Ansätze zur flexiblen Navigation in Prozessabläufen

Diplomarbeit

im Fachgebiet Mathematik

vorgelegt von:	Christoph Günther
Studienbereich:	Mathematik
Matrikelnummer:	1018651
Erstgutachter:	Prof. Dr. Reinhard Laue
Zweitgutachter:	Prof. Dr. Stefan Jablonski

Abstract

Prozessnavigationssysteme oder Workflow Management Systeme (WFMS) sind ein wesentlicher Bestandteil vieler Betriebe und Organisationen. Aktuell zeigen sich einige Probleme wie Inakzeptanz durch die Benutzer und Effizienzverlust in Betriebsprozessen, da derzeit gängige System starr und unflexibel sind.
In dieser Arbeit soll ein wesentlich flexiblerer Ansatz betrachtet werden. Es wird nur noch ein Rahmen vorgegeben, der den Erfolg des Workflow sicher stellt. Die Entscheidung der Prozesschrittabfolge wird hierbei nur beratend unterstützt, dem Benutzer jedoch die Entscheidung überlassen. Das System sucht also alle möglichen nächsten Prozesschritte und schlägt sie dem Benutzer vor. Durch die große Anzahl der Kombinationsmöglichkeiten sind innovative Ansätze gefordert um ein performantes System zu bieten. In dieser Arbeit werden Probleme aufgezeigt, diskutiert und gelöst, um am Ende einen sehr performanten und hochflexiblen Ansatz zur Prozessnavigation vorzustellen.

Inhaltsverzeichnis

1. Einleitung

Prozessnavigationssysteme oder auch Workflow-Management-Systeme (WFMS) haben in der Wirtschaft mittlerweile eine große Bedeutung erlangt. Sie kontrollieren Unternehmensabläufe, stellen Kriterien für die Qualität eines Prozessablaufs bereit und garantieren gleichbleibende Verarbeitungsstandards.

Ein WFMS ist hierbei nach WFMC [1999] wie folgt definiert:

Definition 1 *Prozessnavigationssystem (WFMS)*
Ein Prozessnavigationssystem (WFMS) ist ein System, das den Ablauf eines Workflows definiert, erzeugt und verwaltet. Es verwendet hierzu Software, die auf einer oder mehreren Workflow-Engines läuft und in der Lage ist Prozessdefinitionen zu interpretieren, mit Workflow Teilnehmern zu interagieren und - so benötigt - die nötigen IT Ressourcen bereitstellt.

Firmen und Organisationen, die sich für ein WFMS entschieden haben verfolgen verschiedene Ziele. Einige sollen hier genannt werden (Wikipedia [2009],BUSSLER und JABLONSKI [1996], WFMC [1999]):

Qualität Die Qualität der Prozesse soll gesteigert werden.
Dies zielt auf die vom WFMS bereitgestellten Kriterien zur Güte eines Prozessablaufs ab. Eine Organisation möchte Prozesse, die unter dem desidierten Standard liegen, erkennen und verbessern.

Standardisierung Die Prozesse sollen vereinheitlicht werden.
Verschiedene Prozesse sollen einem einheitlichen Muster unterworfen werden, um zum Beispiel größere Redundanz in Ressourcen oder Mitarbeitern zu erreichen.

Effizienz Die Bearbeitungszeiten und Kosten sollen reduziert werden.
Dies benötigt wiederum Kriterien und geeignete Messinstrumente.

Transparenz Die Informationsverfügbarkeit soll erhöht werden.
Durch die Dokumentationsfunktionalität des WFMS sind Informationen schnell zugänglich und der Status eines Ablaufs lässt sich schnell bestimmen.

Flexibilität Die Flexibilität der Prozesse soll erhöht werden.
Ein WFMS soll die Abarbeitung eines Prozesses nach Möglichkeit nicht strikt vorgeben, sondern einen situationsbedingt günstigen Ablauf des Prozesses wählen. Hierzu muss ein WFMS entsprechend flexibel konzipiert sein, um eine dynamische Entscheidung oder Strategieänderung zu akzeptieren.

Vor allem der letzte Punkt wird in aktuellen WFMS noch nicht ausreichend berücksichtigt und soll den Schwerpunkt dieser Arbeit bilden. Einige häufig genannte Kritikpunkte bei aktuellen WFMS stellen den Ausgangspunkt der Untersuchungen dar.

1.1. Motivation

Bei aktuellen WFMS gibt es einige begründete Kritikpunkte. Alle zu nennen ist wohl nicht möglich, jedoch bietet Wikipedia [2009] eine Auflistung der am häufigsten genannten:

- Auf seltene oder nicht vorgesehene Ereignisse kann gar nicht oder nur verzögert reagiert werden
- Mitarbeiter verlieren ihre Eigenverantwortung, da sie sich nur an den Workflow halten. Abweichungen vom vorgeschriebenen Handlungsstrang werden sanktioniert.
- Insbesondere die Leistungsträger unter den Mitarbeitern fühlen sich oft gegängelt.
- Häufig resultiert eine geringere Motivation (Dienst nach Vorschrift).
- Bestehende inoffizielle Teams werden (gewollt) gestört und brechen auseinander.
- Kreativität und Ideen zur Verbesserung der Geschäftsprozesse werden durch den gegebenen Rahmen eher gebremst.

Diese Sammlung von Meinungen zur Einführung von WFMS verdeutlicht deren Problematik. Das WFMS wird **über** den Menschen gestellt. Dies schränkt Mitarbeiter ein und resultiert in verschieden gearteten Nachteilen wie Demotivation, Inakzeptanz und insgesamt Effizienzverlust. Es wäre also sinnvoll, die gegebene Funktionalität von WFMS

beizubehalten, aber eine wesentlich größere Flexibilität bei der Ausführung der Prozesse zuzulassen, sowie dem Benutzer eine Entscheidungsgewalt einzuräumen und dem System eine beratende Funktion zu geben. Dies führt uns zum Ziel dieser Arbeit.

1.2. Ziel der Arbeit

Es soll ein WFMS entstehen, das nicht strikt einen Prozessablauf vorgibt, sondern verschiedene Optionen anbietet. Hierbei soll das System Entscheidungshilfen anbieten, mögliche Konsequenzen im Vorfeld zeigen und die Wahrung der Prozessintention gewährleisten. Die Prozessintention meint hiermit den erfolgreichen Abschluss des durch den Workflow modellierten Arbeitsschrittes.

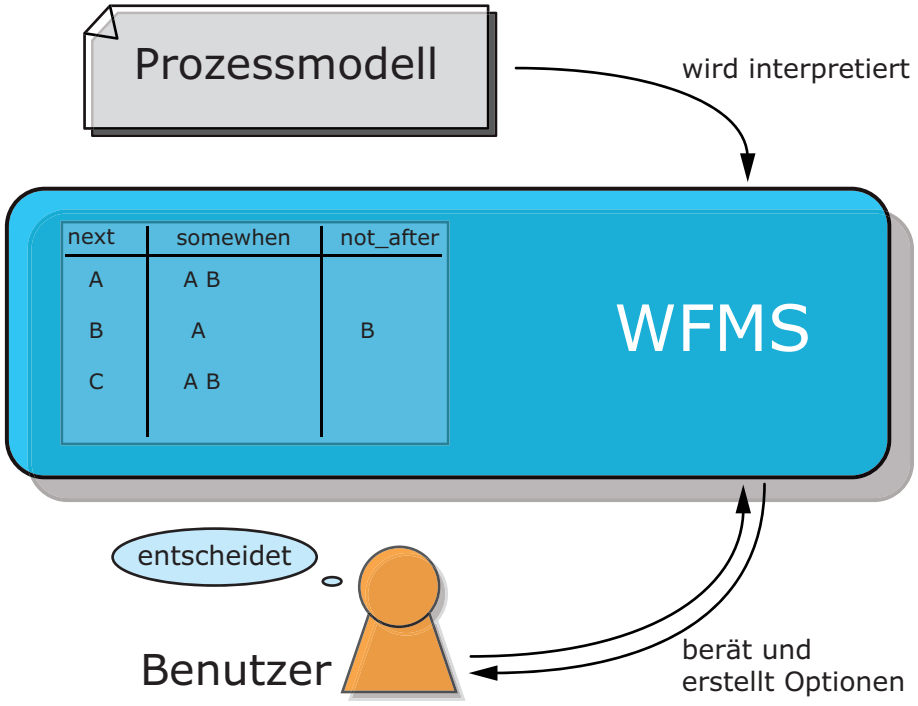

Abbildung 1.1.: Die (gewünschte) Funktionsweise eines Prozessnavigationssystems

Es ist also ein System gefordert, das die Möglichkeiten den Prozess auszuführen vollständig überblickt, dem Benutzer sinnvolle Entscheidungsoptionen präsentiert und Zusatzin-

formationen als Entscheidungshilfe evaluiert und bereitstellt.
In dieser Arbeit soll besonders die effiziente Implementierung des Hintergrundsystems behandelt werden. Es wird eine Möglichkeit gesucht alle Lösungen zu einem Prozessmodell zu generieren damit dem Benutzer maximale Entscheidungsfreiheit geboten werden kann. Die hierzu verwendete Syntax ist neu, wurde von IGLER erstmals vorgestellt und wird in Abschnitt 2.2 erläutert. Hierauf basierend soll eine Möglichkeit gefunden werden, effizient, größtmögliche Flexibilität bei der Navigation durch einen Prozess zu bieten. Im Laufe der Arbeit wird sich zeigen, dass das Hintergrundsystem unabhängig von der verwendeten Syntax ist. Dies bietet ein großes Potenzial bei der Anwendung in verschiedenen Bereichen, da eine auf die jeweilige Domäne spezifizierte Syntax verwendet werden kann. Die Möglichkeiten zur Implementierung und Gestaltung einer Benutzerschnittstelle, sowie das Finden von Kriterien um spezifische Lösungen zu beurteilen, sind durch den modularen Ansatz unabhängig von dem hier geforderten Hintergrundsystem und sollen deshalb nicht Teil dieser Arbeit sein.

1.3. Aufbau der Arbeit

Dieses Kapitel soll eine Kurzeinführung in die Anwendungsbereiche von Prozessnavigationssystemen geben. Neben einer kurzen Auflistung der Gründe für die Entscheidung für ein WFMS sollen auch die Problematiken derselben kurz erläutert und hiermit die vorliegende Arbeit motiviert werden.
Eine Einführung in die verwendete Syntax zur Prozessmodellierung erfolgt in Kapitel 2. Hier werden die Charakteristiken eines Modells erläutert und Modellierungselemente definiert und erklärt. Zuletzt wird eine Abschätzung der Größenordnung des gegebenen Problems angegeben, die die Schwierigkeit des Findens aller Lösungen verdeutlicht.
Kapitel 3 diskutiert zwei Lösungsansätze. Zuerst wird ein grammatikbasierter Ansatz, der aus dort genannten Gründen nicht erfolgversprechend scheint, behandelt. Danach wird das gewählte Vorgehen erklärt. Die Effizienz des Systems wird hierbei auf drei Konzepten begründet: Kapselung, Generierung eines Automaten und Interaktion. Diese werden in den zugehörigen Abschnitten ausführlich erläutert.
Mit der Implementierung des Systems befasst sich Kapitel 4. Hier wird die Architektur

des Grundsystems erklärt. Der hier generierte endliche Automat wird in drei Unterabschnitten behandelt, nämlich der Inititialisierung, der Laufzeitphase und den informativen Methoden. Anschließend werden die zugrunde liegenden Objekte behandelt, die für eine effiziente Speicherung der gewonnenen Information benötigt werden. Zuletzt behandelt das Kapitel einige effizienzsteigernde Maßnahmen des Systems.
In Kapitel 5 schließlich wird die Lösung der vorangegangenen Kapitel evaluiert und ein Ausblick auf weitere Forschungsfelder gegeben.

2. Problemstellung

In folgendem Kapitel soll erklärt werden, wie die flexible Navigation durch einen Prozess funktioniert. Hierzu werden Begriffe wie **Prozessschritt** und **Verknüpfung** definiert und deren Bedeutung erläutert. Zuletzt wird eine Abschätzung der Größenordnung des zu behandelnden Problems der vollständigen Angabe aller möglichen Prozesswege angegeben.

2.1. Flexible Prozessnavigation

Das Ziel der flexiblen Prozessnavigation ist es, dem Benutzer maximale Entscheidungsfreiheit in der Reihenfolge der Schritte eines Prozesses zu bieten. Es werden also lediglich Rahmenbedingungen vom Prozessmodell vorgegeben die gewährleisten, dass die Ausführung auch der Prozessintention entspricht. Die Prozessintention ist der Zweck des Prozesses, also zum Beispiel ein erfolgreicher medizinischer Eingriff, die Fertigstellung eines Produkts oder die Problemlösung bei einem technischen Supportgespräch.
Der jeweilige Benutzer wird nach IGLER durch den Prozess geführt, indem ihm zu jeder Zeit angezeigt wird, welchen nächsten Prozessschritt er ausführen kann, welche Prozessschritte danach noch abgearbeitet werden müssen und welche Prozessschritte er nach Ausführung eines Prozessschrittes nicht mehr ausführen kann (Tabelle 2.1).

next	**somewhen**	**not after**
A	B C D E	A
B	C D E	A B
C	D E	A B C
D	B C D E	A
DONE:	**A**	

Tabelle 2.1.: Navigator zu Modell 2.6 nach einmaliger Ausführung von A

Diese Informationen sollen dem Benutzer mit möglichst kurzer Latenz angezeigt werden können. Es muss also eine Lösung gefunden werden, diese Daten entweder im Vorfeld zu berechnen und zu speichern oder sie zur Laufzeit zu berechnen. Zur Idee, die Möglichkeiten zu berechnen und zu speichern, sei hier auf Abschnitt 2.3 verwiesen, der eine Abschätzung der Größenordnung bietet.

Für die Hintergründe zu Prozessnavigationssystemen sei hier auf BUSSLER und JABLONSKI [1996] verwiesen. Hier werden neben weiteren interessanten Aspekten zur Prozessmodellierung und allgemeinen Prozessnavigationssystemen, die in dieser Arbeit nicht betrachtet werden können, in Abschnitt 5.2.1 folgende Anforderungen an Prozessnavigationssysteme angegeben:

Erweiterbarkeit
: Um neuen Anforderungen an Prozessmodelle gerecht zu werden, muss ein Prozessnavigationssystem erweiterbar sein.

Anpassbarkeit
: Ein Prozessnavigationssystem muss anpassbar sein, um auf individuelle Bedürfnisse des Anwendungsbereichs optimal konfiguriert werden zu können.

Dynamisch modifizierbar
: Um auf Änderungen im produktiven Betrieb reagieren zu können, müssen dynamische Modifikationen möglich sein.

Vollständigkeit
: Ein Prozessnavigationssystem muss die zugehörigen Prozessmodelle vollständig implementieren. Dies meint die vollständige und adäquate Repräsentation der im Modell beschriebenen Prozessabläufe.

Wiederverwendbarkeit
: Um die Kosten für die Entwicklung von Prozessnavigationssystemen gering zu halten, ist die Wiederverwertbarkeit wichtig. Ein System sollte auf viele Anwendungsbereiche angepasst werden können.

Integrierbarkeit
: Ein Prozessnavigationssystem muss in bestehende Infrastruktur integriert werden können und eine Möglichkeit zur Integration bestehender Funktionalität bieten.

Oben genannte Punkte werden in Abschnitt 5.1 näher betrachtet und das System des Autors anhand dieser Qualitätskriterien evaluiert. Des Weiteren sollen natürlich die in Abschnitt 1.1 genannten Kritikpunkte im eigenen System berücksichtigt werden. Deren Überprüfung wird ebenfalls in Abschnitt 5.1 erfolgen.

2.2. Nebenbedingungen

Basierend auf den Definitionen der Arbeit von IGLER, allgemeinen Definitionen und Ausführungen zu Prozessmodellen [BUSSLER und JABLONSKI 1996, Kapitel 5- 8] und Abschnitt 2.1 werden nun die Bausteine des dieser Arbeit zugrunde liegenden Vorgehens erläutert.
Die Abarbeitungsreihenfolge von Prozessschritten ist nicht beliebig. Durch das Prozessmodell werden verschiedene mögliche Abarbeitungsreihenfolgen determiniert. Hierzu werden zunächst Modellierungswerkzeuge und die Merkmale eines Prozessschrittes betrachtet.

Definition 2 *Prozessschritt*
Ein Prozessschritt $P \in \mathcal{P}$ ist ein eindeutig bezeichnetes, einen beliebigen Vorgang repräsentierendes Element eines Prozesses.
Er lässt sich als 4-Tupel darstellen:

$$
\begin{aligned}
P &:= (b, c, min, max)\,, \text{ mit} \\
b &\quad \text{einem eindeutigen Bezeichner} \\
c &\quad \text{einer Beschreibung des Prozessschrittes} \\
min \in \mathbb{N}_0 \cup \mathcal{P} &\quad \text{der minimal zulässigen Ausführungshäufigkeit,} \\
max \in \mathbb{N} \cup \{*\} \cup \mathcal{P} &\quad \text{der maximal zulässigen Ausführungshäufigkeit,}
\end{aligned}
$$

Die einmalige Ausführung eines Prozessschrittes heißt Instanz. Ist $max > 1$ müssen alle Instanzen eines Prozessschrittes immer direkt hintereinander ausgeführt werden.

Die Ausführungshäufigkeit eines Prozessschrittes nennt man auch **Kardinalität**.

