UNIVERSITÄT
BAYREUTH

DATENBANKEN UND
INFORMATIONSSYSTEME

**Universität Bayreuth**
**Lehrstuhl für Angewandte Informatik IV**
Datenbanken und Informationssysteme

# ESProNa

## Eine Constraintsprache zur multimodalen Prozessmodellierung und navigationsgestützten Ausführung

Dissertation zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von

**Dipl.-Inform. Michael Igler**

Bayreuth - 2011

**Erstgutachter:** *Prof. Dr.-Ing. Stefan Jablonski, Universität Bayreuth*
**Zweitgutachter:** *Prof. Dr. Marc Erich Latoschik, Universität Würzburg*

**Einreichung:** *03.11.2011*
**Kolloquium:** *31.07.2012*

*Verfasst unter der Betreuung von/Thesis written under supervision (advisor) of*

***Prof. Dr.-Ing. Stefan Jablonski***

*Professor am Lehrstuhl für Angewandte Informatik IV,*
*Universität Bayreuth*

*Verfasst unter der Begutachtung von/Thesis written under supervision (co-advisor) of*

***Prof. Dr. Paulo Moura***

*Assistant Professor at the Dep. of Computer Science,*
*University of Beira Interior, Portugal*

*„Damit das Mögliche entsteht, muss immer wieder das Unmögliche versucht werden.“*
Hermann Hesse

*„To achieve the possible, you must try to achieve the impossible.“*
Hermann Hesse

## Zusammenfassung

Deklarative Prozessmodellierungssprachen erfreuen sich aufgrund ihrer Ausdrucksstärke und der kompakten Prozessmodelle einer immer größer werdenden Beliebtheit. Ziel dieser neuen Art der Modellierung ist es, Geschäftsprozesse einfacher und effizienter aufnehmen zu können. Ein bekanntes Konzept aus den deklarativen Programmiersprachen, die strikte Trennung zwischen Problemstellung und Lösung, wird auf den Bereich der Prozessmodellierung übertragen. Somit wird eine Vereinfachung der zu modellierenden Geschäftsprozesse erreicht. Um die Prozesse in ihrer Gesamtheit zu erfassen, wird das Konzept der perspektivenorientierte Prozessmodellierung (POPM) verwendet. Weiterhin werden neben den Anforderungen an eine Prozessmodellierungssprache zusätzliche Konzepte erarbeitet, die für eine effiziente Modellierung von Geschäftsprozessen sinnvoll sind. Die im ersten Kapitel der Arbeit angesprochenen Probleme aktueller Prozessmodellierungssprachen werden in den nachfolgenden Kapiteln aufgegriffen und gelöst. Neue Forschungsergebnisse, wie etwa die entwickelte Prozessnavigation zur navigationsgestützten Ausführung der erstellten Geschäftsprozesse oder das Modellieren von subjektiven Empfehlungen, werden ebenfalls behandelt. Durch letzteres Konzept kann das empirische Verhalten der Geschäftsprozesse modelliert und zum Zeitpunkt der Ausführung präsentiert werden. Es wurden nicht nur die Konzeptionen und Lösungen der Problemstellungen erarbeitet, sondern auch gezeigt, wie diese implementiert und verwendet werden können. Alle Ergebnisse der vorliegenden Arbeit sind in der deklarativen Prozessmodellierungssprache **ESP**ro**N**a umgesetzt.

## Abstract

Declarative process modeling languages enjoy an increasing popularity in research through its expressiveness and brief process models. One of the goals of this new technique is to simplify the layout of business processes. As a result of that style, based on the separation of concerns between problem and solution, a simplification of the desired process models can be achieved. The ultimate goal of our research is the accurate and consistent representation of business processes. This is achieved by using a Perspective-Oriented Process Modeling (POPM) technique. The exploration presents concepts which recognize the previously mentioned goals. Beside the elaborated requirements on a process modeling language, different additional concepts are developed which are useful for efficient modeling of business processes. Within the first chapter complications are mentioned that are discussed and solved in succeeding chapters. The newly developed process navigation has proven to support proper execution of business processes. By using the concept of modeling recommendations the empirical behavior of business processes can be modeled and presented at runtime. Concepts and solutions have proven to show a beneficial use in achieving business processes. **ESP**ro**N**a is the business process language that has come from this study.

# Inhaltsverzeichnis

**Inhaltsverzeichnis** **11**

**1 Problemstellung** **15**
1.1 Warum Prozessmanagement . . . 15
1.2 Probleme aktueller Modellierungssprachen . . . 16
1.2.1 Komplexität bei agilen Prozessmodellen . . . 16
1.2.2 Schwache Ausdrucksfähigkeit . . . 19
1.3 Ziele . . . 20
1.4 Lösungsansatz . . . 21
1.4.1 Von der imperativen zur deklarativen Problemformulierung . . . 21
1.4.2 Agilität und Erweiterbarkeit durch Prozessconstraints . . . 23
1.4.3 Navigation bei der Prozessausführung . . . 24
1.5 Umsetzung . . . 24
1.6 Aufbau der Dissertation . . . 25

**2 Architektur** **27**
2.1 Überblick . . . 27
2.2 Modellierung in iPM$^2$ . . . 28
2.3 Ausführung im ProcessNavigator . . . 29
2.3.1 Worklist . . . 30
2.3.2 Navigation . . . 31
2.4 Zusammenfassung . . . 31

**3 Konzeption** **33**
3.1 Deklaratives Programmieren . . . 33
3.2 Deklarative Prozesse . . . 36
3.2.1 Prozess und Perspektiven . . . 36
3.2.2 Detaillierter Aufbau eines Prozesses . . . 38
3.2.3 Trennung von Prozess und Zustand . . . 39
3.3 Kategorisierung von Prozessconstraints . . . 40
3.3.1 POPM-Perspektiven . . . 40
3.3.2 Prozesshandlungen . . . 41
3.3.3 Prozesszustände . . . 42
3.3.4 Prozessinstanzen . . . 42
3.4 Zusammenfassung . . . 42

**4 Modellierung** **45**
4.1 Klinischer Anwendungsfall . . . 45
4.2 Modellbasierte Umsetzung . . . 46
4.2.1 Funktionale Perspektive . . . 46
4.2.2 Verhaltensbezogene Perspektive . . . 47
4.2.3 Organisatorische Perspektive . . . 47
4.2.4 Datenbezogene Perspektive . . . 48
4.2.5 Operationale Perspektive . . . 49
4.3 Constraintbasierte Umsetzung . . . 49
4.3.1 Funktionale Constraints . . . 49
4.3.2 Verhaltensbezogene Constraints . . . 52
4.3.3 Organisatorische Constraints . . . 53
4.3.4 Datenbezogene Constraints . . . 54
4.3.5 Operationale Constraints . . . 54
4.4 Zusammenfassung . . . 55

**5 Validierung** **57**
5.1 Auswertung der POPM-Prozessconstraints . . . 57
5.1.1 Funktionale Prozessconstraints . . . 58
5.1.2 Verhaltensbezogene Prozessconstraints . . . 58
5.1.3 Organisatorische Prozessconstraints . . . 58
5.1.4 Datenbezogene Prozessconstraints . . . 59
5.1.5 Operationale Prozessconstraints . . . 59
5.2 Validierung der Prozesshandlungen . . . 59
5.3 Kommunikation mit dem ProcessNavigator . . . 62
5.4 Zusammenfassung . . . 63

**6 Erweiterbarkeit** **65**
6.1 Neue Modellierungskonstrukte . . . 65
6.1.1 Prozessverbindende Symbole . . . 66
6.1.2 Prozessübergreifende Symbole . . . 67
6.1.3 Übersetzung nach **ESP**ro**N**a . . . 68
6.2 Implementierung neuer Perspektiven . . . 70
6.2.1 Anpassung der Prädikate . . . 70
6.2.2 Schritte der Anpassung . . . 72
6.3 Zusammenfassung . . . 73

**7 Prozessnavigation** **75**
7.1 Konzeption . . . 75
7.1.1 Zustand eines Prozessmodells . . . 76
7.1.2 Berechnung der Folgezustände . . . 77
7.1.3 Zustandsübergangsrelationen . . . 78
7.1.4 Suchalgorithmen . . . 79
7.2 Navigation zwischen beliebigen Zuständen . . . 82
7.3 Interaktion mit dem Nutzer . . . 84
7.4 Verzögerte Validierung . . . 86
7.5 Zusammenfassung . . . 87

**8 Modellierung von Empfehlungen 89**
8.1 Prozessbezogene Empfehlungen . . . 89
8.1.1 Multimodale Logik . . . 89
8.1.2 Modale Kategorisierung der Prozessconstraints . . . 91
8.2 Erweiterung der Zustandsübergangsrelation . . . 93
8.3 Prozessübergreifende Empfehlungen . . . 95
8.3.1 Funktionale Perspektive . . . 95
8.3.2 Verhaltensbezogene Perspektive . . . 96
8.4 Korrigiertes Suchverhalten . . . 98
8.5 Zusammenfassung . . . 99

**9 Implementierung 101**
9.1 Strukturelle Übersicht . . . 101
9.2 Zusammenhänge der einzelnen Komponenten . . . 103
9.3 Zusammenfassung . . . 105

**10 Related Work 107**
10.1 Anforderungen . . . 107
10.1.1 Ausdrucksstärke . . . 107
10.1.2 Erweiterbarkeit . . . 107
10.1.3 Prozessnavigation . . . 108
10.1.4 Empfehlungen . . . 108
10.2 Imperative Modellierungssprachen . . . 108
10.2.1 BPMN . . . 108
10.2.2 YAWL . . . 109
10.3 Deklarative Modellierungssprachen . . . 110
10.3.1 DECLARE und ConDec . . . 110
10.3.2 EM-BrA$^2$CE . . . 111
10.4 Zusammenfassung . . . 112

**11 Resümee und Ausblick 115**

**A Installation 119**

**B Quellcode des klinischen Prozessmodells 123**

**C Sprachdefinition 129**
C.1 Funktionale Constraints . . . 129
C.2 Verhaltensbezogene Constraints . . . 130
C.3 Organisatorische Constraints . . . 131
C.4 Datenbezogene Constraints . . . 133
C.5 Operationale Constraints . . . 133

**D Signaturen der ESProNa-Objekte 135**
D.1 Functional Constraint Category . . . 136
D.2 Behavioral Constraint Category . . . 139
D.3 Organizational Constraint Category . . . 140
D.4 Data Constraint Category . . . 141
D.5 Operational Constraint Category . . . 143

D.6 Functional Heuristic . . . 144
D.7 Behavioral Heuristic . . . 145
D.8 Organizational Heuristic . . . 146
D.9 Data Heuristic . . . 147
D.10 Operational Heuristic . . . 148
D.11 Process . . . 149
D.12 Process State . . . 152
D.13 Model State . . . 154
D.14 State Space . . . 156
D.15 Heuristic State Space . . . 158
D.16 Process Planning . . . 160
D.17 Search Strategy . . . 161
D.18 Blind Search . . . 162
D.19 Heuristic Search . . . 163
D.20 Depth First Search . . . 165
D.21 Breadth First Search . . . 166
D.22 Hill Climbing Search . . . 167
D.23 Best First Search . . . 168

**Abbildungsverzeichnis** **169**

**Quellcodeverzeichnis** **171**

**Tabellenverzeichnis** **173**

**Literaturverzeichnis** **175**

# Kapitel 1

# Problemstellung

## 1.1 Warum Prozessmanagement

Immer mehr Unternehmen haben das Bedürfnis, ihre Unternehmensabläufe detailliert zu dokumentieren. Die Gründe hierfür sind sehr vielfältig. Mit einer qualitativ angemessenen Dokumentation die Geschäftsprozesse zu belegen, kann zum Beispiel für eine Zertifizierung sehr wichtig sein [1][2][3][4]. An diesem Punkt kommen Prozessmodellierungssprachen (im Weiteren mit PMS abgekürzt) zum Einsatz. Die formulierten Prozessmodelle können die Abläufe bei der Fertigung eines Produktes (Auto, Kaffeemaschine etc.) beschreiben oder aber den genauen Ablauf einer Dienstleistung festlegen (Erstellung einer Versicherungspolice, Freischaltung eines Benutzerkontos o. ä.). Diese sogenannten Anwendungsprozesse beschreiben auf grafischer Ebene die Erfüllung einer Aufgabe in einem Unternehmen. Die Modellierung dieser Geschäftsvorgänge erfolgt dabei in einer PMS. Für die Erstellung solcher Prozessmodelle ist zum einen ein sehr detailliertes Verständnis der Abläufe in einem Unternehmen wichtig, zum anderen benötigt man ein Modellierungswerkzeug, mit dem sich diese exakt in einem beschreibenden Prozessmodell festhalten lassen. Derjenige Mitarbeiter, der für die Umsetzung der Unternehmensprozesse zuständig ist, wird oft als Prozessmodellierer bezeichnet und ist maßgeblich dafür verantwortlich, ein möglichst genaues Abbild der Geschäftsprozesse in ein Modell umzusetzen.

Neben der Modellierung von Geschäftsprozessen entwickeln sich zusätzliche Erwartungen an die Optimierung der modellierten Prozesse [5]. Die aufgenommenen Prozesse können verwendet werden, um quasi aus der „Vogelperspektive“ die Unternehmensprozesse zu analysieren, zu bewerten sowie eine Verbesserung der Abläufe zu erreichen. Des Weiteren können diese Modelle dazu verwendet werden, eine Hilfestellung während der Prozessausführung zu geben. Die in diesem Zusammenhang verwendeten Anwendungsprogramme [6] sind in der Literatur als Workflowmanagementsysteme [5][7] (im Weiteren mit WMS abgekürzt) bekannt und stellen die technische Umsetzung der Geschäftsprozesse dar. Das ausgeführte Abbild eines Anwendungsprozesses wird als Workflow bezeichnet. Das Hauptziel eines WMS ist in erster Linie die Ausführung von Prozessen. Der Mitarbeiter des Unternehmens bekommt, nach der Anmeldung am WMS, gewisse Aufgaben zugewiesen, die bestimmte Teile des auszuführenden Geschäftsprozesses ausmachen. Durch die Abarbeitung dieses Workflows zusammen mit anderen Mitarbeitern wird der Prozessablauf komplettiert. Hierbei ist für den reibungslosen Arbeitsablauf in einem Unternehmen wichtig, dass die Abarbeitung der Prozessabläufe ohne größere Verzögerungen möglich ist.

„Viele Wege führen nach Rom“. Dieser Satz wird häufig verwendet, um auszudrücken, dass es mehrere Lösungen zu einem Problem gibt und diese sehr vielfältig und individuell sein können. Auch wenn es unter der Menge der verschiedenen Lösungen „bessere“ und „schlechtere“ gibt, so haben doch alle Lösungen das gleiche Ziel vor Augen: einen „Weg nach Rom“ zu beschreiben. Ein weiterer Gesichtspunkt neben der Vielzahl an Wegen ist die in dieser Aussage assoziierte Entscheidungsfreiheit des „Reisenden“. Es bleibt ihm überlassen, welchen Weg er wählen möchte, solange er das Ziel „Rom“ beibehält. Die Entscheidungen können dabei von vielen Kriterien und Gegebenheiten abhängen und von Person zu Person unterschiedlich sein. Für den einen ist das Kriterium Benzinverbrauch wichtig, für einen anderen die Zeitdauer der Reise, für eine dritte Person sind es die Sehenswürdigkeiten entlang der Route. Weiterhin können sich diese Kriterien auch situationsbedingt ändern, das heißt, bei schlechtem Wetter nimmt man lieber die direkte Route, bei schönem die mit den meisten Sehenswürdigkeiten. Insgesamt hat der Reisende sehr viele Möglichkeiten sich zu entscheiden, um seinen individuellen Weg auszuwählen und zu gehen. Andererseits sind aber bestimmte Regeln zu beachten: Es müssen zum Beispiel Straßen vorhanden sein oder aber die Verkehrsregeln auf dem Weg zum Ziel berücksichtigt werden. Weiterhin können bei bestimmten Routen Mautgebühren anfallen, die natürlich vor Antritt der Reise bekannt sein sollten. Doch solange all diese Regeln eingehalten werden, steht der Individualität und Kreativität des Reisenden prinzipiell nichts im Wege.

Ganz ähnlich wie bei der „Reise nach Rom“ verhält es sich auch bei der Prozessmodellierung und der Ausführung. Situationsbedingt muss es möglich sein, unterschiedliche Entscheidungen an bestimmten Punkten eines Prozesses bzw. Workflows treffen zu können. Dieses empirische Verhalten findet sich auch in unserem alltäglichen Leben wieder, sodass es auch in Prozessmodelle mit integriert werden muss. Damit die Entscheidungsfreiheit in einem Workflow zur Verfügung gestellt werden kann, muss sie allerdings auch in das Prozessmodell mit aufgenommen werden. Der Begriff Agilität, der unser alltägliches Handeln im Alltag in seiner Flexibilität und Individualität zusammenfasst, umschreibt dieses Konzept sehr treffend. Das nächste Unterkapitel zeigt, warum aktuelle PMS Probleme damit haben.

## 1.2 Probleme aktueller Modellierungssprachen

Das angeführte Beispiel „Viele Wege führen nach Rom“ dient dabei als Motiv für den Forschungsbereich PMS. Fakt ist, dass nach menschlichen Erfahrungswerten Dinge oft auf verschiedene Art und Weise erledigt werden können. Diese Individualität ist erlaubt, solange keine Regeln verletzt werden. Ob nun die eine oder die andere Lösung die bessere oder schlechtere ist, sei vorerst einmal dahingestellt. Es geht im Wesentlichen zunächst darum, unser menschliches und individuelles Verhalten (in Bezug auf die Abläufe in Unternehmen) möglichst gut abbilden zu können. Und genau darin liegt das Problem der aktuellen PMS, das sich zum einen in sehr komplexen Prozessmodellen äußert und zum anderen darin, dass nicht der ganze Sachverhalt abgedeckt wird. Diese beiden Kritikpunkte gilt es zunächst genauer zu erläutern.

### 1.2.1 Komplexität bei agilen Prozessmodellen

„Was nicht explizit erlaubt ist, ist verboten“. Diesem Grundgedanken folgen die meisten kommerziellen PMS. Für die zugrunde liegenden Geschäftsprozesse bedeutet dies, dass

jeder mögliche Ablauf (das heißt, jeder „Weg nach Rom“) explizit im Prozessmodell durch einen bestimmten Ablaufpfad modelliert werden muss. Für sehr strikte Prozessabläufe, das heißt, es wird explizit gewünscht, dass es nur genau einen möglichen Ablauf geben kann, stellt dies kein Problem dar. Für sehr agile Unternehmensabläufe allerdings ist dieser restriktive Ansatz ein Problem: Nicht alle Abläufe von Ausführungspfaden können explizit modelliert werden. Die Anzahl der einzelnen Pfade ist einfach zu groß. Als Beispiel [8] seien hier drei Prozesse (einfachheitshalber A, B und C genannt) gewählt, bei denen B zwischen ein- und dreimal (1..3) durchgeführt werden kann. Alle drei Prozesse müssen nacheinander und in genau dieser Reihenfolge ablaufen. Der Freiraum besteht nun darin, es dem Prozessausführenden zu überlassen, wie oft er Prozess B ausführen möchte. Allerdings muss diese Entscheidung vor der Ausführung von Prozess C getroffen werden. Transferiert auf das Prozessmodell bedeutet dies, dass darin sechs explizite Ausführungspfade aufgenommen werden müssten. Abbildung 1.1 zeigt das erstellte Prozessmodell mit den verschiedenen Ablaufszenarien; es verschleiert aber die eigentliche Prozessbeschreibung, das heißt, die Semantik der Prozesse ist nicht mehr erkennbar.

Unter Bezug auf den Grundsatz „Was nicht explizit erlaubt ist, ist verboten“ bedeutet dies, dass, falls nicht jeder dieser Abläufe modelliert wird, er während der Ausführung auch nicht zur Verfügung steht. Somit ist der Modellierer gezwungen, jedes noch so kleine Stückchen Flexibilität explizit zu modellieren. Oft ist dies aber aufgrund der Tatsache, dass die resultierenden Prozessmodelle zu komplex und vor allem auch zu zeitaufwendig in der Erstellung werden, nicht möglich. Außerdem muss dabei beachtet werden, dass die Flexibilität, die zur Modellierungszeit nicht mit in das Prozessmodell aufgenommen wurde, konsequenterweise zur Ausführungszeit auch nicht zur Verfügung steht. Daraus folgt während der Abarbeitung von Prozessmodellen, dass eine softwaregestützte Prozessausführung diese Agilität auch nicht zur Verfügung stellen kann, da sie im Prozessmodell nicht definiert wurde.

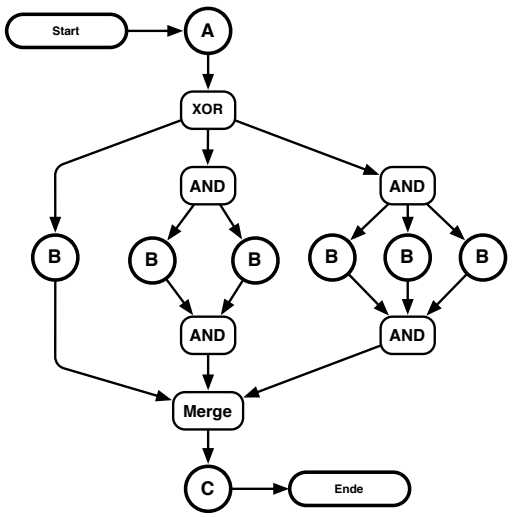



Abbildung 1.1: Drei Prozesse in bestimmter Ausführungskomposition

Möchte der Prozessmodellierer aber die Freiheit lassen, die Ausführungsreihenfolge einzelner Prozessschritte selbst zu wählen, so ergibt sich ein ähnliches Problem, das folgendes Beispiel verdeutlicht: Auszugehen ist wieder von den drei Prozessschritten A,

B und C. Jeder Schritt wird diesmal nur genau einmal durchgeführt, und während seiner Abarbeitung darf kein anderer Prozess parallel ablaufen, das heißt, jeder Prozess besitzt exklusive Ausführungsrechte. Die Flexibilität besteht hier darin, dass A, B und C in beliebiger Reihenfolge abgearbeitet werden können.

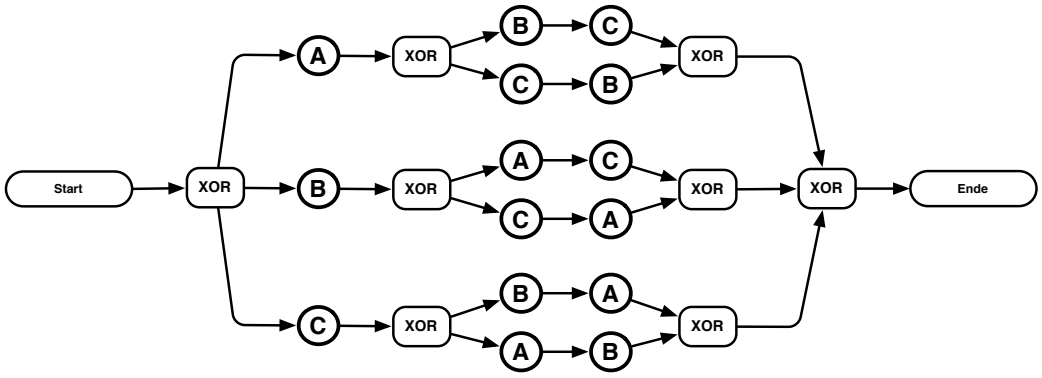



Abbildung 1.2: Drei Prozesse in beliebiger Ausführungsreihenfolge

Im Prozessmodell in Abbildung 1.2 ist diese Ausführung umgesetzt. Bei den drei genannten Prozessen ergeben sich 3! mögliche Abläufe (= 6 Lösungswege), die alle explizit im Modell abgebildet wurden. Nimmt man nun beispielsweise 5 verschiedene Prozessschritte an, so ergeben sich bereits 5! (= 120) mögliche Lösungswege, die alle im Modell untergebracht werden müssten. Das ist für den Modellierer eine sehr zeitaufwendige und müßige Arbeit, nur um auszudrücken, dass die Prozesse in beliebiger Reihenfolge ausgeführt werden können!

Dieses Problem ließe sich umgehen, wenn man benutzerdefinierte Modellierungskonstrukte einführt und diese mit einer konkreten Bedeutung für zum Beispiel die Ausführungsreihenfolge belegt. Als Beispiel sei hier ein gestrichelter Pfeil gewählt, der, genau wie der bisherige durchgezogene Pfeil, zwei Prozesse miteinander verbindet. Allerdings wird seine Bedeutung gegenüber dem durchgezogenen Pfeil insoweit abgeändert, als nun die Ausführungsreihenfolge der miteinander verbundenen Prozesse beliebig ist. Das Prozessmodell aus Abbildung 1.3 zeigt, wie durch die Verwendung des neuen Modellierungskonstrukts, das Prozessmodell aus Abbildung 1.2 stark vereinfacht werden kann. Der Prozessausführende kann nun mit einem der drei Prozesse A, B oder C beginnen, alle drei sind also direkt ausführbar. Die Modellierung des komplexen Ablaufverhaltens vereinfacht sich. Nur noch ein Weg muss modelliert werden, der aber den vielen Wegen aus Abbildung 1.2 semantisch äquivalent ist. Wäre es also möglich, die PMS so zu erweitern, dass benutzerdefinierte Modellierungskonstrukte möglich sind, dann könnte der Prozessmodellierer dieses Konzept nutzen, um neue Konstrukte semantisch zu definieren und somit die Pfadkomplexität zu reduzieren oder ganz zu umgehen.

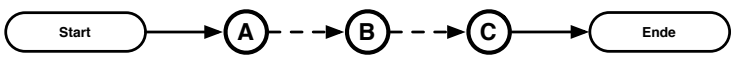



Abbildung 1.3: Vereinfachung der Komplexität durch neues Modellierungssymbol

#### Konsequenzen

Fazit ist, dass Flexibilität prinzipiell zwar mit aktuellen PMS modelliert werden kann (siehe auch Kapitel 10), allerdings die erstellten Modelle dann extrem komplex werden. Folgende Erfahrungen, die sich durch fehlende Ausführungsflexibilität ergeben, sind im klinischen Umfeld gemacht worden: Der Nutzer des Workflowsystems fühlt sich in seiner Entscheidungsfreiheit stark eingeschränkt. Ausnahmen oder Abweichungen vom Standardverhalten ergeben sich aber in der Realität nur allzu oft und müssen dementsprechend auch handhabbar sein. Aktuelle WMS werden jedoch den Anforderungen in dieser Hinsicht nicht gerecht und werden deshalb nicht oder nur ungern eingesetzt.

### 1.2.2 Schwache Ausdrucksfähigkeit

Ein weiteres Problem aktueller PMS ist ihre schwache Ausdrucksfähigkeit. Darunter versteht man, wie gut bestimmte gedankliche Vorstellungen mit einer Sprache (im vorliegenden Fall der PMS) umgesetzt bzw. modelliert werden können, hier also die detaillierten Geschäftsabläufe, die der Prozessmodellierer in ein Modell umsetzen möchte. Jedes Konzept, das sich der Modellierer vorstellt und das für den Anwendungsbereich wichtig ist, muss (effizient) modellierbar sein. Hierbei unterscheidet man nun zwei Fälle: Der erste Fall ergibt sich aus dem Problem, dass die Konzepte, die der Modellierer umsetzen möchte, mit der PMS nicht umsetzbar sind. Der zweite Fall tritt auf, wenn das erstellte Modell nur eine Approximation des realen Unternehmensablaufs darstellt. Beide Probleme müssen genauer erläutert werden.

#### Nicht realisierbare Konzepte

Als Beispiel dienen zwei Prozessschritte, wovon der erste von einem beliebigen Mitarbeiter eines Unternehmens ausgeführt werden kann; der nachfolgende Prozessschritt muss aber von dessen Vorgesetztem umgesetzt werden. Die Umsetzung genau dieser Restriktion ist mit den meisten kommerziellen PMS nicht möglich, da die benötigten Modellierungskonstrukte fehlen. In diesen PMS kann zum einen keine Beziehung zwischen den Ausführenden der beiden Prozesse hergestellt werden, und zum anderen ist die hierarchische Relation (Vorgesetztenrolle) zwischen beiden Personen in der Modellierungssprache nicht abbildbar. Als Beleg für diese These dient die Evaluierung verschiedener PMS aus [8]. Das gewählte Beispielprozessmodell wird dort als „Pattern No. 4“ bezeichnet und beschreibt den obigen Anwendungsfall der Autorisierung. Tabelle 1.1 zeigt, dass nur die Sprachen COSA und FLOWer das Konzept der Autorisierung unterstützen (durch + gekennzeichnet). Alle weiteren Mainstream-Sprachen wie etwa BPMN, jBPM oder Oracle BPEL unterstützen dieses Konzept nicht (durch – gekennzeichnet). In den gegenwärtigen PMS gibt es also bis auf die zwei genannten Ausnahmen keine Möglichkeit, die Rolle des Vorgesetzten in Abhängigkeit von einem vorangegangenen Prozess darzustellen. Und dies ist nur ein sehr einfaches Beispiel. Weitere könnten sein, dass immer der Dienstälteste oder der Außendienstmitarbeiter mit dem zugeordneten Kundenkontakt den Prozess ausführen soll. Sind diese Konzepte in der gewählten PMS nicht implementiert, so entsteht für den Modellierer ein Problem.

Wäre die PMS aber an die obigen Anforderungen des Unternehmens individuell anpassbar, wäre das Problem gelöst. Der Modellierer könnte die Sprache nach den Bedürfnissen des Unternehmens ausrichten, das heißt, er könnte die obigen Rollenbeziehungen

| Modellierungssprache | Version | unterstützt |
|---|---|---|
| Staffware | 9 | - |
| Websphere MQ Workflow | 3.4 | - |
| FLOWer | 3.0 | + |
| COSA | 4 | + |
| iPlanet | 3.1 | - |
| BPMN | 1.0 | - |
| UML | 2.0 | - |
| Oracle BPEL | 10.1.2 | - |
| jBPM | 3.1.4 | - |
| OpenWFE | 1.7.3 | - |
| Enhydra Shark | 2 | - |

Tabelle 1.1: Sprachbezogene Umsetzung des Autorisierungspatterns

zwischen den Personen selbst in die PMS integrieren. Somit könnten auftretende Probleme durch eine Anpassung der Sprache seitens des Modellierers gelöst werden. Allerdings entscheiden die Entwickler einer Modellierungssprache, was genau in ihr umgesetzt wird und was nicht. Der Modellierer muss damit auskommen. Durch die Evaluierung in [8] bekommen die Entwickler der jeweiligen Sprache zwar Feedback seitens der Community, welche Pattern und Modellierungskonstrukte benötigt werden, jedoch obliegt deren Umsetzung allein ihnen.

#### Konsequenzen

Die Konsequenz ist, dass obige Rollenbeziehungen nicht ins Prozessmodell mit aufgenommen werden können. Als Folge davon entstehen unpräzise Modelle, die nicht deckungsgleich mit den realen Geschäftsprozessen sind. Sie stellen nur eine Annäherung an die komplexen Sachverhalte dar. Neben der Modellierung darf man nicht vergessen, dass die entstandenen Prozessmodelle auch im Nachhinein wieder aufbereitet und verstanden werden müssen. Zu einem vorhandenen Prozessmodell möchte man den empirischen Unternehmensablauf wieder extrahieren und verstehen lernen. Dies kann zu einem späteren Zeitpunkt durch den Autor des Modells erfolgen, aber auch eine andere Person, die das Modell vorher noch nie gesehen hat, kann es benutzen, um den Unternehmensablauf zu verstehen. Allerdings ist der aus dem Prozessmodell wiedergewonnene Ablauf oft sehr unpräzise oder sogar falsch, das heißt, er ist nicht identisch mit dem, den man zum Zeitpunkt der Modellierung festhalten wollte. Zusammenfassend ergibt sich also, dass bei mangelnder Ausdrucksfähigkeit von PMS die Gefahr besteht, dass die zu modellierenden Geschäftsabläufe ungenau modelliert sind und im Nachhinein nicht korrekt reproduziert werden können. Die aus dem Modell gewonnene Information ist nicht dieselbe, die der Modellierer zum Zeitpunkt des Erstellens einbringen wollte. So entsteht ein Informationsverlust.

## 1.3 Ziele

Aus den aufgezeigten Problemen lassen sich nun Konzepte entwickeln, die eine PMS für den Modellierer und Prozessausführenden mitbringen muss, um eine breitere Akzeptanz

zu finden. Der Prozessmodellierer soll in der Lage sein, die PMS individuell den Gegebenheiten im Unternehmen anpassen zu können. Die für den Modellierer optimale PMS muss also zum einen so erweiterbar sein, dass agile Unternehmensabläufe mit der darin enthaltenen semantischen Komplexität präzise in ein Modell umgesetzt werden können, zum anderen aber auch soweit an die flexiblen Veränderungen der Geschäftsabläufe anpassungsfähig sein, dass neue Modellierungskonzepte einfach zu integrieren sind. Ein weiteres Ziel ist die Bereitstellung eines WMS, um eine Hilfestellung während der Prozessausführung zu ermöglichen. Bei der Zuteilung der Prozesse zu der Aufgabenliste der Nutzer ist es wichtig, eine effektive Auslastung zu gewährleisten. Grundvoraussetzung hierfür ist, eine flexible und agile PMS zu benutzen, die den Anwender in der Ausführung eines Prozesse nur dann einschränkt, wenn es nötig ist. Auch werden Prozessmodelle referenziert, um den semantischen Kontext von Unternehmensabläufen aus einem Prozessmodell zu extrahieren. Für diesen Punkt ist es wichtig, übersichtliche Prozessmodelle zu haben, aus denen man den semantischen Kontext einfach wiedergewinnen kann. Die gewählte PMS muss dazu einen Spagat zwischen der Inklusion aller flexiblen Abläufe und einer geringen Pfadkomplexität schaffen, also klar strukturierte und verständliche Prozessmodelle gewährleisten. Nur so ist sichergestellt, dass modellierte Geschäftsabläufe im Nachhinein auch korrekt wieder reproduziert werden können.

## 1.4 Lösungsansatz

Blickt man auf die aktuellen Modellierungssprachen, so erkennt man darin einen sehr starken imperativen Charakter, der dem Konzept „Was nicht explizit erlaubt ist, ist verboten“ folgt. Imperativ bedeutet in diesem Kontext, dass jeder nächstmögliche Prozessschritt im Prozessmodell genau vordefiniert ist [9]. Hierbei spricht man von Determinismus. Um nun die angesprochenen Probleme aktueller PMS zu lösen, ist es notwendig, dieses sehr strikte und imperative Konzept zu ändern.

### 1.4.1 Von der imperativen zur deklarativen Problemformulierung

Die grundlegende Idee für den Wechsel von der imperativen zur deklarativen Problemformulierung macht ein grafisches Beispiel (Abbildung 1.4) deutlich: Hier werden vom Start (links) zum Ziel (rechts) verschiedene Wege explizit modelliert. Der Ablauf eines Weges verläuft immer von links nach rechts und spiegelt jeweils einen möglichen Lösungspfad wider. Möchte man nun einen zusätzlichen Lösungsweg hinzufügen (gepunktete Linie), so muss dieser explizit beschrieben und hinzugefügt werden.

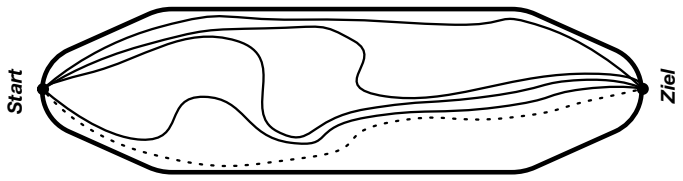



Abbildung 1.4: Explizite Lösungswege bei imperativen Modellierungssprachen

Abbildung 1.5 beschreibt die deklarative Welt als Gegensatz zu den expliziten Pfadangaben der imperativen Welt. Die grauen, rechteckigen Gebilde beschreiben die Zonen,

die verboten sind. Man kann sich also auf der weißen Fläche frei bewegen, darf aber keine der grauen Zonen betreten. Das Hinzufügen des gepunkteten Weges muss bei dieser Art der Modellierung nicht explizit erfolgen, da er, wie Abbildung 1.6 zeigt, schon enthalten und somit bereits implizit aufgenommen ist.

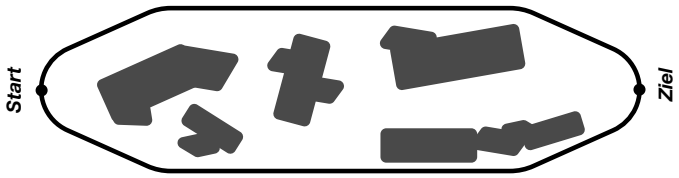



Abbildung 1.5: „Verbotene Zonen“ bei deklarativen Modellierungssprachen

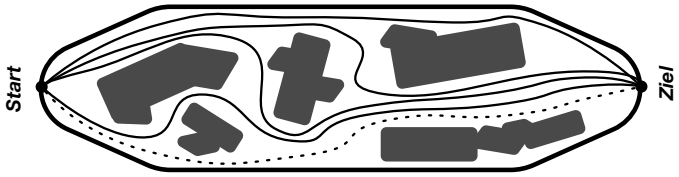



Abbildung 1.6: Zusätzliche Pfade sind bereits enthalten

Ähnlich wie in diesem grafischen Beispiel verhält es sich auch mit dem Wechsel von der imperativen zur deklarativen Prozessmodellierung. In Abbildung 1.4 kann man sich nur auf den gekennzeichneten Pfaden bewegen, das heißt, „Was nicht explizit erlaubt ist (der Weg), ist verboten“. Möchte man weitere Wege zulassen (gepunktete Linie), so muss man diese explizit hinzufügen. Abbildung 1.5 stellt den konträrer Ansatz dar: „Was hier nicht (durch die grauen Inseln) verboten ist, ist erlaubt.“ Flexibilität ist also nativ gegeben. Der gepunktete Lösungsweg muss, neben vielen weiteren alternativen Pfaden, nicht explizit modelliert werden, da er bereits enthalten ist.

Deklarative Programmiersprachen bilden in der Informatik eine Sprachgruppe, bei der die Beschreibung des Problems im Vordergrund steht [10] [11]. Sie stellen einen Kontrast zu den imperativen Programmiersprachen dar. Man gibt die Spielregeln vor, nach denen gespielt wird, sowie den Ausgangspunkt und das Ziel. Der Lösungsweg wird automatisch berechnet. Nimmt man wieder das Beispiel des „Weges nach Rom“, so gibt man in diesem Fall also nur Start, Ziel und Zwischenstationen an sowie die Verkehrsregeln und die vorhandenen Straßen. Sudoku ist ein gutes Beispiel für die Anwendung einer deklarativen Programmiersprache und wird oft als Paradebeispiel dafür in Vorlesungen benutzt (Kapitel 3.1). Der Grundgedanke dabei ist, dass man nur die Regeln beziehungsweise Constraints des Spiels angibt sowie die Ausgangssituation, das heißt, die teilweise mit Zahlen belegte Matrix. Die Programmiersprache setzt nun sogenannte freie Variable an die unbesetzten Stellen der Sudoku-Matrix und versucht, die Werte der Variablen so zu belegen, dass sie zusammen mit den vorgegebenen Werten eine Lösung gemäß den Spielregeln bilden. Dieses Binden von Werten an die Variablen geschieht automatisch. Der Anwender oder Programmierer muss sich also nicht explizit um die Lösungsfindung kümmern, sondern programmiert lediglich die Constraints des Spiels.

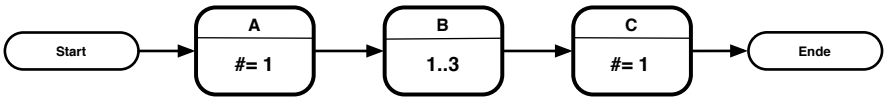



Abbildung 1.7: Deklarative Formulierung des Prozessmodells 1.1

Bezogen auf das Prozessmodell aus Abbildung 1.1, bei dem der Prozess B zwischen einmal und dreimal ausgeführt werden kann, muss also im deklarativen Fall nur dieses Constraint modelliert werden („1..3“ im Prozess B in Abbildung 1.7). Die Regeln innerhalb der Prozesse markieren, wie oft diese ausgeführt werden dürfen. Das ist bei Prozess B auf den ersten Blick bereits klar. Der Leser muss nicht erst die einzelnen Pfade im Modell nachverfolgen, um das extrahieren zu können.

Man stelle sich nun vor, man müsste die beiden imperativen Prozessmodelle aus den Abbildungen 1.1 und 1.2 kombinieren. Zum Prozessmodell aus Abbildung 1.1 kommt hinzu, dass die Prozesse A, B und C in beliebiger Reihenfolge ausführbar sind. Mit dem neuen Modellierungssymbol des gestrichelten Pfeils ist dies kein Problem mehr. Zusammen mit dem Constraint in Prozess B, das die mögliche Anzahl an Ausführungen wiedergibt, erhält man das in Abbildung 1.8 dargestellte, sehr kompakte Prozessmodell. Die imperative Variante hingegen würde aus 48 expliziten Pfadangaben bestehen und ist aus Platzgründen hier nicht aufgeführt.


Abbildung 1.8: Deklarative Komposition der Prozessmodelle aus den Abbildungen 1.1 und 1.2

### 1.4.2 Agilität und Erweiterbarkeit durch Prozessconstraints

Das in Kapitel 1.4.1 erläuterte Konzept aus dem Bereich der deklarativen Programmiersprachen dient nun als Vorbild, um eine deklarative PMS zu entwickeln. Dabei soll die Handhabung dem Prozessmodellierer soweit erleichtert werden, dass er sich in erster Linie nur noch um die Problemstellung kümmern muss, also das WAS, das heißt, er muss nicht mehr die expliziten Lösungswege dafür im Prozessmodell angeben, sondern nur noch die Constraints zu den einzelnen Prozessen. Eine Fokussierung auf den eigentlichen Prozess während der Umsetzung von Unternehmensabläufen steht dabei im Vordergrund. Dies erleichtert die Arbeit des Prozessmodellierers, da die expliziten Lösungen, das heißt, WIE die Prozessregeln umgesetzt werden können, nicht mehr modelliert werden müssen. Durch dieses Modellieren der Prozessconstraints erreicht er einen Konzeptwechsel von „Was nicht explizit erlaubt ist, ist verboten“ zu „Was nicht verboten ist, ist erlaubt“. Die Prozessabläufe sind nur dann noch in ihrer Ausführung eingeschränkt, wenn eine bestimmte Voraussetzung nicht erfüllt ist und ein Constraint diese als Notwendigkeit vorgibt. Wird allerdings kein Constraint eines Prozesses verletzt, so steht dessen Ausführung nichts im Wege.

Durch das in Abbildung 1.3 erstmalig neu eingeführte Modellierungssymbol (gestrichelter Pfeil) lässt sich erkennen, wie durch eine Erweiterung der Sprache um neue Modellierungskonstrukte die Prozessmodelle vereinfacht werden können. Die Grundlage für die Definition dieses Modellierungssymbols bilden die Prozessconstraints. Auf Basis dieser Regeln ist es möglich, die Semantik eines neuen Modellierungskonstrukts klar zu definieren.

### 1.4.3 Navigation bei der Prozessausführung

Durch die neu hinzugekommene Agilität der Prozessmodelle kann das Problem entstehen, dass der Prozessausführende durch die vielen Auswahlmöglichkeiten an manchen Stellen überfordert ist. Dies soll durch das Navigationskonzept kompensiert werden, das in der im Rahmen dieser Dissertation entwickelten PMS auch realisiert wurde. Es ist nun möglich, Abläufe beziehungsweise Pfade zu ganz bestimmten Punkten im Prozessmodell oder Unternehmensablauf zu berechnen. Ist der Prozessausführende an einem Punkt mit zu vielen Entscheidungsmöglichkeiten überfordert, so kann er in der Navigationskomponente das Ziel angeben, zu dem er im Prozessmodell navigieren möchte. Als Ergebnis bekommt er einen Arbeitsplan berechnet, den er abarbeiten muss. Somit ist das Problem der zu vielen Entscheidungsmöglichkeiten kompensiert.

Kommen nun noch Ausnahmesituationen hinzu, die einen gewünschten Prozessablauf verhindern, so kann dies ebenfalls ein Grund dafür sein, dass der Mitarbeiter bei der Ausführung der Prozesse schnell die Übersicht verliert. Ein ähnliches Problem ist aus dem Straßenverkehr geläufig: In bekannter Umgebung findet man sich gut zurecht, das heißt, man weiß bei auftretenden Problemen (zum Beispiel Straßensperrungen), welchen anderen Weg man nehmen kann, um an das Ziel zu kommen (ohne dass man eine Straßenkarte benötigt). In unbekannter Umgebung ist dies allerdings kaum möglich. Ähnlich verhält es sich bei den Prozessen: Alltägliche Prozessabläufe kennt man im Detail, und bei hier auftretenden Problemen kann man schnell Lösungen finden. Allerdings sieht es bei unbekannten Prozessmodellen anders aus. Ein unerfahrener oder neuer Mitarbeiter kann hier den Überblick bei Ausnahmesituationen verlieren, und es hilft ihm ungemein, ein Navigationssystem zur Hand zu haben.

Durch die integrierte Flexibilität der PMS wird der Nutzer bei Ausführung der Prozesse nur noch dann eingeschränkt, wenn ein Constraint die Ausführung eines Prozesses verbietet. Dies stellt sicher, dass sich der Anwender gemäß den Unternehmensrichtlinien verhält und die Validität des gesamten Prozessablaufes im Unternehmen garantiert ist. Weiterhin wird er bei Ausnahmesituationen unterstützt, um schnellstmöglich ein Lösung zu einem Problem finden zu können. Durch eine zusätzliche multimodale Gewichtung der Prozessconstraints während der Modellierung können Empfehlungen durch den Modellierer mit ins Prozessmodell aufgenommen werden. Hierdurch erreicht er eine gewisse Einflussnahme auf die Ausführung, ohne aber den Ausführenden in seinen Entscheidungen einzuschränken.

## 1.5 Umsetzung

**E**ngine for **S**emantic **Pro**cess **Na**vigation, kurz **ESP**ro**N**a, ist der Name der deklarativen PMS, die im Rahmen dieser Dissertation entwickelt wurde. Diese constraintbasierte [12] PMS beinhaltet außerdem ein WMS, um eine Ausführung der modellierten

Prozessmodelle zu ermöglichen. **ESP**ro**N**a selbst wurde als Applikation in der logisch-objektorientierten Programmiersprache Logtalk [13] verfasst und muss in der Ausführungsumgebung SWI-Prolog [14] geladen werden. Logtalk ist eine Erweiterung der Programmiersprache Prolog [15][16] und fügt dieser objektorientierte Konzepte [17][18] hinzu. Durch die Beschreibungssprache OWL [19] (Kurzform für **W**eb **O**ntology **L**anguage) ist es möglich, Unternehmensstrukturen wie beispielsweise die in Kapitel 1.2.2 genannte Vorgesetztenrelation festzuhalten. Die dabei erstellten Ontologien und deren Strukturen können durch eine Referenzierung im Prozessmodell benutzt werden. Hierdurch erreicht man ein modulares Design, mit dem die Erweiterbarkeit der PMS sichergestellt ist. Zusätzlich zur Sprache OWL wird das Modellierungswerkzeug Protégé [20] eingesetzt, mit dem die Unternehmensstrukturen grafisch modelliert werden können. Aus architekturbezogener Perspektive kann **ESP**ro**N**a als zentrale Bibliothek für Prozessmodellierungsapplikationen und WMS angesehen werden. Das implementierte Prozessnavigationssystem kann nicht nur während der Prozessausführung als Hilfestellung dienen, sondern auch in der Modellierungsphase zur Verifikation der möglichen Prozessabläufe. Der Prozessmodellierer kann somit unerwünschte Nebeneffekte rechtzeitig erkennen und vermeiden. **ESP**ro**N**a ermöglicht zusätzlich eine Unterscheidung zwischen notwendigen und empfohlenen Prozessconstraints. Der Modellierer kann dadurch Empfehlungen in sein Prozessmodell mit aufnehmen. So können die Entscheidungen des Prozessausführenden beeinflusst werden, ohne ihn aber darin zu stark einzugrenzen.

