Modellierung und Verifikation medizinischer Leitlinien

Dissertation zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Angewandte Informatik
der Universität Augsburg
im Jahr 2008 von

Jonathan Schmitt

Universität Augsburg

Amtierender Dekan:	Prof. Dr. Wolfgang Reif
Gutachter:	Prof. Dr. Wolfgang Reif
	Prof. Dr. Bernhard Bauer
Tag der Prüfung:	25. November 2008
Prüfer:	Prof. Dr. Wolfgang Reif
	Prof. Dr. Bernhard Bauer
	Prof. Dr. Alexander Knapp

Zusammenfassung

Das Gesundheitssystem befindet sich derzeit im Wandel. Medizin wird zunehmend nicht mehr als Kunst betrachtet, bei der ein Arzt sich auf seine Intuition verlässt, wenn er eine Diagnose erstellt oder eine Behandlung auswählt. Die steigenden Kosten zwingen die Beteiligten dazu, die Effizienz der Behandlungen zu steigern. Ein Mittel dies zu erreichen sind medizinische Leitlinien. Dabei handelt es sich um Anleitungen, die den „Goldstandard“ der Behandlung einer Krankheit zusammenfassen und so dem Arzt Wissen kompakt zur Verfügung stellen. Basierend auf diesem Wissen sollen Ärzte Behandlungen und Diagnosen durchführen. Die Erwartung ist, dass durch den Einsatz von Leitlinien in einer Behandlung die Qualität der Behandlung steigt während gleichzeitig die Behandlungskosten sinken.

Mit zunehmender Verbreitung gewinnen medizinische Leitlinien immer mehr an Bedeutung. Damit stellt sich die Frage nach der Qualitätssicherung von Leitlinien. Bisher agieren die Organisationen, die Leitlinien erstellen, wie etwa die niederländische CBO, auf Basis der Annahme, dass eine Leitlinie keine nennenswerten Fehler mehr enthält, wenn sie von Experten erstellt und korrekturgelesen wurde. Dabei bietet die Existenz einer Leitlinie alleine keine Garantie dafür, dass Behandlungen in der richtigen Reihenfolge ausgeführt werden, dass veraltete Behandlungen unterlassen werden oder stets die günstigste Form der Behandlung gewählt wird. In Anbetracht der Tatsache, dass tausende Patienten anhand von einer Leitlinie behandelt werden und ein Fehler in einer Leitlinie sehr hohe Kosten und sehr großes Leid verursachen kann, scheint die Annahme der Korrektheit fahrlässig.

In der Softwaretechnik werden formale Methoden auf ähnliche Probleme angewendet. Dabei gibt der Einsatz formaler Methoden die Garantie, dass eine Modell keine Fehler mehr enthält. Diese Arbeit bringt medizinische Leitlinien und formale Methoden zusammen, und zeigt auf diese Weise, dass eine Leitlinie fehlerfrei ist. Damit ist beispielsweise gemeint, dass vorgeschriebene Behandlungssequenzen eingehalten werden oder veraltete Behandlungen nicht von der Leitlinie vorgeschrieben werden. Um dies verifizieren zu können, wird ein Modell der Leitlinie erstellt, welches mit einer formalen Semantik unterlegt ist. Dieses Modell wird dann als Programm interpretiert, wobei der Arzt als „Ausführungseinheit“ agiert und mit Diagnosen und Behandlungen den Zustand des Patienten abfragt und ändert. Mit dieser Interpretation werden Methoden zur Qualitätssicherung von Programmen auf Leitlinien angewendet. Das Ergebnis dieses Ansatzes sind Leitlinien, die garantiert fehlerfrei sind. Gleichzeitig sind die auf diese Weise qualitätsgesicherten Leitlinien strukturell nicht von den bisherigen zu unterscheiden, wodurch vermieden wird, dass das medizinische Personal sich beim Einsatz gesicherter Leitlinien gegenüber dem bisherigen Vorgehen umstellen muss.

Danksagung

Die vorliegende Arbeit wäre nie entstanden, hätten mich nicht zahlreiche Menschen bei der Erstellung unterstützt und gefördert.

Im Besonderen gilt mein Dank Prof. Dr. Wolfgang Reif dafür, dass er mir die Möglichkeit gegeben hat, diese Arbeit anzufertigen. Ich danke ihm für sein Vertrauen in mich und meine Fähigkeiten, für seine Unterstützung und Förderung, seinen Rat sowie seine fachlichen Anmerkungen.

Dr. Michael Balser gilt mein Dank für seine kompetenten Ratschläge und seine herausragenden Forschungsergebnisse, auf denen ich aufsetzen durfte. Außerdem danke ich Michael für die Schubser in die richtige Richtung, die er mir in seiner immer ruhigen und geduldigen Art gegeben hat.

Mit Simon Bäumler konnte ich jederzeit über temporallogische Details und speziell Fragen zur Modularisierung und Beweisdekomposition diskutieren. Außerdem hat Simon die Diplomarbeit von Andriy Dunets betreut, die die Anwendungen der Modellprüfungstechniken auf Asbru erforscht.

Ich danke Alwin Hoffmann für seine jahrelange Unterstützung, sowohl als studentische Hilfskraft bei der Umsetzung der prototypischen Implementierung der Semantik als auch bei seiner Ausarbeitung des Gegenbeispiel-Mechanismus im Rahmen seiner Diplomarbeit.

Dr. Gerhard Schellhorn war der Anker im Sturm der Zeit, stets bereit und willens in langen Diskussionen seltsame Probleme unkonventionell zu lösen und den formalen Teil der Arbeit zu lesen.

Nina Moebius danke ich für ihre liebevolle Unterstützung, beim Korrekturlesen der Arbeit und auch beim Überbrücken der tiefen mentalen Täler, die ich auf dem Weg durchschritten habe.

Dr. Holger Grandy hat mir mit seinen Anmerkungen, Diskussionen und Korrekturanmerkungen kurz vor dem Abschluss dieser Arbeit sehr geholfen.

Ich danke weiterhin dem gesamten Protocure II Team, zum Einen für die fachliche Inspiration und Hilfe, zum Anderen für die schöne Zeit, die wir zusammen hatten. Speziell danken möchte ich hier Dr. Mar Marcos und Begoña Martinez, die immer geduldig für Rückfragen bereitstanden und auch das Experiment unternommen haben, die interaktive Beweistechnik selbst einzusetzen. Außerdem gilt mein Dank Andreas Seyfang für unsere stunden- bzw. seitenlangen Diskussionen per Mail und auf den Protocure Treffen, um die widerspenstige Semantik von Asbru zu zähmen.

Inhaltsverzeichnis

I Einführung, Grundlagen und Fallstudie **1**

1 Gliederung dieser Arbeit **3**

2 Motivation und Einordnung in den Leitlinien-Entwicklungsprozess **5**
2.1 Leitlinien zur Verbesserung der Medizin 5
2.2 Qualität der Leitlinien 6
2.3 Qualitätssicherung bei Leitlinien 7
2.4 Abstrakte Sicht auf die Medizin 9
2.5 Medizinische Aussagen als Beweisproblem 10
2.6 Ergebnisse dieser Arbeit 11
2.7 Einbettung dieser Arbeit in den Rahmen von Protocure II 12
2.8 Verwandte Arbeiten 12

3 Einführung in Asbru **15**
3.1 Asbru – an einem Beispiel 15
3.2 Asbru Details 20
3.3 Asbru und Asbru Light 21
3.4 Begriffswelt 21
3.5 Formulierung von Asbru Aussagen mit GDL 22

4 Grundlagen **25**
4.1 Interaktive Verifikation mit KIV 25
4.2 Die Temporallogik ITL+ 26
4.3 Verifikation temporallogischer Aussagen in KIV 34

5 Die Brustkrebs-Fallstudie **39**
5.1 Graphische Darstellung der Planhierarchien 39
5.2 DCIS und operabeler, invasiver Brustkrebs 40
5.3 Behandlungsbegleitende systematische Therapie 42
5.4 Lokal fortgeschrittener Brustkrebs 43
5.5 Nachsorge-Untersuchungen 44
5.6 Lokal wiederkehrender Brustkrebs 45
5.7 Metastasierender Brustkrebs 45

II Syntax und Semantik von Asbru **53**

6 Syntax von Asbru **55**
6.1 Ein einfaches Beispiel 55
6.2 Asbru-Plan Datenstruktur (**asbru-plan**) 55
6.3 Die Plantypen Datenstruktur (**plan-type**) 58
6.4 Abstrakte Zeit-Annotationen (**abstract-time-annotation**) 61

6.5 Unterplanlisten (**plan-list**) . . . 62
6.6 Plannamen (**plan-name**) . . . 63
6.7 Asbru-Bedingungen (**asbru-condition**) . . . 63
6.8 Bedingungen (**condition**) . . . 64
6.9 Konditional (**conditional**) . . . 64
6.10 Der Zahlentyp int-bottom (**int-bottom**) . . . 65
6.11 Abstrakte Referenzpunkte (**abstract-asbru-clock**) . . . 65
6.12 Planinstanznamen (**instance-name**) . . . 66
6.13 Planzustände (**plan-state**) . . . 67
6.14 Wartebedingungen (**wait-for**) . . . 68

7 Semantik des Zustands von Asbru Systemen 71
7.1 Zeitliche Aspekte in Asbru . . . 72
7.2 Umgebungssignale . . . 74
7.3 Hierarchiezustand . . . 77
7.4 Asbru-Zustands-Historie . . . 79
7.5 Krankenakte . . . 83
7.6 Zustand von Asbru . . . 84

8 Semantik von Bedingungen in Asbru 87
8.1 Auswertung von Asbru-Bedingungen . . . 87
8.2 Auswertung von Wartebedingungen . . . 105

9 Semantik des Asbru-Zustandsübergangssystems 111
9.1 Syntax des Asbru-Zustandsübergangssystems (**asbru-system**) . . . 111
9.2 Zustandsübergänge für einzelne Pläne . . . 111
9.3 Zustandsübergänge für Planhierarchien . . . 138

10 Ablaufsemantik von Asbru 141
10.1 Der Ablaufbegriff . . . 141
10.2 Modellierung durch Asbru-Systeme . . . 142
10.3 Mikro- und Makro-Schritt . . . 142
10.4 Umgebungsübergang . . . 142
10.5 Initiale Zustände . . . 144
10.6 Geschlossene Asbru-Abläufe . . . 145

11 Validierung der Asbru-Semantik 147
11.1 Validierung von Zeit-Annotationen . . . 147
11.2 Modellwechsel als Methode zur Validierung . . . 150
11.3 Validierung durch unabhängige Spezifikation . . . 151

III Einbettung von Asbru in KIV 153

12 Implementierung der SOS-Regeln 155
12.1 Technische Prozeduren . . . 155
12.2 Zustandsübergänge ausgehend von der `Selection`-Phase . . . 157
12.3 Zustandsübergänge ausgehend von der `Execution`-Phase . . . 159
12.4 Prozeduren der Plankontrolltypen . . . 161

13 Korrektheit der Implementierung 169
13.1 Beweisidee . . . 169

13.2 Abbildung der Plannamen und Instanznamen . . . 170
13.3 Zustandsabbildung . . . 171
13.4 Korrektheit der Prädikate . . . 172
13.5 Abbildung der Prozeduren . . . 173
13.6 Korrektheit der Implementierung . . . 175
13.7 Zusammenfassung . . . 179

IV Formale Verifikation medizinischer Aussagen 181

14 Formalisierung von Aussagen mit GDL 183
14.1 Die GDL Syntax . . . 183
14.2 Formalisierung von Aussagen . . . 184

15 Symbolische Ausführung von Asbru 191
15.1 Aussagenformulierung . . . 191
15.2 Symbolische Ausführung . . . 196

16 Beweisdekomposition 199
16.1 Motivation der Beweisdekomposition . . . 199
16.2 Beweisdekomposition . . . 201

17 Nachweis von Gegenbeispielen 207
17.1 Problembeschreibung . . . 207
17.2 Lösungsansätze . . . 207

18 Automatische Modell-Prüfung 211
18.1 Generierung des abstrahierten Modells . . . 211
18.2 Verifikation von Aussagen . . . 211

19 Formalisierung des Patientenverhaltens 215

20 Beschreibung der Aussagentypen 217
20.1 Syntax der Asbru-Modelle . . . 217
20.2 Strukturelle Eigenschaften . . . 217
20.3 Medizinische Aussagen . . . 218
20.4 Bewertung des Ansatzes . . . 219

V Verifikation der Fallstudie 221

21 Aussage 20 223
21.1 Inhalt der Aussage und Formalisierung . . . 223
21.2 Beweis der Aussage . . . 224
21.3 Besonderheiten der Aussage . . . 231

22 Aussage 35 233
22.1 Inhalt der Aussage und Formalisierung . . . 233
22.2 Beweis der Aussage . . . 235
22.3 Besonderheiten der Aussage . . . 236

23 Aussage 38 237
23.1 Inhalt der Aussage und Formalisierung . . . 237

23.2 Beweis der Aussage . . . 237
23.3 Besonderheiten der Aussage . . . 237

24 Aussage 1 **239**
24.1 Inhalt der Aussage und Formalisierung . . . 239
24.2 Beweis der Aussage . . . 240
24.3 Besonderheiten der Aussage . . . 240

25 Bewertung des Ansatzes **243**
25.1 Aufwand der Verifikation . . . 243
25.2 Qualität der Leitlinie . . . 243
25.3 Anomalien in der Leitlinie . . . 244
25.4 Zusammenfassung . . . 244

VI Verwandte Arbeiten und Diskussion **247**

26 Verwandte Arbeiten **249**
26.1 Ansätze zur Qualitätssicherung in der Medizin . . . 249
26.2 Alternative Verifikationstechniken . . . 253

27 Diskussion und Ausblick **255**

VII Anhang **259**

A Implementierung der statischen Asbru-Komponenten **261**
A.1 Spezifikationen in KIV . . . 261
A.2 Spezifikationsgraph von Asbru . . . 263
A.3 Basisdatentypen . . . 265
A.4 Zustandsdatentypen . . . 271

B Korrektheit der Einbettung der Prozeduren **279**
B.1 `considered`-Zustand . . . 279
B.2 `possible`-Zustand . . . 281
B.3 `ready`-Zustand . . . 283
B.4 `activated`-Zustand . . . 285
B.5 `suspended`-Zustand . . . 287
B.6 Zustandsübergänge der `sequential`-Plankontrolle . . . 289
B.7 Zustandsübergänge der `anyorder`-Plankontrolle . . . 291
B.8 Zustandsübergänge der `parallel`-Plankontrolle . . . 293
B.9 Zustandsübergänge der `unordered`-Plankontrolle . . . 296
B.10 Zustandsübergänge der `onabort`-Plankontrolle . . . 298
B.11 Zustandsübergänge der `ifThenElse`-Plankontrolle . . . 300
B.12 Zustandsübergänge der `ask`-Plankontrolle . . . 303
B.13 Zustandsübergänge der `user`-Plankontrolle . . . 304
B.14 Zustandsübergänge der `cyclical`-Plankontrolle . . . 304

C Prädikate zur Formulierung der SOS-Regeln **309**
C.1 Plankontrolltyp-Prädikate . . . 309
C.2 Prädikate des Planzustand . . . 309
C.3 Planzustandspropagation . . . 311

C.4 Funktionen zur Aktualisierung des Zustands 312
C.5 Planzustandsprädikate für Unterpläne 315
C.6 Planselektoren 317
C.7 Erfüllbarkeit 321
C.8 Planname Prädikate 323
C.9 Prädikate spezifisch für einzelne Plantypen 324
C.10 Das sync-Prädikat 325

D Bewiesene Aussagen **329**
D.1 Aussage 1 329
D.2 Aussage 5 330
D.3 Aussage 6 330
D.4 Aussage 7 331
D.5 Aussage 13 332
D.6 Aussage 14 332
D.7 Aussage 19, Teil II 333
D.8 Aussage 20 333
D.9 Aussage 35 333
D.10 Aussage 37 334
D.11 Aussage 38 334
D.12 Jaundice-Indikator 334

Literaturverzeichnis **337**

Teil I

Einführung, Grundlagen und Fallstudie

1 Gliederung dieser Arbeit

Diese Arbeit beschreibt die theoretischen Grundlagen der Verifikation medizinischer Leitlinien sowie deren technische Umsetzung und Einbettung in ein Beweissystem. Darüber hinaus stellt die Arbeit Beweistechniken zum Umgang mit konkreten Fallstudien vor und zeigt die Anwendung der vorgestellten Techniken auf eine medizinische Leitlinie zur Behandlung von Brustkrebs.

