksichtigt er lediglich das erste Netz und davon die erste Seite; andere PNML-Objekte werden von dem erstellten Microservice nicht berücksichtigt. Abbildung 16 zeigt ein beispielhaftes, einfaches Petri-Netz. Codeausschnitt 16 zeigt die PNML-Repräsentation des Netzes.

PNML basiert auf dem Datenaustauschformat XML und lässt sich somit einfach in XML-Daten integrieren. Aus diesem Grund wird als Format für die Rückgaben XML verwendet. Das äußerste XML-Element gibt dabei immer an, um welche Rückgabe es sich handelt. Entsprechend den angebotenen Funktionen (siehe Abschnitt 3.1 oder 3.4) heißt das äußerste Element *hasCyclesResult*, *fireableTransitionsResult*, *simulateStepResult* oder *reachabilityTreeResult*. Eine Beschreibung der Rückgaben der einzelnen Endpoints und Beispiele für diese Rückgaben sind in Abschnitt 3.4 zu finden.

### 3.4 Benutzung des Services

Für die Benutzung des Microservices ist es zuerst nötig, ihn auf einem Server auszuführen. Dies wird kurz in Unterabschnitt 3.4.1 beschrieben.

Die eigentliche Nutzung erfolgt durch Senden von Requests von einem Client an den Server. Der Microservice bietet seinen Clients vier Funktionen des Programms PASIPP an, die in einer HTTP REST API umgesetzt wurden. Für jede der Funktionen gibt es einen eigenen Endpoint. Unterabschnitt 3.4.2 bis 3.4.5 erklären die vier Funktionen, inklusive dem HTTP Endpoint, seiner erwarteter Parameter und seiner Rückgabewerte. Die Rückgabe dieser Endpoints erfolgt dabei immer im XML-Format, wie in Abschnitt 3.3

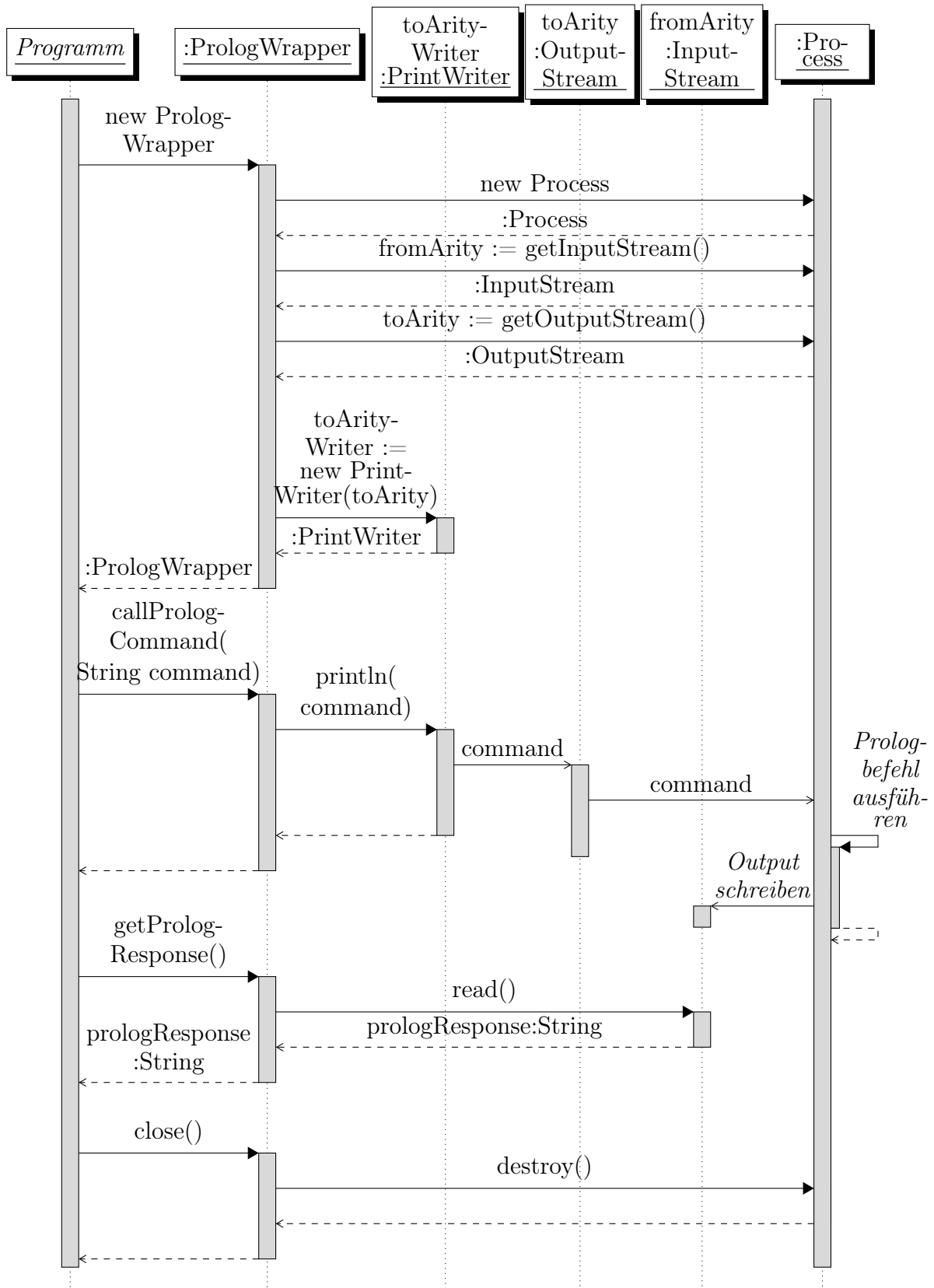


Abbildung 15: Veranschaulichung der öffentlichen Methoden der Java-Klasse *PrologWrapper* (Konstruktor, *void callPrologCommand(String command)*, *String getPrologResponse()*, *void close()*) als Sequenzdiagramm

```
1 <pnml xmlns="http://www.pnml.org/version-2009/grammar/pnml">
2 <net id="net0" type="http://www.pnml.org/version-2009/grammar/ptnet">
3   <page id="page0">
4     <place id="place1">
5       <initialMarking><text>2</text></initialMarking>
6     </place>
7     <place id="place2">
8       <initialMarking><text>3</text></initialMarking>
9     </place>
10    <place id="place3">
11      <initialMarking><text>1</text></initialMarking>
12    </place>
13    <place id="place4"/>
14    <transition id="transitionA"/>
15    <transition id="transitionB"/>
16    <arc id="arc1" source="place1" target="transitionA"/>
17    <arc id="arc2" source="place1" target="transitionA"/>
18    <arc id="arc3" source="place2" target="transitionA"/>
19    <arc id="arc4" source="transitionA" target="place3"/>
20    <arc id="arc5" source="place3" target="transitionB"/>
21    <arc id="arc6" source="place3" target="transitionB"/>
22    <arc id="arc7" source="transitionB" target="place4"/>
23  </page>
24 </net>
25 </pnml>
```

Codeausschnitt 16: Petri-Netz in PNML

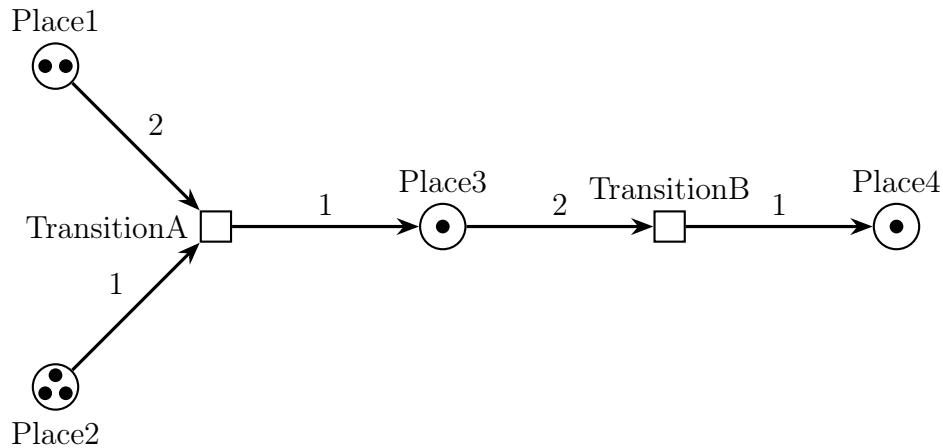


Abbildung 16: Beispiel-Petri-Netz

erklärt. Unterabschnitt 3.4.6 geht auf zwei zusätzliche Endpoints ein, die die Nutzung der zurückgegebenen Daten vereinfachen.

Bei der Beschreibung des Interfaces wird davon ausgegangen, dass der Microservice unter der URL *http://localhost:8080/prologService* zu erreichen ist. Die Fehler, die bei der Nutzung des Interfaces auftreten können, sind getrennt in Abschnitt 3.5 erklärt.

### 3.4.1 Installation und Ausführung

Der erstellte Microservice ist als ausführbare jar-Datei im digitalen Anhang dieser Arbeit [23] verfügbar. Der Microservice benötigt zur Ausführung die PASIPP-Quelldateien und den Arity/Prolog-Interpreter. Diese sind beide ebenfalls im digitalen Anhang dieser Arbeit verfügbar. Vor der ersten Nutzung müssen erst einige Dinge eingerichtet werden:

- Anlegen des Verzeichnisses *C:\PASIPP*.
  - Kopieren der Datei *prologMicroservice.jar* in das Verzeichnis
  - Falls der Microservice zum Umwandeln von HORUS-PNML in standardkonformes PNML benutzt werden soll (siehe Unterabschnitt 3.1.1): Kopieren der Datei *PetriAnalyzer.jar* und des Ordners *petrialyzer-res* nach *C:\PASIPP*.
  - Kopieren der PASIPP-Quelldateien in das Verzeichnis *C:\PASIPP\pasippSources*.
- Einrichten von Arity/Prolog:
  - Arity/Prolog-Dateien in das Verzeichnis *C:\PASIPP\Arity* kopieren.
  - Die folgenden Systemumgebungsvariablen in Windows setzen oder ergänzen:
    - \* Path: *C:\PASIPP\Arity\bin*

- \* lib: C:\PASIPP\Arity\lib
- \* include: C:\PASIPP\Arity\include
- Zu Testzwecken kann *curl*<sup>12</sup> eingerichtet werden:
  - *curl* in das Verzeichnis *C:\PASIPP\curl* kopieren.
  - C:\PASIPP\curl\bin zur Systemumgebungsvariable Path hinzufügen

Wurden diese Schritte ausgeführt, kann die Datei *prologMicroservice.jar* ausgeführt werden. Bei der Ausführung können drei Parameter angegeben werden:

- *-pasipp* oder *-pasippPath* zur Angabe des Ordners, in dem sich die PASIPP-Quelldateien befinden. Wird dieser Parameter ausgelassen, wird der Standardpfad *C:\PASIPP\pasippSources* benutzt.
- *uri* (alternativ *baseUri*, *url* oder *baseUrl*) zur Angabe der Basis-URL, auf der der Microservice verfügbar sein soll. Wird dieser Parameter ausgelassen, wird die Standard-URL *http://localhost:8080/prologService* benutzt.
- *-yEd* zur Ausgabe von Labels in den generierten Erreichbarkeitsbäumen (mehr dazu in Unterabschnitt 3.4.5).

Eine Ausführung des Services in einem Kommandozeilenfenster kann zum Beispiel so aussehen:

```
java -jar prologService.jar -pasipp 'C:\PASIPP\pasippSources' -uri
↪ 'http://localhost:8080/prologService' -yEd
```

In diesem Aufruf können die Argumente *-pasipp* und *-uri* auch weggelassen werden, da lediglich die Standardwerte übergeben werden. Der Aufruf ist also gleichbedeutend mit folgendem Aufruf:

```
java -jar prologService.jar -yEd
```

Um die jar-Datei nutzen zu können, ist es einerseits notwendig, dass die PASIPP-Quelldateien auf dem Rechner verfügbar sind und bei der Ausführung der korrekte Ordner übergeben wird. Andererseits ist auch notwendig, dass Arity/Prolog32 auf dem Rechner installiert ist. Arity/Prolog32 ist als gezippte Distribution auf der Webseite von Peter Gabel [11] verfügbar. Arity/Prolog32 wurde - inklusive der Dokumentation - außerdem im digitalen Anhang dieser Arbeit [23] verfügbar gemacht.