## 1.6 Aufbau der Dissertation

Die Arbeit gliedert sich in insgesamt elf Kapitel. Das vorliegende erste hat sich mit der Problemstellung beschäftigt und die Notwendigkeit der Implementierung neuer PMS und WMS gezeigt. Das zweite Kapitel zeigt die Architektur der am Lehrstuhl entwickelten Applikationen auf und gibt einen Überblick über die Modellierung und Ausführung von Prozessmodellen. Das dritte Kapitel widmet sich der Konzeption von **ESP**ro**N**a und zeigt, wie deklarative Prozesse und Prozessconstraints modelliert werden. Kapitel Vier zeigt anhand eines klinischen Anwendungsfalls, wie die Beschreibung eines Geschäftsprozesses in ein deklaratives und constraintbasiertes Prozessmodell umgesetzt werden kann. Die Validierung von modellierten Prozessen zur Ausführungszeit wird im fünften Kapitel besprochen und gezeigt wie die einzelnen Prozessperspektiven Einfluss auf diese nehmen. Kapitel Sechs zeigt auf, wie die PMS **ESP**ro**N**a erweitert werden kann: Zum einen werden dort neue Modellierungssymbole definiert, um anderen an einem Beispiel eine neue POPM-Perspektive implementiert. Prozessnavigation bildet den Inhalt des siebten Kapitels. Hier werden der Zustand eines Prozessmodells sowie die Zustandsübergangsrelationen im Detail erklärt, die für die Prozessnavigation wichtig sind. In Kapitel Acht wird gezeigt, wie der Prozessmodellierer Empfehlungen in Form von speziellen Prozessconstraints mit ins Prozessmodell aufnehmen kann. Kapitel Neun gibt einen Überblick über die Implementierung von **ESP**ro**N**a und beschreibt die Ordnerstruktur sowie das Zusammenspiel der beteiligten Objekte. Related Work wird im zehnten Kapitel behandelt, das einleitend aufführt, welche Argumente für eine Analyse mit ähnlichen PMS wichtig sind. Es teilt sich anschließend in einen Vergleich zwischen imperativen und deklarativen Modellierungssprachen auf. Kapitel Elf zieht ein Resümee der Ergebnisse dieser Dissertation und zeigt anhand eines Ausblicks, welche nachfolgenden Arbeiten auf diesem Forschungsgebiet noch möglich und notwendig sind.

# Kapitel 2

# Architektur

In diesem Kapitel werden die einzelnen Komponenten, die von der Prozessmodellierung bis hin zur Prozessausführung beteiligt sind, näher erläutert. In den einzelnen Teilkapiteln geht es darum, speziell auf die jeweiligen Komponenten einzugehen, mit denen der Prozessmodellierer die Abläufe in einem Unternehmen modelliert und der Prozessausführende die Modelle benutzt, um die täglichen Arbeitsabläufe zu organisieren. Abschnitt 2.1 gibt zunächst einen Überblick, wie das Zusammenspiel zwischen Modellierung und Ausführung von Prozessmodellen erfolgt. Beide Konzepte werden dann im Detail in den Abschnitten 2.2 und 2.3 besprochen.

## 2.1 Überblick

In Abbildung 2.1 wird das Zusammenspiel zwischen der Prozessmodellierung und der Prozessausführung grafisch verdeutlicht. Der Prozessmodellierer hält mittels iPM$^2$ die Unternehmensprozesse in grafischen Modellen fest. Ein Prozessmodell ist schematisch im oberen Teil der Grafik skizziert. Nach dem Modellieren der Geschäftsprozesse werden diese bei der Speicherung automatisch in einem speziellen Format (LMM [21]) abgelegt. Durch eine semantische Modelltransformation werde die LMM-Modelle in ein **ESP**ro**N**a-Prozessmodell konvertiert. Da nicht jedes in iPM$^2$ verwendete Modellierungskonstrukt ohne weiteres direkt in die PMS **ESP**ro**N**a umgesetzt werden kann, spricht man hier von einer semantischen Transformation. Dies bedeutet, dass ein Modellierungskonstrukt erst in seiner konzeptionellen Bedeutung (Semantik) erfasst werden muss. Anschließend kann ein Transformation festgelegt werden, die die konkrete Bedeutung des Konstrukts aus iPM$^2$ auf die der **ESP**ro**N**a-Sprache abbildet. In Kapitel 6 wird dieses Verfahren noch näher erläutert.

Im ProcessNavigator können die erstellten Prozessmodelle geladen und ausgeführt werden. **ESP**ro**N**a stellt für den ProcessNavigator über eine spezielle Schnittstelle (vgl. Anhang A) Informationen zur Verfügung, welche Prozessmodelle geladen und ausgeführt werden können. Durch das nachfolgende Laden eines bestimmten Modells wird davon eine Instanz erzeugt [22]. Diese Information wird durch **ESP**ro**N**a generiert und an den ProcessNavigator übermittelt. Da dieser ein Mehrbenutzer-System ist, bei dem sich verschiedene Nutzer am System anmelden können, wird für jeden Benutzer eine individuelle Aufgabenliste generiert. Dadurch, dass die Nutzer die einzelnen Listen abarbeiten, wird die Ausführung der Instanz eines Prozessmodells vervollständigt.

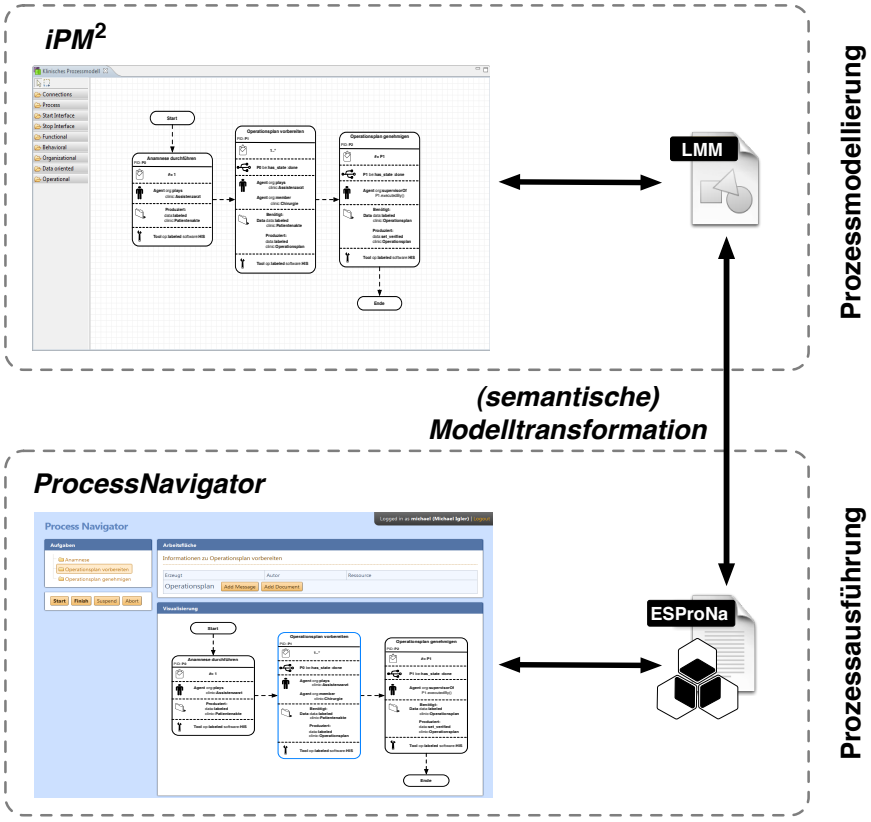



Abbildung 2.1: Architekturübersicht

## 2.2 Modellierung in iPM²

Am Lehrstuhl Angewandte Informatik 4 hat man es sich zur Aufgabe gemacht, eine neue PMS zu entwickeln, die Geschäftsprozesse in ihrer Gesamtheit und Dynamik erfassen kann. Um dies zu realisieren, ist es nicht nur wichtig, eine deklarative Sprache wie **ESP**ro**N**a als Grundlage zu definieren, sondern auch ein Werkzeug zur Verfügung zu stellen, mit dem das Modellieren von Unternehmensabläufen überhaupt erst realisiert werden kann.

iPM² ist die Neuimplementierung der Prozessmodellierungsumgebung iPM, gründet allerdings im Gegensatz zur Vorgängerversion auf einem metamodellierungsbasierten und modellgetriebenen Konzept [21]. Dieser Ansatz ermöglicht es, eine erweiterbare und domänenspezifische Modellierungsumgebung über sogenannte Modelle und Metamodelle zu generieren. Weiterhin ist es mit diesem metamodellierungsbasiertem Ansatz möglich, verschiedene Prozessmodellierungsdialekte mit nur einer Modellierungsumgebung zu unterstützen. Im vorliegenden Fall wird ein sogenannter **ESP**ro**N**a-Dialekt definiert, der dann in iPM² geladen werden kann. In ihm wird festgelegt, wie einzelne Prozesse konkret modelliert und welche inhaltlichen Regeln in diesen Prozessen angegeben werden.

Abbildung 2.2 zeigt einen Screenshot aus iPM², bei dem ein klinisches Prozessmodell im **ESP**ro**N**a-Dialekt modelliert wurde. Man erkennt in der Abbildung, dass es sich bei iPM² um eine Eclipse-Applikation handelt. Der linke Teil des Screenshots gliedert sich in eine Auswahl von Modellierungssymbolen (unter anderem einem Symbol für die Pro-

zesse), mit denen das Prozessmodell (rechts daneben) auf der gerasterten Zeichenfläche erstellt wurde. Die Symbole können per Drag&Drop in die Zeichenfläche gezogen und mittels Pfeilen (im Screenshot als Connections bezeichnet) verbunden werden. Weiterhin gibt es ein Start Interface (Start) bzw. ein Stop Interface (Stop). Hierbei handelt es sich um speziell ausgewiesene Prozesse, die Anfangs- und Endpunkte markieren, mit anderen Worten Beginn und Ende der Prozessausführung.

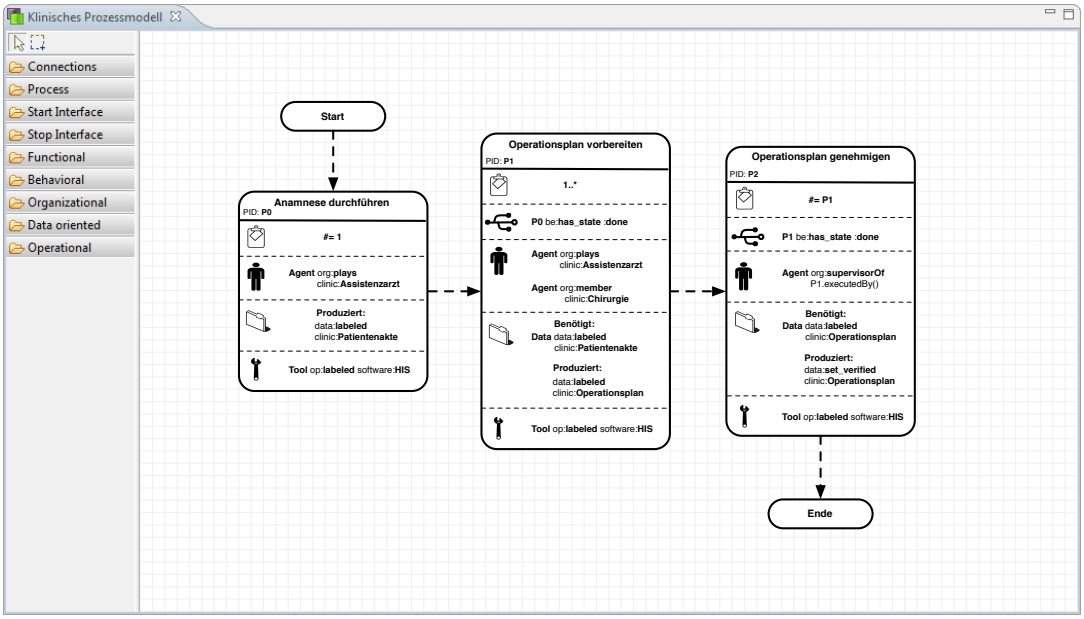



Abbildung 2.2: Screenshot iPM$^2$ mit **ESP**ro**N**a-Prozessmodell

Das Prozessmodell beginnt mit dem (abstrakten) Startprozess (Start Interface) und endet mit dem (abstrakten) Endprozess (Stop Interface). Alle Prozesse dazwischen sind über gestrichelte Pfeile miteinander verbunden. Diese alternierende Kombination aus Prozess und Verbindungssymbol stellt eine Art Prozesskette dar, die den Standardarbeitsablauf im Unternehmen widerspiegeln soll. Der Modellierer kann somit den alltäglichen bzw. üblichen Ablauf von Geschäftsprozessen erkennen und modellieren. Auch wenn man dieses Modell vorher nicht gekannt hat, erhält man schon auf den ersten Blick eine Übersicht über den gedachten Ablauf der Prozesse, ohne sich in vielen Pfaden zu verlieren.

## 2.3 Ausführung im ProcessNavigator

Der ProcessNavigator hat das Ziel, dem Benutzer des Systems bei der Ausführung modellierter Unternehmensabläufe zur Seite zu stehen. Der Begriff ProcessNavigator ist ein Kompositum aus Prozess und Navigation. Letzteres Konzept wird in Kapitel 7 noch im Detail erklärt werden. Es realisiert das in Kapitel 1.4.3 geforderte Modul zur Kompensation der Unübersichtlichkeit bei zu vielen Entscheidungsmöglichkeiten. Der Nutzer kann mit Hilfe dieses Konzepts seine Prozessziele dem System übermitteln und erhält als Ergebnis eine Aufgabenliste mit Anweisungen, die nach und nach abzuarbeiten sind, um zum gewünschten Ziel zu kommen. Bezüglich auf das in Kapitel 1 angesprochene Beispiel

„Viele Wege führen nach Rom“ kann man für die Prozessnavigation gewisse Parallelen ziehen. Der Prozessausführende möchte vielleicht bei dem Weg durch das Prozessmodell bestimmte Situationen durchlaufen oder aber auch vermeiden. Dies kann ähnlich dem „Rom“-Beispiel von gewissen Situationen abhängen. Dort waren sie zum Beispiel wetterabhängig, für die Welt der Prozessausführung könnten entsprechend betriebliche Ausfälle wie Krankheit oder urlaubsbedingte Abwesenheit eines Kollegen den Ablauf beeinflussen.

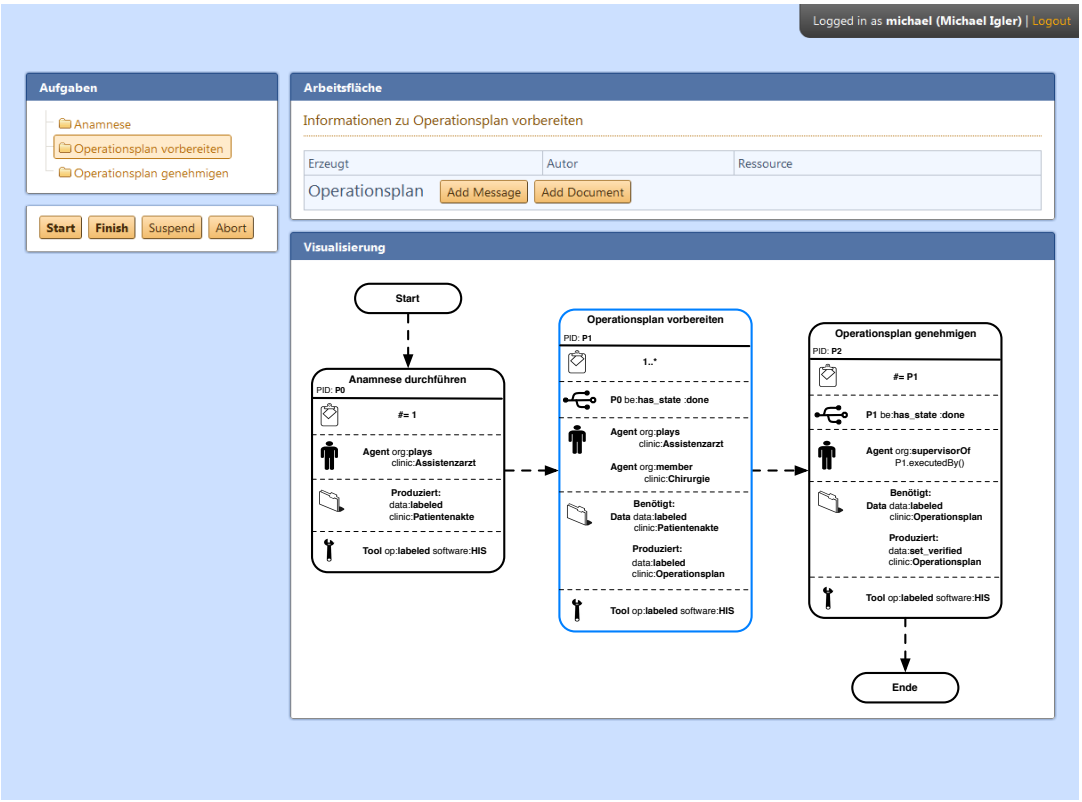



Abbildung 2.3: Screenshot ProcessNavigator

Prozessmodelle können im ProcessNavigator geladen werden und ermöglichen es dem Nutzer, sie strukturiert abzuarbeiten. In Abbildung 2.3 ist ein Screenshot der Applikation zu sehen, die als Webanwendung implementiert wurde. Somit ist es möglich, die Anwendung betriebssystemübergreifend zur Verfügung zu stellen. Die Oberfläche des ProcessNavigators ist in zwei Teilbereiche gegliedert: Aufgabenliste (Worklist) und Arbeitsfläche.

### 2.3.1 Worklist

Die Aufgabenliste zeigt an, welche Aufgaben für den eingeloggten Benutzer anstehen. Durch deren schrittweises Abarbeiten durch alle im Modell spezifizierten Agenten komplettiert sich die Abarbeitung des Prozessmodells. Die Arbeitsfläche auf der rechten Seite dient der Darstellung von aktuellen Kontextinformationen, die im Detail beschreiben, wie genau ein bestimmter Prozessschritt auszuführen und was bei dessen Bearbeitung zu beachten ist. Weiterhin werden hier Dokumente mit der datenbezogenen Perspektive eines

aktuell ausgeführten Prozesses verknüpft. Es stehen aber auch die modellierten Werkzeuge der operationalen Perspektive zur Auswahl, um die geforderten Daten zu bearbeiten. Eine genaue Beschreibung der Arbeitsfläche ist in [23] speziell in Kapitel 4 dargestellt.

Die Berechnung, welche Prozesse in die Aufgabenliste aktuell aufgenommen werden und welche nicht, basiert auf dem geladenen Prozessmodell sowie dem aktuellen Zustand der Ausführung. Das heißt, **ESP**ro**N**a teilt dem ProcessNavigator mit, welche Entscheidungen durch den Nutzer getroffen werden können. Konkret bedeutet dies, dass nur Prozesse zur Ausführung zur Verfügung stehen, die kein modelliertes Prozessconstraint verletzen. Aus der Worklist kann der Mitarbeiter nun den Prozess auswählen, mit dem er beginnen möchte. Nach der Auswahl eines Schrittes kann er als abgearbeitet und nachfolgend als abgeschlossen im System hinterlegt werden. Basierend auf dieser Information erscheinen jetzt in der Worklist möglicherweise andere Prozesse, die vorher nicht ausführbar waren. **ESP**ro**N**a hat dann ermittelt, dass die Voraussetzungen für die Ausführung dieser Prozesse jetzt erfüllt sind, nachdem der eben ausgeführte Prozess abgeschlossen ist. Es kann aber auch sein, dass in der Worklist eines anderen Agenten neue Aufgaben erscheinen, da diese jetzt durch **ESP**ro**N**a freigegeben wurden.

### 2.3.2 Navigation

Die Implementierung der Prozessnavigation ist konzeptionell gesehen der Ebene von **ESP**ro**N**a zuzuordnen. Sie implementiert den Kern des Navigationssystems, der ProcessNavigator stellt diese Information grafisch dem Endnutzer dar. Wie bereits angemerkt, kann es situationsbedingt sein, dass Personen, die zur Abarbeitung des Prozessmodells benötigt werden, nicht anwesend sind (etwa urlaubs- oder krankheitsbedingt). Nun soll allerdings vermieden werden, dass diesen Personen Aufgaben in ihre Worklist zugeteilt werden. Die Ausführung aller anderen Prozesse würde sich aufgrund von deren Abwesenheit verzögern. Mit Hilfe des Navigationssystems kann man diese Situationen erkennen, um die Strukturen eines Unternehmens zu verbessern. Oft ist nicht von vornherein bekannt, dass bei Ausfall einer bestimmten Person Probleme entstehen, sondern erst während der Ausführung von Prozessen treten diese auf. Dann ist es aber meist schon zu spät. Mit Hilfe des Navigationssystems kann man solche Situationen vermeiden, da Prozessabläufe simuliert werden können. Ähnlich dem Navigationssystem im Auto können Problemmeldungen wie etwa eine krankheitsbedingte Verzögerung der Ausführung (ähnlich einem Stau) in den berechneten Pfad eingeblendet werden. Der Benutzer kann diese erkennen und ein Umleitung planen. Im obigen Fall würde dies bedeuten, dass der abwesenden Personen momentan keine Aufgaben zugeteilt werden.

## 2.4 Zusammenfassung

In diesem Kapitel wurde das Zusammenspiel zwischen der Prozessmodellierung und der Prozessausführung dargestellt. Dabei erfolgte eine Trennung zwischen der Modellierung und der Ausführung von Geschäftsprozessen. In den nächsten Kapiteln werden, basierend auf dieser Architektur, die einzelnen Komponenten im Detail besprochen und können so besser in das Gesamtkonzept eingeordnet werden.

# Kapitel 3

# Konzeption

Ziel dieses Kapitels ist es, die Konzeption der deklarativen PMS **ESP**ro**N**a genauer zu erläutern. Das Grundkonzept, zum einen native Flexibilität zu gewährleisten und zum anderen eine Trennung zwischen der Problemstellung und der Lösung des Problems zu erreichen, steht dabei an oberster Stelle. Wie bereits in Kapitel 1.4 erwähnt, soll dabei unter anderem ein Konzeptwechsel von „Was nicht explizit erlaubt ist, ist verboten“ zu „Was nicht verboten ist, ist erlaubt“ erreicht werden. Nur falls eine Regel die Ausführung explizit verbietet, wird diese unterbunden. Somit ist sichergestellt, dass sich der Endnutzer an die Regeln des Unternehmens hält und keine davon verletzt, trotzdem in seiner Entscheidungsfreiheit aber nicht eingeschränkt wird. Eine weitere Vorgabe ist, dass strikte Ausführungsreihenfolgen (es gibt an jedem Punkt immer nur genau eine weitere Entscheidungsmöglichkeit für den Endnutzer) genauso modelliert werden können wie auch flexible Ablaufszenarien. Diese Breite ist notwendig, um eine Akzeptanz der Modellierungssprache für verschiedenste Unternehmensgegebenheiten zu erreichen.

## 3.1 Deklaratives Programmieren

Der Einstieg in die Welt der deklarativen Programmierung erfolgt beispielhaft anhand des Logikrätsels Sudoku, das hier als Anwendung in der deklarativen Sprache Prolog verfasst ist. Wichtig dabei ist, dem Leser die strikte Trennung zwischen den Regeln des Spiels und dem Lösen des Problems zu verdeutlichen. Aus diesem Beispiel heraus erfolgt die Konzeption für die Modellierung von deklarativen Geschäftsprozessen. Eine klare Trennung der Prozessperspektiven wird ebenso definiert wie die Trennung von Prozess und Zustand. Des Weiteren werden feingranulare Kategorisierungsmöglichkeiten der Prozessregeln eingeführt.

In der Informatik beschreiben Algorithmen allgemein die Lösungen zu ganz bestimmten Problemen. Nach der Definition in [15] ist ein Algorithmus die Kombination aus Logik und Kontrolle (Algorithm = Logic + Control). Oft wird in diesem Kontext die Logik mit der Problemstellung assoziiert, also WAS ist das Problem. Die Kontrolle wird mit der Umsetzung bzw. Lösung des Problems gleichgesetzt, dem WIE. In imperativen Programmiersprachen sind beide Konzepte miteinander vermischt, sodass keine explizite Trennung zwischen dem WAS (Logik) und dem WIE (Kontrolle) zu erkennen ist. Im Kontext von deklarativen Programmiersprachen wird nur das WAS, also die Regeln des Spiels, angegeben. Die Lösung des Problems (das WIE) übernimmt der Interpreter der Programmiersprache. Um die explizite Trennung zwischen der Problemstellung und der

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | | | 7 | | | | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | 2 | 8 | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | A3 | A4 | 7 | A6 | A7 | A8 | A9 |
| 6 | B2 | B3 | 1 | 9 | 5 | B7 | B8 | B9 |
| C1 | 9 | 8 | C4 | C5 | C6 | C7 | 6 | C9 |
| 8 | D2 | D3 | D4 | 6 | D6 | D7 | D8 | 3 |
| 4 | E2 | E3 | 8 | E5 | 3 | E7 | E8 | 1 |
| 7 | F2 | F3 | F4 | 2 | F6 | F7 | F8 | 6 |
| G1 | 6 | G3 | G4 | G5 | G6 | 2 | 8 | G9 |
| H1 | H2 | H3 | 4 | 1 | 9 | H7 | H8 | 5 |
| I1 | I2 | I3 | I4 | 8 | I6 | I7 | 7 | 9 |

Abbildung 3.1: Sudoku-Beispiel (links) mit ungebundenen Variablen A bis I (rechts)

Lösung an einem konkreten Beispiel aufzuzeigen, wird hierfür – wie oben bereits erwähnt - das Spiel Sudoku herangezogen.

Sudoku (im Japanischen bedeutet das in etwa so viel wie „Isolieren Sie die Zahlen“) ist ein Logikrätsel und ähnelt den „Magischen Quadraten“. Ausgangspunkt ist ein Gitter, in dem bereits mehrere Ziffern vorgegeben sind. Das Rätsel wurde von Howard Garns, einem Amerikaner, entwickelt. 1979 unter dem Namen „NumberPlace“ erstmals in einer Rätselzeitschrift veröffentlicht, wurde es ab 1986 zuerst in Japan populär, wo es auch seinen heutigen Namen Sudoku erhielt. Abbildung 3.1 zeigt ein solches Sudoku-Rätsel, das hier im weiteren Verlauf benutzt und gelöst werden soll. Man erkennt in der Abbildung die teilweise gefüllte 9×9-Matrix mit den 3×3-Unterquadraten. Die Regeln sind einfach zu beschreiben: Ziel ist es, das 9×9-Gitter mit den Ziffern 1 bis 9 so zu füllen, dass jede Ziffer in jeder Spalte, in jeder Zeile und in jedem Block (dick umrandete 3×3-Unterquadrate) nur genau einmal vorkommt.

```
?- sudoku(
    5, 3, A3, A4, 7, A6, A7, A8, A9,
    6, B2, B3, 1, 9, 5, B7, B8, B9,
    C1, 9, 8, C4, C5, C6, C7, 6, C9,
    8, D2, D3, D4, 6, D6, D7, D8, 3,
    4, E2, E3, 8, E5, 3, E7, E8, 1,
    7, F2, F3, F4, 2, F6, F7, F8, 6,
    G1, 6, G3, G4, G5, G6, 2, 8, G9,
    H1, H2, H3, 4, 1, 9, H7, H8, 5,
    I1, I2, I3, I4, 8, I6, I7, 7, 9).

A3 = 4, A4 = 6, A6 = 8, A7 = 9, A8 = 1, A9 = 2,
B2 = 7, B3 = 2, B7 = 3, B8 = 4, B9 = 8,
C1 = 1, C4 = 3, C5 = 4, C6 = 2, C7 = 5, C9 = 7,
D2 = 5, D3 = 9, D4 = 7, D6 = 1, D7 = 4, D8 = 2,
E2 = 2, E3 = 6, E5 = 5, E7 = 7, E8 = 9,
F2 = 1, F3 = 3, F4 = 9, F6 = 4, F7 = 8, F8 = 5,
G1 = 9, G3 = 1, G4 = 5, G5 = 3, G6 = 7, G9 = 4,
H1 = 2, H2 = 8, H3 = 7, H7 = 6, H8 = 3,
I1 = 3, I2 = 4, I3 = 5, I4 = 2, I6 = 6, I7 = 1.
```

Listing 3.1: Aufruf und Ergebnis des Algorithmus

Im rechten Teil der Abbildung sind an den freien Plätzen der Matrix ungebundene Variablen platziert (grün markiert). Ziel ist es nun, an die jeweiligen Variablen eine

Zahl aus der Zahlenmenge 1..9 so zu binden, dass alle Belegungen zusammengenommen eine Lösung ergeben. Listing 3.1 zeigt, wie man den in Listing 3.2 abgebildeten Programmcode aufrufen kann (Zeilen 1 bis 10). An die Stelle der unbelegten Plätze wandern freie Variablen (`A1-A9, B1-..., I1-I9`), die im Quellcode fett markiert sind. An diese werden dann die Lösungen gebunden. Alles, was der Programmierer zu tun hat, ist die Regeln des Spiels zu modellieren. In Listing 3.2 ist der Prolog-Quellcode abgebildet, der den Logikteil (das WAS) der Problemstellung darstellt. Es sind sehr viele Regeln enthalten (jede Spalte, jede Zeile und jeder Block wird durch eine Regel modelliert); der Code könnte wesentlich effizienter dargestellt werden, jedoch ist er auf diese Weise gerade für Neulinge im Bereich der deklarativen bzw. logischen Sprachen einfacher zu verstehen. Die Zeilen 1 bis 10 geben den sogenannten Kopf des Prädikates an. Durch diese Signatur wird festgelegt, wie man dieses Prädikat aufrufen kann. Die Variablen werden teilweise vorbelegt, nämlich mit den Werten des vorgegebenen Rätsels (siehe Listing 3.1). So wird beispielsweise an die Variable `A1` der Wert 5 gebunden, an die Variable `A2` der Wert 3, an die Variable `A5` der Wert 7 etc. Die ungebundenen Variablen (`A3, A4, A6, ...`) werden seitens des Prolog-Interpreters automatisch so durchprobiert, dass sie eine Lösung gemäß den angegebenen Regeln ergeben. Die Zeilen 12 bis 20 des Listings 3.1 spiegeln das Ergebnis wider, das der Prolog-Interpreter ausgibt.

```
1  sudoku( A1,A2,A3, A4,A5,A6, A7,A8,A9,
2          B1,B2,B3, B4,B5,B6, B7,B8,B9,
3          C1,C2,C3, C4,C5,C6, C7,C8,C9,
4
5          D1,D2,D3, D4,D5,D6, D7,D8,D9,
6          E1,E2,E3, E4,E5,E6, E7,E8,E9,
7          F1,F2,F3, F4,F5,F6, F7,F8,F9,
8
9          G1,G2,G3, G4,G5,G6, G7,G8,G9,
10         H1,H2,H3, H4,H5,H6, H7,H8,H9,
11         I1,I2,I3, I4,I5,I6, I7,I8,I9) :-
12
13         /* Alle Zahlen pro Zeile sind verschieden */
14         all_vars_different([A1,A2,A3,A4,A5,A6,A7,A8,A9]),
15         all_vars_different([B1,B2,B3,B4,B5,B6,B7,B8,B9]),
16         ...
17         all_vars_different([I1,I2,I3,I4,I5,I6,I7,I8,I9]),
18
19         /* Alle Zahlen pro Spalte sind verschieden */
20         all_vars_different([A1,B1,C1,D1,E1,F1,G1,H1,I1]),
21         all_vars_different([A2,B2,C2,D2,E2,F2,G2,H2,I2]),
22         ...
23         all_vars_different([A9,B9,C9,D9,E9,F9,G9,H9,I9]),
24
25         /* Alle Zahlen pro 3x3-Block sind verschieden */
26         all_vars_different([A1,A2,A3,B1,B2,B3,C1,C2,C3]),
27         all_vars_different([A4,A5,A6,B4,B5,B6,C4,C5,C6]),
28         ...
29         all_vars_different([G7,G8,G9,H7,H8,H9,I7,I8,I9]).
30
31
32 all_vars_different(L) :-
33     length(L,9),
34     L ins 1..9,
35     all_different(L).
```

Listing 3.2: Modellierte Regeln des Sudoku-Rätsels in der Sprache Prolog

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 6 | 7 | 8 | 9 | 1 | 2 |
| 6 | 7 | 2 | 1 | 9 | 5 | 3 | 4 | 8 |
| 1 | 9 | 8 | 3 | 4 | 2 | 5 | 6 | 7 |
| 8 | 5 | 9 | 7 | 6 | 1 | 4 | 2 | 3 |
| 4 | 2 | 6 | 8 | 5 | 3 | 7 | 9 | 1 |
| 7 | 1 | 3 | 9 | 2 | 4 | 8 | 5 | 6 |
| 9 | 6 | 1 | 5 | 3 | 7 | 2 | 8 | 4 |
| 2 | 8 | 7 | 4 | 1 | 9 | 6 | 3 | 5 |
| 3 | 4 | 5 | 2 | 8 | 6 | 1 | 7 | 9 |

Abbildung 3.2: Lösung des Sudoku-Beispiels

In den Zeilen 14 bis 17 des Listings 3.2 werden die Regeln für jede Zeile des Spiels einzeln modelliert. Die Zeilen 20 bis 23 geben die Regeln für die Spalten an und die Zeilen 26 bis 29 die Regeln für die 3×3-Blöcke. Das Prädikat `all_vars_different` wird von allen Regeln aufgerufen. Es überprüft, ob die übergebene Liste an Variablen (gebundene und ungebundene zusammen) die Länge neun hat und sichert in Zeile 34 zu, dass alle Werte, die bereits gebunden sind oder noch gebunden werden, im Wertebereich von 1 bis 9 liegen. Das Prädikat `all_different` ist eine systeminternes Prädikat der `clpfd`-Bibliothek, das nun die ganze Arbeit erledigt, das heißt, alle Variablen so durchprobiert, dass sie zusammengenommen alle verschieden sind und eine Lösung ergeben. Wie der Algorithmus des Prolog-Interpreters genau funktioniert, soll hier jedoch nicht näher erläutert werden. Dem interessierten Leser sei [15] empfohlen, wo sich ein sehr guter, mit vielen Beispielen untermauerter Einstieg in die Sprache Prolog findet.

## 3.2 Deklarative Prozesse

Wie in Kapitel 1 bereits erwähnt, steht das Beschreiben der Problemstellung im Vordergrund. Zu diesem Zweck wählt man einen deklarativen Ansatz zur Darstellung der Unternehmensabläufe. Die Grundidee hierbei ist, die Problemstellung von der Lösung zu separieren. Bezogen auf die Unternehmensprozesse bedeutet dies, dass man in erster Linie die Prozesse und deren Zusammenhänge untereinander beschreibt. Genau wie bei dem angeführten Sudoku-Beispiel soll man sich bei der Prozessmodellierung nur auf die Regeln konzentrieren und sich nicht mehr um eine explizite Lösung dafür bemühen müssen.

### 3.2.1 Prozess und Perspektiven

Bei der Modellierung von Geschäftsprozessen wird in **ESP**ro**N**a ein klarer Fokus auf den Prozess, der in Form eines abgerundeten Rechtecks dargestellt wird, gelegt (vgl. Abbildung 3.3). Dies geschieht durch die regelbasierte Modellierung von Unternehmensabläufen. Die erstellten Regeln werden direkt in den Prozess selbst geschrieben. In Abbildung 3.3 ist der schematische Aufbau eines solchen Prozesses grafisch verdeutlicht: Er gliedert sich in einen Kopfteil, der die Bezeichnung (Prozessname) enthält, und die einzel-

nen Perspektiven. Jeder Prozessschritt hat zusätzlich zur sprachlichen Kurzbeschreibung einen Prozessidentifikator `PID`, der zum einen sicherstellt, dass jeder Prozess eindeutig identifiziert werden kann, und zum anderen als Kurzreferenz in den Regeln dient. Im unteren Teil sind die einzelnen Perspektiven [24] eines Prozesses aufgelistet. Pro Prozess können mehrere Perspektiven (gekennzeichnet durch verschiedene Symbole in den gestrichelten Unterteilungen) modelliert werden. Alle Perspektiven zusammengenommen repräsentieren den Prozess in seiner Gesamtheit. Mit Hilfe dieses Konzeptes werden die Prozessregeln kategorisiert, und es wird eine klare Strukturierung erzielt. Durch das erweiterbare und modulare Konzept können neue Perspektiven einfach und schnell hinzugefügt werden.

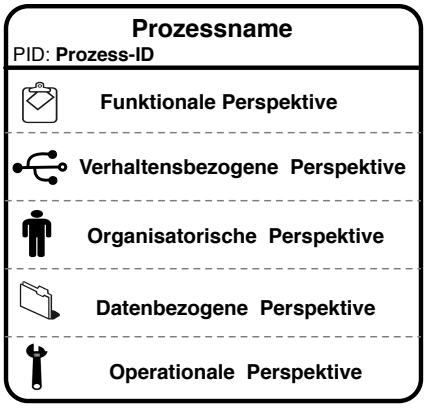



Abbildung 3.3: Schematischer Aufbau eines Prozesses

Insgesamt können aktuell in **ESP**ro**N**a pro Prozess fünf Perspektiven modelliert werden. Im Einzelnen sind dies:

- **Funktionale Perspektive** (Klemmbrettsymbol): beschreibt die individuelle Sicht auf den Prozess selbst. Hier werden Aussagen getroffen, die nur den Prozess selbst betreffen.
- **Verhaltensbezogene Perspektive** (quer liegendes USB-Symbol): beschreibt den speziellen Zustand anderer Prozesse, die für die Ausführung des aktuellen Prozesses wichtig sind. Durch die Auswertung der verhaltensbezogenen Perspektive aller Prozesse ergeben sich der Kontrollfluss bzw. die möglichen Ablaufszenarien eines Prozessmodells.
- **Organisatorische Perspektive** (Personensymbol): beschreibt Personen bzw. allgemein Agenten, die den Prozess ausführen können. Man spricht von Agenten, da auch eine Softwareapplikation einen Prozess ausführen kann.
- **Datenbezogene Perspektive** (Registratursymbol): beschreibt, welche Daten für den Prozess zur Ausführung benötigt werden bzw. welche Daten der Prozess während der Ausführung produziert.
- **Operationale Perspektive** (Werkzeugschlüsselsymbol): beschreibt, welche Programme, Maschinen, etc. im Prozess benötigt werden.

Bei der Validierung bezüglich der Ausführbarkeit eines Prozesses erfolgt eine logische UND-Verknüpfung aller Perspektiven. Abbildung 3.4 zeigt dies schematisch. Dabei werden, wie in der Abbildung unten rechts zu sehen ist, verschiedene Ergebnisse berechnet. Das Ergebnis der funktionale Perspektive sagt aus, ob eine weitere Instanz des Prozesses ausgeführt werden kann, und wird in Abbildung 3.4 durch das Klemmbrettsymbol symbolisiert. Das zweite Symbol von links symbolisiert das Ergebnis der verhaltensbezogenen Perspektive. Auch hier wird durch Auswertung von speziellen Prozessregeln überprüft, ob aus Sicht dieser Perspektive der Prozess ausgeführt werden kann. Das Personensymbol beschreibt das Ergebnis aus der Validierung der organisatorischen Perspektive. Hier wird der Agent ermittelt, der für die Ausführung des Prozesses zuständig ist. Mit dem Registratursymbol wird die datenbezogene Perspektive assoziiert. Es beschreibt die für den Prozess benötigten Daten. Das Symbol des Werkzeugschlüssels definiert allgemein die Applikation, mit der ein Prozess auszuführen ist.

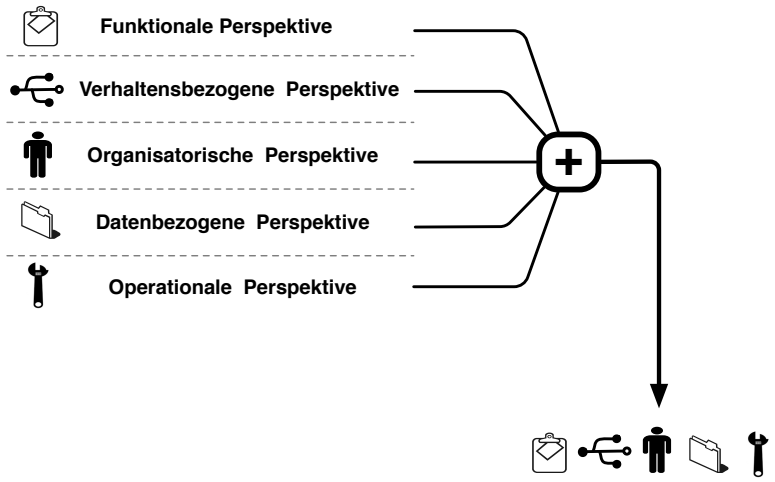



Abbildung 3.4: Auswertung aller Perspektiven

Eine Erweiterung der PMS mit neuen Perspektiven ist ohne größeren Aufwand möglich. Abbildung 3.5 zeigt schematisch, wie eine neue Perspektive zur Validierung mit eingebunden wird. In Kapitel 6.2 wird dies auf technischer Ebenen genauer erläutert. Es sei vorweggenommen, dass die neu hinzugekommene Perspektive lediglich implementiert und in die Prozessbeschreibung importiert werden muss. Das dadurch neu hinzugekommene Konzept (symbolisiert durch den Stern) wird bei bei der Validierung, wie in Abbildung 3.5 dargestellt, mitberechnet.

### 3.2.2 Detaillierter Aufbau eines Prozesses

In Abbildung 3.6 ist der detaillierte Aufbau eines Prozesses dargestellt. Innerhalb der Perspektiven sind nun im Gegensatz zu Abbildung 3.3 zusätzliche Constraints schematisch abgebildet. Ein Constraint kann zum einen ein Tatsache repräsentieren, also zum Beispiel angeben, wie oft ein Prozess ausgeführt werden kann oder welche Person einen Prozess ausführen muss. Es kann sich aber auch auf andere Prozesse beziehen, das heißt, es modelliert beispielsweise, dass der Agent des Prozesses der Vorgesetzte des Agenten eines anderen Prozesses sein muss. Zum anderen kann es auch eine Regel repräsentieren,

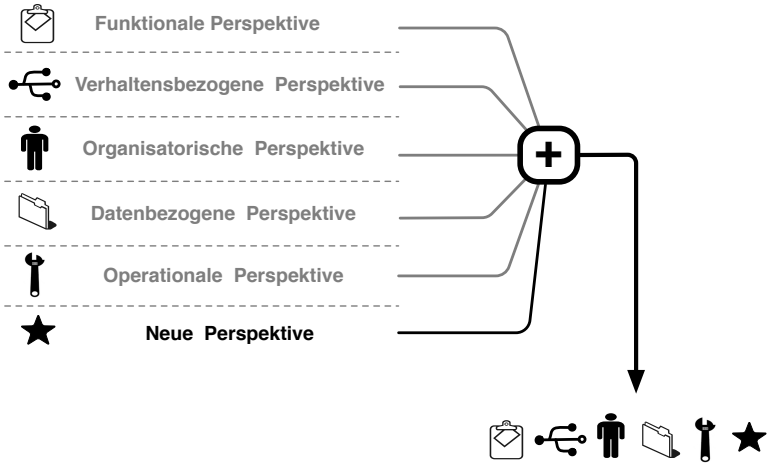



Abbildung 3.5: Hinzufügen einer neuen Perspektive mit Auswertung des neuen Konzepts

also eine Wenn-Dann-Beziehung darstellen. In diesem Fall bezieht man sich auf einen bestimmte Situation (Wenn), und abhängig von dieser Situation müssen bestimmte Dinge eintreten (Dann). Eine Regel könnte etwa lauten: Wenn der Prozess mehr als dreimal abgebrochen wurde, so kann er nur noch vom Vorgesetzten der Abteilung wieder ausgeführt werden. Die genaue Syntax der Constraints der entsprechenden Perspektiven ist in Anhang C spezifiziert.

Auf der rechten Seite der Grafik in Abbildung 3.6 erkennt man, welche Perspektive Einfluss auf welche Konzepte des Prozesses nimmt. Die Ausführung eines Prozesses wird immer als Prozessinstanz bezeichnet. Wie viele Instanzen es geben kann, wird in der funktionalen Perspektive spezifiziert. Durch die Auswertung der Constraints in dieser Sektion werden die genauen Instanzen eines Prozesses ermittelt, die ausgeführt werden können. Die verhaltensbezogene Perspektive beeinflusst den Kontrollfluss der Ausführung eines Prozessmodells. Durch Auswertung der organisatorischen Perspektive werden der Agent beziehungsweise die Agenten ermittelt, die den Prozess ausführen können. In der datenbezogenen Perspektive werden die Daten zusammengefasst, die zum einen zur Ausführung benötigt werden, zum anderen durch Abarbeitung des Prozesses erzeugt werden. Das Werkzeug hierzu wird durch die operationale Perspektive beschrieben.

### 3.2.3 Trennung von Prozess und Zustand

Um die eingangs erwähnte strikte Trennung zwischen Logik (WAS) und Kontrolle (WIE) zu erreichen, ist es wichtig, den Zustand eines Prozesses explizit auszuweisen, also vom Prozess zu entkoppeln. Dies hat den Vorteil, dass hierdurch die in Kapitel 7 angesprochene Prozessnavigation sehr einfach umzusetzen ist. Ein weiterer Vorteil betrifft die Kommunikation mit dem ProcessNavigator. Durch den ausgelagerten Zustand eines Prozesses ist jede Anfrage vom ProcessNavigator an **ESP**ro**N**a in sich geschlossen, das heißt, sie beinhaltet alle Informationen über den Anwendungszustand. Hierdurch werden Client/Server-Anfragen zwischen den Systemen effizienter, da etwa zusätzliche Transportschichten wie SOAP [25] oder Sitzungsverwaltungen über HTTP-Cookies [26] nicht nötig sind.

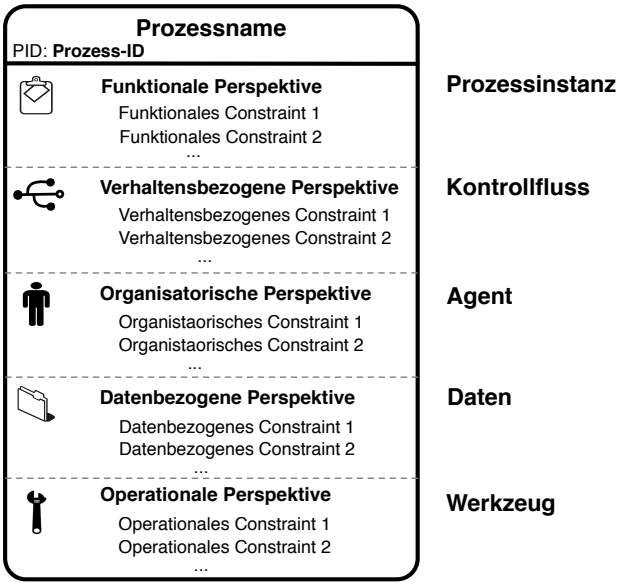



Abbildung 3.6: Deklarativer Prozess im Detail

In den einzelnen Zuständen der im Modell enthaltenen Prozesse werden Informationen über deren Historie gespeichert. Diese beinhaltet Informationen darüber, welche detaillierten Aktionen auf einen Prozess ausgeführt worden sind. Die Struktur dieser Informationen gliedert sich wieder nach den POPM-Perspektiven; Instanz, Agent, Daten und verwendete Werkzeuge spiegeln sich also darin wider. Durch diese detaillierte Speicherung können Prozessconstraints modelliert werden wie etwa „Jede zweite Ausführung eines Prozesse muss vom Vorgesetzten erledigt werden“. Um den Vorgesetzten zu ermitteln, ist es wichtig zu wissen, welche Instanz gerade ausgeführt wird und wer die vorangegangene ausgeführt hat. All diese Informationen sind in der Historie abgespeichert.

## 3.3 Kategorisierung von Prozessconstraints

Pro Perspektive können beliebig viele Constraints eingetragen werden. Um eine möglichst detaillierte Abbildung der Unternehmensabläufe zu bekommen, hat sich eine Kategorisierung der Prozessregeln nach POPM-Perspektiven [24], Prozesshandlungen und Prozesszuständen als essentielles Konzept erwiesen. Das wird in den nachfolgenden Abschnitten näher erklärt.

### 3.3.1 POPM-Perspektiven

Die verschiedenen Prozessregeln, die zusammengenommen die Definition eines Prozesses ausmachen, werden primär nach den einzelnen POPM-Perspektiven kategorisiert. In der organisatorischen Perspektive beispielsweise werden nur Regeln abgebildet, die den Agenten betreffen, der den Prozess ausführen kann. In der operationalen Perspektive ist es sinnvoll, Regeln anzugeben, die das Werkzeug spezifizieren, mit dem man den Prozess ausführen möchte. Dieses Konzept der perspektivenorientierten Kategorisierung hat

zum einen Vorteile bei der Lesbarkeit der Prozessmodelle, zum anderen können aber auch ganze Perspektiven ausgeblendet werden. Dies kann nötig sein, wenn beispielsweise in einem Unternehmen gewünscht wird, keine operationale Perspektive zu modellieren. In diesem Fall würde man sie einfach dadurch weglassen, dass keine Regeln in dieser Perspektive modelliert werden.