Abschnitt I der Arbeit gibt einen Einblick in die Welt der medizinischen Leitlinien. Weiterhin werden die Grundlagen der verwendeten Leitlinienmodellierungssprache Asbru, der Temporallogik ITL+ sowie die als Fallstudie verwendete Leitlinie beschrieben. Abschnitt II definiert die Syntax und die Semantik der Sprache Asbru und beschreibt vorgenommene Untersuchungen zur Validierung der Semantik. Abschnitt III der Arbeit beschäftigt sich mit der Einbettung der Sprache Asbru in das interaktive Verifikationssystem KIV sowie den Nachweis der Korrektheit der Einbettung. Anschließend werden in Abschnitt IV Beweistechniken zur Verifikation von Asbru vorgestellt sowie eine erste Bewertung des Ansatzes vorgenommen. Darauf folgend wird in Abschnitt V die Verifikation einiger Aussagen der Fallstudien vorgestellt. Abschließend wird in Abschnitt VI der Ansatz mit anderen Ansätzen zur Qualitätssicherung medizinischer Leitlinien verglichen, die Ergebnisse diskutiert und ein Ausblick auf eine mögliche Fortführung der Arbeit gegeben.

Abschnitt I gliedert sich dabei wie folgt: Kapitel 2 beschreibt den Leitlinien-Entwicklungsprozess und motiviert die Einführung neuartiger Techniken zur Qualitätssicherung. Anschließend wird in Kapitel 3 die Modellierungssprache Asbru eingeführt, die als Grundlage für die hier vorgestellten Techniken zur Verifikation medizinischer Leitlinien dient. Für die Verifikation wird eine Werkzeugunterstützung benötigt. Das verwendete Beweiswerkzeug KIV sowie die eingesetzte Temporallogik ITL+ werden in Abschnitt 4 beschrieben. Daran anschließend wird informell das Modell der verwendeten Fallstudie – eine Leitlinie zur Behandlung von Brustkrebs – in Kapitel 5 beschrieben.

Abschnitt II beschreibt die Syntax und Semantik von Asbru. Dabei wird zunächst in Kapitel 6 die Syntax von Asbru definiert. Im Anschluss daran wird in Kapitel 7 Syntax und Semantik des Zustands von Asbru-Systemen beschrieben. Darauf folgt die Beschreibung der Semantik von Bedingungen in Asbru in Kapitel 8. Mit dieser ist es möglich die Semantik des Zustandsübergangssystems von Asbru zu definieren, was in Abschnitt 9 geschieht. Durch Anwendung der Zustandsübergangsregeln auf einen Zustand und ein Asbru-System unter Einbeziehung einer Umgebung entstehen Asbru-Abläufe. Diese werden in Kapitel 10 vorgestellt. Der Abschnitt endet mit einer Beschreibung der Untersuchungen zur Validierung der Asbru-Semantik, die in Kapitel 11 dokumentiert ist.

In Abschnitt III wird die Einbettung der definierten Syntax und Semantik von Asbru in das interaktive Beweissystem KIV beschrieben. Dazu wird zunächst in Kapitel 12 beschrieben, wie die Zustandsübergangsregeln von Asbru in KIV eingebettet werden. Darauf folgend wird in Kapitel 13 die Korrektheit der Einbettung bewiesen. Die Einbettung der Asbru-Datentypen in KIV ist trivial und findet sich daher im Anhang in Abschnitt A.

Abschnitt IV beschreibt die Beweistechnik zur Verifikation von Aussagen über Asbru-Systemen. Dazu wird zunächst in Kapitel 14 beschrieben, wie informelle medizinische Aussagen schrittweise in temporallogische Formeln umgewandelt werden können. Auf diese Art formalisierte Aussagen werden dann entweder mit der Beweistechnik der symbolischen Ausführung oder der Beweisdekomposition verifiziert. Erstere ist in Kapitel 15 beschrieben

und eignet sich besonders für Aussagen über kleine Asbru-Planhierarchien. Für größere Planhierarchien wird die Beweisdekomposition eingesetzt, die in Kapitel 16 vorgestellt wird. Zum Nachweis von Gegenbeispielen wird der Gegenbeispielmechanismus verwendet. Dieser wird in Kapitel 17 vorgestellt. Für eine eingeschränkte Klasse von Aussagen kann statt der interaktiven Verifikation die automatische Modellprüfung verwendet werden. Diese wird in Kapitel 18 vorgestellt. Kapitel 19 beschäftigt sich mit der Frage, wie Wissen über Zustandsänderungen des Patienten formalisiert und für die Beweisführung verwendet werden kann. Der Abschnitt wird beschlossen durch eine Beschreibung von Aussagenklassen sowie einer Bewertung welche Beweistechniken sich für welche Aussageklassen eignen. Dies wird in Kapitel 20 beschrieben.

Abschnitt V dokumentiert detailliert die Verifikation von vier Aussagen der Brustkrebs Fallstudie. Diese wurden aufgrund besonderer Eigenschaften der Beweise ausgewählt. Die Beweise der übrigen Aussagen der Fallstudie sind im Anhang in Kapitel D aufgeführt.

Die Arbeit wird mit Abschnitt VI abgeschlossen. Dort werden in Kapitel 26 Arbeiten anderer Forschergruppen vorgestellt und deren Relevanz in Bezug auf diese Arbeit diskutiert. Danach werden in Kapitel 27 die Ergebnisse der Arbeit zusammengefasst und Themen für weiterführende Forschungsarbeiten eingeführt.

2 Motivation und Einordnung in den Leitlinien-Entwicklungsprozess

Dieser Abschnitt führt den Begriff der medizinischen Leitlinien ein. Daneben wird der Prozess, der hinter der Erstellung von Leitlinien steht erläutert. Damit wird erklärt, worin das Problem der Qualität von Leitlinien besteht und warum die bisherigen Ansätze zur Qualitätssicherung nicht weit genug führen. Darauf aufsetzend wird der hier vorgeschlagene Ansatz skizziert und in den Kontext anderer Forschungsarbeiten gesetzt.

2.1 Leitlinien zur Verbesserung der Medizin

Jedes Jahr werden viele medizinische Studien durchgeführt und damit so viele neue medizinische Erkenntnisse veröffentlicht, dass es für einzelne Ärzte schwer bis unmöglich ist, ihr Wissen auf dem aktuellen Stand zu halten. [20] gibt allein die Zahl medizinischer Journale, die pro Woche veröffentlicht werden mit über 1000 an. Selbst für spezialisierte Ärzte, wie etwa Onkologen, bedeutet das Studium der für sie relevanten Studien erheblichen Aufwand. Dabei ist zu beachten, dass die Kenntnis der Studien alleine noch nicht ausreichend ist. Vielmehr ist auch eine Bewertung der Qualität der Studien erforderlich. Dabei muss unterschieden werden, ob die Studien nach wissenschaftlichen Standards, beispielsweise unter Einsatz von Kontrollgruppen, erstellt wurden oder lediglich Expertenmeinungen darstellen. Auch eine Bewertung, ob die Ersteller der Studie möglicherweise durch Interessenkonflikte in ihrem Urteil beeinflusst wurden, ist nötig.

Die Idee von Leitlinien ist es, diese Arbeit zu zentralisieren und den einzelnen Ärzten abzunehmen. Statt den Ärzten sollen spezialisierte Expertengruppen das für eine Krankheit verfügbare Wissen zusammentragen, bewerten und daraus Dokumente erstellen, die dann möglichst kurz und prägnant das vorhandene Wissen zusammenfassen. Damit wird dem Arzt eine Wissensbasis gegeben, die den Goldstandard der Behandlung beschreibt, also die bestmöglichen bekannten Behandlungsmöglichkeiten einer Krankheit. Auf diese Weise soll erreicht werden, dass der Patient überall im Wirkungsbereich der Leitlinie gleich gut behandelt wird. Dies bedeutet, dass als unnötig erkannte Behandlungen oder Diagnosen nicht mehr durchgeführt, neue Behandlungsmethoden aufgezeigt und so Kosten und Fehldiagnosen vermieden werden.

[20] nennt ein Beispiel zur Beleuchtung der Schwierigkeiten im Umgang mit medizinischen Studien. Im Jahr 1958 wurde eine Studie publiziert, aus der hervorgeht, dass die Gabe von Streptokinase den Verlauf von Herzanfällen positiv beeinflussen kann. In den frühen 70er Jahren des letzten Jahrhunderts wurde diese Studie wiederholt und die Ergebnisse von 1958 bestätigt. Etwa 10 Jahre später wurde eine Meta-Analyse durchgeführt, die mehrere Studien zu diesem Thema zusammenfasst und das Ergebnis von 1958 bestätigt. Eine formale Empfehlung diese Behandlung durchzuführen wurde erst 13 Jahre nach dem Vorliegen dieses wissenschaftlichen Nachweises der Wirksamkeit der Behandlung ausgesprochen. Mit der Einführung von Leitlinien ist die Hoffnung verbunden, den Zeitraum zwischen dem Vorliegen wissenschaftlich gesicherter Information und der tatsächlichen Empfehlung an die breite Ärzteschaft zu verringern.

Leitlinien entlasten den Arzt lediglich bei der Recherche und Bewertung der Literatur. Beschreibungen, wie eine Behandlung durchgeführt werden soll sind nicht Teil einer Leitlinie.

Der Arzt ist nach wie vor in der Pflicht, die in der Leitlinie aufgeführten Prozeduren selbst auszuführen und sich die dazu nötigen Kenntnisse beizubringen. Außerdem soll der Arzt in der Lage sein, die Leitlinie zu hinterfragen. Um dies zu ermöglichen, enthalten Leitlinien Literatur-Referenzen, die die Studien benennen, die getroffene Aussagen und Empfehlungen begründen.

Moderne Leitlinien werden auf der Grundlage evidenzbasierter Medizin erstellt, wie sie etwa in [26] beschrieben wird. Evidenzbasierte Medizin verlangt, dass bei der Entscheidungsfindung, welche Behandlung durch eine Leitlinie empfohlen wird, zunächst alle relevante Literatur gesichtet werden soll. Die Qualität der ausgewählten Studien soll danach bewertet werden. Im einen Extremfall handelt es sich bei dem Ergebnis der Studie um die Expertenmeinung eines Angestellten eines Pharmaunternehmens, im anderen Extremfall um eine doppelblinde Studie[1], die von einem unabhängigen Forschungsinstitut durchgeführt wurde. Bei der Aufnahme von Studien in eine Leitlinie muss diese Form der Qualität berücksichtigt werden. In der Konsequenz werden einzelne Studien deswegen möglicherweise nicht in die Leitlinie aufgenommen.

Verschiedene Untersuchungen kommen zu dem Ergebnis, dass die Qualität medizinischer Behandlungen mit dem Einsatz von Leitlinien steigt. Beispielsweise wird in [19] gezeigt, dass durch den Einsatz von Leitlinien die Behandlungskosten um ein Viertel reduziert werden können, während gleichzeitig [1] belegt, dass die Qualität der Behandlung beim Einsatz von Leitlinien steigt.

2.2 Qualität der Leitlinien

Leitlinien werden in der Regel von Ärztegremien als informelle Dokumente erstellt. Dabei enthalten Leitlinien verschiedene Komponenten wie freien Text, Tabellen und Grafiken. Das Dokument wird mit Literaturreferenzen angereichert, um die einzelnen Inhalte zu untermauern.

Bevor eine Leitlinie in der Praxis im großen Umfang eingesetzt wird, muss die Qualität der Leitlinie bewertet werden. Damit stellt sich die Frage, wie die Qualität von Leitlinien bemessen werden kann, bzw. welche Qualitätskriterien es gibt. Qualitätskriterien von Leitlinien können in drei Gruppen eingeteilt werden.

Zunächst gibt es Kriterien, die den Erstellungsprozess selber betrachten. Um diese Kategorie bewerten zu können, ist es beispielsweise notwendig, zu evaluieren, ob die Leitlinienersteller in Interessenskonflikte verwickelt waren, ob die Leitlinie in einem Pilotversuch getestet wurde oder, ob systematische Methoden zur Literaturrecherche angewendet und dokumentiert wurden.

Eine zweite Qualitätskategorie befasst sich mit dem eigentlichen Leitliniendokument. In diesem sollte beispielsweise die Konsistenz einzelner Abschnitte zueinander sichergestellt werden. Weiterhin muss auch die Vollständigkeit des Dokumentes überprüft werden, etwa ob informelle Formulierungen ausreichend durch konkretes Zahlenmaterial spezifiziert werden. Ein Beispiel dafür ist ein Freitextblock, der sich mit „stark steigenden“ Blutwerten befasst, aber nicht definiert, wie sich diese konkret bemessen. Ein letztes Beispiel für diese Art der Qualitätsbemessung ist die Vollständigkeit von Aufzählungen. Bei textuellen Aufzählungen verschiedener Patientengruppen kann es passieren, dass einzelne Patienten in mehrere Gruppen eingeordnet werden, weil etwa orthogonale Kriterien wie Alter und das Stadium der Krankheit verwendet werden. Dadurch kann es vorkommen, dass Patienten in überhaupt keine Gruppe eingeordnet werden können oder in mehrere Gruppen passen. Dies kann ein Problem darstellen, wenn

[1] Als doppelblind wird eine Studie bezeichnet, wenn neben der behandelten Patientengruppe eine zweite Gruppe von Patienten existiert, die statt der echten Behandlung lediglich Placebos erhält und weder die Patienten noch die behandelnden Ärzte wissen, welcher Patient in welcher Gruppe ist.

dadurch unklare Vorgaben für die konkrete Behandlung dieser Patienten entstehen. Die hier genannten Beispiele beschreiben sogenannte Anomalien, die im Rahmen der Forschungsarbeit der Protocure-Projekte, zu der auch die vorliegende Arbeit gehört, gefunden wurden. Diese Fehler konnten in Leitlinien nachgewiesen werden, die tatsächlich für die Patientenversorgung eingesetzt wurden.

Die letzte Kategorie von Qualitätskriterien steht außerhalb der Leitlinie. Sie umfasst rechtliche Vorgaben der Behandlung oder Beschreibungen des gegenwärtigen Goldstandards. Beispiele hierfür sind die rechtliche Vorgabe, dass Patienten vor chirurgischen Eingriffen informiert werden müssen und ihre Einwilligung in die Prozedur geben müssen. Genauso können dies Beschreibungen kontraindizierter Behandlungsabläufe sein. Beispielsweise ist nach einem chrirugischen Eingriff bei der Krebsbehandlung eine bestimmte Form aggressiver Strahlentherapie zu gefährlich, um sie anzuwenden. Ein letztes Beispiel für solche Kriterien sind die Qualitätsindikatoren, die Krankenhäuser und Ärzte in Zukunft veröffentlichen müssen, um den Patienten im Vorfeld der Behandlung über Erfolgsquoten der Einrichtung aufzuklären. Dabei handelt es sich um statistische Daten, die etwa die Zahl der Komplikationen, die Langzeitüberlebenschancen oder ähnliches messen. Kriterien dieser Kategorie lassen sich unterteilen in Prozess- und Ergebnisvorgaben. Prozessvorgaben treffen Aussagen über die Sequenz von Behandlungen und beschreiben Abläufe, die gefordert, verboten bzw. erlaubt sind. In den obigen Beispielen wird etwa gefordert, dass eine Patienteninformation dem Eingriff vorangestellt wird, bzw. verboten, dass eine aggressive Strahlentherapie im Anschluss an eine Operation durchgeführt wird. Ergebnisvorgaben beschreiben dagegen nicht den Behandlungsablauf, sondern das Ergebnis. In den obigen Beispielen wären die Komplikationen bzw. die Langzeitüberlebenschancen Ergebnisvorgaben, da sich diese nicht auf den Behandlungsablauf sondern auf das Behandlungsergebnis beziehen.

2.3 Qualitätssicherung bei Leitlinien

Um die Qualität von Leitlinien zu sicherzustellen, müssen Werkzeuge bereitgestellt werden, die die in Abschnitt 2.2 aufgeführten Qualitätskriterien nachweisen können. Die Gemeinschaft der Leitlinienentwickler hat dies zum Teil erkannt. Aus diesem Grund wurde das AGREE-Instrument[2] geschaffen. Dieses versucht, die erste Kategorie der Qualitätskriterien in Leitlinien zu überprüfen. Dazu werden Experten herangezogen, die mit der Erstellung der Leitlinie nicht befasst waren und keinem Interessenskonflikt unterliegen. Diese sollen einen Fragenkatalog zu der Leitlinie beantworten und dabei bewerten, ob beispielsweise alle relevanten Interessenvertreter angehört wurden, ob die Anwendergruppe der Leitlinie ausreichend beschrieben wurde und ähnliches. Für die Einhaltung einzelner Kriterien werden von den Experten Punkte vergeben, die dann zusammengenommen einen AGREE-Wert bilden. Dieser soll einen möglichst einfach zu erfassenden Anhaltspunkt für die Qualität darstellen.