---

<sup>12</sup><https://curl.haxx.se/>



Arity/Prolog32 wurde für Windows geschrieben und ist unter Windows 7 und 10 lauffähig (andere Windows-Versionen wurden im Rahmen der Arbeit nicht getestet; daher kann hierüber keine Aussage getroffen werden). Zur Installation von Arity/Prolog32 muss die zip-Datei in einem beliebigen Ordner entpackt werden. Danach muss der enthaltene Ordner *bin* zur Systemumgebungsvariable *PATH* hinzugefügt werden, der Ordner *lib* zur Systemumgebungsvariable *LIB* und der Ordner *include* zur Systemumgebungsvariable *INCLUDE*.

### 3.4.2 hasCycles

Die Funktion *hasCycles* gibt an, ob das gegebene Petri-Netz frei von Zyklen ist oder nicht. Wird eine Netzdatei in das Programm PASIPP geladen, so gibt dieses automatisch eine Warnmeldung aus, falls das geladene Petri-Netz einen Vorwärtszyklus enthält, und eine getrennte Warnmeldung, falls es einen Rückwärtszyklus enthält. Für den Microservice wurde dies als eigenständige Funktionalität übernommen. Der Microservice überprüft dabei die Ausgabe von PASIPP. Gibt PASIPP eine der Warnungen aus, gibt der Microservice dies entsprechend nach Ausführung zurück. Der Microservice erzeugt bei jedem HTTP Request eine neue Netzdatei und lädt diese in PASIPP, um die von REST geforderte Zustandslosigkeit (siehe Abschnitt 2.7) zu gewähren. Entsprechend erhält der Microservice bei jedem Request die Information, ob ein Netz Zyklen enthält oder nicht. Da es sich hierbei um eine wichtige Information bei der Analyse und Simulation von Petri-Netzen handelt, wurde die Information auch in die Rückgaben der anderen beiden Funktionen eingefügt.

Neben den Warnmeldungen bezüglich des Vorhandenseins von Zyklen gibt PASIPP auch Warnmeldungen aus, falls eine geladene Markierungsdatei gegen Kapazitätsregeln verstößt. Der Microservice arbeitet allerdings ohne Kapazitätsregeln und gibt daher auch keine Kapazitätsregeln an PASIPP weiter. Die entsprechenden Warnmeldungen können somit nicht auftreten und werden deshalb auch nicht abgefangen.

### Interface-Beschreibung

**hasCycles:** Gibt zurück, ob das gegebene Petri-Netz Vorwärts- und/oder Rückwärtszyklen enthält.

### Endpoint

POST <http://localhost:8080/prologService/hascycles>

### Parameter

Parameter können im Request Body als *multipart/form-data* eingefügt werden.

Name	Beschreibung
pnml	Petri-Netz als PNML-Dokument

## Beispiel-Request

```
1 curl -F "pnml=@./pnmlFile.txt" -X POST
   ↪ http://localhost:8080/prologService/hascycles
```

## Response Body

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <hasCyclesResult>
3   <hasForwardCycle>false</hasForwardCycle>
4   <hasBackwardCycle>false</hasBackwardCycle>
5 </hasCyclesResult>
```

Name	Typ	Beschreibung
hasCyclesResult	XML-Objekt	Ergebnis-Objekt
hasCyclesResult. hasBackwardCycle	boolean	Vorhandensein von Vorwärtszyklen
hasCyclesResult. hasForwardCycle	boolean	Vorhandensein von Rückwärtszyklen

### 3.4.3 fireableTransitions

#### Interface-Beschreibung

**fireableTransitions:** Gibt alle Transitionen zurück, die bei der aktuellen Netzmarkierung schalten können.

#### Endpoint

POST <http://localhost:8080/prologService/fireabletransitions>

#### Parameter

Parameter können im Request Body als *multipart/form-data* eingefügt werden.

Name	Beschreibung
pnml	Petri-Netz als PNML-Dokument

## Beispiel-Request

```
1 curl -F "pnml=@./pnmlFile.txt" -X POST
   ↪ http://localhost:8080/prologService/fireabletransitions
```

## Response Body

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <fireableTransitionsResult>
3    <cycles>
4      <hasForwardCycle>false</hasForwardCycle>
5      <hasBackwardCycle>false</hasBackwardCycle>
6    </cycles>
7    <fireableTransitions>
8      <id>transitionA</id>
9      <id>transitionX</id>
10   </fireableTransitions>
11 </fireableTransitionsResult>

```

Name	Typ	Beschreibung
fireableTransitionsResult	XML-Objekt	Ergebnis-Objekt
fireableTransitionsResult.cycles	XML-Objekt	Gibt an, ob Zyklen im Petri-Netz vorhanden sind, siehe <a href="#">Unterabschnitt 3.4.2</a>
fireableTransitionsResult.fireableTransitions	Array	Liste der schaltbaren Transitionen
fireableTransitionsResult.fireableTransitions.id	String	ID einer schaltbaren Transition

### 3.4.4 simulateStep

#### Interface-Beschreibung

**simulateStep:** Simuliert eine einzelne Transitionsschaltung, und gibt das Petri-Netz mit der resultierenden Markierung zurück.

#### Endpoint

POST <http://localhost:8080/prologService/simulatestep>

#### Parameter

Parameter können im Request Body als *multipart/form-data* eingefügt werden.

Name	Beschreibung
pnml	Petri-Netz als PNML-Dokument
id	ID der zu schaltenden Transition; wird keine ID angegeben, wird die Transition geschaltet, die von <i>fireableTransitions</i> (Unterabschnitt 3.4.3) an erster Stelle zurückgegeben wird.

## Beispiel-Request

```
1 curl -F "pnml=@./pnmlFile.txt" -F "id=transitionX" -X POST
   ↪ http://localhost:8080/prologService/simulatestep
```

## Response Body

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <simulateStepResult>
3   <cycles>
4     <hasForwardCycle>false</hasForwardCycle>
5     <hasBackwardCycle>false</hasBackwardCycle>
6   </cycles>
7   <resultingPetriNet>
8     <pnml xmlns="http://www.pnml.org/version-2009/grammar/pnml">
9       [PNML-Petri-Netz, siehe zum Beispiel Codeausschnitt 9]
10    </pnml>
11  </resultingPetriNet>
12 </simulateStepResult>
```

Name	Typ	Beschreibung
simulateStepResult	XML-Objekt	Ergebnis-Objekt
simulateStepResult.cycles	XML-Objekt	Gibt an, ob Zyklen im Petri-Netz vorhanden sind, siehe Unterabschnitt 3.4.2
simulateStepResult.resultingPetriNet	PNML-Dokument	Petri-Netz mit aktualisierter Markierung, in PNML-Notation (siehe zum Beispiel Codeausschnitt 9)

Ist man lediglich an dem resultierenden PNML-Netz interessiert, kann man statt *simulatestep* auch den Endpoint *simulatesteppnml* verwenden, siehe Unterabschnitt 3.4.6.

### 3.4.5 reachabilityTree

#### Interface-Beschreibung

**reachabilitytree:** Erzeugt einen Erreichbarkeitsbaum für ein gegebenes Netz inklusive gegebener Startmarkierung und gibt den Baum zurück.

## Endpoint

POST `http://localhost:8080/prologService/reachabilitytree`

## Parameter

Parameter können im Request Body als *multipart/form-data* eingefügt werden.

Name	Beschreibung
pnml	Petri-Netz als PNML-Dokument

## Beispiel-Request

```
1 curl -F "pnml=@./pnmlFile.txt" -X POST
   ↪ http://localhost:8080/prologService/reachabilitytree
```

## Response Body

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <reachabilityTreeResult>
3   <cycles>
4     <hasForwardCycle>false</hasForwardCycle>
5     <hasBackwardCycle>false</hasBackwardCycle>
6   </cycles>
7   <Tree>
8     <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
9       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
10      xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
11        http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
12       [GraphML-Datenstruktur, siehe zum Beispiel Codeausschnitt 17]
13     </graphml>
14   </Tree>
15 </reachabilityTreeResult>
```

Name	Typ	Beschreibung
reachabilityTreeResult	XML-Objekt	Ergebnis-Objekt
reachabilityTreeResult.cycles	XML-Objekt	Gibt an, ob Zyklen im Petri-Netz vorhanden sind, siehe Unterabschnitt 3.4.2
reachabilityTreeResult.Tree	XML-Objekt	Enthält den Erreichbarkeitsbaum in GraphML-Notation, siehe unten.

Ist man lediglich an dem resultierenden GraphML-Baum interessiert, kann man statt *reachabilitytree* auch den Endpoint *reachabilitytreegraphml* verwenden, siehe Unterabschnitt 3.4.6.

## Erreichbarkeitsbaum

Der Microservice gibt Erreichbarkeitsbäume in der Beschreibungssprache GraphML (siehe [4]) aus. Dies ist eine standardisierte Sprache zur Beschreibung von Graphen. Dadurch ist es einfacher, die Ergebnisse in anderen Programmen weiter zu verarbeiten.

Ein Beispiel für einen solchen Graphen ist in Codeausschnitt 17 zu sehen. Er repräsentiert den Erreichbarkeitsbaum, der in Abbildung 17 graphisch dargestellt ist. Die in Abbildung 17 verwendete Notation entspricht der PASIPP-Notation für Erreichbarkeitsbäume (siehe Unterabschnitt 2.4.2). Zuerst werden mehrere *keys* definiert (Zeile 6 bis 8). Diese geben die zusätzlichen Attribute an, die bestimmte GraphML-Elemente enthalten können. Dies sind *originalMarking*, das angibt ob eine Netzmarkierung die ursprüngliche Markierung ist und *placeId* und *count* zum Angeben der Markierungen einzelner Stellen des Netzes.

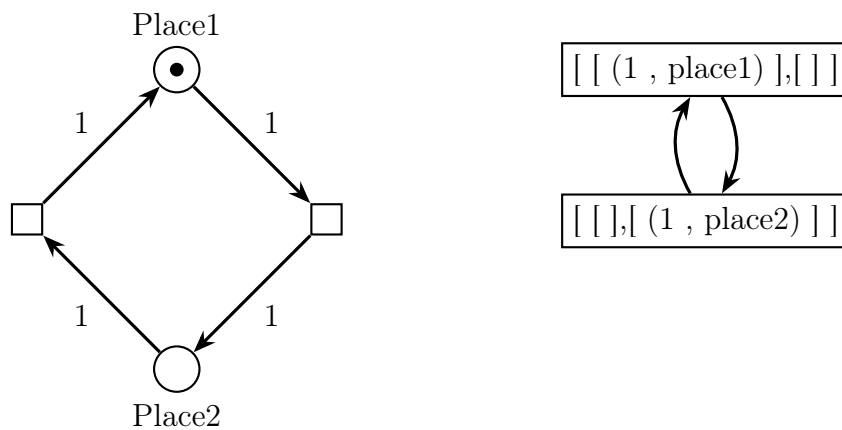


Abbildung 17: Beispiel-Petri-Netz (links) mit entsprechendem Erreichbarkeitsbaum (rechts)

Der Graph besteht aus Knoten (*nodes*), die jeweils eine Netzmarkierung darstellen, und Kanten (*edges*), die diese Markierungen verbinden. Eine Kante bedeutet dabei, dass man durch Schalten einer Transition vom Ursprungsknoten der Kante zu dessen Zielknoten gelangen kann. Jede Netzmarkierung besteht aus einer Menge von Stellenmarkierungen. Diese werden jeweils in einem Untergraphen des Netzmarkierungsknotens definiert (im Beispiel Zeile 12 bis 16 und 19 bis 23). Der Untergraph enthält für jede Stelle, die in der aktuellen Netzmarkierung über Tokens verfügt, einen Knoten. Dieser definiert die ID dieser Stelle (*placeId*) und die Anzahl ihrer Tokens (*count*) (Zeile 13 bis 15 und 20 bis 22).