Es kommt manchmal vor, dass der Modellierer Regeln angeben möchte, die zwei oder mehrerer Perspektiven betreffen. Eine davon könnte - in sprachlicher Form ausgedrückt - etwa lauten: „Der Chefarzt der chirurgischen Abteilung wird die Erstellung des Operationsplans selbst durchführen, wenn die Daten des Patienten zeigen, dass dieser privat versichert ist“. Diese Regel betrifft übergreifend zwei verschiedene Perspektiven. In diesem Fall hat der Modellierer die Möglichkeit, die Regel entweder der organisatorischen oder der datenbezogenen Perspektive zuzuordnen.

### 3.3.2 Prozesshandlungen

Prozesshandlungen wurden bisher immer unter dem Begriff „Aktion“ geführt und ein Prozess immer „ausgeführt“. Eine genauere Differenzierung ist dabei bisher aber vermieden worden. Ab jetzt können ganz individuell verschiedene Aktionen auf einen Prozess definiert werden. Das „Ausführen“ wird in verschiedene Aktionen aufgeteilt. Grundlegende Handlungen, etwa das Starten, Abschließen oder auch das Abbrechen eines Prozesses, werden in der funktionalen Perspektive des Prozesses als Constraint modelliert. Abbildung 3.7 zeigt dies grafisch.

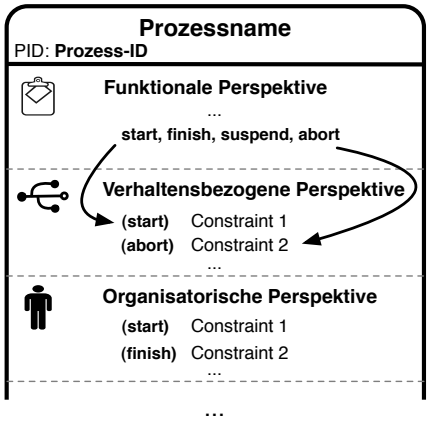



Abbildung 3.7: Definition der möglichen Handlungen eines deklarativen Prozesses

Die Liste der Handlungen kann von Unternehmen zu Unternehmen verschieden sein, sodass es ein grundlegendes Ziel bei der Entwicklung von **ESP**ro**N**a war, sie dynamisch zu gestalten. Dabei ging man sogar noch einen Schritt weiter: Die Menge an möglichen Handlungen pro Prozess kann individuell eingeschränkt werden. Ein Grund für diese Konzeption war, dass es zum Beispiel bei einem Prozess Sinn machen kann, ihn zu suspendieren, sozusagen „schlafenzulegen“, bei einem anderen Prozess aber nicht. Als Konsequenz aus dieser Tatsache erlaubt **ESP**ro**N**a, pro Prozess eine Liste von Handlungen anzugeben, die auf den Prozess ausgeführt werden können. In Abbildung 3.7 sind vier

Handlungen aufgeführt, die allerdings nicht vollständig sind, sondern beliebig erweitert werden können.

Bei der Spezifikation eines Prozesses in iPM$^2$ kann man sich also entscheiden, ob man die globalen Standardhandlungen des Prozessmodells erlauben oder ob man diese noch individualisieren möchte. Somit erreicht man insgesamt eine sehr feingranulare Dosierbarkeit der Prozessregeln. Man kann jetzt bei jeder Regel angeben, ob sie für alle Handlungen des Prozesses (kein Prefix vor der Regel) oder nur für eine ganz spezielle Handlung eines Prozesses (Prefix mit Namen der Handlung vor der Regel) gelten soll.

Abbildung 3.7 zeigt, wie die Kategorisierung der Constraints erfolgen kann. Hier wird das `Constraint 1` der verhaltensbezogenen Perspektive nur für die Handlung `start` aktiviert. Das `Constraint 2` gilt nur für `abort`. Wird zum Zeitpunkt der Ausführung des Prozesses überprüft, welche Handlungen ausgeführt werden können, so werden nur diejenigen Constraints in Erwägung gezogen, die entsprechend kategorisiert sind. Prozessconstraints ohne Kategorisierung werden - wie bereits angemerkt - für alle Handlungen evaluiert.

### 3.3.3 Prozesszustände

Durch die weitere Kategorisierung der Prozessregeln, basierend auf dem Zustand eines (anderen) Prozesses, kann man die Flexibilität in Bezug auf die Ausführbarkeit eines Prozessschrittes noch zusätzlich individualisieren. Weiterhin ist es durch die Referenzierung von Zuständen innerhalb der Constraints möglich, Entscheidungsfälle zu modellieren. Eine Regel könnte etwa lauten: „Wurde der Prozess zweimal abgebrochen, so kann er nur vom Vorgesetzten der Person wieder gestartet werden, die ihn abgebrochen hat.“ Somit ist es möglich, einen eventuell eintretenden Sonderfall (zweimaliges Abbrechen des Vorgangs) innerhalb des Prozesses zu modellieren und nicht wie bei herkömmlichen Modellierungssprachen durch Entscheider und Verzweigungen außerhalb des Prozesses. Dies vereinfacht das Erstellen und die Übersichtlichkeit von Prozessmodellen.

### 3.3.4 Prozessinstanzen

Die höchste und zugleich auch feingranularste Ebene, mit der man ein Prozessconstraint kategorisieren kann, stellt die der Prozessinstanz dar. Eine Prozessinstanz spiegelt eine Ausführung (erfolgreich oder auch nicht) eines Prozesses wider und kann verschiedene Zustände haben: also gestartet, beendet oder auch abgebrochen. Der Wertebereich des funktionalen Constraints entscheidet demnach, wie viele Instanzen es von einem Prozess geben kann. Basierend auf Prozesshandlung und Zustand kann man nun ein Constraint einer bestimmten Perspektive soweit einschränken, dass dieses nur für eine bestimmte Instanz gilt. Ein Beispiel für ein solches Constraint wäre: „Jede zweite Instanz eines Prozesses muss immer vom Vorgesetzten beendet werden“. In Kapitel 4.2.1 wird näher erklärt, wie man die mögliche Anzahl an Instanzen eines Prozesses modellieren kann.

## 3.4 Zusammenfassung

Ziel dieses Kapitels war es, dem Leser die Konzepte, die zur Implementierung der deklarativen PMS **ESP**ro**N**a beigetragen haben, näher zu erläutern. Durch den deklarativen Ansatz kann jetzt eine strikte Trennung zwischen Problemstellung und Lösung

erreicht werden. Hierdurch muss die Lösung, das heißt, der Ablauf der Unternehmensprozesse nicht mehr explizit modelliert werden, sondern der Modellierer kann sich auf das Wesentliche (den Prozess) konzentrieren. Weiterhin trägt dies zu übersichtlichen und kompakten Prozessmodellen bei. Durch die Kategorisierung der Prozessconstraints nach POPM-Perspektiven, Prozesshandlungen, Zuständen und Instanzen wird weiterhin eine sehr feingranulare und detaillierte Modellierung der Unternehmensprozesse erreicht. Dies ist wichtig, damit diese optimal in ein Prozessmodell umgesetzt und vollständig bis ins Detail abgebildet werden können.

# Kapitel 4

# Modellierung

Ziel dieses Kapitels ist es, die Umsetzung einer verbalen Prozessbeschreibung in ein Modell aufzuzeigen. Aus Sicht eines klinischen Anwendungsfalls wird dargestellt, wie diese informelle Beschreibung eines klinischen Ablaufs in ein Prozessmodell umgesetzt werden kann. Im Anschluss daran wird erklärt, wie dieses Modell in der constraintbasierten PMS **ESP**ro**N**a dargestellt werden kann.

## 4.1 Klinischer Anwendungsfall

Anhand eines klinischen Anwendungsfalles, der in Abbildung 4.1 in ein Prozessmodell umgesetzt wurde und die Prozessschritte `Anamnese durchführen`, `Operationsplan vorbereiten` und `Operationsplan genehmigen` beinhaltet, wird zunächst die Erstellung eines Prozessmodells dargestellt. Die Anamnese wird genau einmal pro Patient von einem Assistenzarzt während der Sprechstunde festgehalten. Dabei werden Daten produziert, die die Krankheit beschreiben, also für eine Diagnose unabdingbare Voraussetzung sind. Diese Daten werden in der Patientenakte abgespeichert. Dabei kommt die HIS-Applikation zum Einsatz.

Aus der Diagnose entwickelt sich dann die Therapie, im dargestellten Fall ist eine Operation notwendig. Man erkennt dies an den nachfolgenden Prozessschritten `Operationsplan vorbereiten` und `Operationsplan genehmigen`. Alternativ können sich aus der Anamnese auch andere Therapien ergeben, zum Beispiel, dass eine Operation nicht notwendig erscheint. Diese Alternativen sind allerdings im vorliegenden Prozessmodell nicht abgebildet. Alle Prozesse beziehungsweise deren Instanzen werden mit dem sogenannten HIS (Hospital Information System) bearbeitet. Dieses System stellt für alle am Prozessmodell beteiligten Personen und im Krankenhaus allgemein die zentrale Schnittstelle zur Dateneingabe dar. Der Prozessschritt `Operationsplan vorbereiten` muss mindestens einmal, kann aber auch beliebig oft von einem Assistenzarzt der Chirurgie ausgeführt werden. Damit dies möglich ist, benötigt man die Daten (die Diagnose) aus der Anamnese. Ein Operationsplan soll bereits vorbereitet werden können, auch wenn die Anamnese noch nicht beendet ist. Es wird dabei lediglich vorausgesetzt, dass sie bereits gestartet wurde.

Im finalen Schritt `Operationsplan genehmigen` wird der Operationsplan vom Vorgesetzten des Assistenzarztes, einem Oberarzt oder auch dem Chefarzt, überprüft und anschließend genehmigt. Das Ergebnis dieses Prozessschrittes - der genehmigte Operationsplan - kann nun für weitere Schritte als Datenreferenz herangezogen werden. Dieser

Schritt muss sooft ausgeführt werden, wie es erstellte Operationspläne gibt. Ein Abbrechen des Prozesses zur Laufzeit soll nur durch den Chefarzt selbst möglich sein. Damit Operationspläne bereits genehmigt werden können, auch wenn deren Vorbereitung noch nicht abgeschlossen ist, muss lediglich Prozesses `P1` bereits gestartet sein.

## 4.2 Modellbasierte Umsetzung

In den nachfolgenden Unterkapiteln wird dargestellt, wie die verbale Beschreibung des klinischen Ablaufs mithilfe von Prozessregeln umgesetzt werden kann. Abbildung 4.1 zeigt die in iPM$^2$ modellierte grafische Umsetzung der einzelnen Prozessschritte des obigen Anwendungsfalles.

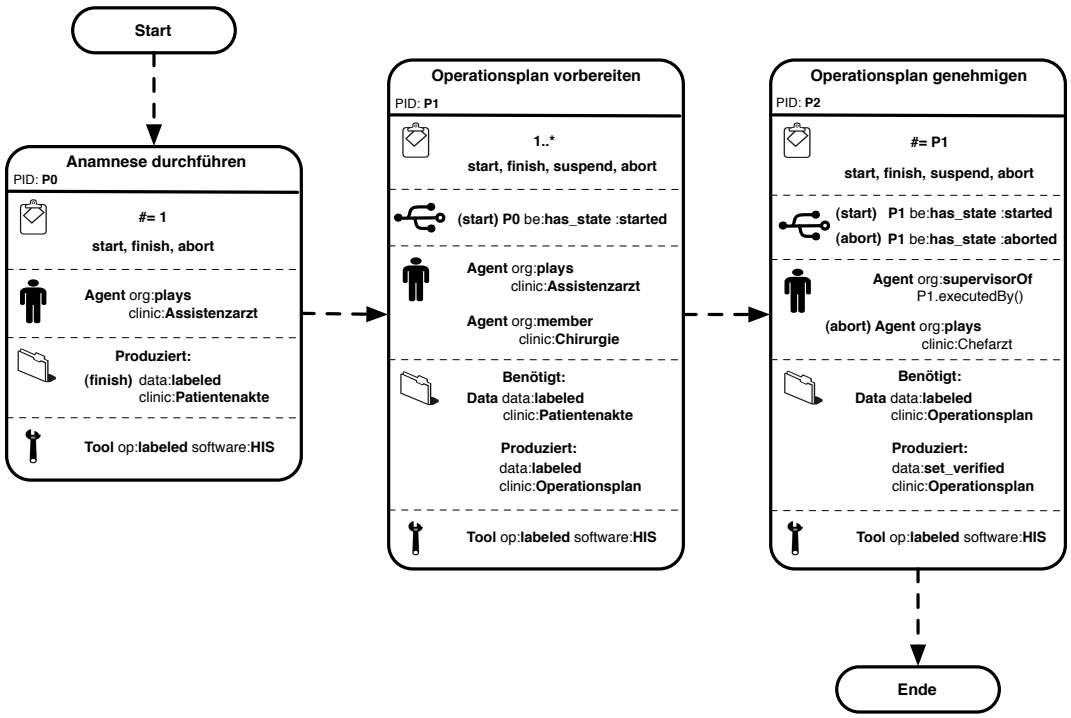



Abbildung 4.1: Prozessmodell für den klinischen Anwendungsfall

### 4.2.1 Funktionale Perspektive

Die funktionale Perspektive, gekennzeichnet durch das Klemmbrettsymbol, beschreibt die Sicht auf den Prozess selbst. Hier werden Constraints angegeben, die nur den Prozess betreffen. Der Ausführungszähler ist ein solches Konzept und beschreibt, wie oft ein Prozess ausgeführt werden kann. Jede laufende Ausführung eines Prozesses bezeichnet man als Prozessinstanz. Im Modell in Abbildung 4.1 ist der Schritt `Anamnese durchführen` soweit eingeschränkt, dass er nur genau einmal ausgeführt werden kann. Es gibt somit genau eine Prozessinstanz. Dies wird mittels der Regel `#= 1` realisiert. Im Prozessschritt `Operationsplan vorbereiten` wurde dagegen modelliert, dass dieser Prozess beliebig

oft ausgeführt werden kann. Konsequenterweise kann es hier auch beliebig viele Prozessinstanzen geben. Alternativen in der Anzahl der möglichen Ausführungen sind mit der Regel `1..*` (siehe Prozessschritt `Operationsplan vorbereiten`) umsetzbar. Hier wird ausgedrückt, dass der Prozess mindestens einmal ausgeführt werden muss, Obergrenze gibt es allerdings keine (modelliert durch das $*$-Zeichen); der Prozess kann somit beliebig oft durchgeführt werden. Im Prozessschritt `Operationsplan genehmigen` soll modelliert werden, dass dieser Prozess genausooft ausgeführt werden muss wie der Prozess `Operationsplan vorbereiten`, das heißt, es sollen genausoviele Prozessobjekte instanziiert werden wie in Prozessschritt `Operationsplan vorbereiten`. Hiermit wird erreicht, dass jeder vorbereitete Operationsplan auch genehmigt werden muss. Dies wird durch die Zuweisungsregel `#= P1` realisiert. `#= P1` wird in diesem Kontext als Alias für den Prozessschritt `Operationsplan vorbereiten` benutzt. Man ist folglich unabhängig von der exakten Anzahl der Ausführungen des Prozessschritts `P1` während der Laufzeit und kann diese auch sehr viel einfacher modellieren.

### 4.2.2 Verhaltensbezogene Perspektive

Die verhaltensbezogene Perspektive, symbolisiert durch das quer liegende USB-Symbol, beschreibt, welche Voraussetzungen (diese beziehen sich dabei immer auf den Zustand eines anderen Prozesses) für einen Prozess erfüllt sein müssen, damit dieser ausgeführt werden kann. Diese Voraussetzungen berechnen sich aus der Historie eines Prozesses. Im obigen Beispiel beschreibt das Constraint `be:has_state :started`, dass der Prozess `P0` als gestartet markiert sein muss, damit der Prozess `P1` ausgeführt werden kann. Die Syntax der angegebenen Prozessregeln beschreibt einen Dreisatz aus Subjekt, Prädikat und Objekt. Die Angabe der Regeln in diesem SPO-Format (Subjekt, Prädikat und Objekt) hat den Vorteil, dass die angegebenen Sätze bzw. Regeln sowohl für den softwaregestützten Interpreter als auch für den Menschen gut lesbar sind.

Wird also überprüft, ob der Prozess `P0` als gestartet markiert ist, so wird anhand der Historie des Prozesses `Anamnese durchführen` ermittelt, ob es eine gestartete Prozessinstanz gibt. Mit dem Prefix `(start)` vor dem Constraint wird die Überprüfung der Regel auf das Starten des Prozesses eingeschränkt. Bei der Überprüfung der restlichen Handlungen (`finish`, `suspend` oder `abort`) wird das Constraint nicht validiert. Aus Sicht der verhaltensbezogenen Perspektive im Prozess `P1` gibt es für diese Handlungen keine Einschränkungen.

Für den Prozessschritt `Operationsplan genehmigen` gelten in Bezug auf die Handlung `start` die gleichen Regeln wie in Prozess `P1`. Ein zusätzliches zweites Constraint `be:has_state :aborted` mit Prefix `(abort)` stellt sicher, dass im Falle eines Abbruches der Prozess `Operationsplan vorbereiten` bereits abgebrochen wurde.

### 4.2.3 Organisatorische Perspektive

Gekennzeichnet durch das Personensymbol beschreibt die organisatorische Perspektive, welche Agenten einen Prozessschritt ausführen können und welche Bedingungen dabei gelten müssen. Man benutzt den Begriff Agent, da neben einer Person auch ein Computer oder ein Programm einen Prozessschritt (in der Regel automatisiert) ausführen kann. Die Beziehungen, Entitäten zwischen den Agenten sowie zusätzliche Regeln auf diesen Fakten werden in einer Ontologie abgespeichert. Im Falle des Prozessschrittes `Anamnese überprüfen` wird ausgedrückt, dass der Agent, der den Schritt ausführen

kann, ein Assistenzarzt sein muss: `Agent org:plays clinic:Assistenzarzt`. Subjekt ist in diesem Falle der Agent, Prädikat ist ein "Spielen der Rolle" (`org:plays`), und das Objekt bildet die Assistenzarztrolle (`clinic:Assistenzarzt`). Mit dieser Regel, die gleichzeitig auch als Suche in der klinischen Organisationsstruktur genutzt wird, kann man den Agenten in diesem Fall auf diejenigen Personen einschränken, die Assistenzärzte sind. Das Subjekt `Agent` stellt in diesem Kontext eine Variable dar (siehe Sudoku-Beispiel in Kapitel 3.1), und es werden alle Personen daran gebunden, die die Rolle des Assistenzarztes ausüben.

Bei der Auswertung einer Regel der organisatorischen Perspektive wird immer wenigstens ein Agent ermittelt. Es können aber auch mehrere Agenten sein, die als Kandidaten für die Ausführung des Prozesses in Frage kommen. Es sind diejenigen Personen, die sich später einmal am ProcessNavigator mit ihren Zugangsdaten anmelden, um den entsprechenden Prozessschritt auszuführen. In Prozessschritt `P1` (`Operationsplan vorbereiten`) findet man die gleichen Regeln wie im Prozess `P0`, man sucht also für die Ausführung zum einen nach Assistenzärzten, schränkt dies mit der zweiten Regel jedoch weiter ein: Die gefundenen Assistenzärzte müssen noch zusätzlich in der Abteilung Chirurgie arbeiten. Beide Regeln werden zusammen als logische `UND`-Verknüpfung gesehen, es wird also die Schnittmenge aus der durch die erste Regel ermittelten Agentenmenge mit der aus der zweiten Regel gebildet. Neben der `UND`-Verknüpfung von Regeln (die standardmäßig eingestellt ist und nicht explizit angegeben werde muss) sind auch andere logische Verknüpfungen möglich, die jedoch explizit modelliert werden müssen.

In Prozessschritt `P2` ist vorgegeben, dass der Prozess vom Supervisor, also dem Vorgesetzten derjenigen Person ausgeführt werden muss, die den Prozessschritt `P1` (`Operationsplan vorbereiten`) ausgeführt hat. In einem klinischen Umfeld sind dies für die Assistenzärzte in der Regel die Ober- oder Chefärzte einer Abteilung. In dem angegebenen Constraint sollen allerdings nicht die spezifischen Personen ermittelt werden, sondern diese sollen ungebunden bleiben. Mit anderen Worten, es ist nur wichtig, dass hier ein Vorgesetzter den Operationsplan überprüft, nicht jedoch, ob dieser nun ein Ober- oder Chefarzt ist. Konsequenterweise gibt es bei mehreren Vorgesetzten auch mehrere Möglichkeiten, welche Person den Schritt ausführen kann. Dies ist ein Beispiel für die integrierte Flexibilität in **ESP**ro**N**a, ohne dass diese durch mehrere explizite Prozessschritte implementiert werden muss.

Durch das in Klammern vor das nachfolgende Constraint gestellte Prefix `(abort)` wird sichergestellt, dass, falls der Prozessschritt `Operationsplan genehmigen` abgebrochen werden muss, dies von einem Chefarzt ausgeführt wird.

### 4.2.4 Datenbezogene Perspektive

Die datenbezogene Perspektive, symbolisiert durch das Registratursymbol, beschreibt zum einen, welche Daten ein Prozess zur Ausführung benötigt, zum anderen aber auch, welche Daten durch die Ausführung produziert werden. Ähnlich der organisatorischen Perspektive basiert auch diese auf einer Ontologie und beschreibt Daten anhand des bereits erwähnten SPO-Konstrukts. Die praktische Umsetzung der datenbezogenen Perspektive ist allerdings noch in einem sehr frühen Stadium. Hier muss noch sehr viel an Forschungsarbeit geleistet werden: Die Betriebssysteme und deren Applikationen müssen auf die Nutzung einer datenbezogenen Ontologie vorbereitet werden. Nur so kann sichergestellt werden, dass die erzeugten Daten auch interpretiert und vom ProcessNavigator weiterverarbeitet werden können.

### 4.2.5 Operationale Perspektive

Das Werkzeugschlüsselsymbol stellt die operationale Perspektive dar und beschreibt, welche Werkzeuge, also Programme, Maschinen etc., für die Ausführung des Prozessschrittes benötigt werden. Genau wie die organisatorische und die datenbezogene Perspektive wird zur Beschreibung der verschiedenen genannten Entitäten und deren Relationen untereinander eine Ontologie verwendet. Es sei hier noch angemerkt, dass sich die verschiedenen Ontologien auch gegenseitig referenzieren können. Dies ist zum Beispiel der Fall, wenn man beschreibt, welches Programm welche Dateien öffnen kann. Hier nutzt man bei der Beschreibung des Programmes den sogenannten Datentyp (zum Beispiel einfache .txt-Datei, ein Microsoft Word-.docx-Dokument etc.). Die Beschreibung des Typs obliegt allerdings der datenbezogenen Ontologie und wird auch dort gespeichert.

## 4.3 Constraintbasierte Umsetzung

Das nachfolgende Unterkapitel widmet sich speziell dem Thema, wie das Prozessmodell aus Abbildung 4.1 übersetzt werden kann, das heißt, wie die modellierten Prozesse und deren Constraints in **ESP**ro**N**a abgebildet werden können. Das daraus resultierende Prozessmodell besteht nur aus Prozessobjekten und darin enthaltenen Prozessconstraints. Alle anderen grafischen Notationen aus iPM$^2$ sind bei der Übersetzung durch die semantische Modelltransformation in Constraints umgewandelt worden. Anhand des Quellcodes in Listing 4.1 wird das Ergebnis aus dieser Transformation für den Prozess `Anamnese durchführen` aufgezeigt. Die Zeilen 1 bis 2 und die Zeile 59 markieren dabei Beginn und Ende des Prozesses. In dieser Objektdefinition wird als Argument eine Komposition aus der ID des Prozessmodells und dem Prozessidentifikator abgespeichert, um den Prozess `Anamnese durchführen` eindeutig identifizieren zu können (`set_up_surgery_plan#pid_0`). Zeile 4 definiert den Prozess als dynamisch, um das Entladen des Prozesses zu ermöglichen. Dies ist notwendig, damit ein geladenes Prozessmodell zur Laufzeit durch ein anderes Modell ersetzt werden kann. Hierzu müssen alle Prozessobjekte löschbar sein, was durch das Attribut `dynamic` ausgedrückt wird. Die Zeilen 6 und 8 speichern Titel und Beschreibung des Prozesses in Textform. Diese Information wird in der Ausführungsumgebung benötigt, um den Prozess grafisch für den Endnutzer zu kennzeichnen. Zeile 10 zeigt die Zugehörigkeit zum entsprechenden Modell, in dem der Prozess modelliert wurde.

### 4.3.1 Funktionale Constraints

Ab Zeile 12 beginnt der Block, in dem die Constraints der funktionalen Perspektive abgespeichert werden. Die Handlungen, die auf den Prozess `Anamnese durchführen` ausgeführt werden können, sind in Zeile 16 als Liste codiert: `start`, `finish` und `abort`. In den Zeilen 20 und 21 wird der Wertebereich angegeben, in dem sich der Zähler für erfolgreiche Prozessausführungen befinden darf. Im Prozessmodell aus Abbildung 4.1 wurde für den Prozessschritt `Anamnese durchführen` angegeben, dass dieser genau einmal ausgeführt werden darf. Dies muss nun in eine syntaktische Form übersetzt werden, die von **ESP**ro**N**a verstanden werden kann; sie wird als `clpfd`-Domänenbereich codiert. `clpfd` [27] [28] ist eine Bibliothek für SWI Prolog [14], mit der es möglich ist, Einschränkungen auf ganzzahlige Wertebereiche logisch zu beschreiben und auszuwerten. Für das Prozessmodell aus Abbildung 4.1 werden die Constraints `#= 1`, `1..*` und `#= P1` an `clpfd` übergeben.

```
:- object('set_up_surgery_plan#pid_0'(_),
   extends(process)).

    :- dynamic.

    process_title('Anamnese durchführen').

    process_description('Der Prozess Anamnese umfasst die ...').

    contained_in_process_model(set_up_surgery_plan).

    /*-----------------------------------------------------------------
       SECTION FUNCTIONAL PERSPECTIVE CONSTRAINTS
      -----------------------------------------------------------------*/
    /* Deklaration der möglichen Prozesshandlungen */
    process_actions([start, finish, abort]).

    /* Deklaration des Wertebereichs für die Anzahl der
       möglichen Prozessausführungen*/
    process_domain([], (PID_0 #= 1)) :-
        parameter(1, PID_0).

    /* Prozesshandlung kann nur ausgeführt werden, wenn
       keine der handlungsspezifischen Regeln verletzt wird. */
    functional_constraint([Action], State, Instance,
    (
        pid_0(_)::actions_applicable(State, Action, Instance)
    )).

    /*-----------------------------------------------------------------
       SECTION ORGANIZATIONAL PERSPECTIVE CONSTRAINTS
      -----------------------------------------------------------------*/
    /* Der Prozess muss von einem Arzt ausgeführt werden. */
    organizational_constraint([start, finish, abort], _, _, [Agent],
    (
        instantiates_class(Agent, 'org#Person'),
        Agent::'org#plays'('clinic#Arzt')
    )).

    /*-----------------------------------------------------------------
       SECTION DATA PERSPECTIVE CONSTRAINTS
      -----------------------------------------------------------------*/
    /* Prozess produziert die Patientenakte */
    data_production([start], _, _, [Data],
    (
        instantiates_class(Data, 'clinic:Patientenakte')
    )).

    /*-----------------------------------------------------------------
       SECTION OPERATIONAL PERSPECTIVE CONSTRAINTS
      -----------------------------------------------------------------*/
    /* Prozess wird mit Hilfe des HIS-Systems ausgeführt */
    operational_constraint([start, finish, abort], _, _, [Tool],
    (
        instantiates_class(Tool, 'op:Software'),
        Tool::'op:labeled'(software:HIS')
    )).

:- end_object.
```

Listing 4.1: Prozess P0, modelliert in **ESP**ro**N**a

```
1 ?- PID_0 #= 1, PID_1 #>= 1, PID_2 #= PID_1.
2 PID_0 = 1,
3 PID_1 = PID_2,
4 PID_2 in 1..sup.
```

Listing 4.2: Wertebereiche mit Ergebnis, formuliert in `clpfd`-Syntax

Listing 4.2 zeigt in einer Prolog-Session, wie diese Wertebereiche berechnet werden, die für die angegebenen Prozesse gültig sind. In Zeile 1 ist die Anfrage mit den codierten Wertebereichen der Prozesse abgebildet. Da der Prozess `Operationsplan vorbereiten` beliebig oft ausgeführt werden kann (`1..*`), gibt es unendlich viele Kombinationsmöglichkeiten zwischen den Prozessen. Dies erkennt man nochmals an der Ausgabe (Listing 4.2) in Zeile 4. `sup` wird hier als die Abkürzung von Supremum benutzt und bezeichnet plus unendlich ($+\infty$). Die Ergebnisse aus dieser Berechnung werden durch **ESP**ro**N**a im Hintergrund abgespeichert, da sie für die Auswertung der weiteren funktionalen Constraints sehr wichtig sind.

Die nachfolgend besprochenen Constraints ab Zeile 25 in Listing 4.1 weisen alle die gleiche Struktur auf. Diese Konzeption hat sich als vorteilhaft erwiesen, da bei einer Erweiterung der Sprache die Änderungen am Code minimal bleiben und Übersetzungspattern bei der Modelltransformation wiederverwendet werden können. Der erste Parameter eines Constraints schränkt dessen Wirkungsbereich auf ganz bestimmte Handlungen ein, die in Form einer Liste übergeben werden. Nur bei der Validierung der Handlungen, die in der Liste vorkommen, wird das Constraint ausgewertet. Der zweite Parameter ist immer die gebundene Variablen `State`, die den aktuellen Zustand des Prozessmodells speichert (vgl. Kapitel 7.1.1, wo der genaue Aufbau der Zustandsvariablen erläutert wird). Durch Angabe dieses `State` ist es möglich, wie in Kapitel 3.2.3 besprochen, den Wirkungskreis der Regel auf einen ganz bestimmten Zustand einzuschränken. Der dritte Parameter ist die sogenannte Prozessinstanz (`Instance`). Sie kann sowohl gebunden als auch ungebunden verwendet werden. Durch die erstere Form, in diesem Fall wird eine konkrete Instanz spezifiziert und an die Variable gebunden, schränkt man den Wirkungsgrad des Constraints genau auf diese Instanz ein. In der ungebunden Variante (es wird keine spezielle Instanz angegeben) wirkt das Constraint für alle Instanzen des Prozesses.

In den Zeilen 25 bis 28 von Listing 4.1 wird ein funktionales Constraint spezifiziert, das speziell die Handlungen des Prozesses einschränkt. Die Semantik des Constraints besagt, dass die Handlung, die ausgeführt werden soll, auch grundsätzlich ausführbar sein muss. Zwei Regeln schränken diese Ausführbarkeit ein. Die erste Regel betrifft das Starten des Prozesses. Sie besagt, dass es nur möglich ist, einen Prozess zu starten, wenn eine neue Instanz erzeugt werden kann. Die zweite Regel betrifft die restlichen Handlungen, also im obigen Fall `finish` und `abort`. Ein Prozess kann nur dann beendet oder abgebrochen werden, wenn es eine gestartete Instanz gibt. Die Überprüfung der Regeln bei entsprechender Handlung übernimmt das Prädikat `actions_applicable/3` in Zeile 27. Die Spezifikation `/3`, die an den Namen des Prädikats angehängt wird, sagt aus, wie viele Parameter bei Aufruf übergeben werden. **ESP**ro**N**a erhält seitens der Ausführungsumgebung unter anderem den aktuellen Zustand (`State`). Basierend darauf erfolgt die Auswertung der Constraints; sie wird in Zeile 27 an das Prädikat `applicable/3` weitergegeben.

Eine Auswertung des Prädikates `actions_applicable/3` (Listing 4.3) bei geladenem Prozessmodell sieht wie folgt aus: In Zeile 1 wird der initiale Zustand (Variable `State`) durch das Prädikat `initial_state/1` im Objekt `process_planning` erzeugt

(auf dieses Prädikat wird in Kapitel 7.1.1 noch genauer im Zusammenhang mit der Planung bzw. Navigation von Prozessen eingegangen). Im erzeugten Zustand ist noch kein Prozess ausgeführt worden, das Modell befindet sich im Prozess `Start` (Abbildung 4.1). Dieser Zustand wird dann an das Prädikat `actions_applicable/3` in Zeile 27 in Listing 4.1 weitergegeben. Bei der Auswertung des Prädikats wird versucht, an die ungebundenen Variablen `Action` und `Instance` valide Werte zu binden. Für den obigen initialen Zustand, bei dem noch keine Prozessinstanz gestartet wurde, sind dies die Werte `start` für `Action` und `1` für `Instance`. Jede Instanz wird fortlaufend durchnummeriert, ähnlich einem atomaren Primärschlüssel in der Datenbanktechnologie. Wird eine neue Instanz gestartet, so inkrementiert der alte Zähler um eins. Dies passiert aber nur, wenn der Wertebereich des Prozesses dies zulässt. Bei `Anamnese durchführen` wurde modelliert, dass der Prozess genau einmal ausgeführt werden kann. Somit gibt es nur eine Instanz, die gestartet werden kann. Bei Prozess `Operationsplan vorbereiten` ist dies anders: Hier kann es unendlich viele Prozessinstanzen geben (Wertebereich `1..*`). Das Prädikat `actions_applicable/3` prüft nun automatisch, ob eine neue Instanz des Prozesses gestartet werden kann. Im initialen Zustand (es wurde bisher noch keine Instanz gestartet) validiert dieses Prädikat erfolgreich. An die Variable `Action` kann im Initialzustand keine der Handlungen `finish` oder `abort` gebunden werden, da die zweite Handlungsregel verletzt wurde: Eine Instanz kann nur beendet oder abgebrochen werden, wenn es bereits eine gestartete gibt. Dies ist im übermittelten Zustand nicht der Fall, und somit validiert die entsprechende Regel auch nicht.

```
1 ?- process_planning::initial_state(set_up_surgery_plan, State),
2    pid_0(_)::actions_applicable(State, Action, Instance).
3
4 State = [ (pid_0, [], 0), (pid_1, [], 0), (pid_2, [], 0)],
5 Action = start,
6 Instance = 1.
```

Listing 4.3: Aufruf des Prädikats `actions_applicable/3`

### 4.3.2 Verhaltensbezogene Constraints

Ein verhaltensbezogenes Constraint ist im Prozess `Anamnese` nicht vorhanden. Deshalb soll an dieser Stelle das verhaltensbezogene Constraint aus dem Prozess `P1` herangezogen werden: Es spezifiziert, dass für das Starten der Prozess `Anamnese durchführen` bereits gestartet sein muss. Das in der **ESP**ro**N**a-Sprache codierte Constraint ist in Listing 4.4 abgebildet.

```
1 behavioral_constraint([start], State, _,
2 (
3     pid_0(_)::exists_instance(State, start, _)
4 )).
```

Listing 4.4: Verhaltensbezogenes Constraint im Prozess `P1`

Im ersten Parameter der Prozessregel ist zu erkennen, dass das Constraint nur bei der Evaluierung der Handlung `start` ausgewertet wird. Im zweiten Parameter ist der aktuelle Zustand der laufenden Prozessausführung an die Variable `State` gebunden. Eine spezielle Instanz ist für dieses Constraint nicht von Bedeutung, somit wird an die Stelle der Instanzvariablen das Wildcardsymbol (_) eingesetzt. In Zeile 3 wird nun anhand des Prädikates `exists_instance/3` überprüft, ob es vom Prozess `P0` im übermittelten Zustand `State` eine gestartete Instanz gibt. Beim dritten Parameter des Prädikats wäre es möglich, eine bestimmte Instanz zu binden, um zu überprüfen, ob sie gestartet wurde. Diese Information kann nachfolgend für weitere Restriktionen im Constraint benutzt werden. Somit ist das Prädikat `exists_instance/3` in vielerlei Hinsicht einsetzbar. Im obigen Fall dient es lediglich der Existenzquantifizierung: Es prüft, ob es eine gestartete Instanz gibt; ist dies der Fall, so validiert das gesamte Constraint erfolgreich.

### 4.3.3 Organisatorische Constraints

Ein Constraint der organisatorischen Perspektive ist in Listing 4.1 von Zeile 34 bis Zeile 38 spezifiziert. Das Constraint betrifft die Handlungen `start`, `finish` und `abort`, wie im ersten Parameter des Constraints spezifiziert. Der zweite Parameter ist durch eine Wildcard (_) ersetzt worden. Hierdurch wird ausgedrückt, dass das Constraint unabhängig von einem Zustand ist und somit für alle möglichen Zustände des Prozesses `Anamnese durchführen` gilt. Das gleiche Prinzip wird auch bei der Instanz angewandt, das heißt, es gilt auch für alle möglichen Instanzen des Prozesses. Der vierte Parameter des Constraints ist eine Liste, die die ungebundene Variable `Agent` enthält. Sie wird verwendet, da man manche Prozesse soweit einschränken möchte, dass sie von mehreren Personen gleichzeitig ausgeführt werden müssen. In diesem Fall werden mehrere ungebundene Variablen in diese Liste eingefügt.

Folgendes Szenario sei als Beispiel angeführt: Ein Prozess muss immer von einem Angestellten und seinem Vorgesetzten gleichzeitig ausgeführt werden. Man nennt dieses Konzept das n-Augenprinzip. In diesem Fall würde die Liste in Parameter 4 aus zwei Agentenvariablen bestehen. An die erste Variable wird der Angestellte und an die zweite der Vorgesetzte gebunden. Im fünften Parameter des Constraints wird nun der Agent ermittelt, der den Prozess ausführen kann. Im Prozessmodell aus Abbildung 4.1 wurde dieser auf einen Arzt eingeschränkt. In den Zeilen 36 bis 37 werden nun diejenigen Personen ermittelt, die an die Variable `Agent` gebunden werden können. Dabei werden die Daten aus einer Ontologie ermittelt und alle Personen, Rollen, Beziehungen etc., die im Prozessmodell referenziert sind, darin abgespeichert.

Bei der Ermittlung des Agenten, der den Prozess ausführen kann, werden zwei Teilschritte nacheinander ausgeführt: In einer ersten Anfrage (Zeile 36) werden alle entsprechenden Personen aus der Ontologie ermittelt und an die Variable `Agent` gebunden. In einem zweiten Verfeinerungsschritt (Zeile 37) werden diese Personen nochmals eingeschränkt: Die Person muss gleichzeitig auch ein Arzt sein (`org:plays clinic:Arzt`).

Um zu verdeutlichen, wie eine Suche in einer klinischen Ontologie aussehen kann, zeigt Abbildung 4.2, wie klinische Strukturen codiert werden können. Die abgebildete Ontologie stellt nur einen kleinen Auszug aus einer weitaus größeren klinischen Ontologie dar, allerdings ist sie für die Erklärung der Auswertung der organisatorischen Constraints ausreichend. Im linken Teil der Grafik ist die chirurgische Abteilung (`clinic:Chirurgie`) zu sehen, die vom Typ `org:Abteilung` ist, genau wie die kardiologische Abteilung (`clinic:Kardiologie`) auf der rechten Seite. Beide Abteilungen haben Mitarbeiter

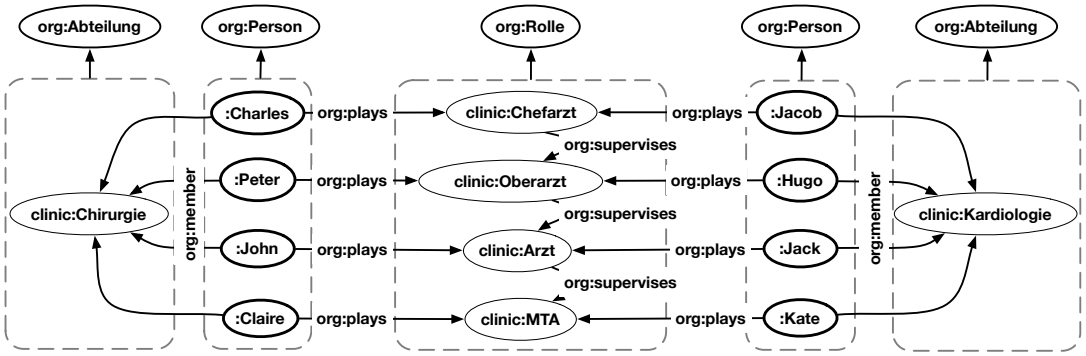



Abbildung 4.2: Auszug aus einer klinischen Ontologie

(`org:member`), die vom Typ Person sind (`org:Person`). In der chirurgischen Abteilung sind dies `Charles`, `Peter`, `John` und `Claire`, in der kardiologischen Abteilung `Jacob`, `Hugo`, `Jack` und `Kate`. Jede Person hat eine gewisse Rolle in der ihr zugeordneten Abteilung. Dies wird durch eine Relation (`org:plays`) zwischen der Person und der Rolle modelliert. Chefarzt, Oberarzt, Arzt und MTA (Medizinisch-technische/r Angestellte/er) sind Beispiele für Rollen, die man in einem Krankenhaus vorfindet. `Charles` wäre im vorliegenden Beispiel der Chefarzt (`clinic:Chefarzt`) der chirurgischen und `Jacob` der der kardiologischen Abteilung. Im organisatorischen Constraint aus dem codierten Prozessmodell in Listing 4.1 werden in einer ersten Anfrage alle Personen ermittelt (Zeile 36). Diese Personen werden nun in einer zweiten Anfrage auf den Kreis der Ärzte eingeschränkt (Zeile 37). Mit Bezug auf die klinische Beispielontologie sind `John` und `Jack` Kandidaten, also diejenigen Personen, die den Prozess `Anamnese durchführen` ausführen können.

### 4.3.4 Datenbezogene Constraints

In der datenbezogenen Perspektive ist primär eine Unterscheidung zwischen der Datenproduktion und der Datenabhängigkeit zu treffen. Erstere spezifiziert, welche Daten der Prozess bei der Ausführung erzeugt. Dies wird im Prozess `Anamnese durchführen` durch ein Faktum in Zeile 44 bis 47 (Listing 4.1) deklariert: Beim Starten des Prozesses wird die sogenannte Patientenakte angelegt. Ein bestimmter Zustand (Parameter 2 des Constraints) oder eine bestimmte Instanz (Parameter 3) werden in diesem Beispiel nicht referenziert. Eine Datenabhängigkeit ist im Prozess nicht vorhanden, weshalb an dieser Stelle das Constraint aus dem Prozessschritt `Operationsplan vorbereiten` referenzieren wird. Listing 4.5 zeigt den Codeausschnitt der Datenabhängigkeit. Hier ist spezifiziert, dass die im Prozessschritt `Anamnese durchführen` erzeugte Patientenakte benötigt wird. Dies ist wiederum unabhängig von einem bestimmten Zustand (Parameter 2) oder einer bestimmten Instanz (Parameter 3). Das Constraint betrifft alle Handlungen, die auf den Prozess ausgeführt werden können (Parameter 1).

### 4.3.5 Operationale Constraints

Im Prozessmodell in Listing 4.1 ist von Zeile 53 bis Zeile 57 ein operationales Constraint modelliert. Es ist auf die Handlungen `start`, `finish` und `abort` (Parameter 1)

```
1 data_constraint([start, finish, abort], _, _, [Data],
2 (
3     instantiates_class(Data, 'clinic:Patientenakte')
4 )).
```

Listing 4.5: Modellierte Datenabhängigkeit im Prozess P1

fokussiert, spezifiziert aber keinen bestimmten Zustand oder keine bestimmte Instanz (Wildcards in Parameter 2 und 3). In einem ersten Schritt wird in der Ontologie nach allen Software-Applikationen gesucht (Zeile 55), was dann auf die HIS-Anwendung eingeschränkt wird (Zeile 56).

```
1 instantiates_class(Tool, 'op:Software'),
2 instantiates_class(DataType, 'data:data_type'),
3 DataType::'data:has_label'('RTF'),
4 Tool::'software:can_read'(DataType).
```

Listing 4.6: Ermittlung aller .rtf-fähigen Textverarbeitungsprogramme

Das Werkzeug, mit dem der Prozess bearbeitet werden kann, muss nicht unbedingt auf ein einziges eingeschränkt werden. So kann man beispielsweise auch eine Anfrage an die operationale Ontologie starten, die alle Applikationen herausfindet, mit denen man ein Datei von Typ .rtf (*Rich Text Format*) öffnen kann. Ein Constraint in **ESP**ro**N**a ist in Listing 4.6 abgebildet. Durch die Anfrage werden mehrere Programme ermittelt. Der Prozessausführende kann sich dabei aus der Liste der ermittelten Applikationen diejenige aussuchen, mit der er am besten zurechtkommt.

## 4.4 Zusammenfassung

Ziel dieses Kapitels war es, aufzuzeigen, wie die constraintbasierte Modellierung von Prozessen mit **ESP**ro**N**a erfolgen kann. Anhand eines Anwendungsfalls wurde die verbale Beschreibung eines klinischen Ablaufes zuerst in ein grafisches Prozessmodell und nachfolgend in Prozessconstraints umgesetzt. Mit Hilfe dieser Spezifikationen kann im nachfolgenden Kapitel gezeigt werden, wie die Validierung eines Prozesse erfolgt, um zur Laufzeit zu entscheiden, ob dieser ausgeführt werden kann oder nicht.

# Kapitel 5

# Validierung

Ziel dieses Unterkapitels ist es, einen Überblick zu geben, wie die Auswertung der in den **ESP**ro**N**a-Modellen enthaltenen Prozessconstraints erfolgt. Danach wird erklärt, wie dem ProcessNavigator mitgeteilt wird, welche Prozesse durch den Benutzer ausgeführt werden dürfen und welche nicht.

## 5.1 Auswertung der POPM-Prozessconstraints

Bei der Überprüfung, ob ein Prozess zur Laufzeit ausführbar ist oder nicht, erfolgt eine Validierung über alle Perspektiven hinweg. Jede der einzelnen Regeln in den jeweiligen Perspektiven muss erfolgreich validieren. Aus logischer Sicht erfolgt zuerst eine UND-Verknüpfung der einzelnen Regeln. Durch die Auswertung dieser konjunktionalen Verbindung von Prozessconstraints werden die ausführbare Prozessinstanz (funktionale Perspektive), der Agent (organisatorische Perspektive), die benötigten Daten (datenbezogene Perspektive) sowie das Werkzeug (operationale Perspektive) ermittelt, mit denen der Prozess ausgeführt werden kann. Die verhaltensbezogene Perspektive liefert in diesem Zusammenhang keine Daten, sondern überprüft den Zustand der anderen Prozesse. Der Kontrollfluss eines Prozessmodells, also die verschiedenen Ausführungs- und Entscheidungsmöglichkeiten für den Endnutzer, ergibt sich aus der Gesamtheit aller Perspektiven.

Im Folgenden soll nun die Auswertung der Prozessregeln durch **ESP**ro**N**a untersucht werden. Für die einzelnen Handlungen, die auf einen Prozess ausgeführt werden, müssen gewisse Voraussetzungen (sie sind in den Prozessregeln codiert) erfüllt sein. Um die Auswertung dieser Voraussetzungen genauer zu erläutern, ist ein chronologisches Vorgehen sinnvoll: Es wird angenommen, das Prozessmodell aus Abbildung 4.1 soll ausgeführt werden, und seitens des ProcessNavigators wird **ESP**ro**N**a aufgefordert, es zu laden. Nachdem die Syntaxprüfung erfolgt ist und keine Fehler festgestellt wurden, liefert **ESP**ro**N**a der Ausführungsumgebung den initialen Zustand des Prozessmodells zurück. Dieser Initialzustand spiegelt den Zustand im Start-Prozess des Modells wider und wird zusammen mit einer ersten Anfrage an **ESP**ro**N**a übermittelt. Weiterhin kann dieser Zustand abgespeichert werden (Dateisystem, Datenbank, etc.), um zum Beispiel eine Sicherung bei noch nicht abgeschlossenen Prozessmodellen am Ende eines Arbeitstages zu erstellen. Als nächstes kann die Ausführungsumgebung bei **ESP**ro**N**a abfragen, welche Prozesse, basierend auf dem übermittelten Zustand, gestartet werden können. Dabei werden alle Handlungen, basierend auf den modellierten Constraints des jeweiligen Prozesses, über-

prüft. Welche Handlungen prinzipiell ausgeführt werden können, ist in Abbildung 4.1 in der funktionalen Perspektive gekennzeichnet. Bei Prozess `Anamnese durchführen` sind dies `start`, `finish` und `abort`. **ESP**ro**N**a überprüft nachfolgend für die Ausführungsumgebung, welche der drei Handlungen für den Prozess aktiviert werden können. Exemplarisch soll im Folgenden für den Prozess `Anamnese durchführen` die Validierung der Handlung `start` durchgeführt werden.