Allerdings bietet die Leitliniengemeinschaft darüberhinaus keine anderen Werkzeuge zur Qualitätssicherung, die über informelle Gutachten hinausgehen. An dieser Stelle setzt das Protocure-Projekt[3] an, in dessen Rahmen diese Arbeit entstanden ist. Diesem Projekt liegt die Erkenntnis zugrunde, dass Leitlinien, wenn sie geeignet interpretiert werden, als eine Art von (Computer-) Programmen angesehen werden können. Dabei ist der Arzt der „Prozessor“, der das „Leitlinien-Programm“ ausführt und dabei den Zustand der Patienten verändert.

Aus dieser Interpretation erwächst die Hoffnung, dass es möglich ist, das Wissen und die Methoden, die bereits bestehen, um die Qualität von Software sicherzustellen, in die medizinische Domäne zu übertragen. Damit sind Qualitätssicherungsmethoden wie die formalen Verifikation gemeint. Die Protocure-Projektgruppe schlägt einen werkzeugunterstützten An-

[2] http://www.agreecollaboration.org/, besucht am 18.7.2008
[3] www.protocure.org, besucht am 18.7.2008

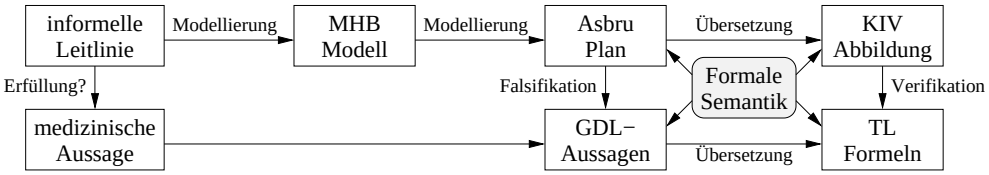


Abbildung 2.1: Der Protocure Ansatz

satz vor, der das informelle Leitlinien-Dokument schrittweise formalisiert bis zu dem Punkt, an dem ein formales, ausführbares Modell vorliegt. Dieses wird dann verwendet um Aussagen zu beweisen und damit die Korrektheit der Leitlinie zu prüfen.

Dazu wird in Protocure ein Ansatz verfolgt, der in Abbildung 2.1 dargestellt wird. Aus dem informellen Dokument wird zunächst in eine semi-formales Modell in der Sprache MHB [66] erstellt. Diese Sprache beinhaltet formale Strukturierungselemente wie auch freie Textblöcke. So erlaubt MHB, Textblöcke der Leitlinie zu strukturieren und beispielsweise temporale Abhängigkeiten zwischen ihnen festzulegen. Ausgehend von diesem Modell wird ein Asbru-Modell [64, 65] generiert. Asbru ist eine Sprache, die für die Modellierung medizinischer Leitlinien geeignet ist und im Rahmen dieser Arbeit vollständig mit einer formalen Semantik unterlegt wird. Somit ist es möglich, Asbru-Modelle auszuführen. Gleichzeitig werden die Qualitätssicherungsaussagen formalisiert. Damit ergibt sich die Möglichkeit, die Leitlinie auszuführen und das Ergebnis mit den formalisierten Aussagen abzugleichen. Auf diese Weise können Aussagen falsifiziert werden.

Aussagen werden in sogenannte GDL-Aussagen übersetzt. GDL-Aussagen erlauben eine schrittweise Formalisierung der informellen Aussage bis zu einem Punkt, an dem eine vollautomatische Übersetzung in temporallogische Formeln möglich ist. Details dieser Übersetzung sind in Abschnitt 14 dokumentiert. Eine Übersetzung des Asbru-Modells in den interaktiven Beweiser KIV [57, 58] erlaubt schlussendlich einen computerunterstützten Korrektheitsbeweis bezüglich dieser Formeln durchzuführen und somit die Korrektheit der Leitlinien relativ zu den Aussagen nachzuweisen.

Bei der Umsetzung dieses Ansatzes hat sich gezeigt, dass sich das vorgeschlagene Vorgehen dazu eignet, die Qualitätssicherungslücke zu schließen. Die schrittweise Formalisierung der Leitlinie zwingt zu einer Auseinandersetzung mit dem Dokument auf einer Ebene, die in der Regel von informellen Untersuchungen nicht erreicht werden kann. Da der freie Text in Entscheidungsbäume und Prozessabläufe übersetzt werden muss, werden sprachliche Mehrdeutigkeiten im Dokument aufgedeckt und müssen eindeutig festgelegt werden. Definitionsauslassungen werden dabei ebenso bemerkt wie Inkonsistenzen zwischen einzelnen Textabschnitten. Jede der so entdeckten Anomalien wird mit den Leitlinienerstellern diskutiert, um eine Klärung der Probleme zu erreichen. Beispielsweise kann ein Ergebnis solcher Diskussionen sein, dass auf einen offenen Punkt der Leitlinie deutlich hingewiesen wird, da das verfügbare medizinische Wissen nicht ausreicht ihn eindeutig zu klären. In der Regel werden solche Anomalien aber dazu führen, dass die Leitlinie präzisiert bzw. korrigiert wird.

Mit dem formalen Modell, das das Ergebnis dieser Übersetzung ist, kann anschließend formale Verifikation durchgeführt werden. Die dazu notwendigen Aussagen lassen sich aus der dritten Kategorie von Qualitätskriterien, also den Prozess- und Ergebnisaussagen, formulieren. Dabei ist zu beachten, dass die Korrektheit einiger Aussagen außerhalb des betrachteten Systems liegt. Beispielsweise ist die Zahl der bei Operationen auftretenden Komplikationen nur begrenzt durch die Leitlinie beeinflussbar. Stattdessen hängt die Korrektheit der Aussage vom behandelnden Arzt ab.

In solchen Fällen ist zu überprüfen, ob eine Umformulierung der Aussage möglich ist,

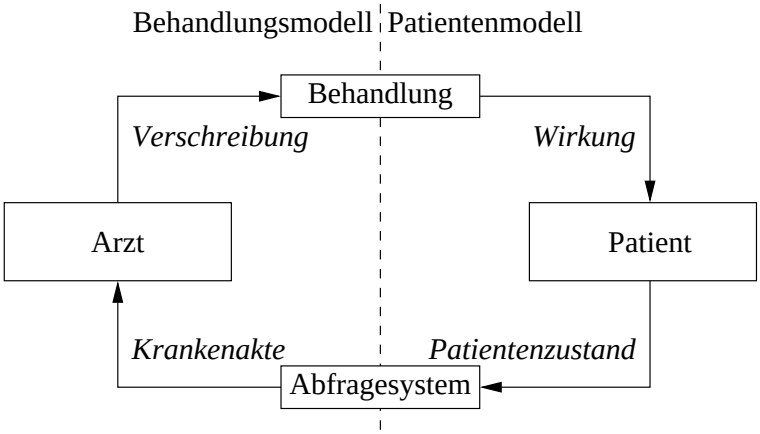


Abbildung 2.2: Abstraktes Modell medizinischer Behandlungen

und so wenigstens Teilaspekte überprüft werden können. Wenn beispielsweise ein bestimmter Operationsablauf die Zahl der Komplikationen reduziert, könnte überprüft werden, ob dieser von der Leitlinie vorgeschrieben oder zumindest erlaubt wird. In anderen Fällen kann die Einbeziehung von Hintergrundwissen helfen, Aussagen zu verifizieren oder zu falsifizieren. Dazu wird versucht, ein Patientenmodell zu erstellen. Dieses soll die Reaktionen des Patienten auf bestimmte medizinische Interventionen festlegen.

Ein wichtiger Aspekt des in dieser Arbeit vorgestellten Ansatzes ist, dass er keine Umstellung der Arbeitsweise des Arztes bei der Anwendung der Leitlinien vorsieht. Die Qualitätssicherung nach dem Protocure-Ansatz kann orthogonal zum eigentlichen Leitlinienentwicklungsprozess durchgeführt werden. Für einen behandelnden Arzt ist es nicht ersichtlich, ob eine Leitlinie mit dem Protocure Ansatz gesichert wurde oder nicht, da Struktur und Inhalt des Dokuments darüber keinen Aufschluss geben.

Umstellungsaufwand entsteht lediglich für Entwickler von Leitlinien. Diese müssen an der Formalisierung der Leitlinien beteiligt werden. In der Regel verfügen die Sprachexperten, die die Formalisierung der Leitlinien durchführen nur über eingeschränkte Kenntnis der medizinischen Domäne. Daher brauchen diese einen Ansprechpartner mit medizinischem Wissen, idealerweise aus der Gruppe derer, die die betreffende Leitlinie erstellt hat.

2.4 Abstrakte Sicht auf die Medizin

Abstrakt betrachtet interagieren bei einer medizinischen Behandlung der Arzt, der Medikamente verschreibt oder Eingriffe vornimmt und der Patient, dessen Zustand sich daraufhin ändert. Bei der Formulierung der Zustandsänderung des Patienten muss beachtet werden, dass vielfach ungeklärt ist, ob und wie genau Medikamente bei einem bestimmten Patienten wirken. Nebenwirkungen treten beispielsweise nur bei einigen Patienten auf. Auch kann ein Arzt im Allgemeinen nicht sicher sein, dass sein Patient sich auch tatsächlich vollständig an die Verordnungen hält. Weitere Unsicherheiten kommen dadurch zustande, dass Messungen in der Regel zu verzögerten Resultaten führen. Eine Blutuntersuchung im Labor, die 12 Stunden dauert, beschreibt den Blutwert bei Entnahme der Probe, nicht bei Bekanntwerden des Ergebnisses.

Um diese Unsicherheiten behandeln zu können, soll ein möglichst generisches Modell verwendet werden. Fest vorgegeben ist dabei lediglich die Schnittstelle zwischen Arzt und Patient sowie zwischen Patient und Arzt. Dabei soll der abstrakte Patient Informationen darüber bekommen, welche Behandlungen derzeit vom Arzt verschrieben werden. Umgekehrt soll der

Patient den Arzt über seinen Zustand informieren. In dieser abstrakten Sicht passiert dies mittels eines Abfragesystems. Das Konzept „Abfragesystem“ beinhaltet alle mit den Messungen verbundenen Unsicherheiten, beispielsweise die Verzögerung zwischen dem Zeitpunkt einer Messung und dem Vorliegen von Ergebnissen oder Messfehler.

Dieses System wird in Abbildung 2.2 dargestellt. Das Behandlungsmodell beschreibt die Handlungen des Arztes, die durch eine Leitlinie festgelegt werden. Die Entscheidungen des Arztes werden in Form einer Verschreibung festgehalten und an das Patientenmodell weitergereicht. Ein Konzept, das in der Grafik mit „Behandlung“ gekennzeichnet ist, extrahiert aus den Handlungen des Arztes die Behandlungen des Patienten, die diesem in Form der Wirkung übergeben werden. Auf Basis dieser Wirkungen verändert der Patient seinen Zustand und übergibt diesen an das Abfragesystem. Dieses erzeugt aus dem Patientenzustand eine Krankenakte. Dabei werden, wie bereits erläutert, Messverzögerungen und Messfehler berücksichtigt. Auf Basis dieser Krankenakte beginnt ein neuer Zyklus.

Der abstrakte Patient reagiert auf die vom Arzt verschriebenen Behandlungen. Allerdings sieht dieses Modell vor, dass im Allgemeinen diese Reaktion vollkommen zufällig und unvorhersehbar ist. Damit soll der Tatsache Rechnung getragen werden, dass vielfach keine generelle Aussage darüber getroffen werden kann, ob und wie eine Intervention auf den Patienten wirkt. Sollte allerdings Wissen über Patientenreaktionen bestehen so kann dieses fallstudienspezifisch dem Modell zugefügt werden. Beispielsweise kann formalisiert werden, dass eine Dosis von mindestens 500mg Paracetamol oral eingenommen Fieber jenseits der 38 Grad Celsius innerhalb von zwei Stunden absenkt.

2.5 Medizinische Aussagen als Beweisproblem

Mit der abstrakten Sicht auf die Interaktionen zwischen Patienten und dem medizinischen Personal ist es möglich, die Korrektheit von Leitlinien als Verifikationsproblem aufzufassen. Dabei geben das Modell des Patienten und das Modell der medizinischen Interventionen ein System vor. Dieses entsteht dadurch, dass immer abwechselnd das Patientenmodell ausgeführt wird, auf Basis der Ergebnisse des Abfragesystems dann das Modell der medizinischen Interventionen abgearbeitet wird. Dessen Ausgabe, also die derzeit verschriebenen Behandlungen, dienen danach dem Patientenmodell als Eingabe für den nächsten Zyklus. Dieses System lässt sich beschreiben als eine potentiell unendliche Menge von Abläufen. Gleichzeitig lässt sich eine medizinische Aussage als eine Menge von Abläufen auffassen, die zulässig sind. Beispielsweise kann festgelegt werden, dass nur solche Abläufe zugelassen sind, bei denen Fieber mit mehr als 38 Grad Celsius mit einer Gabe von Paracetamol behandelt wird. Geprüft werden muss nun, ob eine Inklusion der Abläufe vorliegt, ob also alle Abläufe des Modells auch von der Aussage zugelassenen sind. Dies wird im Beispiel so gemacht, dass verlangt wird, dass jedesmal, wenn das Abfragesystem einen entsprechenden Fieberwert feststellt im nächsten Schritt eine Verschreibung von Paracetamol gelten muss. Diese Ablaufinklusion kann auf die Beweisführung in temporallogischen Systemen zurückgeführt werden, wie in der vorliegenden Arbeit geschehen. Damit kann eine Menge bereits existierender Werkzeuge zur technischen Unterstützung der Verifikation verwendet werden.

Einige temporallogische Ansätze unterscheiden zwischen dem Konzept des Systems und dem der Umgebung. Mittels verschiedener Konstrukte ist es dann möglich, System und Umgebung miteinander interagieren zu lassen und so die Abläufe des Gesamtsystems zu erhalten. Die hier verwendete Logik ITL+, die in [8] vorgestellt wird, erlaubt die Integration der System und Umgebungskonzepte auf einem sehr intuitiven Weg, indem bereits bei Entwurf der Logik eine Unterscheidung zwischen System und Umgebungsschritten vorgesehen wurde. System und Umgebung werden automatisch auf eine geeignete Weise miteinander kombiniert, wie in Abbildung 2.3 dargestellt. Bei korrekter Beschreibung der Handlungsmöglichkeiten

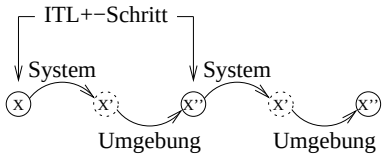


Abbildung 2.3: Abfolge von System und Umgebungsschritten in ITL+

von System und Umgebung - jeweils getrennt voneinander - ist also sichergestellt, dass das Gesamtsystem alle zu betrachtenden Abläufe enthält. Die Beweistechnik der symbolischen Ausführung verbunden mit temporallogischer Induktion sowie Generalisierungen ermöglicht es, die Korrektheit von Aussagen zu beweisen, indem inkrementell die Abläufe des Systems sowie der Aussage erzeugt und miteinander verglichen werden.

2.6 Ergebnisse dieser Arbeit

Die dieser Dissertation zugrundeliegende Forschungsarbeit hat verschiedene Ergebnisse hervorgebracht. Als erstes kann dabei die formale Semantik der Sprache Asbru genannt werden. Diese wird gegenüber Vorarbeiten verfeinert. Auch werden im Gegensatz zu den bestehenden Entwürfen alle Randfälle betrachtet. Mit Versuchen zur Validierung der Semantik wurden zahlreiche Fehler in der Semantik gefunden und behoben. Unklarheiten über die Interpretation von Asbru-Konstrukten, die selbst bei den Asbru-Entwicklern bestanden, konnten geklärt werden. Dazu wurden die zeitlichen Aspekte von Asbru auf eine formale Grundlage gestellt. Dabei wurde aufgedeckt, dass bei diesen zum Teil miteinander unvereinbare Anforderungen bestanden, die infolgedessen neu formuliert werden mussten. Die definierte Semantik wurde durch verschiedene Versuche ausgiebig validiert, um sicherzustellen, dass die Definition tatsächlich der Semantik entspricht, die von den Sprachdesignern vorgesehen ist.

Die Definition der formalen Semantik der Sprache Asbru ist die Grundlage für alle Experimente zur Validierung und Verifikation. Auf Basis der Semantik ist es möglich, die Asbru-Modelle in eine Eingabesprache für Modellprüfer zu übersetzen. Damit ist die Grundlage für den automatischen Aussagenbeweis geschaffen. Auch für die interaktive Verifikation ist die formale Semantik die Grundlage. Zuletzt muss auch für die Interpretation der Asbru-Modelle im Sinne eines Entscheidungsunterstützungssystems eine formale Semantik existieren.