Abbildung 4 auf Seite 20 zeigt das gleiche Petri-Netz wie in Abbildung 17, allerdings mit geänderter Markenausgabe in der linken Transition. Dadurch können beide Stellen beliebig viele Marken ansammeln. PASIPP erkennt diese Situation und gibt entsprechend den Wert *infini* im Erreichbarkeitsbaum aus, der auf der rechten Seite der Abbildung zu

```

1 <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
4     http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
5
6   <key id="originalMarking" for="node" attr.name="originalMarking"
7     ↪ attr.type="boolean"><default>false</default></key>
8   <key id="placeId" for="node" attr.name="placeId"
9     ↪ attr.type="string"/>
10  <key id="count" for="node" attr.name="count" attr.type="string"/>
11
12  <graph id="G" edgedefault="directed">
13    <node id="netMarking1">
14      <graph id="netMarking1:" edgedefault="directed">
15        <node id="netMarking1::1">
16          <data key="placeId">place2</data> <data
17            ↪ key="count">1</data>
18        </node>
19      </graph>
20    </node>
21    <node id="netMarking2"> <data key="originalMarking">true</data>
22      <graph id="netMarking2:" edgedefault="directed">
23        <node id="netMarking2::1">
24          <data key="placeId">place1</data> <data
25            ↪ key="count">1</data>
26        </node>
27      </graph>
28    </node>
29    <edge source="netMarking1" target="netMarking2"/>
30    <edge source="netMarking2" target="netMarking1"/>
31  </graph>
32</graphml>

```

Codeausschnitt 17: GraphML-Erreichbarkeitsbaum für Abbildung 17

sehen ist. Der Microservice gibt in der GraphML-Ausgabe für solche Markierungen den String *infinity* als Markenanzahl aus.

Da GraphML eine standardisierte Sprache ist, gibt es auch Graphvisualisierungsprogramme, mit denen man sich die erzeugten Erreichbarkeitsbäume anzeigen lassen kann. Beispiele hierfür sind yEd<sup>13</sup>. Abbildung 18 zeigt einen Screenshot von yEd für einen Erreichbarkeitsbaum, der von dem Microservice erzeugt wurde. Das zugrunde liegende Petri-Netz ist dem Beispiel aus Abbildung 2 auf Seite 16 entnommen.

Der GraphML-Standard definiert nur die Struktur von Graphen, nicht allerdings deren Darstellung. Darstellungsinformationen werden von jedem Programm anders gespeichert. Um die Datenstruktur möglichst allgemein zu halten, gibt der Microservice daher nur die Struktur des Graphen zurück. Die im Beispiel sichtbare hierarchische Ordnung der Knoten wurde von yEd selbst erzeugt.

Es ist allerdings möglich, den Microservice Label-Angaben für das Programm yEd erzeugen zu lassen. Die Angabe dieser Label kann beim Ausführen des Programms mit der Option *-yEd* aktiviert werden. Wird diese Option nicht angegeben, werden die Labels standardmäßig nicht erzeugt. Abbildung 19 zeigt den gleichen Graphen wie Abbildung 18, allerdings mit der Angabe von Labels. Durch die Angabe von Labels ist es für Betrachter deutlich einfacher, den Erreichbarkeitsbaum zu verstehen.

Zur Anzeige der Labels sind zusätzliche Informationen nötig. Codeausschnitt 18 zeigt einen Netzmarkierungsknoten ohne Label (Zeile 3 bis 12) und den gleichen Knoten mit Label (Zeile 14 bis 33). Zeilen 17 bis 21 und 25 bis 29 enthalten die zusätzlichen graphischen Informationen. Außerdem ist es nötig, einen zusätzlichen *key* zu definieren (Zeile 1).

### 3.4.6 Komprimierte Rückgabe

Die Funktionen *simulateStep* und *getReachabilityTree* geben jeweils Daten im PNML-beziehungsweise GraphML-Format aus. Zur weiteren Verarbeitung dieser Daten kann es nützlich sein, diese in einer eigenen Datei verfügbar zu haben. So lassen sich zum Beispiel *.graphml*-Dateien direkt mit einem externen Tool wie zum Beispiel yEd<sup>14</sup> anzeigen. Außerdem kann das Vorhandensein der Daten in einer Datei auch die maschinelle Weiterverarbeitung vereinfachen.

Die beiden genannten Methoden geben allerdings nicht nur die entsprechenden PNML-beziehungsweise GraphML-Daten aus, sondern noch zusätzliche Informationen (siehe Unterabschnitt 3.4.4 und 3.4.5). Um die Nutzung der GraphML- und PNML-Daten zu ver-

---

<sup>13</sup><https://www.yworks.com/products/yed>

<sup>14</sup><https://www.yworks.com/products/yed>



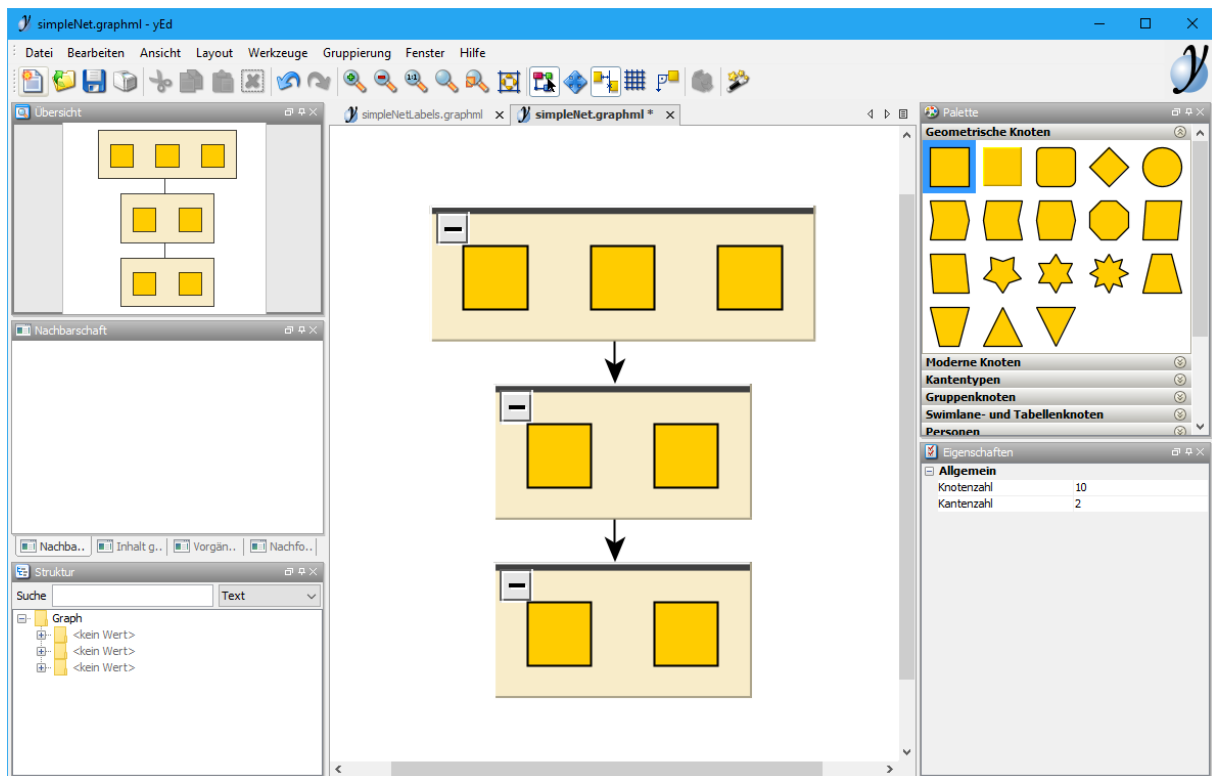


Abbildung 18: Screenshot des Programms yEd mit dem Erreichbarkeitsbaum für Abbildung 2 (Seite 16), ohne Label

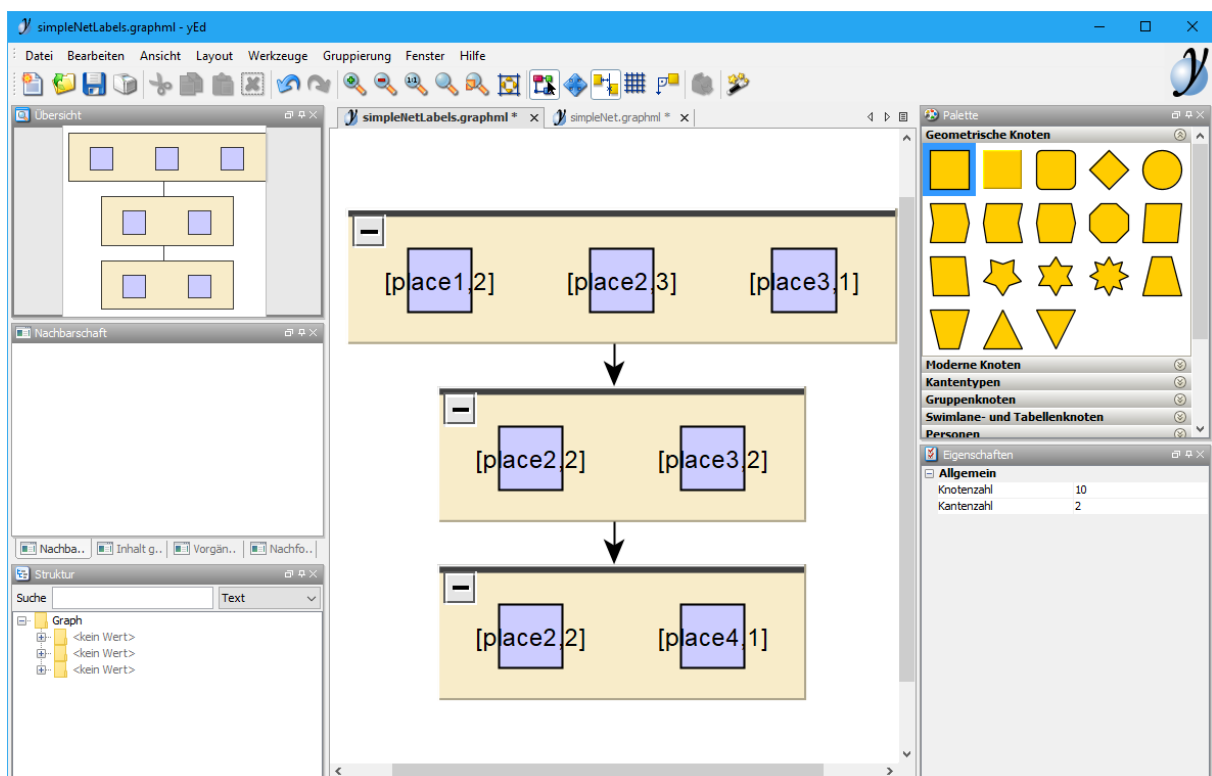


Abbildung 19: Screenshot des Programms yEd mit dem Erreichbarkeitsbaum für Abbildung 2 (Seite 16), mit Label

```

1  <key for="node" id="d0" yfiles.type="nodegraphics"/>
2
3  <node id="netMarking1">
4    <graph id="netMarking1:" edgedefault="directed">
5      <node id="netMarking1::1">
6        <data key="placeId">place2</data> <data key="count">2</data>
7      </node>
8      <node id="netMarking1::2">
9        <data key="placeId">place3</data> <data key="count">2</data>
10     </node>
11   </graph>
12 </node>
13
14 <node id="netMarking1">
15   <graph id="netMarking1:" edgedefault="directed">
16     <node id="netMarking1::1">
17       <data key="d0">
18         <y:ShapeNode>
19           <y:NodeLabel>[place2,2]</y:NodeLabel>
20         </y:ShapeNode>
21       </data>
22       <data key="placeId">place2</data> <data key="count">2</data>
23     </node>
24     <node id="netMarking1::2">
25       <data key="d0">
26         <y:ShapeNode>
27           <y:NodeLabel>[place3,2]</y:NodeLabel>
28         </y:ShapeNode>
29       </data>
30       <data key="placeId">place3</data> <data key="count">2</data>
31     </node>
32   </graph>
33 </node>

```

Codeausschnitt 18: Vergleich eines Netzmarkierungsknotens mit und ohne Labelinformation

einfachen wurden zwei zusätzliche Endpoints eingeführt: *simulatesteppnml* und *reachabilitytreegraphml*. Diese haben das gleiche Interface wie die Methoden *simulatestep* und *reachabilitytree*, geben allerdings ausschließlich die PNML- beziehungsweise GraphML-Daten aus, ohne zusätzliche Informationen. Somit lässt sich die Antwort des Microservices für diese Endpoints ohne weitere Bearbeitung direkt in einer Datei abspeichern. Diese Datei kann durch einen Anwender, externe Tools oder aufrufende Clients weiterverwendet werden. Ein entsprechender *curl*-Aufruf könnte wie folgt aussehen:

```
1 curl -F pnml="@./petriNet.pnml" -X POST
   ↪ http://localhost:8080/prologService/reachabilitytreegraphml >>
   ↪ reachTree.graphml
```

Dieser Aufruf speichert den zurückgegebenen Erreichbarkeitsbaum in der Datei *reachTree.graphml*, die dann weiterverwendet werden kann.