### 5.1.1 Funktionale Prozessconstraints

In der zeitlichen Abfolge wird als erstes die funktionale Perspektive validiert. Im obigen Prozessmodell ist das Constraint `#= 1` modelliert, das beschreibt, dass der Prozess genau einmal ausgeführt werden kann. Im initialen Zustand des Prozessmodells ist der Zähler für `Anamnese durchführen` auf 0 gesetzt. Dieser Zähler beschreibt die Anzahl an erfolgreichen Ausführungen, die der Prozess durchlaufen hat. Um zu verstehen, wie **ESP**ro**N**a validiert, das heißt, ob der Zähler für die Ausführungen erhöht werden kann oder nicht, ist es wichtig zu wissen, dass aus allen funktionalen Constraints aller Prozesse eine bzw. mehrere Lösung gebildet werden. Im obigen Fall werden die Constraints `#= 1`, `1..*` und `#= P1` einem Algorithmus übermittelt, der sie auswertet, und als Ergebnis verschiedene Wertebereiche zurückgibt, in denen die Zähler der Prozesse liegen dürfen. Im obigen Fall ergibt dies für den Prozess `Anamnese durchführen` der Wert 1, für den Prozess `Operationsplan vorbereiten` einen Wert zwischen 1 und unendlich und für den Prozess `Operationsplan genehmigen` immer den gleichen Wert wie bei Prozess `P1`. Mögliche Kombinationen wären also zum Beispiel `P0 = 1, P1 = 1, P2 = 1`, `P0 = 1, P1 = 2, P2 = 2`, `P0 = 1, P1 = 3, P2 = 3` usw. Da hier noch der initiale Zustand vorliegt und der Prozess `Anamnese durchführen` noch nicht ausgeführt worden ist (der Zähler ist 0), kann er also aus Sicht der funktionalen Perspektive gestartet werden. Auch sind keine weiteren Regeln angeben, die dies verbieten könnten. Nun gilt es, alle weiteren Perspektiven zu validieren.

### 5.1.2 Verhaltensbezogene Prozessconstraints

In der verhaltensbezogenen Perspektive sind im Prozess `Anamnese durchführen` keine Constraints modelliert, somit ist diese Perspektive per se valide (keine Regel ist verletzt).

### 5.1.3 Organisatorische Prozessconstraints

In der organisatorischen Perspektive wurde im Prozess `Anamnese durchführen` modelliert, dass er von einem Assistenzarzt ausgeführt werden muss. Folgendes Constraint spezifiziert dies: `Agent org:plays clinic:Assistenzarzt`. Seitens der Ausführungsumgebung wird bei einer Anfrage (ob der Prozess `Anamnese durchführen` startbar ist) auch die Person mit übermittelt, die sich am System mit ihren Zugangsdaten, bestehend aus Login und Passwort, angemeldet hat. Bei diesen Daten handelt es sich um eine genaue Identifikation der Person bzw. des Agenten. Als nächsten Schritt wird **ESP**ro**N**a anhand der organisatorischen Ontologie überprüfen, ob diese Person darin gespeichert ist und ob sie auch Arzt ist. Ist dies der Fall, so ist auch diese Perspektive valide, und die Person ist autorisiert, den Schritt aus Sicht der organisatorischen Perspektive auszuführen.

### 5.1.4 Datenbezogene Prozessconstraints

Als nächstes soll die datenbezogene Perspektive untersucht werden. Hier muss man zwischen den Daten unterscheiden, die produziert werden, und denen, die für den Prozess benötigt werden. Die datenbezogene Perspektive gliedert sich somit in die Produktion von Daten und in das Benötigen von Daten. Der Prozessschritt `Anamnese durchführen` spezifiziert nur ersteres, er gibt also an, welche Daten beim Ausführen des Schrittes generiert werden. Im obigen Fall ist es die Patientenakte, die erstellt wird. Für eine Evaluierung, ob der Prozess gestartet werden kann oder nicht, ist dieses Faktum nicht von Interesse, da nur benötigte Daten eine Abhängigkeit darstellen und die Voraussetzung zum Starten des Prozesses verhindern können.

Im Prozess `Operationsplan vorbereiten` werden die durch den Prozessschritt `Anamnese durchführen` erzeugten Daten dann als datenbezogene Vorraussetzung spezifiziert. Konkret bedeutet dies, dass der Prozess `P1` nicht ohne diese Daten ausgeführt werden kann. Hier sieht man also eine weitere Abhängigkeit (neben der verhaltensbezogenen) zwischen den Prozessen

### 5.1.5 Operationale Prozessconstraints

Die operationale Perspektive des Prozesses `P1` (`Anamnese durchführen`) modelliert das Werkzeug, mit dem man den Prozess ausführen kann: das HIS (Hospital Information System). Bei diesem System handelt es sich um ein Anwendungsprogramm, das alle informationsverarbeitenden Teilapplikationen zur Bearbeitung medizinischer und administrativer Daten im Krankenhaus zur Verfügung stellt. Ähnlich der organisatorischen Perspektive übermittelt nun die Ausführungsumgebung ihre eigene Identifikation. **ESP**ro**N**a kann nachfolgend diese Information mit dem in der Perspektive modellierten Constraint validieren. Da es sich bei der Ausführungsumgebung um die HIS-Applikation handelt, kann auch diese Validierung semantisch korrekt abgeschlossen werden.

## 5.2 Validierung der Prozesshandlungen

Nachdem die einzelnen Constraints der jeweiligen Perspektiven im Detail erklärt wurden, soll jetzt auf die Validierung eingegangen werden, also welche Handlungen durch den am ProcessNavigator angemeldeten Nutzer ausführbar sind. Hierzu wird das implementierte Prädikat `validate_action/3` genauer untersucht. Es überprüft, ob eine bestimmte Handlung auf einen Prozess ausgeführt werden kann. In 5.1 wurde die Validierung der einzelnen POPM-Perspektiven am Beispiel der Handlung `start` dargestellt. Diese wäre dann bei Aufruf des Prädikates `validate_action/3` an den Parameter `Action` gebunden und wird an die einzelnen Perspektiven weitergegeben, die wiederum jede für sich diese Handlung evaluieren.

Listing 5.1 zeigt den Quellcode des Prädikates `validate_action/3`. Das Prinzip, das in dem Prädikat implementiert ist, ist die logische Evaluierung der einzelnen Perspektiven und Ermittlung der gültigen Prozessparameter. Dabei macht man sich das Konzept der ungebundenen und gebundenen Variablen zunutze, das bereits in Kapitel 3.1 im Sudoku-Beispiel verwendet wurde. Sind konkrete Werte bereits vorgegeben, das heißt, man weiß beispielsweise genau, welcher Agent überprüft werden soll, so bindet man diese Werte fest an bestimmte Variablen. Das Prädikat `validate_action/3` überprüft für diese Werte, ob es eine oder eventuell auch mehrere Lösungskombinationen mit anderen Variablen

```
1  :- public(validate_action/3).
2  :- mode(validate_action(?var(action),
3                          +list(state),
4                          +compound_term(concept_identifiers)),
5          zero_or_more).
6
7  validate_action(Action, State, Instance-Agents-Data-Tools) :-
8      ::process_actions(ActionList),
9      list::member(Action, ActionList),
10     ::functional_constraints_conform(     Action, State, Instance),
11     ::behavioral_constraints_conform(     Action, State, Instance),
12     ::organizational_constraints_conform( Action, State, Instance, Agents),
13     ::data_constraints_conform(           Action, State, Instance, Data),
14     ::operational_constraints_conform(    Action, State, Instance, Tools).
```

Listing 5.1: Prädikat `validate_action/3`

gibt und teilt diese dem Nutzer mit. Bis auf den Zustand kann so mit allen Variablen verfahren werden.

Die Zeilen 1 und 5 sind Teile der sogenannten Signatur des Prädikates. Hier wird unter anderem dessen Zugriff spezifiziert, also von welchen anderen Objekten aus dieses Prädikat ansprechbar ist. Durch die Deklaration `public` ist es demnach von allen weiteren Objekten des **ESP**ro**N**a-Projektes aus aufrufbar. In den Zeilen 2 bis 5 werden die Typen der Eingabeparameter für das Prädikat (erster Parameter von `mode`) und die Anzahl der Lösungen (zweiter Parameter), die durch das Prädikat ermittelt werden, festgelegt.

Der erste Teil des Prädikats `validate_action/3` ist eine atomare Variable: Die zu überprüfende Handlung (`Action`) kann frei oder gebunden sein (spezifiziert durch `?var`). Dies wird durch das Prefix `?` vor der Typdeklaration festgelegt. Bis auf den Zustand (`+list(state)`) sind auch die restlichen Variablen frei oder gebunden; der Zustand allerdings muss gebunden sein (Prefix `+`). Es wird erwartet, dass dieser mit einem konkreten Wert vorbelegt an das Prädikat übermittelt wird. Dessen Ergebnistyp `zero_or_more` sagt aus, dass es keine, eine oder aber auch mehrere Lösungen geben kann. Dies entspricht genau dem bereits besprochenen agilen Ansatz. Es ist demnach möglich, dass ein Prozess im aktuellen Zustand noch nicht gestartet werden kann (keine Lösung), dass es genau eine Möglichkeit (eine Lösung) oder aber mehrere Varianten gibt, um ihn zu starten (mehrere Lösungen). Dies wird durch verschiedene Kombinationen von Instanzen, Agenten, Daten und Werkzeugen erreicht, die bei der Auswertung der Constraints ermittelt werden.

Nach Aufruf des Prädikates `validate_action/3` wird in einem ersten Schritt in Zeile 8 und 9 überprüft, ob die an die Variable `Action` gebundene Handlung auch Teil der definierten Prozesshandlungen ist. Somit wird sichergestellt, dass keine Handlungen validiert werden, die gar nicht ausgeführt werden dürfen. Die erlaubten Handlungen eines Prozesses wurden im Prozessconstraint `process_actions` spezifiziert (siehe Listing 4.1 Zeile 16) und fließen in die Validierungen der einzelnen Perspektiven mit ein (Zeilen 10 bis 14). Diejenige Handlung, die momentan validiert wird (angenommen, `start` ist an die Variable `Action` gebunden), wird auch an die einzelnen Perspektiven bzw. deren spezielle Validierung weitergegeben. Die funktionale Perspektive beispielsweise wird in Zeile 10 überprüft. Durch das aufgerufene Prädikat `functional_constraints_conform/3` werden alle funktionalen Constraints validiert, die Auswirkungen auf eine spezielle Handlung (momentan ist `start` gebunden) haben. Des Weiteren werden der gebundene Zustand

(`State`) und die Variable `Instance` übermittelt. Üblicherweise ist letztere Variable ungebunden, das heißt, bei der Validierung der funktionalen Perspektive wird an die Variable ein Wert gebunden, der die eigentliche Instanz (deren ID) genau spezifiziert.

Bei der Validierung der funktionalen Constraints in Zeile 10 wird das Constraint in Zeile 25 aus Listing 4.1 überprüft. Es ermittelt dabei bei übergebenem Zustand `State`, welche Instanz gestartet werden kann, und gibt diese Information an das Prädikat `validate_action/3` zurück. Das Prädikat `functional_constraints_conform/3` ist das einzige, das bei der Instanz eine freie oder gebundene Variable ermöglicht. Alle anderen Validierungsprädikate der restlichen Perspektiven (Zeilen 11 bis 14) erwarten die Variable `Instance` als gebunden, also mit einem konkretem Wert vorbelegt. Der Grund dafür liegt darin, dass Instanzen bzw. speziell deren Generierung allein der funktionalen Perspektive zugeordnet werden sollen.

Warum bekommen die restlichen Perspektiven eine Referenz auf die Instanzvariable als dritten Parameter? Man möchte beispielsweise ein organisatorisches Constraint schreiben, das folgenden Sachverhalt wiedergibt: „Jede erste und letzte Instanz eines Prozesses muss vom Chef der Abteilung ausgeführt werden. Alle dazwischenliegenden Instanzen können von den Angestellten der Abteilung ausgeführt werden.“ Um zu ermitteln, von welcher Person eine bestimmte Instanz gestartet werden kann, ist es aus Sicht der organisatorischen Perspektive wichtig zu wissen, welche Instanz gerade überprüft wird, um die entsprechende Person an die Variable `Agent` zu binden. Durch Aufruf des Prädikates `validate_action/3` werden also bei übermitteltem Zustand die Instanzen, Agenten, Daten und Werkzeuge an die freien Variablen gebunden, die zum einen valide sind und zum anderen zum Ausführen des Prozesses benötigt werden. Sie entsprechen gültigen Werten, mit denen der Prozess gestartet werden kann.

```
1 ?- process_planning::initial_state(set_up_surgery_plan, InitialState),
2    'set_up_surgery_plan#pid_0'(_)::validate_action(Action, InitialState,
3                                                    Instance-Agents-Data-Tools).
4
5 Action = start,
6 Instance = 1,
7 Agents = ['org#Jack'],
8 Data = [],
9 Tools = ['HIS'] ;
10
11 Action = start,
12 Instance = 1,
13 Agents = ['org#John'],
14 Data = [],
15 Tools = ['HIS'] ;
```

Listing 5.2: Aufruf des Prädikates `validate_action/3`

```
1 :- public(perform_action/4).
2 :- mode(perform_action( +var(action), +list(state),
3                         +compound_term(concept_identifiers),
4                         ?list(state)),
5         zero_or_one).
```

Listing 5.3: Signatur des Prädikates `perform_action/4`

Listing 5.2 zeigt einen Beispielaufruf des Prädikats in **ESP**ro**N**a. In der ersten Zeile wird der initiale Zustand (`InitialState`) für das Prozessmodell `set_up_surgery_plan`

generiert, der dann in der zweiten Zeile benutzt wird, um alle Handlungen, die an `Action` gebunden werden können, zu überprüfen. Diese Handlungen wurden im Prozessmodell, genauer gesagt im jeweiligen Prozess, spezifiziert. Während der Evaluierung werden für den Prozess `Anamnese durchführen` (`set_up_surgery_plan#pid_0`) zwei Ergebnisse zurückgeliefert, mit anderen Worten, zwei Möglichkeiten, wie der Prozess ausgeführt werden kann. Im vorliegenden Fall kann er zum einen von `Jack` und zum anderen von `John` gestartet werden. Angenommen, beim Aufruf des Prädikates `validate_action/3` war die Variable `Action` ungebunden: Während der Validierung wird nun versucht, gültige Handlungen zu binden, sodass das Prädikat `validate_action/3` erfolgreich validiert. Im aktuellen Fall kann der Prozess `Anamnese durchführen` lediglich gestartet werden, wobei es hier zwei verschiedene Möglichkeiten gibt (Zeilen 5 bis 9 und Zeilen 11 bis 15). Diese Informationen werden an die Ausführungsumgebung weitergeleitet und ausgewertet. Für den Nutzer (im obigen Fall `Jack` und `John`) wird somit die Schaltfläche `Starte Prozess Anamnese` nutzbar, denn er ist autorisiert, den Prozess mit den benötigten Daten und vorhandenen Werkzeugen zu starten. Durch das Drücken der Schaltfläche wird nachfolgend an **ESP**ro**N**a übermittelt, dass die angemeldete Person (Agent) den Prozess mit entsprechenden Daten und dem entsprechenden Werkzeug (bzw. Applikation) gestartet hat. Gleichzeitig wird in **ESP**ro**N**a das Prädikat `perform_action/4` aufgerufen (siehe Listing 5.3). Das Prädikat bekommt als Input bis auf den letzten Parameter gebundene Variablen übermittelt. Dies erkennt man an der Signatur der `mode`-Direktive (alle Variablen sind mit `+` gekennzeichnet). `perform_action/4` ruft nachfolgend weitere Prädikate auf, die die Berechnung des Folgezustandes durchführen. Dieser berechnete Zustand wird an die freie Variable des siebten Parameters gebunden und repräsentiert den Zustand, in dem sich das ausgeführte Prozessmodell befindet, nachdem der Prozess gestartet wurde. Der ermittelte Zustand wird nachfolgend an die Ausführungsumgebung zurückübermittelt und dort abgespeichert. Bei einer erneuten Anfrage wird er genutzt, um zu validieren, welche Handlungen - basierend auf dem neuen Zustand - nun ausgeführt werden können.

## 5.3 Kommunikation mit dem ProcessNavigator

Nach erfolgreicher Validierung der Perspektiven übermittelt **ESP**ro**N**a an den ProcessNavigator, dass die Voraussetzungen zum Starten des Prozesses erfüllt sind. **ESP**ro**N**a selbst validiert und übermittelt nur, ob der Prozess gestartet werden kann oder nicht. Der Prozessausführende selbst, also die am `HIS` angemeldete Person, entscheidet letztendlich, ob sie den Prozess startet oder nicht. Bei den restlichen Prozessen des Modells wird ähnlich verfahren: **ESP**ro**N**a validiert auch hier für den ProcessNavigator, ob sie gestartet werden können. Allerdings schlägt diese Validierung hier fehl, da der Prozess `P1` in der verhaltensbezogenen Perspektive erwartet, dass der Prozess `P0` gestartet worden ist. Nachdem dies aber noch nicht der Fall ist, sind die Voraussetzungen hierfür nicht erfüllt. Ähnlich ist es bei Prozess `P2`: Auch hier schränkt die verhaltensbezogene Perspektive ein, dass der Prozess `Operationsplan vorbereiten` erst gestartet sein muss, damit die Vorbedingungen erfüllt sind. Insgesamt kann also nur der Prozess `P0` gestartet werden.

Weiterhin sind noch die restlichen Handlungen zu überprüfen. Bei Prozess `P0` wären dies `finish` und `abort`. Diese Handlungen können im aktuellen Zustand noch nicht ausgeführt werden, denn in den Grundeinstellungen des Prozessmodells sind sogenannte

Basisregeln modelliert, die ein Beenden (`finish`) bzw. Abbrechen (`abort`) des Prozesses erst zulassen, nachdem dieser gestartet ist. Diese Regeln werden in den Einstellungen von iPM$^2$ modelliert und beim Speichern des Prozessmodells mit aufgenommen. Insgesamt kann also die Validierung der genannten Handlungen `finish` und `abort` im initialen Zustand nicht erfolgreich sein, da noch keine Prozessinstanzen gestartet sind, die beendet bzw. abgebrochen werden könnten.

Angenommen, der Nutzer hat den Prozess `Anamnese durchführen` gestartet: Diese Information wird seitens der Ausführungsumgebung an **ESP**ro**N**a übermittelt und daraus der neue Zustand des Prozessmodells berechnet. Darin ist gespeichert, dass der Prozess `P0` gestartet wurde und eine Instanz dieses Prozesses gerade läuft. Dieser Zustand wird an die Ausführungsumgebung zurückgemeldet, und sie kann den Zustand erneut benutzen, um bei **ESP**ro**N**a zu erfragen, welche Handlungen jetzt auf den Prozessen ausgeführt werden können. Nachfolgend wird erneut eine Validierung aller Prozesse, basierend auf dem neuen Zustand, durchgeführt. Dabei würde die Validierung für das mögliche Starten des Prozesses `Operationsplan vorbereiten` aus Sicht der verhaltensbezogenen Perspektive erfolgreich sein, da die modellierten Constraints erfüllt sind: Der Prozess `P0` ist jetzt im Zustand „gestartet“ (`P0 be:has_state :started`). Nun ist es auch möglich, die gestartete Instanz des Prozesses zu beenden oder abzubrechen.

## 5.4 Zusammenfassung

In diesem Kapitel wurde gezeigt, wie die Validierung der POPM-Perspektiven durch **ESP**ro**N**a erfolgt. Durch die Überprüfung, welche Prozesshandlungen ausführbar sind, werden die Constraints der einzelnen Perspektiven ausgewertet und die Agenten, Daten und Werkzeuge ermittelt, mit denen der Prozess ausführbar ist. Durch die minimale Definition der Validierung in einem Prädikat kann eine Erweiterung der Sprache um neue Perspektiven sehr einfach erfolgen. Dies wird im zweiten Teil des nächsten Kapitels beschrieben.

# Kapitel 6

# Erweiterbarkeit

Eines der in Kapitel 1 angesprochenen Probleme ist die mangelnde Erweiterbarkeit und die fehlende Anpassbarkeit der bisherigen PMS an die Ansprüche des Prozessmodellierers. Ziel dieses Kapitels ist es zu zeigen, wie die deklarative PMS **ESP**roN**a** erweitert werden kann. Es wird dabei im Detail erklärt, wie die Implementierung neuer Modellierungskonstrukte erfolgen kann, und auch darauf eingegangen, wie die semantische Bedeutung der neuen Symbole in deklarative Constraints übersetzt werden können. Des Weiteren wird gezeigt, wie eine neue POPM-Perspektive in **ESP**roN**a** hinzugefügt werden kann, um die Ausdrucksstärke der Sprache zu erweitern.

## 6.1 Neue Modellierungskonstrukte

Durch die Erweiterung einer Modellierungssprache um individuelle und mächtige Symbole kann die Komplexität eines Prozessmodells verringert werden [29]. Bevor jedoch ein neues Konstrukt verwendet werden kann, muss zuerst eine klare semantische Definition des Modellierungssymbols erstellt werden. Diese semantische Beschreibung wird in iPM$^2$ erstellt, da die Symbole auch dort bei der Modellierung verwendet werden. Das in Kapitel 3.3 verwendete Konzept zur Kategorisierung der Prozessconstraints nach POPM-Perspektiven, Prozesshandlungen, Prozesszuständen und Prozessinstanzen kann auch hier mit eingebracht werden. Bei der Neukonstruktion eines Modellierungskonstruktes muss man sich zunächst die Frage stellen, ob das Symbol eine oder mehrere POPM-Perspektiven betreffen soll. Der in imperativen PMS verwendete durchgezogene Pfeil, der zwei Prozesse miteinander verbindet, fokussiert alleine die verhaltensbezogene Perspektive. Doch können bei der Definition eines neuen Konstrukts auch mehrere Perspektiven von dessen semantischer Bedeutung betroffen sein. Eine weitere Frage ist, ob das neue Modellierungssymbol als prozessverbindendes oder als prozessübergreifendes Symbol kategorisiert werden muss. Ersteres verbindet zwei Prozesse miteinander und wird im nachfolgenden Abschnitt 6.1.1 genau erklärt. Ein Beispiel für ein prozessübergreifendes Symbol wird in Abschnitt 6.1.2 gezeigt. Beide Konzepte können auch miteinander kombiniert werden, wie in diesem Unterkapitel ebenfalls gezeigt wird. Die Frage, ob es neben diesen beiden Konzepten noch weitere grundlegende Arten an Verbindungstypen gibt, muss an dieser Stelle offen gelassen werden. Im Zuge der geleisteten Forschungsarbeiten und der erwünschten Vereinfachung von Prozessmodellen reichten Modellierungssymbole aus den beiden genannten Kategorien aus, um die gewünschten Ziele zu erreichen.

### 6.1.1 Prozessverbindende Symbole

Beschreibt man den durchgezogenen Pfeil in seiner Bedeutung für den Kontrollfluss eines Prozessmodells, so sagt dieser aus, dass der Prozess B, bei dem der Pfeil eingeht, erst nach dem Prozess A, von dem der Pfeil ausgeht, ausgeführt werden kann. Für den Prozess B ergibt sich folglich eine Abhängigkeit von Prozess A.

Da die aktuellen kommerziellen PMS einem imperativen Stil folgen, das heißt, jeder Schritt genau vordefiniert ist, lässt sich für die Bedeutung des durchgezogenen Pfeils eine weitere Frage bezüglich der semantischen Definition ableiten. Bisher wurde er so definiert, dass der Prozess, bei dem der durchgezogene Pfeil eingeht, unmittelbar nach dem Prozess, von dem der Pfeil ausgeht, ausgeführt wird. Doch kann es durchaus sein, dass - zeitlich gesehen - dazwischen auch andere Prozesse ausgeführt werden. Das gewünschte Verhalten muss bei der Definition des Symbols beachtet werden und führt zum Gedanken, ein zweites und modifiziertes Konstrukt zuzulassen. Zuvor soll allerdings der Vollständigkeit halber noch ein weiteres Modellierungskonstrukt, der gestrichelte Pfeil, erwähnt werden. Dieser hat sich erst während der vorliegenden Forschungsarbeiten auf dem Gebiet der deklarativen Prozessmodellierung entwickelt. Seine Bedeutung ist sehr einfach, denn er verbindet lediglich zwei Prozesse miteinander, ohne eine Wirkung auf den Kontrollfluss zu haben. So können die zwei Prozesse, die durch einen gestrichelten Pfeil verbunden sind, in beliebiger Reihenfolge ausgeführt werden. Weiterhin hat man durch dieses Modellierungskonstrukt die Möglichkeit, einen gewissen Standardablauf zu modellieren, ohne jedoch die Flexibilität des Prozessmodells einzuschränken. Abbildung 6.1 gibt einen Überblick über die bisher definierten Modellierungssymbole. Sie finden anschließend im nachfolgenden Prozessmodell in Abbildung 6.2 Verwendung.

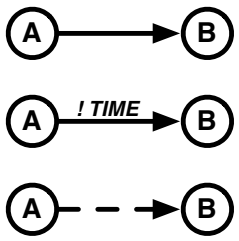



Abbildung 6.1: Übersicht über die bisher definierten Modellierungssymbole

Mit Blick auf das Prozessmodell in Abbildung 6.2 stellt sich die Frage, welcher Kontrollfluss sich aus den Prozessen A bis D ergibt. Aus der Definition des ersten Pfeiles leitet sich ab, dass Prozess B den vorangehenden Prozess A benötigt, also dass dieser abgeschlossen sein muss. Die Definition verbietet allerdings nicht, dass zwischen der Ausführung beider Prozesse noch ein anderer Prozess abgearbeitet werden darf. Genauso können auch vor der Ausführung von A noch andere Prozesse durchgeführt werden. Da Prozess D an den Prozess C durch den gestrichelten Pfeil lose gekoppelt und für diesen Pfeil keine Ablaufreihenfolge definiert ist, kann Prozess D somit noch vor Prozess A ausgeführt werden. Bei der Verbindung zwischen B und C wurde durch den Zusatz `!TIME` spezifiziert, dass Prozess C unmittelbar auf B folgt und dass dieser abgeschlossen sein muss.

Fasst man all diese Informationen zusammen, so ergibt sich für das Modell aus Abbildung 6.2 eine Ablaufsemantik wie in Abbildung 6.3 dargestellt. Man kann - wie bereits


Abbildung 6.2: Verwendung der definierten Modellierungssymbole

erwähnt - nach dem Starten der Ausführung des Prozessmodells auch mit dem Prozess D beginnen. Dies wird durch den gestrichelten Pfeil dargestellt. Oder aber man beginnt mit Prozess A und entscheidet sich danach, ob man dem Fluss des Prozessmodells folgen möchte, um mit Schritt B weiterzumachen (mittlerer Pfad), oder aber man schiebt Prozess D dazwischen, bevor man mit Prozess B weitermacht (linker Pfad). Man erkennt allerdings in der Abbildung 6.3, dass, nachdem Prozess B ausgeführt wurde, Schritt C immer sukzessiv folgt, das heißt, unmittelbar danach ausgeführt wird. Dies verdeutlicht das zusätzliche `!TIME`-Symbol auf dem Pfeil in Abb. 6.2.

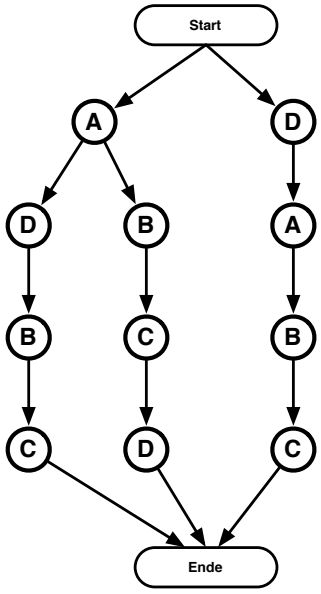



Abbildung 6.3: Mögliche Abläufe des Prozessmodells aus Abbildung 6.2

### 6.1.2 Prozessübergreifende Symbole

Als weiteres Beispiel für die Vereinfachung eines Prozessmodells durch ein neues Modellierungssymbol wird das Prozessmodell aus Abbildung 6.4 herangezogen. Um die Prozesse A, B und C ist ein Rechteck (im Folgenden als Box bezeichnet) modelliert, das eine verhaltensbezogene Aggregation der drei Prozesse ermöglicht. Es modelliert eine Abhängigkeit zwischen allen Prozessen in der Box und dem Prozess D. Dieses Prozessmodell mit dem neuen Box-Konstrukt ist ein Beispiel dafür, dass eine neues Modellierungssymbol nicht auf das Konzept der Pfeile eingeschränkt sein muss.

Im initialen Zustand kann mit Prozess A, B oder C begonnen werden. Alle Prozesse sind über den gestrichelten Pfeil miteinander verbunden. Somit bestehen keinerlei Ab-

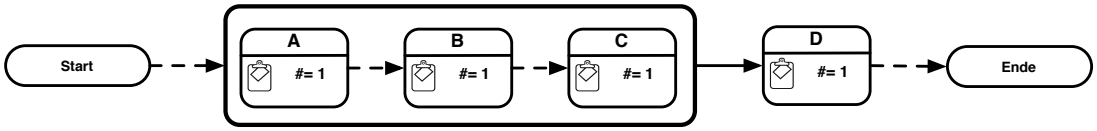



Abbildung 6.4: Prozessmodell mit verhaltensbezogener Aggregation

hängigkeiten zwischen den Prozessen A, B und C. Der Prozess D allerdings benötigt alle drei vorhergehenden Prozesse. Deshalb ist er auch über einen durchgezogenen Pfeil mit der Box verbunden. Benötigt ein Prozess also mehr als einen anderen Prozess, so macht es - wie im obigen Modell gezeigt - Sinn, diese Prozesse in einer Box zu aggregieren und über einen durchgezogenen Pfeil zu verbinden. Man erspart sich hierdurch mehrere Pfeile und macht das Modell übersichtlicher. In Abbildung 6.5 ist erneut die Ablaufsemantik wiedergegeben, wie sie in einer ähnlicher Form in einem imperativen Prozessmodell definiert werden müsste.

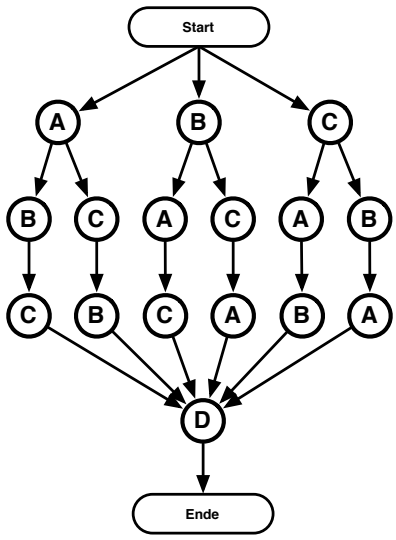



Abbildung 6.5: Ablaufsemantik des Prozessmodells aus Abbildung 6.4

### 6.1.3 Übersetzung nach ESProNa

Die in Abschnitt 6.1.1 und 6.1.2 erwähnten Symbole Pfeil und Box müssen bei der semantischen Modelltransformation in Prozessconstraints übersetzt werden. Dies ist notwendig, da sich **ESP**ro**N**a rein auf Prozessconstraints stützt. Für den in Abbildung 6.1 enthaltenen gestrichelten Pfeil müssen bei der Übersetzung nach **ESP**ro**N**a keine Constraints angegeben werden. Die Flexibilität dieses Pfeils ist im Grundkonzept von **ESP**ro**N**a bereits enthalten. Der durchgezogene Pfeil (ohne das `!TIME`-Symbol) wird auf ein verhaltensbezogenes Constraint im Prozess B abgebildet. Listing 6.1 zeigt den Quellcode der verhaltensbezogenen Perspektive des Prozesses B. In Zeile 3 wird überprüft, ob es im Prozess A eine Instanz gibt, die im aktuellen Zustand auch beendet ist. Im Prozess A, von dem der durchgezogene Pfeil ausgeht, müssen keine Constraints modelliert werden.

```
1 :- object(pid_b(_),
2     extends(process)).
3     ...
4     behavioral_constraint(necessity, [start], State, _,
5     (
6          pid_a(_)::exists_instance(State, finish, _)
7     )).
8     ...
9 :- end_object.
```

Listing 6.1: **ESP**roN**a**-Constraint im Prozess B für den durchgezogenen Pfeil

Der durchgezogenen Pfeil mit dem auf der Verbindung abgebildeten `!TIME`-Symbol verlangt, dass nach der Ausführung von Prozess B unmittelbar der Prozess C ausgeführt wird (siehe Abbildung 6.2). Dies bedeutet, dass zwischen der Ausführung von B und C keine anderen Prozesse ausgeführt werden dürfen. Um dieses Verhalten abbilden zu können, muss in allen anderen Prozessen des Prozessmodells ein Constraint eingefügt werden, dass dies sicherstellt. Angenommen, der Prozess B wurde ausgeführt, so muss die Überprüfung, ob neben Prozess C noch andere Prozesse ausgeführt werden, fehlschlagen. Um dies zu realisieren, wird in den anderen Prozessen ein spezielles Constraint in der verhaltensbezogenen Perspektive eingefügt. Listing 6.2 zeigt dies für den Prozess D, der, wie in Abbildung 6.2 dargestellt, nach Prozess C kommt. Das Constraint wird bei der Überprüfung, ob der Prozess C gestartet werden kann, ausgewertet (der zweite Parameter `start` des Constraints in Zeile 4 spezifiziert dies). Das Prädikat in Zeile 6 überprüft, ob der Prozess B ausgeführt, also beendet (`finish`) wurde. Ist dies der Fall, so muss auch der Prozess C beendet sein (Zeile 7), sonst wird die Auswertung gezielt fehlschlagen (Zeile 9), da ja vorher erst C ausgeführt sein muss.

```
1 :- object(pid_d(_),
2     extends(process)).
3     ...
4     behavioral_constraint(necessity, [start], State, _,
5     (
6          pid_b(_)::exists_instance(State, finish, _) ->
7          pid_c(_)::exists_instance(State, finish, _)
8          ;
9          fail
10    )).
11    ...
12 :- end_object.
```

Listing 6.2: **ESP**roN**a**-Constraint im Prozess D

Die Box drückt eine verhaltensbezogene Abhängigkeit zwischen dem Prozess D und allen in ihr enthaltenen Prozessen aus. Somit müssen in Prozess D drei Prozessconstraints codiert werden, die diese verhaltensbezogene Abhängigkeit ausdrücken. Listing 6.3 zeigt den Quellcode der verhaltensbezogenen Perspektive für den Prozess D aus Prozessmodell 6.4. Von Zeile 3 bis 5 wird bei Auswertung des Constraints überprüft, ob für die drei Prozesse A, B und C beendete Instanzen existieren.

Bei der Definition neuer Modellierungssymboliken in iPM$^2$ ist eine genaue Abbildung zwischen der Semantik des Symbols und den Constraints wichtig. **ESP**roN**a** stellt eine Reihe an vordefinierten Prädikaten zur Verfügung, die zur Definition der Prozessconstraints benutzt werden können. Darüber hinaus können eigene Prädikate hinzugefügt

```
1 behavioral_constraint(necessity, [start], State, _,
2 (
3     pid_a(_)::exists_instance(State, finish, _),
4     pid_b(_)::exists_instance(State, finish, _),
5     pid_c(_)::exists_instance(State, finish, _)
6 )).
```

Listing 6.3: **ESP**roN**a**-Code für Prozess D

werden, um **ESP**roN**a** an die individuellen Bedürfnisse der Modellierungsumgebung anzupassen.

## 6.2 Implementierung neuer Perspektiven

**ESP**roN**a** wurde mit dem Ziel konzipiert, die Sprache für neue Konzepte erweiterbar zu gestalten. Es sei angenommen, dass eine weitere Perspektive implementiert werden soll, die sich mit Quality of Service (QoS) in einem Unternehmen beschäftigt [30]. Durch die Nutzung dieses Konzeptes sollen QoS-Parameter modellierbar sein, sodass zur Laufzeit Aufgaben nur ganz bestimmten Agenten zugewiesen werden, die diesen Anforderungen entsprechen. Es soll an dieser Stelle weniger auf die Details der Perspektive eingegangen, sondern lediglich aufgezeigt werden, wie grundsätzlich neue Perspektiven in **ESP**roN**a** implementiert werden können.

### 6.2.1 Anpassung der Prädikate

Um das genannte QoS-Konzept zu aktivieren, muss das Prädikat `validate_action/3` in Listing 6.4 in der Signatur (Zeile 1) um einen weiteren Parameter (`QoSParameter`) ergänzt werden. In Zeile 9 wird ähnlich zu den bereits existierenden Perspektiven eine Verknüpfung mit dem Prädikat `conform/4` der neu hinzugekommenen Perspektive angelegt. Hierdurch werden bei Evaluierung des Prädikates `validate_action/3` alle Constraints der neu hinzugekommenen Perspektive mit ausgewertet und die darin berechneten `QoSParameter` dem Prädikat zur Verfügung gestellt.

```
1 validate_action(Action, State, Instance-Agents-Data-Tools-QoSParameter) :-
2     ::process_actions(ActionList),
3     list::member(Action, ActionList),
4     ::functional_constraints_conform(     Action, State, Instance),
5     ::behavioral_constraints_conform(     Action, State, Instance),
6     ::organizational_constraints_conform( Action, State, Instance, Agents),
7     ::data_constraints_conform(           Action, State, Instance, Data),
8     ::operational_constraints_conform(    Action, State, Instance, Tools),
9     ::qos_constraints_conform(            Action, State, Instance, QoSParameter).
```

Listing 6.4: Prädikat `validate_action/3`

Neben der Verlinkung der Perspektive mit dem Prädikat `validate_action/3` muss die Perspektive noch in **ESP**roN**a** implementiert werden. Hierzu wird sie, wie alle anderen Perspektiven auch, als sogenannte `category` angelegt (Listing 6.6), die dann nachfolgend in den Prozess importiert wird (Listing 6.5).

Durch dieses Importverfahren wird das Prozessobjekt zum Kompositum aus allen POPM-Perspektiven. Weiterhin muss in der neuen Perspektive bei der Implementierung

```
1 :- object(process,
2     imports(functional_constraint, behavioral_constraint, organizational_constraint,
3             data_constraint, operational_constraint, qos_constraint)).
4     ...
```

Listing 6.5: Import der einzelnen Perspektiven in den Prozess

des Objekts das Prädikat `qos_constraints_conform/4` hinzugefügt werden. Es stellt die Schnittstelle zwischen dem Prozess und der Perspektive dar. Listing 6.6 zeigt die Definition der neu hinzugekommenen QoS-Perspektive. Das Prädikat ist nach dem gleichen Prinzip aufgebaut wie in allen anderen Perspektiven auch. Es sammelt die im Prozessmodell spezifizierten Constraints ein (Zeilen 4 bis 9) und wertet diese aus. Dabei wird bei der Auswertung (Zeile 11) die für die Perspektive charakteristische Variable (hier der `QoSParameter`) an die berechneten Werte gebunden. Durch dieses generische Prinzip ist der Code zur Integration der bestehenden und auch neuen Perspektiven sehr kompakt.

Neben dem `qos_constraints_conform/4`-Prädikat können zusätzlich Hilfsprädikate zum Code in Listing 6.6 hinzugefügt werden. Diese Prädikate stehen dann bei der Spezifikation der Prozessregeln im Prozessmodell zur Verfügung. Sie tragen unter anderem dazu bei, die Constraints prägnanter, übersichtlicher und modularer zu gestalten. Ein Beispiel hierfür ist in Listing 4.1 im Constraint der funktionalen Perspektive zu sehen (Zeile 27). Dort wird das Prädikat `actions_applicable/3` verwendet. Der Code für die Überprüfung, ob die Handlungen im übergebenen Zustand und für die spezifizierte Instanz auch anwendbar sind, wurde in die funktionale Perspektive ausgelagert und wird hier nur referenziert.

```
1  :- category(qos_constraint).
2      ...
3      qos_constraints_conform(Action, ModelState, Instance, QoSParameter) :-
4              bagof(  Constraints,
5                      (   ::qos_constraint(necessity, ActionList, ModelState, Instance,
6                                           QoSParameter, Constraints),
7                          list::member(Action, ActionList)
8                      ),
9                      List
10             ),
11             meta::map({call}, List).
12     ...
13 :- end_category.
```

Listing 6.6: Auszug aus dem Quellcode der neuen QoS-Perspektive

Neben `validate_action/3` muss im Prozessobjekt noch das in Kapitel 5.2 erwähnte Prädikat `perform_action/4` angepasst werden. Dieses Prädikat ist dafür zuständig, dass Informationen über die Ausführung des Prozessschrittes in die Prozesshistorie eingetragen werden. Listing 6.7 zeigt den Quellcode des Prädikates, die geänderten Stellen sind hervorgehoben.

Das Eintragen der Informationen in die Historie des Prozesses wird über das Prädikat `update_model_state/3` realisiert. In Zeile 5 (Listing 6.7) wird der neue QoS-Parameter lediglich an den Term im zweiten Parameter des Prädikats angehängt. Der Code des Prädikats `update_model_state/3` ist so generisch gehalten, dass keine weiteren Änderungen notwendig sind.

```
1 perform_action(Action, ModelState, Instance-Agents-Data-Tools-QoSParameter,
2                NextModelState) :-
3     ...
4     ::id(PID),
5     ModelState::update_model_state(PID, Instance-Action-Agents-Data-Tools-QoSParameter,
6                                    NextModelState).
```

Listing 6.7: `perform_action/4` mit neuer QoS-Perspektive

### 6.2.2 Schritte der Anpassung

Die Erweiterung **ESP**ro**N**a's um eine neue Perspektive gliedert sich in folgende Teilschritte:

- Erweiterung des dritten Parameters des Prädikats `validate_action/3` um den Parameter des neuen Konzepts.
- Implementierung des neuen Konzepts als `category` und Speicherung dieser Datei mit der Endung `.lgt` in einem neuen Ordner `qos_perspective` im übergeordneten Ordern `popm`.
- Import der neu erstellten `category` in das zentrale `process`-Objekt.
- Anpassung des Prädikates `perform_action/4` im `process`-Objekt durch Hinzufügen des neuen Parameters an den vierten Parameter des Prädikates.

Die einzelnen Schritte müssen nicht in genau dieser Reihenfolge ablaufen, sondern können beliebig sein. Wichtig ist allerdings, dass bevor die neue Kategorie in die `loader`-Datei von **ESP**ro**N**a eingebunden wird, alle Arbeiten abgeschlossen sind. Sonst würden beim Starten von **ESP**ro**N**a Fehlermeldungen erscheinen und diesen unterbinden. In Anhang A wird gezeigt, wie **ESP**ro**N**a installiert werden kann und wie es durch Aufruf der `loader`-Datei gestartet werden kann. Listing 6.8 zeigt einen Ausschnitt aus der veränderten `loader`-Datei mit der neu hinzugefügten Perspektive.

```
1  ...
2  logtalk_library_path(op_perspective, popm('operational_perspective/')).
3  logtalk_library_path(qos_perspective, popm('qos_perspective/')).
4  ...
5  write('+++ Loading POPM Operational Perspective definitions...'),
6  logtalk_load(op_perspective(operational_constraint)),
7  ...
8  write('+++ Loading POPM QoS Perspective definitions...'),
9  logtalk_load(qos_perspective(qos_constraint)),
10 ...
```

Listing 6.8: Angepasste `loader`-Datei

Zeile 3 macht den neuen Ordner für **ESP**ro**N**a bekannt, und in Zeile 9 wird die neu erstellte Perspektive geladen. Eine weitere und optionale Aufgabe ist die Ergänzung der `qos_category.lgt`-Datei um individuelle Prädikate, die in den Constraints zu dieser Perspektive verwendet werden können.

## 6.3 Zusammenfassung

In diesem Kapitel wurde gezeigt, wie **ESP**ro**N**a durch die Definition von neuen Modellierungssymbolen erweitert werden kann. Durch dieses Konzept wird dem Modellierer die Möglichkeit gegeben, die PMS den individuellen Gegebenheiten des Unternehmens anzupassen. Durch das Hinzufügen von neuen Modellierungssymbolen mit unterschiedlichen Semantiken kann die Pfadkomplexität eines Prozessmodells deutlich verringert werden. Modellierte Geschäftsprozesse, die dieses Konzept verwenden, haben den Vorteil, dass die erstellten Modelle wesentlich übersichtlicher werden. Durch den modularen und generischen Ansatz ist die Sprache zudem leicht erweiterbar, wie am Beispiel der QoS-Perspektive gezeigt wurde.

# Kapitel 7

# Prozessnavigation

Dieses Kapitel beschreibt die Konzeption der Prozessnavigation, die das in Kapitel 1.4.3 angesprochene Problem der zu vielen Auswahlmöglichkeiten kompensiert. Dem Nutzer wird ein Navigationssystem zur Seite gestellt, das ihm dabei hilft, in verschiedensten Situationen den Überblick zu behalten. Aus technischer Sicht wird in diesem Kapitel der Zustand eines Prozessmodells beleuchtet und erklärt, wie mit Hilfe einer Zustandsübergangsrelation ein Prozessnavigationssystem implementiert werden kann. Außerdem werden verschiedene Suchalgorithmen für den Zustandsraum eines Prozessmodells vorgestellt und deren jeweilige Vor- und Nachteile verglichen. Abschließend wird gezeigt, wie aus dem Navigationssystem Informationen für den Nutzer extrahiert und dargestellt werden können.

## 7.1 Konzeption

Durch das flexible Grundkonzept in **ESP**ro**N**a, bei dem die Ausführung eines Prozesses nur dann blockiert ist, wenn eine Prozessregel sie verbietet, kann es unter Umständen zu einem Sekundärproblem kommen. Bei zu vielen Entscheidungsmöglichkeiten an Prozessschritten, mit denen der Nutzer fortfahren kann, ist es möglich, dass er sich überfordert fühlt. Ein Experte des Prozessmodells wird sicher die Freiheiten zu schätzen wissen, die sich durch das flexible Grundkonzept ergeben, ein Neuling dagegen kann an manchen Stellen überfordert sein und nicht wissen, welcher von den vielen Schritte in der Auswahl der geeignetste ist. Genau zu diesem Zweck wurde die Prozessnavigation entwickelt. Sie kann nicht nur einen Weg vom Start des Prozessmodells hin zum Ende berechnen, sondern auch zwischen beliebigen Zuständen innerhalb eines Prozessmodells navigieren. Dem Nutzer des Systems ist es somit möglich, ausgehend von einer bestimmten Situation, die Prozessnavigation nach einem Weg zu einem anderen Zustand zu fragen. Dabei wird der aktuelle Ausführungsstand des Prozessmodells als Input geliefert sowie die Bedingungen, die im Zielzustand gelten sollen. Diese Bedingungen können ganz vielfältig sein: Man könnte etwa eine Anfrage an die Navigation stellen, ob es einen Weg zu einem bestimmten Zielzustand gibt, bei dem man auf die Verwendung einer Applikation explizit verzichten kann. Aber auch viele andere Möglichkeiten sind denkbar, so zum Beispiel, wenn das Programm gerade nicht verfügbar ist.

Man muss aber nicht immer vom aktuellen Zustand aus einen Weg suchen, sondern kann zwischen beliebigen Zuständen navigieren, um zu prüfen, ob man sich nicht gewisse Möglichkeiten verstellt, wenn man bestimmt Entscheidungen trifft. Um dies zu erklären,

sei angenommen, der Benutzer befindet sich in einem gewissen Zustand des Prozessmodells und steht vor der Entscheidung, einen bestimmten Prozessschritt auszuführen, von dem er vermutet, dass er zu Problemen bei der Ausführung eines anderen Prozessschrittes führen könnte. Der Prozessausführende möchte also an diesem Punkt eine Art Simulation durchführen, noch bevor er die reale Ausführung beginnt.