Der im Vorfeld bestehende Prototyp für die Einbettung von Asbru in das interaktive Beweiswerkzeug KIV wurde verworfen und komplett neu entwickelt. Anstelle von manueller Übersetzung der Asbru-Konstrukte in temporallogische Programme wurden generische Prozeduren entwickelt, die die Semantik von Asbru unabhängig von den konkreten Asbru-Plänen abbilden. Die Asbru-spezifischen, veränderlichen Teile des Modells werden in Form von Datentypen spezifiziert, die von den generischen Prozeduren ausgelesen werden. Diese Datentypen sind so gestaltet, dass die Asbru-Konstrukte damit wörtlich in das Verifikationswerkzeug übernommen werden können. Unsicherheit bezüglich der Übersetzung besteht somit nicht mehr.

Die Implementierung der Prozeduren wurde in vielfacher Weise validiert, bevor in einem letzten Schritt die Verifikation der Korrektheit der Prozeduren durchgeführt wurde. Dazu wurde eine Abbildung von Datentypen der Semantik und der Implementierung angegeben und nachgewiesen, dass bei isomorphem Ausgangszustand durch Anwendung von Regeln der Semantik und der Implementierung ein isomorpher Endzustand erreicht wird. Damit ist sichergestellt, dass Ergebnisse Gültigkeit besitzen, die durch Verifikation mit dem interaktiven Verifikationswerkzeug KIV durchgeführt werden.

Mit Hilfe dieser Beweisunterstützung wurde eine Fallstudie bearbeitet, die sich mit der Behandlung von Brustkrebs befasst. Die vorgegebenen Qualitätssicherungsaussagen wurden als Beweisverpflichtungen für das Beweissystem verwendet. Dazu war es notwendig, eine Beweistechnik zu entwickeln, die in der Lage ist, mit den sehr großen Planhierarchien der Fallstudie umgehen zu können. Neben der direkten symbolischen Ausführung wurde dazu die Technik der Beweisdekomposition eingeführt, die auf dem Rely-Guarantee-Ansatz basiert, wie ihn beispielsweise Lamport in [4] vorstellt. Um falsche Aussagen widerlegen zu können, wurde eine Technik zur Verifikation von Gegenbeispielen entworfen und angewendet [34].

Weiterhin wurden Techniken vorgeschlagen, mit denen eine Modellierung des Patientenverhaltens möglich ist. So kann medizinisches Hintergrundwissen, das nicht in den Leitlinien enthalten ist, formalisiert und in die Beweise eingebracht werden. Das bedeutet, dass nicht nur Prozessaussagen formuliert werden können, sondern auch Metaaussagen, die beispielsweise verlangen, dass ein Patient möglichst wenig Leid durch die Behandlung erfahren soll.

Zuletzt können die hier verwendeten Techniken als ein Muster angesehen werden, wie die formale Spezifikation hierarchischer Sprachen, die ähnlich zu Asbru sind, effizient durchgeführt werden kann und ebenso, wie die Implementierung solcher Spezifikationen mittels rekursiver Prozeduren und der dynamischen Prozessgenerierung durchgeführt werden kann.

2.7 Einbettung dieser Arbeit in den Rahmen von Protocure II

Das Projekt Protocure II, in dessen Rahmen diese Arbeit entstanden ist, ist ein von der EU im Rahmen von FP6[4] gefördertes Forschungsprojekt. An diesem Forschungsprojekt waren insgesamt sieben Forschergruppen aus vier Ländern beteiligt. Die Forschergruppen verteilen ihre Kompetenz über das gesamte Spektrum des Forschungsbereichs formaler Qualitätssicherung medizinischer Leitlinien.

Auf der medizinischen Seite unterstützen die Praxispartner von CBO und IKO das Projekt. CBO[5] ist das niederländische Institut für die Qualitätssicherung im Gesundheitswesen. Das CBO ist in den Niederlanden für die Erstellung und die Pflege der medizinischen Leitlinien verantwortlich. Das Projekt erhielt Unterstützung in Form realer Patientendaten aus dem IKO[6], dem Krebszentrum Ost, das verantwortlich ist für die Verwaltung und Behandlung von Krebsfällen, die im Osten der Niederlande auftreten. Die restlichen Projektpartner sind universitäre Forschergruppen und beschäftigen sich mit der Wissensmodellierung und dem Sprachdesign.

Mit dieser Gruppe wurde das Asbru-Modell einer Leitlinie zur Behandlung von Brustkrebs erstellt und vollständig formal untersucht. Aus dem Vorgängerprojekt Protocure I existieren bereits formalisierte Leitlinien zur Behandlung von Gelbsucht bei Neugeborenen sowie der Behandlung von Diabetes Typ-2.

2.8 Verwandte Arbeiten

Neben dem Protocure-Ansatz zur vollständigen Verifikation medizinischer Leitlinien existieren eine Reihe anderer Arbeiten zur Untersuchung der Korrektheit von Leitlinien. Die Initiative für diese Ansätze geht meist von medizinischen Forschungsgruppen oder Sprachexperten aus. Deren Ansätze erheben nicht den Anspruch der Untersuchung der formalen Korrektheit. Stattdessen liegt deren Fokus auf einer möglichst praxisnahen Untersuchung der Leitlinien mit dem Ziel, den Ärzten selbst ein Mittel zur Qualitätsprüfung der Leitlinien an die Hand zu geben.

[4] http://cordis.europa.eu/fp6/dc/index.cfm?fuseaction=UserSite.FP6HomePage, besucht am 18.09.2008
[5] http://www.cbo.nl/english/default_view, besucht am 18.7.2008
[6] http://www.iko.nl/, besucht am 18.7.2008

Dieses Ziel wird durch die Implementierung von Interpretern für die Modellierungssprachen erreicht. Leitlinien können damit anhand von realen oder idealisierter Patientendaten getestet werden. Dazu wird verglichen, ob die durch die Leitlinie angeordneten Behandlungen mit denen übereinstimmen, die vom Arzt erwartet werden. Durch die geringe Zahl an Testfällen, die untersucht werden kann, kann diese Technik nur als Validierungstechnik angesehen werden.

Über solche Validierung hinaus geht nur der GLARE Ansatz, der Modellprüfung zur Verifikation von Leitlinien einsetzt. Dieser Ansatz wird im Detail in Abschnitt 26.1.4 beschrieben und bewertet. Eine Übersicht aller verwandten Arbeiten ist in Abschnitt 26 aufgeführt.

3 Einführung in Asbru

Dieses Kapitel gibt eine Einführung in die Plansprache Asbru. Dazu wird wie folgt vorgegangen. Zunächst werden in Abschnitt 3.1 die Konzepte von Asbru an dem Beispiel der Behandlung von Brust-Krebs präsentiert. Dieses Beispiel soll Verständnis für die wichtigsten Asbru Konzepte schaffen. Daran anschließend wird in Abschnitt 3.2 detaillierter der Funktionsumfang von Asbru beschrieben. In Abschnitt 3.3 wird der Unterschied zwischen Asbru und Asbru Light erläutert. Abschnitt 3.4 klärt einige wichtige in dieser Arbeit verwendeten Begriffe. Abschließend beschreibt Abschnitt 3.5, wie medizinische Aussagen mittels der Sprache GDL formalisiert werden können.

3.1 Asbru – an einem Beispiel

In diesem Abschnitt soll Asbru an einem Beispiel vorgestellt werden. Dazu werden zunächst in Abschnitt 3.1.1 grundlegende Konzepte mittels eines Beispiels motiviert und dann im Anschluss in Abschnitt 3.1.2 das Zustandsmodell von Asbru erläutert.

3.1.1 Asbru-Konzepte

Asbru ist eine planbasierte Sprache zur Beschreibung von hierarchisch unterteilten Abläufen mit besonderem Fokus auf die Spezifikation zeitlicher Aspekte. Die Hauptanwendung von Asbru ist die Modellierung medizinischer Abläufe [46], jedoch wurden beispielsweise auch Trainingspläne für Sportler mit Asbru modelliert [44]. Das Grundkonzept von Asbru ist dabei ein Plan. Dieser steht in einer Hierarchie und kann Unterpläne steuern sowie selbst Unterplan eines anderen Plans sein. Eine Plandefinition enthält Informationen über die Unterpläne des Plans. Weiterhin enthält die Plandefinitionen Bedingungen, an die die Ausführung des Plans geknüpft ist. Diese beschreiben Situationen, in denen der Plan beispielsweise seine Ausführung einstellen soll. In der Plandefinition sind ebenfalls Informationen darüber enthalten, wie ein Plan seinen Unterplänen steuert. Zum Beispiel kann spezifiziert werden, dass die Unterpläne nacheinander oder alle gleichzeitig ausgeführt werden sollen.

Die hierarchische Struktur von Asbru kann genutzt werden, um eine medizinische Intervention darzustellen und schrittweise zu verfeinern. Repräsentiert ein Plan eine Intervention als solches, so beschreiben Unterpläne eine Aufteilung dieser Intervention in ihre Komponenten. Die Konzepte von Asbru sollen nun durch ein Beispiel aus der Fallstudie zur Behandlung von Brustkrebs erläutert und motiviert werden.[1]

Die Behandlung von DCIS[2] ist in Abbildung 3.1 dargestellt. Sie besteht aus dem Stellen der Diagnose des Krebsbefalls und einer anschließenden Behandlung gemäß den Ergebnissen dieser Diagnose. Damit können Diagnose und Behandlung als Unterpläne des Behandlungsplans von DCIS dargestellt werden. Die Behandlung wird ausgehend vom Ergebnis der Diagnose weiter untergliedert. Rechtliche Vorgaben verlangen, dass der Patient vor einer medizinischen Intervention über die Behandlungsrisiken aufgeklärt werden muss. Im Anschluss daran kann

[1] Dieses Beispiel ist angelehnt an die Fallstudie Brustkrebs, wird jedoch hier zu Illustrationszwecken stark vereinfacht dargestellt. Die korrekte Version dieser Behandlungsleitlinie kann in Kapitel 5 nachgelesen werden.

[2] DCIS steht für „Ductual Carcinoma in Situ“ und beschreibt einen Befall des Milchleiters der Brust mit Krebs.

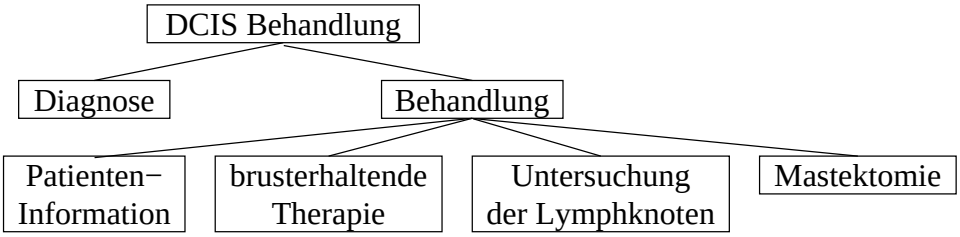


Abbildung 3.1: Darstellung der Behandlung von DCIS

die eigentliche Behandlung durchgeführt werden. Diese wiederum besteht entweder aus einer brusterhaltenden Therapie oder einer Mastektomie, also der Brustentfernung. Entsprechend dieser Aufteilung wird der Behandlungsplan in drei Unterpläne untergliedert. Die beiden chirurgischen Eingriffe dürfen allerdings erst nach erfolgreichem Abschluss der Patienteninformation ausgeführt werden. Somit ist dies die Vorbedingung der Pläne, die den chirurgischen Eingriff beschreiben. Weiterhin hat der Patient weitgehende Entscheidungsfreiheit bezüglich der Frage, ob brusterhaltende Therapie oder Brustentfernung durchgeführt werden soll. Eingeschränkt wird diese Entscheidungsfreiheit durch die Ergebnisse der Diagnose. Durch die Diagnose können Kriterien erkannt werden, die die brusterhaltende Therapie unmöglich machen. Statt einem chirurgischen Eingriff kann der behandelnde Arzt auch zunächst einen Eingriff anordnen, der den Befall von Lymphknoten mit Krebs prüft. Dies wird im Verdachtsfall präventiv getan oder aber verpflichtend im Anschluss an die brusterhaltende Therapie. Entscheidungen, die vom Zustand des Patienten oder von seinen Vorlieben abhängen werden in Asbru in Form von Bedingungen modelliert. Der Plan zur brusterhaltenden-Therapie hat beispielsweise eine Vorbedingung, die dessen Ausführung verhindert, wenn der Patient einen anderslautenden Wunsch ausgesprochen hat oder medizinische Fakten vorliegen, die der Therapie entgegenstehen. Weiterhin enthält der Plan eine Vorbedingung, die verlangt, dass die Patienteninformation erfolgreich abgeschlossen wurde.

Der brusterhaltende chirurgische Eingriff kann nur als erfolgreich gewertet werden, wenn die Schnittränder des entnommenen Gewebes tumorfrei sind. Kann dieses Ergebnis durch den Operateur nicht erreicht werden, so muss die brusterhaltende Therapie erfolglos abgebrochen werden und stattdessen eine Mastektomie durchgeführt werden, um eine Ausbreitung des Tumors im Körper zu verhindern. Dies ist wiederum über Bedingungen modelliert. Ein laufender Plan überwacht seine Ausführung mittels zweier Bedingungen, die dafür sorgen, dass die Planausführung umgehend abgebrochen wird, sollte dies nötig sein. Im Beispiel kann das der Fall sein, wenn der Arzt während der Operation keine tumorfreien Schnittränder erzielen kann. Ein erfolgreicher Abschluss ist dagegen nur möglich, wenn die Operation beendet wurde und eine Untersuchung ergibt, dass der Krebs vollständig entfernt werden konnte.

Dieses Beispiel zeigt verschiedene wichtige Aspekte von Asbru. Es wird deutlich, dass es notwendig ist, über die Zustände der Pläne Buch zu führen und unterscheiden zu können, mit welchem Ergebnis ein Plan beendet wurde. In diesem Beispiel ist es wichtig, unterscheiden zu können, ob die brusterhaltende Therapie erfolgreich beendet werden konnte oder abgebrochen werden musste. In anderen Fällen kann es auch notwendig sein, zu unterscheiden, ob die brusterhaltende Therapie abgebrochen oder zurückgewiesen wurde. Unter einem zurückgewiesenen Plan versteht Asbru einen solchen, dessen Vorbedingung nicht erfüllt werden konnte. Im Beispiel könnte dies der Fall sein, wenn ein Patient die komplette Brustentfernung wünscht. Dann würde die brusterhaltende Therapie zurückgewiesen, nicht abgebrochen.

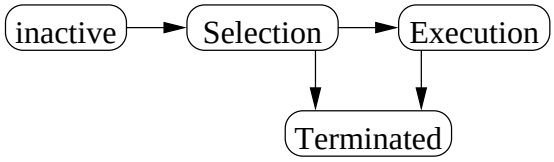


Abbildung 3.2: Überzustände bei Asbru-Plänen

3.1.2 Planzustände in Asbru

Im Folgenden soll das Konzept der Zustände genauer erläutert werden. Jeder Asbru Plan befindet sich stets in einem von mehreren möglichen Zuständen bzw. Phasen. Zunächst wird grob beschrieben welche Phasen es gibt, danach erfolgt eine Erläuterung, wie diese Phasen weiter in Zustände untergliedert werden können.

Ein Asbru Plan kennt vier mögliche Phasen, in denen er sein kann. Diese sind in Abbildung 3.2 abgebildet. Zum Einen kann ein Plan noch nicht gestartet worden sein. Er befindet sich dann in einer inaktiven Phase. Diese wird durch den Zustand `inactive` symbolisiert. Dies ist beispielsweise bei dem Plan zur Patienteninformation der Fall während die Diagnose noch nicht abgeschlossen ist. Offensichtlich kann in diesem Beispiel der Patient noch nicht aufgeklärt werden, da der Befund noch nicht vorliegt. Im Anschluss an den Planstart wird ein Plan auf seine Anwendbarkeit hin untersucht. In diesem Fall wird sein Zustand in die `Selection`-Phase eingeordnet. Dort wird überprüft, ob die notwendigen Vorbedingungen der Anwendbarkeit des Plans erfüllt sind. Ist dies nicht der Fall, wird der Plan zurückgewiesen. Sein Zustand fällt dann in die Kategorie der beendeten Fälle, d.h. die `Terminated`-Phase. Ist die Vorbedingung hingegen erfüllt, signalisiert der Plan seinem übergeordneten Plan, dass er bereit zur Ausführung ist. Es können mehrere Unterpläne eines Planes gleichzeitig zur Ausführung bereit sein. Entsprechend muss spezifiziert sein, welche dieser Pläne in welcher Reihenfolge zur Ausführung gelangen. Sind beispielsweise gleichzeitig die brusterhaltende Therapie sowie die Untersuchung der Lymphknoten der Achsel bereit zur Ausführung, so muss entschieden werden, wie weiter verfahren werden soll. Wird ein Plan ausgeführt, so kann er der vierten und letzten Phase zugeordnet werden, der `Execution`-Phase. In dieser Phase ist der Plan so lange, bis seine Ausführung beendet wird, er also in die `Terminated`-Phase wechselt.