Definition 3 *Domäne*

$$d(P) = \{min, \ldots, max\} \text{ heißt } \textbf{Domäne} \text{ von } P. \tag{2.1}$$

Man sagt „der Prozessschritt liegt in seiner Domäne", wenn z die Ausführungshäufigkeit des Prozessschrittes $P \in \mathcal{P}$ und $min \leq z \leq max$.
*Ist min oder $max \in \mathcal{P}$ so heißt die Domäne **dynamisch**. Ist $max = *$ bedeutet dies beliebige maximale Ausführungshäufigkeit.*

Ein Prozessmodell ergibt sich durch Verknüpfung mehrerer Prozessschritte. Dies modelliert gleichzeitig im Ablauf zu berücksichtigende Abhängigkeiten, wie in der jeweiligen Anwendung vorgeschriebene Ausführungsreihenfolgen (z. B. Entwurf → Produktion → Prüfung), als auch Datenabhängigkeiten nicht direkt voneinander abhängiger Prozessschritte. Ein Prozessmodell wird, nach aktuellem Stand, nicht zur Laufzeit modifiziert, so dass sich Verknüpfungen und Abhängigkeiten nicht nach der Inititalisierungsphase (vgl. Abschnitt 4.1.1) verändern. Ist dies der Fall so ist ein Neustart der Prozessausführung nötig. Um verschieden geartete Abhängigkeiten modellieren zu können, werden zwei unterschiedliche Verknüpfungen definiert.

Definition 4 *Verbindung*
Eine Verbindung ist die Verknüpfung zweier Prozessschritte. Es gibt zwei verschiedene Typen von Verbindungen:

solide Verbindung

*Zwei Prozessschritte $A, B \in \mathcal{P}$ heißen genau dann **solide verbunden**, wenn A in einer bezüglich $d(A)$ zulässigen Kardinalität vor B ausgeführt wird und keine Instanz von A nach einer Instanz von B ausgeführt wird.*
Schreibweise: `solid_arrow(A,B)`

Abbildung 2.1.: Solide Verbindung

flexible Verbindung

Zwei Prozessschritte $A, B \in \mathcal{P}$ heißen genau dann ***flexibel verbunden****, wenn entweder gilt $solidarrow(A, B)$ oder $solidarrow(B, A)$.*
Schreibweise: `dashed_arrow(A,B)`

Abbildung 2.2.: Flexible Verbindung

Zwei durch eine gemeinsame Verbindung verknüpfte Prozessschritte heißen verbunden.

Eine flexible Verbindung gibt also nicht explizit eine Reihenfolge vor, verbietet jedoch die Instanzen der Prozessschritte zu vermischen. Oft werden aus Gründen der Übersichtlichtlichkeit flexible Verbindungen nicht explizit ausgezeichnet. So repräsentiert zum Beispiel:

Listing 2.1: flexible Verbindungen

```
1 proc(a,'A',1,1) .
2 proc(b,'B',1,1) .
3 proc(c,'C',1,1) .
4
5 dashed_arrow(a,b).
6 dashed_arrow(b,c).
```

Die resultierenden Prozessschrittfolgen sind, wie aus Abbildung 2.3 ersichtlich und in Tabelle 2.2 aufgelistet, alle möglichen Permutationen der drei Prozessschritte. Hierbei spielt die Groß- bzw. Kleinschreibung der Bezeichner keine Rolle. Die unterschiedlichen Schreibungen sind nötig, weil Prolog großgeschriebene Ausdrücke immer als Variable interpretiert.

ABC	ACB	BAC	BCA	CBA	CAB

Tabelle 2.2.: Zustände zum Modell 2.1

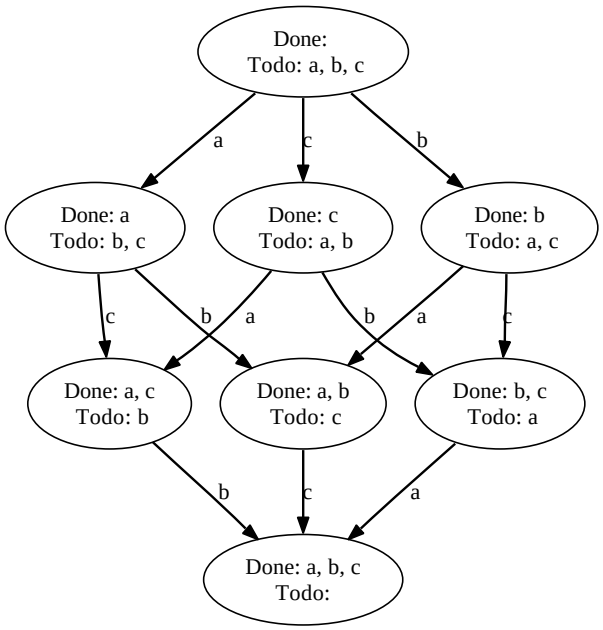

Abbildung 2.3.: Der Zustandsgraph zu Listing 2.1

Bemerkung 1 *Schreibweisen und abhängige Kardinalität*
Die Domäne $d(B)$ eines Prozessschrittes wird wie in Definition 3 festgelegt. Gibt es ein mit Prozessschritt $A \in \mathcal{P}$ solide verbundenes Element $B \in \mathcal{P}$, und ist die Domäne $d(B)$ dynamisch, also min oder $max \in \mathcal{P}$ so gilt für
$proc(B, A, max)$ *B muss mindestens so oft ausgeführt werden wie A und für*
$proc(B, min, A)$ *B darf höchstens so oft ausgeführt werden wie A.*

Da sich die Domänengrenzen auf den Zählerwert des referenzierten Objektes beziehen, muss dieses vor der Ausführung des Prozessschrittes mit dynamischer Domäne in gültiger Kardinalität ausgeführt sein. Eine weitere Ausführung des referenzierten Prozessschrittes nach Ausführung des referenzierenden Prozessschrittes ist aufgrund der soliden Verknüpfung nicht mehr möglich. So ist gewährleistet, dass sich dynamische Domänengrenzen nicht nachträglich ändern und somit die Kardinalität des referenzierenden Prozessschrittes gültig bleibt.

Definition 5 *Prozess*
*Ein **Prozess** ist eine Folge verbundener Prozessschritte. Durch die unterschiedlichen Verbindungen sind verschiedene Abfolgen von Prozessschritten möglich, diese heißen Prozessabfolgen. Eine vollständige Prozessabfolge heißt **Weg**.*

Demnach sind alle Prozessschritte eines Weges in gültiger Kardinalität ausgeführt und bezüglich der modellierten Verknüpfungen valide.

Ein typisches Beispiel könnte wie folgt aussehen:

Listing 2.2: Anwendungsbeispiel

```
1  proc(as, 'A schält Kartoffeln', 1, 4).
2  proc(bs, 'B schält Kartoffeln', 1, 4).
3  proc(cs, 'C schält Kartoffeln', 1, 4).
4
5  proc(ak, 'A kocht Kartoffeln', 1, as).
6  proc(bk, 'B kocht Kartoffeln', 1, bs).
7  proc(ck, 'C kocht Kartoffeln', 1, cs).
8
9  solid_arrow(as, ak).
10 solid_arrow(bs, bk).
11 solid_arrow(cs, ck).
```

Hier wird modelliert, dass drei Personen `A`, `B` und `C` nacheinander Kartoffeln schälen und kochen wollen. Es gibt nur einen Schäler und einen Topf, deshalb kann immer nur einer der drei schälen und einer der drei kochen. Jeder der drei möchte einen Teil der Kartoffeln kochen, die er selbst geschält hat. Deshalb muss jeder erst schälen, bevor er kochen kann. Die flexiblen Verknüpfungen wurden aus Gründen der Übersichtlichkeit in Programmlisting 2.2 weg gelassen. Die möglichen Prozessabfolgen sind in Abbildung 2.4 aufgezeigt.

Definition 6 *Box*
*Die Kapselung einer Prozessabfolge heißt **Box**. Die Box wird von außen atomar behandelt, d.h. es wird nicht zwischen Box und Prozessschritt unterschieden. Dies entspricht der üblichen Definition einer Klammer. Wird ein Prozessschritt P aus der Box B ausgeführt, so müssen alle Elemente $E \in B$ in zulässiger Kardinalität bezüglich ihrer Domäne*

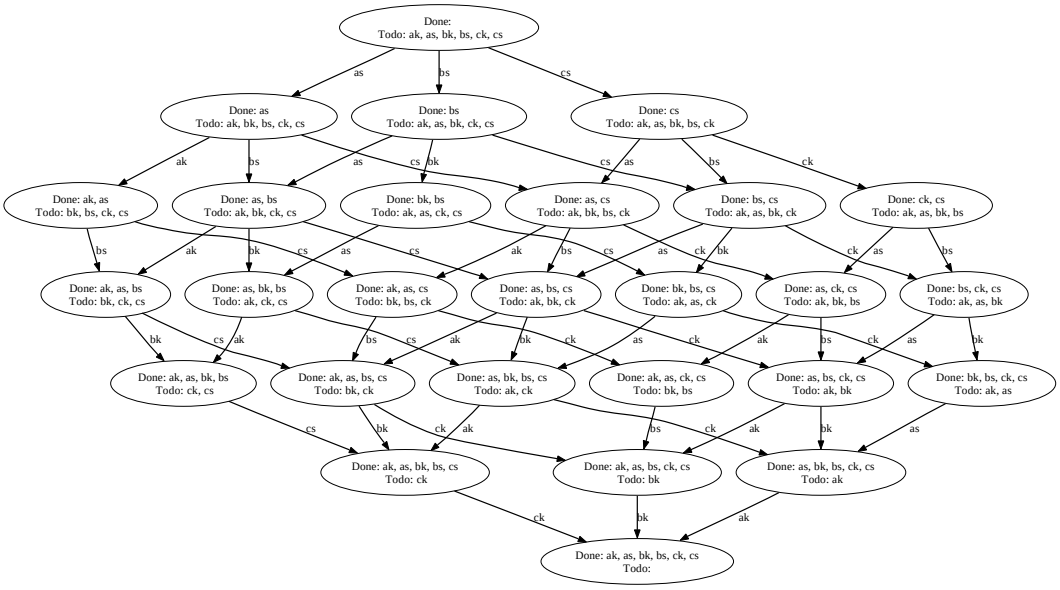

Abbildung 2.4.: Der Zustandsraum zu Beispiel 2.2

ausgeführt worden sein, bevor wieder ein Prozessschritt außerhalb der Box ausgeführt werden kann.

Hierzu könnte man die Anwendung modellieren, dass drei Kartoffelfelder abgegangen werden und, nachdem man zurückgekehrt ist, das Schälen der jeweiligen Ausbeute beginnt. Dabei sei die Reihenfolge des Sammelns sowie des Schälens unerheblich.

Listing 2.3: Eine Box Anwendung

```
1  proc(as, 'Sammeln auf Feld A', 0, *).
2  proc(bs, 'Sammeln auf Feld B', 0, *).
3  proc(cs, 'Sammeln auf Feld C', 0, *).
4
5  proc(kas, 'Kartoffeln von A schälen', 0, as).
6  proc(kbs, 'Kartoffeln von B schälen', 0, bs).
7  proc(kcs, 'Kartoffeln von C schälen', 0, cs).
8
9  box(sammeln, [as, bs, cs]).
10 box(schälen, [kas, kbs, kcs]).
11
12 solid_arrow(sammeln, schälen).
```

Abbildung 2.5 zeigt das Verhalten von Modell 2.3. Die Prozessschritte `as`, `bs`, `cs` werden in beliebiger Reihenfolge ausgeführt, bis der gemeinsame Zustand `[done: [as, bs, cs], kas, kbs. kcs]` erreicht ist. Dann werden die Zustände `kas`, `kbs`, `kcs` in beliebiger Reihenfolge ausgeführt.

Die resultierenden Möglichkeiten das Modell zu durchlaufen sind dann:

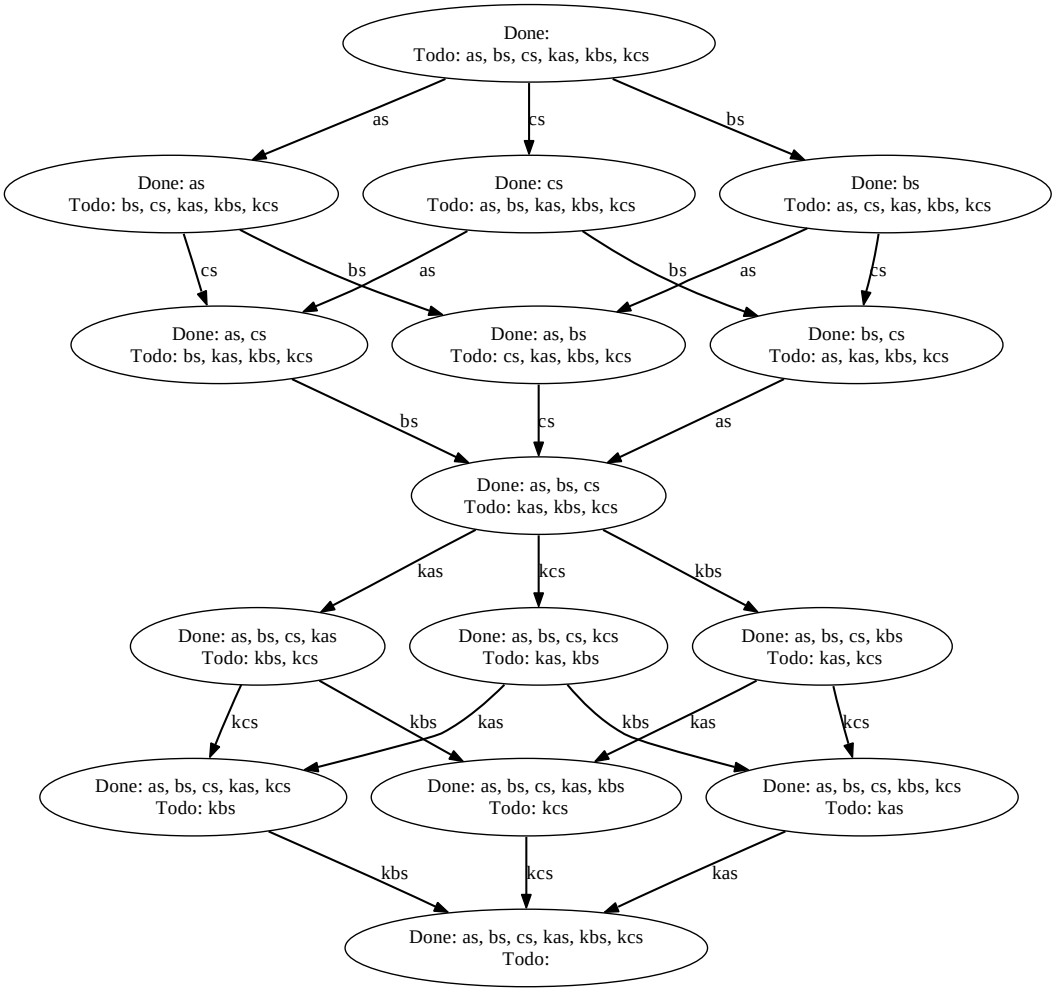

Abbildung 2.5.: Der Zustandsraum zu Beispiel 2.3

2.3. Zustandsraum

Die Ausführungen und Beispiele aus Abschnitt 2.2 lassen vermuten, dass es für viele Modelle sehr viele Wege durch den Prozess gibt. Satz 1 liefert eine Möglichkeit zur Berechnung der möglichen Ausführungsreihenfolgen.

Satz 1 *Berechnung der möglichen Prozessabfolgen*
Seien $p_i \in \mathcal{P}, i \in \{1, \ldots, N\}$ *die Prozessschritte,* $d(p_i) \subset \mathbb{N} \cup \{\infty\}$ *deren Domäne,* $s_j \subset \mathcal{P}, j \in \{1, \ldots, m\}, s_j \cap s_k = \emptyset, j \neq k$ *die paarweise disjunkten, soliden Folgen und* $|\mathcal{A}| \in \mathbb{N}$*, die Anzahl der Elemente in der Menge* $\mathcal{A}$*. Weiter sei:*

$$K_0 = 0 \qquad\qquad K_j = K_{j-1} + |s_i|, \quad \forall j \in \{1, \ldots, m\}$$

Dann ist die Größe des Zustandsraumes M*:*

$$M = \prod_{j=1}^{m} \binom{N - K_{j-1}}{|s_j|} \cdot (N - K_m)! \cdot \prod_{n=1}^{N} |d(p_n)| \tag{2.2}$$

Beweis Man kann sich eine mögliche Prozessabfolge als Objekte (Prozessschritte oder Boxen), die auf N Plätze verteilt werden, vorstellen. Da sich innerhalb einer soliden Liste die Reihenfolge nicht verändert, können deren Elemente als nicht unterscheidbar angenommen werden. Man fängt also mit der ersten soliden Liste an und verteilt deren $|s_1|$ Objekte auf die N Plätze. Es bleiben $N - |s_1|$ Plätze. Dieses Vorgehen führt man sukzessive fort, bis alle Objekte, die in soliden Listen vorkommen, auf die N Plätze verteilt sind. Die restlichen Objekte sind nicht in soliden Listen, müssen also einzeln verteilt werden. Hierfür bleiben $N - K_m$ Plätze, also $(N - K - m)!$ Möglichkeiten. Schließlich sind innerhalb eines Objekts noch verschiedene Ausführungshäufigkeiten gekapselt. Diese entsprechen den Elementen in der Domäne, da nur diese in einem gültigen Weg durch den Prozess vorkommen, oder den Prozessschritten bzw. Objekte in einer Box.