### 3.5 Fehlerbehandlung

Beim Start des Microservices wird zuerst überprüft ob sich an dem übergebenen Pfad die benötigten PASIPP-Quelldateien befinden. Wird kein Pfad als Parameter angegeben, wird stattdessen der Standardpfad *C:\PASIPP\pasippSources* darauf überprüft. Werden die benötigten Quelldateien nicht gefunden, so wird die Fehlermeldung *ERROR: Indicated path does not contain PASIPP sources (or default path does not contain PASIPP sources, if no path was given)* ausgegeben, und das Java-Programm wird beendet.

Tritt während der Bearbeitung eines Client-Requests ein Fehler auf, wird dieser abgefangen, und es wird statt eines Simulations- oder Analyseergebnisses eine Fehlermeldung an den Client zurückgesendet. Es gibt dabei folgende Fehlermeldungen:

- *Encountered Invalid ID exception; petri net type might be missing*: Innerhalb des PNML Frameworks ist eine *InvalidIDException* aufgetreten. Dies wurde höchstwahrscheinlich dadurch verursacht, dass die Definition des Petri-Netztyps in der PNML-Datei ausgelassen wurde.
- *Could not convert PNML string / file to a PNML object (ImportException)*: Fehler bei der Erstellung eines PNML-Objektes aus einem PNML-String oder einer PNML-Datei. Dies kann daran liegen, dass ein PNML-Petri-Netztyp angegeben wurde, der dem PNML Framework nicht bekannt ist (also insbesondere nicht existente PNML-Petri-Netztypen), oder an einem Fehler innerhalb der PNML-Daten.
- *Wrong Petri Net type; must be a PNML Place/Transition Net*: Es wurde ein Petri-Netztyp angegeben, der von dem Microservice nicht unterstützt wird. Der Microser-

vice erwartet PNML Place-/Transition-Netze (vom Typ <http://www.pnml.org/version-2009/grammar/ptnet>).

- *XML / PNML parsing error*: Die übergebene XML- oder PNML-Datei enthält einen Fehler und kann deshalb nicht geparsed werden.
- *Server-Side IO error*: Auf dem Server ist ein Fehler beim Lesen / Schreiben von Dateien aufgetreten. Dies kann an schreibgeschützten Dateien / Ordnern oder an mangelnden Rechten des Java-Prozesses liegen.
- *The Prolog process exceeded its available ressources*: Der Prolog-Prozess konnte aufgrund von zu wenig Ressourcen nicht zu Ende arbeiten. Diese Meldung kann bei der Generierung von Erreichbarkeitsbäumen auftreten.
- *Unknown error* Es ist ein unbekannter / unerwarteter Fehler aufgetreten.

Für den Fall, dass eine inkorrekte PNML-Datei an den Service gesendet wurde, ist in Codeausschnitt 19 eine beispielhafte Rückgabe des Services zu sehen. Für die anderen Fehler würde zwischen den *error*-Tags die entsprechende Fehlermeldung stehen.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <error>
3     XML / PNML parsing error
4 </error>
```

Codeausschnitt 19: Fehlerrückgabe

## 4 Implementierung und mögliche Erweiterungen

Wie in Kapitel 3 konzipiert, wurde der Microservice als Java-Programm basierend auf dem Jersey Framework umgesetzt. Dieses Kapitel geht auf Probleme und Besonderheiten ein, die sich während der Implementierung ergaben. Abschnitt 4.1 geht auf Änderungen ein, die an dem originalen PASIPP-Programm in der Version 2.3 vom April 1991 vorgenommen wurden und warum diese vorgenommen wurden. Abschnitt 4.2 geht auf eine sogenannte Wettlaufsituation (engl. *race condition*, [17]) ein, die sich bei der Benutzung des Services ergeben kann. Abschnitt 4.3 stellt mögliche zukünftige Erweiterungen des Microservices vor.

### 4.1 Anpassungen an PASIPP

Zur Nutzung der PASIPP-Quelldateien wurden zwei kleine Änderungen am Code des Programms vorgenommen. Diese erleichtern das Senden von Eingaben an das Programm und das Lesen der Ausgaben des Programms; sie verändern allerdings nicht den Aufbau von Ein- und Ausgabedaten, und auch nicht die Geschäftslogik von PASIPP.

Die erste Änderung betrifft die Regel *read\_string* in der Datei *SPECIAL.ARI*. Die Änderung ist in Codeausschnitt 20 zu sehen. In der ursprünglichen Regel wurde erst mit *read\_string(40, List)* die User-Eingabe in eine Liste gelesen, die dann mit *atom\_string(String, List)* in einen String umgewandelt wurde. Das erste Argument der *read\_string*-Regel gab dabei die *maximale* Anzahl Buchstaben an, die Prolog einlesen sollte. Wurde vor dieser Anzahl die Enter-Taste gedrückt (bzw. ein `\n` gesendet bei Maschinen- statt User-Input), wurde die Eingabe nur bis dahin eingelesen. Das unter aktuellen Windows-Versionen lauffähige Arity/Prolog32 hat allerdings eine andere Implementierung von *read\_string*. Das erste Argument gibt dabei *genau* an, wie viele Buchstaben eingelesen werden sollen. Wird vorher die Enter-Taste gedrückt, wird trotzdem weitergelesen. *read\_string* wurde deshalb durch *read\_line* ersetzt, das immer genau eine Eingabezeile einliest. Der erste Parameter von *read\_string* gibt dabei an, von wo der String gelesen werden soll. Die Zahl 0 entspricht dem Standard-Eingabestream, also der Kommandozeile (*read\_string* hat diesen Parameter auch; im Unterschied zu *read\_line* kann er bei *read\_string* allerdings ausgelassen werden, wenn man den Standard-Eingabestream benutzen will).

```
1 -read_string(String) :- read_string(40,List),atom_string(String,List).
2 +read_string(String) :- read_line(0,String).
```

Codeausschnitt 20: Änderung der Methode *read\_string* (Datei *SPECIAL.ARI*)

Die zweite Änderung betrifft die Regel *ausgabe\_one* in der Datei *HELP1.ARI*, zu sehen in Codeausschnitt 21. Das Programm PASIPP wurde ursprünglich für DOS-Betriebssysteme geschrieben, die über keine Scrolling-Funktionalität verfügten. Daher war es nötig, die Länge von Ausgaben anzupassen an den auf dem Bildschirm verfügbaren Platz. Dafür wurde die Regel *ausgabe\_one* geschrieben, die immer 10 Elemente einer Liste ausgibt und dann auf eine Bestätigung des Nutzers wartet, um die Ausgabe fortzusetzen.

*ausgabe\_one* nimmt dabei zwei Argumente entgegen: eine Liste und eine Zählervariable. Ist die Liste leer, wird die erste Regeldefinition ausgeführt, und *Ende der Ausgabe* wird ausgegeben.

Ist die Liste nicht leer, und die Zählervariable entspricht 10, wird die zweite Definition der Regel ausgeführt. Dabei wird zuerst die Regel *weiter* aufgerufen, ebenfalls zu sehen in Codeausschnitt 21. Sie liest einen String ein und speichert diesen in einer anonymen Variablen. Das Einlesen eines Strings wird durch die Enter-Taste abgeschlossen. Da die anonyme Variable nach der Regelausführung nicht mehr zur Verfügung steht, besteht die Funktionalität der Regel darin, das Programm so lange anzuhalten, bis der Benutzer die Enter-Taste drückt. Danach wird das nächste Element der Liste ausgegeben, und *ausgabe\_one* wird rekursiv mit dem Rest der Liste und dem Wert 1 für die Zählervariable aufgerufen.

In allen anderen Fällen - also wenn die Liste nicht leer ist und die Zählervariable einen anderen Wert als 10 hat - wird das nächste Listenelement ausgegeben, und *ausgabe\_one* wird rekursiv mit dem Rest der Liste und einer um 1 erhöhten Zählervariable aufgerufen (dritte Regeldefinition).

Da die Beschränkung der Ausgabe nicht mehr nötig war, wurde der Aufruf von *weiter* entfernt. Dies resultiert darin, dass immer alle Listenelemente zusammen ausgegeben werden, was die programmatische Nutzung von PASIPP erleichtert.

## 4.2 Wettlaufsituation

Um Petri-Netze und Petri-Netzmarkierungen in PASIPP zu nutzen, müssen diese in Prolog-Notation in einer Datei gespeichert werden und danach in das Programm geladen werden. Der entwickelte Microservice läuft auf einem Server, der Anfragen von Clients erhält. Um die von REST geforderte Zustandslosigkeit (siehe Abschnitt 2.7) zu gewähren, werden diese Dateien bei jedem Client-Request aus der Anfrage heraus generiert und in PASIPP geladen. Erhält der Server zwei zeitnahe Anfragen, kann es sein, dass er zuerst versucht, die Dateien beider Anfragen zu generieren und zu speichern und danach in das Programm zu laden. Verwendet man für die Namen dieser Dateien fest gecodierte Werte, kann es dazu kommen, dass eine Datei zweimal beschrieben wird, bevor sie in PASIPP

```
1 weiter :-
2     nl,write('                Weiter mit <Return> '),nl,
3     read_string(_),nl,nl.
4
5 [...]
6
7 ausgabe_one([],_) :- nl,write('Ende der Ausgabe!'),nl,nl.
8
9 ausgabe_one([X|Tail],10) :-
10    -weiter,
11    writeq(X),nl,
12    ausgabe_one(Tail,1),!.
13
14 ausgabe_one([X|Tail],ZeilenNr) :-
15     Neue_Nr is ZeilenNr + 1,
16     writeq(X), nl,
17     ausgabe_one(Tail,Neue_Nr).
```

Codeausschnitt 21: Die Regel *weiter* und die Änderung der Regel *ausgabe\_one* (beide in der Datei HELP1.ARI)

geladen wird. Dadurch können falsche Netze oder Netzmarkierungen geladen werden, was zu falschen Ergebnissen führt. Es handelt sich dabei um eine *Wettlaufsituation* (engl. *race condition*; vgl. [17]). Um dies zu umgehen, werden die Dateien abhängig von der aktuellen Systemzeit benannt. Dabei wird die Zahl der Nanosekunden der aktuellen Zeit an den Dateinamen angehängt, siehe Codeausschnitt 22. Nachdem die Anfrage beantwortet wurde, werden die dafür generierten Dateien wieder gelöscht, da sich ansonsten bei jeder Anfrage neue Dateien im Dateisystem ansammeln würden.