Im Folgenden werden zur Verbesserung des Leseflusses folgende Begriffe synonym zu den Asbru Bezeichnungen verwendet. Ein Plan ist inaktiv, wenn sein Zustand `inactive` ist. Fällt sein Zustand in die `Selection`-Phase, so wird seine Anwendbarkeit evaluiert. Ein Plan in der `Execution`-Phase wird ausgeführt und ein Plan dessen Ausführung beendet oder abgebrochen wurde, ist in der `Terminated`-Phase.

Für jeden Plan wird spezifiziert, von welchem Plantyp er ist. Dieser Plantyp regelt, ob nur ein Unterplan gleichzeitig zur Ausführung kommt oder mehrere gleichzeitig ausgeführt werden können. Außerdem wird geregelt, ob alle Pläne gleichzeitig aus dem `inactive` Zustand in die `Selection`-Phase versetzt werden oder ob dies nacheinander geschehen soll. Auch existieren spezielle Plantypen, die beispielsweise den Start eines Unterplans vom Abbruch eines anderen Unterplans abhängig machen.

Die Behandlung von Brustkrebs aus dem Beispiel definiert, dass zwar die Anwendbarkeit aller Behandlungspläne gleichzeitig überprüft werden muss, jedoch nur einer der Pläne auf einmal aktiviert werden darf. So soll zwar gleichzeitig getestet werden, ob die brusterhaltende Therapie und die Brustentfernung möglich sind, letztendlich kann aber nur eine Behandlung tatsächlich durchgeführt werden. Welche dies ist, wird durch die Patientenentscheidung sowie andere äußere Umstände festgelegt. Um solche äußeren Einflüsse auf einen Plan zu

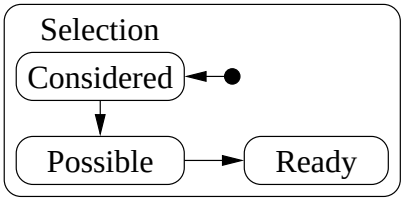


Abbildung 3.3: `Selection`-Phase von Asbru-Plänen

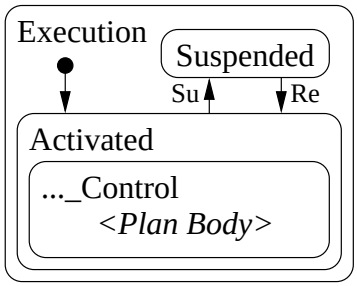


Abbildung 3.4: `Execution`-Phase von Asbru-Plänen

spezifizieren erlaubt Asbru die Definition so genannter Asbru-Bedingungen. Insgesamt sechs Bedingungen sind in Asbru vorgesehen, um das Ablaufverhalten eines Plans zu spezifizieren. Diese Bedingungen unterscheiden sich hinsichtlich des Zeitpunktes zu dem sie ausgewertet werden. Beispielsweise sind zwei Bedingungen vorgesehen, die die Anwendbarkeit des Plans in der `Selection`-Phase überprüfen. Weitere Bedingungen spezifizieren das Terminierungsverhalten, wie im Beispiel bereits beschrieben. Die Bedingungen der `Selection`-Phase werden **filter**- und **setup**-Bedingung genannt. Diese werden nacheinander ausgewertet. Dies ist grafisch in Abbildung 3.3 dargestellt. Ein Plan tritt in die `Selection`-Phase ein und wird damit in einen Zustand namens `considered` versetzt. In diesem Zustand wird der Wahrheitswert der **filter**-Bedingung überprüft. Ist diese erfüllt, so kann der Plan vom Zustand `considered` in den Zustand `possible` wechseln. Abhängig vom Zustand kann der Plan bei Nicht-Erfüllen seiner **filter**-Bedingung entweder die Ausführung beenden oder temporär im Zustand `considered` verbleiben. Im Zustand `possible` wird analog dazu die **setup**-Bedingung ausgewertet. Diese Auswertung kann wieder zu einem Fortschreiten führen – diesmal in den Zustand `ready` – oder zu einem Stillstand im Zustand `possible` bzw. zu einer Zurückweisung des Plans. Im Brustkrebs-Beispiel hat der Plan der die brusterhaltende Therapie steuert als **filter**-Bedingung eine Anforderung an die Diagnose. So muss zur Anwendung der brusterhaltenden Therapie gewährleistet sein, dass der Tumor auf einem Mammographiebild klar abgegrenzt werden kann. Sind dagegen die Tumor-Ränder verschwommen, so muss eine Mastektomie durchgeführt werden. Sollte ein Mammographie-Ergebnis noch nicht vorliegen, so wird die Frage der Anwendbarkeit der brusterhaltenden Therapie so lange zurückgestellt, bis dieses vorliegt.

Wird ein Plan aktiviert, dann verlässt er die `Selection`-Phase zugunsten der `Execution`-Phase. Diese untergliedert sich, wie in Abbildung 3.4 dargestellt, in zwei Zustände, einmal den aktivierten (`activated`) und dann den unterbrochenen (`suspended`) Zustand. Ein Plan im Zustand `activated` wird ausgeführt. Dies bedeutet, dass er seinerseits Unterpläne startet. Die Ausführung eines Plans kann unterbrochen werden, wenn Asbru-Bedingung mit Namen **suspend**-Bedingung erfüllt ist. Dann wird der Plan in den Zustand `suspended` versetzt. In diesem Zustand wird der Plan keine weiteren

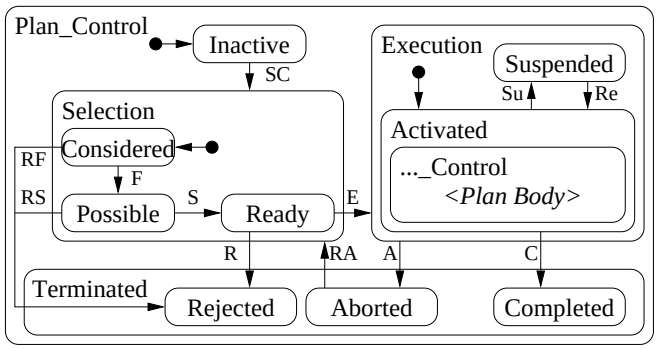
Unterpläne starten und die bereits gestarteten Unterpläne werden ebenfalls unterbrochen, sofern sie bereits in der `Execution`-Phase sind. Praktisch verwendbar ist dies beispielsweise als Teil der brusterhaltenden Therapie. Dort könnte festgelegt sein, dass vor Ende der Operation Gewebe entnommen wird und die Geweberänder auf Tumorreste untersucht werden sollen. Bis das Ergebnis vorliegt, sollte dann die eigentliche Operation unterbrochen werden. Mit Vorliegen des Ergebnisses sollte dann die Operation wieder aufgenommen und dann abgeschlossen werden. Die Wiederaufnahme der Ausführung ist an die sogenannte **reactivate**-Bedingung geknüpft. Ein Plan darf wieder vom `suspended` Zustand in den `activated` Zustand zurückwechseln, sobald diese Bedingung zu wahr ausgewertet wird. Logischerweise sollten im Beispiel alle Unterpläne der Operation mit dieser unterbrochen werden und auch bleiben, solange der Operationsplan unterbrochen ist. Das bedeutet, ein Plan wird unterbrochen, wenn entweder die entsprechende Bedingung vorliegt oder der übergeordnete Plan unterbrochen wird. Reaktiviert kann er nur werden, wenn die **reactivate**-Bedingung wahr ist und der übergeordnete Plan nicht (mehr) unterbrochen ist.

Ein Plan kann auf drei Arten seine Ausführung beenden. Ist eine der Vorbedingungen der `Selection`-Phase nicht erfüllbar, so wird der Plan zurückgewiesen (`rejected`). Wurde der Plan bereits in die `Execution`-Phase versetzt und wird dort die Abbruchbedingung (**abort**-Bedingung) wahr, so wird der Plan abgebrochen (`aborted`). Ein Plan wird erfolgreich beendet, wenn seine Beendigungsbedingung (**complete**-Bedingung) erfüllt ist. Er erreicht dann den Zustand `completed`.

Damit ein Plan abgeschlossen, also erfolgreich beendet werden kann, muss die **complete**-Bedingung erfüllt sein. Im Fall der brusterhaltenden Therapie besteht diese Bedingung aus zwei Komponenten. Zum Einen muss die eigentliche Operation, die durch Unterpläne repräsentiert wird, abgeschlossen sein, zum Anderen muss ein positives Ergebnis bezüglich der Schnittränder vorliegen. Technisch bedeutet das, dass sowohl die Ausführung der Unterpläne abgeschlossen, als auch ein bestimmter Zustand des Patienten und der Umgebung gegeben sein muss. Dies Überprüfung wird durch zwei unterschiedliche Asbru Konstrukte implementiert: Eine Asbru-Bedingung überprüft den Zustand, während ein spezielles Wartekonstrukt angibt, auf welche Unterpläne der Plan warten soll. Im Fall der Behandlung von DCIS wurde als Wartebedingung für den Behandlungsplan angegeben, dass entweder die Mastektomie oder aber die brusterhaltende Therapie in Verbindung mit der Untersuchung der Achsel-Lymphknoten abgeschlossen sein muss.

Ein Plan wird abgebrochen, wenn seine **abort**-Bedingung erfüllt ist oder ein Unterplan abgebrochen wurde, auf den der Plan wartet. Im Beispiel kann das der Fall sein, wenn die Schnittränder nicht tumorfrei sind. Weiterhin könnte die Gesamtbehandlung abbrechen, wenn weder die brusterhaltende Therapie noch die Mastektomie erfolgreich abgeschlossen werden können. In diesem Fall können die notwendigen Unterpläne des Behandlungsplans nicht erfolgreich abgeschlossen werden, was zu einer negativen Auswertung der Wartebedingung führt.

Asbru kennt den Begriff der Planzustandspropagation. Gemeint ist damit eine Weitergabe des Planzustandes an übergeordnete Pläne sowie Unterpläne. Ist in einem Plan einer seiner Unterpläne als Teil der Wartebedingung spezifiziert, so muss der Plan seine Bearbeitung abbrechen, wenn der Unterplan abgebrochen oder zurückgewiesen wird. Der Gedankengang dahinter ist, dass beispielsweise die DCIS Therapie nicht erfolgreich beendet werden kann, wenn bereits die Diagnose fehlschlägt. Dieses weitergeben des eigenen Planzustands an Unterpläne oder übergeordnete Pläne heißt Planzustandspropagation. Ebenfalls unter den Begriff der Planzustandspropagation fällt die Unterbrechung von Plänen, da ein Unterplan eines unterbrochenen Planes selbst unterbrochen werden soll, bis der übergeordnete Plan wieder aktiviert wurde.



F : satisfied(filter) and Activated(parent)
RF : not satisfiable(filter) or Terminated(parent)
S : satisfied(setup) and Activated(parent)
RS : not satisfiable(setup) or Terminated(parent)
Su : satisfied(suspend)
Re : satisfied(resume)
C : satisfied(complete)
A : satisfied(abort) or Terminated(parent)

Abbildung 3.5: Überblick über Asbru

3.2 Asbru Details

Nach dem beispielorientierten Überblick über Asbru soll hier informell der Funktionsumfang von Asbru dargestellt werden. Der Ablauf von Asbru Plänen wird im Wesentlichen von dem in Abbildung 3.5 angegebenen Statechart beschrieben. Dieser soll nun erläutert werden.

Ein Plan ist initial im Zustand `inactive`. In diesem Zustand wartet der Plan auf ein Signal des übergeordneten Plans. Im Diagramm wird dieses Signal mit `SC` beschrieben. Sobald das Signal empfangen wurden, betritt der Plan die `Selection`-Phase. Dort erreicht er zunächst den Zustand `considered`. In diesem Zustand wird die **filter**-Bedingung ausgewertet. Ist diese erfüllt, so geht der Plan entlang der mit F markierten Transition in den Zustand `possible`. Ist die **filter**-Bedingung nicht erfüllt, so wechselt der Plan über die Transition RF in den Zustand `rejected`. Kann die **filter**-Bedingung nicht ausgewertet werden, so bleibt der Zustand in `considered` stehen. Analog ist das Vorgehen im Zustand `possible`, in dem die **setup**-Bedingung ausgewertet wird. Ist diese erfüllt, so erreicht der Plan den Zustand `ready`. Ist die Bedingung nicht erfüllt, so wechselt der Plan wieder in den Zustand `rejected`. Wie auch im Zustand `considered` bleibt der Plan im Zustand `possible` stehen, wenn die **setup**-Bedingung nicht ausgewertet werden kann.

Im Zustand `ready` erwartet der Plan das Aktivierungssignal des übergeordneten Plans. Erst wenn dieses Signal empfangen wurde wechselt der Plan in die `Execution`-Phase, in der der Plan seine eigenen Unterpläne starten kann. Wird in der `Execution`-Phase die **complete**-Bedingung wahr und sind alle notwendigen Unterpläne des Plans im Zustand `completed`, so kann der Plan selbst in den Zustand `completed` wechseln. Ergibt sich aus dem Zustand, dass die notwendigen Unterpläne nicht mehr erfolgreich abschließen können oder ist die **abort**-Bedingung wahr, so wechselt der Plan in den Zustand `aborted`. In diesem Zustand kann ein Plan vom übergeordneten Plan ein Reaktivierungssignal erhalten mit dem der Plan wieder zurück in die `Selection`-Phase versetzt wird. Ist der Plan im Zustand `activated`, so kann die **suspend**-Bedingung wahr sein. In diesem Fall wechselt der Plan in den Zustand `suspended`. Im Zustand `activated` selbst steuert der Plan seine Unterpläne, in dem er die `SC` und `E` Signale an seine Unterpläne schickt.

In welcher Reihenfolge diese Signale genau verschickt werden, bestimmt der Plantyp. Dieser kann beispielsweise `sequential` sein, womit die Unterpläne der Reihe nach zuerst das `SC` erhalten und dann, sobald der `ready` Zustand erreicht wird, das `E` Signal. Erst nach Terminierung eines Unterplans wird bei diesem Plantyp der nächste Plan gestartet. Im Gegensatz dazu schickt ein Plan mit `anyorder` Plantyp zunächst allen Plänen das `SC` Signal und wählt dann jeweils unter allen Plänen im Zustand `ready` zufällig einen aus, der das `E` Signal erhält. Weiterführende Details zum Verhalten von Asbru Plänen finden sich in Abschnitt II.

3.3 Asbru und Asbru Light

Die Syntaxdefinition von Asbru [64] beschreibt eine sehr mächtige und umfangreiche Sprache, die noch weit mehr Konstrukte unterstützt, als die in dieser Arbeit vorgestellten Konzepte. Häufig gilt für diese zusätzlichen Konzepte, dass deren intendierte Semantik unklar bzw. Konzepte redundant sind und daher anderweitig ausgedrückt werden können.

Beispielsweise unterstützt Asbru eingebettete Pläne, die es erlauben, statt einer Unterplanbezeichungung einen vollständigen, anonymen Plan als Unterplan anzugeben. Dieses Konzept kann leicht dadurch abgebildet werden, dass der anonyme, eingebettete Plan getrennt spezifiziert wird und einen Namen erhält.

Es wurde daher festgelegt, dass ausschließlich der Teil von Asbru mit einer formalen Semantik unterlegt werden soll, der für die Formulierung der medizinischen Fallstudien notwendig ist. Diese Untermenge von Asbru wird als Asbru Light bezeichnet. Wenn im Folgenden in der Arbeit der Begriff „Asbru“ verwendet wird, so ist stets die formal behandelte Untermenge Asbru Light gemeint.

3.4 Begriffswelt

3.4.1 Starten eines Plans

Ein Plan wird ausgewählt bzw. gestartet, wenn seine Ausführung beginnt. Damit ist gemeint, dass der Plan damit anfängt, den Zustand der Umgebung und der anderen Komponenten der Planhierarchie wahrzunehmen und Entscheidungen basierend auf dieser Wahrnehmung zu treffen. Technisch bedeutet der Start eines Planes, dass er vom Zustand `inactive` in den Zustand `considered` umgesetzt wird, da das `SC` Signal empfangen wurde.

3.4.2 Aktivieren eines Plans

Ein Plan wird aktiviert, wenn alle Überprüfungen auf seine Anwendbarkeit erfolgreich abgeschlossen wurden und der übergeordnete Plan das Aktivierungssignal an den Plan sendet. Damit erhält der Plan die Erlaubnis, die `Selection`-Phase zu verlassen und in die `Execution`-Phase einzutreten. Die Aktivierung eines Planes erfolgt durch Senden des Aktivierungssignales `E` an einen Plan im Zustand `ready`.

3.4.3 Ausführen eines Plans

Ein Plan wird ausgeführt, wenn er von der Evaluierungsphase in die `Execution`-Phase wechselt. Das bedeutet, dass das `E`-Signal empfangen wurde und der Plan aufgrund dessen vom Zustand `ready` in den Zustand `activated` wechselt. Die Ausführung dauert so lange an, wie der Plan entweder im Zustand `activated` oder im Zustand `suspended` ist. Voraussetzung dafür, dass ein Plan ausgeführt werden kann, ist, dass er zuvor aktiviert wurde.