♢

Satz 1 vernachlässigt die Verwendung von Boxen. Diese können durch Interpretation als

Prozessschritt/Objekt und anschließender Multiplikation der Möglichkeiten innerhalb der Box berücksichtigt werden, da der Inhalt der Box vollkommen unabhängig von deren Umfeld ist.

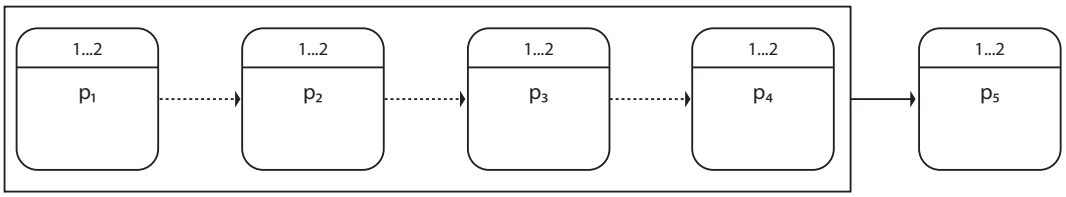

Abbildung 2.6.: Ein einfaches Prozessmodell

Es soll die Größe des Zustandsraumes an einem Beispiel verdeutlicht werden. Man nehme hierzu an, es sollten vier Prozessschritte (p_1, p_2, p_3, p_4) in beliebiger Reihenfolge ausgeführt werden, bevor ein fünfter (p_5) ausgeführt wird. Die zulässige Kardinalität jedes Prozessschrittes sei $d(p_i) = \{1, 2\}, \quad i \in \{1, \ldots, 5\}$. Das Modell nach Abschnitt 2.2 [IGLER] ist in Abbildung 2.6 dargestellt.

Tabelle 2.3 zeigt die möglichen Kombinationen für $d(p_i) = \{1\}$. Erweitert man die Domäne auf, wie in Modell 2.6 gefordert, $d(p_i) = \{1, 2\}$, so erweitert sich der Raum der Möglichkeiten auf das 32-fache von Tabelle 2.3, da jeder der Prozessschritte entweder ein- oder zweimal ausgeführt werden kann.

$p_1p_2p_3p_4p_5$	$p_2p_1p_3p_4p_5$	$p_3p_1p_2p_4p_5$	$p_4p_1p_2p_3p_5$
$p_1p_2p_4p_3p_5$	$p_2p_1p_4p_3p_5$	$p_3p_1p_4p_2p_5$	$p_4p_1p_2p_5p_3$
$p_1p_3p_2p_4p_5$	$p_2p_3p_1p_4p_5$	$p_3p_2p_1p_4p_5$	$p_4p_2p_1p_3p_5$
$p_1p_3p_4p_2p_5$	$p_2p_3p_4p_1p_5$	$p_3p_2p_4p_1p_5$	$p_4p_2p_3p_1p_5$
$p_1p_4p_2p_3p_5$	$p_2p_4p_1p_3p_5$	$p_3p_4p_1p_2p_5$	$p_4p_3p_1p_2p_5$
$p_1p_4p_3p_2p_5$	$p_2p_4p_3p_1p_5$	$p_3p_4p_2p_1p_5$	$p_4p_3p_2p_1p_5$

Tabelle 2.3.: Zustände für $d(p_i) = \{1\}$ zu Modell 2.6

Nach Satz 1 errechnet sich die Anzahl der möglichen Abarbeitungsreihenfolgen von Modell 2.6 mittels:

$N = 2$ nämlich die Box und p_5

$d(box) = \{1\}$, $d(p_5) = \{1, 2\}$ den Domänen

$s_1 = \{box, p_5\}$ da die Box und p_5 solide verbunden sind

$m = 1$ da es nur eine solide Folge gibt

Somit ist:

$$M_{Ges} = \prod_{j=1}^{1} \binom{2 - K_{j-1}}{|s_j|} \cdot (2 - K_1)! \cdot \prod_{n=1}^{2} |d(p_n)| \cdot M_{Box} \tag{2.3}$$

$$= \binom{2-0}{2} \cdot (2-2)! \cdot |d(box)| \cdot |d(p_5)| \cdot M_{Box} \tag{2.4}$$

$$= 1 \cdot 1 \cdot 1 \cdot 2 \cdot M_{Box} \tag{2.5}$$

In der Box ist

$N = 4$ nämlich $p_1, \ldots, p_4$

$d(p_i) = \{1, 2\}$ für alle Prozessschritte

$m = 0$ da es keine solide Folge innerhalb der Box gibt

und somit

$$M_{Box} = (4 - K_0)! \cdot \prod_{n=1}^{4} |d(p_n)| \tag{2.6}$$

$$= (4-0)! \cdot |d(p_1)| \cdot |d(p_2)| \cdot |d(p_3)| \cdot |d(p_4)| \tag{2.7}$$

$$= 24 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \tag{2.8}$$

$$= 384 \tag{2.9}$$

und damit ist

$$M_{Ges} = 2 \cdot 384 = 768 \tag{2.10}$$

Es ist offensichtlich, dass es nur für sehr kleine Modelle praktikabel ist, alle Wege durch den Prozess zu berechnen. Die hier berechneten 768 Wege durch den Graphen sind zwar weit davon entfernt die Grenzen der Berechenbarkeit auf aktuellen Rechnersystemen zu

erreichen, jedoch bestehen praxisrelevante Arbeitsprozesse auch nicht aus fünf Prozesschritten. Solche Systeme sind oft aus weit über 30 Schritten zusammengesetzt, meist mit recht großer Domäne. Wenn man den Worstcase betrachtet wären dies $30!$ Möglichkeiten, was auch mit aktuellen Rechnern kapazitiv nahezu unmöglich ist.
Deshalb werden im folgenden Kapitel Strategien zur Navigation durch den Prozess untersucht, die eine Vorabberechnung der Wege vermeiden und dabei trotzdem alle möglichen Ausführungsreihenfolgen berücksichtigen. Hierzu sind verschiedene Ansätze denkbar, einer erschien dem Autor jedoch besonders erfolgversprechend.

3. Lösungsansatz

Für praxisrelevante Probleme ist es kaum möglich, alle möglichen Wege vorab zu berechnen. Deshalb werden alternative Ansätze benötigt. Zunächst soll die erste Idee des Autors, eine grammatikbasierte Generierung aller Lösungen, erläutert werden. Da diese leider konzeptionell bedingt nicht zum Ziel führt, wird danach der letztendlich verwendete Algorithmus erklärt. Dieser verwendet einen endlichen Automaten der aus dem Prozessmodell generiert wird, um mit Hilfe einer geeigneten Zustandsübergangsrelation mögliche nächste Prozessschritte zu finden. Durch Abstraktion der Prozessschrittkardinalität und Interaktion mit externen Entscheidungsinstanzen wird hierbei der Aufwand für das Finden der Möglichkeiten wesentlich verringert.

3.1. Grammatiken

Der Grundgedanke des **grammatikbasierten Ansatzes** ist einfach. Das geforderte Verhalten wird mit Hilfe der deklarativen Programmierung in eine Grammatik konvertiert. Anschließend überprüft das System beliebige Lösungen auf Validität bezüglich der Grammatik. Natürlich können umgekehrt auch beliebige, bezüglich der Grammatik valide, Ausdrücke erzeugt werden.
Um die Abhängigkeit bestimmter Prozessschritte abzubilden, muss auf eine kontextsensitive Grammatik zurückgegriffen werden. Nur sie ermöglicht das Herstellen eines Zusammenhangs spezieller Bausteine, was bei den spezifischen Abhängigkeiten der Prozessmodellierung unabdingbar ist. Nachfolgend wird eine Möglichkeit zum Implementieren einer solchen Grammatik gezeigt. Hierfür wurde eine Bottom-Up Version gewählt, ebenso wäre ein Top-Down Ansatz denkbar. Das letzte Element muss nicht zwingend fest definiert sein, da man sich hier mit einem virtuellen End- bzw. Anfangselement helfen kann.

Listing 3.1: kontextsensitive Grammatik

```
1 % Ein Pfad besteht aus einem Element und einer Verbindung
2 path     --> fin(X),conn(X,Y),  {all_done}.
3
4 % Eine Verbindung kann leer sein wenn alle Prozessschritte abgearbeitet sind
5 conn(X,_)    --> []     ,{alldone(X)}.
6
7 % Eine Verbindung ist entweder solide oder flexibel und mit einer weiteren Verbindung verknüpft
8 conn(X,Y)    --> soli(X,Y), conn(Y,_) ,{do(Y)}.
9 conn(X,Y)    --> dash(X,Y), conn(Y,_) ,{do(Y)}.
```

Ein Pfad nach Listing 3.1 besteht also aus einem letzten Element `X` und einer Verbindung nach X. Sind alle Prozessschritte abgearbeitet, so bleibt die Verbindung leer: `[]`. Sind noch Prozessschritte Y durchzuführen so sind diese entweder solide (`soli(X,Y)`) oder flexibel (`dash(X,Y)`) mit X verbunden. Der Prozessschritt `Y` ist weiter verbunden, was durch den rekursiven Aufruf `conn(Y,_)`) realisiert wird.

In ausführlichen Tests mit einer Implementierung hat sich gezeigt, dass sich dieser Ansatz aus verschiedenen Gründen als nicht sinnvoll erweist:

Die Validität der erzeugten Wege bezüglich des Modells kann nicht direkt in die Grammatik implementiert werden und muss über Zusatzbedingungen realisiert werden. Dies ist grundsätzlich kein Problem, reduziert jedoch die Vorzüge einer kontextsensitiven Grammatik durch additiven Aufwand. Der Ansatz scheitert jedoch aus konzeptionellen Gründen schon für relativ kleine Probleme, da ein grammatikbasiertes System immer den vollständigen Satz durchlaufen muss um seine Richtigkeit bezüglich der Grammatik zu prüfen. Im ungünstigsten Fall gibt es $N!$ Möglichkeiten ein Problem von Komplexität N zu durchlaufen (vgl. Abschnitt 2.3), die ein grammatikbasierter Ansatz jeweils einzeln prüfen muss. Der grammatikbasierte Ansatz bietet also effektiv keinen Vorteil gegenüber der kanonischen Vorabberechnung aller Wege.

Im Folgenden wird ein wesentlich vielversprechenderer Ansatz betrachtet, der auf Reduktion der zu betrachtenden Möglichkeiten basiert.

3.2. ESProNa

Basierend auf der Arbeit von IGLER, der erstmals die Idee der Konvertierung in einen endlichen Automaten verfolgte, soll in folgendem Kapitel eine Variation und Verbesserung dessen behandelt werden. Die folgenden Vereinfachungsschritte haben die Leistungsfähigkeit des ursprünglichen Systems exorbitant gesteigert, was vor allem der effizienten Speicherung (vgl. Abschnitt 4.2), der Reduktion des Zustandsraumes (Abschnitt 3.2.1 und 3.2.3) und dem Schrumpfen zu durchsuchender Listen (vgl. Abschnitt 4.3) zu verdanken ist.

3.2.1. Kapselung

Der erste Vereinfachungsschritt des Problems ist die Kapselung der Kardinalität. Ziel ist es, den Zustandsraum um die Größenordnung der Domänen zu vereinfachen. Hierzu wird die einzelne Ausführung eines Prozessschrittes nicht mehr explizit im Zustand aufgeführt, sondern nur noch zwischen `todo`, `active` und `executed` unterschieden. Durch die in Abschnitt 4.2 beschriebenen Datenstrukturen muss hierbei sicher gestellt werden, dass die Übergänge zwischen den drei Prozessschrittzuständen nur durchgeführt werden können, wenn die jeweils erforderlichen Voraussetzungen gegeben sind.

`todo`
: ist der Grundzustand nach der Initialisierung des Prozessmodells. Er kann wieder erreicht werden, wenn ein als `active` gekennzeichneter Prozessschritt in seiner Ausführung abgebrochen wird.

`active`
: ist der Zustand, in dem Instanzen des Prozessschrittes ausgeführt werden können. Er kann nur aus dem Zustand `todo` erreicht werden.

`executed`
: ist der Endzustand eines Prozessschrittes. Er kann aus dem Zustand `active` erreicht werden, wenn der Prozessschritt in bezüglich seiner Domäne gültiger Kardinalität ausgeführt wurde.

Diese recht trivial wirkende Invention birgt erstaunliches Potenzial. Zunächst müssen keine Prüfungen der Domänen und Kardinalitäten von Vorgängerschritten durchgeführt

werden, da davon ausgegangen werden kann, dass ein als `executed` gekennzeichneter Prozessschritt in seiner Domäne liegt. Dazu vereinfacht sich der Zustandsraum um die Größenordnung der Domäne. So kann ein relativ langer Pfad erheblich vereinfacht werden, wie Listing 3.2 und die dazugehörige Abbildung 3.1 zeigen.

Listing 3.2: Prozessschritte mit großer Domäne

```
1 proc(a, '7 to 25', 7, 25).
2 proc(b, '1 to 60', 1, 60).
3
4 solid_arrow(a,b).
```

Es werden nicht alle ausgeführten Einzelinstanzen aufgelistet, sondern die Menge aller Instanzen gekapselt. Statt 19 verschiedene zulässige Kardinalitäten für a, bzw. 60 gültige Ausführungshäufigkeiten für b einzeln als Pfad aufzuführen, werden diese in einem Knoten zusammengefasst.

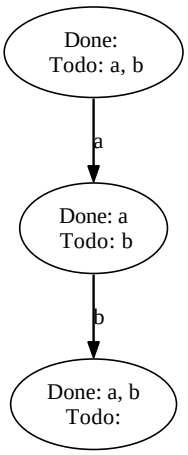

Abbildung 3.1.: Der Zustandsgraph zu Listing 3.2 ist trotz großer Domäne sehr klein.

Für eine spätere Überprüfung der Kardinalität eines Prozessschrittes muss demnach die Sicht geändert werden. Da für die Abhängigkeiten der Prozessschritte die exakte Kardinalität nicht wichtig ist, wird dieses Datum dort auch nicht bereitgestellt, sondern auf die Metainformation `todo`, `active`, `executed` abstrahiert. Ist für eine weitere Anwendung die Kardinalität interessant, so muss diese von der Datenstruktur eine andere Sichtweise zur Verfügung gestellt bekommen. In Abschnitt 4.2 wird ausführlicher erläutert, wie auf diese vorbereiteten Daten zugegriffen werden kann.

3.2.2. Endlicher Automat

Nachdem die Möglichkeiten den Prozess zu durchlaufen um Faktor $\prod_{p \in \mathcal{P}} |dp|$ verringert wurden, soll der Grundgedanke des Algorithmus betrachtet werden. Dieser geht auf IGLER zurück.
Aus dem Prozessmodell, also einer Menge von Prozessschritten $\mathcal{P} = \{p_1, \ldots, p_n\}$ und Verknüpfungen, wird ein endlicher Automat $\mathcal{A} = (\Phi, \Sigma, \delta, S, F)$, wie ihn NOLTEMEIER [1993] definiert, dynamisch erzeugt.

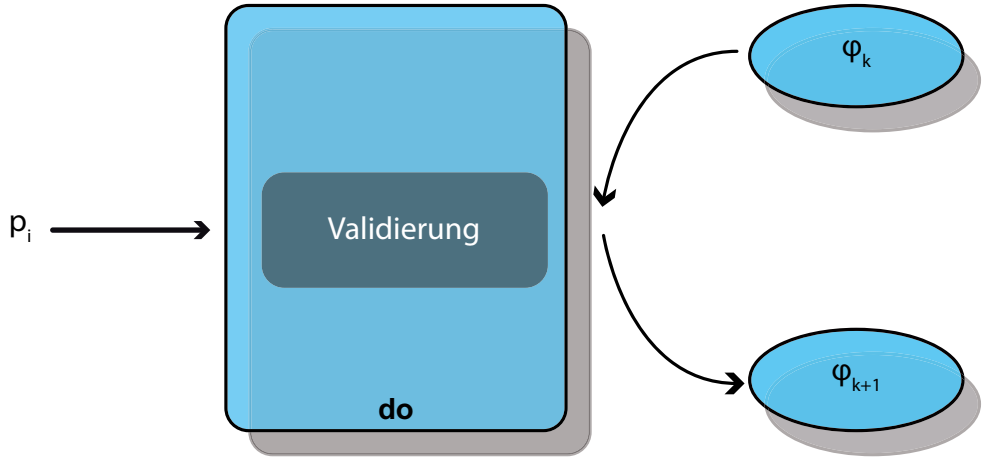

Abbildung 3.2.: do als Übergang δ zwischen zwei Zuständen φ_k und φ_{k+1}

Φ ist die Zustandsmenge, mit $\Phi \subset 2^{\mathcal{P}}$

Σ ist das Eingabealphabet, mit $\Sigma = \mathcal{P}$

δ ist die dynamisch erzeugte Zustandsübergangsfunktion, vgl. hierzu die Erläuterungen zu do in Abschnitt 4.1.2.1

S ist der Startzustand ($S = \emptyset$)

F ist der Endzustand ($F = \mathcal{P}$)

Die Zustandsübergangsrelation do prüft in jedem Zustand φ_k, ob die Abhängigkeiten des übergebenen Prozessschrittes p_i erfüllt sind. Ist dies der Fall geht der Automat in einen neuen Zustand $\varphi_{k+1} = \varphi_k \cup \{p_i\}$ über. Dieser Vorgang wird detailliert in Abschnitt 4.1.2.1 beschrieben. Eine solche implizit gegebene Zustandsübergangsrelation bietet den Vorteil,

dass nicht alle möglichen Übergänge gespeichert werden müssen. Stattdessen wird nur überprüft, ob alle Voraussetzungen für den geforderten Zustandsübergang erfüllt sind. Ist dies der Fall wird der Zustandsübergang vollzogen. Dies ist aufgrund der statischen Problemstruktur, die sich nach der Initialisierung nicht mehr verändert, effizient durch Speicherung der Abhängigkeiten möglich.
Die Möglichkeiten für den nächsten auszuführenden Prozesschritt p_i, also die Menge gültiger Eingaben im aktuellen Zustand, liefert die Funktion `next`, die in Abschnitt 4.1.2.2 beschrieben wird. Im finalen Zustand sind alle Prozessschritte ausgeführt worden.