```

1 Long time = System.nanoTime();
2 netFile = new File(pasippFolder + "/plNet" + time + ".net");
3 markingFile = new File(pasippFolder + "/plMark" + time + ".dat");

```

Codeausschnitt 22: Zeitabhängige Benennung der PASIPP-Eingabedateien (PnmlToPrologConverter.java)

### 4.3 Möglichkeiten zur zukünftigen Erweiterung des Services

Der Microservice ist so geschrieben, dass zusätzliche Funktionalitäten von PASIPP relativ einfach integriert werden können, und andere Ausgabeformate (zum Beispiel JSON) einfach hinzugefügt werden können. Dazu sind Änderungen sowohl in der Simulator-Komponente als auch in der Server-Komponente notwendig (siehe Abschnitt 3.1). Erweiterungen der Simulator-Komponente werden in Unterabschnitt 4.3.1 vorgestellt, Erweiterungen der Server-Komponente in Unterabschnitt 4.3.2.

#### 4.3.1 Erweiterungen der Simulator-Komponente

Die Simulator-Komponente ist der Teil des Microservices, der PASIPP ausführt, mit dem Prolog-Interpreter interagiert und Prolog-Eingabe- und Ausgabedaten erzeugt beziehungsweise verarbeitet. Um eine zusätzliche Funktion von PASIPP in den Microservice zu integrieren, sollte daher zuerst diese Komponente erweitert werden. Dies erfordert eine neue öffentliche Methode in der Klasse *Simulator.java* zur Bereitstellung der Funktionalität, und eventuell neue Hilfsklassen, zum Beispiel zur Rückgabe von komplexen Simulations- oder Analyseergebnissen. Die hinzugefügte Methode kann der Struktur der anderen öffentlichen Methoden von *Simulator.java* folgen.

Die bisherigen öffentlichen Methoden sind jeweils drei Mal mit unterschiedlichen Parametern definiert. Nur jeweils eine dieser Methoden enthält die eigentliche Funktionalität, die anderen wandeln lediglich Parameter um und rufen dann eine der anderen beiden Metho-



den auf (siehe Unterabschnitt 3.1.1). Dies macht es einfacher, das Interface zu benutzen. Für neue Funktionalitäten kann der gleiche Ansatz genutzt werden.

Die prinzipielle Struktur der *Simulator*-Methoden, die die eigentliche Funktionalität erbringen, ist in Codeausschnitt 23 zu sehen. Alle methodenspezifischen Teile wurden dabei entfernt oder vereinfacht, sodass nur der allgemeine Rahmen übrigbleibt. Am Anfang der Funktion werden zwei Prolog-Dateien erzeugt (Zeile 10): *netFile*, die die Netzstruktur enthält, und *markingFile*, die die Netzmarkierung enthält.

PASIPP gibt, sobald eine Netzdatei geladen wird, aus, ob das Netz Zyklen enthält. Entsprechend gibt die Methode *createAndLoadPrologFiles* ein *CycleObject* aus, das diese Information enthält.

Darauf werden mit *callPrologCommand()* Eingaben an den Prolog-Interpreter geschickt (wie in Zeile 14 und 17 zu sehen). *callPrologCommand()* kann Regeln aufrufen oder, wenn eine aufgerufene Regel auf User-Input wartet, User-Input an diese Regel senden. Allgemein benötigte Prolog-Regelaufrufe werden dabei schon von der Methode *initPasipp* ausgeführt, die ihrerseits von *createAndLoadPrologFiles* aufgerufen wird. Diese allgemein benötigten Regelaufrufe sind das Laden der PASIPP-Quelldateien und das Initialisieren von PASIPP.

Über *getPrologResponse* kann die Ausgabe des Prolog-Prozesses erhalten werden (Zeile 20). Dies kann auch mehrfach geschehen, zum Beispiel um abhängig von der letzten Ausgabe zu entscheiden, welche Prolog-Regel als nächstes ausgeführt werden soll. *getPrologResponse* gibt dabei immer die gesamte Ausgabe seit dem letzten Aufruf von *getPrologResponse* zurück.

Der von *getPrologResponse* zurückgegebene String muss daraufhin verarbeitet werden, was im Beispiel-Code in der Methode *processResponse* erledigt wird (Zeile 22). Bei komplexen Ergebnissen kann es sich lohnen, eine eigene Klasse für die Rückgabe der Ergebnisse zu erstellen (im Beispiel *MyResultClass* (Zeile 22)).

Am Ende der Methode sollte - auch wenn während der Ausführung Fehler aufgetreten sind - der Prolog-Prozess beendet werden. Um dies zu garantieren, wird *close* in einem *finally*-Block aufgerufen (Zeile 30).

#### 4.3.2 Erweiterungen der Server-Komponente

Die Server-Komponente ist der Teil des Programms, der auf HTTP Requests reagiert und Analyse- beziehungsweise Simulationsergebnisse zurückgibt. Dazu ruft die Komponente intern die Funktionen der Simulator-Komponente auf, wandelt deren Rückgabe in ein standardisiertes Format um, und gibt dies dann zurück.

```

1  private File netFile;
2  private File markingFile;
3
4  [...]
5
6  public MyResultClass newPasippFunctionality(PetriNet pnmlNet
   ↪  /*möglicherweise mehr Parameter*/) throws IOException {
7      try {
8          // Erzeugt die Netzdatei und die Markierungsdatei und hinterlegt sie
9          // in netFile und markingFile.
10         CycleObject cycles = createAndLoadPrologFiles(pnmlNet);
11         // Hier folgt der methodenspezifische Code.
12         // Mit "callPrologCommand()" können Befehle an den Prolog Interpreter
13         // gesendet werden.
14         callPrologCommand("myPrologCommand.");
15         // Über "netFile" und "markingFile" kann auf die Prolog-Dateien
16         // verwiesen werden.
17         callPrologCommand("doSomethingWithTheNetFile(" + netFile.getName() +
   ↪         ").");
18         // Erhalte die Ausgabe des Prolog Interpreters
19         // (seit dem letzten Aufruf von "getPrologResponse").
20         String prologResponse = getPrologResponse();
21         // Verarbeiten der Prolog-Ausgabe.
22         MyResultClass result = processResponse(prologResponse);
23         // Rückgabe des Ergebnisses.
24         return result;
25     } finally {
26         // createAndLoadPrologFiles() startet den Prolog-Interpreter-Prozess;
27         // createAndLoadPrologFiles() kann dabei eine IOException werfen.
28         // close() muss in einem finally-Block aufgerufen werden,
29         // um sicherzustellen, dass der Interpreter-Prozess beendet wird.
30         close();
31     }
32 }

```

Codeausschnitt 23: Struktur der öffentlichen Methoden der Klasse *Simulator.java* (mit erklärenden Kommentaren)

Um neue Funktionalitäten über das REST-Interface anzubieten, beziehungsweise um die vorhandene Funktionalität in einem anderen Format zurückgeben zu können, sind daher Änderungen an der Server-Komponente notwendig. Dieser Abschnitt gibt einen Überblick, wie diese Erweiterungen umgesetzt werden können.

### Hinzufügen eines neuen Ausgabeformats

Das REST-Interface besteht aus mehreren *Endpoints*, die jeweils eine Funktionalität nach außen hin anbieten. Jeder dieser Endpoints wird von einer eigenen Java-Klasse umgesetzt, deren prinzipieller Aufbau in Codeausschnitt 24 zu sehen ist. Dabei wurden klassenspezifische Details vereinfacht oder weggelassen, sodass nur die allgemeine Struktur zu sehen ist.

Die bisherigen Klassen verfügen alle über eine Funktion *getReturnString* (Zeile 30), die eine Liste von Parametern entgegennimmt und den String erzeugt, der im HTTP Response Body zurückgegeben wird. Diese Methode macht momentan nichts anderes, als die Methode *buildXMLString* mit genau den gleichen Parametern aufzurufen. Will man nach außen hin ein anderes Rückgabeformat - zum Beispiel JSON - anbieten, muss daher die Methode *getReturnString* geändert werden. Am einfachsten wäre es in diesem Fall, eine Methode *getJSONString* hinzuzufügen und in *getReturnString* entweder immer diese Methode aufzurufen oder anhand der HTTP-Request-Parameter zu entscheiden, welche der beiden Methoden aufgerufen wird.

Darüber hinaus können Änderungen an einzelnen Klassen der *Simulator*-Komponente sinnvoll sein. So bietet zum Beispiel die Klasse *Tree* die Methode *String toGraphML()* an, die das *Tree*-Objekt in einen GraphML-String überführt. Hier könnte eine zusätzliche Methode zum Überführen in eine JSON-Struktur sinnvoll sein.

### Erweiterung des REST-Interfaces

Um nach außen hin eine neue Funktionalität anzubieten, empfiehlt es sich, diese zuerst in der Simulator-Komponente umzusetzen (siehe Unterabschnitt 4.3.1). In der Server-Komponente muss daraufhin eine neue Klasse hinzugefügt werden, deren prinzipieller Aufbau dem in Codeausschnitt 24 folgen kann.

Die Klasse muss einen Pfad definieren (Zeile 1), der an die Basis-URL angehängt wird und an der die neue Funktionalität angeboten werden wird. Die Klasse *SimulatorRequest* enthält einige allgemeine nützliche Funktionen (*getSimulator*, *prettifyXML* und Rückgabe von Fehlermeldungen) und kann als Elternklasse der neuen Klasse dienen (Zeile 2).

Die *main*-Methode übergibt den Pfad des PASIPP-Ordners mithilfe eines Konfigurationsobjekts an alle Ressourcen-Klassen. Um den Pfad nutzen zu können, ist es nötig, ein *Configuration*-Objekt<sup>15</sup> als Kontext der Klasse zu definieren (Zeile 3, 4).

---

<sup>15</sup>ein Objekt der Klasse `javax.ws.rs.core.Configuration`