### 7.1.1 Zustand eines Prozessmodells

Im klinischen Prozessmodell in Abbildung 4.1 sind zwei spezielle Prozesse ausgewiesen: der Start- und der Endprozess, die implizit den Anfangs- und Endzustand eines Prozessmodells markieren. Für jedes geladene Prozessmodell werden - abhängig von der Anzahl der Prozesse - ein initialer und ein finaler Zustand automatisch berechnet. Der Anfangszustand lässt sich leicht beschreiben: Hier wurde noch keiner der Prozesse ausgeführt. Die Prozesshistorien, die jede Handlung im Detail dokumentieren, sind leer. Listing 7.1 gibt eine Anfrage an **ESP**ro**N**a mit dem Ergebnis des initialen Zustands des klinischen Prozessmodells aus Abbildung 4.1 wieder. In Zeile 1 wird die Anfrage an **ESP**ro**N**a gestellt. `process_planning` ist die Bezeichnung des Objekts, in dem das Prädikat `initial_state/2` definiert ist. In Zeile 2 ist das Ergebnis des Prädikats dargestellt: An die Variable `IS` wurde der initiale Zustand des Prozessmodells gebunden, der zum einen eine Liste mit einzelnen Prozesszuständen, zum anderen ein Flag, das das Abarbeiten des Prozessmodells signalisiert, enthält. Es ist im Initialzustand mit dem Wert 0 belegt und trägt im Finalzustand den Wert 1.

```
1 ?- process_planning::initial_state(set_up_surgery_plan, IS).
2 IS = model_state([  process_state(pid_0, []),
3                     process_state(pid_1, []),
4                     process_state(pid_2, [])], 0).
```

Listing 7.1: Definition des Initialzustandes

Ein Prozesszustand aus der oben erwähnten Liste der Prozesszustände besteht aus zwei Komponenten: `Prozess-ID` und `Historie`. Die `Prozess-ID` wird benötigt, um das Objekt eindeutig identifizieren zu können. Die `Historie` - sie ist im initialen Zustand leer - wird während der Ausführung des Prozessmodells mit Einträgen der Form

```
Prozessinstanz-Handlung-Personenliste-Datenliste-Werkzeugliste
```

gefüllt. Diese Einträge spiegeln die auf den Prozess ausgeführten Handlungen wider. Wird zum Beispiel die Instanz `1` des Prozesses `Anamnese` mit der Applikation `software#HIS` und Produktion der Daten `clinic#Patientenakte` durch die Person `John` beendet, so würde folgender Listeneintrag erstellt werden:

```
1-finish-['org#John']-['clinic#Patientenakte']-['software#HIS']].
```

Nach jeder Handlung wird überprüft, ob das Flag des `model_state` angepasst werden kann. Dieses Flag zeigt, ob die Prozesse bereits erfüllt sind oder noch nicht. Eine `0` bedeutet, dass die Prozesse noch nicht abgearbeitet sind, eine `1` markiert sie als erfüllt. Der Folgezustand (`NS`), der sich aus dem Starten des Prozesses `Anamnese` ergibt, ist in Listing 7.2 abgebildet.

```
1 NS = model_state([  process_state(pid_0, [1-start-['org#John']-[]-['software#HIS']]),
2                     process_state(pid_1, []),
3                     process_state(pid_2, [])], 0) .
```

Listing 7.2: Folgezustand nach Starten des Prozesses `Anamnese`

Der Endzustand beschreibt die Situation, in der alle Prozesse abgearbeitet sind. Listing 7.3 zeigt den von **ESP**ro**N**a berechneten Zustand. Die Historienlisten der einzelnen Prozesse sind durch spezielle ungebundene Variable ersetzt und beginnen mit dem Wildcard-Symbol. Bei der Definition des Endzustandes stellt sich natürlich die Frage, wie die Historien dabei aussehen können. Nachdem hier aber sehr viele unterschiedliche Kombinationen bezüglich der Reihenfolge möglich sind, in der die Prozesse ausgeführt werden können, kann man sie nicht alle explizit angeben. Es ist hier aber nicht wichtig, in welcher genauen Abfolge die Prozesse ausgeführt werden, da diese Entscheidung ja dem Prozessausführenden überlassen bleiben soll. Wichtig für den Endzustand ist lediglich, dass alle Prozesse als erfüllt markiert sind, also das Flag des `model_state` auf `1` gesetzt ist.

```
1 ?- process_planning::goal_state(GS).
2 GS = model_state([  process_state(pid_0, _G635),
3                     process_state(pid_1, _G629),
4                     process_state(pid_2, _G623)], 1).
```

Listing 7.3: Definition des Endzustandes

### 7.1.2 Berechnung der Folgezustände

In Kapitel 5.2 wurde der Ablauf der Validierung und der Ausführung eines Prozessschrittes exemplarisch am Prozess `Anamnese durchführen` aufgezeigt. Zuerst erfolgt die Validierung durch das Prädikat `validate_action/3`. Es wird durch den Process-Navigator aufgerufen, um zu überprüfen, welcher Prozess des Modells im aktuellen Zustand ausführbar ist. Dabei erfolgt eine Validierung gegen alle Prozesse. Diese Information wird in der Ausführungsumgebung ausgewertet und dem Nutzer präsentiert. Hat er einen ausführbaren Prozess ausgewählt, wird dieser gestartet. Diese Aktivität wird **ESP**ro**N**a mitgeteilt und der aus der Handlung resultierende Folgezustand durch das Prädikat `perform_action/4` berechnet. In Abbildung 7.1 ist dieser Ablauf nochmals grafisch dargestellt. ❶ markiert den Initialzustand des Prozessmodells. Das Prädikat `validate_action/3` ermittelt zwei Möglichkeiten bzw. Nachfolgezustände: Der linke Zweig zeigt die Ausführung des Prozesses durch den Agenten `John` aus der Chirurgie und endet im Zustand ❷. Die Kanten des Graphen tragen als Bezeichnung die Handlungen mit den genauen Daten, das heißt, die durchgeführte Handlung, die betroffene Prozessinstanz sowie Agent(en) und produzierte Daten und Werkzeug(e), die an der Aktion beteiligt waren.

Der rechte Zweig markiert die Ausführung des Prozesses `Anamnese` durch den Agenten `Jack` aus der Kardiologie. Die Historie im resultierenden Zustand ❸ unterscheidet sich vom linken Zweig, da hier `Jack` und nicht `John` den Prozess ausgeführt hat. Listing 7.4 zeigt nochmals die Möglichkeiten, die durch **ESP**ro**N**a berechnet werden. Die erste Lösung (Zeilen 5 bis 13) entspricht dem linken, die zweite Lösung (Zeilen 15 bis

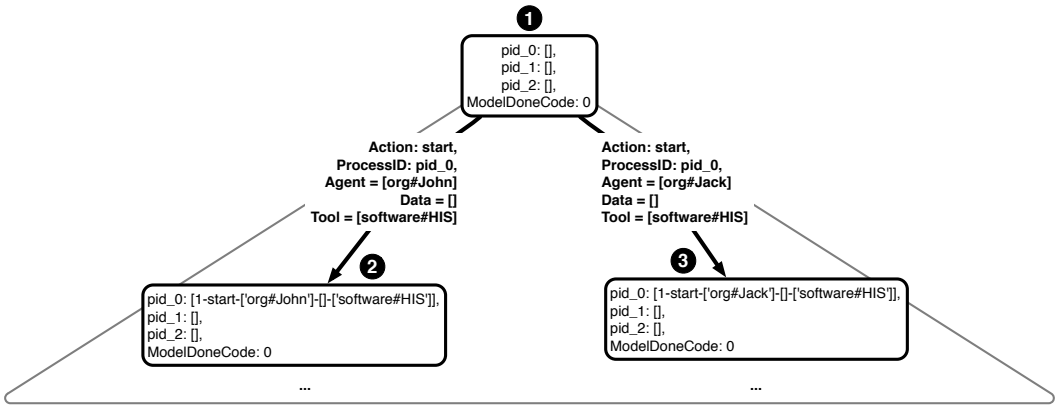



Abbildung 7.1: Initialzustand mit Folgezuständen

20) dem rechten Zweig. Mit diesen berechneten Daten ist es jetzt möglich, das Prädikat `perform_action/4` aufzurufen, um die aus den jeweils ausgeführten Handlungen die resultierenden Zustände zu berechnen. Diese sind in ❷ und ❸ abgebildet. Basierend auf diesen neuen Zuständen kann das Prädikat `validate_action/3` erneut aufgerufen werden, um weitere Ausführungsmöglichkeiten zu ermitteln. Eine Iteration dieses Verfahrens aus der alternierenden Kombination von `validate_action/3` und `perform_action/4` führt zu dem im nächsten Abschnitt angesprochenen Konzept der Zustandsübergangsrelation.

```
1 ?- process_planning::initial_state(set_up_surgery_plan, State),
2    extends_object(ProcessID, process),
3    ProcessID::validate_action(Action, State, Instance, Agents, Data, Tools).
4
5 State = model_state([process_state(pid_0, []),
6                      process_state(pid_1, []),
7                      process_state(pid_2, [])], 0),
8 ProcessID = pid_0,
9 Action = start,
10 Instance = 1,
11 Agents = ['org#John'],
12 Data = [],
13 Tools = ['software#HIS'] ;
14
15 ProcessID = pid_0,
16 Action = start,
17 Instance = 1,
18 Agents = ['org#Jack'],
19 Data = [],
20 Tools = ['software#HIS'] ;
```

Listing 7.4: Validierung aller ausführbaren Prozesse

### 7.1.3 Zustandsübergangsrelationen

Zu dem bereits erwähnten Konzept des Initial- und Endzustandes (`initial_state/2` bzw. `goal_state/2`) kommt noch die Zustandsübergangsrelation `next_state/2` hinzu. Diese drei Prädikate reichen aus, um die Prozessnavigation in **ESP**ro**N**a zu realisieren.

In Listing 7.5 ist das Prädikat `next_state/2` abgebildet: In Zeile 5 des Prädikates werden in einem ersten Schritt alle Prozesse des geladenen Prozessmodells ermittelt und für jeden dieser Prozesse dessen mögliche Handlungen überprüft (Zeilen 6 und 7). Zeile 8 validiert, ob jede dieser Handlungen im aktuellen Zustand (gebunden an die `State`-Variable) durchgeführt werden kann. Bei dieser Validierung werden Prozessinstanzen, Agenten und Werkzeuge zur Ausführung des Prozesses ermittelt. Die gewonnenen Informationen werden nachfolgend an das Prädikat `perform_action/4` weitergegeben, um den neuen Zustand zu berechnen und an die Variable `NextState` zu binden.

Das Prädikat `next_state/2` kann bei gebundenem Zustand mehrere Lösungen für diese Variable ermitteln. Es handelt sich hierbei um ein nichtdeterministisches Prädikat. Die datenbezogene Perspektive ist bei der Validierung (Zeile 8) und Durchführung der Handlung (Zeile 9) durch das Wildcard-Symbol (_) ersetzt. Das Prädikat `validate_action/3` ermittelt unter anderem die benötigten Daten zur Ausführung des Prozesses und bindet diese an den fünften Parameter. Diese Daten werden aber für das Prädikat `perform_action/4` nicht benötigt, sondern hier werden beim Aufruf die produzierten Daten an die Variable im fünften Parameter gebunden. Es erfolgt somit bei der datenbezogenen Perspektive keine Übergabe der Information zwischen den Prädikaten `validate_action/3` und `perform_action/4`. Die erwähnten Suchalgorithmen rufen, ausgehend vom vorher berechneten Zustand, das Prädikat `next_state/2` so oft auf, bis einer der neu berechneten Zustände `NextState` das gleiche Muster aufweist wie der kalkulierte Endzustand des Prozessmodells. In diesem finalen Zustand ist der `ModelDoneCode` auf `1` gesetzt. Mit der Berechnung des Folgezustands durch das Prädikat `next_state/2` wird also immer auch der `ModelDoneCode` überprüft und angepasst, damit die Suche erkennt, wann der Endzustand erreicht ist, und somit erfolgreich terminieren kann.

```
1 :- public(next_state/2).
2 :- mode(next_state(+list(state), ?list(state)), zero_or_more).
3
4 next_state(State, NextState) :-
5     extends_object(Process, process),
6     Process::process_actions(ActionList),
7     list::member(Action, ActionList),
8     Process::validate_action(Action, State, Instance-Agents-_-Tools),
9     Process::perform_action( Action, State, Instance-Agents-_-Tools, NextState).
```

Listing 7.5: Zustandsübergangsrelation `next_state/2`

Abbildung 7.2 zeigt eine mögliche Lösung zum Durchlauf des Prozessmodells vom Initialzustand (oben) bis hin zum Endzustand (doppelt umrandet). An den Kanten sind die detaillierten Informationen der Zustandsübergangsrelationen abgebildet. Sie zeigen nochmals auf, welche Handlungen auf den Prozessen bzw. Instanzen ausgeführt wurden.

### 7.1.4 Suchalgorithmen

Breitensuche und Tiefensuche sind für den Informatiker bekannte Vertreter von Suchalgorithmen in Zustandsräumen. Hinzu kommen – wie bereits angesprochen - die heuristischen Verfahren, um mittels eines Kostenmodells die Laufzeit der Algorithmen bei sehr großen Zustandsräumen zu beschleunigen. **ESP**ro**N**a wurde in der logisch- und objektorientierten Sprache Logtalk [13] implementiert. Die Programmiersprache stellt den Code für Breitensuche, Tiefensuche und heuristische Verfahren [31][32] zur Verfügung. Durch

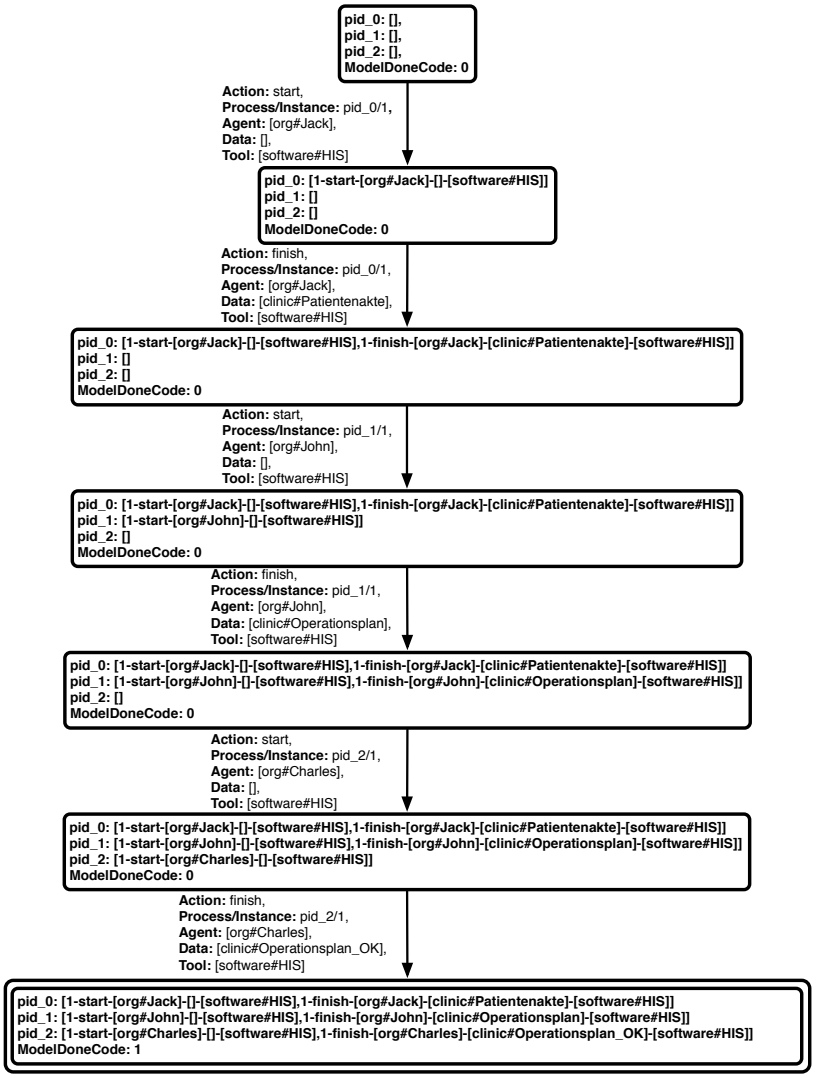



Abbildung 7.2: Lösungsweg für das klinische Prozessmodell vom Initial- zum Endzustand

spezifische Änderungen am Quellcode konnten die Suchalgorithmen für die Implementierung eines Prozessnavigationssystems verwendet werden. **ESP**ro**N**a kommuniziert mit den Suchalgorithmen über eine fest definierte Schnittstelle, in der festgelegt wird, welche Prädikate implementiert werden müssen.

Wie schon der Name „Breitensuche“ sagt, wird bei diesem Verfahren zuerst in der Breite gesucht; dabei werden primär alle möglichen Verzweigungen weiterverfolgt. Für jede dieser Verzweigungen wird wieder parallel weitergesucht, bis eine Lösung gefunden wird. Abbildung 7.1 zeigt den Grundgedanken der parallelen Verarbeitung im Ansatz. Wendet man diese Suche auf das klinische Prozessmodell an, so müssen sehr viele Zustände durchlaufen werden, um einen Lösungsweg zum Ziel zu finden. Listing 7.6 zeigt einen Aufruf der Breitensuche (Zeile 4) mit generiertem Initial- (Zeile 2) und Finalzustand (Zeile 3). Durch das Einbinden einer Performanzanalyse (Zeilen 1 und 5) ist es möglich, detaillierte Informationen über den Verlauf der Suche zu erhalten. Die Lösung ergibt

sieben Prozessschritte (Zeile 7), allerdings müssen dazu beim Suchverfahren Breitensuche sehr viele Zustände bzw. Zustandsübergänge generiert und untersucht werden (Zeile 8). Das bedeutet aber für den Nutzer eine relativ lange Wartezeit, bis er eine Antwort auf seine Navigationsanfrage bekommt. Die lange Laufzeit von 3,7 Sekunden (Zeile 10), die für das relativ kleine Prozessmodell mit drei Schritten durch den Report gemessen wird, wird von dem Verhältnis zwischen der Anzahl an untersuchten Zuständen und der Lösungslänge in Zeile 9 untermauert. Dieser Quotient ist mit 0,0025 sehr niedrig und deutet darauf hin, dass bei der Suche zum Ziel oft in Pfade verzweigt wird, die nicht zur Lösung beitragen.

```
1 ?- performance::init,
2    process_planning::initial_state(set_up_surgery_plan, IS),
3    process_planning::goal_state(set_up_surgery_plan, GS),
4    esprona_breadth_first(6)::solve(process_planning, IS, GS, Path),
5    performance::report.
6 ...
7 solution length: 7
8 state transitions (including previous solutions): 2800
9 ratio solution length / state transitions: 0.0025
10 time: 3.7 sec
```

Listing 7.6: Aufruf und Performanzreport der Breitensuche

Die Charakteristik der abgebildeten Tiefensuche aus Listing 7.7 ist konträr zur Breitensuche. Bei diesem Suchalgorithmus wird ein bestimmter Weg immer weiter in die Tiefe verfolgt, bis eine bestimmte Grenze oder eine Lösung gefunden wurde. Im Performanzreport in Listing 7.7 ist zu erkennen, dass das Verhältnis zwischen der Lösung und den untersuchten Zuständen (Zeile 9) wesentlich besser ist als bei der Breitensuche. Auch ist die Suchzeit mit 0,03 Sekunden um ca. den Faktor 123 kürzer als die bei der Breitensuche. Der Vorteil hieraus ergibt sich für den Nutzer des ProcessNavigators. Deutlich verkürzte Wartezeiten bei Navigationsanfragen machen die Nutzung dieses Features wesentlich angenehmer.

```
1 ?- performance::init,
2    process_planning::initial_state(set_up_surgery_plan, IS),
3    process_planning::goal_state(set_up_surgery_plan, GS),
4    esprona_depth_first(6)::solve(process_planning, IS, GS, Path),
5    performance::report.
6 ...
7 solution length: 7
8 state transitions (including previous solutions): 23
9 ratio solution length / state transitions: 0.304
10 time: 0.03
```

Listing 7.7: Aufruf und Performanzreport der Tiefensuche

Wenn man nun allerdings vermutet, dass die Tiefensuche den am besten geeigneten Algorithmus darstellt, so trifft das nicht zu. Die Laufzeit des Algorithmus hängt auch sehr viel von dem im ersten Parameter übermittelten Grenzwert ab, der bei der Suche mit übergeben wird: `esprona_depth_first(6)`. Bei größeren Prozessmodellen muss dieser angepasst werden. Des Weiteren darf man nicht die kombinatorische Komplexität vergessen, die man bereits bei der Breitensuche erkennen konnte: 2800 generierte Zustandsübergänge bei einem Prozessmodell mit lediglich drei Prozessschritten! Aus die-

sem Grund beschäftigt sich das folgende Kapitel 8 mit den heuristischen Verfahren, die auf einer Erweiterung des Prädikates `next_state/2` aufbauen.

## 7.2 Navigation zwischen beliebigen Zuständen

Für die spezielle Art der Navigation, bei der Pfade zwischen beliebigen Zuständen gefunden werden sollen, wird das Prädikat `navigate/4` verwendet. Es ermittelt bei Input eines gegebenen Zustands und Bedingungen für den gesuchten Zielzustand den Pfad dorthin. Dabei muss dieser Zielzustand nicht bekannt sein, sondern er wird automatisch durch das Verfahren mitberechnet. Das Prädikat versucht dabei unter Zuhilfenahme der Suchalgorithmen einen Pfad zum Zielzustand zu ermitteln, in dem die spezifizierten Bedingungen gelten. Diese müssen sich nicht nur alleine auf den Zielzustand beziehen, sondern dürfen auch Einschränkungen auf dem Pfad dorthin modellieren. Ähnlich dem Navigationssystem im Auto, bei dem man zum Beispiel die „schnellste Route“ als Option wählen kann, kann dies auch für die Prozessnavigation interessant sein.

Listing 7.8 zeigt detailliert den Aufbau des Prädikates `navigate/4`. Der erste Parameter `FromState` repräsentiert den Zustand, in dem sich der Prozessausführende gerade befindet. Der Zustand `ToState` (zweiter Parameter) entspricht dem gesuchten Zustand, in dem bestimmte Bedingungen gelten sollen. Diese werden durch die Variable `Conditions` repräsentiert, die im vierten Parameter angegeben wird. Dieser Zustand wird normalerweise als ungebundene Variable an das Prädikat übergeben. Die spezifizierten Bedingungen des vierten Parameters markieren Restriktionen, die im Zustand `ToState` gelten müssen. Diese Einschränkungen müssen angegeben werden und sind in der `mode`-Direktive nochmals codiert: `+term(conditions)`. Der Code des Prädikates ist sehr einfach und besteht lediglich aus zwei Zeilen. Liest man die Zeilen 7 und 8, so darf man diese nicht als aufeinanderfolgenden Ablauf verstehen, in dem zuerst das Problem durch den Aufruf des Prädikates `solve/4` gelöst und nachfolgend die Bedingungen aufgerufen werden. Vielmehr wird durch das in Prolog implementierte Konzept des Backtrackings versucht, beide Bedingungen zu erfüllen. Dabei werden alternierend so lange Suche (Zeile 7) und Überprüfung der Bedingungen (Zeile 8) durchgeführt, bis entweder eine Lösung gefunden wurde oder die Suche fehlgeschlagen ist. Die Lösung wird im Erfolgsfall an die Variable `Path` gebunden und dem Nutzer mitgeteilt. Sie entspricht einer Worklist, die abgearbeitet werden muss, um zum gewünschten Zustand zu kommen.

```
1 :- public(navigate/4).
2 :- mode(navigate(+list(state), ?list(state), ?list(path),
3                  +term(conditions)),
4         zero_or_more).
5
6 navigate(FromState, ToState, Path, Conditions) :-
7     esprona_depth_first(6)::solve(process_planning, FromState, ToState, Path),
8     {Conditions}.
```

Listing 7.8: Navigation zwischen bestimmten Prozesszuständen

In Listing 7.9 ist der Code für eine Beispielanfrage einer Suche zwischen zwei bestimmten Zuständen abgebildet. In Zeile 1 ist der Ausgangszustand zu sehen, in dem sich der Prozessausführende befindet: Der Prozess `P0` wurde von `John` gestartet. Ausgehend von diesem Zustand möchte man nun zu einem Zustand `ToState` navigieren, in dem der Prozess `P2` von einem Agenten der chirurgischen Abteilung ausgeführt werden

kann. Dies ist im Listing 7.9 in den Zeilen 8 und 9 als Bedingungen codiert. Insofern das Prädikat erfolgreich terminiert, also ein Pfad gefunden wurde, wird dieser an die Variable `Path` des dritten Parameters gebunden. Durch den ProcessNavigator wird das Ergebnis aufbereitet, das bedeutet, der Pfad in eine ansprechende Worklist umcodiert und dem Nutzer präsentiert. Dieser kann sich dann entscheiden, ob er sie abarbeiten will, um zum gewünschten Zustand `ToState` zu gelangen, oder nicht.

```
1  ?- InState = model_state(
2                      [process_state(pid_0, [1-start-['org#John']-[]-['software#HIS']]),
3                       process_state(pid_1, []),
4                       process_state(pid_2, [])],
5                      0),
6     process_planning::navigate(InState, ToState, Path,
7     (
8          pid_2(_)::validate_action(start, ToState, Instance, [Agent], _, _),
9          'org#SurgeryDepartment'::'org#member'(Agent)
10    )).
11
12 Instance = 1,
13 Agent = 'org#Charles'
14
15 Instance = 1,
16 Agent = 'org#Peter'
```

Listing 7.9: Aufruf der Navigation in **ESP**ro**N**a

Als Ergebnis werden unter anderem die in den Bedingungen ungebundenen Variablen an valide Werte gebunden. Nachdem im Prozessmodell aus Abbildung 4.1 modelliert wurde, dass der Prozess `P2` vom Vorgesetzten derjenigen Person ausgeführt werden muss, die den Prozess `P1` gestartet hat, können nur `Charles` oder `Peter` den Prozess in diesem Zustand (`ToState`) starten.

Die Ausgabe der `Path`-Variablen ist in Abbildung 7.3 grafisch aufbereitet: Der oberste Zustand markiert den Ausgangszustand. Die Zustandsübergänge ❶ bis ❺ markieren die verschiedenen Handlungen, die nach und nach ausgeführt werden müssen, um zum doppelt umrandeten Zielzustand zu kommen. Er entspricht dem an die Variable `ToState` gebundenen Zustand, der im Prädikat `navigate/4` spezifiziert wurde. Dort gelten die aus der im vierten Parameter der Anfrage angegebenen Bedingungen: Der Prozess kann in diesem Zustand von einem Agenten aus der chirurgischen Abteilung gestartet werden. Sieht man sich die einzelnen Abfolgen genauer an, so fällt auf, dass zum Beispiel bei ❸ eine unnötige Handlung ausgeführt wird: Eine zweite Instanz des Prozesses `Operationsplan vorbereiten` wird gestartet. Auch in ❹ wird eine weitere Instanz gestartet. Diese beiden Handlungen sind jedoch nicht nötig, denn eine gestartet Instanz des Prozesses `P1` hätte alleine auch schon genügt, damit der Prozess `P2` gestartet werden kann. Somit sind die Zustandsübergangsrelationen ❸ und ❹ überflüssig.

Es ergibt sich daraus die Frage, inwieweit man den im Prädikat `navigate/4` benutzten Suchalgorithmus so beeinflussen kann, dass dieser erkennt, dass die beiden genannten Handlungen unnötig sind. Die Antwort darauf wird im folgenden Kapitel 8 beschrieben. Dort wird gezeigt, wie man mit Priorisierung und heuristischen Verfahren den Suchalgorithmus so beeinflussen kann, dass unnötige Ausführungen bei der Navigation nicht mehr berechnet werden. Es sei nur soviel vorweggenommen, dass der dritte Parameter des Prädikates `next_state/3` die Empfehlung einer Ausführungsmöglichkeit widerspiegelt. Bei der heuristischen Suche wird lediglich der nächste Zustand mit der besten Empfehlung

weiterverfolgt, alle weiteren möglichen Zustandsübergänge mit geringeren Gewichtungen aber nicht. Dadurch verläuft die Pfadsuche zum Zielzustand wesentlich schneller ab.

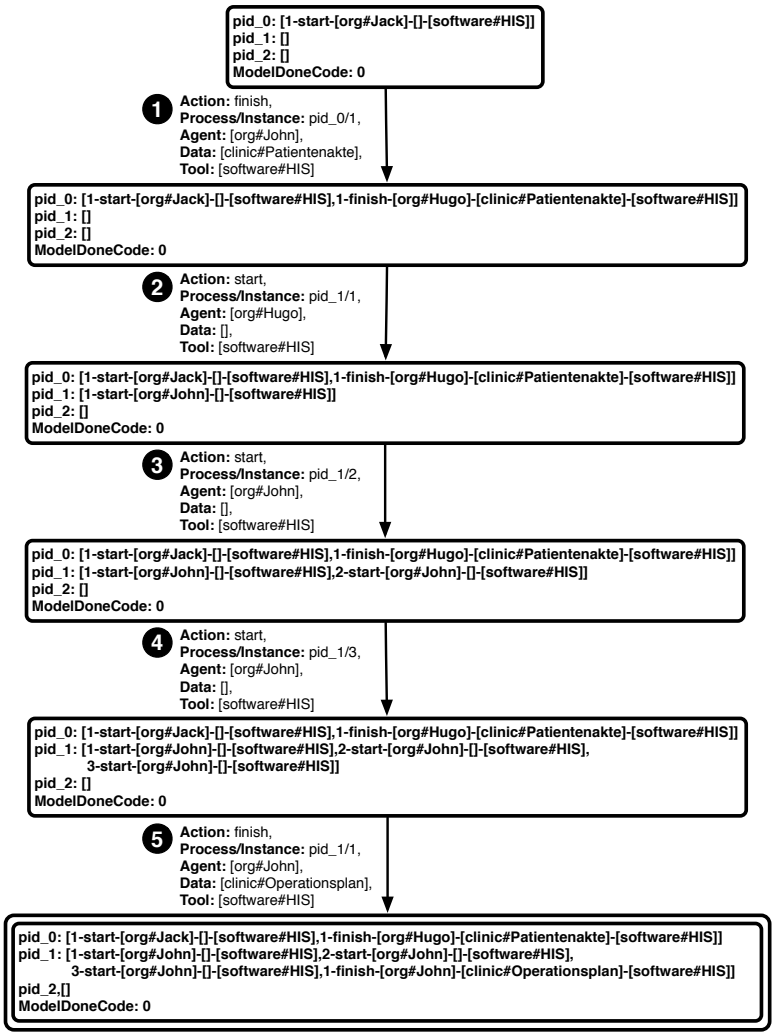



Abbildung 7.3: Zustandsraum der Navigation zwischen bestimmten Prozesszuständen

## 7.3 Interaktion mit dem Nutzer

Die angesprochenen Prädikate und Algorithmen dienen als Grundlage für die visuelle Navigation, die dem Nutzer des ProcessNavigators präsentiert wird. Die in Kapitel 2.3.1 eingeführte Worklist, die dem Nutzer die als nächstes anstehenden Aufgaben und Auswahlmöglichkeiten anzeigt, kann jetzt durch die Navigation erweitert werden. Mögliche Ideen hierzu wurden bereits in einem Paper veröffentlicht [33]. Dort werden der Worklist weitere Spalten hinzugefügt, die eine Art Simulation der zukünftigen Abläufe bei vorausgewähltem Prozessschritt ermöglichen. Konkret bedeutet dies, dass der Nutzer des ProcessNavigators einen der nächstmöglichen Schritte anklickt, ihn aber noch nicht ausführt. Basierend auf dieser Information berechnet **ESP**ro**N**a weitere Daten, die für den

Nutzer auf dem Weg zum Zielzustand interessant sein könnten. Anhand eines einfachen Prozessmodells wird das Konzept der erweiterten Worklist in Abbildung 7.4 verdeutlicht und dargestellt.

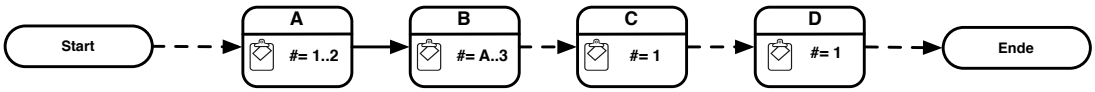



Abbildung 7.4: Prozessmodell zur Verdeutlichung der Navigationsinteraktion

Das Modell besteht aus vier Prozessen (A bis D) wobei hier nur die funktionalen Perspektiven als Constraints modelliert sind. Die verhaltensbezogene Perspektive wird mittels der in Kapitel 6 neu eingeführten Symbolen definiert. Weitere Perspektiven sind nicht modelliert, um das Beispiel zu vereinfachen. Im Initialzustand berechnet **ESP**ro**N**a, dass mit den Prozessen A, C oder D begonnen werden kann. B ist der einzige Prozess, der nicht direkt ausgeführt werden kann, da er den Prozessschritt A benötigt. Sobald der Nutzer einen der Prozessschritte auswählt, berechnet **ESP**ro**N**a die möglichen Konsequenzen. Würde sich der Prozessausführende für den Prozess C entscheiden und diesen auswählen, so würden Informationen wie in Abbildung 7.5 berechnet werden.

| Worklist | Worklist danach | zukünftige Schritte | nicht mehr ausführbar |
|---|---|---|---|
| A<br>C<br>D | A<br>D | A<br>B<br>D | C |

Abbildung 7.5: Informationsausgabe bei Vorauswahl von Prozess C

Das Rechteck um den Prozess C in der Worklist (linke Spalte in Abbildung 7.5) zeigt, dass der Nutzer diesen Prozess vorausgewählt hat. Dabei werden die restlichen Spalten neu berechnet. In der zweiten Spalte wird darüber informiert, welche Prozesse in der nachfolgenden Worklist auftauchen würden, falls Prozess C ausgeführt wird. Prozess B ist hier noch nicht mit aufgelistet, da Prozess A noch nicht ausgeführt wurde. In der Spalte „zukünftige Schritte“ werden alle Prozesse aufgelistet, die nach Ausführung von Prozess C irgendwann später noch möglich sind. Da Prozess C nur einmal ausgeführt werden kann, erscheint er in der vierten Spalte. Dort werden alle Prozesse gelistet, die nach der Ausführung des vorausgewählten Prozesses nicht mehr durchführbar sind. Würde der Nutzer jetzt auf den Prozess B in der Spalte der „zukünftige Schritte“ klicken, so öffnet sich ein kleines Informationsfenster, das anzeigt, dass vor Prozess B erst Schritt A ausgeführt werden muss; Abbildung 7.6 zeigt das.

Insgesamt kann also die Simulation von Ausführungen dazu benutzt werden, um dem Prozessausführenden Informationen zu geben, die eventuell seine Entscheidungen im aktuellen Zustand beeinflussen. So könnte die Information, dass der Prozess C nach einer einmaligen Ausführung nicht mehr erneut ausgeführt werden kann, den Nutzer eventuell davon abhalten, ihn zuerst durchzuführen.

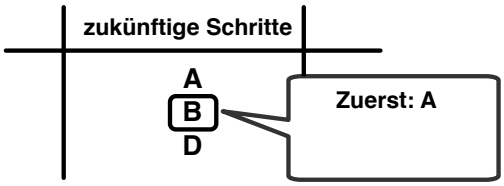



Abbildung 7.6: Informationsausgabe beim Anklicken von Prozess B

## 7.4 Verzögerte Validierung

Als nächstes soll an dieser Stelle die verzögerte Validierung eingeführt werden. Sie ergänzt die Validierung aus Kapitel 5.2 um das Konzept der verzögerten Auswertung und wurde bereits in einem Paper [34] auf der Konferenz CollaborateCom [35] veröffentlicht. Die Idee hinter diesem Verfahren lässt sich am besten an einem Beispiel erklären: Abbildung 7.7 zeigt dazu ein minimales Prozessmodell. Es modelliert lediglich die funktionale, verhaltensbezogene und organisatorische Perspektive, was für die Erklärung des Prinzips aber völlig ausreichend ist. Der durchgezogene Pfeil zwischen den beiden Prozessen `Operationsplan vorbereiten` und `Operationsplan genehmigen` modelliert eine verhaltensbezogene Abhängigkeit zwischen den Prozessen, die dazu führt, dass der Prozess `P2` erst nach dem Prozess `P1` ausgeführt werden kann. Des Weiteren bezieht sich das im Prozess `P2` spezifizierte Constraint der organisatorischen Perspektive auf die ausführende Person des Prozesses `P1`. Es modelliert nämlich, dass der Agent in `P2` ein Vorgesetzter des Agenten aus `P1` sein muss. Nach der Ausführung des Prozesses `Operationsplan vorbereiten` wird anhand der Historie und Ontologie diejenige Person ermittelt, die Vorgesetzter des Agenten aus `P1` ist. Es sei angenommen, das Prozessmodell verwende die in Kapitel 4 aufgezeigte klinische Ontologie aus Abbildung 4.2, und `John` ist der Agent, der den Prozess `Operationsplan vorbereiten` ausgeführt hat. Als Agenten für den Prozess `Operationsplan genehmigen` kommen `Peter` und `Charles` in Frage, da sie die Vorgesetztenrolle erfüllen.

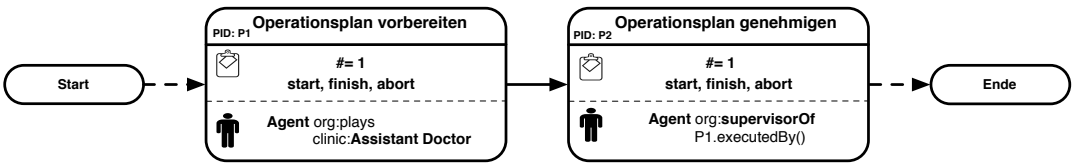



Abbildung 7.7: Klinisches Minimalbeispiel mit durchgezogenem Pfeil

Doch was würde passieren, wenn der durchgezogenen Pfeil in Abbildung 7.7 durch einen gestrichelten Pfeil (Abbildung 7.8) ersetzt wird? Er modelliert jetzt keine Abhängigkeit mehr, sodass aus verhaltensbezogener Sicht mit beiden Prozessen begonnen werden kann. Was bedeutet dies aber für die organisatorische Perspektive? **ESP**ro**N**a verfährt nun so, dass die Agenten, die für den Prozess `P2` in Frage kommen, Vorgesetzte sein müssen. Diejenige Person, die den Prozessschritt `Operationsplan genehmigen` durchführen darf, muss ein Vorgesetzter einer anderen Person sein. An dieser Stelle soll

jetzt nicht der Prozess `P1` als erstes ausgeführt werden, sondern es wird angenommen, dass `Hugo` den Prozess `P2` als erstes abgearbeitet hat und `P1` noch nicht ausgeführt wurde. Welche Personen für den Prozessschritt `Operationsplan vorbereiten` nachfolgend in Frage kommt, muss zuerst anhand der Ontologie ermittelt werden. Das Ergebnis der Anfrage spezifiziert, dass `Hugo` Vorgesetzter von `Jack` und `Kate` ist. Somit können nur noch diese beiden Agenten den Prozess `P1` ausführen, damit das gesamte Szenario noch valide bleibt. Da es zwischen den Abteilungen keine Vorgesetztenbeziehungen gibt, können Agenten aus der chirurgischen Abteilung nicht mehr in Frage kommen.

An diesem Beispiel wird das Prinzip der „Verzögerung“ deutlich. Die Auswertung der für den Prozess `P1` (Abbildung 7.8) in Frage kommenden Agenten kann auch noch nach Ausführung des Prozesses `P2` erfolgen, allerdings mit gewissen Einschränkungen. Deshalb wird die Validierung als „verzögert“ bezeichnet.

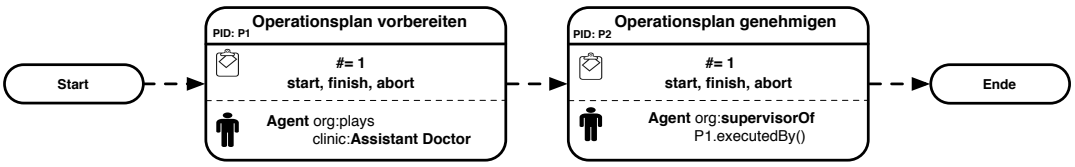



Abbildung 7.8: Klinisches Minimalbeispiel (vgl. 7.7) mit gestricheltem Pfeil

## 7.5 Zusammenfassung

In diesem Kapitel wurde gezeigt, wie sich das in Kapitel 1 erwähnte Konzept der Prozessnavigation in **ESP**ro**N**a integriert und wie mit Hilfe des expliziten Zustands eines Prozessmodells sowie mit verschiedenen Suchalgorithmen die Navigation umgesetzt und genutzt werden kann. Durch die Simulation von Prozessschritten wurde deutlich, wie die Information dem Nutzer des ProcessNavigators helfen kann, seine Entscheidungen zu optimieren. Der letzte Abschnitt hat gezeigt, wie flexibel und vielfältig **ESP**ro**N**a bei der Validierung der Prozessconstraints eingesetzt werden kann. Kleine Änderungen an der Semantik eines Prozessmodells können große Wirkung auf die Ausführungsmöglichkeiten haben.

# Kapitel 8

# Modellierung von Empfehlungen

Ziel dieses Kapitels ist es, zu klären, wie in **ESP**ro**N**a Empfehlungen durch den Modellierer ins Prozessmodell mit integriert, wie durch multimodale Logik eine Kategorisierung und Gewichtung der Prozessconstraints erzielt werden kann und welche Auswirkungen dieses Konzept auf die Prozessnavigation hat. Basierend auf dem Modell der Gewichtungen wird ein Kostenmodell für heuristische Suchverfahren abgeleitet und gezeigt, wie die Antwortzeiten der Suchalgorithmen beschleunigt werden können.

## 8.1 Prozessbezogene Empfehlungen

Prozessbezogene Empfehlungen sollen dem Modellierer die Möglichkeit geben, dem Prozessmodell aus seiner Sicht sinnvolle Vorschläge hinzuzufügen, um den Ausführenden der Prozesse in seinen Entscheidungen zu beeinflussen. Wichtig dabei ist, dass diese sogenannten Soft Constraints die Flexibilität des Nutzers nicht einschränken, sondern dass lediglich eine Gewichtung der möglichen Entscheidungen erfolgt. Möchte der Modellierer also an bestimmten Stellen eines Prozesses eine Empfehlung hinzufügen, so kann er dies in Form eines speziell markierten Constraints tun. Dieses Prozessconstraint wird bei der Validierung, ob ein Prozess ausgeführt werden kann, nicht evaluiert. Es wird erst bei der Präsentation der flexiblen Entscheidungsmöglichkeiten im ProcessNavigator evaluiert, wenn es darum geht, die multiplen Entscheidungsmöglichkeiten zu gewichten. Bevor im Detail erklärt wird, wie die Soft Constraints modelliert werden können, soll ein Exkurs in die modale Logik [36] die Idee und das Konzept der Entwicklung prozessbezogener Empfehlungen zeigen.

### 8.1.1 Multimodale Logik

„Modallogiken sind eine Erweiterung der klassischen Logik um die Konzepte 'Zustände' und 'Zustandsübergänge'. Während Prädikatenlogik davon ausgeht, dass – bezüglich einer gegebenen Interpretation – eine Aussage entweder wahr oder falsch ist, erlaubt Modallogik, dass die Interpretation einer Aussage, also ihr Wahrheitswert, sich ändern kann.“ Weiter heißt es in [37]: „In der einfachsten Modallogik gibt es zusätzlich zu den üblichen logischen Verknüpfungen und Quantoren $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ die beiden einstelligen Operatoren $\Box$ und $\Diamond$. Traditionell wird der $\Box$-Operator als der notwendig-Operator und der $\Diamond$-Operator als der möglich-Operator bezeichnet. Etwas ist notwendigerweise wahr, wenn es in allen vorstellbaren Welten wahr ist.“ Diese Aussage wurde erstmals durch Gottfried Wilhelm Leibniz bereits Anfang des 18. Jahrhunderts formuliert. Basierend

auf dieser Aussage entwickelte Saul Kripke 1959 die modelltheoretische Semantik der Modallogik [38] [39]. „Kripke betrachtete also eine Menge von Welten (Zuständen), deren Elementen jeweils eine klassische Interpretation zugeordnet ist. Zusätzlich nahm er eine zweistellige Relation R zwischen Welten an, mit deren Hilfe der Begriff vorstellbar repräsentiert werden soll. Intuitiv bedeutet R(x,y): Die Welt y ist in der Welt x vorstellbar. Heute spricht man von R als der Erreichbarkeitsrelation oder auch Zugriffsrelation. Jede Welt x entspricht also eine mögliche Interpretation [sic!] (einem möglichen Zustand) und jede andere, von dieser erreichbaren Welt y, ist in x vorstellbar“ [37]. Abbildung 8.1 zeigt die Konzeption nochmals grafisch. Die Welt y ist von x aus erreichbar, und somit vorstellbar.

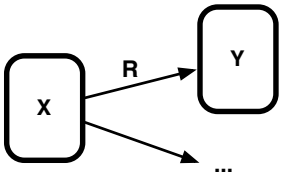



Abbildung 8.1: Relation zwischen den Welten x und y

„Die Semantik der Modaloperatoren ist jetzt mit Hilfe der Erreichbarkeitsrelation definiert: $\Box\Phi$ ist wahr in einem Zustand z, wenn $\Phi$ selbst wahr ist in allen von z aus erreichbaren Zuständen. $\Diamond\Phi$ ist wahr in einem Zustand z, wenn $\Phi$ selbst wahr ist in mindestens einem von z aus erreichbaren Zustand.“ [37]. Das Prinzip ist in Abbildung 8.2 aufgezeigt. In der linken Hälfte ist der Modaloperator $\Box$ mit den von z aus erreichbaren Welten x und y abgebildet. Zur Vereinfachung werden immer nur zwei Welten abgebildet. Sowohl in der Welt x, als auch in der Welt y gilt $\Phi$. Aus diesem Grund wird in z $\Box\Phi$ notiert, da es in allen von sich aus erreichbaren Welten gilt. $\Phi$ selbst kann stellvertretend für verschiedene Aussagen stehen, wie zum Beispiel „Der Prozess wurde vom Chefarzt gestartet“. Dies bedeutet, dass, ausgehend von der Welt z, ein Chefarzt den Prozess startet. Da es zwei Welten ausgehend von z gibt bedeutet dies, dass es zwei verschiedene Chefärzte gibt. Somit unterscheiden sich x und y im Detail der Person, die geforderte Aussage $\Box\Phi$ bleibt aber konstant

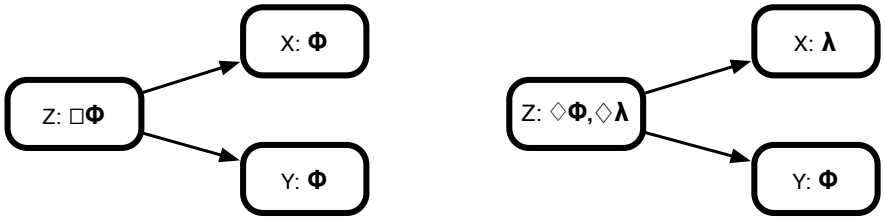



Abbildung 8.2: Semantik der Modaloperatoren $\Box$ (links) und $\Diamond$(rechts)

In der rechten Hälfte der Abbildung 8.2 ist das Prinzip des modalen $\Diamond$-Operators aufgezeigt, der vor die Aussagen $\Phi$ und $\lambda$ positioniert wurde. $\lambda$ gilt nur in der von z aus erreichbaren Welt x, die Aussage $\Phi$ nur in der Welt y. Da die Aussagen $\Phi$ und $\lambda$ nicht in beiden Welten gleichzeitig gelten, kann hier nur der $\Diamond$-Operator verwendet wer-

den. Für die Prozessmodellierung bedeutet dies, dass in Welt x der „Prozess von einem Assistenzarzt ausgeführt wurde“ ($\lambda$), in der Welt y der „Prozess von einem Oberarzt ausgeführt wurde“ ($\Phi$). Anhand dieser beiden unterschiedlichen Rollen lässt sich der modale $\Diamond$-Operator erklären. Es gibt von Welt z aus gesehen zwei Möglichkeiten fortzufahren, die dann aber in unterschiedlichen Zuständen resultieren. Dieses Konzept ist auch in **ESP**ro**N**a implementiert und spiegelt die Flexibilität wider, mit der es möglich ist, aus einem Zustand durch unterschiedliche Entscheidungen in verschiedene andere Zustände zu gelangen. Das Resultat aus den Entscheidung zeigen die Aussagen $\Phi$ bzw. $\lambda$. Das modellierte Constraint im Prozess entspricht der Aussage in Verbindung mit dem Modaloperator, das heißt, bevor die Handlung durch den Prozessausführenden durchgeführt wurde. Die Entscheidung, in welche Welt er als nächstes gelangen möchte, trifft der Nutzer selbst. **ESP**ro**N**a zeigt ihm lediglich die Möglichkeiten auf.

Die angesprochenen Konzepte der Klassifikation in Notwendigkeit und Möglichkeit von Aussagen sollen nachfolgend mit Bezug auf die Modellierung von Empfehlungen in **ESP**ro**N**a integriert werden. Die Idee dabei ist, Prozessconstraints in notwendige und empfohlene Constraints zu klassifizieren. Mit Blick auf das Modell der Welten bedeutet dies, dass verschiedene Empfehlungen in verschiedene Welten übergehen. Weiterhin gibt es eine Ergänzung der modalen Logik, die als multimodale Logik bezeichnet wird [40][41]. Bei diesem Konzept wird eine Gewichtung der ausgehenden Kanten zu den nächsten Zuständen bzw. Welten erreicht, indem die modalen Operatoren indiziert werden. Abbildung 8.3 zeigt ein Beispiel: Die Modaloperatoren wurden vor den Aussagen $\Phi$ und $\lambda$ mit zusätzlichen Indizes versehen, die zu einer Gewichtung der von der Welt z ausgehenden Kanten führt.