3.2.3. Interaktion

Den größten Vorteil bei der Vereinfachung des Problems bringt die Einführung der Interaktion. Hierzu wurde für das Finden möglicher nächster Schritte/Elemente eine Modifikation der Breitensuche [LUGER 2001, Seite 122 -129] in Tiefe 1 verwendet. Hier wird eine Menge von Elementen auf gegebene Kriterien geprüft. Durch die Einschränkung auf Tiefe 1 sind dies maximal N Elemente pro Suche. Der Zustandsraum vereinfacht sich wesentlich, da die Entscheidung für ein Element an eine externe Instanz, wie zum Beispiel einen Benutzer, eine Heuristik oder Software übertragen wird, ohne weitere Möglichkeiten vorab zu prüfen.
Im speziellen Fall wird pro Zustandsübergang überprüft, welche Prozessschritte ausführbar sind. Anschließend wird die Entscheidung für den auszuführenden Prozessschritt p_i dem Benutzer überlassen. Somit sind pro Zustandsübergang im Durchschnitt nur noch $N/2$ Prozessschritte zu prüfen. Die Gesamtkomplexität beläuft sich also $\frac{N^2}{2}$. Hiermit wurde die Größe des Zustandsraumes logarithmiert und es sind erheblich größere Probleme lösbar. Implementierungsdetails hierzu finden sich im Abschnitt 4.1.2.2.
Obiges Vorgehen lässt sich auf viele Probleme übertragen und schließt die Benutzung von zusätzlichem Wissen, zusätzlicher Heuristiken oder lernender Algorithmen nicht aus. Wesentlich ist lediglich die Einschränkung auf den kurzen Lookahead. Dieser kann je nach Anwendung und Notwendigkeit auch größer gewählt werden, was allerdings den Faktor der Vereinfachung negativ beeinflusst. So wird durch die Einschränkung der Entscheidung auf den jeweils nächsten Schritt das Problem erheblich vereinfacht. Durch die statische

Natur des Problems muss bei dieser Einschränkung nicht gesondert auf Abhängigkeiten innerhalb des Weges geachtet werden, da diese bereits in der Initialisierungsphase (vgl. Abschnitt 4.1.1) vollständig erfasst und gespeichert wurden. Deshalb genügt die Suche nach dem jeweils nächsten Element, die auf die gespeicherten Abhängigkeiten zugreift und somit die Validität des Ergebnisses gewährleistet.

4. Effizienz

In Abschnitt 3.2 wurde die Grundidee des Prozessnavigationssystems erläutert. In diesem Abschnitt soll nun dessen effiziente Implementierung behandelt werden. Die Programmierung erfolgte in PROLOG, für dessen Grundlagen hier auf die zahlreiche Literatur verwiesen sei. (BLACKBURN u. a. [2006], BELLI [1988]) Für die Objekte in Abschnitt 4.2 wurde LOGTALK verwendet. Hierzu finden sich viele nützliche Informationen in der Literatur (MOURA, MOURA [2009]).
Der erste Abschnitt befasst sich hierbei mit der Implementierung des Grundsystems, also der Generierung des Automaten. Der zweite Abschnitt behandelt die objektorientierte Komponente und zum Schluss dieses Kapitels wird auf einige Feinheiten bei der Implementierung eingegangen die zur Effizienz des Systems beitragen.

4.1. Implementierung

Dieser Abschnitt behandelt die Generierung des endlichen Automaten aus Abschnitt 3.2.2 in drei Abschnitten. In der Initialisierungsphase werden Start und Endzustand generiert, Objekte (vgl. Abschnitt 4.2) erzeugt und Abhängigkeiten berechnet. Im Abschnitt zu Funktionen zur Laufzeit wird vor allem die Zustandsübergangsfunktion behandelt. Schließlich soll der Abschnitt zu informativen Methoden dazu dienen zu erklären, wie der Benutzer durch den Prozessweg geführt wird und die dargebotenen Entscheidungsoptionen gefunden werden.
Die Generierung des beschriebenen endlichen Automaten, wird durch Angabe eines Startzustands, eines Endzustands und einer Zustandsübergangsfunktion realisiert.
Die ausgezeichneten Zustände sind einfach zu realisieren. In der Initialisierungsphase wird die Menge $\mathcal{P}$ der Prozessschritte durchlaufen und Start- und Endzustand gesetzt.

4.1.1. Initialisierung

Beim Start des Systems wird eine Prozessmodell übergeben. Wie in Abschnitt 2.2 bereits angedeutet, ist das Prozessmodell ab diesem Zeitpunkt statisch und kann nicht mehr verändert werden. Um eine Änderung des Modells zu berücksichtigen, muss das Prozessnavigationssystem neu gestartet werden. Dies ist nötig, um eine Änderung der Abhängigkeiten zur Laufzeit auszuschließen. Dies würde in Verbindung mit der in Abschnitt 3.2.3 beschriebenen Einschränkung des Lookahead der `next()`-Suche auf 1 im Allgemeinen zu ungültigen Ergebnissen führen, da die Suche nur die in der Initialisierungsphase erkannten Abhängigkeiten (Abschnitte 4.1.1.2 und 4.1.1.3) berücksichtigt. Hierdurch könnten, bezüglich des geänderten Modells, nicht zulässige Abfolgen vorgeschlagen und ausgeführt werden. Neben den Abhängigkeiten könnten auch die ausgezeichneten Zustände geändert werden, was die Einschränkung auf statische Modelle weiter begründet.
Die Erzeugung der Zustände wir in folgendem Abschnitt erläutert.

4.1.1.1. Start- und Endzustand

Listing 4.1: Initialisierung von Start- und Endzustand

```
1 init_state ([done:[]| Inits ]) :-
2         findall (X, proc(X,_,_,_), Init),
3         sort(Init, Inits).
4
5 final_state ([done:Final |[]])     :-
6         findall (X, proc(X,_,_,_), Fin),
7         sort(Fin, Final).
```

Start- und Endzustand bestehen aus zwei sortierten Listen. Das erste Element der Prozessschrittliste ist eine Liste mit vorangestelltem `done:`. Dies beschreibt die bereits abgearbeiteten Prozessschritte. An zweiter bis letzter Stelle stehen die Identifier der abzuarbeitenden Prozessschritte. Somit wächst im Laufe der Prozessausführung das erste Glied, nämlich die `done`-Liste, während der Rest der Liste schrumpft. Es rutschen also sukzessive Elemente aus der Liste nach vorne, bis die Liste nur noch aus einem Element, nämlich der `done`-Liste, besteht.

Für die Realisierung des endlichen Automaten fehlt noch eine geeignete Zustandsübergangsrelation. Sie wird durch die Funktion do bereitgestellt. Da der Automat dynamisch aus dem Prozessmodell erzeugt wird, muss die implementierte Zustandsübergangsrelation ebenfalls in der Initialisierungsphase die nötigen Informationen sammeln. Dies geschieht über die Betrachtung der im Modell definierten Abhängigkeiten und Verbindungen. Hierzu wird überprüft, wie und mit welchen anderen Prozessschritten ein Prozessschritt X verbunden ist.

4.1.1.2. Statische Abhängigkeiten

Listing 4.2: Abhängigkeiten der Prozessschritte

```
1 require(X, R) :-
2     sconn(W,X)                    -> depList(W, R);
3     (sconn(V,W), rec_box(W, X)) -> depList(V, R);
4                                    R = [].
```

Gibt es eine solide Verbindung mit einem anderen Element (Box oder Prozessschritt), so muss dieser zuvor ausgeführt worden sein. Dies übernimmt die Funktion `depList()`. Ist ein Prozessschritt X Element einer Box B, und ist B solide mit einem anderen Element Y verbunden, so muss Y vor dem Element X ausgeführt werden. Den Transfer dieser Abhängigkeit übernimmt wiederum die Funktion `depList()`.

`depList()` berechnet rekursiv Abhängigkeiten von Elementen. Hierzu prüft die Funktion in jedem Durchlauf, ob das gerade betrachtete Element X ein Prozessschritt oder eine Box ist.

Listing 4.3: Die rekursive Suche nach Abhängigkeiten

```
1 depList(X, RL)      :-  xor((box(X, List),depRList(List, RL))   ,   RL = [X]).
2
3 depRList ([],[]) .
4 depRList([X|L], RL) :-
5             depList(X, DL),
6             depRList(L, DRL),
7             union(DL, DRL, RL).
```

Ist das übergebene Element X eine Box, muss die Elementeliste $List$ der Box als Abhängigkeit übernommen werden. `DepRList()` durchläuft die Liste und ruft wiederum `depList()` für jedes Element auf. Das ist nötig, weil in eine Box weitere Boxen geschachtelt sein können. Deshalb muss jeweils überprüft werden ob es sich um einen Prozessschritt handelt, um gegebenenfalls auf eine Box reagieren zu können.
Ist das übergebene Element X ein Prozessschritt, so wird X der Abhängigkeitsliste hinzugefügt. Durch die rekursiven Aufrufe fügt `depRList()` gegebenenfalls mehrere Abhängigkeitslisten zusammen.

4.1.1.3. Dynamische Abhängigkeiten

Es werden im aktuellen System noch weitere Abhängigkeiten überprüft. Diese werden aktuell und im bestehenden Modell nur benötigt, um flexibel verbundene Boxen zu realisieren. Sie bieten allerdings wesentlich größeres Potenzial. Die Funktionsweise der **dynamischen Abhängigkeiten** entspricht dem einer nachrichtengesteuerten Blackbox. Ein Abhängigkeitsobjekt erhält Informationen über den aktuellen Zustand des Systems und antwortet mit den Abhängigkeiten des Prozessschrittes im aktuellen Zustand. Der Zustand wird auf durch die dynamischen Abhängigkeitsregeln definierte Triggerelemente überprüft und gibt die zugehörigen Elemente zurück. In der aktuellen Version sind dies:

Listing 4.4: dynamische Abhängigkeiten

```
1  dyndeps(X, DL) :-  setof(D, findDyn(X,D), DyL) -> DL = DyL;
2                                DL =[].
3
4  findDyn(X, D)  :-  dconn(X,Y),
5                     depList(Y,BList),
6                     sort(BList, Box),
7                     D=Box:Box.
8
9  findDyn(X, D)  :-  rec_box(Y,X),
10                    dconn(Z,Y),
11                    depList(Z,BList),
12                    sort(BList, Box),
13                    D=Box:Box.
```

In Listing 4.4 werden die Prozessschritte X lediglich auf ein Faktum überprüft:
Die erste `findDyn()` Regel überprüft ob ein Element X flexibel mit einer Box verbunden ist. Ist dies der Fall, so werden alle Elemente der Box als Trigger in die dynamischen Abhängigkeiten von X geschrieben, die wiederum alle Elemente der Box als Abhängigkeit auslösen. Hierzu wird, wie oben, CodedepList() verwendet. Zur Laufzeit bedeutet dies: wird ein Element der Box **vor** X ausgeführt, müssen alle Elemente vor X ausgeführt werden.
Die zweite `findDyn()` Regel ist für das umgekehrte Verhalten zuständig. Sie überprüft ob ein Element in einer Box liegt, die flexibel verbunden ist. Hierbei kann das Element X beliebig tief verschatelt sein. Ist dies der Fall wird das verknüpfte Element als Trigger und Event, mit Hilfe von `depList()`, in die dynamischen Abhängigkeiten des Prozessschrittes X geschrieben. Dies wäre für Boxen die mit einem Einzelschritt verknüpft sind nicht wichtig, da der Schritt sobald er den Trigger auslösen kann bereits ausgeführt ist. Bei flexibler Verknüpfung zweier Boxen wird hierdurch jedoch sichergestellt, dass flexibel verbundene Boxen tatsächlich atomar behandelt werden, Also erst alle Elemente einer Box ausgeführt worden sein müssen, bevor mit der Ausführung der Schritte, die in der anderen Box gekapselt, sind begonnen werden kann.
`dynDeps()` ist bewusst flexibel angelegt, so dass jederzeit weitere Regeln (`findDyn()`) ergänzt werden können. Dies bietet größtmögliche Flexibilität bei der Interpretation eines Modells, so dass theoretisch viele verschiedene Sprachen mit demselben System zusammen geführt werden können. Hierzu Weiteres im Abschnitt 5.2.

Nachdem die Abhängigkeiten in der Initialisierungsphase festgestellt und wie in Abschnitt 4.2.3 beschrieben gespeichert wurden, muss die Zustandsübergangsrelation nur noch die pro Prozessschritt gespeicherten Abhängigkeiten überprüfen, um einen zulässigen Zustandsübergang zu vollführen.

4.1.2. Laufzeit

Nach der Initialisierung steht ein endlicher Automat im Startzustand mit Zustandsübergangsfunktion und ausgezeichnetem Endzustand bereit. Es können also Zustandsübergänge vollzogen, ergo Prozessschritte ausgeführt werden.

4.1.2.1. Zustandsübergang

Die Zustandsübergangsfunktion `do()` ist in drei Teile aufgeteilt.

`start()` Überprüft die Abhängigkeiten und aktiviert den Prozessschritt.

`execute/run()` Führen Instanzen eines Prozessschrittes aus, wenn er aktiviert ist.

`stop()` Stoppt die Ausführung des Prozesschrittes und erzeugt einen neuen Zustand.

Listing 4.5: Die Zustandsübergangsrelation do

```
1  % Do a process 'X' in a certain 'State' N-times. Get 'NewState' back....
2  do(State, X:N, NewState)   :-
3                  % Start the Process
4                  start(State, X),
5                  % Execute it N times
6                  X::execute(N),
7                  % Stop the execution and create the new State
8                  stop(State, X, NewState).
9
10 start([done:Done|Todo], X) :-
11                 % Find a executable process step
12                 member(X,Todo),
13                 % Check the prerequisites of the process step
14                 X::getDeps(Done, Dependencies),
15                 % RULE: All prerequisites must be done
16                 done(Done, Dependencies),!,
17                 % Start the process step
18                 X::start.
19
20 run(X)      :- % Verify activeness of the process step
21                 X::active,
22                 % Run the process step once
23                 X::run.
24
25 cancel(X)   :-      X::cancel.
26
27
28 stop(State, X, NewState)   :-
29                 % Stop the process step
30                 X::stop,
31                 % Create New State
```

```
32                  create_new_state(State, X, NewState).
33
34 % Subtract X from ToDo and return new state
35 create_new_state([done:Done|Todo], X, [done:NDone|LeftTodo]) :-
36                  subtract(Todo, [X], LeftTodo),
37                  sort([X|Done], NDone),!.
```

In `start()` wird ein Elemente X aus der Liste der ausführbaren Elemente entnommen und dessen Abhängigkeiten überprüft. Wurden alle Prozessschritte ausgeführt, die in der Abhängigkeitsliste von X stehen, so wird der Prozessschritt aktiviert.
Die Funktion `execute(N)` führt einen Prozessschritt N-mal aus. `run()` überprüft erst den Status des Prozessschrittes und führt ihn gegebenenfalls einmal aus. Diese Überprüfung ist bei `execute()` nicht nötig, da die Funktion nicht außerhalb von `do()`, in der `start()` zwangsläufig vorher ausgeführt wird, aufgerufen wird.
Der letzte aufgerufene Befehl ist schließlich `stop()`. Er stoppt die Ausführung des Prozessschrittes und erzeugt einen neuen Zustand. Die Überprüfung der Domäne auf Gültigkeit erfolgt hierbei implizit in der `stop()`-Funktion des Prozessschrittes, so dass die Funktion nur einen neuen Zustand zurückgibt, wenn die Kardinalität des Prozessschritt in seiner Domäne liegt.

4.1.2.2. Informative Methoden

Im laufenden Betrieb stehen weitere informative Methoden, wie in Abschnitt 2.1 beschrieben, zur Verfügung. Hierzu werden Zustandsübergänge simuliert, um Folgen einer Entscheidung zu evaluieren. Eine dieser informativen Methoden ist `next()`.

Listing 4.6: Informative Methoden

```
1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 % Informational Methods
3 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
4 next(State, Next)   :-  setof(X, executable(State,X), Next).
5
6 somewhen([done:_|Todo], Todo).
7
8 not_after(State, X, Never_Again)   :-  do_notRLY(State, X, [done:Never_Again|_]).
```

`next()` sammelt alle Prozessschritte die laut `executable()` ausführbar sind und liefert so eine Liste möglicher nächster Elemente. `somewhen()` und `not_after()` nutzen die Struktur des Zustands aus, indem sie die entsprechenden Listen im Zustand zurückgeben, ohne tatsächliche Berechnungen durchführen zu müssen.

Die Funktionen `executable()` und `not_after()` greifen auf eine Zustandsübergangssimulation zu.