```

1 fire([transitionsname,Inputvariablen,Outputvariablen]) :-
2     entferne(anzahl[1],stellename[1](marke[1])),
3     ...
4     entferne(anzahl[i],stellename[i](marke[i])),
5     transitionsbeschriftung,
6     einfuege(anzahl[i+1],stellename[i+1](marke[i+1])),
7     ...
8     einfuege(anzahl[n],stellename[n](marke[n])),

```

Codeausschnitt 24: Struktur der HTTP Ressource-Klassen (mit erklärenden Kommentaren)

Die eigentliche Methode ist in Zeile 6 bis 28 definiert. Sie definiert eine HTTP-Request-Methode (zum Beispiel *@POST*, Zeile 6) und einen Rückgabewert (*@Produces*, Zeile 7) und nimmt ein Objekt der Klasse *FormDataMultiPart* als Parameter entgegen.

*FormDataMultiPart* enthält die Daten des Request-Bodys. Über *getField* können dessen Werte erhalten werden (Zeile 11).

Mit *getSimulator(config)* kann ein *Simulator*-Objekt erhalten werden (Zeile 14), der den PASIPP-Ordner aus *config* benutzt. Die Benutzung des Simulators kann Fehler verursachen, weshalb sie in einem *try-catch*-Block ausgeführt wird (Zeile 16 bis 27). Tritt ein Fehler auf, wird dieser von der Oberklasse *SimulatorRequest* behandelt (Zeile 26). Dies verhindert die Duplizierung von Code, da viele Fehler in mehreren Endpoints auftreten können. Die Rückgabe der Ergebnisse (Zeile 23) wurde weiter oben in diesem Abschnitt bereits beschrieben.

## 5 Evaluierung

Um den erstellten Microservice zu evaluieren, wurden die einzelnen Funktionen mit einem Geschäftsprozessmodell getestet, das mit dem HORUS Business Modeler<sup>16</sup> erstellt wurde. Auf diese Weise kann sowohl die Funktionalität des Microservices selbst getestet werden als auch die Anbindung an den externen Service, der zur Umwandlung von HORUS-PNML in standardkonformes PNML genutzt wird (siehe Unterabschnitt 3.1.1). Der Microservice wurde dabei mit Java 8 und 11, den beiden aktuellsten Langzeitsupportversionen von Java, getestet.

Das Beispielmmodell, mit dem die Funktionen getestet werden, ist in Abbildung 20 zu sehen. Es handelt sich dabei um ein leicht abgewandeltes Beispiel-Ablaufmodell aus dem HORUS Business Modeler. Um den Umgang mit Zyklen in Petri-Netzen zu evaluieren, wurde das Netz in Abbildung 21 genutzt, das bis auf eine zusätzliche Transition dem zuvor genannten Netz entspricht.

In diesem Abschnitt werden die vier Funktionen des Microservices anhand der genannten Beispielnetze evaluiert. Abschnitt 5.1 diskutiert die Funktion *hasCycles*, Abschnitt 5.2 behandelt die Funktionen *fireableTransitions* und *simulateStep* und Abschnitt 5.3 geht auf die Funktion *reachabilityTree* ein.

Bei den Beispielen ist darauf zu achten, dass HORUS für erstellte Transitionen und Stellen (Places) automatisch eine ID erzeugt. Diese entspricht nicht den vom Benutzer vergebenen Namen der Transitionen beziehungsweise Stellen. Der erstellte Microservice arbeitet mit den genannten IDs und gibt diese auch in seinen Ergebnissen zurück. Die Zuweisung von IDs und Namen für die Netze aus Abbildung 20 und 21 sind in Tabelle 1 und 2 zu sehen.

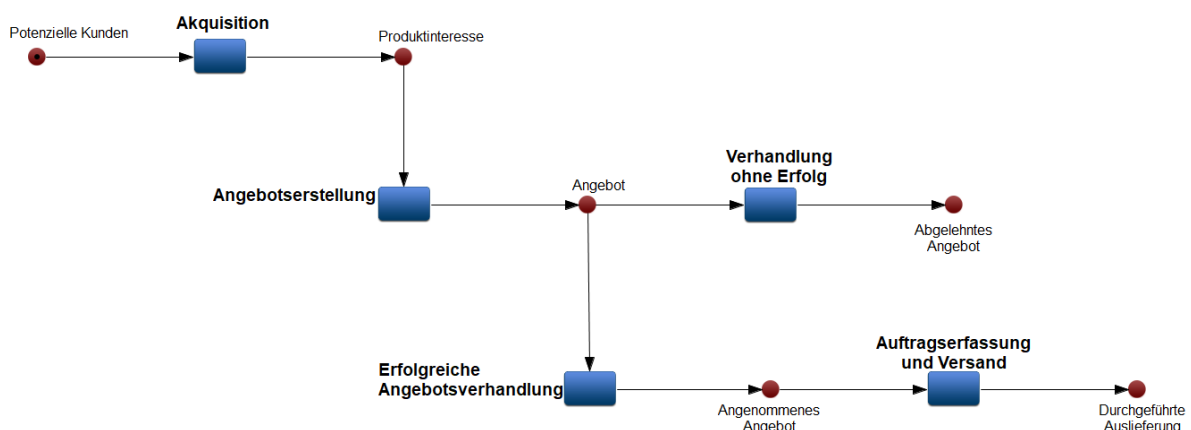


Abbildung 20: Beispiel-Petri-Netz (Bild aus dem HORUS Business Modeler)

<sup>16</sup><https://www.horus.biz/de/downloads/>

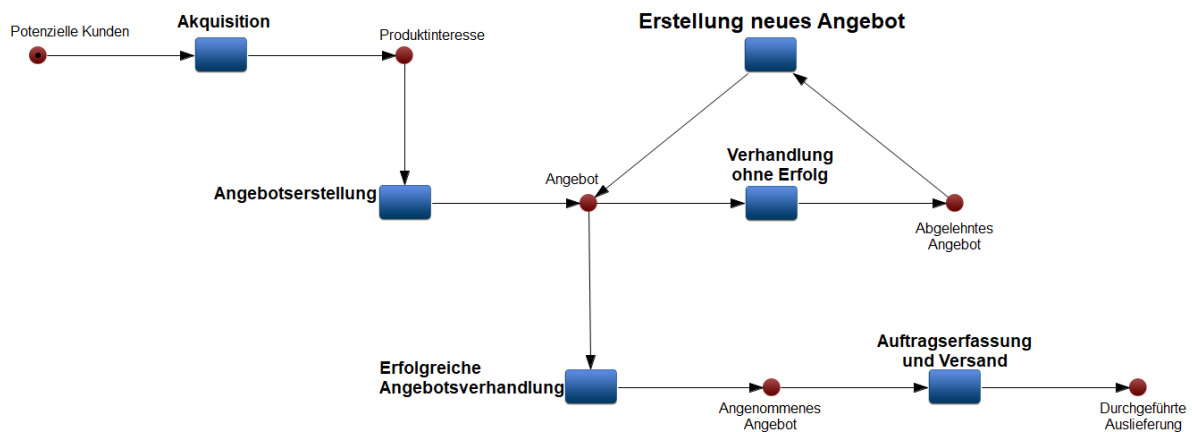


Abbildung 21: Beispiel-Petri-Netz mit Zyklus (Bild aus dem HORUS Business Modeler)

Stellenname	Stellen-ID
Produktinteresse	p18842731
Abgelehntes Angebot	p29930036
Angenommenes Angebot	p18291126
Durchgeführte Auslieferung	p3746855
Angebot	p29668729
Potenzielle Kunden	p29244518

Tabelle 1: Zuordnung von Namen und IDs der Stellen für die Netze in Abbildung 20 und 21

## 5.1 hasCycles

Diese Funktion gibt zurück, ob die Struktur eines gegebenen Netzes mindestens einen Zyklus enthält oder nicht. Ist das Petri-Netz zyklensfrei, kann das Vorkommen von Schleifen während der Simulation des Netzes beziehungsweise während der Ausführung des Geschäftsprozesses ausgeschlossen werden.

Für das Petri-Netz aus Abbildung 20 gibt die Funktion zurück, dass das Netz zyklensfrei ist (siehe Codeausschnitt 25); für das Netz aus Abbildung 21 gibt sie zurück, dass das Netz Zyklen enthält (siehe Codeausschnitt 26).

## 5.2 fireableTransitions und simulateStep

Die Funktion *fireableTransitions* gibt für ein gegebenes Petri-Netz alle Transitionen zurück, die unter der aktuellen Markierung schalten können. Für das Petri-Netz und die Markierung aus Abbildung 20 ist dies die Transition *Akquisition* mit der ID *t11819817*. Codeausschnitt 27 zeigt den Aufruf der Funktion *fireableTransitions* für das genannte Petri-Netz und die Rückgabe des Microservices.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <hasCyclesResult>
3   <hasForwardCycle>false</hasForwardCycle>
4   <hasBackwardCycle>false</hasBackwardCycle>
5 </hasCyclesResult>
```

Codeausschnitt 25: Rückgabe der Funktion *hasCycles* für das Petri-Netz aus Abbildung 20

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <hasCyclesResult>
3   <hasForwardCycle>true</hasForwardCycle>
4   <hasBackwardCycle>true</hasBackwardCycle>
5 </hasCyclesResult>
```

Codeausschnitt 26: Rückgabe der Funktion *hasCycles* für das Petri-Netz aus Abbildung 21

```
1 C:\PASIPP\pnmlFiles>curl -F pnml="@./Akquisition2Auslieferung.pnml" -X
   ↪ POST http://localhost:8080/prologService/fireabletransitions
2 <fireableTransitionsResult>
3   <cycles>
4     <hasForwardCycle>false</hasForwardCycle>
5     <hasBackwardCycle>false</hasBackwardCycle>
6   </cycles>
7   <fireableTransitions>
8     <id>t11819817</id>
9   </fireableTransitions>
10 </fireableTransitionsResult>
```

Codeausschnitt 27: Rückgabe der Funktion *fireableTransitions* für das Petri-Netz aus Abbildung 20

Transitionsname	Transition-ID
Akquisition	t11819817
Erfolgreiche Angebotsverhandlung	t24157276
Auftragserfassung und Versand	t8477064
Verhandlung ohne Erfolg	t10984042
Angebotserstellung	t23469803
Erstellung neues Angebot	t14563918

Tabelle 2: Zuordnung von Namen und IDs der Transitionen für die Netze in Abbildung 20 und 21

Der Inhalt dieser Rückgabe lässt sich nutzen, um die Funktion *simulateStep* auszuführen. *simulateStep* schaltet eine einzelne Transition und gibt das resultierende Petri-Netz zurück. Die Funktion nimmt dafür ein Petri-Netz in PNML-Notation und eine Transitions-ID als Parameter entgegen (wird die ID ausgelassen, wird einfach die erste gefundene, schaltbare Transition geschaltet). Mögliche Transitions-IDs für *simulateStep* lassen sich erhalten, indem zuerst *fireableTransitions* aufgerufen wird. Für das Beispiel aus Abbildung 20 lässt sich die Transition *Akquisition* (ID *t11819817*) schalten, indem folgender Request an den Microservice gesendet wird:

```

1 C:\PASIPP\pnmlFiles>curl -F pnml="@./Akquisition2Auslieferung.pnml"
  ↪ id="t11819817" -X POST
  ↪ http://localhost:8080/prologService/simulatesteppnml >
  ↪ resultingNet1.pnml

```

Dieser *curl*-Befehl schaltet die Transition *Akquisition* und speichert das resultierende Petri-Netz in der Datei *resultingNet1.pnml*. Diese Datei kann in folgenden Requests an den Microservice direkt weiterverwendet werden, zum Beispiel um die jetzt möglichen Transitionsschaltungen zu erhalten. Ruft man *fireableTransitions* erneut für *resultingNet1.pnml* auf, wird die ID *t23469803* zurückgegeben, die der Transition *Angebotserstellung* entspricht.

Durch die Kombination der Funktionen *fireableTransitions* und *simulateStep* lässt sich durch den Client ein *Token Game* für das Netz umsetzen. Ein Token Game ist eine Simulation, bei der der User eine zu schaltende Transition auswählen kann. Die Transition wird daraufhin geschaltet, die resultierende Markierung des Netzes wird angezeigt, und der User kann die nächste zu schaltende Transition auswählen. Ein beispielhafter Ablauf an *curl*-Requests für ein Token Game ist in Codeausschnitt 28 zu sehen. Für eine bessere Lesbarkeit wurden die Rückgaben von *fireableTransitions* auf die IDs gekürzt, und die Terminal-Ausgabe für die Datei-Schreibprozesse wurden ausgelassen. Während des Token



Games werden nacheinander die Transitionen *Akquisition*, *Angebotserstellung*, *Erfolgreiche Angebotsverhandlung*, und zuletzt *Auftragserfassung und Versand* geschaltet. Danach können keine weitere Transitionen schalten. Im Beispiel wird jedes Petri-Netz, das durch den Microservice nach einer Schaltung zurückgegeben wird, in einer separaten Datei gespeichert. Falls die Zwischenergebnisse später nicht mehr benötigt werden, kann dafür allerdings auch die gleiche Datei verwendet werden.

### 5.3 reachabilityTree

Die Funktion *reachabilityTree* erstellt für ein gegebenes Petri-Netz einen Erreichbarkeitsbaum im GraphML-Format. Ein Erreichbarkeitsbaum enthält alle erreichbaren Netzmarkierungen. Jede Netzmarkierung besteht aus einem Untergraphen, der jeweils alle Stellenmarkierungen anzeigt (alle Stellen mit mehr als 0 Marken). Die erstellten Erreichbarkeitsbäume können mit externen Tools, zum Beispiel yEd<sup>17</sup>, angezeigt werden.

Abbildung 22 zeigt den Erreichbarkeitsbaum für das Petri-Netz aus Abbildung 20. Abbildung 22a zeigt den Erreichbarkeitsbaum mit den generierten IDs des HORUS Business Modeler; Abbildung 22b zeigt, wie der Erreichbarkeitsbaum aussehen würde, wenn statt der IDs die entsprechenden Namen aus Tabelle 1 und 2 verwendet werden. Die Erreichbarkeitsbäume können einen Überblick über alle erreichbaren Netzmarkierungen geben und darüber, wie weit eine bestimmte Markierung von einer anderen Markierung, zum Beispiel der Startmarkierung, entfernt ist. Auch erhält man mit den Erreichbarkeitsbäumen schnell eine Vorstellung über die Anzahl der von der Startmarkierung aus möglichen Netzmarkierungen. Außerdem lassen sich an den Bäumen leicht Zyklen innerhalb des Petri-Netzes erkennen. Ein Beispiel für einen Erreichbarkeitsbaum mit Zyklus ist Abbildung 23, die den Erreichbarkeitsbaum für das Netz aus Abbildung 21 zeigt.

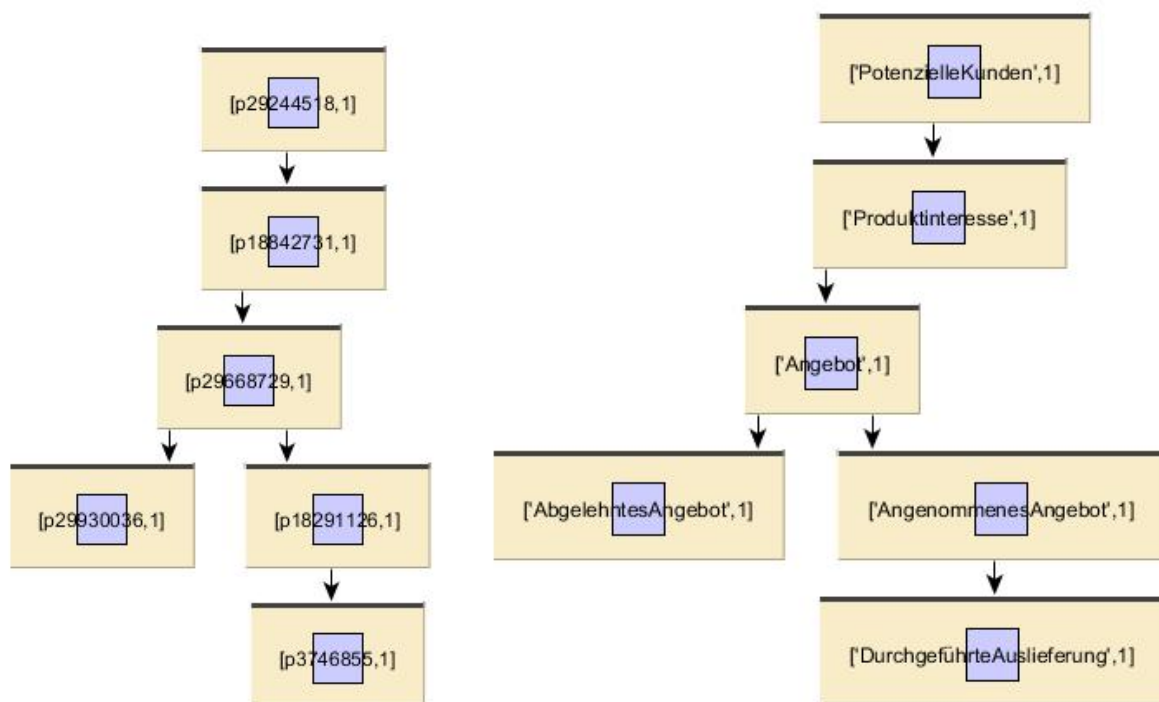
Die graphische Darstellung der Erreichbarkeitsbäume kann allerdings schnell unübersichtlich werden, wie an Abbildung 24 und 25 zu sehen ist. Diese stellen den Erreichbarkeitsbaum für Abbildung 20 dar, wobei die Stelle *Potenzielle Kunden* in der Startmarkierung nicht einen Token, sondern zwei beziehungsweise drei Tokens enthält.

---

<sup>17</sup><https://www.yworks.com/products/yed>