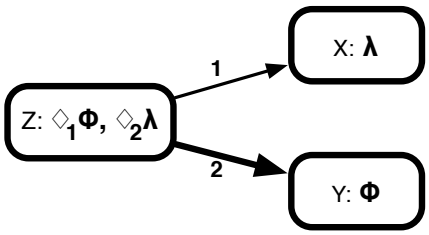



Abbildung 8.3: Semantik der Modaloperationen

Es sei jetzt angenommen, der Prozessausführende ist in der Welt z oder in einem Zustand, der dieser entspricht. Er hat nun zwei Möglichkeiten fortzufahren: Er kann nach x oder y gelangen; diese beiden Alternativen stehen ihm offen. Allerdings wird bei der Liste mit Entscheidungsmöglichkeiten, die im ProcessNavigator angezeigt werden, y an erster Stelle stehen, da es gegenüber x eine höhere Gewichtung hat. Im nächsten Abschnitt wird erklärt, wie die Gewichtung der Prozessconstraints in **ESP**ro**N**a umgesetzt wurde und bei der Modellierung von Prozessconstraints verwendet werden kann.

### 8.1.2 Modale Kategorisierung der Prozessconstraints

Prozessconstraints, die eine Empfehlung ausdrücken, können sowohl im positiven als auch im negativen Sinn spezifiziert werden. Hierdurch kann der Prozessmodellierer ausdrücken, dass aus seiner Sicht ein bestimmter Ablauf sehr oder aber auch gar nicht empfohlen wird. Der maximale Grad, das heißt, wie stark eine Regel positiv oder negativ

gewichtet werden kann, ist in jedem Prozess individuell einstellbar. Abbildung 8.4 gibt einen Überblick, wie die Gewichtungen der einzelnen Regeln verlaufen. Notwendige Prozessregeln werden mit 0 gewichtet, verursachen demnach keine Kosten und entsprechen den Prozessconstraints (Hard Constraints), die bisher verwendet wurden. Empfohlene Prozessregeln (Soft Constraints) werden durch negative Zahlen (-3 ... -1) beschrieben, nicht empfohlene durch positive Zahlen gewichtet (1 ... 3). Der Grund für diese Art der Spezifikation liegt in der eingangs erwähnten Abbildung auf ein Kostenmodell. Durch diese spezielle Art der Gewichtung von Empfehlungen können die Wertebereiche direkt als Input für die Algorithmen der heuristischen Suche verwendet werden.

Gewichtet man beispielsweise eine empfohlene Prozessregel mit -3, so drückt man damit aus, dass etwas sehr stark (+++) empfohlen wird. Ob nun eine Prozessregel notwendig oder empfohlen ist, wird durch ein zusätzliches Präfix im ersten Parameter der Prozessregel angegeben. Listing 8.1 gibt die Signatur der neuen Klassifikation wieder (Zeilen 2 bis 7). Des Weiteren wird ein sogenannter Alias eingeführt, um Prozessregeln, die kein modales Präfix benutzen, als automatisch notwendige Constraints zu klassifizieren (Zeilen 10 bis 12).

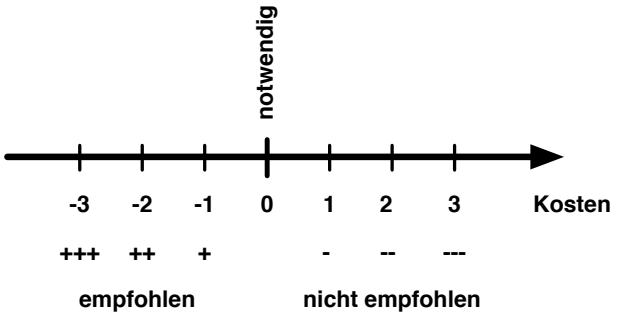



Abbildung 8.4: Kostenabbildung für notwendige und empfohlene Constraints

```
1  :- public(organizational_constraint/6).
2  :- mode(organizational_constraint(  +modal_prefix,
3                                      +list(state),
4                                      +instance_identifier,
5                                      ?list(agents),
6                                      +callable),
7          zero_or_more).
8
9  /* Constraints ohne modales Präfix sind automatisch notwendige Constraints. */
10 organizational_constraint(necessity,
11                           ActionList, State, Instance, AgentList, Constraints) :-
12     ::organizational_constraint(ActionList, State, Instance, AgentList, Constraints).
```

Listing 8.1: Constraints ohne modales Präfix sind automatisch notwendige Constraints

Listing 8.2 zeigt mehrere Beispielregeln, wie organisatorische Constraints im Prozess `Anamnese durchführen` unterschiedlich klassifiziert (Notwenigkeit bzw. Empfehlung) und gewichtet werden können. In den Zeilen 2 bis 6 wird eine notwendige Prozessregel spezifiziert. Hier wird ausgedrückt, dass die Person, die den Prozess ausführen kann, ein Mitarbeiter der kardiologischen Abteilung sein muss. Angewandt auf die Ontologie

aus Abbildung 4.2 bedeutet dies, dass entweder `Kate`, `Jack`, `Hugo` oder `Jacob` für die Ausführung des Prozesses in Frage kommen. Ab der Zeile 9 bis zur Zeile 33 finden sich die Empfehlungen des Prozessmodellierers. In der ersten Empfehlung (Zeilen 9 bis 13) wird sehr stark empfohlen, dass der Prozess von einem Assistenzarzt gestartet wird. Falls der Prozess abgebrochen werden muss, so wird empfohlen, dass dies der Oberarzt ausführt (Regel in Zeilen 16 bis 20). Die Ausführung des Prozesses durch den Chefarzt der Abteilung wird nicht angeraten (Zeilen 23 bis 27). In den Zeilen 30 bis 33 wird schließlich noch sehr empfohlen, dass der Prozess von derjenigen Person beendet wird, die ihn auch gestartet hat.

```
1  /* Notwenige Prozessregel */
2  organizational_constraint(necessity, [start, finish, abort], _, _, [Agent],
3  (
4      instantiates_class(Agent, 'org#Person'),
5      'org#Cardiology'::'org#member'(Agent)
6  )).
7
8  /* Empfohlene Prozessregel: +++ (sehr stark empfohlen) */
9  organizational_constraint(recommendation(-3), [start], _, _, [Agent],
10 (
11     instantiates_class(Agent, 'org#Person'),
12     Agent::'org#plays'('org#AssistantDoctor')
13 )).
14
15 /* Empfohlene Prozessregel: + (empfohlen) */
16 organizational_constraint(recommendation(-1), [abort], _, _, [Agent],
17 (
18     instantiates_class(Agent, 'org#Person'),
19     Agent::'org#plays'('org#AssistantMedicalDirector')
20 )).
21
22 /* Empfohlene Prozessregel: --- (absolut nicht empfohlen) */
23 organizational_constraint(recommendation(3), [start, abort], _, _, [Agent],
24 (
25     instantiates_class(Agent, 'org#Person'),
26     Agent::'org#plays'('org#MedicalSuperintendent')
27 )).
28
29 /* Empfohlene Prozessregel: ++ (sehr empfohlen) */
30 organizational_constraint(recommendation(-2), [finish], State, Instance, AgentList,
31 (
32     pid_0(_)::instance_agent_data_tool(State, start, Instance, AgentList, _, _)
33 )).
```

Listing 8.2: Empfehlungen der organisatorischen Perspektive

## 8.2 Erweiterung der Zustandsübergangsrelation

Um die aus den Empfehlungen berechneten Kosten für die Suchalgorithmen der Prozessnavigation zu aktivieren, muss das Prädikat `next_state/2`, das in Kapitel 7.1.1 besprochen wurde, um den Parameter `Costs` erweitert werden. Dieser Parameter wird als Input für die heuristischen Suchalgorithmen benötigt. Listing 8.3 zeigt den Code für das erweiterte Prädikat. Die Zeilen 4 bis 10 sind bis auf eine Variable `RData`, die bei `next_state/2` nicht benötigt wird, gleich. Diese Variable benötigt man als Input für die Ermittlung der Kosten der datenbezogenen Perspektive. Von Zeile 12 bis 20 erfolgt die Kalkulation der Kosten der jeweiligen Perspektive.

```
1  :- public(next_state/3).
2  :- mode(next_state(+list(state), ?list(state), ?costs), zero_or_more).
3
4  next_state(State, NextState, Costs) :-
5      extends_object(Process, process),
6      Process::process_actions(ActionList),
7      list::member(Action, ActionList),
8
9      PID::validate_action(Action, State, Instance-Agents-RData-Tools),
10     PID::perform_action(Action, State, Instance-Agents-_-Tools, NextState),
11
12     functional_heuristic::costs_do_process(     Process, Action, State, Instance, _,
13                                                 Costs_FU),
14     behavioral_heuristic::costs_do_process(     Process, Action, State, Instance, _,
15                                                 Costs_BE),
16     organizational_heuristic::costs_do_process(Process, Action, State, Instance, Agents,
17                                                 Costs_ORG),
18     data_heuristic::costs_do_process(           Process, Action, State, Instance, RData,
19                                                 Costs_DATA),
20     operational_heuristic::costs_do_process(    Process, Action, State, Instance, Tools,
21                                                 Costs_OP),
22
23     Costs is Costs_FU + Costs_BE + Costs_ORG + Costs_DATA + Costs_OP.
```

Listing 8.3: Zustandsübergangsrelation `next_state/3`

Bei Auswertung der organisatorischen Perspektive beispielsweise (Zeile 16) werden die Empfehlungen, die in Listing 8.2 modelliert wurden, evaluiert und daraus die Kosten berechnet. Folgende Werte ergeben sich bei der Auswertung für die einzelnen Personen der kardiologischen Abteilung beim Starten des Prozesses: `Kate = 0`, `Jack = -3`, `Hugo = 0` und `Jacob = 3`. Für `Kate` und `Hugo` sind keine speziellen Empfehlungen angegeben, und sie fallen somit in die Kategorie der notwendigen Prozessregeln, die mit Kosten 0 bewertet werden. Für `Jack` trifft die erste Empfehlung (Zeilen 9 bis 13) zu und wird mit Kosten -3 bewertet. Die negative Empfehlung für `Jacob` wird aus der dritten Regel abgeleitet (Zeilen 23 bis 27) und erhält die Kosten 3. Das Constraint in Zeilen 16 bis 20 betrifft nur das Abbrechen des Prozesses. Zwar kann der Prozess von einer beliebigen Person der kardiologischen Abteilung abgebrochen werden, allerdings wird hier befürwortet, dass der Oberarzt diese Aufgabe übernimmt. Eine letzte Empfehlung wird in den Zeilen 30 bis 33 gegeben: Es wird hier angeraten, dass diejenige Person, die den Prozess gestartet hat, diesen auch beendet. Ähnlich wird mit den restlichen Perspektiven verfahren. In Zeile 23 werden alle Kosten aufsummiert und stehen als Gesamtkosten zur Verfügung.

Abbildung 8.5 zeigt das Weltenmodell für die Handlung `start` mit den dazugehörigen Aussagen aus multimodallogischer Perspektive. Ist man im Zustand z, so gelangt man durch Starten des Prozesses in die Welten a, b oder c. In allen von z aus erreichbaren Welten gilt $\Phi$ (`org#Cardiology::org#member(Agent)`), da dies als notwendiges Constraint in Listing 8.2 (Zeile 2 bis 6) modelliert wurde. In Welt a gilt lediglich $\Phi$, die Kante dorthin ist mit der Gewichtung 0 gekennzeichnet, in der Welt b gilt zusätzlich zu $\Phi$ noch $\lambda$, das für die Aussage `Agent::org#plays(org#AssistantDoctor)` steht und mit negativen Kosten -3 versehen wurde. Die Welt b entspricht dem Zustand, in dem `Jack` den Prozess gestartet hat. In c gilt neben $\Phi$ noch $\pi$. Das steht für `Agent::org#plays(org#MedicalSuperintendent)` und wird nicht empfohlen. Deshalb wird die Kante zu dieser Welt mit Kosten 3 gewichtet. Dennoch ist es möglich, in

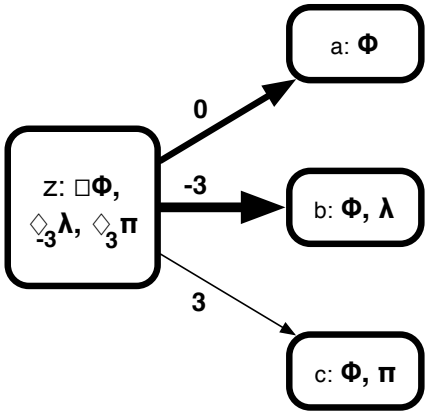



Abbildung 8.5: Visualisierte Welten für die Prozessregeln der Handlung `start`

diese Welt zu gelangen, da es ja nicht direkt verboten ist, den Prozess durch den Chefarzt auszuführen, sondern lediglich nicht empfohlen.

Mit all den Prozessregeln aus Listing 8.2 hat der Prozessmodellierer den organisatorischen Ablauf eines klinischen Prozesses zum einen gemäß den Richtlinien erstellt (notwendige Prozessregeln), aber zum anderen auch noch Empfehlungen in das Modell mit integrieren und hierdurch alltägliche Erfahrung im klinischen Alltag mit in das Prozessmodell aufnehmen können. Ähnlich kann dies auch bei den restlichen POPM-Perspektiven erfolgen.

## 8.3 Prozessübergreifende Empfehlungen

Prozessübergreifende Empfehlungen können als globale Soft Constraints angesehen werden. Sie sind nicht in den Prozessen modelliert, sondern werden in einer separaten Konfiguration zum Prozessmodell abgelegt. Bei der Modellierung der globalen Empfehlungen haben sich die funktionale und die verhaltensbezogene Perspektive als geeignete Kandidaten herausgestellt und werden an dieser Stelle genauer erklärt.

### 8.3.1 Funktionale Perspektive

Die Kalkulation der Kosten für die funktionale Perspektive basiert auf einem speziellen Regelwerk und kann individuell vom Prozessmodellierer angepasst werden. In den Beispielmodellen der aktuellen **ESP**ro**N**a-Version sind folgende Direktiven enthalten, mit dem Ziel, eine Gewichtung der ausführbaren Handlungen aus Sicht der funktionalen Perspektive zu erreichen.

- Ein Abbrechen eines Prozesses verursacht höhere Kosten als die restlichen Handlungen.
- Minimierung der gestarteten Instanzen: starten und danach beenden, anstatt weitere Instanzen zu starten.

Diese Regeln betreffen alle Prozesse eines Modells und sind unabhängig von einem bestimmten Zustand. Die erste Regel hat das Ziel, dass ein Abbrechen eines Prozesses

zwar nicht verhindert werden, aber eben nicht bei der Präsentation im ProcessNavigator an oberster Stelle gleichwertig mit allen anderen Handlungen stehen sollte. Die zweite Regel minimiert die Instanzen eines Prozesses. Wurde in der funktionalen Perspektive eines Prozess ein Wertebereich für die möglichen Ausführungen eines Prozess modelliert, der beispielsweise zwischen zwei und fünf Ausführungen liegen kann, so ist das Ziel, den Prozess nur zweimal auszuführen, also zwei Instanzen zu erzeugen. Diese Direktive bezieht sich wiederum alleine auf die Navigation. Möchte der Nutzer zwei oder mehr Instanzen starten (maximal bis zu fünf), so kann er natürlich auch dies tun.

Der Grund für die Entwicklung dieser Regeln liegt auch darin begründet, die Suchalgorithmen der Prozessnavigation zu beschleunigen, wenn keine Empfehlungen durch den Modellierer angegeben werden. In diesem Fall würden bei größeren Prozessmodellen mit mehr als zehn Prozessschritten die Antwortzeiten bei der Berechnung von Navigationsanfragen mehr als drei Sekunden betragen und zu unangenehmen Verzögerungen führen. Genau wie bei den prozessbezogenen Empfehlungen werden diese Regeln ebenfalls auf ein Kostenmodell abgebildet und zusammen mit den Soft Constraints für die heuristische Suche genutzt.

Listing 8.4 zeigt einen Beispielaufruf des Prädikates `next_state/3` in einem Zustand, in dem der Prozess `Anamnese durchführen` vom Assistenzarzt `Jack` gestartet wurde. In den Zeilen 4 und 5 sind die Kosten für die Handlungen `finish` und `abort` abgebildet. Die Kosten für die funktionale Perspektive sind im Listing fett markiert. Das Abbrechen des Prozesses verursacht höhere Kosten (Kosten = 2) als ein Beenden (Kosten = 1). Da der Prozess `Anamnese` nur einmal ausgeführt werden kann, ist in Listing 8.4 die Handlung `start` nicht aufgeführt. Es sei jedoch angemerkt, dass die Kosten bei einem erneuten Starten eines Prozesses, der bereits eine gestartete Instanz besitzt, 1 sind und unterhalb der Kosten für das Abbrechen liegen.

```
1 ?- process_planning::initial_state(set_up_surgery_plan, IS),
2    process_planning::next_state(IS, NS, Costs).
3  ...
4 finish process pid_0 with costs: 0(FU) + 0(BE) + -3(ORG) + 0(DATA) + 0(OP) = -3
5 abort process pid_0 with costs: 2(FU) + 0(BE) + -3(ORG) + 0(DATA) + 0(OP) = -1
```

Listing 8.4: Aufruf der Zustandsübergangsrelation `next_state/3`

### 8.3.2 Verhaltensbezogene Perspektive

Die Kalkulation der Kosten der verhaltensbezogenen Perspektive beruht auf den Abhängigkeiten zwischen den Prozessen. Oft benötigen Prozesse bestimmte andere Prozesse, speziell deren Zustand, um ausgeführt werden zu können. Die Grundidee ist nun, diese Prozesse, die von vielen anderen Prozessen benötigt werden, schneller abzuarbeiten. Die Abarbeitung eines Prozesses, der in vielen anderen Prozessen referenziert wird, hat geringere Kosten als ein Prozess, der kaum bis gar nicht referenziert wird. Somit entscheidet sich der heuristische Algorithmus bei der Auswahl zwischen mehreren möglichen Prozessen für denjenigen, der am häufigsten von anderen Prozessen benötigt wird. Als Beispiel dient hier das etwas modifizierte Modell aus Abbildung 6.4 (Kapitel 6), das in Abbildung 8.6 dargestellt ist:

Um die Prozesse A, B und C wurde ein Rechteck modelliert, das eine verhaltensbezogene Aggregation der drei Prozesse ermöglicht, es modelliert also eine Abhängigkeit zwischen allen Prozessen im Rechteck und dem Prozess D, der erst ausgeführt werden

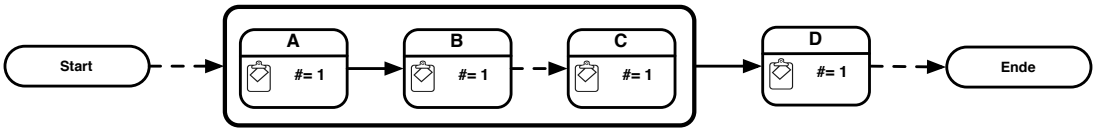



Abbildung 8.6: Modifiziertes Prozessmodell aus Abbildung 6.4

kann, wenn A, B und C abgearbeitet sind. Im initialen Zustand kann mit Prozess A und Prozess C begonnen werden. C ist mit B durch einen gestrichelten Pfeil verbunden, somit bestehen für Prozess C keine verhaltensbezogenen Abhängigkeiten, und er kann im Initialzustand direkt gestartet werden. Prozess B hingegen benötigt Prozess A und kann erst nach dessen Abarbeitung erfolgreich validiert werden. In Abbildung 8.7 ist in der linken Hälfte der Abhängigkeitsgraph der verhaltensbezogenen Perspektive dargestellt, der durch **ESP**ro**N**a berechnet wird. Die Prozesse A und C, die im Initialzustand ausgeführt werden können, sind durch zusätzliche, gestrichelte Kreise markiert. Durch die eingehenden Kanten im Prozess A ist zu erkennen, dass dieser von B und D benötigt wird. Des Weiteren hängt die Ausführung des Prozesses D von B und C ab. Die Kosten der verhaltensbezogenen Perspektive werden nun - basierend auf den eingehenden Kanten - berechnet. Prozess A beispielsweise hat zwei eingehende Kanten und ist derjenige Prozess im Modell, der am meisten referenziert wird. Somit sollte er vor Prozess C abgearbeitet werden, da dieser im Abhängigkeitsgraphen nur eine eingehende Kante aufweist. Listing 8.5 zeigt die ermittelten Kosten für die Ausführung von Prozess A (Zeilen 1, 2) und Prozess C (Zeilen 4, 5). **ESP**ro**N**a empfiehlt aus Sicht der verhaltensbezogenen Perspektive, den Prozess A vor Prozess C auszuführen.

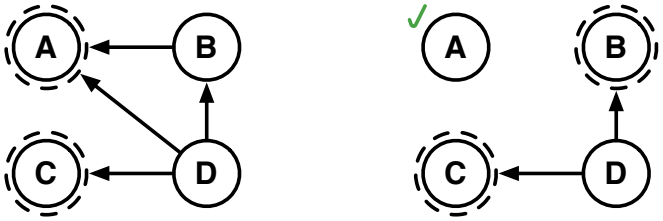



Abbildung 8.7: Abhängigkeitsgraph im Initialzustand (links) und nach erfolgreicher Ausführung von Prozess A (rechts)

```
1 2 ?- behavioral_heuristic::costs_do_process(pid_a(_), start, _, _, _, Costs).
2 Costs = 0.
3
4 3 ?- behavioral_heuristic::costs_do_process(pid_c(_), start, _, _, _, Costs).
5 Costs = 1.
```

Listing 8.5: Ermittlung der Ausführungskosten der Prozesse A und B im Initialzustand

In einem Zustand, in dem der Prozess A bereits ausgeführt wurde, verändern sich die Abhängigkeitsverhältnisse. Im rechten Teil der Abbildung 8.7 ist der neue Abhängigkeitsgraph abgebildet. Prozess A wird nicht mehr für die Auswertung hinzugezogen, da er

bereits als erledigt markiert wurde. B und C sind diejenigen Prozesse, die jetzt ausführbar sind (markiert durch die gestrichelten Kreise). Für beide Prozesse sind die Kosten gleich 0, und somit werden beide Prozesse gleich gewichtet. Mit Blick auf das Prozessmodell in Abbildung 6.4 erkennt man, dass der Prozessmodellierer allerdings den Prozess B vor Prozess C gestellt hat. Somit zeigt sich eine gewisse Präferenz dafür, dass Prozess B vor Prozess C ausgeführt werden sollte. Diese Präferenz wird durch **ESP**ro**N**a erkannt und auch berechnet. Als Konsequenz daraus erhält der Prozess B geringere Kosten als Prozess C.

## 8.4 Korrigiertes Suchverhalten

Am Ende des Kapitels 7 wurde bereits der durch die Tiefensuche berechnete Ablauf erläutert und zum einen das Problem erkannt, dass bestimmte unnötige Prozessschritte aufgeführt wurden (siehe Abbildung 7.3). Die Zustandsübergänge ❸ und ❹ waren dort überflüssig. Zum anderen besteht bei der Verwendung der Tiefen- und Breitensuche das Problem der längeren Wartezeiten bei Navigationsanfragen mit größeren Prozessmodellen. Durch die prozessbezogenen sowie globalen Empfehlungen zusammen mit heuristischen Suchalgorithmen können aber die bestehenden Probleme gelöst werden.

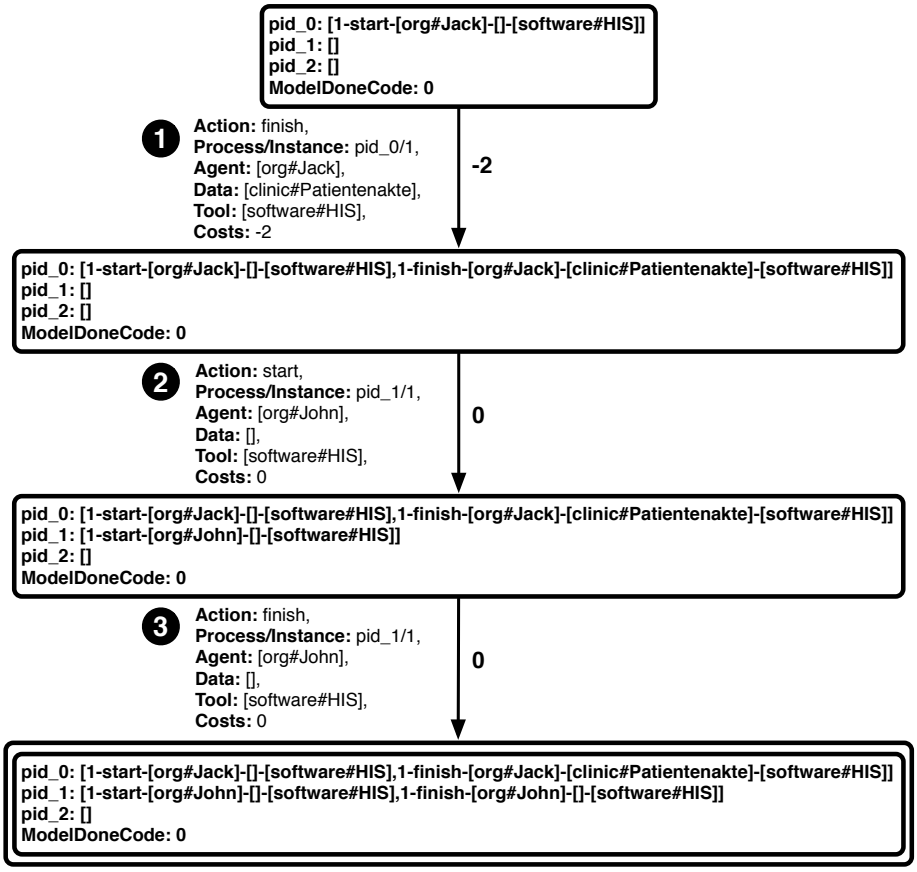



Abbildung 8.8: Heuristische Navigation unter Vermeidung unnötiger Ausführungen

Abbildung 8.8 zeigt den Ablauf, den eine heuristische Suche, basierend auf dem Empfehlungskonzept dieses Kapitels, berechnet. Ausgangspunkt ist - wie auch in Kapitel 7 - der Zustand, in dem der Prozess `Anamnese durchführen` von `Jack` gestartet wurde. Dies entspricht auch der Empfehlung, die der Modellierer gegeben hat. In der durch die heuristische Suche berechneten Zustandsübergangsrelation ❶ wird den Empfehlungen des Prozessmodellierers (Listing 8.2, Zeilen 30 bis 33) ebenfalls Folge geleistet, und der Prozess wird von `Jack` beendet. Als nächstes ❷ wird eine Prozessinstanz `PID 1` durch `John` gestartet und auch wieder durch ihn beendet ❷. Der aus dieser Handlung resultierende Endzustand (doppelt umrandet) ist zugleich auch der Zustand, der das Ziel der Navigation ist: Eine Person der chirurgischen Abteilung, die zugleich die modellierten Prozessregeln erfüllt (Vorgesetzter von `John`), kann jetzt den Prozess ausführen. Die in der Tiefensuche noch unnötigerweise berechneten Zustandsübergänge ❸ und ❹ sind in der heuristischen Suche nicht mehr enthalten.

## 8.5 Zusammenfassung

In diesem Kapitel wurde gezeigt, wie prozessbezogene Empfehlungen durch den Modellierer mit in das Prozessmodell aufgenommen werden können. Zusammen mit den globalen Empfehlungen konnten beide auf ein Kostenmodell abgebildet werden, das als Input für heuristische Suchverfahren verwendet werden kann. Diese Suchalgorithmen beschleunigen Navigationsanfragen bei größeren Prozessmodellen, und es werden keine unnötigen Schritte mehr berechnet.

# Kapitel 9

# Implementierung

**ESP**ro**N**a wurde als Applikation in der logisch-objektorientierten Programmiersprache Logtalk verfasst. Logtalk ist eine Erweiterung der Programmiersprache Prolog und fügt dieser objektorientierte Konzepte hinzu. Logtalk ist mit verschiedenen Prolog-Interpretern kompatibel (siehe [42]) und kann als Applikation in diesen geladen werden. Durch die Unterstützung sowohl eines prototypbasierten als auch eines klassenbasierten Ansatzes deckt Logtalk alle Vererbungshierarchien ab [43]. Um Anfragen im objektorientierten Stil an OWL-Ontologien zu ermöglichen, wurde der Quellcode des Thea2-Projektes [44] in die **ESP**ro**N**a-Bibliothek mit aufgenommen und modifiziert. Somit werden aus den Ontologien Objekte erzeugt, die in Logtalk geladen und zur weiteren Verarbeitung abgefragt werden können. Da Thea2 die `semweb`-Bibliothek aus SWI-Prolog [14] [45] benötigt, kann **ESP**ro**N**a momentan nur in SWI-Prolog ausgeführt werden. YAP-Prolog [46] plant zwar die Portierung der `semweb`-Bibliothek, allerdings war dieses Vorhaben während der Implementierung **ESP**ro**N**as noch nicht abgeschlossen. Somit bleibt SWI-Prolog zum aktuellen Zeitpunkt der einzige Interpreter, in dem **ESP**ro**N**a geladen und ausgeführt werden kann. Zum Zeitpunkt der Veröffentlichung von **ESP**ro**N**a wurde SWI-Prolog in Version 5.11.25 zusammen mit Logtalk in Version 2.43.0 benutzt.

## 9.1 Strukturelle Übersicht

Der Quellcode, der die Implementierung beinhaltet, gliedert sich in die in Abbildung 9.1 gezeigte Baumstruktur. Die blau gekennzeichneten Ordner markieren die Top-level-Ebenen. `contributions` beinhaltet externe Projekte wie etwa das für **ESP**ro**N**a angepasste Thea2 oder den Programmcode zur Messung der Performanz verschiedener Suchalgorithmen. Der Ordner `debug` enthält den Programmcode zur detaillierten Ausgabe der Zustandsübergangsrelationen auf die Konsole. Im Ordner `doc` wird die automatisch generierte Dokumentation abgelegt und ist dort in Form von .html- und .pdf-Dokumenten verfügbar. Der Quellcode des **ESP**ro**N**a-Kerns befindet sich in `library`. Im Ordner `planning` befinden sich die Definitionen für den Prozesszustand, die Schnittstellen zu den Suchalgorithmen (diese befinden sich im Unterordner `solvers`) sowie der Programmcode zur Visualisierung des berechneten Zustandsraumes (Ordner `visualize`). Im Ordner `process` liegt der Programmcode der Prozessobjekte sowie ein Unterorder `popm` mit den Definitionen zu den einzelnen POPM-Perspektiven. Diese werden in das Prozessobjekt importiert. In `state` sind die Objekte des Zustandsraumes gespeichert. `utilities` beinhaltet ein Objekt, das verschiedene Prädikate zur Visualisierung zur Verfügung stellt.

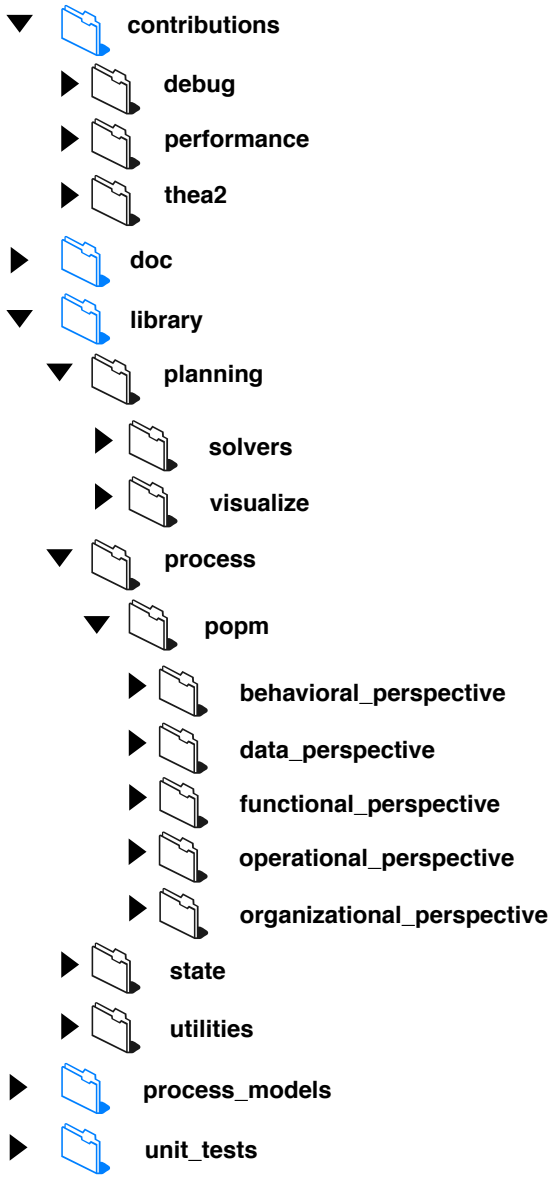



Abbildung 9.1: Ordnerstrukturen in **ESP**ro**N**a

Der Ordner `process_models` enthält weitere Unterordner mit Prozessmodellen. In jedem Ordner befindet sich unter anderem das Prozessmodell mit den Definitionen und Constraints der einzelnen Prozesse. Weiterhin sind hier die Ontologien der organisatorischen, datenbezogenen und operationalen Perspektive abgelegt. In einer weiteren `loader.lgt`-Datei werden diese Ontologien beim Laden eines Prozessmodells übersetzt, sodass sie in den Prozessconstraints benutzt werden können. Im Unterordner der Prozessmodelle ist außerdem das grafische Modell als PDF gespeichert. Im Ordner `unit_tests` befinden sich Testobjekte, mit denen die Konsistenz von **ESP**ro**N**a nach diversen Änderungen am Code überprüft werden kann.

## 9.2 Zusammenhänge der einzelnen Komponenten

Abbildung 9.2 zeigt eine Sichtweise auf das **ESP**ro**N**a-Projekt, in der die Zusammenhänge zwischen den einzelnen Objekten dargestellt sind. Alle POPM-Perspektiven werden, wie bereits in Kapitel 6.2 beschrieben, in das Prozessobjekt importiert, sodass neue Konzeptionen einfach implementiert werden können. Zwischen dem Prozessobjekt in Abbildung 9.2 und dem `process_model`-Objekt, das die Definitionen aller Prozesse des Prozessmodells beinhaltet, besteht eine 1:N-Beziehung. Durch eine prototyp-basierte Instanziierung der Prozessobjekte im Prozessmodell können ohne explizite Instanziierungsanweisungen Prozesse mit entsprechenden Prozessconstraints spezifiziert werden (siehe Listing 4.1). Da ein Prozessmodell in der Regel mehrere Prozessdefinitionen (N) enthält, ist ein prototyp-basierter Ansatz hier von Vorteil. Das Laden der Prozessmodelle (Aufruf siehe Listing 9.1) erfolgt über die `loader`-Datei, die sich in jedem Ordner eines Prozessmodells befindet. Listing 9.2 zeigt den detaillierten Aufbau dieser Datei, die das Laden des klinischen Prozessmodells im Ordner `set_up_surgery_plan` übernimmt. Die Ontologien der einzelnen Perspektiven werden in den Zeilen 1 bis 3 eingelesen. Danach werden sie in Klassen (Zeile 4) und Instanzen (Zeile 5) übersetzt. Vor dem abschließenden Laden des Prozessmodells in Zeile 9 werden noch temporäre Daten zurückgesetzt (Zeile 6).

```
1 ?- logtalk_load(set_up_surgery_plan(loader)).
```

Listing 9.1: Laden eines klinischen Prozessmodells

```
1 owl2_to_logtalk:load_axioms('organization.owl'),
2 owl2_to_logtalk:load_axioms('tools.owl'),
3 owl2_to_logtalk:load_axioms('data.owl'),
4 owl2_to_logtalk:save_axioms_ontology('popm_objects.lgt', owlpl, [no_base(_)]),
5 owl2_to_logtalk:save_axioms_individuals('popm_individuals.lgt', owlpl, [no_base(_)]),
6 owl2_to_logtalk:cleanup,
7 logtalk_load(popm_objects, [unknown(silent)]),
8 logtalk_load(popm_individuals, [unknown(silent)]),
9 process_model_loader::load(process_model).
```

Listing 9.2: Laden des klinischen Prozessmodells im Detail

Nun können Anfragen an das Prozessmodell nach dessen Laden erfolgen. Für den Prozessausführenden ist zum Beispiel interessant, welche Prozesse ausführbar sind (siehe Listing 9.3).

```
1 ?- extends_object(Process, process),
2    process_planning::initial_state(set_up_surgery_plan, IS),
3    Process::validate_action(start, IS, Instance-Agents-Data-Tools).
```

Listing 9.3: Welche Prozesse sind im Initialzustand `IS` ausführbar?

In der unteren Hälfte der Architektur (Abbildung 9.2) sind die Komponenten der Prozessnavigation abgebildet. `process_planning` instanziiert den heuristischen Zustandsraum, der eine Spezialisierung des Zustandsraumes `state_space` darstellt. Im Objekt `process_planning` sind somit neben den Zustandsübergangsrelationen `next_state/2`

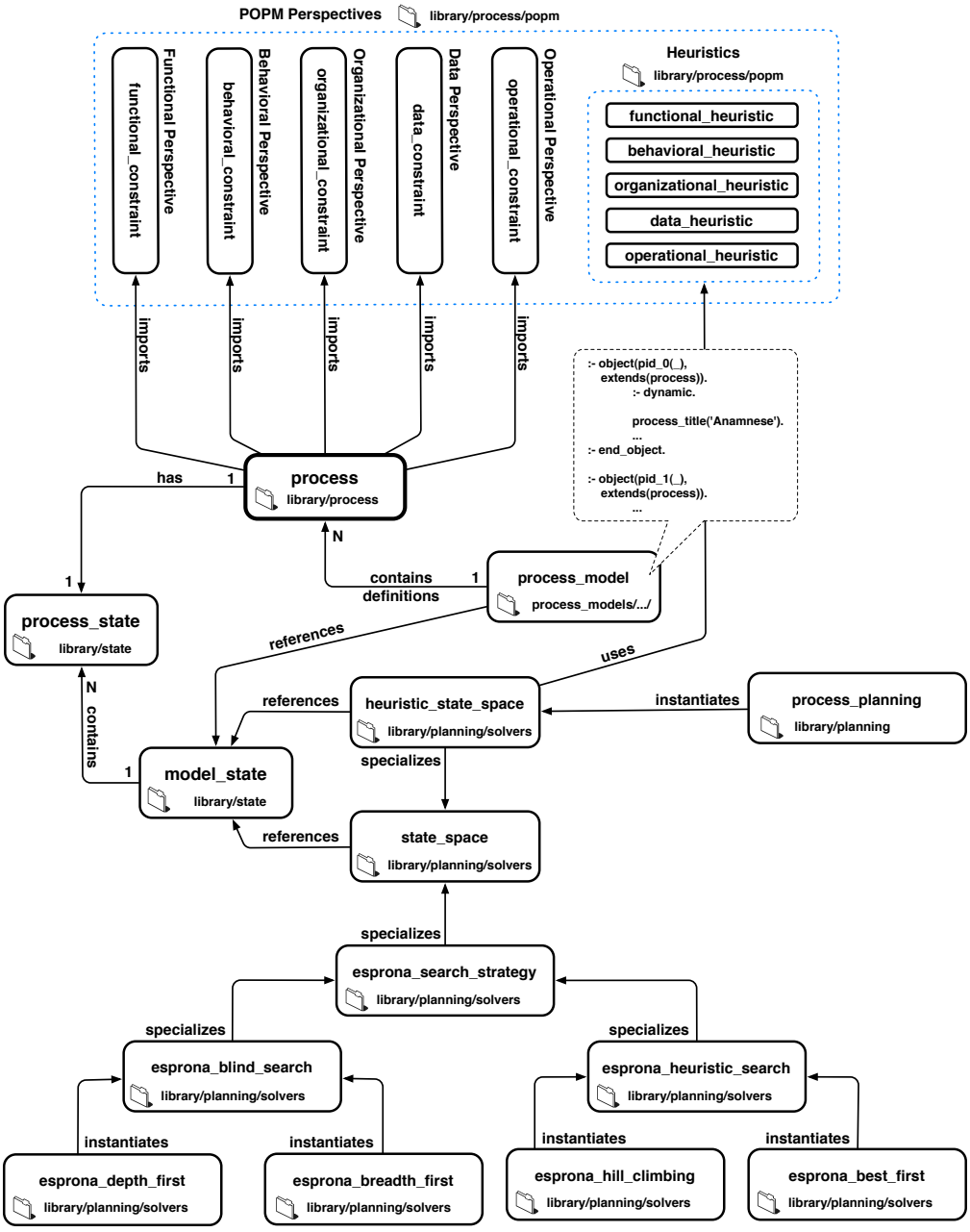



Abbildung 9.2: Architekturübersicht **ESP**ro**N**a

und `next_state/3` noch die Prädikate `navigate/4` bzw. `navigate/5` enthalten sowie `heuristic/2` zur Abschätzung der Kosten vom aktuellen Zustand zum Zielzustand. Da alle notwendigen Constraints immer die Kosten 0 haben, können bei einer Ausführung, die keine Empfehlungen beachtet, die Gesamtkosten mit 0 abgeschätzt werden. Beide Zustandsräume referenzieren das sogenannte `model_state`-Objekt, das den Zustand des geladenen Prozessmodells beschreibt. Es besteht aus einer Liste von einzelnen Prozesszuständen (`process_state`) sowie dem Flag `ModelDoneCode`. Diese Variable kann die Werte 0 (Prozesse noch nicht abgearbeitet) oder 1 (Prozesse sind abgearbeitet) annehmen. Die Objekte der Prozesszustände speichern im Einzelnen für jeden Prozess die Prozess-ID sowie die Ausführungshistorie des Prozesses. Der Zustandsraum `state_space` wird von der `esprona_search_strategy` spezialisiert. Diese Suchstrategie ist wiederum eine Abstrahierung für die sogenannten blinden Suchalgorithmen wie Tiefen- und Breitensuche (`esprona_depth_first`, `esprona_breadth_first`), aber auch für die heuristische Suche (`esprona_hill_climbing`).

## 9.3 Zusammenfassung

In diesem Kapitel wurde der strukturelle Aufbau **ESP**ro**N**as gezeigt. Die Beziehungen zwischen den einzelnen Objekten wurden ebenfalls behandelt. Anhand der Architekturübersicht in Abbildung 9.2 wurde nochmals aufgezeigt, wie die einzelnen POPM-Perspektiven in das Prozessobjekt importiert werden. Der Zustand eines Prozessmodells und seine Bedeutung für die Navigation wurde ebenfalls nochmals grafisch verdeutlicht.

# Kapitel 10

# Related Work

Um einen Überblick über die verschiedenen PMS zu erhalten, gilt es nun, **ESP**ro**N**a anderen Modellierungssprachen und Ausführungsumgebungen gegenüberzustellen. Ausdrucksstärke, Prozessnavigation und die Modellierung von Empfehlungen waren die Kernkonzepte, die in dieser Dissertation angesprochen und entwickelt wurden. Im Folgenden werden diese Kernkonzepte mit anderen Forschungsarbeiten und Industriestandards verglichen. Der erste Teil dieses Kapitels gliedert sich in eine Übersicht an Anforderungen, mit denen die Prozessmodellierungssprachen aus Industrie und Forschung verglichen werden, wobei nochmals die Anforderungen aus Kapitel 1 aufgegriffen werden. Daran anschließend erfolgt der Vergleich der Vertreter der imperativen und deklarativen Prozessmodellierungssprachen, basierend auf diesen Argumenten, mit **ESP**ro**N**a.

## 10.1 Anforderungen

Als Anforderungen für den Vergleich mit anderen Prozessmodellierungssprachen aus Industrie und Forschung dienen Ausdrucksstärke, Erweiterbarkeit, Prozessnavigation und die Modellierbarkeit von Empfehlungen.

### 10.1.1 Ausdrucksstärke

Im Begriff Ausdrucksstärke verbirgt sich, dass alle POPM-Perspektiven in einer Modellierungssprache abgebildet werden können. Hierdurch wird sichergestellt, dass ein Geschäftsprozess in seiner Gesamtheit erfasst und modelliert werden kann. Nur so ist garantiert, dass keine Informationen verloren gehen und die Prozessabläufe eines Unternehmens im Nachhinein korrekt reproduziert werden können.

### 10.1.2 Erweiterbarkeit

Unter der Erweiterbarkeit einer PMS versteht man, wie gut eine Sprache individuell an die Anforderungen des Prozessmodellierers anpassbar ist. Die Frage, die sich hier stellt, ist, inwieweit der Prozessmodellierer die PMS selbstständig den Gegebenheiten des Unternehmens anpassen kann.

In Kapitel 6 wurde gezeigt, wie neue Modellierungskonstrukte die Komplexität eines Prozessmodells vereinfachen können und wie durch die Erweiterung der bekannten Modellierungskonstrukte mit neuen Bausteinen die Pfadkomplexität eines Prozessmodells deutlich verringert werden kann. Schließlich gilt es noch, die Fähigkeit einer Sprache

in Bezug auf die Ausführungsreihenfolge zu untersuchen. Dabei spielt die Komplexität der resultierenden Modelle eine große Rolle.

Weiterhin wurde durch die Möglichkeit des Hinzufügens einer neuen Perspektive gezeigt, wie neue Konzepte in eine Sprache integriert werden können.

### 10.1.3 Prozessnavigation

Unter diesem Begriff versteht man, wie gut der Prozessausführende zu einem bestimmten Zustand im Prozessmodell geleitet wird. Durch die Angabe von Bedingungen, die im Zielzustand gelten müssen, sowie Ereignissen, die auf dem Weg zu diesem Zustand eintreten sollen, erhält der Prozessausführende ein Navigationssystem, mit dem er stets den Überblick behält. Doch nicht nur der ausführende Agent der Prozesse eines Unternehmens kann die Navigation benutzen, sondern auch der Autor des Prozessmodells. Durch die Evaluierung der verschiedenen Ablaufszenarien kann der Modellierer Feedback erhalten, um im Bedarfsfall das Modell gewissen Entscheidungswegen anzupassen.

### 10.1.4 Empfehlungen

Im Kapitel über die Modellierung von prozessbezogenen Empfehlungen wurde angesprochen, inwieweit der Modellierer Einfluss auf die Entscheidungen des Prozessausführenden nehmen kann. Weiterhin kompensiert dieses Konzept zusammen mit der Prozessnavigation das Problem der Überforderung bei zu vielen Entscheidungsmöglichkeiten. Der Prozessausführende erhält durch die berechnete Gewichtung der möglichen Entscheidungen die Gewissheit, sich stets konform den Empfehlungen des Modellierers verhalten zu können.

## 10.2 Imperative Modellierungssprachen

Imperative Modellierungssprachen leiten sich von dem Konzept der imperativen Programmiersprachen ab. Sie benutzen Konstrukte wie etwa if-then-else, for- oder while-Schleifen etc., um Prozessabläufe darzustellen. Durch die Codierung der Geschäftsprozesse in flussbasierten Modellen konzentrieren sich diese PMS hauptsächlich auf den Ablauf der Unternehmensprozesse und weniger auf den Prozess an sich. Eine allgemeine Folge dieser ablaufbasierten Modellierung ist, dass jeder nächste Schritt genau vorgegeben ist und auch explizit modelliert werden muss.

### 10.2.1 BPMN

BPMN [47] ist einer der großen industriellen Vertreter der imperativen PMS. Der IBM-Mitarbeiter Stephen White entwickelte im Jahr 2002 die erste Version. Im Januar 2011 wurde die nächste Version, der Standard BPMN 2.0 [48], von der OMG veröffentlicht. BPMN-Prozessmodelle der ersten Version (1.x) sind nicht direkt ausführbar. Es ist zwar möglich, eine Transformation nach BPEL, einer XML-basierten Sprache zur Ausführung von Geschäftsprozessen, durchzuführen, allerdings ist diese Übersetzung unvollständig [49]. Die Konsequenz daraus ist, dass manche modellierten Konzepte aus BPMN nicht in BPEL übertragen werden können und bei der Transformation verloren gehen. Die Gründe hierfür liegen in den unterschiedlichen Konzeptionen der beiden Sprachen. BPEL ist eine blockstrukturierte, BPMN eine graphbasierte Sprache und kann als Obermenge

von BPEL angesehen werden. Eine Übersetzung von BPEL-Modellen nach BPMN ist ohne Probleme möglich, allerdings ist die Umkehrung, also BPMN nach BPEL, nicht immer möglich (siehe [49]). Es gibt also keine isomorphe Abbildung zwischen den Sprachen. Da hierdurch eine vollständige und semantisch korrekte Ausführung von BPMN 1.x-Modellen nicht möglich ist, kann auch kein Vergleich mit der Prozessnavigation gezogen werden, da dieses Konzept eine Ausführungsumgebung voraussetzt. Durch das in BPMN 2.0 hinzugekommene Serialisierungskonzept besteht nun eine Interoperabilität zwischen den einzelnen BPMN-Werkzeugen. Da BPMN 2.0 noch sehr neu ist, geht man allgemein davon aus, dass zu Beginn nur einfache Prozesse zwischen den BPMN-Modellierungsumgebungen und den WMS ausgetauscht werden können. Es bleibt abzuwarten, wie gut und vor allem wie vollständig (in Bezug auf semantische Korrektheit) die Transformationen verlaufen werden.