Listing 4.7: Zustandsübergangssimulation

```
1  do_notRLY([done:Done|Todo], X, NewState) :-
2              % Find a executable process step
3                      member(X,Todo),
4
5                      % Check the prerequisites of the process step
6                      X::getDeps(Done, Dependencies),
7
8                      % RULE: All prerequisites must be done
9                      done(Done, Dependencies),
10
11                     % Create new (fake) State
12                     create_new_state([done:Done|Todo], X, NewState).
```

`do_notRLY()` verhält sich wie in Abschnitt 4.1.2.1 und Listing 4.5 beschriebene Zustandsübergangsfunktion, nur führt sie die Prozessschritte nicht aus. Sie ändert also die zugehörigen Objekte nicht.

4.2. Datentypen

Die Speicherung gewonnener Daten ist ein grundsätzliches Problem der funktionalen Programmierung. Als effiziente Möglichkeit Daten zum Prozessschritt, Domänen und Abhängigkeiten zu speichern, wurde vom Autor ein objektorientierter Ansatz gewählt. Grundsätzlich unterscheidet sich die Funktionsweise kaum von der bekannten in der imperativen Programmierung, ist jedoch an manchen Stellen etwas unkonventionellg. Für weitere Informationen zur Objektorientierung sei hier auf die gängige Literatur verwiesen. (POETZSCH-HEFFTER [2000]) Grundsätzlich werden die Objekte und Klassen in der `Prolog` ähnlichen Sprache `Logtalk` definiert und anschließend durch einen Precompiler in `Prolog`-Code mit einer beschreibenden `XML`-Datei konvertiert. Im `Prolog` Programm werden die Objekte durch Nachrichten angesprochen, die durch `::` zu erkennen sind.

Im speziellen Fall wurden die Prozessschritte in drei Objekte zerlegt.

Dependency
: Das Abhängigkeitsobjekt speichert statische Abhängigkeiten, stellt Methoden zur Berechnung dynamischer Abhängigkeiten bereit und berücksichtigt Datenabhängigkeiten.

Domain
: Das Domänenobjekt speichert die Domänengrenzen, berechnet dynamische Domänengrenzen zur Laufzeit und stellt Methoden zur Validierung eines Wertes bereit.

Process
: Das Prozessschrittobjekt stellt Methoden zum *Starten*, *Ausführen*, *Stoppen*, *Abbrechen*, etc. bereit, verfügt über einen Zähler, referenziert Domänen- und Abhängigkeitsobjekt, speichert den Zustand des Prozessschrittes und implementiert zahlreiche Methoden zum Zugriff und zur Verifikation.

Diese Gliederung ist aus verschiedenen Gründen sinnvoll: Die Programmierung wird durch die Unterteilung übersichtlicher und der Zugriff auf betriebskritische Attribute kann durch Deklaration als `private` kontrolliert werden. Des Weiteren können so erzeugte Abhängigkeits- und Domänenobjekte für viele Prozessschrittobjekte verwendet werden, was den erforder-

lichen Speicherplatz reduziert.

4.2.1. Prozessschrittobjekt

Das Prozessschrittobjekt ist das Herzstück des objektorientierten Ansatzes. Es gewährleistet Interaktivität, Flexibilität und dient als Wrapper für Domänen- und Abhängigkeitsobjekt. Die Funktionalität des Objekts entspricht den benötigten aus Abschnitt 3.2.3 und 4.1.2.1. Es stellt Befehle zum Starten, Stoppen, etc des Prozessschrittes bereit (vgl. Abbildung 4.1).

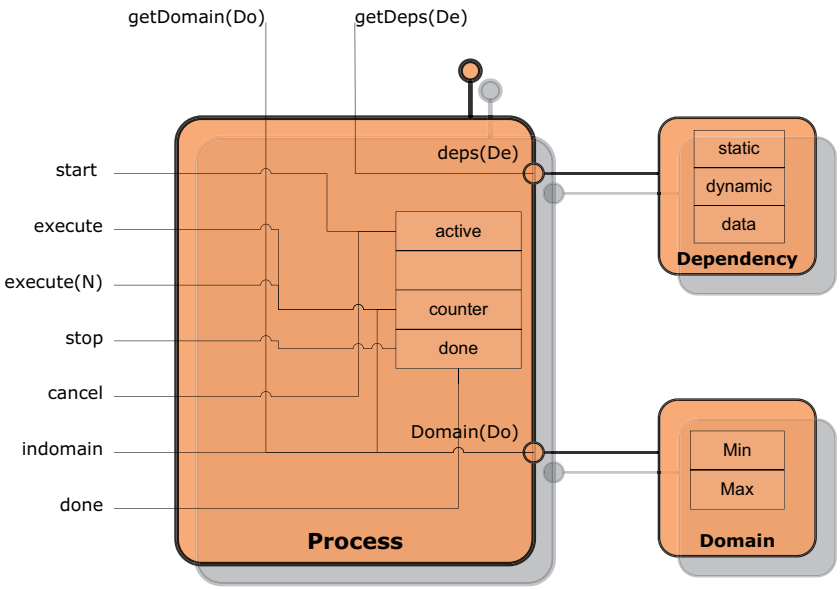

Abbildung 4.1.: Das Prozessschrittobjekt

start

Überprüft ob der Prozessschritt nicht schon ausgeführt wurde, startet gegebenenfalls die Ausführung eines Prozessschrittes.

execute

Überprüft ob der Prozessschritt gestartet ist, und erhöht den Zähler um 1 wenn damit die Prozessschrittdomäne nicht verletzt wird.

execute(N)

Überprüft ob der Prozessschritt gestartet ist und erhöht den Zähler um N.

stop

Überprüft ob der Prozessschritt gestartet ist und der Zähler in der Prozessschrittdomäne liegt und stoppt anschließend den Prozess.

cancel

Überprüft ob der Prozessschritt gestartet ist und bringt den Prozessschritt in den Anfangszustand.

indomain

Überprüft ob der Prozessschrittzähler in seiner Domäne liegt.

done

Überprüft ob der Prozessschritt bereits ausgeführt wurde.

domain(Do)

Liefert die Referenz des Domänenobjekts.

deps(De)

Liefert die Referenz des Abhängigkeitsobjekts.

Obige Funktionen sichern ein vorhersehbares Verhalten der Prozessschritte. Ein Prozessschritt kann nur *ausgeführt* sein, wenn sein Zähler innerhalb der Domäne liegt. Dies gilt auch für dynamische Domänengrenzen, da modellbedingt dynamische Domänen nur Prozesschritte vor dem abhängigen referenzieren dürfen. Ein Abbruch eines Prozessschrittes bewirkt das Zurücksetzen aller Attribute, so dass der Prozessschritt sich nicht von einem ungestarteten unterscheidet.

4.2.2. Domänenobjekt

Das Domänenobjekt übernimmt drei Aufgaben. Es speichert die Domänengrenzen, stellt eine Validierungsfunktion bereit und automatisiert den Zugriff auf die Zähler der referenzierten Objekte bei dynamischen Domänengrenzen.

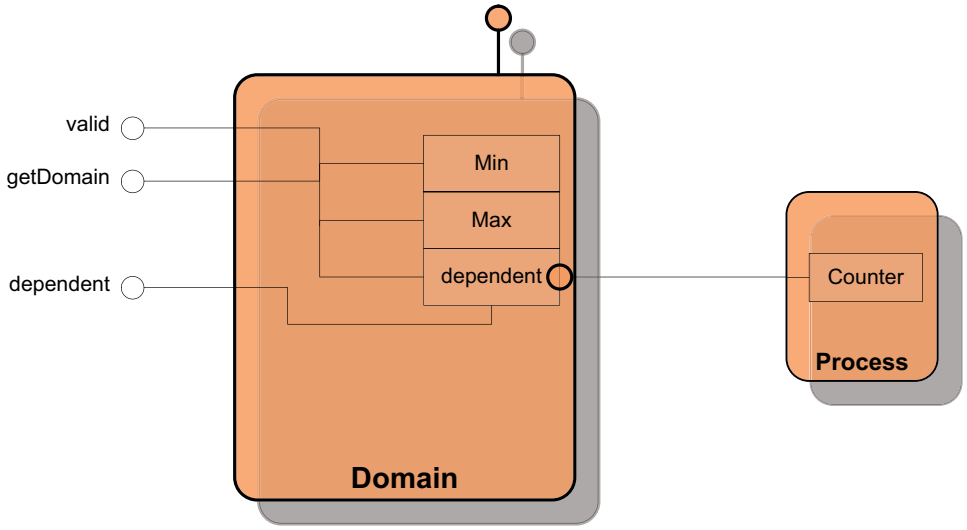

Abbildung 4.2.: Das Domänenobjekt

dependent

Liefert die Information, ob es sich um eine dynamische Domäne handelt.

getDomain(Min,Max)

Gibt die (ggf. dynamischen) Domänengrenzen zurück.

valid(X)

Überprüft ob X in der Domäne liegt.

Die Rückgabe der dynamischen Domänengrenzen ist ein kritischer Punkt. Tatsächlich könnten die Domänengrenzen angefordert werden, bevor die referenzierten Prozessschritte ausgeführt wurden. In diesem Fall würde die Abfrage 0 zurück geben, da dies der Zählerwert des referenzierten Counters ist. Deshalb ist in diesem Fall das Triggern einer Warn- oder Fehlermeldung sinnvoll!

4.2.3. Abhängigkeitsobjekt

Das Abhängigkeitsobjekt dient dem komfortablen Zugriff auf verschiedene Abhängigkeitstypen. Es speichert Abhängigkeiten von Prozessschritten, Daten und dynamische Abhängigkeiten.

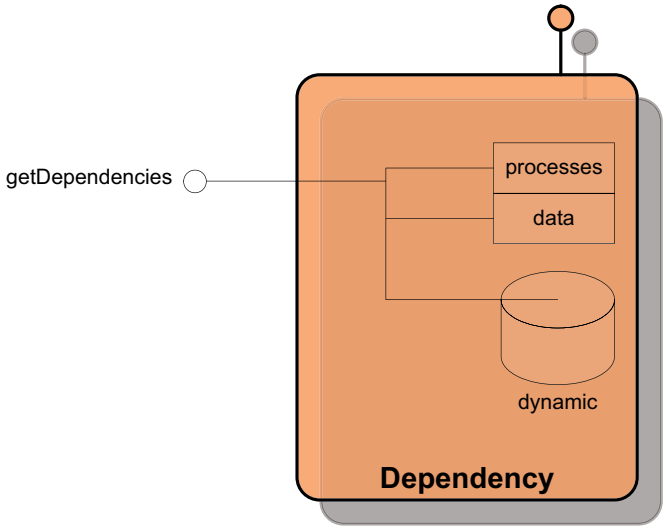

Abbildung 4.3.: Das Abhängigkeitsobjekt

getDependencies

Liefert die Abhängigkeiten des Prozessschrittes zurück. Bei einem einfachen Aufruf werden nur statische Abhängigkeiten (Prozessschritte und Daten) zurückgegeben. Werden der Funktion zusätzlich die bereits abgearbeiteten Prozessschritte übergeben, liefert es auch die dynamischen Abhängigkeiten.

Die Daten- und dynamischen Abhängigkeiten erlauben eine sehr hohe Modellierungsflexibilität. Hier können zusätzliche Abhängigkeiten vermerkt (Daten), oder Wenn-Dann Beziehungen implementiert werden (dynamische Abhängigkeiten, Abschnitt 4.1.1.3). Dies macht das System noch wesentlich flexibler als es für die Implementierung der gegebenen Prozessmodelle nötig gewesen wäre.

4.3. Geschwindigkeitssteigerung

Zusätzlich zu den konzeptionellen Stärken des Systems konnten einige weitere Maßnahmen die Performanz zur Laufzeit steigern. Hierzu wurde vor allem die Struktur der Zustände ausgenutzt.

Listing 4.8: Ein typischer Zustand

```
1 State = [done:[p1, p2], p3, p4]
```

Der Zustand besteht aus drei Elementen. Der done-Liste und zwei nicht ausgeführten Prozessschritten. Dies kann durch Prologs Operator für den Zugriff auf das erste Element einer Liste dazu benutzt werden, schon im Funktionskopf irrelevante Daten auszublenden.

Listing 4.9: Ein typischer Zustand

```
1 function ([done:Done|Todo]) :- do_something(Done), do_something_else(Todo).
```

Es werden also keine wie sonst oft üblichen aufwändigen append Aufrufe benötigt um unterschiedliche Elemente des Zustandsraumes zu untersuchen. Die Trennung in done-Zustände und todo-Zustände kann einfach durch die Separation der Liste nach dem ersten Element erfolgen.

Ein weiterer Punkt bei der Geschwindigkeitssteigerung ist die Reduktion obsoleter Funktionsaufrufe. Diese trivial wirkende Formulierung ist in der Praxis nicht allzu leicht umzusetzen, denn nur durch geschicktes Ausnutzen der Programmkausalitäten können einige Überprüfungen weggelassen werden. Ein Beispiel hierfür ist die Funktion execute() aus Listing 4.5, die, weil sie in do() fest eingebunden ist, auf eine Überprüfung der Aktivität des zugehörigen Prozessschrittes verzichten kann.

5. Zusammenfassung

Abschließend soll nun das beschriebene Verfahren evaluiert werden. Des Weiteren soll betrachtet werden, inwiefern die angesprochenen Kritikpunkte beachtet werden konnten. Zuletzt soll dieses Kapitel einen Ausblick auf zukünftige Forschungsfelder geben, die sich bei dieser Arbeit als interessant erwiesen, aber nicht oder nicht erschöpfend behandelt werden konnten.

5.1. Evaluation

In Abschnitt 2.1 wurden einige allgemeine Qualitätskriterien für WFMS nach BUSSLER und JABLONSKI [1996] angegeben. Nun soll das System aus Abschnitt 3.2 daran gemessen werden.

Erweiterbarkeit

Ein wichtiges Kriterium für WFMS ist die Erweiterbarkeit. ESProNa realisiert dies durch die freie Regeldefinition zur Interpretation von Abhängigkeiten und die zusätzlichen Attribute in den Objekten. So können, zur Laufzeit, beliebig geartete zusätzliche Daten gespeichert und wieder abgefragt werden.

Anpassbarkeit

Für die Anpassbarkeit gelten die gleichen Argumente wie für die Erweiterbarkeit. Durch die offene Architektur und den modularen Charakter, kann das System auf nahezu alle Anwendungsbereiche zugeschnitten werden. Die redefinierbare Syntax, das große Potenzial der dynamischen Abhängigkeiten und die Unabhängigkeit des Hintergrundsystems von der Benutzerschnittstelle machen das System vollkommen adaptierbar.

Dynamisch modifizierbar

Durch die dynamischen Abhängigkeiten und die Möglichkeit zur Speicherung zusätzlicher Attribute ist das System auch im produktiven Betrieb noch variabel. Die

bei konventionellen WFMS oft gestellte Problematik des Änderns eines Standardvorgangs stellt sich konzeptionell nicht, da ESProNa immer alle Möglichkeiten einen Prozessweg zu durchlaufen berücksichtigt.

Vollständigkeit

Die vollständige Implementierung der von IGLER vorgestellen Modellarchitektur wurde in zahlreichen Testanwendungen evaluiert. Die Kernfunktionen sind: solide und flexible Verbindungen, Prozesschritte und deren KArdinalität sowie Boxen. Vor allem die Verknüpfung mehrerer Boxen erforderte innovative Konzepte, wie die dynamischen Abhängigkeiten. Hierdurch kann ebenfalls garantiert werden, dass Erweiterungen und Modifikationen am Modell auch zukünfig integriert werden können, da hierzu lediglich die entsprechenden Regeln ergänzt werden müssen.

Wiederverwendbarkeit

Durch die freie Wählbarkeit der Modellierungskonstrukte und deren Interpretation kann das System auf beliebige Anwendungen zugeschnitten werden. Somit kann das Hintergrundsystem beliebig wiederverwendet werden und unterschiedlichste Abläufe modellieren, sowie unterschiedliche Benutzerschnittstellen bedienen.

Integrierbarkeit

Die modulare Architektur ermöglicht eine Kapselung des Hintergrundsystems. Dies ermöglicht die Implementierung der Benutzerschnittstelle in einer zu der gegebenen Plattform passenden Sprache zu realisieren. Die Kommunikation der Schnittstelle mit dem Hintergrundsystem erfolgt über Nachrichten, die alle gängigen Betriebssysteme unterstützen.

Nach der Überprüfung der allgemeinen Kriterien soll in diesem Abschnitt auf die im Eingangskapitel (Abschnitt 1.1) zitierten Kritikpunkte Bezug genommen werden.
Durch die Interaktivität des Systems, kombiniert mit der Vollständigkeit möglicher Lösungen, kann der Benutzer auf jede Situation adäquat reagieren. Dem Mitarbeiter steht vollkommen frei, welche Abfolge er wählt. Hiermit fühlt sich der Mitarbeiter nicht bevormundet und behält seine Eigenverantwortlichkeit. Jeder Mitarbeiter kann gewohnte und eingespielte Handlungsabfolgen beibehalten, entwickeln und neu finden, da das System nur beratend zur Seite steht. Es gibt also keine festen Abläufe vor und bemängelt auch keine Abweichungen von einem gegebenen Handlungsstrang. Das System stellt lediglich sicher, dass eine Prozessschrittfolge auch zum gewünschten Ergebnis führt. Es stellt also

einen groben Ablaufrahmen zur Verfügung, der alle im Modell definierten Kausalitäten berücksichtigt.
Die Mitarbeiter sind somit frei in ihrer Arbeitsweise und Kooperation, so dass bestehende, eingespielte Teams nicht zerstört werden. Durch das Angebot aller validen Optionen als nächsten auszuführenden Schritt werden innovative Ansätze gefördert, da der Benutzer Alternativen vor Augen hat. Er kann also einen neuen Lösungsweg probieren und muss sich dabei keine Sorge um den Erfolg des Workflow machen, da dieser durch die Option im System bereits gesichert ist. Dies bietet die Möglichkeit unkonventionelle Lösungsansätze auch in kritischen Workflows zu testen.