```
1 C:\PASIPP\pnmlFiles>curl -F pnml="@./Akquisition2Auslieferung.pnml" -X
  ↪ POST http://localhost:8080/prologService/fireabletransitions
2 ... <id>t11819817</id>...
3 C:\PASIPP\pnmlFiles>curl -F pnml="@./Akquisition2Auslieferung.pnml"
  ↪ id="t11819817" -X POST
  ↪ http://localhost:8080/prologService/simulatesteppnml >
  ↪ resultingNet1.pnml
4 C:\PASIPP\pnmlFiles>curl -F pnml="@./resultingNet1.pnml" -X POST
  ↪ http://localhost:8080/prologService/fireabletransitions
5 ... <id>t23469803</id>...
6 C:\PASIPP\pnmlFiles>curl -F pnml="@./resultingNet1.pnml" id="t23469803"
  ↪ -X POST http://localhost:8080/prologService/simulatesteppnml >
  ↪ resultingNet2.pnml
7 C:\PASIPP\pnmlFiles>curl -F pnml="@./resultingNet2.pnml" -X POST
  ↪ http://localhost:8080/prologService/fireabletransitions
8 ... <id>t10984042</id>
9   <id>t24157276</id>...
10 C:\PASIPP\pnmlFiles>curl -F pnml="@./resultingNet2.pnml" id="t24157276"
  ↪ -X POST http://localhost:8080/prologService/simulatesteppnml >
  ↪ resultingNet3.pnml
11 C:\PASIPP\pnmlFiles>curl -F pnml="@./resultingNet3.pnml" -X POST
  ↪ http://localhost:8080/prologService/fireabletransitions
12 ... <id>t8477064</id>...
13 C:\PASIPP\pnmlFiles>curl -F pnml="@./resultingNet3.pnml" id="t8477064" -X
  ↪ POST http://localhost:8080/prologService/simulatesteppnml >
  ↪ resultingNet4.pnml
14 C:\PASIPP\pnmlFiles>curl -F pnml="@./resultingNet4.pnml" -X POST
  ↪ http://localhost:8080/prologService/fireabletransitions
15 (keine schaltbaren Transitionen mehr)
```

Codeausschnitt 28: Token Game für das Petri-Netz aus Abbildung 20.



(a) Erreichbarkeitsbaum mit den ge- (b) Erreichbarkeitsbaum mit lesbaren Labels (den Namen  
nerierten IDs des HORUS Business der Stellen, mit entfernten Leerzeichen)  
Modeler

Abbildung 22: Erreichbarkeitsbaum für das Petri-Netz aus Abbildung 20

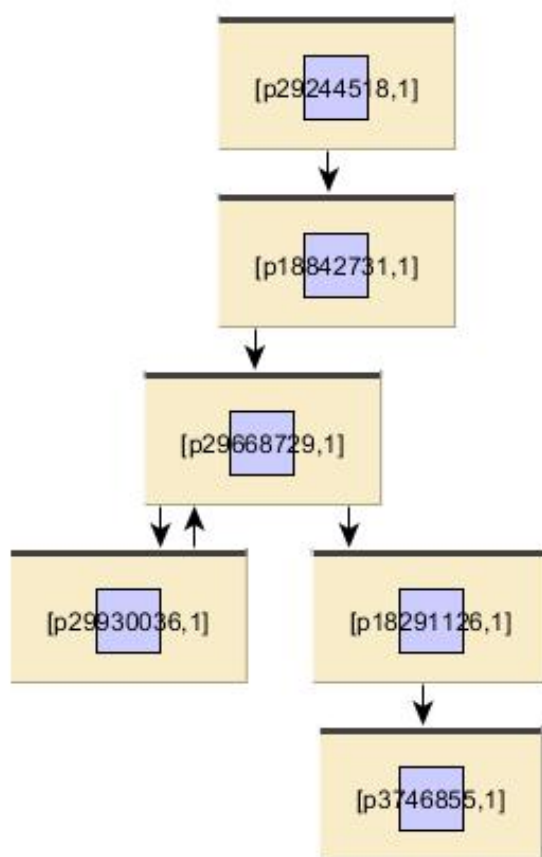


Abbildung 23: Erreichbarkeitsbaum für das Petri-Netz aus Abbildung 21

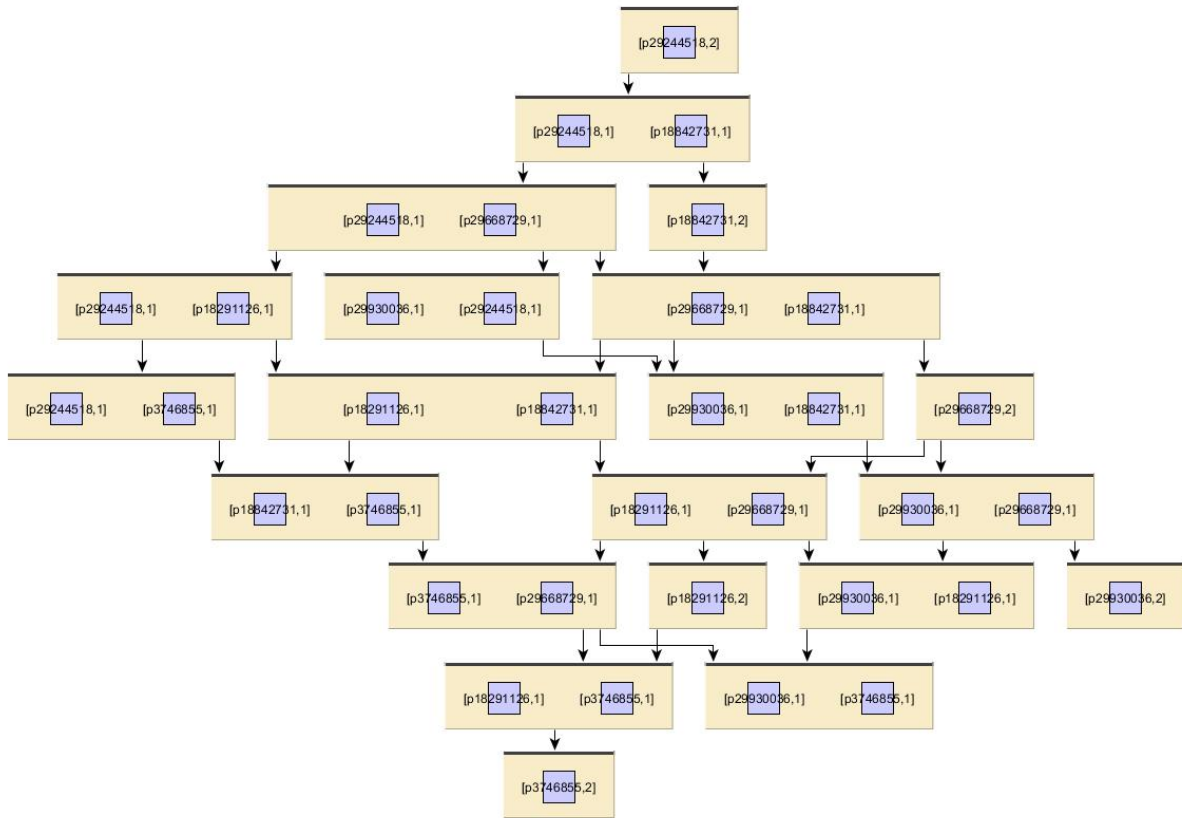


Abbildung 24: Erreichbarkeitsbaum für das Petri-Netz aus Abbildung 20, mit anfänglich 2 Tokens in der Stelle "Potenzielle Kunden"

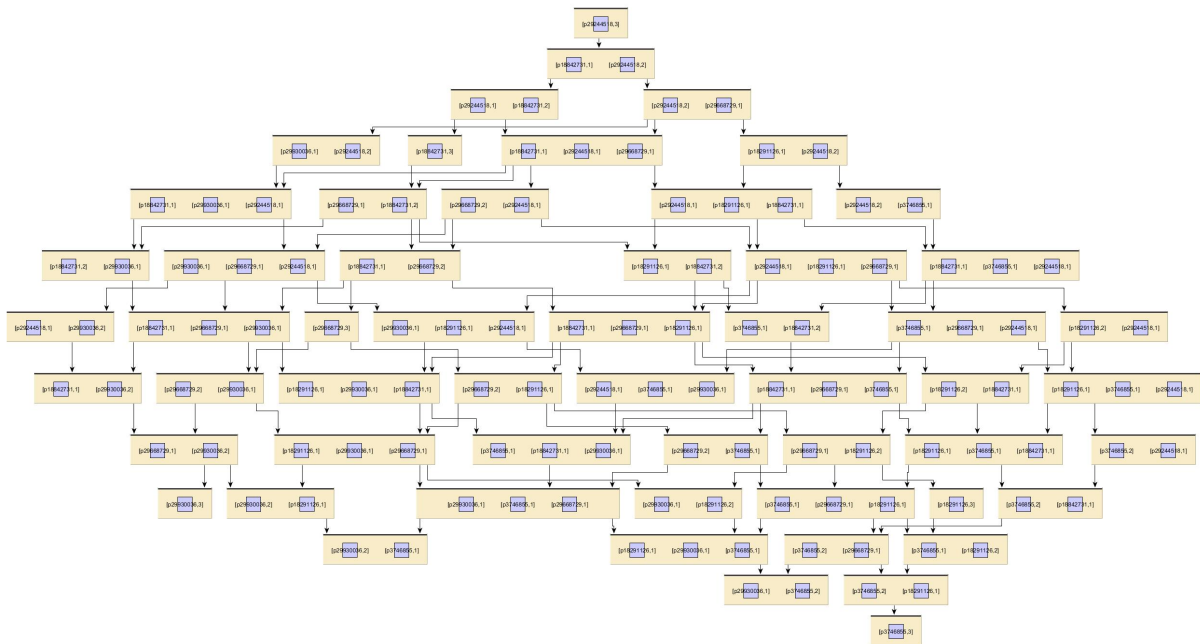


Abbildung 25: Erreichbarkeitsbaum für das Petri-Netz aus Abbildung 20, mit anfänglich 3 Tokens in der Stelle "Potenzielle Kunden"



## 6 Fazit und Ausblick

Das Microservice-Architekturmuster findet heutzutage eine immer weitere Verbreitung innerhalb der Softwareentwicklung. Eine der möglichen positiven Eigenschaften des Architekturmusters ist, dass bestehende Software einfacher in neue Projekte integriert werden kann. Dazu kann zum Beispiel ein Microservice geschrieben werden, der als Wrapper für die bestehende Software dient und deren Funktionalität in einem feingranularen Interface nach außen hin anbietet.

Dadurch können die einzelnen Funktionen des bestehenden Programms gezielt genutzt werden, ohne dass die nutzenden Services die Implementierungsdetails des Programms kennen müssen. Auch kann der Wrapperservice in möglichen Folgeprojekten verwendet werden, da Microservices grundsätzlich unabhängig von dem Rest der Software ausgeführt werden. Versucht man ein altes, großes monolithisches Softwaresystem mit einem einzelnen solchen Wrapper in modernen Architekturen wieder verfügbar zu machen, besteht jedoch die Gefahr, dass der resultierende Microservice ebenso groß und unhandlich wird wie das alte System, was gegen die Microservicephilosophie verstößt. Eine mögliche Lösung für dieses Problem ist es, mehrere Microservices zu schreiben, die unterschiedliche Funktionalitäten des alten Systems nach außen hin anbieten. Diese nutzen zwar intern das gleiche Altsystem, sind aber unabhängig voneinander ausführbar und wartbar.

Ziel dieser Arbeit war es, anhand eines konkreten Beispiels die Integration einer bestehenden Software in eine vorgeschlagene Microservice-Architektur zu ermöglichen und so die Machbarkeit dieses Ansatzes zu demonstrieren. Bei der bestehenden Software handelt es sich um PASIPP, ein Prolog-Programm zur Analyse und Simulation von Petri-Netzen, in der Version 2.3 aus dem Jahr 1991.

Der erstellte Microservice dient als Wrapper für PASIPP und bietet mehrere Funktionalitäten des Programms in einem feingranularen Interface nach außen hin an. Als Eingabeformat erwartet der Microservice Petri-Netze im PNML-Format und er gibt Ergebnisse in XML zurück. Bei den Petri-Netzen kann es sich sowohl um standardkonforme Place-/Transition-Netze handeln, als auch um Netze, die mit dem Horus Business Modeler erstellt wurden<sup>18</sup>. Bei Eingabe von Petri-Netzen im HORUS-PNML-Format wird ein weiterer Microservice genutzt, der solche Modelle in Standard-PNML-Format übertragen kann. Dies demonstriert die prinzipielle, einfache Erweiterbarkeit des Ansatzes durch weitere Microservices.

Während PASIPP eine Vielzahl an Simulations- und Analysefunktionen anbietet, hat der Microservice nur vier davon umgesetzt. Eine Idee von Microservices ist die Unabhängigkeit der Eingabe- und Ausgabedaten von ihrer internen Repräsentation und damit verbunden

---

<sup>18</sup><https://www.horus.biz/de/downloads/>

die Möglichkeit, unterschiedliche Formate an einen Client anzubieten. Der vorliegende Microservice ist aber auf PNML und XML als Ein- und Ausgabeformate festgelegt.

Mögliche zukünftige Entwicklungen sind dementsprechend die Erweiterung des Microservices um weitere Funktionen des Programms PASIPP und um weitere Ein- und Ausgabeformate. Eine mögliche Anwendung des Microservices ist die Erweiterung des HORUS Business Modeler um Funktionen des Microservices. Der HORUS Business Modeler soll dabei um eine Funktion zur Generierung von Erreichbarkeitsbäumen erweitert werden. Diese Funktion soll intern den Microservice benutzen und den erhaltenen Erreichbarkeitsbaum dem Benutzer graphisch anzeigen. Idealerweise werden dem Benutzer dabei nicht nur die Eltern-Kind-Beziehungen der Netzmarkierungen angezeigt, sondern auch die Transition, die jeweils schalten muss, um vom Eltern- zum Kindknoten zu gelangen. Dazu wäre die Erweiterung der Erreichbarkeitsbäume um Schaltungsinformationen nötig. Auch wünschenswert wäre die Angabe von Stellennamen in Erreichbarkeitsbäumen anstatt oder zusätzlich zu der Angabe von IDs, was in den beiden Erreichbarkeitsbäumen in Abbildung 22 in Abschnitt 5.3 zu sehen ist. Die Möglichkeit, Petri-Netze mit Teilnetzen (zum Beispiel Transitionen in einem Geschäftsprozess, die selbst einem eigenen Teilprozess entsprechen) zu simulieren und zu analysieren, ist eine weitere Funktion, die zukünftig entwickelt werden könnte.