3.4.4 Zurückweisen eines Plans

Ein Plan wird zurückgewiesen, wenn er sich in der `Selection`-Phase befindet und die **setup**- oder **filter**-Bedingung im jeweils relevanten `considered` bzw. `possible` Zustand nicht erfüllt ist. Ist das der Fall, so wechselt er in den Zustand `rejected` und stellt die Ausführung ein.

Zusätzlich kann ein Plan zurückgewiesen werden, wenn der übergeordnete Plan seine Ausführung beendet, solange der Plan sich in der Evaluierungsphase befindet.

3.4.5 Abbruch eines Plans

Ein Plan wird abgebrochen, wenn während der `Execution`-Phase ein kritischer Zustand festgestellt wird, der die weitere Ausführung des Planes unmöglich macht. Dies kann drei Ursachen haben, erstens den Eintritt der **abort**-Bedingung, zweitens die fehlende Erfüllbarkeit der Wartebedingung und drittens die Terminierung des übergeordneten Planes.

In all diesen Fällen wechselt der Plan in den Zustand `aborted`. Hat der übergeordnete Plan den Marker zur Reaktivierung abgebrochener Pläne gesetzt, so kann der Plan neu gestartet werden.

3.4.6 Abschließen eines Plans

Ein ausgeführter Plan kann (erfolgreich) abgeschlossen werden, sofern die **complete**-Bedingung erfüllt ist und alle notwendigen Unterpläne abgeschlossen sind, die in der Wartebedingung spezifiziert sind. Durch den Abschluß stellt der Plan die Terminierung ein und ändert den Planzustand auf `completed` ab.

3.4.7 Unterbrechen bzw. Aussetzen eines Plans

Ein Plan wird unterbrochen bzw. ausgesetzt, wenn er ausgeführt wird und der übergeordnete Plan aussetzt oder aber die **suspend**-Bedingung erfüllt ist. Im ausgesetzten Zustand startet der Plan keine weiteren Unterpläne mehr und kann seine Ausführung nicht mehr abschließen. Es ist nur noch möglich, den Plan abzubrechen oder ihn zu reaktivieren, wenn der übergeordnete Plan nicht mehr ausgesetzt ist und auch die Reaktivierungsbedingung erfüllt ist.

3.5 Formulierung von Asbru Aussagen mit GDL

Ziel dieser Arbeit ist die Verifikation medizinischer Aussagen über dem Modell einer Leitlinie. Um dieses Ziel zu erreichen, muss ein formales, ausführbares Modell der Leitlinie vorliegen. Zusätzlich dazu muss auch ein formales Modell der zu beweisenden Aussage gegeben sein. Um dieses formale Modell der Aussagen zu erzeugen, müssen die vorwiegend als natürlichsprachiger Text vorliegenden Aussagen formalisiert werden. In [67] wird sowohl ein Prozess zur Formalisierung als auch eine formale Sprache zur Definition von medizinischen Aussagen vorgestellt, welche hier verwendet werden soll. Der Prozess beschreibt eine Übersetzung der informellen Aussage in fünf Stufen, der Reduktion, der Normalisierung, der Formalisierung, der Anpassung sowie als letztem Schritt der Übersetzung.

Initial können die zu beweisenden Aussagen in nahezu beliebig „schlechtem“ Zustand vorliegen. Dies rührt daher, dass völlig unterschiedliche Quellen für die Formulierung der Aussagen herangezogen werden. Aufgrund der Tatsache, dass einige Quellen beispielsweise Aussagen so formulieren, dass sie für statistische Vergleiche unterschiedlicher medizinischer Einrichtungen geeignet sind, können die Aussagen mit statistischen Bezugsgrößen „verunreinigt“ sein, die für die formale Verifikation nicht relevant sind. Ziel der Reduktion ist es, solche Artefakte zu entfernen, so dass sie den restlichen Ablauf der Formalisierung nicht stören.

Ziel der Normalisierung ist es, die Aussagen auf ein bestimmtes, einheitliches Format zu bringen. Dabei werden die Aussagen in drei Komponenten zerlegt, die Patientengruppe, für die die Aussage wahr sein soll, der Zeitraum, in dem die Aussage wahr sein soll und das Verhalten, dass innerhalb des angegebenen Zeitraums beobachtet werden soll. Mit dem Ergebnis dieser Strukturierung wird dann die Formalisierung der Aussagen durchgeführt. Ziel der Normalisierung ist es, statt dem informellen Text der ursprünglichen Aussage, Textbausteine zu verwenden, die eine klar definierte Semantik haben. Bei einer formalisierten Aussage handelt

es sich noch immer um natürlichsprachigen Text, der jedoch zumindest teilweise mit formaler Semantik unterlegt ist.

Im nächsten Schritt der Anpassung wird die formalisierte Aussage an das Modell der Leitlinie angepasst. Dabei werden die in der Aussage verwendeten Begriffe auf Variablenbezeichner und Plannamen der in Asbru modellierten Leitlinie angepasst. Ziel dieses Schrittes ist die Aussage in einer Form vorliegen zu haben, so dass sie einerseits zum formalen Modell der Leitlinie passt, andererseits aber auch noch informell verständlich ist. Damit soll ein übergeordnetes Ziel des GDL-Prozesses erreicht werden. Experten der medizinischen Domäne sollen den Vorgang der schrittweisen Formalisierung verfolgen können und so dazu beitragen, dass während der Formalisierung keine Fehler in das Modell der Aussage eingeführt werden.

In einem letzten Schritt wird die Aussage in eine temporallogische Formel übersetzt. Dieser Schritt kann vollständig automatisch geschehen. Damit wird erreicht, dass die Domänenexperten alle kritischen Aspekte der Formalisierung kontrollieren können und der einzige, nicht-kontrollierbare Schritt vollautomatisch geschehen kann.

4 Grundlagen

In diesem Abschnitt werden die Grundlagen zur Verifikation medizinischer Behandlungsleitlinien beschrieben. Dazu beschreibt Abschnitt 4.1 das Beweiswerkzeug KIV. Der danach folgende Abschnitt 4.2 beschreibt die verwendete Temporallogik ITL+. Das Kapitel schließt ab mit Abschnitt 4.3, indem die Verifikation von ITL+-Aussagen mit Hilfe von KIV beschrieben wird.

4.1 Interaktive Verifikation mit KIV

Die Verifikation medizinischer Behandlungsleitlinien, wie sie in dieser Arbeit vorgestellt wird, verwendet die Unterstützung computerbasierter Beweiswerkzeuge. Das hier hauptsächlich verwendet Werkzeug ist das interaktive Beweisunterstützungssystem KIV [57, 58]. Dieses System unterstützt die Verifikation von Prädikatenlogik, dynamische Logik, Logiken höherer Ordnungen und Temporallogik. KIV verwendet den Sequenzenkalkül zur Verifikation von Aussagen. Dieser wird in Abschnitt 4.1.1 vorgestellt. Wie die Beweise mit diesem durchgeführt werden, beschreibt Abschnitt 4.1.2. Im Anschluss daran wird in Abschnitt 4.2 die Temporallogik ITL+ sowie deren Einsatz zur Verifikation medizinischer Behandlungsleitlinien beschrieben.

4.1.1 Sequenzenkalkül

Der Sequenzenkalkül wird beispielsweise in [12] genauer vorgestellt, darum soll er hier nur kurz erläutert werden. Basis des Sequenzenkalküls sind Sequenzen der Form $\Gamma \vdash \Delta$, wobei Γ und Δ Formelmengen darstellen. Die Sequenz wird dabei interpretiert als $\bigwedge \Gamma \rightarrow \bigvee \Delta$. In einer Sequenz $\Gamma \vdash \Delta$ wird Γ als Antezedent bezeichnet, Δ als Sukzedent.

Für den Sequenzenkalkül werden Regeln und Axiome definiert. Dabei können Axiome als Spezialfall von Regeln aufgefasst werden. Eine Regel hat eine Menge von Prämissen und eine Conclusio. Allgemein kann eine Regel wie folgt geschrieben werden:

$$\frac{P_1, P_2, ..., P_n}{C}$$

Dabei sind die P_i die Prämissen, C ist die Conclusio. Eine Regel ist so zu lesen, dass die Conclusio wahr ist, wenn alle Prämissen der Regel wahr sind. Eine Regel heißt invertierbar, wenn aus der Korrektheit der Conclusio folgt, dass alle Prämissen korrekt sind. Ein Axiom ist eine Regel ohne Vorbedingungen.

Die Strategie des Beweises mit dem Sequenzenkalküls ist es, eine Sequenz, die auch Beweisverpflichtung genannt wird, durch Regelanwendungen zu vereinfachen. Dabei wird in Kauf genommen, dass durch die Anwendung einer Regel mehrere Beweisverpflichtungen entstehen können, die getrennt voneinander eruiert werden müssen. Ziel ist es, dass die ursprüngliche Sequenz durch Regelanwendung auf solche Sequenzen zurückgeführt werden kann, die den Conclusionen der Axiome entsprechen.

4.1.2 Beweise in KIV

Das Beweissystem KIV unterstützt den Benutzer bei der Anwendung von Regeln des Sequenzenkalküls. Bei einer gegebenen Sequenz berechnet KIV durch Mustererkennung die Menge der anwendbaren Regeln und gibt diese dem Anwender bekannt. Dieser kann entweder eine der angebotenen Regeln anwenden, oder ein vorher definiertes Theorem in die Sequenz einfügen.

Die Anwendung von Regeln wird in einem sogenannten Beweisbaum graphisch dargestellt. Initial besteht ein Beweisbaum aus einem Knoten, der die ursprüngliche Sequenz repräsentiert. Die Anwendung einer Regel lässt den Beweisbaum „von unten nach oben“ wachsen. Dabei entstehen für jede angewendete Regel so viele neue Blätter, wie die Regel Prämissen hat.

Um einen höheren Automatisierungsgrad der Beweise zu erreichen, wird der sogenannte Simplifier verwendet. Regeln, die einer bestimmten, syntaktischen Form genügen, können als sogenannte Simplifier-Regeln markiert werden. Der Simplifier berechnet, welche der so markierten Regeln zu der aktuell bearbeiteten Sequenz passen und wendet diese automatisch an. Dadurch werden verschiedene Ziele erreicht. Zum Ersten kann auf diese Weise die Sequenz immer in einer kanonischen Form dargestellt werden. Häufig gibt es für Datenstrukturen unterschiedliche, semantisch äquivalente Darstellungsformen. Um als Mensch eine Sequenz schneller verstehen zu können, ist es hilfreich, wenn alle Vorkommen eines Datentyps in einer Sequenz in der gleichen kanonischen Form vorliegen. Ein Beispiel dafür sind Mengen mit Elementen auf denen eine Ordnung definiert ist. Solche Mengen sind leichter verständlich, wenn alle Elemente der Menge alphabetisch geordnet sind. Ein zweites Ziel, das der Simplifier verfolgt, ist es, einfache Prämissen einer Regel automatisiert soweit zu vereinfachen, dass diese ohne weitere Interaktionen bewiesen werden kann. Ist dies möglich, wird diese Prämisse dem Benutzer gar nicht mehr angezeigt. Ein drittes Ziel ist die automatische Vereinfachung von Sequenzen. Ein häufiges Phänomen beim Führen von Beweisen ist, dass innerhalb eines Beweises (oder mehrerer Beweise in der gleichen Domäne) immer wieder eine identische Folge von Regelanwendungen notwendig ist, um den Beweis zu schließen. In solchen Fällen kann der Simplifier verwendet werden, diese Regelfolge automatisch anzuwenden, so dass die Zahl der notwendigen manuellen Schritte zurückgeht.

4.2 Die Temporallogik ITL+

In dieser Arbeit wird eine Intervalltemporallogik namens ITL+ [8] verwendet. Diese setzt auf den Grundlagen von ITL [16, 49] auf, erweitert diese jedoch um eine explizite Modellierung der Umgebung reaktiver Systeme. Während bei ITL jeweils zwei Schritte des Systems aufeinanderfolgen, besteht ein temporallogischer Schritt in ITL+ aus einem System- und einem Umgebungsschritt. Dies wird in Abbildung 4.1 dargestellt.

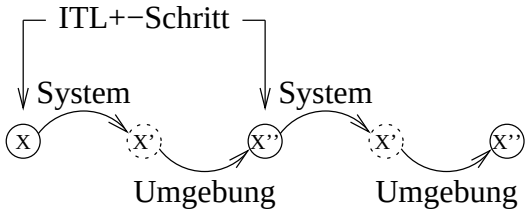


Abbildung 4.1: Zustandsübergangssystem in ITL+

In dieser Abbildung ist das Zustandsübergangsmodell von ITL+ abgebildet. Der Ausgangs-

zustand eines Schrittes wird mit ungestrichenen Variablen beschrieben, was in der Abbildung durch X angedeutet wird. Das System beschreibt eine Transition von den ungestrichenen Variablen X zu den einfach gestrichenen Variablen X′. Nach dem Systemschritt wird ein Umgebungsschritt ausgeführt, der die einfach gestrichenen Variablen X′ überführt in die zweifach gestrichenen Variablen X′′. Die zweifach gestrichenen Variablen eines Schritts stellen gleichzeitig die ungestrichenen Variablen des nächsten Zustands dar.

Die Umgebung ist in der Lage, den Zustand zwischen zwei Systemschritten nach Belieben zu ändern. Um diese Freiheit einzuschränken, ist es möglich, Annahmen an die Umgebung zu formulieren, so dass diese den Zustand nur begrenzt verändern kann. Es ist möglich, der Umgebung gänzlich zu verbieten bestimmte Variablen zu ändern. Die Umgebung kann auch gezwungen werden, bestimmte Änderungen der Variablenwerte durchzuführen. Solche Forderungen an die Umgebung werden als Umgebungsannahme bezeichnet.

Die explizite Berücksichtigung der Umgebung bedeutet, dass die parallel Operatoren kompositional sind. Dies ergibt sich daraus, dass für ein System nicht unterscheidbar ist, ob eine Zustandsänderung durch die Umgebung oder ein paralleles System durchgeführt wurden. Dazu soll ein Beispiel betrachtet werden:

$$Y = 0, X = 0, [X := 3; \alpha \parallel_s Y := 4], X'' = X', Y'' = Y'$$

Das Beispiel beschreibt zwei synchron parallel ausgeführte Programme. Innerhalb eines Systemschrittes wird der Wert der Variablen X auf 3 gesetzt, also X′ = 3 und der Wert der Variablen Y auf den Wert 4, also Y′ = 4. Die Umgebungsannahme verlangt, dass die Umgebung die Werte von X und Y unverändert bleiben sollen. Vor Ausführen des Schritts waren beide Variablen auf den Wert 0 gesetzt.

Der erste Prozess kennt nur seine lokalen einfach gestrichenen Variablen, nicht die globalen einfach gestrichenen Variablen. Aus seiner lokalen Sicht kann der Prozess daher nicht entscheiden, ob der Zustand von Y durch einen parallelen Prozess oder die „echte“ Umgebung verändert werden. Aus Sicht dieses Prozesses ist also folgende Modellierung von der obigen nicht unterscheidbar:

$$Y = 0, X = 0, [X := 3; \alpha], X'' = X', Y'' = 4$$

Dadurch ist es möglich, zu einem Prozess parallele Prozesse in dessen Umgebung zu verschieben. Das ermöglicht die Anwendung von Rewriting von Formeln und Programmen unterhalb eines parallel-Operators. Im Gegensatz etwa zu TLA [39] gilt die Kompositionalität unabhängig davon, ob Formeln in Normalformen angegeben sind.

Im Folgenden sollen die Grundlagen der verwendeten Logik soweit vorgestellt werden, wie dies für das Verständnis der Verifikation von Asbru notwendig ist. Zunächst wird in Abschnitt 4.2.1 die Semantik von ITL+ vorgestellt, wie sie bei [8] definiert wird. Im Anschluss daran werden in Abschnitt 4.2.2 die ITL+-Operatoren definiert. Abschnitt 4.2.4 schließt mit einer Betrachtung der Parallelität in ITL+ ab.

4.2.1 ITL+-Semantik

Temporallogische Formeln in ITL+ werden über Zustandsabfolgen ausgewertet. Diese werden auch Intervalle genannt. Wie in ITL werden in ITL+ endliche sowie unendliche Zustandsfolgen berücksichtigt. In ITL besteht ein Intervall aus einzelnen Zuständen. Im Gegensatz dazu besteht ein Intervall in ITL+ aus Zustandspaaren. Die Transitionen zwischen den Zuständen eines Zustandspaares sind die Umgebungsübergänge, Transitionen zwischen Zustandspaaren sind die Systemübergänge. Formal wird die wie folgt gefasst:

Definition

Ein Intervall I hat die Form
I = (σ_0, σ'_0, σ''_0, ..., $\sigma'_{\overline{n}}$, $\sigma''_{\overline{n}}$)
Intervalle bestehen aus einem initialen Zustand σ_0 und einer endlichen oder unendlichen Zustandsübergangsfolge $(\sigma'_i, \sigma''_i)_{i=0}^{\overline{n}}$. Dabei gilt, dass $\sigma''_i = \sigma_{i+1}$.
Der Zustandsübergang von σ'_i nach σ''_i wird Umgebungsübergang genannt, der Zustandsübergang von σ_i nach σ'_i Systemübergang.
Dabei ist $\overline{n}$ aus der Menge der natürlichen Zahlen vereinigt mit Unendlich.