5.2. Ausblick

Im Folgenden sollen einige Aspekte behandelt werden, die bei der Bearbeitung des Themas aufgefallen sind und leider nicht ausreichend betrachtet werden konnten, aber für folgende Arbeiten Beachtung verdienen.

5.2.1. Sichten und Abstraktion

Für verschiedene Anwendungsbereiche kann es sinnvoll erscheinen, weitere Abstraktionen vorzunehmen. So könnte der Ansatz aus Abschnitt 3.2.1 auf verschiedene andere Modellierungselemente, wie die Box, angewandt werden. Ist es für eine Anwendungsdomäne interessant, die Abläufe, etwa bedingt durch personelle, räumliche oder zeitliche Trennung, stärker zu Kapseln, brächte dies einen wesentlichen Performanzvorteil. Dieser wird jedoch durch wesentlich höheren Aufwand in der Abhängigkeitsmodellierung geschmälert, so dass die Sinnhaftigkeit für jede Anwendungsdomäne neu überprüft werden muss.
Ein solcher Ansatz hätte weiter den Vorteil, dass verschiedenen Mitarbeitern nur Zugriff auf eine gekapselte Instanz gewährt werden könnte.

5.2.2. Dynamische Modelle

Aktuell ist das System der flexiblen Prozessnavigation aus Abschnitt 3.2 auf statische Prozessmodelle beschränkt. Dies bedeutet einerseits eine einfache Möglichkeit zur Ab-

hängigkeitsberechnung in der Initialisierungsphase, auf der anderen Seite jedoch einen recht großen Aufwand in sich häufig ändernden Anwendungsbereichen. Wird nämlich ein Prozessmodell geändert, so muss das System neu initialisiert werden, was aktuelle Abläufe zum Abbruch zwingt.
Es wäre wünschenswert, eine Möglichkeit zu finden die es ermöglicht, eine Abhängigkeit zur Laufzeit zu verändern und so dynamische Modelle zu erlauben. Dies wäre gerade in der Modellierungsphase, in der Modelle häufig angepasst werden, oder bei strukturellen Änderungen hilfreich. Problematisch hierbei ist die Wahrung der Validität vorhandener oder gestarteter Prozessausführungen. Hierfür gilt es passende Ansätze und Kriterien zu finden.

5.2.3. Multithreading

Das aktuelle System verfolgt eine fest definierte Lösungsstrategie. Zwar ist dies eine sehr effiziente Möglichkeit große Probleme zu lösen, allerdings ist es denkbar für spezifische Anwendungen und Modelle Spezialalgorithmen zu finden, die besser geeignet sind. Ein solcher Ansatz wäre in der Lage, viele verschiedene Lösungsstrategien gleichzeitig anzuwenden und die schnellste, am besten geeignete, günstigste oder nach beliebigen Kriterien präferierte Lösung zurückzugeben. Für diesen Fall wäre ein Multithreadingansatz mit kopetitierenden Algorithmen denkbar. Da Prolog nativ Multithreading unterstützt, geht hiermit nur minimaler Overhead einher.

5.2.4. Dynamische Abhängigkeiten

Das meiste Potenzial im System bieten die dynamischen Abhängigkeiten. Durch die hochflexible `Trigger:Event` Architektur können sie für viele der oben genannten Aspekte verwendet werden. Dynamische Abhängigkeiten überprüfen den aktuellen State auf Trigger, wird einer dieser Trigger gefunden, werden die zugehörigen Abhängigkeiten als Event zurückgegeben. Besonders interessant an diesem Ansatz ist die freie Definierbarkeit von Trigger und Event Konstellationen. Es ist natürlich möglich, auch statische Abhängigkeiten über dynamische zu realisieren, jedoch ist wesentlich interessanter, die Möglichkeiten zur Abhängigkeitsdefinition völlig frei zu stellen. Hiermit wird ermöglicht, das System exakt auf eine Anwendungsdomäne und deren spezifisches Vokabular zuzuschneiden und zu-

sätzliche Modellierungswerkzeuge zu schaffen. Dies macht das vorliegende System sowohl auf verschiedene Modellierungssprachen, Domänen und spezifische Anwendungsfälle adaptierbar. Um dies in vollem Umfang zu ermöglichen wäre die Schaffung eines Standards für die Abhängigkeitsmodellierung und deren Implementierung in das System sinnvoll. Dann könnte zusammen mit Spezialisten aus der Anwendungsdomäne ein spezifisches Regelwerk erarbeitet werden, das den Bedürfnissen der Domäne optimal entspricht. Hierzu sind keine Anpassungen im Hintergrundsystem nötig. Es ist also möglich, vollkommen unterschiedliche Verhaltensweisen und Spezialanwendungen mit einem System zu realisieren.

5.2.5. Benutzerschnittstelle

Durch die offene Gestaltung des Hintergrundsystems und lässt sich zusammen mit der objektorientierten Prozessschrittimplementierung eine spezialisierte Benutzerschnittstelle erzeugen. Hier ist ein Anwendungsspezifisches Informationsangebot denkbar. Das Hintergrundsystem ist hierfür, durch die Möglichkeit zur Erweiterung der zugrunde liegenden Objekte und Klassen durch Zusatzattribute, optimal gerüstet. Das System kann je nach Präferenz und Anwendung textuell, tabellarisch oder grafisch ausfallen, da es vollkommen unabhängig von der Hintergrundarchitektur ist. Durch eine geschickte Wahl der Architektur kann Plattformunabhängigkeit und breite Kompatibilität erreicht werden.

5.3. Fazit

Mit ESProNa ist es gelungen ein Workflow Management System zu entwickeln, das konzeptionell die Kritikpunkte an gängigen Anwendungen umgeht. Durch den flexiblen Ansatz ergeben sich neue, ungewöhnliche Problemstellungen die in dieser Arbeit aufgezeigt, diskutiert und gelöst wurden. Die Implementierung des Systems in `Prolog` ist einer der ungewöhnlichen Wege die eingeschlagen wurden um das System performant und flexibel zu gestalten.
Viele der hier gebotenen Lösungsansätze können auf andere Problembereiche übertragen

werden, wie zum Beispiel die Kapselung, die Zustandsraumreduktion oder auch die Trigger:Event Architektur.
ESProNa ist ein weiterer, wenn auch unkonventioneller, Schritt Workflow Management Systeme praxistauglich, anpassbar und mitarbeitersorientiert zu gestalten.

Literaturverzeichnis

BELLI 1988

BELLI: *Einführung in die logische Programmierung mit PROLOG*. 1988 4

BLACKBURN u. a. 2006

BLACKBURN, Patrick ; BOS, Johan ; STRIEGNITZ, Kristina: *Learn Prolog Now*. Paperback, online. http://cs.union.edu/~striegnk/learn-prolog-now/lpnpage.php?pageid=online. Version: 2006 4

BOTHNER und KÄHLER 1991

BOTHNER, P. ; KÄHLER, W.-M.: *Programmieren in Prolog*. Vieweg Braunschweig, 1991

BRAUER 1984

BRAUER, Wilfried: *Automatentheorie*. B.G. Teubner Stuttgart, 1984

BUSSLER und JABLONSKI 1996

BUSSLER, Christoph ; JABLONSKI, Stefan: *Workflow Management*. Thomson, 1996 1, 2.1, 2.2, 5.1

CLARK und HOLTON 1994

CLARK, John ; HOLTON, Derek A.: *Graphentheorie*. Spektrum Heidelberg, 1994

DOORES u. a. 1987

DOORES, J. ; REIBLEIN, A. ; VADERA, S.: *Prolog - Programming for tomorrow*. Sigma Wilmslow, 1987

HEINL u. a. 1999

HEINL, Petra ; HORN, Stefan ; JABLONSKI, Stefan ; NEEB, Jens ; STEIN, Katrin ; TESCHKE, Michael: A Comprehensive Approach to Flexibility in Workflow

Management Systems. In: *ACM SIGSOFT Software Engineering Notes* 24 (1999), S. 79 – 88

HOLLINGSWORTH 1994

HOLLINGSWORTH, David: The Workflow Reference Model 10 Years On. In: *Document TC00-1003, Workflow Management Coalition* (1994), S. 295 – 312

HOMUTH 1977

HOMUTH, Horst H.: *Einführung in die Automatentheorie*. Vieweg Braunschweig, 1977

IGLER

IGLER, Michael: *A Declarative Approach to Compact Process Modeling and Flexible Process Execution*. – Definitions of the rules of ESProNa Models and invention of the user interface 1.2, 2.1, 2.2, 2.3, 3.2, 3.2.2, 5.1

IGLER 2008

IGLER, Michael: *ESProNa*. November 2008. – first implementation of the idea 5.3, A.7, A.8, A.9

JABLONSKI 1997

JABLONSKI, Stefan: Architektur von Workflow-Management-Systemen. In: *Informatik - Forschung und Entwicklung* 12 (1997), S. 72 – 81

JABLONSKI 2001

JABLONSKI, Stefan: *Von der ersten Anwenderanalyse zu ersten Systemkonzepten für Workflow-Management-Lösungen*. Universität Erlangen-Nürnberg, 2001

LUGER 2001

LUGER, George F.: *Künstliche Intelligenz*. Pearson, 2001 3.2.3

MOHAN u. a. 1995

MOHAN, C. ; ALONSO, G. ; GÜNTHÖR, R. ; KAMATH, M.: Exotica: A Research Perspective on Workflow Management Systems. In: *Data Engineering Bulletin* (1995), S. 18 – 24

MONJAU und SCHULZE 1992

MONJAU, D. ; SCHULZE, S.: *Objektorientierte Programmierung*. Vieweg Braunschweig, 1992

MOURA

MOURA, Paulo: Multithreaded Programming in Logtalk. 4

MOURA 2009

MOURA, Paulo: *Logtalk Reference*. http://logtalk.org/documentation.html. Version: 01 2009 4

NILSSON und MALUSZYNSKI 1995

NILSSON, Ulf ; MALUSZYNSKI, Jan: *Logic, Programming and Prolog*. John Wiley & Sons, New York, 1995

NOLTEMEIER 1993

NOLTEMEIER, H.: *Informatik I*. Hanser München, 1993 3.2.2

NOLTEMEIER und LAUE 1984

NOLTEMEIER, H. ; LAUE, R.: *Informatik II*. Hanser München, 1984

POETZSCH-HEFFTER 2000

POETZSCH-HEFFTER, Arndt: *Konzepte objektorientierter Programmierung*. Springer, 2000 4.2

WEGENER 2003

WEGENER, Ingo: *Komplexitätstheorie*. Springer, 2003

WFMC 1999

WFMC: *Workflow Management Coalition Terminology & Glossary*. http://wfmc.org/index.php?option=com_docman&task=doc_download&gid=93&Itemid=74. Version: 02 1999. – Free website account needed. 1, 1

Wikipedia 2009

WIKIPEDIA: *Workflow-Management — Wikipedia, Die freie Enzyklopädie*. http://de.wikipedia.org/w/index.php?title=Workflow-Management&oldid=58128327. Version: 2009. – [Online; Stand 25. März 2009] 1, 1.1

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Diplomarbeit mit dem Thema

Ansätze zur flexiblen Navigation in Prozessabläufen

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Mir ist bekannt, dass ich meine Diplomarbeit zusammen mit dieser Erklärung fristgemäß nach Vergabe des Themas in doppelter Ausfertigung und gebunden im Prüfungsamt der Universität Bayreuth abzugeben oder spätestens mit dem Poststempel des Tages, an dem die Frist abläuft, zu senden habe.

Bayreuth, den 6. April 2009

Christoph Günther

A. Anhang

Listing A.1: ESProNa 0.50

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ESPRONA v.50
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% NEW in v0.50
% ------------
% Process Steps, Domains and Dependencies are now Logtalk-objects.
%

% Calculations
:- use_module(library(clpfd)).


% ESProNa Files
:- consult('frequent.pl').
:- consult('sld2Graphviz.pl').
:- consult('export2Graphviz.pl').

% LogTalk Files
:- consult('D:/Data/Studium/DA/logtalk/configs/swi.pl').
:- consult('D:/Data/Studium/DA/logtalk/configs/swihook.pl').
:- consult('D:/Data/Studium/DA/logtalk/configs/xpcehook.pl').
:- consult('D:/Data/Studium/DA/logtalk/configs/swi_set_logtalk_context.pl').

:- consult('C:/Program Files/Devel/Logtalk/compiler/logtalk.pl').

:- logtalk_compile('attributes').
:- logtalk_load('attributes').

:- logtalk_compile('domain').
```

```
33 :- logtalk_load('domain').
34
35 :- logtalk_compile('dependency').
36 :- logtalk_load('dependency').
37
38
39 :- logtalk_compile('process').
40 :- logtalk_load('process').
41
42 :- logtalk_compile('tools').
43 :- logtalk_load('tools').
44
45
46 :- dynamic dashed_arrow/2.
47 :- dynamic solid_arrow/2.
48 :- dynamic box/2.
49 :- dynamic node/1.
50 :- dynamic edge/3.
51
52 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
53 % RULES
54 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
55
56 % Flexible connection between process/box.
57 dconn(X,Y) :- dashed_arrow(X,Y).
58 dconn(X,Z) :- dashed_arrow(Y,Z), dconn(X,Y).
59
60
61 % Strict connection between process/box.
62 sconn(X,Y) :- solid_arrow(X,Y).
63
64
65 process(X, RList) :- require(X, RList).
66
67 require(X, R) :-
68     sconn(W,X)                    -> depList(W, R);
69     (sconn(V,W), rec_box(W, X)) -> depList(V, R);
70                                    R = [].
71
72 depList(X, RL)      :- xor((box(X, List),depRList(List, RL)), RL = [X]).
73 depRList ([],[]) .
74 depRList([X|L], RL) :- depList(X, DL), depRList(L, DRL), union(DL, DRL, RL).
```

```
75
76
77 rec_box(Box, X):- box(Box, MList), ( not((member(X, MList),!)) -> rec_list_box(MList, X);
78                                                                  true).
79 rec_list_box ([],[]) .
80 rec_list_box([B|BList], X) :- not((rec_box(B, X),!)) -> rec_list_box(BList,X);
81                                                       true.
82
83 dyndeps(X, DL) :- setof(D, findDyn(X,D), DyL) -> DL = DyL; DL =[].
84 findDyn(X, D) :- dconn(X,Y), depList(Y,BList), sort(BList, Box),D=Box:Box.
85 findDyn(X, D) :- rec_box(Y,X), dconn(Z,Y), depList(Z,BList), sort(BList, Box),D=Box:Box.
86
87
88 done(_,[]).
89 done(Done,Deps) :- subset(Deps, Done).
90
91 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
92 % DECLARE executable/2 AND STATE TRANSITION do/3
93 % and TRANSITION SIMULATION do_notRLY/3
94 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
95
96 % Which processes are executable in current 'State'?
97 executable(State, X) :- do_notRLY(State, X, _).
98
99
100 % Do a process 'X' in a certain 'State' N-times. Get 'NewState' back....
101 do(State, X:N, NewState) :-
102                           % Start the Process
103                           start(State, X),
104
105                           % Execute it N times
106                           X::execute(N),
107
108                           % Stop the execution and create the new State
109                           stop(State, X, NewState).
110
111
112 start([done:Done|Todo], X) :-
113
114                           % Find a executable process step
115                           member(X,Todo),
116
```

```
117                         % Check the prerequisites of the process step
118                         X::getDeps(Done, Dependencies),
119
120                         % RULE: All prerequisites must be done
121                         done(Done, Dependencies),!,
122
123                         % Start the process step
124                         X::start.
125
126 run(X)       :-
127                         % Verify activeness of the process step
128                         X::active,
129
130                         % Run the process step once
131                         X::run.
132
133 cancel(X)    :-
134                         % Check the prerequisites of the process step
135                         X::cancel.
136
137 stop(State, X, NewState) :-
138
139                         % Stop the process step
140                         X::stop,
141
142                         % Create New State
143                         create_new_state(State, X, NewState).
144
145
146 do_notRLY([done:Done|Todo], X, NewState) :- % CHECK THE RULES
147
148                         % Find a executable process step
149                         member(X,Todo),
150
151                         % Check the prerequisites of the process step
152                         X::getDeps(Done, Dependencies),
153
154                         % RULE: All prerequisites must be done
155                         done(Done, Dependencies),
156
157                         % Create new (fake) State
158                         create_new_state([done:Done|Todo], X, NewState).
```