## Literatur

- [1] Abbott, Martin L. und Michael T. Fisher: *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*, Kapitel Splitting Applications for Scale, Seite 340. Addison-Wesley Professional, 1. Auflage, 2009.
- [2] Alpers, Sascha, Christoph Becker, Andreas Oberweis und Thomas Schuster: *Microservice Based Tool Support for Business Process Modelling*. In: *2015 IEEE 19th International Enterprise Distributed Object Computing Workshop*, Seiten 71–78, Sept 2015.
- [3] Blackburn, Patrick, Johan Bos und Kristina Striegnitz: *Learn Prolog Now!*, Band 7 der Reihe *Texts in Computing*. College Publications, 2006.
- [4] Brandes, Ulrik, Markus Eiglsperger, Jürgen Lerner und Christian Pich: *Graph Markup Language (GraphML)*. In: Tamassia, Roberto (Herausgeber): *Handbook of graph drawing visualization*, Discrete mathematics and its applications, Seiten 517–541. CRC Press, Boca Raton [u.a.], 2013.
- [5] Chen, Lianping: *Microservices: Architecting for Continuous Delivery and DevOps*. In: *2018 IEEE International Conference on Software Architecture (ICSA)*, Seiten 39–46, April 2018.
- [6] Colmerauer, Alain und Philippe Roussel: *History of Programming languages—II*. Kapitel The Birth of Prolog, Seiten 331–367. ACM, New York, NY, USA, 1996.
- [7] Conway, Melvin E.: *How Do Committees Invent?* *Datamation*, 14(4):28–31, April 1968. [http://www.melconway.com/Home/Conways\\_Law.html](http://www.melconway.com/Home/Conways_Law.html).
- [8] Fielding, Roy Thomas: *Architectural Styles and the Design of Network-based Software Architectures*. Dissertation, University of California, Irvine, Januar 2000.
- [9] Fowler, Martin und James Lewis: *Microservices*. <https://martinfowler.com/articles/microservices.html>. Zuletzt zugegriffen: 18.05.2018.
- [10] Francis, Sabine und Andreas Gerstlauer: *A reactive and adaptive data flow model for network-of-system specification*. *IEEE Embedded Systems Letters*, 9(4):121–124, Dec 2017.
- [11] Gabel, Peter: *Arity/Prolog32*. <http://peter-gabel.com/content/arityprolog32>. Zuletzt zugegriffen: 28.06.2018.
- [12] Hee, Kees M van, Natalia Sidorova und Jan Martijn van der Werf: *Business Process Modeling Using Petri Nets*. In: *Transactions on Petri Nets and Other Models of Concurrency VII*, Seiten 116–161. Springer, 2013.

- 
- [13] Langlois, Richard N.: *Modularity in technology and organization*. Journal of Economic Behavior and Organization, 49(1):19 – 37, 2002.
- [14] Lee, Kent D.: *Programming Languages*, Kapitel Formal Semantics, Seite 227. Springer US, Boston, MA, 2008.
- [15] Oberweis, Andreas: *Integritätsbewahrendes Prototyping von verteilten Systemen*. In: Paul, Manfred (Herausgeber): *GI — 19. Jahrestagung I*, Seiten 215–230, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.
- [16] Oberweis, Andreas, Juergen Seib und Georg Lausen: *PASIPP: Ein Hilfsmittel zur Analyse und Simulation von Prolog-beschrifteten Praedikate/Transitionen-Netzen*. Wirtschaftsinformatik, 33:219 – 230, 1991.
- [17] Praun, Christoph von: *Race conditions*. In: Padua, David (Herausgeber): *Encyclopedia of Parallel Computing*, Seiten 1691–1697, Boston, MA, 2011. Springer US.
- [18] Richardson, Chris: *Microservices: Decomposing Applications for Deployability and Scalability*. <https://www.infoq.com/articles/microservices-intro>. Zuletzt zugegriffen: 19.05.2018.
- [19] Sarkar, Santonu, Shubha Ramachandran, G. Sathish Kumar, Madhu K. Iyengar, K. Rangarajan und Saravanan Sivagnanam: *Modularization of a Large-Scale Business Application: A Case Study*. IEEE Software, 26(2):28–35, March 2009.
- [20] Schwickert, Axel C. und Kim Fischer: *Der Geschäftsprozeß als formaler Prozeß: Definition, Eigenschaften und Arten*. In: *Arbeitspapiere WI*, Band 4. Lehrstuhl für Allg. BWL und Wirtschaftsinformatik, Johannes Gutenberg-Universität, Mainz, 1996.
- [21] Snelting, Gregor und Oliver Hummel: *Übung zur Vorlesung Programmierparadigmen - Blatt 8 - Beispiellösung*. <https://pp.info.uni-karlsruhe.de/lehre/WS201314/paradigmen/uebung/>, 2013. Zuletzt zugegriffen: 05.09.2018; Downloads sind über das VPN des Karlsruher Instituts für Technologie möglich (<https://vpn.kit.edu>).
- [22] Stephens, Rod: *Beginning Software Engineering*, Seite 94. Wrox Press Ltd., Birmingham, UK, UK, 1. Auflage, 2015.
- [23] Stolz, Fabian: *PasippMicroservice*. Digitaler Anhang zu dieser Arbeit. <https://github.com/fstolz/PasippMicroservice>. Zuletzt zugegriffen: 31.10.2018.
- [24] Theelen, Bart D., Marc C. W. Geilen, Twan Basten, Jeroen P. M. Voeten, S. Valentin Gheorghita und Sander Stuijk: *A scenario-aware data flow model for combined long-run average and worst-case performance analysis*. In: *Fourth ACM and IEEE In-*

- ternational Conference on Formal Methods and Models for Co-Design, 2006. MEM-OCODE '06. Proceedings.*, Seiten 185–194, July 2006.
- [25] Ullrich, Meike: *PetriAnalyzer*. <https://github.com/cgrossde/PetriAnalyzer>. Zuletzt zugegriffen: 05.10.2018.
- [26] Victor, Frank und Gerd Woetzel: *Ein Prolog-Simulator für Prädikat/Transitions-Netze*. In: *Simulationstechnik : 6. Symposium [Simulationstechnik] in Wien, September 1990; Tagungsband*, Band 1 der Reihe *Fortschritte in der Simulationstechnik*, Seiten 284–289. Vieweg, Wiesbaden, 1990.
- [27] Viegas Damásio, Carlos, Anastasia Analyti, Grigoris Antoniou und Gerd Wagner: *Open and Closed World Reasoning in the Semantic Web*. In: *Proceedings of IPMU*, Seiten 1850–1857, 2006.
- [28] Weber, Michael und Ekkart Kindler: *The petri net markup language*. In: Ehrig, Hartmut, Wolfgang Reisig, Grzegorz Rozenberg und Herbert Weber (Herausgeber): *Petri Net Technology for Communication-Based Systems: Advances in Petri Nets*, Seiten 124–144. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.
- [29] Zimmermann, Olaf: *Microservices tenets*. *Computer Science - Research and Development*, 32(3):301–310, Jul 2017.