In puncto Ausdrucksstärke gilt es zu überprüfen, wie gut und effizient zum einen BPMN die fünf POPM-Perspektiven unterstützt, und zum anderen, ob die Sprache auch erweiterbar ist. Die funktionalen und verhaltensbezogenen Perspektiven sind die am weitesten entwickelten und auch bekanntesten Perspektiven in PMS. Wie bereits einleitend erwähnt, setzen imperative PMS Unternehmensprozesse in ablauforientierte Prozessmodelle um. Die funktionale Perspektive, die sich eigentlich auf den Prozess als Ganzes konzentriert, wird zum Teil in Abläufen außerhalb des eigentlichen Prozesses wiedergegeben. Dies drückt sich zum Beispiel darin aus, dass ein „Prozess B muss einmal mehr als Prozess A ausgeführt werden“ in einem Ablauf codiert wird und weniger im Prozess B selbst. Ähnlich verhält es sich mit der verhaltensbezogenen Perspektive. Die Voraussetzungen, die für die Ausführung eines Prozesses erfüllt sein müssen, werden auch durch Abläufe codiert. Somit wird es schwierig, Flexibilität in ein Prozessmodell zu integrieren, da diese Voraussetzungen auch immer mitmodelliert werden müssen. Zusätzlich gibt es bei den Handlungen, die auf einen Prozess ausgeführt werden, auch keine differenzierte Unterscheidung wie bei **ESP**ro**N**a, das heißt, man kann einen Prozess nicht auf ganz bestimmte Aktionen einschränken. Die in BPMN verfügbaren Prozessaktionen sind automatisch für alle Prozesse anwend- und verfügbar.

Die Frage, ob die Modellierungssprache BPMN individuell erweiterbar ist, muss klar mit Nein beantwortet werden; der Modellierer kann also nicht wie bei iPM$^2$ neue Modellierungssymbole auf Basis von Prozessconstraints definieren. Zwar wurde mit BPMN 2.0 das Konzept der Artefakte eingeführt, sodass es möglich ist, individuelle Symbole in die Prozessmodelle mit aufzunehmen, allerdings haben diese keinen Einfluss auf die Ausführungssemantik und sorgen somit nur für einen optischen Effekt.

In Bezug auf die Priorisierung von Prozessregeln bzw. die Prozessabläufe im Kontext der imperativen PMS lässt sich dieses Konzept in BPMN nur durch Textanmerkungen an den Entscheidungsstellen lösen. Hierdurch werden die Prozessmodelle allerdings unnötig überfrachtet, und eine Evaluierung dieser Empfehlungen zum Ausführungszeitpunkt ist nicht möglich.

### 10.2.2 YAWL

YAWL [50] steht für *Yet Another Workflow Language* und wurde im Jahre 2007 mit *new*YAWL [51] aktualisiert. Mit dem Update wurde eine formale Definition der Sprache ergänzt, die sich auf die sogenannte *Colored Petri Nets* [52] stützt. Der Fokus bei der Neuimplementierung der Sprache lag auf der Umsetzung der *Workflow Patterns* [53]. Ziel war es, mit einer relativ kleinen Menge an Modellierungskonstrukten einen Großteil

dieser Muster in Bezug auf die verhaltensbezogene, datenbezogene und operationale Perspektive zu unterstützen. Die funktionale und organisatorische Perspektive werden nicht explizit bzw. nur sehr rudimentär unterstützt. Agenten können in den Prozessschritten zwar angegeben und Rollendefinitionen und Zugehörigkeiten erstellt werden, allerdings sind diese Strukturen flach, das heißt, hierarchische Beziehungen zwischen den Rollen sowie individuelle Konzepte können nicht erstellt werden. Weiterhin ist es sehr schwierig, Flexibilität zu modellieren, da, wie bei allen imperativen PMS, diese explizit durch Ablaufpfade modelliert werden muss. Somit werden in *new*YAWL erstellte Prozessmodelle mit wachsender Anzahl an Prozessschritten und flexiblen Entscheidungsmöglichkeiten rasch unübersichtlich.

Eine individuelle Erweiterung der Sprache ist nicht vorgesehen, sodass der Modellierer mit der vorgegebenen Menge an Modellierungskonstrukten auskommen muss.

Eine Priorisierung der möglichen Entscheidungen sowie ein Navigationssystem zur Unterstützung von Modellierer und Prozessausführendem sind in der Sprache ebenfalls nicht vorgesehen.

## 10.3 Deklarative Modellierungssprachen

Deklarative Modellierungssprachen stellen den Gegenpart zu den imperativen Sprachen dar. Die Beschreibung eines Problems, also das „WAS ist das Problem“, steht hier klar im Vordergrund. Wurden in den imperativen Modellierungssprachen Problem und Lösung noch vermischt, so ist in den deklarativen Sprachen eine klare Trennung zwischen beiden Welten zu erkennen. Die expliziten Abläufe, die in den imperativen PMS die Lösung des Problems darstellen, sind hier in den Modellen nicht notwendigerweise gegeben. Vielmehr geht es bei der deklarativen Modellierung primär um Regeln und Ziele sowie deren Erfüllung im Kontext der Geschäftsprozesse.

### 10.3.1 DECLARE und ConDec

DECLARE [54] wurde an der technischen Universität Eindhoven entwickelt. Der Programmcode ist unter [55] verfügbar und steht unter der *GNU General Public License* in Version 2 [56]. DECLARE selbst kann mit iPM$^2$ verglichen werden. Es können verschiedene Sprachdialekte referenziert werden. In dem entsprechenden Paper wird gezeigt, wie neue Modellierungssymbole in DECLARE mit Hilfe sogenannter Templates erstellt werden können. Die Definitionen dieser Modellierungssymbole können dabei auf der Basis verschiedener Constraintsprachen erfolgen: DecSerFlow [57] und ConDec [58] werden in [54] beschrieben, wobei die zuerst genannte Sprache den Fokus auf Web Services [59] hat. Deshalb soll hier der Blick auf ConDec gerichtet werden.

Die Constraintsprache ConDec zur Modellierung deklarativer Prozessmodelle, in Verbindung mit DECLARE, basiert auf einem Ansatz der linearen temporalen Logik (LTL) [60]. LTL fundiert auf der Modallogik [36], wobei die Operatoren „notwendig“ und „möglich“ zeitlich interpretiert werden und neue temporale Operatoren auf dieser Basis geschaffen werden. Durch LTL ist es nun möglich, Aussagen über Bedingungen zu treffen, die in der Zukunft, ausgehend von einem bestimmten Standpunkt aus, gelten müssen bzw. können. Aussagen wie etwa „gilt bis zu einem Zeitpunkt“ oder „wird danach immer gelten“ können durch LTL spezifiziert werden.

Die in DECLARE erstellten Modellierungssymbole werden jetzt durch ConDec in LTL abgebildet. Das *response constraint* beispielsweise, das durch einen speziellen Pfeil

ausgedrückt wird und zwei Aktivitäten (Prozesse) miteinander verbindet, wird semantisch durch eine LTL-Formel definiert. Abbildung 10.1 zeigt den Pfeil zusammen mit der temporalen Formel. Die LTL-Formel drückt aus, dass einer Ausführung von Prozess A eventuell eine Ausführung von Prozess B nachfolgt. Hierdurch wird nicht etwa ausgedrückt, dass B optional ist, sondern lediglich, dass nach der Ausführung von A noch andere Prozesse folgen können.


Abbildung 10.1: LTL-Definition eines in DECLARE gezeichneten Pfeils

**ESP**ro**N**a unterscheidet von ConDec in Bezug auf die Definition der Modellierungssymbole, dass diese nicht durch kryptische Formeln, sondern durch noch lesbare Prädikate ausgedrückt werden. In puncto Ausdrucksfähigkeit fällt bei ConDec bzw. DECLARE auf, dass lediglich die funktionale und die verhaltensbezogene Perspektive in den Papern behandelt werden. Es ist zwar möglich, Agenten den Prozessen zuzuweisen, allerdings können keine komplexen Unternehmensstrukturen abgebildet und modelliert werden. Die datenbezogene und die operationale Perspektive sind in DECLARE nicht vorhanden.

Flexibilität kann mit den beiden PMS abgebildet und auch die Erweiterbarkeit der Modellierungssprache um neue Symbole durch den Modellierer vorgenommen werden.

Eine Navigation während der Prozessausführung wird durch die beiden Sprachen nicht zur Verfügung gestellt; auch kann keine Priorisierung der Prozessconstraints vorgenommen werden.

### 10.3.2 EM-BrA$^2$CE

EM-BrA$^2$CE [61] steht als Abkürzung für *Enterprise Modeling using Business Rules, Agents, Activities, Concepts and Events*. Es konzentriert sich bei der Modellierung hauptsächlich auf die verbale Beschreibung von Geschäftsprozessen durch die sogenannte *Business Rules*. Diese Regeln werden im EM-BrA$^2$CE-Vokabular verfasst, sodass sie auch von einem elektronischen Interpreter (Computer) ausgewertet werden können. Das Vokabular selbst ist eine Ergänzung zur *Semantics of Business Vocabulary and Business Rules (SBVR)* [62]. SBVR wurde von der Object Management Group (OMG) veröffentlicht und stellt eine Spezifikation zur Beschreibung des Vokabulars sowie der Verbalisierung von Regeln in Unternehmen bereit. Da SBVR keine prozessbezogenen Konzepte wie Agenten, Aktivitäten etc. unterstützt, wurde es durch das EM-BrA$^2$CE-Vokabular erweitert. Zusätzlich enthält EM-BrA$^2$CE genau wie **ESP**ro**N**a auch das Konzept der Zustandsvariablen, um den Zustand eines Geschäftsprozesses zu beschreiben und in den Prozessregeln sich darauf zu beziehen. Die Ausführung von Modellen erfolgt dabei in einem sogenannten *Colored Petri Net* (CP-Net) [52].

Im Gegensatz zu **ESP**ro**N**a werden Geschäftsprozesse nicht in grafischen, sondern in verbalen Prozessmodellen festgehalten. Ein Beispiel hierfür ist der Spezifikation der verhaltensbezogenen Perspektive aus der EM-BrA$^2$CE-Dokumentation [61] entnommen. Dort wird ein Prozessmodell zur Spezifikation einer Bestellung abgebildet, woraus eine spezielle Prozessregel referenziert werden soll:

*Activities that have type place order, accept order, reject order and ship order must not be performed in parallel.* [1]

Das Beispiel besagt, dass die Prozessschritte *Bestellung durchführen* (place order), *Bestellung akzeptieren* (accept order), *Bestellung zurückweisen* (reject order) und *Bestellung verschicken* (ship order) nicht parallel ausgeführt werden dürfen. Wie an diesem Beispiel zu erkennen ist, werden Schlüsselwörter unterschiedlich gekennzeichnet. Durch diese spezielle Kennzeichnung ist es möglich, ausgewiesenen Textpassagen eine formale Bedeutung zuzuordnen, die dann durch EM-BrA$^2$CE ausgewertet werden kann. Im Einzelnen sind dies:

- Nominale Konzepte werden grün markiert und unterstrichen.
- Verbale Konzepte werden blau markiert.
- Die Markierung von linguistischen Teilen, die eine zusammengesetzte Aussage bilden, ist rot.

Ausdrucksstärke, das heißt, die Umsetzung aller erwähnten POPM-Perspektiven, ist in EM-BrA$^2$CE nur zum Teil vorhanden. Die Dokumentation [61] erklärt ab Kapitel 5.4, wie verhaltensbezogene, datenbezogene und organisatorische Perspektiven umgesetzt werden können. Eine funktionale und eine operationale Perspektive sind im Handbuch nicht explizit ausgewiesen, können aber sicherlich in EM-BrA$^2$CE implementiert werden. In 5.4.15 der Dokumentation wird ein Beispiel für ein Autorisierungsconstraint angeführt. Durch Regeln, zusammen mit dem verbalen Konzept der *has function*, können Agenten bestimmte Rollen zugewiesen werden. Somit ist es in EM-BrA$^2$CE möglich, Unternehmensstrukturen abzubilden und in den Prozessbeschreibungen zu verwenden. Im Gegensatz zu **ESP**ro**N**a werden diese Konzepte nicht in Ontologien, sondern ebenfalls in SBVR modelliert.

Flexibilität kann in EM-BrA$^2$CE beschrieben und umgesetzt werden. In manchen Passagen tauchen in den angegebenen Beispielregeln Textelemente wie *It is necessary that* (siehe 5.4.15) oder *It is not advisable that* (siehe 5.4.16) auf. Hierdurch erreicht man in EM-BrA$^2$CE eine modale Kategorisierung der Regeln in notwendig und empfehlenswert, die für eine Einflussnahme bei einer Navigation benutzt werden könnten. **ESP**ro**N**a verwendet im Gegensatz dazu eine multimodale Gewichtung der Constraints, wodurch die Empfehlungen ein zusätzliches Ranking erhalten. Eine Prozessnavigation bei der Ausführung von Prozessbeschreibungen ist in EM-BrA$^2$CE nicht gegeben.

## 10.4 Zusammenfassung

In diesem Kapitel wurde **ESP**ro**N**a mit anderen Modellierungssprachen und Ausführungsumgebungen verglichen. Der erste Abschnitt des Kapitels zeigte die Konzepte auf, mit denen PMS und WMS aus Industrie und Forschung verglichen wurden. In Tabelle 10.1 sind nochmals alle Modellierungssprachen aufgelistet. Weiterhin wird dort gezeigt, wie gut diese Sprachen die Konzepte aus der Tabelle umsetzen können. Folgende Abkürzungen dafür verwendet:

[1] Unterschiedliche farbliche Kennzeichnungen sowie Textstile tragen zur Kategorisierung der verschiedenen Schlüsselwörter bei.

- Ausdrucksstärke einer PMS mit Bezug auf die Umsetzung aller POPM-Perspektiven (AS)
- Erweiterbarkeit einer PMS durch das Hinzufügen neuer Modellierungskonstrukte und Perspektiven (EW)
- Prozessnavigation zur Unterstützung des Nutzers bei der Prozessausführung (PN)
- Modellierung von subjektiven Empfehlungen des Prozessmodellierers zur Beeinflussung des Prozessausführenden (EM)

Ein + bei der entsprechenden PMS und Konzeption bedeutet, dass es in der entsprechenden Sprache umgesetzt ist. Ein – zeigt, dass es nicht vorhanden ist.

| **Modellierungssprache** | **AS** | **EW** | **PN** | **EM** |
|---|---|---|---|---|
| BPMN | - | - | - | - |
| YAWL | - | - | - | - |
| DECLARE | - | + | - | - |
| EM-BrA$^2$CE | + | + | - | + |

Tabelle 10.1: Bewertung der verglichenen PMS

# Kapitel 11

# Resümee und Ausblick

Die Konzeption und Implementierung **ESP**ro**N**as bietet die Möglichkeit, die im ersten Kapitel angesprochenen Probleme und Anforderungen an eine PMS zu lösen bzw. ihnen gerecht zu werden. Durch den Konzeptwechsel von der imperativen hin zur deklarativen Problemformulierung wird ein sehr kompakter und effizienter Modellierungsstil ermöglicht. Hier werden nun die in Kapitel 1 angesprochenen Probleme nochmals aufgegriffen und die Konzepte, die zu ihrer Lösung beigetragen haben, kurz rekapituliert:

- Das Problem der Komplexität bei agilen Prozessmodellen (Kapitel 1.2.1) wurde durch einen Konzeptwechsel von „Was nicht explizit erlaubt ist, ist verboten“ hin zu „Was nicht verboten ist, ist erlaubt“ erreicht. Die Ausführung eines Prozesses wird somit nur noch dann eingeschränkt, wenn die Validierung einer der Prozessperspektiven, genauer gesagt, mindestens einer der Prozessconstraints fehlschlägt.

- Lösungspfade müssen nicht mehr explizit modelliert werden. Einschränkungen auf Prozessen, realisiert über die sogenannten Prozessconstraints, sind nun direkt in diesen selbst und nicht mehr in Abläufen codiert. Der Prozessmodellierer kann sich auf die Problemstellung konzentrieren (das WAS) und muss sich nicht mehr explizit um die Lösung der Probleme bemühen (das WIE). Dies erleichtert ein Verständnis der Prozessmodelle durch dritte Personen, die die Geschäftsprozesse später nachvollziehen möchten.

- Pfadkomplexität (siehe Abbildung 1.2) konnte durch das Einführen neuer und semantisch angereicherter Modellierungssymbole gelöst werden (siehe Abbildung 1.3). Der Prozessmodellierer erhält somit die Möglichkeit, die PMS individuell den Gegebenheiten des Unternehmens anpassen zu können. Die neu zu definierenden Modellierungskonstrukte müssen lediglich auf Prozessconstraints abgebildet werden, um deren Bedeutung exakt und formal festhalten zu können (siehe hierzu Kapitel 6).

- Das in Kapitel 1.2.2 angesprochenen Problem der schwachen Ausdrucksfähigkeit aktueller PMS und die daraus resultierenden Probleme, dass gewisse Konzepte aus den realen Geschäftsprozessen nicht ins Modell mit aufgenommen werden können, wurde durch die Verwendung von Ontologien gelöst. Hierdurch konnten die Perspektiven soweit flexibel gehalten werden, dass neue Konzepte, wie etwa die Vertretungsrolle (siehe Kapitel 1.2.2), durch Anpassung der Ontologie nachimplementiert werden können. Das angesprochene Problem, dass gewisse Konzepte nicht

umsetzbar waren, sowie die Konsequenz ungenauer Prozessmodelle konnte somit behoben werden.

- Das Sekundärproblem der zu vielen Entscheidungsmöglichkeiten bei zu großer Flexibilität konnte durch das Navigationskonzept kompensiert werden. Hierdurch erhält der Prozessausführende die Gewissheit, bei der Ausführung in allen Situationen stets den Überblick zu behalten. Ausnahmesituationen können in die Navigation zum Ziel mit einbezogen werden, sodass sie für den Ausführenden keine Schwierigkeiten mehr darstellen.

- Durch das Konzept der Empfehlungen ist dem Modellierer die Möglichkeit gegeben, positive und negative Empfehlungen mit in das Modell aufzunehmen, um die Entscheidungen des Prozessausführenden zu beeinflussen. Dieser hat bei deren Befolgung die Gewissheit, sich stets konform den Richtlinien des Unternehmens verhalten zu können. Trotzdem kann er aber auch von diesen Empfehlungen abweichen, um in bestimmten Situationen, die während der Erstellung des Modells nicht vorhersehbar waren, entsprechend reagieren zu können.

Durch die vorliegende Forschungsarbeit ist der Grundstein für eine flexible und ausdrucksstarke PMS sowie ein unterstützendes WMS gelegt. Durch die Integration in die Modellierungsumgebung iPM$^2$ und in die Ausführungsumgebung ProcessNavigator erhalten Modellierer und Prozessausführender die Möglichkeit, **ESP**ro**N**a anhand von alltäglichen Geschäftsprozessen zu evaluieren.

## Offene Forschungsarbeiten

Zukünftige Forschungsarbeiten bzw. offene Fragen, die sich aus dieser Dissertation ergeben haben, gibt es noch bei folgenden Punkten:

- Integration der in den Perspektiven verwendeten Ontologien in das Betriebssystem (datenbezogene und operationale Perspektive). Nur durch die Zusammenarbeit zwischen Betriebssystem und ProcessNavigator auf Basis einer Ontologie können die benötigten Daten und Werkzeuge semantisch umfassend beschrieben und verwaltet werden.

- Die Applikationen für die Organisation des Personals eines Unternehmens müssen einen Export in OWL2-Ontologien unterstützen, um sicherzustellen, dass die Beziehungen zwischen Personen und Rollen in **ESP**ro**N**a ausgewertet werden können. Weiterhin müssen diese Konzepte auch in iPM$^2$ Verwendung finden um eine semantisch korrekte Anbindung zwischen beiden Systemen implementieren zu können.

- Durch die Möglichkeit der Erweiterbarkeit von **ESP**ro**N**a können neue Perspektiven in die Prozessmodellierungssprache sehr einfach integriert werden. Sie können die Spezifikation eines Prozesses so erweitern, dass zukünftige Unternehmenskonzepte in **ESP**ro**N**a als Constraints abgebildet werden können.

- An der Konzeption neuer Modellierungssymbole zur Vereinfachung der Prozessmodelle kann noch zusätzliche Forschungsarbeit geleistet werden. Diese Symbole können nachfolgend in iPM$^2$implementiert und in Prozessconstraints übersetzt werden. Hierdurch erreicht man eine Vereinfachung der erstellten Prozessmodelle, was

zu einem vereinfachten Verständnis und einer verbesserten Übersichtlichkeit der Modelle beiträgt.

Obige Liste ist sicherlich nicht vollständig, und es werden im Laufe der Zeit und Forschung sicherlich noch mehr offene Punkte hinzukommen. Des Weiteren können sicherlich noch mehr konzeptionelle Arbeiten entstehen, beispielsweise im Bereich der POPM-Perspektiven. Hier können zusätzliche Sichtweisen auf den Prozess entwickelt werden, um ihn an die sich veränderten Unternehmensstrukturen und Geschäftsprozesse anzugleichen. **ESP**ro**N**a stellt hierfür die Strukturen und die generelle Möglichkeit der Erweiterbarkeit zur Verfügung, wie bereits in Kapitel 6 gezeigt wurde. Abschließend bleibt zu sagen, dass **ESP**ro**N**a durch seine Erweiterbarkeit, seine Ausdrucksstärke, sein Navigationskonzept und die Möglichkeit zur Modellierung von Empfehlungen viele Features mitbringt, sodass die Anforderungen an heutige Geschäftsprozesse effizient umgesetzt werden können.

# Anhang A

# Installation

```
    _____________ _____________________                 ________
    \_   _____//   _____/\_______   \_________  ____  \       \ _____
     |    __)_ \_____  \  |     ___/\_  __ \/  _ \ /    |    \\__  \
     |        \/        \ |    |     |  | \(  <_> )     |     \/ __ \_
    /________  /________  / |____|     |__|   \____/\____|__   (____  /
             \/         \/                                 \/     \/
+++ Modeling, Execution and Navigation of Declarative Business Processes +++

+++ Loading (modified) OWL2 parser Thea2...
+++ Loading state-space definitions...
+++ Loading debugging tools...
+++ Loading performance measurement tools...
+++ Loading heuristics and solvers...
+++ Loading helper predicates / utilities...
+++ Loading POPM Functional Perspective definitions...
+++ Loading POPM Behavioral Perspective definitions...
+++ Loading POPM Organizational Perspective definitions...
+++ Loading POPM Data Perspective definitions...
+++ Loading POPM Operational Perspective definitions...
+++ Loading process and constraint predicates definitions...
+++ Loading planning components...
+++ Loading visualization (dot-export)...
+++ Loading the process model instantiation / destruction engine...

% (0 warnings)
1 ?-
```

Listing A.1: Terminalausgabe nach erfolgreichem Laden von **ESP**roN**a**

Für die Ausführung von **ESP**roN**a** sind SWI-Prolog und Logtalk zwingende Voraussetzungen. Beide sind kostenlos erhältlich und können unter [45] bzw. [42] geladen werden. Wie diese Programme installiert werden, findet sich in der jeweiligen Dokumentation. Bei der Erstellung vorliegender Dissertation wurden die jeweils aktuellen Versionen SWI-Prolog 5.11.25 und Logtalk 2.43.0 verwendet. Des Weiteren muss vor der Nutzung noch eine Umgebungsvariable gesetzt werden, damit die erzeugte Dokumentation im richtigen Ordner abgelegt wird. Dies geschieht unter Linux, Unix und Mac durch den Befehl `export ESProNa=/Folder/to/ESProNa`, der noch dem Pfad der installierten **ESP**roN**a**-Umgebung angepasst werden muss. Den Befehl kopiert man am besten in die lokale `.profile`-Datei (Mac/Unix) bzw. `.bash_profile`-Datei (Linux). Unter Windows muss die Umgebungsvariable ebenfalls in den Computereinstellungen (System Properties) als benutzerdefinierte Variable hinzugefügt werden. Variablenname

ist `ESProNa`, und der Wert ist der Pfad zum geladenen Quellcode. Nach dem Entpacken der **ESP**ro**N**a-Quelldateien muss eine Shell (Unix Terminal bzw. MS-DOS Eingabeaufforderung) geöffnet und in das Verzeichnis mit den geladenen Dateien gewechselt werden. Dort wird mit Hilfe des Befehls *swilgt -g "{loader}"* eine SWI-Session geöffnet, wodurch **ESP**ro**N**a automatisch geladen wird. Nach erfolgreichem Laden und Übersetzen aller Quelldateien sollte die Ausgabe im Terminal dem Listing A.1 entsprechen.

```
1 ?- logtalk_load(set_up_surgery_plan(loader)).
2
3  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
4     Loading process model: Surgery plan confirmation
5  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
6 +++ Transforming ontologies used within process model...
7 +++ Loading transformed ontologies...
8 +++ Loading process model...
9
10 % (0 warnings)
11 true.
```

Listing A.2: Terminalausgabe nach erfolgreichem Laden des klinischen Prozessmodells

Das Laden eines bestimmten Prozessmodells mit allen referenzierten Ontologien geschieht durch den Befehl `logtalk_load(set_up_surgery_plan(loader)).`, der in die gestartete **ESP**ro**N**a-Session kopiert und bestätigt wird. Hierdurch wird die `loader.lgt`-Datei im Ordner des Prozessmodells geladen. Die Ausgabe hierzu befindet sich in Listing A.2. In Zeile 4 ist zu erkennen, welches Prozessmodell geladen wurde. Es entspricht im vorliegenden Falle dem in dieser Dissertation verwendeten klinischen Ablaufmodell. In den Zeilen 6 und 7 werden die verwendeten Ontologien übersetzt und geladen. Abschließend erfolgt das Kompilieren des eigentlichen Prozessmodells mit den darin enthaltenen Constraints (Zeile 8). Wie in Kapitel 9 angemerkt, befinden sich die Prozessmodelle im Unterordner `process_models`. Jedes Prozessmodell mit allen dazugehörigen Dateien wie etwa Ontologien, PDF des grafischen Modells, Ausgabe der visualisierten Zustandsräume etc. ist nochmals in einem eigenen Ordner gespeichert. Des Weiteren enthält dieser Ordner auch die oben verwendete `loader.lgt`-Datei. Schließlich erfolgt durch das Abarbeiten dieser Datei das Laden der Prozessconstraints, die im Prozessmodell `process_model.lgt` codiert sind. Beispielanfragen, wie etwa die Visualisierung des Zustandsraumes oder die Navigation, sind ebenfalls im jeweiligen Ordner eines Prozessmodells enthalten und befinden sich in der Datei `EXAMPLE_QUERIES.lgt`. Listing A.3 zeigt eine Anfrage aus dieser Datei, mit der eine Visualisierung des partiellen Zustandsraumes erreicht werden kann.

```
1 ?- performance::init,
2    process_planning::initial_state(set_up_surgery_plan, IS),
3    process_planning::goal_state(set_up_surgery_plan, GS),
4    graphviz::start_export('./process_models/set_up_surgery_plan/partial_state_space.dot'),
5    esprona_hill_climbing(6)::solve(set_up_surgery_plan, process_planning, IS, GS, Path,
6                                    Costs),
7    graphviz::finish_export(GS, Path, '"#0066FF"', '"#0066FF"'),
8    performance::report.
```

Listing A.3: Visualisierung des partiellen Zustandsraumes

Der durch die heuristische Suche generierte Zustandsraum ist im *dot*-Dateiformat gespeichert. Die erzeugte Datei des partiellen Zustandsraumes *partial_state_space.dot* wird im Ordner `set_up_surgery_plan` des Prozessmodells abgelegt. Zur Anzeige bzw. Konvertierung dieser Datei in das PDF-Format muss die Software Graphviz installiert sein. Mehr Informationen hierzu finden sich unter [63]. Mit Ausführung des Befehls

*dot -Tpdf partial_state_space.dot > partial_state_space.pdf*

kann in einer Unix-Shell die erzeugte dot-Datei in ein vektorbasiertes und allgemein lesbares PDF umgewandelt werden.

# Anhang B

# Quellcode des klinischen Prozessmodells

In Listing B.1 ist der komplette Quellcode des verwendeten klinischen Prozessmodells aus Abbildung 4.1 mit allen Prozessen dargestellt.

```
:- object('set_up_surgery_plan#pid_0'(_),
    extends(process)).

    /* ---------------------------------------------------------------
       SECTION STATIC FUNCTIONAL SPECIFICATIONS
       --------------------------------------------------------------- */
    :- dynamic.

    /* Declaring process title and description */
    process_title('Anamnese durchführen').
    process_description('In diesem Prozessschritt wird...').

    /* Defining the actions that can be performed. */
    process_actions([start, finish, abort]).

    /* Defining the maximum positive and negative recommendation degree used
       by the heuristic planning algorithms */
    highest_positive_recommendation_degree(3).
    highest_negative_recommendation_degree(-3).

    /* Defining the values range of the process */
    process_domain([], (PID_0 #= 1)) :-
        parameter(1, PID_0).

    /* ---------------------------------------------------------------
       SECTION FUNCTIONAL PERSPECTIVE CONSTRAINTS
       --------------------------------------------------------------- */
    /* Actions must be applicable. */
    functional_constraint(necessity, [Action], State, Instance,
    (
        'set_up_surgery_plan#pid_0'(_)::actions_applicable(State, Action, Instance)
    )).

    /* ---------------------------------------------------------------
       SECTION ORGANIZATIONAL PERSPECTIVE CONSTRAINTS
       --------------------------------------------------------------- */
    /* --- Necessary organizational constraints ----------- */
    organizational_constraint(necessity, [start, finish, abort], _, _, [Agent],
    (
        instantiates_class(Agent, 'org#Person'),
        'org#Cardiology'::'org#member'(Agent)
```

```
42      )).
43
44
45      /* --- Recommended organizational constraints ------- */
46      /* Recommendation: +++ (absolutely recommended) */
47      organizational_constraint(recommendation(-3), [start], _, _, [Agent],
48      (
49          instantiates_class(Agent, 'org#Person'),
50          Agent::'org#plays'('org#AssistantDoctor')
51      )).
52
53      /* Recommendation: + (recommended) */
54      organizational_constraint(recommendation(-1), [abort], _, _, [Agent],
55      (
56          instantiates_class(Agent, 'org#Person'),
57          Agent::'org#plays'('org#AssistantMedicalDirector')
58      )).
59
60      /* Recommendation: --- (absolutely not recommended) */
61      organizational_constraint(recommendation(3), [start, abort], _, _, [Agent],
62      (
63          instantiates_class(Agent, 'org#Person'),
64          Agent::'org#plays'('org#MedicalSuperintendent')
65      )).
66
67      /* Recommendation: ++ (very recommended) */
68      organizational_constraint(recommendation(-2), [finish], State, Instance, AgentList,
69      (
70          'set_up_surgery_plan#pid_0'(_)::instance_agent_data_tool(State, start,
71                                                                   Instance, AgentList, _, _)
72      )).
73
74      /* ----------------------------------------------------------------
75         SECTION DATA PERSPECTIVE CONSTRAINTS
76         ---------------------------------------------------------------- */
77      data_production([start], _, _, [Data],
78      (
79          instantiates_class(Data, 'clinic#Patientenakte')
80      )).
81
82      /* ----------------------------------------------------------------
83         SECTION OPERATIONAL PERSPECTIVE CONSTRAINTS
84         ---------------------------------------------------------------- */
85      operational_constraint([start, finish, abort], _, _, [Tool],
86      (
87          instantiates_class(Tool, 'op#Software'),
88          Tool::'op#labeled'('software#HIS')
89      )).
90
91  :- end_object.
92
93
94  :- object('set_up_surgery_plan#pid_1'(_),
95      extends(process)).
96
97      /* ----------------------------------------------------------------
98         SECTION STATIC FUNCTIONAL SPECIFICATIONS
99         ---------------------------------------------------------------- */
100     :- dynamic.
101
102     /* Declaring process title and description */
103     process_title('Operationsplan vorbereiten').
104     process_description('In diesem Prozessschritt wird...').
105
106     /* Defining the actions that can be performed. */
```

```
107  process_actions([start, finish, abort]).
108
109
110
111  /* Defining the maximum positive and negative recommendation degree used
112     by the heuristic planning algorithms */
113  highest_positive_recommendation_degree(3).
114  highest_negative_recommendation_degree(-3).
115
116  /* Defining the values range of the process */
117  process_domain([], (PID_1 #>= 1)) :-
118      parameter(1, PID_1).
119
120  /* ----------------------------------------------------------------
121     SECTION FUNCTIONAL PERSPECTIVE CONSTRAINTS
122     ---------------------------------------------------------------- */
123  /* Actions must be applicable. */
124  functional_constraint(necessity, [Action], State, Instance,
125  (
126      'set_up_surgery_plan#pid_1'(_)::actions_applicable(State, Action, Instance)
127  )).
128
129  /* ----------------------------------------------------------------
130     BEHAVIORAL PERSPECTIVE CONSTRAINTS
131     ---------------------------------------------------------------- */
132  /* Process PID 0 must be started */
133  behavioral_constraint(necessity, [start], State, _,
134  (
135      'set_up_surgery_plan#pid_0'(_)::exists_instance(State, start, _)
136  )).
137
138  /* ----------------------------------------------------------------
139     SECTION ORGANIZATIONAL PERSPECTIVE CONSTRAINTS
140     ---------------------------------------------------------------- */
141  organizational_constraint(necessity, [start, finish, abort], _, _, [Agent],
142  (
143      instantiates_class(Agent, 'org#Person'),
144      Agent::'org#plays'('org#AssistantDoctor'),
145      'org#Surgery'::'org#member'(Agent)
146  )).
147
148  /* ----------------------------------------------------------------
149     SECTION DATA PERSPECTIVE CONSTRAINTS
150     ---------------------------------------------------------------- */
151  /* ... produces */
152  data_production([finish], _, _, [Data],
153  (
154      instantiates_class(Data, 'clinic#Operationsplan')
155  )).
156
157  /* ... requires */
158  data_constraint(necessity, [start], _, _, RequiredDataItemsList,
159  (
160      RequiredDataItemsList = ['clinic#Patientenakte']
161  )).
162
163  /* ----------------------------------------------------------------
164     SECTION OPERATIONAL PERSPECTIVE CONSTRAINTS
165     ---------------------------------------------------------------- */
166  operational_constraint([start, finish, abort], _, _, [Tool],
167  (
168      instantiates_class(Tool, 'op#Software'),
169      Tool::'op#labeled'('software#HIS')
170  )).
171
```

```
172 :- end_object.
173
174
175
176 :- object('set_up_surgery_plan#pid_2'(_),
177     extends(process)).
178
179     /* ----------------------------------------------------------------
180        SECTION STATIC FUNCTIONAL SPECIFICATIONS
181        ---------------------------------------------------------------- */
182     :- dynamic.
183
184     /* Declaring process title and description */
185     process_title('Operationsplan genehmigen').
186     process_description('In diesem Prozessschritt wird...').
187
188     /* Defining the actions that can be performed. */
189     process_actions([start, finish, abort]).
190
191     /* Defining the maximum positive and negative recommendation degree used
192        by the heuristic planning algorithms */
193     highest_positive_recommendation_degree(3).
194     highest_negative_recommendation_degree(-3).
195
196     /* Defining the values range of the process */
197     process_domain(['set_up_surgery_plan_extended#pid_1'(PID_1)], (PID_2 #= PID_1)) :-
198         parameter(1, PID_2).
199
200     /* ----------------------------------------------------------------
201        SECTION FUNCTIONAL PERSPECTIVE CONSTRAINTS
202        ---------------------------------------------------------------- */
203     /* Actions must be applicable. */
204     functional_constraint(necessity, [Action], State, Instance,
205     (
206         'set_up_surgery_plan#pid_2'(_)::actions_applicable(State, Action, Instance)
207     )).
208
209     /* ----------------------------------------------------------------
210        BEHAVIORAL PERSPECTIVE CONSTRAINTS
211        ---------------------------------------------------------------- */
212     behavioral_constraint(necessity, [start], State, _,
213     (
214         'set_up_surgery_plan#pid_1'(_)::in_domain(State)
215     )).
216
217     /* ----------------------------------------------------------------
218        SECTION ORGANIZATIONAL PERSPECTIVE CONSTRAINTS
219        ---------------------------------------------------------------- */
220     /* Start the process by the supervisor of the person who did pid_1 */
221     organizational_constraint([start, finish, abort], State,  Instance, [Supervisor],
222     (
223         /* get the person who started it */
224         'set_up_surgery_plan#pid_1'(_)::instance_agent_data_tool(State, start, Instance,
225                                                                  StartAgents, _, _),
226         member(StartAgent, StartAgents),
227         instantiates_class(Supervisor, 'org#Person'),
228         Supervisor::'org#supervisorOf'(StartAgent)
229     )).
230
231     /* ----------------------------------------------------------------
232        SECTION DATA PERSPECTIVE CONSTRAINTS
233        ---------------------------------------------------------------- */
234     /* ... produces */
235     data_production([finish], _, _, [Data],
236     (
```

```
237         instantiates_class(Data, 'clinic#Operationsplan'),
238         Data::set_flag('genehmigt', 'true')
239     )).
240
241
242     /* ... requires */
243     data_constraint(necessity, [start], _, _, RequiredDataItemsList,
244     (
245         RequiredDataItemsList = ['clinic#Operationsplan']
246     )).
247
248     /* ---------------------------------------------------------------
249        SECTION OPERATIONAL PERSPECTIVE CONSTRAINTS
250        --------------------------------------------------------------- */
251     operational_constraint([start, finish, abort], _, _, [Tool],
252     (
253         instantiates_class(Tool, 'op#Software'),
254         Tool::'op#labeled'('software#HIS')
255     )).
256
257 :- end_object.
```

Listing B.1: Klinisches Prozessmodell, modelliert in **ESP**ro**N**a

# Anhang C

# Sprachdefinition

## C.1 Funktionale Constraints

Eine Übersicht an funktionalen Constraints, die in den Prozessobjekten angegeben werden können, zeigt Listing C.1. `process_title/1` und `process_description/1` erwarten als Input eine Zeichenkette (`String`). `one` in der `mode`-Direktive eines Constraints besagt, dass es bei dessen Evaluierung genau ein Ergebnis zurückliefert. Das Constraint `process_actions/1` erwartet als Eingabe eine Liste mit spezifizierten Handlungen: `+list(actions)`. Auch dieses Constraint darf im Prozess nur einmal vorkommen, genau wie `process_domain/2`. Letzteres erwartet als ersten Parameter eine Liste mit abhängigen Prozessen. Wird zum Beispiel in einem Prozess `B` spezifiziert, dass dieser genauso oft ausgeführt werden muss wie Prozess `A`, dann ist diese Abhängigkeit hier anzugeben, da sie bei der Unifikation der Domänenvariablen benötigt wird. Als Beispiel dient wieder der Prozess `Operationsplan genehmigen` (`PID 2`) aus dem klinischen Beispielprozessmodell von Abbildung 4.1. Dort wurde modelliert, dass der Prozess genauso oft ausgeführt werden soll wie der Prozess `Operationsplan vorbereiten` (`PID 1`). Demnach müssen also genauso viele Operationspläne genehmigt werden wie erstellt wurden. Listing C.2 gibt die Spezifikation des Constraint aus dem Prozess `PID 2` wieder.

```
1  :- public(process_title/1).
2  :- mode(process_title(+var(string)), one).
3
4  :- public(process_description/1).
5  :- mode(process_description(+var(string)), one).
6
7  :- public(process_actions/1).
8  :- mode(process_actions(+list(actions)), one).
9
10 :- public(process_domain/2).
11 :- mode(process_domain(+list(dependencies), +term(clpfd_domains)), one).
12
13 :- public(functional_constraint/4).
14 :- mode(functional_constraint(+list(actions),
15                                +list(state),
16                                -instance_identifier,
17                                +callable),
18         zero_or_more).
```

Listing C.1: Constraint-Signaturen der funktionalen Perspektive

```
1 process_domain(['set_up_surgery_plan#pid_1'(PID_1)], (PID_2 #= PID_1)) :-
2     parameter(1, PID_2).
```

Listing C.2: Spezifikation abhängiger Wertebereiche zweier Prozesse

Der erste Parameter des `process_domain/2`-Constraints wird als Liste mit Einträgen der Form `Prozess-ID(Prozess-Variable)` gefüllt. Die ermittelte Prozessvariable wird im zweiten Parameter des Constraints für die Spezifikation des `clpfd`-Wertebereichs benötigt. Durch `(PID_2 #= PID_1)` wird ausgedrückt, dass der Prozess `PID 2` genauso oft ausgeführt werden muss (`#=`) wie der Prozess `PID 1`. Mehrere Ausdrücke können logisch miteinander verknüpft werden. Der Wertebereich eines Prozesses mit festen Grenzen (`2..4`) kann beispielsweise wie in Listing C.3 definiert werden.

```
1 process_domain([], (PID_2 #>= 2, PID_2 #=< 4)) :-
2     parameter(1, PID_2).
```

Listing C.3: Spezifikation des Wertebereichs eines Prozesses

Nachdem im zweiten Parameter des Prozesses keine Referenz auf einen anderen Prozess erfolgt, kann die Liste des ersten Parameters leer bleiben. Die Tabellen C.1 und C.2 geben eine Übersicht der möglichen Ausdrücke und Operatoren der `clpfd`-Bibliothek an, die bei der Deklaration der Prozess-Domains benutzt werden können.

| Ausdruck | Grammatik | Bedeutung |
|---|---|---|
| `int` | `int` | `Ganzzahl` |
| `var` | `var` | `Prolog-Variable` |
| `expr` | `expr ::= var \| int` | `expr ist Variable oder Ganzzahl` |

Tabelle C.1: Ausdrücke der `clpfd`-Bibliothek

## C.2 Verhaltensbezogene Constraints

Verhaltensbezogene Constraints beziehen sich auf den Zustand anderer Prozesse. In Listing C.4 ist die Signatur der verhaltensbezogenen Perspektive spezifiziert. Der erste Parameter ist eine Liste mit Handlungen, für die das Constraint gilt. Der zweite Parameter ist der aktuelle Zustand des ausgeführten Prozessmodells. Basierend auf diesem Zustand werden die im vierten Parameter angegebenen Prädikate ausgewertet. Es ist möglich, im dritten Parameter eine bestimmt Instanz des Prozesses anzugeben, sodass bei der Evaluierung nur diese in Betracht gezogen wird. Wird keine Instanz gebunden und der dritte Parameter durch das Wildcard-Symbol ersetzt (_), so gilt das Constraint für alle Instanzen des Prozesses. Am Prefix + der jeweiligen Parameter erkennt man, dass sie alle immer gebunden sein müssen. Im vierten Parameter des Constraints können nun diverse Prädikate referenziert werden, die durch **ESP**ro**N**a zur Verfügung gestellt werden. Im Prozess `Operationsplan vorbereiten` (Listing 4.4, Zeile 3) wurde das Prädikat `exists_instance/3` bereits aufgeführt. Es wertet bei übergebenem Zustand (erster Parameter) aus, ob eine bestimmte Handlung (zweiter Parameter) auf eine Prozessinstanz (dritter Parameter) bereits ausgeführt wurde.

| Logische Operatoren | Assoziativität | Notation | Bedeutung |
|---|---|---|---|
| expr, expr | binär | infix | UND-Verknüpfung |
| expr; expr | binär | infix | ODER-Verknüpfung |

| Arith. Operatoren | Assoziativität | Notation | Bedeutung |
|---|---|---|---|
| -expr | unär | prefix | Negation |
| expr + expr | binär | infix | Addition |
| expr-expr | binär | infix | Subtraktion |
| expr * expr | binär | infix | Multiplikation |
| expr / expr | binär | infix | Ganzzahldivision |
| expr ^ expr | binär | infix | Exponentialoperator |
| min(expr, expr) | binär | prefix | Minimum zweier Ausdrücke |
| max(expr, expr) | binär | prefix | Maximum zweier Ausdrücke |
| expr mod expr | binär | infix | Modulo-Operator |
| abs(expr) | unär | prefix | Betrags-Operator |

| Vergleichsoperatoren | Assoziativität | Notation | Bedeutung |
|---|---|---|---|
| expr #>= expr | binär | infix | größer gleich |
| expr #=< expr | binär | infix | kleiner gleich |
| expr #= expr | binär | infix | gleich |
| expr #\= expr | binär | infix | ungleich |
| expr #> expr | binär | infix | echt größer |
| expr #< expr | binär | infix | echt kleiner |

Tabelle C.2: Operatoren der `clpfd`-Bibliothek

```
1 :- public(behavioral_constraint/4).
2 :- mode(behavioral_constraint(+list(actions),
3                               +list(state),
4                               +instance_identifier,
5                               +callable),
6         zero_or_more).
```

Listing C.4: Constraint-Signatur der verhaltensbezogenen Perspektive

Ein weiteres Prädikat ist `in_domain/1`. Es wird mit Bezug auf einen bestimmten Prozess aufgerufen, um zu überprüfen, ob er abgearbeitet wurde. Einen Beispielaufruf zeigt das Constraint in Listing C.5, wo überprüft wird, ob der Zähler des Prozesses `pid_0` im aktuellen Zustand (`State`) im angegebenen Wertebereich liegt. Im Prozessmodell in Listing 4.1 wurde im Prozess `Anamnese` spezifiziert, dass dieser genau einmal ausgeführt werden muss. Der Wertebereich ist entsprechend `1..1`. Ist der Zähler des im Constraint referenzierten Prozesses `pid_0` innerhalb des Wertebereichs `1..1`, so validiert das aufgerufene Prädikat `in_domain/1` erfolgreich.

## C.3 Organisatorische Constraints

Der Aufbau eines organisatorischen Constraints (siehe Listing C.6) ist ähnlich dem der verhaltensbezogenen Perspektive: Die Signatur der ersten drei Parameter ist identisch, das heißt, eine Liste mit bestimmten Handlungen, der aktuelle Zustand und eine Instanz

```
1 behavioral_constraint([start], State, _,
2 (
3     'set_up_surgery_plan_extended#pid_0'(_)::in_domain(State)
4 )).
```

Listing C.5: Verhaltensbezogenes Constraint `in_domain/1`

werden übergeben. Der vierte Parameter, eine Liste mit Agentenvariablen, kann frei oder gebunden sein. Im ersteren Fall wird bei Auswertung des fünften Parameters derjenige Agent ermittelt, der den Prozess ausführen kann. Im Falle des n-Augenprinzips werden mehrere Agenten eruiert und an die verschiedenen Variablen in der Liste des fünften Parameters gebunden. Nach der Auswertung des Constraints mit gleichzeitiger Ermittlung der Agenten werden diese an das Prädikat `validate_action/3` weitergereicht. Im zweiten Fall, wenn also an die Variable `Agent` ein bestimmter Wert gebunden ist, wird überprüft, ob das angegebene Constraint erfolgreich validiert. Man kann also das Constraint auf zweifache Art und Weise benutzen: zum einen zur Ermittlung, zum anderen aber auch zur Verifikation von Agenten.

```
1 :- public(organizational_constraint/5).
2 :- mode(organizational_constraint(+list(actions),
3                                   +list(state),
4                                   +instance_identifier,
5                                   ?list(agents),
6                                   +callable),
7         zero_or_more).
```

Listing C.6: Constraint-Signatur der organisatorischen Perspektive

Im Prozessmodell in Listing 4.1 sind in Zeile 36 und 37 zwei Anfragen an die Ontologie gerichtet, um diejenigen Agenten zu ermitteln, die den Prozess ausführen können. Das organisatorische Constraint des Prozessmodells validiert dabei zweimal, das heißt, es wird zuerst `Jack` an die Variable `Agent` gebunden und bei der zweiten Auswertung `John`. Listing C.7 zeigt eine Anfrage mit den beiden von **ESP**ro**N**a ermittelten Ergebnissen. In der ersten Zeile wird der Ausgangszustand für die Anfrage erzeugt, der für den Aufruf und die Auswertung des Prädikates `validate_action/3` in Zeile 2 benötigt wird. Nach erfolgreicher Validierung werden zwei Ergebnisse zurückgeliefert, die in Zeile 4 und 5 aufgelistet sind.

```
1 ?- process_planning::initial_state(set_up_surgery_plan, State),
2 pid_0(_)::validate_action(start, State, _-Agent- _- _).
3
4 Agent = ['org#Jack']
5 Agent = ['org#John']
```

Listing C.7: Ermittlung der Agenten

Listing C.8 zeigt den Aufruf der Constraintvalidierung mit einem bestimmten Agenten. An die Variable `Agent` wurde die Identifikation der Person `Jack` gebunden. Somit ist es möglich, für eine bestimmte Person zu überprüfen, ob sie den Prozess ausführen darf. Das Ergebnis der Auswertung des Prädikates wird in Zeile 4 zurückgegeben: `true`. Dies bedeutet, dass `Jack` den Prozess `pid_0` (`Anamnese`) ausführen darf.

```
1 ?- process_planning::initial_state(set_up_surgery_plan, State),
2 pid_0(_)::validate_action(start, State, _-['org#Jack']- _- _).
3
4 true.
```

Listing C.8: Validierung eines bestimmten Agenten (1)

Lässt man nun noch die Handlung ungebunden, ersetzt also den ersten Parameter `start` des Prädikates `validate_action/3` durch eine ungebundene Variable (`Action`), so wird neben der Validierung, ob Jack den Prozess überhaupt ausführen kann, noch ermittelt, welche Handlungen er genau ausführen darf. Listing C.9 zeigt das erwartete Ergebnis. Die Handlungen `finish` und `abort` werden nicht an die Variable `Action` gebunden, da diese Handlungen im initialen Zustand noch nicht möglich sind (siehe funktionales Constraint).

```
1 ?- process_planning::initial_state(set_up_surgery_plan, State),
2    pid_0(_)::validate_action(Action, State, _-['org#Jack']- _- _).
3
4 Action = start
```

Listing C.9: Validierung eines bestimmten Agenten (2)

## C.4 Datenbezogene Constraints

Bei der datenbezogenen Perspektive verhält es sich weitgehend ähnlich wie bei der organisatorischen. Statt der Agenten werden hier diejenigen Daten ermittelt, die zum Ausführen des Prozesses benötigt werden (siehe Listing C.10). Die ermittelten Daten werden als Liste im ungebundenen vierten Parameter (`?list(required_dataitems)`) zurückgegeben. Genau wie bei der organisatorischen Perspektive kann der vierte Parameter auch gebunden (mit bestimmten Werten vorbelegt) sein, um zu überprüfen, ob die in der Liste angegebenen Daten für eine Validierung ausreichend sind.

```
1 :- public(data_constraint/5).
2 :- mode(data_constraint(+list(actions),
3                         +list(state),
4                         +instance_identifier,
5                         ?list(required_dataitems),
6                         +callable),
7         zero_or_more).
```

Listing C.10: Constraint-Signatur der datenbezogenen Perspektive

## C.5 Operationale Constraints

Operationale Constraints modellieren Werkzeuge und Applikationen die zur Ausführung eines Prozess benötigt werden. Sie verwenden das gleiche Muster wie organisatorische und datenbezogene Prozessconstraints. Durch ihre Auswertung werden Werkzeuge bzw. Applikationen anhand der eingesetzten Ontologie ermittelt, mit denen der Prozess ausge-

führt werden kann (siehe Listing C.11). Genau wie bei der organisatorischen und datenbezogenen Perspektive kann der vierte Parameter bidirektional verwendet werden, das heißt, zur Ermittlung, aber auch Verifikation von Werkzeugen.