Für Intervalle werden Zugriffsfunktionen definiert. Diese sind notwendig, um zu definieren, wann ein Intervall Modell einer temporallogischen Formel ist. Die temporallogischen Operatoren werden im Anschluss an die Semantikdefinition in Abschnitt 4.2.2 vorgestellt und definiert.

Definition

Für ein Intervall I mit I = (σ_0, σ'_0, σ''_0, ..., $\sigma'_{\overline{n}}$, $\sigma''_{\overline{n}}$) gilt:
Die Länge des Intervalls | I | ist definiert ist als | I | = $\overline{n}$.
I ist ein unendliches Intervall, falls $\overline{n} = \infty$,
I ist ein endliches Intervall, falls $\overline{n} \neq \infty$.
I ist ein leeres (endliches) Intervall, falls $\overline{n} = 0$.

In jedem Fall gilt, dass $\sigma"_n = \sigma_{n+1}$, sofern $n + 1 \leq \overline{n}$
Für den Zugriff auf Zustände des Intervalls I sind folgende Funktionen definiert:

$$
\begin{aligned}
I(i) &= \begin{cases} \sigma_i, \text{falls } i \leq \overline{n} \\ \sigma_{\overline{n}}, \text{sonst} \end{cases} \\
I(i)' &= \begin{cases} \sigma'_i, \text{falls } i \leq \overline{n} \\ \sigma_{\overline{n}}, \text{sonst} \end{cases} \\
I(i)'' &= I(i+1) \\
\mathrm{I}|_i &= \begin{cases} (\sigma_i, ...), \text{falls } i \leq \overline{n} \\ (\sigma_{\overline{n}}), \text{sonst} \end{cases} \\
\mathrm{I}|^i &= \begin{cases} (\sigma_0, ..., \sigma_i), \text{falls } i \leq \overline{n} \\ (\sigma_0, ..., \sigma_{\overline{n}}), \text{sonst} \end{cases} \\
\mathrm{I}|_i^j &= (I|^j)|_i \\
I(X) &= I(0)(X)
\end{aligned}
$$

Eine Konvention für die Modellierung reaktiver Systeme in ITL+ besagt, dass das eigentliche System in Form eines (parallelen) Programms spezifiziert wird, die Umgebung in Form von ITL+-Aussagen. Programme stellen immer Beziehungen zwischen ungestrichenen und einfach gestrichenen Zuständen her, während ITL+-Aussagen beliebige Beziehungen zwischen ungestrichenen, einfach und zweifach gestrichenen Variablen herstellen können.

4.2.2 TL-Operatoren

ITL+ definiert und verwendet eine Reihe temporallogischer Operatoren. Verwendet werden Operatoren der ITL sowie der LTL-Logik.

ITL-Operatoren

Die folgenden ITL-Operatoren sind aus [49] entnommen:

Definition

$I \models \psi;\varphi$ gdw. es existiert ein n, $n \leq |\, \mathrm{I}\, |$, so dass $\mathrm{I}|_{\mathrm{n}} \models \varphi$ und $\mathrm{I}|^{\mathrm{n}} \models \psi$
oder $|\, \mathrm{I}\, | = \infty$ und $I \models \psi$
$I \models \mathbf{step}$ gdw. $|\, \mathrm{I}\, | = 1$
$I \models \mathbf{more}$ gdw. $I \models \mathbf{step};\mathbf{true}$
$I \models \mathbf{last}$ gdw. $I \models \neg\, \mathbf{more}$
$I \models \varphi^{\star}$ gdw. $|\, \mathrm{I}\, | = 0$
oder es existieren $0 = \mathrm{n}_0 < \mathrm{n}_1 < ... < \mathrm{n}_{\mathrm{m}} <|\, \mathrm{I}\, |$
mit $\mathrm{I}|_{\mathrm{n_i}}^{\mathrm{n_{i+1}}} \models \varphi$ für alle i mit $0 \leq i < m$
und $\mathrm{I}|_{\mathrm{n_m}} \models \varphi$
oder $|\, \mathrm{I}\, | = \infty$
und es existieren unendliche viele $0 = \mathrm{n}_0 < \mathrm{n}_1 < ...$
mit $\mathrm{I}|_{\mathrm{n_i}}^{\mathrm{n_{i+1}}} \models \varphi$ für alle i mit $0 \leq i$

Der chop-Operator $\psi;\varphi$ teilt ein Intervall in zwei Intervalle auf, wobei in der ersten Intervallhälfte ψ gelten muss, in der zweiten Intervallhälfte φ. Damit fällt der chop-Operator mit der Definition der sequentiellen Komposition von Programmen zusammen und kann in der Logik verwendet werden, um diese zu definieren. Der **step**-Operator verlangt, dass ein Intervall genau einen Schritt lang ist. Dieser Operator kann beispielsweise mit dem chop-Operator kombiniert werden, um festzulegen, dass die erste Komponente eines chop genau einen Schritt andauern soll. **more** und das Gegenstück **last** werden verwendet, um prüfen zu können, ob ein Terminierungsfall vorliegt. Der $\varphi^{\star}$-Operator beschreibt potentiell unendliche Zyklen und kann für die Definition der Semantik der **while** Schleife verwendet werden.

LTL-Operatoren

Zusätzlich zu den ITL-Operatoren werden für die Verifikation die klassischen LTL-Operatoren verwendet. Deren Semantik wird gemäß [8] definiert. Dazu wird zunächst die Semantik des **until**-Operators definiert und die übrigen Operatoren von diesem abgeleitet.

Definition

$I \models \varphi\ \mathbf{until}\ \psi$ gdw. es existiert ein n, so dass $n \leq |\, \mathrm{I}\, |$
mit $\mathrm{I}|_{\mathrm{n}} \models \psi$ und $\mathrm{I}|_{\mathrm{m}} \models \varphi$, für alle $0 \leq m < n$

Der **until**-Operator verlangt, dass in einem Intervall so lange die Formel φ gilt, bis mindestens einen Schritt lang ψ gilt. Der wesentliche Unterschied zwischen dem **until**- und dem

chop-Operator erschließt sich bei Betrachtung temporallogischer φ und ψ. Während in der Formel $\varphi;\psi\ \varphi$ tatsächlich nur über den ersten Teil des geteilten Intervalls ausgewertet wird, müssen im Fall von φ **until** ψ bei der Auswertung von φ möglicherweise auch Zustände berücksichtigt werden, die nach dem Eintreten von ψ stattfinden.

Dieser Unterschied wird am Beispiel verdeutlicht. Werden die Formeln $(\Box\ \varphi)$ **until** ψ und $(\Box\ \varphi);\psi$ verglichen, und dabei die unten definierte Semantik für $\Box$ verwendet, dann fordert die erste Formel, dass φ tatsächlich immer gilt, also auch, nachdem ψ bereits wahr wurde. Im zweiten Fall wird das Intervall dagegen geteilt. In der ersten Komponente muss immer φ gelten, in der zweiten Hälfte ψ.

Die meisten übrigen TL-Operatoren werden vom **until**-Operator abgeleitet:

Definition

$$
\begin{array}{lcl}
\Diamond\ \varphi & := & \textbf{true until}\ \varphi \\
\Box\ \varphi & := & \neg \Diamond \neg\ \varphi \\
\varphi\ \textbf{unless}\ \psi & := & \varphi\ \textbf{until}\ \psi\ \vee\ \Box\ \ \varphi \\
\circ\ \varphi & := & \textbf{step};\varphi \\
\bullet\ \varphi & := & \neg \circ\ \neg\ \varphi
\end{array}
$$

Die Operatoren haben die übliche Semantik. Das heißt, der $\Diamond$-Operator verlangt, dass ein Ereignis nach endlich vielen Schritten eintritt, wobei die Zahl der Schritte nicht angegeben wird. Der $\Box$-Operator verlangt, dass eine Formel immer gilt, **unless** ist die abgeschwächte Variante von **until**, bei der es möglich ist, dass die zweite Formel nie wahr wird, wenn die erste dafür immer wahr ist. $\circ$ verlangt, dass die Formel im nächsten Schritt wahr ist und dieser Schritt existiert und $\bullet$ verlangt, dass die Formel im nächsten Schritt gilt, sollte ein solcher existieren.

4.2.3 Simple Programming Language in KIV

In ITL+ sind Programme und Formeln austauschbar. Systemtransitionen können sowohl durch Programmkonstrukte als auch durch Formeln beschrieben werden, wobei alle Programmkonstrukte durch Formeln ausgedrückt werden können. Die Programmkonstrukte orientieren sich an der *Simple Programming Language*, die beispielsweise in [42] definiert wird. Zusätzlich zu den dort vorgestellten Programmkonstrukten wird für die Verifikation von Asbru ein Programmkonstrukt **itlif** benötigt. Dieses entspricht weitgehend einem normalen **if**, führt jedoch die erste Anweisung nach dem Konditional atomar zusammen mit der Auswertung des Konditionals aus.

Die Semantik der Programmkonstrukte wird durch Übersetzung in äquivalente TL Formeln angegeben. Dazu ist eine Hilfsfunktion, die sogenannte Rahmenannahme notwendig. Diese gibt an, dass ein Systemschritt nur eine Menge bezeichneter Variablenbelegungen abändert und den Rest unverändert lässt. Somit stellt die Rahmenannahme eine potentiell unendliche Formel dar, da sie für alle (potentiell unendlichen) dynamischen Variablen eine Gleichheit zwischen ungestrichenem und einfach gestrichenem Zustand verlangt.

Definition

Für eine Menge von Variablen V gilt:

I $\models\lceil$V$\rceil$ gdw. für alle Variablen X gilt: I(X)' = I(X), wenn X nicht in V

Die Rahmenannahme ist notwendig, um den Ausschluss weiterer Änderungen zu formulieren. Beispielsweise macht eine Zuweisung `x := 3` die implizite Zusicherung, dass Werte für weitere Variablen `y, z`, ... unverändert bleiben. Dies kann im Allgemeinen nicht mit endlichen Formeln aufgeschrieben werden. Mit der Rahmenannahme ist es möglich, die Semantik der SPL-Programmkonstrukte festzulegen. Dabei werden nur die Konstrukte hier aufgeführt, die auch zur Definition der Asbru-Prozeduren benötigt werden.

Definition

Die in Asbru verwendeten SPL Programmkonstrukte haben folgende Semantik

$$
\begin{array}{lcl}
x := e & :\equiv & x' = e \wedge \lceil \mathrm{x} \rceil \wedge \circ \textbf{ last} \\
\textbf{skip} & :\equiv & \lceil\rceil \wedge \circ \textbf{ last} \\
\textbf{itlif } \psi \textbf{ then } \varphi_1 \textbf{ else } \varphi_2 & :\equiv & \psi \rightarrow \varphi_1 \wedge \neg \psi \rightarrow \varphi_2 \\
\textbf{itlif } \psi \textbf{ then } \varphi & :\equiv & \textbf{itlif } \psi \textbf{ then } \varphi \textbf{ else skip}
\end{array}
$$

Für die Implementierung der Asbru-Semantik mittels SPL werden drei Programmkonstrukte verwendet. Das erste davon ist die Zuweisung, `x := e`. Durch diese wird der Variablen `x` der Wert des Terms e zugewiesen. Die Semantik dieses Programmkonstrukts legt fest, dass die Zuweisung genau einen temporallogischen Schritt dauert und neben der Zuweisung selbst keine Seiteneffekte hat. Weiterhin verwendet wird **skip**. Dabei handelt es sich um eine nicht-Handlung, das heißt, die Ausführung eines **skip** führt nicht zu Veränderungen des Zustands. Das letzte verwendete Konstrukt ist die bedingte Entscheidung **itlif**. Dieses entscheidet anhand eines Tests, mit welchem der Programme die Ausführung fortfahren soll. Der Unterschied eines **itlif** zu dem üblichen **if** ist dabei, dass letzteres für die Auswertung der Bedingung einen Schritt benötigt, während das **itlif** Auswertungen des Tests in Nullzeit vornimmt.

Weiterhin bei der Ausführung von Asbru verwendet werden parallele Zuweisungen. Mehrere Zuweisungen werden parallel ausgeführt, wenn diese mit Kommas voneinander getrennt werden. In Asbru werden zweifach- und dreifach parallele Zuweisungen verwendet:

Definition

Parallele Zuweisungen in Asbru haben folgende Semantik:

$$
\begin{array}{lcl}
x := e, y := f & :\equiv & x' = e \wedge y' = f \wedge \lceil \mathrm{x, y} \rceil \wedge \circ \textbf{ last} \\
x := e, y := f, z := g & :\equiv & x' = e \wedge y' = f \wedge z' = g \wedge \lceil \mathrm{x, y, z} \rceil \wedge \circ \textbf{ last} \\
w := d, x := e, y := f, z := g & :\equiv & w' = d \wedge x' = e \wedge y' = f \wedge z' = g \wedge \lceil \mathrm{x, y, z, w} \rceil \wedge \circ \textbf{ last}
\end{array}
$$

Die Parallele Zuweisung führt also gleichzeitig alle angegebenen Zuweisungen aus.

4.2.4 Parallelität in ITL+

Die Semantik von Asbru ist parallel synchron, da die Zustandsübergänge aller Pläne jeweils gleichzeitig stattfinden. Um dies in das Beweissystem abzubilden, muss entsprechende

Unterstützung geschaffen werden. Ursprünglich war im Beweiswerkzeug KIV lediglich Interleaving implementiert, also die abwechselnde Ausführung von Prozessen. Ein Versuch, die synchron parallele Semantik von Asbru mit Interleaving zu implementieren scheitert an der Komplexität der entstehenden Programmdefinitionen, die erzwingen müssen, dass die einzelnen Prozesse, die die Asbru-Pläne repräsentieren, jeweils gleichmäßig ausgeführt werden. Dazu muss die Frage der Schreibkonflikte gesondert behandelt werden, was spezielle und komplexe Datentypen zur Folge hätte.

Schreibkonflikte

Eine parallel synchrone Semantik muss Schreibkonflikte berücksichtigen. Schreibkonflikte treten immer dann auf, wenn zwei Prozesse gleichzeitig versuchen, den Wert einer Variablen zu verändern. Dabei ist besonders zu beachten, dass der Umgang mit Schreibkonflikten in Abhängigkeit von der Fallstudie unterschiedlich gehandhabt werden kann. Beispielsweise kann ein Schreibkonflikt durch zwei Zuweisungen dazu führen, dass beliebige Werte geschrieben werden, das System abstürzt oder sich eine der Zuweisungen durchsetzt. Dazu ist auch wichtig zu betrachten, wann zwei Schreibzugriffe auf eine Variable überhaupt als Konflikt gelten. Bei komplexeren Datentypen, etwa Datenfeldern, Speichern oder dynamischen Funktionen, kann es sein, dass ein gleichzeitiger Schreibzugriff auf eine Variable nicht als Schreibkonflikt angesehen werden soll.

In der Asbru Fallstudie gibt es beispielsweise eine Datenstruktur, die die Zustände aller Pläne speichert. Diese Datenstruktur wird gleichzeitig von allen laufenden Plänen aktualisiert, jedoch überschneidungsfrei in dem Sinne, dass jeder Plan nur seinen eigenes Datenfeld beschreibt. Dies wird in der Fallstudie nicht als Schreibkonflikt angesehen. Gleichwohl könnte ein solches Verhalten an anderer Stelle als fehlerhaft gewertet werden.

Um diese unterschiedlichen Anforderungen alle erfüllbar zu machen, wurde das überladene Prädikat `sync` eingeführt, dass für jeden Datentyp definiert sein muss, auf den parallele Schreibzugriffe erfolgen. Dieses Prädikat beschreibt, unter welchen Umständen gleichzeitige Updates auf einen Datentyp zulässig sind und welches Berechnungsergebnis dadurch entsteht. `sync` wird über vier Variablen gleicher Typisierung ausgewertet. Informell entspricht der erste Wert dem Zustand vor Ausführung des Schrittes, der zweite Wert dem Berechnungsergebnis des ersten Prozesses, der dritte Wert dem Berechnungsergebnis des zweiten Prozesses und der vierte Wert dem Gesamtergebnis.