```
159
160
161 % Update the process counter in current state and return new state
162 create_new_state([done:Done|Todo], X, [done:NDone|LeftTodo]) :-
163                             subtract(Todo, [X], LeftTodo),
164                             sort([X|Done], NDone),!.
165
166
167 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
168 % Informational Methods
169 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
170 next(State, Next)   :-  setof(X, executable(State,X), Next).
171
172 somewhen([done:_|Todo], Todo).
173
174 not_after([done:Done|_], X, Never_Again)   :-  do_notRLY(State, X, [done:Never_Again|_]).
175
176
177 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
178 % GENERATE GRAPH FROM STARTSTATE
179 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
180
181 % ?- graph([a:0, b:1, c:0,  ...],  G).
182 graph(StartState, Graph) :- goDown([edge([], StartState, _)], Graph).
183
184 goDown([], []).
185 goDown(PGraph, GraphList) :- genStates(PGraph, L1), goDown(L1, TL), append(L1, TL, GraphList)
      .
186
187 genStates([], []).
188 genStates([edge(CState, RState, _) | T], StateSet) :-
189         % Do not iterate over identical ( reflexive ) edges
190         CState \= RState    ->( followingStates(RState, HeadList),
191                                 genStates(T, TailList),
192                                 append(HeadList, TailList, CombinedSet),
193                                 sort(CombinedSet, StateSet)
194                               );
195                             genStates(T, StateSet).
196
197 followingStates(State, FStates) :- findall( edge(State, NewState, [label=X, style='setlinewidth
      (1.5)'] ),
198                                         do_notRLY(State, X, NewState),
```

```
199                                          FStates ).
200
201 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
202 % EXPORT TO GRAPHVIZ
203 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
204
205 exportSM :- init([done:[]|IS]),
206              stopWatch::start,
207              graph([done:[]|IS], G),
208
209              length(G, CEdges), write('There are '), write(CEdges), write(' edges in the graph')
                      ,nl,
210
211              % Initial Node
212               list2string (IS, ISS),
213              string_concat('Done: \\n Todo: ', ISS, Init),
214              InitNode = node(Init, [label=Init, shape=box, color='#48ff00', style='rounded,
                      setlinewidth (2.0)']),
215
216              % Final Node
217              final_state([done:Done|_]),
218               list2string (Done, DoneS),
219              string_concat('Done: ', DoneS, F1),
220              string_concat(F1, '\\n Todo:', Final),
221              FinalNode = node(Final, [label=Final, shape=box, color='#0066ff', style='rounded,
                      setlinewidth (2.0)']),
222
223              flatten ([InitNode, G, FinalNode], Nodes),
224              Graph = graph(my_graph, Nodes),
225
226              stopWatch::stop,!,
227
228              %export to dot-file
229              datei(File),
230              string_concat(File, '.dot', DotFile),
231              graph_to_file(Graph, dot, DotFile),
232
233              % convert the .dot to PNG
234              string_concat(File, '.png', PngFile),
235              dot2file(DotFile, png, PngFile),
236
237              write('Please view the file "'), write(PngFile), writeln('" in an image viewer'),!.
```

```
238
239 benchmark  :-  writeln('Finds a way from Start to FinalState..'),
240                 stopWatch::start,
241                 init_state(IS), final_state(FS),
242                 dorec(IS, FS), nl,
243                 stopWatch::stop, nl.
244
245 dorec(FS,FS).
246 dorec(State, FS) :- do_notRLY(State,X,Newstate), dorec(Newstate,FS), X::name(N), write(N).
247
248
249 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
250 %  Initialization
251 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
252
253 init([done:[]|IS])  :-  init_state([done:[]|IS]), init_rec(IS), stopwatch::new.
254
255 init_rec([]).
256 init_rec([X|Init])  :-
257             proc(X, Comment, Min, Max),
258
259             require(X, RList), dyndeps(X, DList),
260             dependency::new(Dep, RList, [], DList),
261
262             (   (   proc(Min,_,_,_),proc(Max,_,_,_)) -> domain::new(Do, Min, Max, 3);
263                     proc(Min,_,_,_)                  -> domain::new(Do, Min, Max, 1);
264                                 proc(Max,_,_,_)      -> domain::new(Do, Min, Max, 2);
265                                                         domain::new(Do, Min, Max, 0)),
266
267             process::new(X, X, Comment, Do, Dep),!,
268
269 %           writeList([X, Do, Dep]), writeln('\tcreated'),      %Informational!
270
271             init_rec(Init).
272
273
274
275 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
276 % START STATE AND FINAL STATE
277 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
278 init_state(IS)  :-      findall(X, proc(X,_,_,_), Init), sort(Init, Inits), IS = [done:[]|Inits
      ].
```

```
279
280 final_state([done:Final|[]]) :- findall(X, proc(X,_,_,_), Fin), sort(Fin, Final).
281
282 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
283 % INSTANCE OF A PROCESS MODEL
284 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
285
286 %datei('simple_example').
287 %datei('boxes_states_complex').
288 %datei('just_dashed').
289 %datei('solid_test').
290 %datei('dashed_test').
291 %datei('herzflimmerablation').
292 %datei('ESProNa').
293 %datei('  kardinalitaetstest  ').
294 %datei('box_test').
295 %datei('lots_a_processes').
296 %datei('3_big_boxes').
297 %datei('  kartoffel1  ').
298 %datei('  kartoffel2  ').
299 %datei('zustandsraum').
300 datei('standard_beispiel').
301
302 :- datei(File), consult(File).
303
304 %:- exportSM.
```

Listing A.2: Die Prozessklasse

```
1  :- object(process, imports(attributes)).
2
3      :- private(count/1).
4      :- dynamic(count/1).
5
6      :- private(act/1).
7      :- dynamic(act/1).
8
9      :- private(done/1).
10     :- dynamic(done/1).
11
12     :- private(domain/1).
13     :- private(deps/1).
```

```
14
15      :- public(name/1).
16      :- public(desc/1).
17      :- public(counter/1).
18
19
20      :- public(in_domain/0).
21      :- public(dependent/0).
22
23      :- public(new/5).
24
25      :- public(start/0).
26      :- public(active/0).
27      :- public(executed/0).
28      :- public(stop/0).
29      :- public(cancel/0).
30
31      :- public(execute/1).
32      :- public(run/0).
33
34      :- public(getDeps/1).
35      :- public(getDeps/2).
36      :- public(getDomain/2).
37
38      :- public(print/0).
39
40
41  new(Process, Name, Desc, Domain, Deps) :-
42          self(Self),
43          create_object(Process, [extends(Self)], [],
44          [
45              name(Name), desc(Desc), domain(Domain), deps(Deps), count(0), done(false), act(
                    false)
46          ]).
47
48      counter(X) :- var(X) -> ::count(X);
49                    write('not a variable').
50
51      dependent :-
52          ::domain(D), D::dependent.
53
54      in_domain :-
```

```
55          :: domain(Domain), ::count(Count), !, Domain::valid(Count).
56
57      start  :- not(:: executed) -> ::retract(act(false)),  :: asserta(act(true)).
58
59      execute(N) :- :: active,  :: retract(count(Count)), NCount is Count+N, ::asserta(count(NCount
            )).
60
61      active:- :: act(true).
62
63      run    :- :: execute(1),  :: in_domain -> true; not(::execute(-1)).
64
65      cancel:- :: retract(act(true)),  :: retract(count(_)), :: asserta(count(0)),  :: asserta(act(
            false)).
66
67      stop   :- :: active,  :: in_domain ->
68                (:: retract(done(false)),  :: asserta(done(true)),  :: retract(act(true)),  :: asserta(act
                      (false)));
69                :: cancel  -> fail,  !.
70
71      executed :- :: done(true).
72
73      getDeps(Deps) :-
74          :: deps(DepObject), DepObject::getDependencies(Deps).
75
76      getDeps(Done, Deps) :-
77          :: deps(DepObject), DepObject::getDependencies(Done, Deps).
78
79      getDomain(Min, Max) :-
80          :: domain(Domain), Domain::getDomain(Min,Max).
81
82      print    :-  self(Self),      write('Reference:\t\t'),    write(Self),        nl,
83                   :: name(Name),   write('Name:\t\t\t'),       write(Name),        nl,
84                   :: desc(Desc),   write('Description:\t\t'),  write(Desc),        nl,
85                   :: domain(Do),   write('Domain:\t\t\t'),     write(Do),          nl,
86                                    write('\tDetails:\t'),      Do:: print,         nl,
87                   :: deps(Deps),   write('Dependency:\t\t'),   write(Deps),        nl,
88                                    write('\tDetails:\t'),      Deps:: print,       nl,
89                   :: counter(C),
90                   :: act(A),
91                   :: done(E),                                  nl,
92                                    write('Counter:\t\t'),      write(C),           nl,
93                                    write('active\t\t->\t'),    write(A),           nl,
```

```
94                                  write('executed\t->\t'),    write(E),           nl.
95
96
97 :- end_object.
```

Listing A.3: Die Abhängigkeitsklasse

```
1  :- object(dependency, imports(attributes)).
2
3  %   :- info([
4  %       version is 1.0,
5  %       author is 'Christoph Guenther',
6  %       date is '2009/1/15',
7  %       comment is 'The Dependencies of a process.']).
8
9
10     :- public(rec_app/3).
11     :- mode(rec_app(?list, ?list, ?list), zero_or_one).
12     :- info(rec_app/3, [
13         comment is 'Appends Lists in a List recusrively.',
14         argnames is ['[L1]','[L2]','[L3]']]).
15
16     :- private(recprint/1).
17     :- mode(recprint(?list), zero_or_one).
18     :- info(recprint/1, [
19         comment is 'Recursively Prints Elements in a List.',
20         argnames is ['[EL1, ...]']]).
21
22     :- private(member/2).
23     :- mode(member(?atom, ?list), zero_or_one).
24     :- info(member/2, [
25         comment is 'Checks if X is meber of the List.',
26         argnames is ['X','[List]']]).
27
28     :- private(flatten/2).
29     :- mode(flatten(?list, ?list), zero_or_one).
30     :- info(flatten/2, [
31         comment is 'Flattens a List.',
32         argnames is ['[[List]]','[List]']]).
33
34     :- private(flatten/3).
35     :- mode(flatten(?list, ?list, ?list), zero_or_one).
```

```
36      :- info(flatten/3, [
37          comment is 'flattens the List recursively.',
38          argnames is ['[List]', '[List]' ,'[List]']]).
39
40      :- public(rec_getDep/2).
41      :- mode(rec_getDep(?list, ?list), zero_or_one).
42      :- info(rec_getDep/2, [
43          comment is 'Recursively get the dynamic dependencies.',
44          argnames is ['[[ProcessList]]','[DependencyList]']]).
45
46      :- public(processes/1).
47      :- mode(processes(?list), zero_or_one).
48      :- info(processes/1, [
49          comment is 'List of required processes.',
50          argnames is ['[Processobject, ...]']]).
51
52      :- public(data/1).
53      :- mode(data(?list), zero_or_one).
54      :- info(data/1, [
55          comment is 'List of required data.',
56          argnames is ['[Dataobject, ...]']]).
57
58      :- public(dyndep/2).
59      :- dynamic(dyndep/2).
60      :- mode(dyndep(?list, ?list), zero_or_one).
61      :- info(dyndep/2, [
62          comment is 'List of dynamic dependencies.',
63          argnames is ['[Trigger, List, ..]', '[Dependence, Objects, ..]']]).
64
65      :- private(initdyn/1).
66      :- mode(initdyn(?list), zero_or_one).
67      :- info(initdyn/1, [
68          comment is 'Initalizes the dynamic Dependencies.',
69          argnames is ['[[Triggers]:[Deps], [..]:[..] ...]']]).
70
71      :- public(print/0).
72      :- info(print/0, [
73          comment is 'Prints the Dependencies.']).
74
75      :- private(recprint/0).
76      :- info(recprint/0, [
77          comment is 'Prints a List recursively.']).
```

```
78
79      :- public(getDependencies/1).
80      :- mode(getDependencies(?list), one).
81      :- info(getDependencies/1, [
82          comment is 'Get the dependencies of the process step.',
83          argnames is ['DependencyList']]).
84
85      :- public(getDependencies/2).
86      :- mode(getDependencies(?list, ?list), one).
87      :- info(getDependencies/2, [
88          comment is 'Get all dependencies of the process step.',
89          argnames is ['State', 'dynamic DependencyList']]).
90
91
92      :- public(new/2).
93      :- mode(new(-object_identifier, ?list), one).
94      :- info(new/2, [
95          comment is 'Creates a new Dependency.',
96          argnames is ['Id', 'ProcessList']]).
97
98      :- public(new/3).
99      :- mode(new(-object_identifier, ?list, ?list), one).
100     :- info(new/3, [
101         comment is 'Creates a new Dependency.',
102         argnames is ['Id', 'ProcessList', 'Datalist']]).
103
104     :- public(new/4).
105     :- mode(new(-object_identifier, ?list, ?list, ?list), one).
106     :- info(new/4, [
107         comment is 'Creates a new Dependency.',
108         argnames is ['Id', 'ProcessList', 'Datalist', 'DynList']]).
109
110     new(Dependency, DepList) :-
111         self(Self),
112         create_object(Dependency, [extends(Self)], [], [processes(DepList), data([])]).
113
114     new(Dependency, DepList, DataList) :-
115         self(Self),
116         create_object(Dependency, [extends(Self)], [], [processes(DepList), data(DataList)]).
117
118     new(Dependency, DepList, DataList, DynList) :-
119         self(Self),
```

```
120         create_object(Dependency, [extends(Self)], [], [processes(DepList), data(DataList)]),
121         Dependency::initdyn(DynList).
122
123     initdyn ([]) .
124     initdyn ([ Trigger:Dep|DynList]) :-
125         :: asserta (dyndep(Trigger, Dep)),
126         :: initdyn (DynList).
127
128     getDependencies(Deps) :-
129         :: processes (Procs), ::data(Data), rec_app(Procs,Data,Dependent), flatten(Dependent,
                Deps).
130
131     getDependencies(Done, Deps) :-
132         :: processes (Procs), ::data(Data),
133         ::rec_app(Procs,Data,Static),
134         ::rec_getDep(Done, Dynamic),
135         ::rec_app(Static, Dynamic, Dependent),
136         :: flatten (Dependent, Deps).
137
138     print :-
139         :: processes (DepList), write('Processes:\t['), recprint (DepList), write(']'),
140         ::data(DataList) ->    nl, write('Data:\t['), recprint (DataList), write(']');
141                                true.
142
143     rec_getDep ([],[]) .
144     rec_getDep([X|Tail], Dynamic) :-
145         :: setof (DList, (dyndep(Trigger, DList), member(X, Trigger)), Dyna)
146         -> (    :: flatten (Dyna, Dynf),
147                 ::rec_getDep(Tail, Dyn),
148                 ::rec_app(Dynf, Dyn, Dynamic));
149             rec_getDep(Tail, Dynamic).
150
151     rec_app([],L,L).
152     rec_app([H|T],L2,[H|L3]) :- rec_app(T,L2,L3).
153
154     member(X,[X|_]).
155     member(X,[_|T]) :- member(X,T).
156
157     flatten (Xs,Ys) :-    flatten (Xs ,[], Ys).
158
159     flatten ([X|Xs],As,Ys)    :-   flatten (Xs,As,As1),  flatten (X,As1,Ys).
160     flatten (X,As,[X|As])     :-   integer(X).
```

```
161     flatten (X,As,[X|As])     :-  atom(X), X\=[].
162     flatten  ([], Ys, Ys).
163
164
165     recprint  ([]) .
166     recprint ([H])          :- write(H).
167     recprint ([H,Tail])   :-
168         write(H),  write(', '),  :: recprint ( Tail ).
169
170 :- end_object.
```

Listing A.4: Die Domänenklasse

```
1
2  :- object(domain, imports( attributes )).
3
4  %    :- info ([
5  %         version  is  1.0,
6  %         author  is  'Christoph  Guenther',
7  %         date  is  '2009/1/15',
8  %         comment is 'The Domain used by a process.']) .
9
10
11     :- private (min/1).
12     :- mode(min(?atom), zero_or_one).
13     :- info(min/1, [
14         comment is 'Smallest  valid  value.',
15         argnames is  ['Min']]) .
16
17     :- private (max/1).
18     :- mode(max(?atom), zero_or_one).
19     :- info(max/1, [
20         comment is 'Biggest  valid  value.',
21         argnames is  ['Max']]) .
22
23     :- private (dyn/1).
24     :- mode(dyn(?atom), one).
25     :- info(dyn/1, [
26         comment is 'Is  the  Domain dependent?',
27         argnames is  ['true/false']]) .
28
29     :- public( valid /1).
```

```
30      :- mode(valid(?atom), zero_or_one).
31      :- info(valid/1, [
32          comment is 'Binary information if the Process is in its domain.',
33          argnames is ['true/false']]).
34
35      :- public(print/0).
36      :- info(print/0, [
37          comment is 'Prints the Domain.']).
38
39      :- public(dependent/0).
40      :- info(dependent/0, [
41          comment is 'Is the Domain dependent?']).
42
43      :- public(getDomain/2).
44      :- mode(getDomain(?atom, ?atom), one).
45      :- info(getDomain/2, [
46          comment is 'Gets the Domain.',
47          argnames is ['Min', 'Max']]).
48
49      :- public(new/3).
50      :- mode(new(-object_identifier, +atom, +atom), one).
51      :- info(new/3, [
52          comment is 'Creates a new Domain.',
53          argnames is ['Id', 'Min', 'Max']]).
54
55      :- public(new/4).
56      :- mode(new(-object_identifier, +atom, +atom, +atom), one).
57      :- info(new/4, [
58          comment is 'Creates a new Domain.',
59          argnames is ['Id', 'Min', 'Max', 'Dynamic']]).
60
61      new(Domain, Min, Max) :-
62          new(Domain, Min, Max, 0).
63
64      new(Domain, Min, Max, Dyn) :-
65          self(Self),
66          create_object(Domain, [extends(Self)], [], [min(Min), max(Max), dyn(Dyn)]).
67
68
69      dependent :-
70          ::dyn(X),!, X>0.
71
```