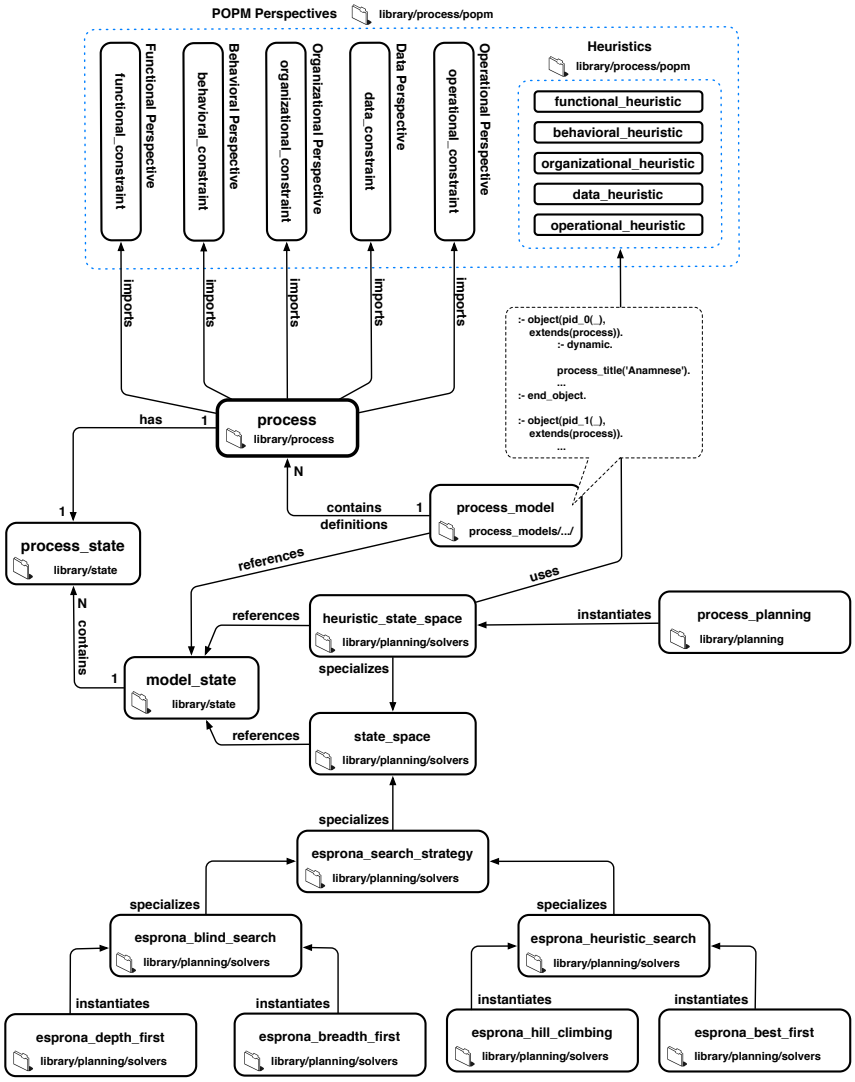
```
1 :- public(operational_constraint/5).
2 :- mode(operational_constraint(+list(actions),
3                                +list(state),
4                                +instance_identifier,
5                                ?list(tools),
6                                +callable),
7         zero_or_more).
```

Listing C.11: Constraint-Signatur der operationalen Perspektive

# Anhang D

# Signaturen der ESProNa-Objekte

Auf den folgenden Seiten sind die Signaturen der einzelnen Objekte aufgelistet. Sie beinhalten neben den `mode`-Direktiven auch eine Übersicht über die in den Objekten enthaltenen Prädikate.



# functional_constraint

*This perspective is conform when the given functional constraints evaluate to true.*

author:
`Michael Igler (michael.igler@uni-bayreuth.de)`
version:
`0.8`
date:
`2010/4/19`

compilation:
`static, context_switching_calls`

*(no dependencies on other files)*

## Public interface

### functional_constraint/4

*Contains the POPM constraints of the functional perspective.*

compilation:
`static`

mode - number of solutions:
`functional_constraint(+list(actions),+modelstate,-instance_identifier, +callable) - zero_or_more`

### functional_constraint/5

*Contains the POPM constraints of the functional perspective.*

compilation:
`static`

mode - number of solutions:
`functional_constraint(+modal_prefix,+list(actions),+modelstate,-instance_identifier,+callable) - zero_or_more`

### functional_constraints_conform/3

*This predicate expects: instantiated Action(+var), instantiated ModelState(+list) and a free or instantiated Instance(?var).*

compilation:
`static`

mode - number of solutions:
`functional_constraints_conform(?action,+modelstate,?instance_identifier) - zero_or_more`

### counter_increasable/1

*Verifies if the counter can be increased by 1. The current counter has to be smaller than supremum of the counter_interval.*

compilation:
`static`

mode - number of solutions:
`counter_increasable(+var) - zero_or_one`

### interval_supremum/1

*Retrieves the interval supremum of a process.*

compilation:
`static`

mode - number of solutions:
`interval_supremum(?var) - zero_or_one`

### interval_infimum/1

*Retrieves the interval infimum of a process.*

compilation:
`static`

mode - number of solutions:
`interval_infimum(?var) - zero_or_one`

### counter_interval/1

*Retrieves the interval of a process.*

compilation:
`static`

mode - number of solutions:
`counter_interval(-interval) - one_or_more`

### correlated_to_other_processes/1

*Checks if the counter_interval of the process is dependent/correlated to other processes.*

compilation:
`static`

mode - number of solutions:
`correlated_to_other_processes(+modelstate) - zero_or_one`

### exists_instance/3

*Verifies if a certain instance has been executed in the given model state.*

compilation:
`static`

mode - number of solutions:
`exists_instance(+modelstate,?action,?instance_identifier) - zero_or_more`

### actions_applicable/3

*Checks of the actions are applicable in the given model state.*

compilation:
`static`

mode - number of solutions:
`actions_applicable(+modelstate,?action,?instance_identifier) - zero_or_more`

### in_domain/1

*Checks if process counter is in the domain interval in a given model state.*

compilation:
`static`

mode - number of solutions:
`in_domain(+modelstate) - zero_or_one`

### all_done/1

*Checks if the counters of all processes build a solution in the given model state.*

compilation:
`static`

mode - number of solutions:
`all_done(+modelstate) - zero_or_one`

### instance_agent_data_tool/6

*Returns which instances are started/finished/aborted in a given model state. Lists of agents, data and tools are also retrieved expressing more detailed information about the execution.*

compilation:
`static`

mode - number of solutions:
`instance_agent_data_tool(+modelstate,+action,?instance_identifier,?list(agent_identifiers),?list(data_identifiers),?list(tool_identifiers)) - zero_or_one`

### optional/0

*Checks if a process is optional.*

compilation:
`static`

mode - number of solutions:
`optional - zero_or_one`

### highest_instance_id/2

*Retrieves highest instance-ID of a process.*

compilation:
`static`

mode - number of solutions:
`highest_instance_id(+list,-number) - zero_or_one`

## Protected interface

*(none)*

## Private predicates

*(none)*

# behavioral_constraint

*This perspective is conform when the given behavioral constraints evaluate to true.*

author:
`Michael Igler (michael.igler@uni-bayreuth.de)`
version:
`0.8`
date:
`2009/10/15`

compilation:
`static, context_switching_calls`

*(no dependencies on other files)*

## Public interface

### behavioral_constraint/4

compilation:
`static`

mode - number of solutions:
`behavioral_constraint(+list(actions),+modelstate,-instance_identifier, +callable) - zero_or_more`

### behavioral_constraint/5

*Contains the POPM constraints of the behavioral perspective.*

compilation:
`static`

mode - number of solutions:
`behavioral_constraint(+modal_prefix,+list(actions),+modelstate,-instance_identifier,+callable) - zero_or_more`

### behavioral_constraints_conform/3

*This predicate expects: instantiated Action(+var), instantiated ModelState(+list), instantiated Instance(+var).*

compilation:
`static`

mode - number of solutions:
`behavioral_constraints_conform(+var(action),+modelstate,?instance_identifier) - zero_or_more`

## Protected interface

*(none)*

## Private predicates

*(none)*

# organizational_constraint

*This perspective is conform when the given organizational constraints evaluate to true.*

author:
`Michael Igler (michael.igler@uni-bayreuth.de)`
version:
`0.8`
date:
`2009/10/15`

compilation:
`static, context_switching_calls`

*(no dependencies on other files)*

## Public interface

### organizational_constraint/5

compilation:
`static`

mode - number of solutions:
`organizational_constraint(+list(actions),+modelstate,+instance_identifier,?list(agent_identifiers),+callable) - zero_or_more`

### organizational_constraint/6

*Contains the POPM constraints of the organizational perspective.*

compilation:
`static`

mode - number of solutions:
`organizational_constraint(+modal_prefix,+list(actions),+modelstate,+instance_identifier,?list(agent_identifiers),+callable) - zero_or_more`

### organizational_constraints_conform/4

*This predicate expects: instantiated Action(+var), instantiated ModelState(+list), instantiated Instance(+var) and free or instantiated AgentList(?list).*

compilation:
`static`

mode - number of solutions:
`organizational_constraints_conform(+action,+modelstate,?instance_identifier,?list(agent_identifiers)) - zero_or_more`

## Protected interface

*(none)*

## Private predicates

*(none)*

# data_constraint

*This perspective is conform when the given data constraints evaluate to true.*

author:
`Michael Igler (michael.igler@uni-bayreuth.de)`
version:
`0.8`
date:
`2009/10/15`

compilation:
`static, context_switching_calls`

*(no dependencies on other files)*

## Public interface

### data_constraint/5

compilation:
`static`

mode - number of solutions:
```
data_constraint(+list(actions),+modelstate,+instance_identifier,?
list(data_identifiers),+callable) - zero_or_more
```

### data_constraint/6

*Contains the POPM constraints of the data perspective.*

compilation:
`static`

mode - number of solutions:
```
data_constraint(+modal_prefix,+list(actions),+modelstate,
+instance_identifier,?list(data_identifiers),+callable) - zero_or_more
```

### data_production/2

compilation:
`static`

### data_constraints_conform/4

*This predicate expects: instantiated Action(+var), instantiated ModelState(+list), instantiated Instance(+var) and free or instantiated DataList(?list).*

compilation:
`static`

mode - number of solutions:
```
data_constraints_conform(+var(action),+modelstate,?instance_identifier,
+list(data_identifiers)) - zero_or_more
```

## Protected interface

*(none)*

## Private predicates

*(none)*

# operational_constraint

*This perspective is conform when the given operational constraints evaluate to true.*

author:
`Michael Igler (michael.igler@uni-bayreuth.de)`
version:
`0.8`
date:
`2009/10/15`

compilation:
`static, context_switching_calls`

*(no dependencies on other files)*

## Public interface

### operational_constraint/5

compilation:
`static`

### operational_constraint/6

*Contains the POPM constraints of the operational perspective.*

compilation:
`static`

### operational_constraints_conform/4

*This predicate expects: instantiated Action(+var), instantiated ModelState(+list), instantiated Instance(+var) and free or instantiated ToolList(?list).*

compilation:
`static`

mode - number of solutions:
```
operational_constraints_conform(+var(action),+modelstate,?instance_identifier,
+list(tool_identifiers)) - zero_or_more
```

## Protected interface

*(none)*

## Private predicates

*(none)*

# functional_heuristic

*Predicates for heuristic searching and planning algorithms of the functional perspective.*

author:
`Michael Igler (michael.igler@uni-bayreuth.de)`
version:
`0.8`
date:
`2011/4/6`

compilation:
`static, context_switching_calls`

implements:
`public heuristic_protocol`

## Public interface

*(see related entities)*

## Protected interface

*(see related entities)*

## Private predicates

### costs_do_process/4

*Calculates the costs for executing a given process ID.*

compilation:
`static`

mode - number of solutions:
`costs_do_process(+process_object,+modelstate,+var(action),-number) - zero_or_one`

# behavioral_heuristic

*Predicates for heuristic searching and planning algorithms of the behavioral perspective.*

author:
`Michael Igler (michael.igler@uni-bayreuth.de)`
version:
`0.8`
date:
`2011/4/6`

compilation:
`static, context_switching_calls`

implements:
`public heuristic_protocol`

## Public interface

### set_up_graph/2

*Sets up a graph representing the dependencies between the different processes.*

compilation:
`static`

mode - number of solutions:
`set_up_graph(+var(action),-graph) - one`

## Protected interface

*(see related entities)*

## Private predicates

*(see related entities)*

# organizational_heuristic

*Predicates for heuristic searching and planning algorithms of the organizational perspective.*

author:
`Michael Igler (michael.igler@uni-bayreuth.de)`
version:
`0.8`
date:
`2011/4/6`

compilation:
`static, context_switching_calls`

extends:
`public heuristic_perspective`

## Public interface

*(see related entities)*

## Protected interface

*(see related entities)*

## Private predicates

*(see related entities)*

# data_heuristic

*Predicates for heuristic searching and planning algorithms of the data perspective.*

author:
`Michael Igler (michael.igler@uni-bayreuth.de)`
version:
`0.8`
date:
`2011/4/6`

compilation:
`static, context_switching_calls`

extends:
`public heuristic_perspective`

## Public interface

*(see related entities)*

## Protected interface

*(see related entities)*

## Private predicates

*(see related entities)*

# operational_heuristic

*Predicates for heuristic searching and planning algorithms of the operational perspective.*

author:
`Michael Igler (michael.igler@uni-bayreuth.de)`
version:
`0.8`
date:
`2011/4/6`

compilation:
`static, context_switching_calls`

extends:
`public heuristic_perspective`

## Public interface

*(see related entities)*

## Protected interface

*(see related entities)*

## Private predicates

*(see related entities)*

# process

*Describes the process object as a composition of all POPM perspectives.*

author:
`Michael Igler (michael.igler@uni-bayreuth.de)`
version:
`0.8`
date:
`2011/3/3`

compilation:
`static, context_switching_calls`

imports:
```
public functional_constraint
public behavioral_constraint
public organizational_constraint
public data_constraint
public operational_constraint
```

## Public interface

### process_title/1

*The title of the process.*

compilation:
`static`

mode - number of solutions:
`process_title(-atom) - one`

### process_description/1

*The description of the process.*

compilation:
`static`

mode - number of solutions:
`process_description(-atom) - one`

### contained_in_process_model/1

*Stores in which process model this process is contained.*

compilation:
`static`

mode - number of solutions:
`contained_in_process_model(-process_model) - one`

### process_actions/1

*Declaring the actions that can be performed on the process.*

compilation:
`static`

mode - number of solutions:
`process_actions(+list(actions)) - one`

### process_domain/2

*The domain of the process is stored here.*

compilation:
`static`

mode - number of solutions:
`process_domain(+list(dependencies),+term(clpfd_domains)) - one`

### highest_positive_recommendation_degree/1

...

compilation:
`static`

mode - number of solutions:
`highest_positive_recommendation_degree(-number) - one`

### highest_negative_recommendation_degree/1

...

compilation:
`static`

mode - number of solutions:
`highest_negative_recommendation_degree(-number) - one`

### validate_action/3

...

compilation:
`static`

mode - number of solutions:
`validate_action(?var(action),+modelstate,+compound_term(concept_identifiers)) - zero_or_more`

### perform_action/4

...

compilation:
`static`

mode - number of solutions:
`perform_action(+var(action),+modelstate,+compound_term(concept_identifiers),?list(state)) - zero_or_one`

### id/1

*Returns the process identifier. Example query: pid_1(_)::id(MyID).*

compilation:
`static`

mode - number of solutions:
`id(?var(process_identifier)) - zero_or_one`

## Protected interface

*(see related entities)*

## Private predicates

*(see related entities)*

# process_state(A,B)

*Represents the state of a process containing Process_ID, Process_History_List, Process_Status.*

author:
`Michael Igler (michael.igler@uni-bayreuth.de)`
version:
`0.8`
date:
`2010/4/19`

compilation:
`static, context_switching_calls`

*(no dependencies on other files)*

## Public interface

### get_highest_instance_id/1

*Retrieves the highest instance-ID of the process.*

compilation:
`static`

mode - number of solutions:
`get_highest_instance_id(-number) - one`

### update_process_state/2

*Updates a process state.*

compilation:
`static`

mode - number of solutions:
`update_process_state(+compound_term(concept_identifiers),-process_state) - one`

### instance_successtype_agents_data_tools/5

*Retrieves information about a certain execution.*

compilation:
`static`

mode - number of solutions:
```
instance_successtype_agents_data_tools(+instance_identifier,+action,+agent,
+data,+tool) - zero_or_one
```

### instance_successtype_agents_data_tools_counter/6

*Counts the executions by a certain person with certain data and tools.*

compilation:
`static`

mode - number of solutions:
```
instance_successtype_agents_data_tools_counter(+instance_identifier,+action,
+agent,+data,+tool,-number) - zero_or_one
```

### print/0

*Prints some informations about the state.*

compilation:
`static`

mode - number of solutions:
`print - one`

### print/1

*Prints some informations about the process state to a stream alias.*

compilation:
`static`

mode - number of solutions:
`print(+var) - one`

## Protected interface

*(none)*

## Private predicates

*(none)*

# model_state(A,B)

*Represents the state of the process model.*

author:
`Michael Igler (michael.igler@uni-bayreuth.de)`
version:
`0.8`
date:
`2010/4/19`

compilation:
`static, context_switching_calls`

*(no dependencies on other files)*

## Public interface

### generate_initial_model_state/2

*Generates the initial ModelState.*

compilation:
`static`

mode - number of solutions:
`generate_initial_model_state(+process_model,?model_state) - zero_or_one`

### generate_goal_model_state/2

*Generates the goal ModelState.*

compilation:
`static`

mode - number of solutions:
`generate_goal_model_state(+process_model,?model_state) - zero_or_one`

### update_model_state/3

*Updates the ProcessState of a certain process. A new HistoryList of that process is calculated.*

compilation:
`static`

mode - number of solutions:
`update_model_state(?process_identifier,+compound_term(concept_identifiers),?model_state) - zero_or_one`

### get_process_state/2

compilation:
`static`

mode - number of solutions:
`get_process_state(?process_identifier,?process_state) - zero_or_one`

### model_state_changes/7

*Retrieves elements that have changed between two different model states.*

compilation:
`static`

mode - number of solutions:

```
model_state_changes(+state,?process_identifier,?instance_identifier,?action,?
agents,?data,?tools) - zero_or_one
```

### print/0

*Prints some informations about the model state.*

compilation:

```
static
```

mode - number of solutions:

```
print - one
print - one
```

### print/1

*Prints some informations about the model state into a stream.*

compilation:

```
static
```

## Protected interface

*(none)*

## Private predicates

*(none)*

# state_space

*State space description predicates including navigation.*

author:
`Written by Paulo Moura, adopted to ESProNa by Michael Igler`
version:
`1.2`
date:
`2011/3/20`

compilation:
`static, context_switching_calls`

instantiates:
`public class`
specializes:
`public object`

## Public interface

### initial_state/2

*Initial state.*

compilation:
`static`

template:
`initial_state(ProcessModelID,State)`

mode - number of solutions:
`initial_state(+process_model,?nonvar) - one_or_more`

### goal_state/2

*Goal state.*

compilation:
`static`

template:
`goal_state(ProcessModelID,State)`

mode - number of solutions:
`goal_state(+process_model,?nonvar) - one_or_more`

### next_state/3

*Generates a state sucessor.*

compilation:
`static`

template:
`next_state(ProcessModelID,State,Next)`

mode - number of solutions:
`next_state(+process_model,+nonvar,-nonvar) - zero_or_more`

### member_path/2

*True if a state is member of a list of states.*

compilation:
`static`

template:
`member_path(State,Path)`

mode - number of solutions:
`member_path(+nonvar,+list) - zero_or_one`

### navigate/5

...

compilation:
`static`

mode - number of solutions:
```
navigate(+process_model,+list(state),?list(state),?list(path),
+term(conditions)) - zero_or_more
```

### print_state/1

*Pretty print state.*

compilation:
`static`

template:
`print_state(State)`

mode - number of solutions:
`print_state(+nonvar) - one`

### print_path/1

*Pretty print a path (list of states).*

compilation:
`static`

template:
`print_path(Path)`

mode - number of solutions:
`print_path(+list) - one`

## Protected interface

*(see related entities)*

## Private predicates

*(see related entities)*

# heuristic_state_space

*Heuristic state space description predicates including navigation.*

author:
`Written by Paulo Moura, adopted to ESProNa by Michael Igler`
version:
`1.1`
date:
`2011/3/20`

compilation:
`static, context_switching_calls`

instantiates:
`public class`
specializes:
`public state_space`

## Public interface

### next_state/4

*Generates a state sucessor.*

compilation:
`static`

template:
`next_state(ProcessModelID,State,Next,Cost)`

mode - number of solutions:
`next_state(+process_model,+nonvar,-nonvar,-number) - zero_or_more`

### heuristic/2

*Estimates state distance to a goal state.*

compilation:
`static`

template:
`heuristic(State,Estimate)`

mode - number of solutions:
`heuristic(+nonvar,-number) - one`

### navigate/6

...

compilation:
`static`

mode - number of solutions:
```
navigate(+process_model,+list(state),?list(state),?list(path),?var(costs),
+term(conditions)) - zero_or_more
```

## Protected interface

*(see related entities)*

## Private predicates

*(see related entities)*

# process_planning

*Process planning object to enable ProcessNavigation in ESProNa.*

author:
`Michael Igler (michael.igler@uni-bayreuth.de)`
version:
`0.8`
date:
`2011/3/3`

compilation:
`static, context_switching_calls, events`
instantiates:
`public heuristic_state_space`

## Public interface

*(see related entities)*

## Protected interface

*(see related entities)*

## Private predicates

*(see related entities)*

# esprona_search_strategy

*State space search strategies.*

author:
`Written by Paulo Moura, adopted to ESProNa by Michael Igler`
version:
`1.0`
date:
`2009/12/3`

compilation:
`static, context_switching_calls`

instantiates:
`public abstract_class`
specializes:
`public object`

## Public interface

### solve/5

*State space search solution.*

compilation:
`static`

template:
`solve(ProcessModelID,Space,FromState,ToState,Path)`

mode - number of solutions:
`solve(+process_model,+object,+nonvar,+nonvar,-list) - zero_or_more`

## Protected interface

*(see related entities)*

## Private predicates

*(see related entities)*

# esprona_blind_search(Bound)

*Blind search state space strategies.*

author:
`Written by Paulo Moura, adopted to ESProNa by Michael Igler`
version:
`1.0`
date:
`2009/12/3`

compilation:
`static, context_switching_calls`

instantiates:
`public class`
specializes:
`public esprona_search_strategy`

## Public interface

### bound/1

*Search depth bound.*

compilation:
`static`

template:
`bound(Bound)`

mode - number of solutions:
`bound(?integer) - zero_or_one`

## Protected interface

### search/6

*State space search solution.*

compilation:
`static`

template:
`search(ProcessModelID,Space,FromState,ToState,Bound,Path)`

mode - number of solutions:
`search(+process_model,+object,+nonvar,+nonvar,+integer,-list) - zero_or_more`

## Private predicates

*(see related entities)*

# esprona_heuristic_search(Threshold)

*Heuristic state space search strategies.*

author:
`Written by Paulo Moura, adopted to ESProNa by Michael Igler`
version:
`1.0`
date:
`2009/12/3`

compilation:
`static, context_switching_calls`

instantiates:
`public class`
specializes:
`public esprona_search_strategy`

## Public interface

### threshold/1

*Search cost threshold.*

compilation:
`static`

template:
`threshold(Threshold)`

mode - number of solutions:
`threshold(?number) - one`

### solve/6

*State space search solution.*

compilation:
`static`

template:
`solve(ProcessModel,Space,State,GoalState,Path,Cost)`

mode - number of solutions:
`solve(+process_model,+object,+nonvar,+nonvar,-list,-number) - zero_or_more`

## Protected interface

### search/7

*State space search solution.*

compilation:
`static`

template:
`search(ProcessModel,Space,State,GoalState,Threshold,Path,Cost)`

mode - number of solutions:
`search(+process_model,+object,+nonvar,+nonvar,+number,-list,-number) - zero_or_more`

## Private predicates

*(see related entities)*

# esprona_depth_first(Bound)

*Depth first state space search strategy.*

author:
`Written by Paulo Moura, adopted to ESProNa by Michael Igler`
version:
`1.2`
date:
`2009/12/3`

compilation:
`static, context_switching_calls, events`

instantiates:
`public esprona_blind_search(Bound)`
uses:
`list`

## Public interface

*(see related entities)*

## Protected interface

*(see related entities)*

## Private predicates

*(see related entities)*

# esprona_breadth_first(Bound)

*Breadth first state space search strategy.*

author:
`Written by Paulo Moura, adopted to ESProNa by Michael Igler`
version:
`1.2`
date:
`2009/12/3`

compilation:
`static, context_switching_calls, events`

source:
`Example adapted from the book "Prolog Programming for Artificial Intelligence"`
`by Ivan Bratko.`

instantiates:
`public esprona_blind_search(Bound)`
uses:
`list`

## Public interface

*(see related entities)*

## Protected interface

*(see related entities)*

## Private predicates

*(see related entities)*

# esprona_hill_climbing(Threshold)

*Hill climbing heuristic state space search strategy.*

author:
`Written by Paulo Moura, adopted to ESProNa by Michael Igler`
version:
`1.2`
date:
`2009/12/3`

compilation:
`static, context_switching_calls, events`

instantiates:
`public esprona_heuristic_search(Threshold)`
uses:
`list`

## Public interface

*(see related entities)*

## Protected interface

*(see related entities)*

## Private predicates

### hill/9

compilation:
`static`

# esprona_best_first(Threshold)

*Best first heuristic state space search strategy.*

author:
`Written by Paulo Moura, adopted to ESProNa by Michael Igler`
version:
`1.2`
date:
`2008/6/9`

compilation:
`static, context_switching_calls, events`

source:
`Example adapted from the book "Prolog Programming for Artificial Intelligence"`
`by Ivan Bratko.`

instantiates:
`public esprona_heuristic_search(Threshold)`
uses:
`list`

## Public interface

*(see related entities)*

## Protected interface

*(see related entities)*

## Private predicates

### expand/10
compilation:
`static`

### succlist/5
compilation:
`static`

### bestf/3
compilation:
`static`

### continue/11
compilation:
`static`

### f/2
compilation:
`static`

### insert/4
compilation:
`static`

# Abbildungsverzeichnis


1.1 Drei Prozesse in bestimmter Ausführungskomposition . . . . . 17
1.2 Drei Prozesse in beliebiger Ausführungsreihenfolge . . . . . 18
1.3 Vereinfachung der Komplexität durch neues Modellierungssymbol . . . . . 18
1.4 Explizite Lösungswege bei imperativen Modellierungssprachen . . . . . 21
1.5 „Verbotene Zonen“ bei deklarativen Modellierungssprachen . . . . . 22
1.6 Zusätzliche Pfade sind bereits enthalten . . . . . 22
1.7 Deklarative Formulierung des Prozessmodells 1.1 . . . . . 23
1.8 Deklarative Komposition der Prozessmodelle aus den Abbildungen 1.1 und 1.2 . . . . . 23

2.1 Architekturübersicht . . . . . 28
2.2 Screenshot iPM$^2$ mit **ESP**ro**N**a-Prozessmodell . . . . . 29
2.3 Screenshot ProcessNavigator . . . . . 30

3.1 Sudoku-Beispiel (links) mit ungebundenen Variablen A bis I (rechts) . . . 34
3.2 Lösung des Sudoku-Beispiels . . . . . 36
3.3 Schematischer Aufbau eines Prozesses . . . . . 37
3.4 Auswertung aller Perspektiven . . . . . 38
3.5 Hinzufügen einer neuen Perspektive mit Auswertung des neuen Konzepts . 39
3.6 Deklarativer Prozess im Detail . . . . . 40
3.7 Definition der möglichen Handlungen eines deklarativen Prozesses . . . . . 41

4.1 Prozessmodell für den klinischen Anwendungsfall . . . . . 46
4.2 Auszug aus einer klinischen Ontologie . . . . . 54

6.1 Übersicht über die bisher definierten Modellierungssymbole . . . . . 66
6.2 Verwendung der definierten Modellierungssymbole . . . . . 67
6.3 Mögliche Abläufe des Prozessmodells aus Abbildung 6.2 . . . . . 67
6.4 Prozessmodell mit verhaltensbezogener Aggregation . . . . . 68
6.5 Ablaufsemantik des Prozessmodells aus Abbildung 6.4 . . . . . 68

7.1 Initialzustand mit Folgezuständen . . . . . 78
7.2 Lösungsweg für das klinische Prozessmodell vom Initial- zum Endzustand 80
7.3 Zustandsraum der Navigation zwischen bestimmten Prozesszuständen . . 84
7.4 Prozessmodell zur Verdeutlichung der Navigationsinteraktion . . . . . 85
7.5 Informationsausgabe bei Vorauswahl von Prozess C . . . . . 85
7.6 Informationsausgabe beim Anklicken von Prozess B . . . . . 86
7.7 Klinisches Minimalbeispiel mit durchgezogenem Pfeil . . . . . 86

7.8 Klinisches Minimalbeispiel (vgl. 7.7) mit gestricheltem Pfeil . . . . . 87

8.1 Relation zwischen den Welten x und y . . . . . 90
8.2 Semantik der Modaloperatoren □ (links) und ◇(rechts) . . . . . 90
8.3 Semantik der Modaloperationen . . . . . 91
8.4 Kostenabbildung für notwendige und empfohlene Constraints . . . . . 92
8.5 Visualisierte Welten für die Prozessregeln der Handlung `start` . . . . . 95
8.6 Modifiziertes Prozessmodell aus Abbildung 6.4 . . . . . 97
8.7 Abhängigkeitsgraph im Initialzustand (links) und nach erfolgreicher Ausführung von Prozess A (rechts) . . . . . 97
8.8 Heuristische Navigation unter Vermeidung unnötiger Ausführungen . . . . 98

9.1 Ordnerstrukturen in **ESP**ro**N**a . . . . . 102
9.2 Architekturübersicht **ESP**ro**N**a . . . . . 104

10.1 LTL-Definition eines in DECLARE gezeichneten Pfeils . . . . . 111

# Quellcodeverzeichnis


3.1 Aufruf und Ergebnis des Algorithmus . . . 34
3.2 Modellierte Regeln des Sudoku-Rätsels in der Sprache Prolog . . . 35
4.1 Prozess `P0`, modelliert in **ESP**ro**N**a . . . 50
4.2 Wertebereiche mit Ergebnis, formuliert in `clpfd`-Syntax . . . 51
4.3 Aufruf des Prädikats `actions_applicable/3` . . . 52
4.4 Verhaltensbezogenes Constraint im Prozess `P1` . . . 52
4.5 Modellierte Datenabhängigkeit im Prozess `P1` . . . 55
4.6 Ermittlung aller .rtf-fähigen Textverarbeitungsprogramme . . . 55
5.1 Prädikat `validate_action/3` . . . 60
5.2 Aufruf des Prädikates `validate_action/3` . . . 61
5.3 Signatur des Prädikates `perform_action/4` . . . 61
6.1 **ESP**ro**N**a-Constraint im Prozess B für den durchgezogenen Pfeil . . . 69
6.2 **ESP**ro**N**a-Constraint im Prozess D . . . 69
6.3 **ESP**ro**N**a-Code für Prozess D . . . 70
6.4 Prädikat `validate_action/3` . . . 70
6.5 Import der einzelnen Perspektiven in den Prozess . . . 71
6.6 Auszug aus dem Quellcode der neuen QoS-Perspektive . . . 71
6.7 `perform_action/4` mit neuer QoS-Perspektive . . . 72
6.8 Angepasste `loader`-Datei . . . 72
7.1 Definition des Initialzustandes . . . 76
7.2 Folgezustand nach Starten des Prozesses `Anamnese` . . . 77
7.3 Definition des Endzustandes . . . 77
7.4 Validierung aller ausführbaren Prozesse . . . 78
7.5 Zustandsübergangsrelation `next_state/2` . . . 79
7.6 Aufruf und Performanzreport der Breitensuche . . . 81
7.7 Aufruf und Performanzreport der Tiefensuche . . . 81
7.8 Navigation zwischen bestimmten Prozesszuständen . . . 82
7.9 Aufruf der Navigation in **ESP**ro**N**a . . . 83
8.1 Constraints ohne modales Präfix sind automatisch notwendige Constraints 92
8.2 Empfehlungen der organisatorischen Perspektive . . . 93
8.3 Zustandsübergangsrelation `next_state/3` . . . 94
8.4 Aufruf der Zustandsübergangsrelation `next_state/3` . . . 96
8.5 Ermittlung der Ausführungskosten der Prozesse A und B im Initialzustand 97
9.1 Laden eines klinischen Prozessmodells . . . 103
9.2 Laden des klinischen Prozessmodells im Detail . . . 103
9.3 Welche Prozesse sind im Initialzustand `IS` ausführbar? . . . 103
A.1 Terminalausgabe nach erfolgreichem Laden von **ESP**ro**N**a . . . 119
A.2 Terminalausgabe nach erfolgreichem Laden des klinischen Prozessmodells 120

A.3 Visualisierung des partiellen Zustandsraumes . . . . . 120
B.1 Klinisches Prozessmodell, modelliert in **ESP**ro**N**a . . . . . 123
C.1 Constraint-Signaturen der funktionalen Perspektive . . . . . 129
C.2 Spezifikation abhängiger Wertebereiche zweier Prozesse . . . . . 130
C.3 Spezifikation des Wertebereichs eines Prozesses . . . . . 130
C.4 Constraint-Signatur der verhaltensbezogenen Perspektive . . . . . 131
C.5 Verhaltensbezogenes Constraint `in_domain/1` . . . . . 132
C.6 Constraint-Signatur der organisatorischen Perspektive . . . . . 132
C.7 Ermittlung der Agenten . . . . . 132
C.8 Validierung eines bestimmten Agenten (1) . . . . . 133
C.9 Validierung eines bestimmten Agenten (2) . . . . . 133
C.10 Constraint-Signatur der datenbezogenen Perspektive . . . . . 133
C.11 Constraint-Signatur der operationalen Perspektive . . . . . 134

# Tabellenverzeichnis

1.1 Sprachbezogene Umsetzung des Autorisierungspatterns . . . . . 20

10.1 Bewertung der verglichenen PMS . . . . . 113

C.1 Ausdrücke der `clpfd`-Bibliothek . . . . . 130
C.2 Operatoren der `clpfd`-Bibliothek . . . . . 131

# Literaturverzeichnis


[1] Klaus Hörmann, Lars Dittmann, Bernd Hindel, and Markus Müller. *Spice in der Praxis: Interpretationshilfe für Anwender und Assessoren.* Dpunkt Verlag, März 2006.

[2] ISO/IEC. *Process assessment - Part 2: Performing an assessment.* Technical report, International Organization for Standardization, 2003.

[3] Howard Smith and Peter Fingar. *Business Process Management: The Third Wave.* Meghan-Kiffer Press, 2003.

[4] Stephanie Meerkamm. *The Concept of Process Management in Theory and Practice - A Qualitative Analysis.* In Stefanie Rinderle-Ma, Shazia Wasim Sadiq, and Frank Leymann, editors, *Business Process Management Workshops*, volume 43 of Lecture Notes in Business Information Processing, pages 429–440. Springer, 2009.

[5] Katrin Stein. *Integration von Anwendungsprozessmodellierung und Workflow-Management*, volume 32. Universität Erlangen-Nürnberg, 1999.

[6] Krasimira P. Stoilova and Todor A. Stoilov. *Evolution of the workflow management systems.* In *Scientific Conference on Information, Communication and Energy Systems and Technologies - ICEST*, pages 225–228, 2006.

[7] Stefan Jablonski and Christoph Bussler. *Workflow Management: Modeling Concepts, Architecture and Implementation.* International Thomson Computer Press, September 1996.

[8] W.M.P. van der Aalst, Arthur H. M. ter Hofstede, and Nick Russell. *Workflow Pattern* [online]. Available from: `http://www.workflowpatterns.com`.

[9] Dirk Fahland, Jan Mendling, Hajo Reijers, Barbara Weber, Matthias Weidlich, and Stefan Zugal. *Declarative vs. Imperative Process Modeling Languages: The Issue of Maintainability.* In Bela Mutschler, Roel Wieringa, and Jan Recker, editors, *1st International Workshop on Empirical Research in Business Process Management (ER-BPM'09)*, pages 65–76, Ulm, Germany, September 2009. (LNBIP to appear).

[10] M. Umeda. *Declarative programming for knowledge management: 16th International Conference on Applications of Declarative Programming and Knowledge Management, INAP 2005, Fukuoka, Japan, October 22-24, 2005: revised selected papers.* Lecture notes in artificial intelligence. Springer, 2006.

[11] J. W. Lloyd. *Practical Advantages of Declarative Programming.* In *Joint Conference on Declarative Programming*, 1994.

[12] Thom Frühwirth und Slim Abdennadher. *Constraint-Programmierung.* Springer-Verlag, Berlin/Heidelberg/New York, 1997.

[13] Paulo Moura. *Logtalk — Design of an Object-Oriented Logic Programming Language.* PhD thesis, Department of Computer Science, University of Beira Interior, Portugal, September 2003.

[14] Jan Wielemaker. *An overview of the SWI-Prolog Programming Environment.* In Fred Mesnard and Alexander Serebenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, December 2003. Katholieke Universiteit Leuven. CW 371.

[15] Ulf Nilsson and Jan Maluszynski. *Logic, Programming, and PROLOG.* John Wiley & Sons, Inc., New York, NY, USA, 1995.

[16] Richard A. O'Keefe. *The craft of Prolog.* MIT Press, Cambridge, MA, USA, 1990.

[17] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall PTR, 2nd edition, March 2000. Available from: `http://www.worldcat.org/isbn/0136291554`.

[18] Henry Liebermann. *Using Prototypical Objects to Implement Shared Behaviour in Object-oriented Systems.* SIGPLAN Notices, 21(11):214–223, November 1986.

[19] Conrad Bock, Achille Fokoue, Peter Haase, Rinke Hoekstra, Ian Horrocks, Alan Ruttenberg, Uli Sattler, and Mike Smith. *OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax* [online]. June 2009. Available from: `http://www.w3.org/TR/2009/CR-owl2-syntax-20090611`.

[20] Holger Knublauch, Ray W. Fergerson, Natalya Fridman Noy, and Mark A. Musen. *The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications.* In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *International Semantic Web Conference*, volume 3298 of Lecture Notes in Computer Science, pages 229–243. Springer, 2004.

[21] B. Volz. *Werkzeugunterstützung für methodenneutrale Metamodellierung.* PhD thesis, University of Bayreuth, Bayreuth, Germany, July 2011.

[22] H. Meerkamm and K. Paetzold. *FORFLOW - Bayerischer Forschungsverbund für Prozess- und Workflowunterstützung zur Planung und Steuerung der Abläufe in der Produktentwicklung (2. Ergebnisbericht).* Print GmbH, München, 2008.

[23] Matthias Faerber. *Prozessorientiertes Qualitätsmanagement.* Gabler research, 1. edition, 2010.

[24] Stefan Jablonski. *Functional and behavioral aspects of process modeling in Workflow Management Systems.* In *CON'94: Proceedings of the Ninth Austrian-informatics conference on Workflow management: challenges, paradigms and products*, pages 113–133. R. Oldenbourg Verlag GmbH, Munich, Germany, 1994.

[25] James Snell, Doug Tidwell, and Pavel Kulchenko. *Programming Web services with SOAP.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[26] D. Kristol and L. Montulli. *HTTP State Management Mechanism*, 2000.

[27] Markus Triska. *Generalising Constraint Solving over Finite Domains.* In Maria Garcia de la Banda and Enrico Pontelli, editors, *Proceedings of the 24th International Conference on Logic Programming*, volume 5366 of Lecture Notes in Computer Science, pages 820–821. Springer-Verlag Berlin Heidelberg New York, December 2008.

[28] Markus Triska. *Constraint Logic Programming over Finite Domains* [online]. Available from: `http://www.swi-prolog.org/man/clpfd.html`.

[29] M. Igler, S Jablonski, and C. Günther. *Semantic Process Modeling and Planning. The Fourth International Conference on Advances in Semantic Processing, October 25 - 30, 2010*, Florence, Italy.

[30] Ramzan Talib, Bernhard Volz, and Stefan Jablonski. *Agent Assignment for Process Management: Goal Modeling for Continuous Resource Management.* In Michael zur Muehlen and Jianwen Su, editors, *Business Process Management Workshops*, volume 66 of Lecture Notes in Business Information Processing, pages 25–36. Springer, 2010.

[31] Ivan Bratko. *Prolog Programming for Artificial Intelligence.* Addison Wesley, 3rd edition, September 2001.

[32] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A modern approach.* Prentice Hall, Upper Saddle River, N.J., 2nd international edition, 2003.

[33] Stefan Jablonski, Michael Igler, and Chistoph Günther. *Supporting collaborative work through flexible Process Execution.* In *CollaborateCom* [35], pages 1–10.

[34] M. Igler, P. Moura, M. Zeising, M. Faerber, and S. Jablonski. *Modeling and Planning Collaboration using Organizational Constraints.* In *The 6th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2010), Chicago, Illinois.*

[35] *The 6th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2010), Chicago, Illinois.* IEEE, October 2010.

[36] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*, volume 53 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 2001.

[37] A. Nonnengart und H. J. Ohlbach. *Modal- und Temporallogik.* In K. H. Bläsius and H.-J. Bürckert, editors, *Deduktionssysteme: Automatisierung des logischen Denkens*, pages 239–284. Oldenbourg, München, 2. edition, 1992.

[38] Saul Kripke. *A Completeness Theorem in Modal Logic.* In *The Journal of Symbolic Logic*, Volume 24(1):1–14, 1959.

[39] Saul A. Kripke. *Semantical Analysis of Modal Logic I. Normal Propositional Calculi.* In *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[40] Paul Weingartner. *Modal Logics with Two Kinds of Necessity and Possibility.* In *Notre Dame Journal of Formal Logic*, 9(2):97–159, 1968.

[41] Linh Anh Nguyen. *Foundations of Modal Logic Programming: The Direct Approach.* Manuscript (release 2.2), available at `http://www.mimuw.edu.pl/~nguyen/publications.html#mlp`, November 2006 (last revised May 2010).

[42] Paulo Moura. *Logtalk web site.* `http://logtalk.org/`.

[43] Daniel Bardou. *Inheritance Hierarchy Automatic (Re)organization and Prototype-Based Languages.* In *Objects and classification: a natural convergence Workshop, 14th European Conference on Object-Oriented Programming (ECOOP 2000), Cannes, France*, June 2000.

[44] Vangelis Vassiliadis, Jan Wielemaker, and Chris Mungall. *Processing OWL2 Ontologies using Thea: An Application of Logic Programming.* In Rinke Hoekstra and Peter F. Patel-Schneider, editors, *OWLED*, volume 529 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009.

[45] Jan Wielemaker. *SWI-Prolog web site.* `http://www.swi-prolog.org/`.

[46] Vítor Santos Costa. *YAP Prolog.* `http://www.dcc.fc.up.pt/~vsc/Yap/documentation.html`.

[47] S. White. *Business Process Modeling Notation (BPMN) — Version 1.0* [online]. May 2004. Available from: `http://www.bpmn.org/Documents/BPMN_V1-0_May_3_2004.pdf`.

[48] OMG. *Business Process Modeling Notation (BPMN) — Version 2.0* [online]. January 2011. Available from: `http://www.omg.org/spec/BPMN/2.0/PDF/`.

[49] Yi Gao. *BPMN - BPEL Transformation and Round Trip Engineering*, 2006. Available from: `http://www.eclarus.com/resources/BPMN_BPEL_Mapping.pdf`.

[50] W. M. P. van der Aalst and Ter. *YAWL: yet another workflow language. Information Systems*, 30(4):245–275, June 2005.

[51] Nick Russell and Arthur H. M. Ter Hofstede. *newYAWL: Specifying a workflow reference language using Coloured Petri Nets.* In *Department of Computer Science, University of Aarhus*, pages 107–126, 2007.

[52] Kurt Jensen. *Coloured Petri Nets: basic concepts, analysis methods and practical use*, volume 1. Springer-Verlag, London, UK, 2. edition, 1996.

[53] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. *Workflow Patterns. Distrib. Parallel Databases*, 14(1):5–51, 2003.

[54] Maja Pesic, Helen Schonenberg, and Wil M. P. van der Aalst. *DECLARE: Full Support for Loosely-Structured Processes.* In *EDOC'07: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, page 287, Washington, DC, USA, 2007. IEEE Computer Society.

[55] M. Pesic. *DECLARE website*. `http://www.win.tue.nl/declare`.

[56] Free Software Foundation. *GNU GENERAL PUBLIC LICENSE – Version 2*. `http://www.gnu.org/licenses/gpl-2.0.html`, June 1991.

[57] Wil M. P. van der Aalst and Maja Pesic. *DecSerFlow: Towards a Truly Declarative Service Flow Language*. In Frank Leymann, Wolfgang Reisig, Satish R. Thatte, and Wil M. P. van der Aalst, editors, *The Role of Business Processes in Service Oriented Architectures, 16.07. - 21.07.2006*, volume 06291 of Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[58] M. Pesic and W. van der Aalst. *A Declarative Approach for Flexible Business Processes Management*. In *In Business Process Management Workshops*, pages 169–180. Springer-Verlag Berlin/Heidelberg/New York, 2006.

[59] Ingo Melzer. *Service-orientierte Architekturen mit Web Services: Konzepte - Standards - Praxis (German Edition)*. Spektrum Akademischer Verlag, 3. edition, October 2008.

[60] D. Gabbay, A. Kurucz, F. Wolter, and M. Zakharyaschev. *Many-Dimensional Modal Logics: Theory and Applications*, volume 148. Elsevier Science Publishers Ltd., 1. edition, Jan 2003.

[61] Stijn Goedertier, Raf Haesen, and Jan Vanthienen. *EM-BrA$^2$CE v0.1: A Vocabulary and Execution Model for Declarative Business Process Modeling*. FETEW Research Report KBI-0728, K.U.Leuven, 2007.

[62] OMG. *Semantics of Business Vocabulary and Business Rules(SBVR), v1.0*. Technical report, OMG, January 2008. Available from: `http://www.omg.org/spec/SBVR/1.0/PDF`.

[63] AT&T Research. *Graphviz – Graph Visualization Software*. `http://www.graphviz.org`.