```
72      getDomain(Min, Max) :-
73          ::min(Mi), ::max(Ma), ::dyn(X),
74          (number(Mi) -> Min = Mi;
75                      (    (X = 1; X = 3) -> (Mi::executed    -> Mi::counter(Min);
76                                                              Min = Mi, writeln('Caution:
                                                                    Dependent Domain
                                                                    insufficiantly  initialized .'));
77                                          fail )),
78          (number(Ma) -> Max = Ma;
79                      (    (X = 2; X = 3) -> (Ma::executed    -> Ma::counter(Max);
80                                                              Max = Ma, writeln('Caution:
                                                                    Dependent Domain
                                                                    insufficiantly  initialized .'));
81                                          Max = * )).
82
83      valid(Count) :-
84          ::getDomain(Min,Max), !,
85          (    number(Min), number(Count) )                   -> (    Count >= Min, (    number
                (Max) -> Count =< Max; true));
86              (    not(number(Count)), not(number(Max)))      -> (    Max = Count ).
87
88
89      print :-
90          ::min(Min), ::max(Max), write(Min), write(' ... '), write(Max), (::dependent -> write(' (
                dependent)'); true).
91
92  :- end_object.
```

Listing A.5: Die Attributenklasse

```
1   :- category( attributes ).
2
3       :- public(set_attribute/2).
4       :- mode(set_attribute(+nonvar, +nonvar), one).
5
6       :- public(get_attribute/2).
7       :- mode(get_attribute(?nonvar, ?nonvar), zero_or_more).
8
9       :- public(del_attribute/2).
10      :- mode(del_attribute(?nonvar, ?nonvar), zero_or_more).
11
12      :- public(del_attributes/2).
```

```
13      :- mode(del_attributes(@term, @term), one).
14
15      :- private(attribute_/2).
16      :- mode(attribute_(?nonvar, ?nonvar), zero_or_more).
17      :- dynamic(attribute_/2).
18
19      set_attribute(Attribute, Value):-
20          :: retractall(attribute_(Attribute, _)),
21          :: assertz(attribute_(Attribute, Value)).
22
23      get_attribute(Attribute, Value):-
24          :: attribute_(Attribute, Value).
25
26      del_attribute(Attribute, Value):-
27          :: retract(attribute_(Attribute, Value)).
28
29      del_attributes(Attribute, Value):-
30          :: retractall(attribute_(Attribute, Value)).
31
32  :- end_category.
```

Listing A.6: Die Werkzeugklasse

```
1   :- object(stopwatch).
2
3       :- private(sysstamp/1).
4       :- mode(sysstamp(?atom), one).
5       :- dynamic(sysstamp/1).
6
7       :- public(start/0).
8
9       :- public(stop/0).
10
11      :- public(stop/1).
12
13      :- public(new/0).
14
15      :- public(new/1).
16      :- mode(new(-object_identifier), one).
17
18
19      new :-
```

```
20          new(stopWatch).
21
22      new(SW) :-
23          self(Self),
24          create_object(SW, [extends(Self)], [], []).
25
26      start :-
27          {get_time(SysTime)}, ::asserta(sysstamp(SysTime)).
28
29      stop :-
30          {get_time(SysTime2)},
31          ::retract(sysstamp(SysTime1)),
32          SysTime is SysTime2 - SysTime1,
33          write(SysTime), writeln('seconds needed to work that out.').
34
35      stop(SysTime) :-
36          {get_time(SysTime2)},
37          ::retract(sysstamp(SysTime1)),
38          SysTime is SysTime2 - SysTime1.
39
40  :- end_object.
```

Listing A.7: Der Grafikexport (IGLER [2008])

```
1   :- module(graphviz,
2             [
3              graph_to_dot/2,
4              graph_to_file/3,
5              display_graph/2,
6              dot2file/3,
7              list2string/2
8             ]).
9
10  % graph_to_dot(+GraphTerm,?DotProgram)
11  % converts a graph representation to a dot string
12  %   @param GraphTerm
13  %   graph(Name,Terms)
14  %   Term = node(ID,Props) | edge(ID1,ID2,Props)
15  graph_to_dot(graph(Name,GraphTerms),Dot):-
16          findall(Part,
17                  (member(Term,GraphTerms),graph_to_dot(Term,Part)),
18                  Parts),
```

```
19          concat_atom(Parts,DotLines),
20          sformat(Dot,
21                  'digraph ~w {~n~w}~n',
22                  [Name,DotLines]).
23
24  graph_to_dot(Type=Val,Dot):-
25          sformat(Dot,'~w="~w"',[Type,Val]).
26
27  graph_to_dot(prop(Prop),Dot):-
28          graph_to_dot(Prop,Inner),
29          sformat(Dot,'~w;~n',[Inner]) .
30
31  graph_to_dot(default(Type,Props),Dot):-
32           findall (PropAsDot,
33                   (    member(Prop,Props),
34                        graph_to_dot(Prop,PropAsDot)),
35                   PropAsDotL),
36          concat_atom(PropAsDotL,', ',Inner),
37          sformat(Dot,
38                  '          ~w [~w];~n',
39                  [Type,Inner]) .
40
41  graph_to_dot(node(ID,Props),Dot):-
42          %write(ID), write(Props),
43          escape_id(ID,IDe), %write(ID), nl, write(IDe), nl, nl,
44           findall (PropAsDot,
45                   (    member(Prop,Props),
46                        graph_to_dot(Prop,PropAsDot)),
47                   PropAsDotL),
48          concat_atom(PropAsDotL,', ',Inner),
49          sformat(Dot,
50                  '          ~w [~w];~n',
51                  [IDe,Inner]) .
52
53  graph_to_dot(nodeset(ID,Label,Nodes),Dot):-
54          escape_id(ID,IDe), %write(ID), nl, write(IDe), nl, nl,
55           findall (Atom,
56                   (    member(Node,Nodes),
57                        graph_to_dot(Node,Atom)),
58                   Atoms),
59          concat_atom(Atoms,' ',NodeSetDot),
60          sformat(Dot,
```

```
61                  '       subgraph cluster_~w {~n    label="~w"~n~w~n }~n',
62                  [IDe,Label,NodeSetDot]).
63
64  graph_to_dot(edge(ID1,ID2,Props),Dot):-
65          escape_id(ID1, ID1e),
66          escape_id(ID2, ID2e),
67           findall (PropAsDot,
68                  (    member(Prop,Props),
69                       graph_to_dot(Prop,PropAsDot)),
70                  PropAsDotL),
71          concat_atom(PropAsDotL,', ',Inner),
72          sformat(Dot, ' ~w -> ~w [~w];~n', [ID1e,ID2e,Inner]).
73
74   list2string ([], _, _).
75   list2string ([H], RL) :-  swritef(RL, '%w', [H]).
76   list2string ([H|T], RL) :- swritef(HList, '%w', [H]),  list2string (T, TList), string_concat(HList
       , ', ', HListC), string_concat(HListC, TList, RL).
77
78
79  %escape_id(ID, IDe) :- list2string(ID, ID2), string_concat ('"[', ID2, ID3), string_concat(ID3,
       ']"', IDe).
80  escape_id([done:Done|Todo], IDe)    :-
81      length(Done, 0) -> Second = '"Done: \\n Todo: ';
82          (  list2string (Done, DoneString),
83            string_concat('"Done: ', DoneString, First),
84            string_concat(First , '\\n Todo: ', Second)),
85            ( length(Todo, 0) -> Third = Second;
86                                 (    list2string (Todo, TodoString),
87                                      string_concat(Second, TodoString, Third))),
88      string_concat(Third, '"', IDe).
89
90  %% graph_to_file(+GraphTerm,?Fmt,?File)
91  %   writes a graph to an img file  using 'dot'
92  %
93  % @param File
94  %   if unground, a temp file  will  be chosed
95  % @param Fmt
96  %   if unground, a dot format  will  be chosen
97  graph_to_file(GraphTerm,Fmt,File):-
98          (nonvar(Fmt) -> true ; Fmt=dot),
99          (nonvar(File) -> true ; tmp_file(Fmt,File)),
100     graph_to_dot(GraphTerm,Dot),
```

```
101         (   Fmt=dot
102         -> DotFile=File
103         ;   tmp_file(dot,DotFile)),
104          tell(DotFile),
105         write(Dot),
106         told,
107         (   Fmt=dot
108         -> true
109         ;   concat_atom([dot,'-o',File,'-T',Fmt,DotFile],' ',Cmd),
110              shell(Cmd)).
111
112 dot2file(Dot, Fmt, File)    :-
113         concat_atom([dot,'-o',File,'-T',Fmt,Dot],' ',Cmd),
114          shell(Cmd).
115
116 %% display_graph(+GraphTerm,+Cmd)
117 %   displays a graph using Cmd
118 display_graph(GraphTerm,DisplayCmd):-
119         display_graph(GraphTerm,_,DisplayCmd).
120 display_graph(GraphTerm,Fmt,DisplayCmd):-
121         graph_to_file(GraphTerm,Fmt,File),
122         concat_atom([DisplayCmd,File],' ',FullCmd),
123          shell(FullCmd).
```

Listing A.8: häufig verwendete Funktionen (IGLER [2008])

```
1  % Library of frequently used predicates
2
3  :- op( 900, fy, not).
4
5  combine(E, [H], [E:H]) :- !.
6  combine(E, [H | T], Comb) :- combine(E, T, L1), conc([E:H], L1, Comb).
7
8  % N-ary combination (12 <=> 21)
9  comb(0,_ , []).
10 comb(N, [H | T], [H | Comb]) :- N > 0, N1 is N-1, comb(N1, T, Comb).
11 comb(N, [_ | T], Comb) :- N > 0, comb(N, T, Comb).
12
13  alldifferent ([]) .
14  alldifferent ([H | T]) :- outof(H, T),  alldifferent (T).
15
16 outof(_, []) .
```

```
17 outof(X:CX, [H:_ | T]) :- X \= H, outof(X:CX, T).
18
19 % conc(L1,L2,L3): list L3 is the concatenation of lists L1 and L2
20 conc( [], L, L).
21 conc( [X | L1], L2, [X | L3])  :-  conc( L1, L2, L3).
22
23 % del(X,L0,L): List L is equal to list L0 with X deleted
24 %  Note: Only one occurrence of X is deleted
25 del( X, [X | Rest], Rest).          % Delete the head
26 del( X, [Y | Rest0], [Y | Rest]):-  del( X, Rest0, Rest).
27
28 % set_difference( Set1, Set2, Set3): Set3 is the list representing
29 % the difference of sets represented by lists Set1 and Set2
30 set_difference ( [] , _ , []) .
31 set_difference ( [X | S1], S2, S3)       :- member( X, S2), !, set_difference ( S1, S2, S3).
32 set_difference ( [X | S1], S2, [X | S3]) :- set_difference ( S1, S2, S3).
33
34 % max( X, Y, Max): Max = max(X,Y)
35 max( X, Y, Max) :-
36   X >= Y, !, Max = X;
37   Max = Y.
38
39 % min( X, Y, Min): Min = min(X,Y)
40 min( X, Y, Min) :-
41   X =< Y, !, Min = X;
42   Min = Y.
43
44
45 xor(A,B) :- not(A), B.
46 xor(A,B) :- A, not(B).
47
48 % Potenzmenge einer Liste
49 powerset ([],[[]]) .
50 powerset([E|R],PM):-powerset(R,PMohneE),
51                      findall ([E|T],member(T,PMohneE),PMmitE),
52                      append(PMohneE,PMmitE,PM).
53
54 writeList ([]) .
55 writeList ([H])       :- write(H).
56 writeList ([H|Tail]) :- write(H), write(',\t'), writeList ( Tail ).
```

Listing A.9: Grafikexport (IGLER [2008])

```
%%% Plotting terms as trees using Graphviz %%%

term_list_linear(false).     % change to true for plotting lists linearly

term(Term):-
    gv_start('D:\Data\Studium\DA\Graphs\term.dot'),
    Term =.. [Functor|Subterms],
    gv_root(Functor,0),
    term_list(Subterms,0),
    gv_stop.

term(Term,N):-
    var(Term),!,
    gv_node(N,Term,_).
term(Term,N):-
    term_list_linear(true),
    list(Term),!,
    gv_node(N,Term,N1),
    term_list(Term,N1).
term([],N):-!,
    gv_node(N,'$empty_list',_).
term(Term,N):-
    Term =.. [Functor|Subterms],
    gv_node(N,Functor,N1),
    term_list(Subterms,N1).

term_list([],_).
term_list([Term|Terms],N):-
    term(Term,N),
    term_list(Terms,N).

%%% Meta-interpreter plotting (part of) the SLD-tree using Graphviz %%%

:-op(1100,fx,sld).   % can write ?-sld Goal instead of ?-sld(Goal)

sld(Goal):-
    sld(Goal,5).     % default depth bound

sld(Goal,D):-
    gv_start('D:\Data\Studium\DA\Graphs\sld.dot'),
```

```
41      gv_root((?-Goal),0),
42      prove_d(Goal,Goal,0,D),
43       fail .    % failure-driven loop to get  all  solutions
44  sld(_,_):-
45      gv_stop.
46
47  % meta-interpreter with complete  resolvent  and depth bound
48  prove_d(true,Goal,N,_):-!,
49      gv_answer(N,Goal).
50  prove_d((A,B),Goal,N,D):-!,
51      D>0, D1 is D-1,
52       resolve (A,C),
53      conj_append(C,B,E),
54      gv_node(N,(:-E),N1),
55      prove_d(E,Goal,N1,D1).
56  prove_d(A,Goal,N,D):-
57      D>0, D1 is D-1,
58       resolve (A,B),
59      gv_node(N,(:-B),N1),
60      prove_d(B,Goal,N1,D1).
61
62   resolve (A,true):-
63      predicate_property(A,built_in) ,!,
64       call (A).
65   resolve (A,B):-
66      clause(A,B).
67
68  %%% Utilities %%%
69   list  ([]) .
70   list  ([_|T]):-list(T).
71
72  conj_element(X,X):- % single-element conjunction
73      X \= true,
74      X \= (_,_).
75  conj_element(X,(X,_)).
76  conj_element(X,(_,Ys)):-
77      conj_element(X,Ys).
78
79  conj_append(true,Ys,Ys).
80  conj_append(X,Ys,(X,Ys)):- % single-element conjunction
81      X \= true,
82      X \= (_,_).
```

```
83  conj_append((X,Xs),Ys,(X,Zs)):-
84      conj_append(Xs,Ys,Zs).
85
86  writes ([]) :-!,nl.
87  writes([H|T]):-!,writes(H),writes(T).
88  writes((A,B)):-!,writes(A),write(',\\n'),writes(B). % break up conjunctions
89
90  writes(:-(A)) :- !,write((:-)),writes(A).
91  %writes(:-A) :- !,write (':-') ,writes(A).
92
93  writes(?-(A)) :- !,write((?-)),writes(A).
94  %writes(?-A) :- !,write('?-') ,writes(A).
95
96  writes('$empty_list'):-!,write ([]) .
97  writes(A):-write(A).      % catch-all
98
99  %%% Graphviz utilities %%%
100
101 gv_max_id(1000).   % max number of nodes in the graph
102
103 % open file and start new graph
104 gv_start(FileName):-
105     tell (FileName),
106     writes (['digraph {']),
107     %writes(['graph [size ="4,6"];']),
108     writes (['node [shape=plaintext, fontname=Courier, fontsize=10]']).
109
110 % next graph
111 gv_next:-
112     writes (['}']),
113     writes (['digraph {']),
114     writes (['node [shape=plaintext, fontname=Courier, fontsize=10]']).
115
116 % finish graph and close file
117 gv_stop:-
118     writes (['}']),
119     told.
120
121 % start new subgraph
122 gv_cluster_start:-
123     ( retract ('$gv_cluster'(N)) -> N1 is N+1
124     ; otherwise -> N1=0
```

```
125     ), assert ('$gv_cluster'(N1)),
126     writes (['subgraph cluster_',N1,' {']),
127     writes (['[style=filled, color=lightgrey];']),
128     writes (['node [style=filled,color=white];']).
129
130 % finish subgraph
131 gv_cluster_stop:-
132     writes (['}']).
133
134 % write the root of a tree and initialise node IDs
135 gv_root(L,N):-
136     writes ([N,' [label="',L,'"];']),
137     gv_init_ids(N).
138
139 % add a node with label L and parent N0
140 gv_node(N0,L,N):-
141     gv_id(N),
142     writes ([N,' [label="',L,'"];']),
143     writes ([N0,' -> ',N,';']).
144
145 % add a specially formatted leaf
146 gv_answer(N0,L):-
147     gv_id(N),
148     writes ([N,' [label="Answer:\\n',L,'", shape=ellipse, style=dotted, fontsize=8];']),
149     writes ([N0,' -> ',N,' [style=dotted, arrowhead=none];']).
150     %writes(['{rank=same;',N0,';', N ,';}']) .
151
152 % generate a new node ID
153 gv_id(N):-
154     retract ('$gv_id'(N0)),
155     gv_max_id(M),
156     N0 < M, % don't generate infinite graphs
157     N is N0+1,
158     assert ('$gv_id'(N)).
159
160 % initialise node IDs, next free ID is N+1
161 gv_init_ids(N) :-
162     retractall ('$gv_id'(_)),
163     assert ('$gv_id'(N)).
```