Diese Zusammenhänge sind dargestellt in der Abbildung 4.2. Dabei ist die Notation so zu lesen, dass das synchron parallele System $I_1 \|_s I_2$ einen Übergang zu $I_1' \|_s I_2'$ machen soll. Ausgehen soll der Übergang vom Zustand σ, die Zustandsänderung, die durch den Schritt ausgelöst wird, wird durch σ' beschrieben. Rekursiv wird dieses Problem nun Zunächst auf die Schritte von I_1 bzw. I_2 zurückgeführt, die jeweils Übergänge σ_1, σ_1' und σ_2, σ_2' durchführen. Dabei entsteht dann das globale σ' durch Anwendung des Prädikates `sync`, im Beispiel als `sync`$(\sigma, \sigma_1', \sigma_2', \sigma')$.

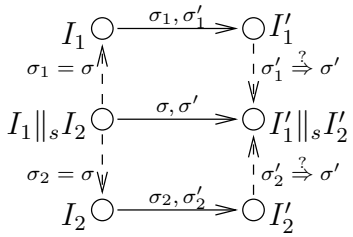


Abbildung 4.2: Synchron parallele Ausführung

In der Asbru Fallstudie sind dynamische Funktionen die wesentlichen Datenstrukturen, auf die parallel geschrieben wird. Diese ordnen einem Identifikator ein Datum zu, beispielsweise der Asbru-Zustand, in dem der Identifikator der Planname ist, das zugehörige Datum der Zustand, in dem sich der Plan befindet. Laufen nun mehrere Pläne parallel, so ist davon auszugehen, dass mehrere Pläne gleichzeitig ihren eigenen Zustand abändern wollen. Dies soll durch die parallele Ausführung auch nicht unterbunden werden. So lange also jeder Plan nur genau sein eigenes Datenfeld aktualisiert, sollen diese Zugriffe keine Konflikte verursachen.

Entsprechend wird das Prädikat `sync` für den Asbru-Zustand so definiert, dass mehrfache Updates auf eine dynamische Funktion, sie sich nicht überschneiden, als konfliktfrei angesehen werden. Wird also jedes Feld gar nicht oder höchstens von einem der beiden Prozesse beschrieben, so wird der Ergebniszustand so berechnet, dass die Zuweisungen jeweils ausgeführt werden.

Definition paralleler Ausführung

Basierend auf der Ausführung sequentieller Programme, wie sie in [8] durchgeführt wurde und mit der bereits erfolgten Definition einer Möglichkeit, mit konkurrierenden Schreibzugriffen umgehen zu können, ist die synchron parallele Ausführung wie folgt definiert.

Definition

Die synchron parallele Ausführung wird durch die nachfolgenden drei Regeln definiert. Diese unterscheiden die drei möglichen Fälle, die sich aus Terminierung und Ausführung ergeben.

Keiner der Prozesse terminiert

$$\frac{I_1 \xrightarrow{\sigma,\sigma_1'} I_1',\ I_2 \xrightarrow{\sigma,\sigma_2'} I_2',\ \mathrm{sync}(\sigma,\ \sigma_1',\ \sigma_2',\ \sigma')}{I_1 \|_s I_2 \xrightarrow{\sigma,\sigma'} I_1' \|_s I_2'}$$

Der erste Prozess terminiert

$$\frac{I_1 \xrightarrow{\sigma,\sigma_1'} \emptyset,\ I_2 \xrightarrow{\sigma,\sigma_2'} t',\ \mathrm{sync}(\sigma,\ \sigma_1',\ \sigma_2',\ \sigma')}{I_1 \|_s I_2 \xrightarrow{\sigma,\sigma'} t'}$$

Der zweite Prozess terminiert

$$\frac{I_1 \xrightarrow{\sigma,\sigma_1'} t',\ I_2 \xrightarrow{\sigma,\sigma_2'} \emptyset,\ \mathrm{sync}(\sigma,\ \sigma_1',\ \sigma_2',\ \sigma')}{I_1 \|_s I_2 \xrightarrow{\sigma,\sigma'} t'}$$

Prinzipiell sind bei der Ausführung von zwei synchron parallelen Prozessen vier Fälle zu unterscheiden. Jeder der Prozesse kann terminieren oder in der Ausführung fortschreiten. Die erste SOS Regel beschreibt den Fall, in dem beide Prozesse nicht terminieren und Schritte durchführen. In diesem Fall muss für beide Prozesse getrennt berechnet werden, welches Ausführungsergebnis sich ergibt. Aus den beiden Ergebnissen wird dann durch das `sync`-Prädikat der globale eingestrichene Zustand berechnet. Die zweite Regel beschreibt den Fall,

in dem der erste Prozess terminiert. In diesem Fall bestimmt der zweite Prozess durch seinen Zustandsübergang den globalen Zustandsübergang. Allgemein gilt für die Semantik von ITL+, dass für den letzten Zustandsübergang $\sigma_{\overline{n}}, \sigma'_{\overline{n}}, \sigma''_{\overline{n}}$ gilt, dass $\sigma_{\overline{n}} = \sigma'_{\overline{n}} = \sigma''_{\overline{n}}$. In der Regel wird damit die Auswertung des `sync`-Prädikat vereinfacht. Da in diesem Fall $\sigma'_2 = \sigma$ gilt, muss `sync`$(\sigma, \sigma'_1, \sigma, \sigma')$ berechnet werden. Da `sync` dazu dienen soll, Schreibkonflikte zu erkennen, wird für die Mehrzahl der sinnvollen Definitionen für `sync` in diesem Fall gelten, dass $\sigma'_1 = \sigma'$. Analog zu diesem Fall verhält sich der durch die dritte Regel beschriebene Fall, in dem der zweite Prozess terminiert und der Zustandsübergang durch den ersten Prozess bestimmt wird.

Zwei Anforderungen an den Operator zur Ausführung synchron-paralleler Prozesse sind die Kommutativität und die Assoziativität. Im Allgemeinen sollte gelten, dass die Ausführung von $I_1 \parallel_s I_2$ zu den gleichen Ergebnissen führt, wie die von $I_2 \parallel_s I_1$. Gleichermaßen sollte die Ausführung von $(I_1 \parallel_s I_2) \parallel_s I_3$ und $I_1 \parallel_s (I_2 \parallel_s I_3)$ zu jeweils gleichen Ergebnissen führen. Die Semantikdefinition der synchron-parallelen Ausführung steht dem nicht im Weg, jedoch ist für das `sync`-Prädikat zu beweisen, dass es die Kommutativität und Assoziativität achtet. Diese Aspekte werden detailliert in [11] beschrieben.

4.3 Verifikation temporallogischer Aussagen in KIV

In diesem Abschnitt wird die Beweistechnik nach [8] zum Beweis temporallogischer Aussagen vorgestellt. Die Beweistechnik setzt auf vier Komponenten auf. Die erste Komponente ist die symbolischen Ausführung, die Ergebnisse temporallogischer Schritte berechnet. Diese wird in Abschnitt 4.3.1 vorgestellt. Im daran anschließenden Abschnitt 4.3.2 wird die Sequenzierung beschrieben, die ähnliche Aufgaben erfüllt wie Techniken zur partial order reduction. Um mit unendlichen Intervallen umgehen zu können, wird eine induktive Beweistechnik benötigt, die in Abschnitt 4.3.3 vorgestellt wird. Durch die Kompositionalität der Parallel-Operatoren ist es möglich Programme und Formeln unterhalb der Parallel-Operatoren abzuschwächen. Dies wird Abstraktion genannt und in Abschnitt 4.3.4 beschrieben.

4.3.1 Symbolische Ausführung

Bei der symbolischen Ausführung werden Systeme schrittweise abgearbeitet. Dazu wird zunächst jede temporallogische Formel in Normalform gebracht, welche die Transitionen des nächsten Schrittes sowie die nach diesen Transitionen entstehenden Restsysteme beschreibt. Dabei wird der erste Schritt als Relation zwischen ungestrichenen, einfach und zweifach gestrichenen Variablen angegeben. Zu beachten ist hierbei, dass jede ITL+-Aussage in Normalform gebracht werden kann.

Als Beispiel wird das Programm `X := 3;` α in Normalform gebracht. Die Semantikdefinition legt fest, dass die Zuweisung `X := 3` genau einen Schritt zur Abarbeitung benötigt. In diesem Schritt wird `X'` der Wert 3 zugewiesen. Nach Abarbeitung dieses Schritts verbleibt ein Restsystem, dass durch das Programm α beschrieben wird. Die Normalform dieser Formel lautet wie folgt:

$$X' = 3 \ \wedge \ \lceil \mathrm{X} \rceil \ \wedge \ \circ \alpha$$

In diesem Beispiel gibt es nur eine mögliche Systemtransition. Häufig kommt es auch vor, dass mehrere Transitionen möglich sind, beispielsweise bei **itlif**-Fallunterscheidungen. Für das Programm **itlif** ε **then** α **else** β beispielsweise gibt es zwei Möglichkeiten der Auswertung, die vom Wahrheitswert des Tests ε abhängig sind.

Allgemein führt die Berechnung der Möglichkeiten der Ausführung temporallogischer Formeln und Programme zu folgender Normalform:

$$\tau_0 \wedge \mathbf{last} \vee (\bigvee_{i=1}^{n} \exists \tilde{X}_{\mathrm{i}}.\tau_i \wedge \circ\, \psi_i)$$

Das heißt, im Allgemeinen existiert eine Transition, die den Terminierungsfall beschreibt sowie eine Menge von Transitionen, in denen das System nicht terminiert.

Symbolische Ausführung besteht nun darin, die Fallunterscheidungen dieser Normalform aufzubrechen und temporallogisch einen Schritt abzuarbeiten, also die nicht- und einfach gestrichenen Zustände der Variablen zu speichern und die Werte der zweifach gestrichenen Variablen an die ungestrichenen Variablen zu binden. Danach werden die $\circ$ -Operatoren der Restsysteme in Normalform entfernt.

(Abschwächende) Simplifikation

In ihrer Effizienz wird die symbolische Ausführung durch prädikatenlogische Simplifikation unterstützt. Können etwa bei der Berechnung der Normalform die τ_i-Transitionen als widersprüchlich erkannt werden, muss die Normalformberechnung in diesen Fällen nicht komplett zu Ende gebracht werden. Beispielsweise wird bei der Normalformberechnung eines **itlif** Konstrukts versucht, anhand des globalen Zustands die Bedingung des **itlif** auszuwerten. Wenn dies gelingt muss nur ein Ast der Auswertung weiter verfolgt werden.

Bei Ausführung des Schrittes werden in der Regel neue Variablen eingeführt, die die „alten“ Variablenwerte speichern. Das führt zu einer recht großen Zahl von Variablen auf der Sequenz, die die Übersichtlichkeit einschränken und die Effizienz automatischer Simplifikation und Normalformberechnung einschränken. Daher enthält die Simplifikation ein Konzept zur Eliminierung von Variablen eingeführt. Eine Heuristik versucht unnötige Variableninformationen zu erkennen. Enthält die Sequenz beispielsweise eine Gleichung `X = 3`, so werden unter bestimmten Umständen alle übrigen Vorkommen von `X` durch den Wert 3 zu ersetzt. Dadurch wird die Variable `X` eliminiert.

Diese Simplifikationsregeln werden für Asbru angepasst, so dass Asbru-Beweisverpflichtungen mit den gleichen Mechanismen vereinfacht werden können, wie die Basis-Datentypen.

4.3.2 Sequenzierung

In temporallogischen Beweisen mit parallelen Programmen kommt es häufig zu der Situation, dass bestimmte, identische Beweisziele auf mehreren Wegen erreicht werden können. Dies kann mehrere Ursachen haben. Beispielsweise kann sich nach einem temporallogischen Schritt herausstellen, dass durchgeführte Fallunterscheidungen unnötig sind, da die Beweis-Strategie in den entstehenden Fällen jeweils identisch ist.

In solchen Fällen ist es möglich die verschiedenen entstehenden Beweisziele zusammenzufassen. Voraussetzung dafür ist, dass die Sequenzen hinreichend ähnlich sind. Sind zwei Sequenzen gegeben die zusammengefasst werden sollen, muss eine auf die andere abgebildet werden, was bedeutet, dass nachgewiesen werden muss, dass die Vorbedingung der einen Sequenz nicht schwächer und gleichzeitig die Nachbedingung nicht stärker ist als die Vor- und Nachbedingungen der anderen Sequenz.

Sind also $\Gamma_1 \vdash \Delta_1$ und $\Gamma_2 \vdash \Delta_2$ zwei Sequenzen, so dass gilt, dass die Vorbedingung Γ_1 schwächer ist als Γ_2 und gleichzeitig die Nachbedingung Δ_1 stärker als Δ_2, das heißt, gilt $\Gamma_1 \rightarrow \Gamma_2$ und zusätzlich $\Delta_2 \rightarrow \Delta_1$, dann wird mit dem Nachweis der Korrektheit der Sequenz $\Gamma_2 \vdash \Delta_2$ automatisch die Gültigkeit von $\Gamma_1 \vdash \Delta_1$ nachgewiesen.

In dem hier verwendeten ITL Kalkül ist diese Art der Beweisvereinfachung unter der Bezeichnung „Insert Proof Lemma“ eingebaut. Dabei wird nach Anwenden der Regel versucht,

automatisch festzustellen, ob die Implikationen von Antezedenten und Sukzedenten gilt. Gegebenenfalls muss diese manuell bewiesen werden.

4.3.3 Temporallogische Induktion

Die Semantik von ITL+ ist definiert mit Hilfe von potentiell unendlichen Intervallen. Damit ein Kalkül mit unendlichen Intervallen umgehen kann, muss eine Induktionstechnik vorhanden sein. Im vorliegenden Fall wird die Technik der VD-Induktion verwendet. Die Anwendung der VD-Induktion verlangt, dass ein Term angegeben wird, der jeweils vom Start der Induktion bis zur Anwendung der Induktionshypothese abnimmt. Dabei sind nur Terme zugelassen, die keine unendlich absteigenden Ketten erlauben. Ein Beispiel für einen gültigen Term ist eine Variable vom Typ der natürlichen Zahlen.

Die VD-Induktion erlaubt statt der expliziten Angabe von Termen auch den Start einer Induktion, wenn $\Diamond$ Aussagen im Antezedenten oder $\Box$ Aussagen im Sukzedenten existieren. Die Rechtfertigung dafür ist, dass $\Diamond$ Formeln verlangen, dass ein Ereignis nach endlich vielen Schritten eintritt. Die Induktion wird über die Zahl der Schritte geführt, bis dieses Ereignis eintritt. Induktion über die Aussagen im Sukzedenten ist aufgrund der Dualität zwischen $\Box$ und $\Diamond$ möglich. Gemäß der Definition der Semantik von $\Box$ gilt, dass $\Box\ \varphi \leftrightarrow \neg \Diamond\ \neg\ \varphi$. Damit kann eine $\Box$-Formel im Sukzedenten analog zu einer $\Diamond$ Formel im Antezedenten angesehen werden.

Um eine gestartete VD-Induktion anwenden zu können, muss im bisherigen Beweis eine Sequenz gefunden werden, für die zwei Voraussetzungen erfüllt sind. Zum Einen muss gezeigt werden, dass das gegenwärtige Beweisziel „im Wesentlichen gleich“ zu der gefundenen Sequenz ist. Zum Anderen muss nachgewiesen werden, dass der Term, über den die Induktion gestartet wurde, von der aufgefundenen Sequenz zur aktuellen Beweisverpflichtung kleiner geworden ist.

Die Anwendung der VD-Induktion wird hier an einem Beispiel erläutert. Das System wird durch die temporallogische Formel $\Box\ (\texttt{N'} = \texttt{N} + 1 \wedge \lceil \texttt{N} \rceil \wedge \neg\ \textbf{last})$ beschrieben. Diese erhöht den Wert der Variablen N unendlich lange. Das Programm wird im Kontext folgender Umgebungsannahme ausgeführt. $\Box\ (\texttt{N''} = \texttt{N'} \wedge \texttt{M''} = \texttt{M'})$. Das heißt, die Umgebung lässt die Variablen N sowie M unverändert. Zu beweisen ist die Aussage $\Diamond\ \texttt{N} > \texttt{M}$. Es ist also zu zeigen, dass irgendwann N größer als die unspezifizierte Variable M wird. Diese Beweisverpflichtung lässt sich informell formulieren als: N wächst über jede Schranke hinaus. Komplett lautet die Aussage

$$\Box\ (\texttt{N'} = \texttt{N} + 1 \wedge \lceil \texttt{N} \rceil \wedge \neg\ \textbf{last}), \Box\ (\texttt{N''} = \texttt{N'} \wedge \texttt{M''} = \texttt{M'}) \vdash \Diamond\ \texttt{N} > \texttt{M}$$

Um diese Aussage beweisen zu können wird die VD-Induktion benötigt. Es wird dazu angenommen, dass sowohl N als auch M natürliche Zahlen sind. Als Induktionsterm wird die Differenz der Werte von N und M verwendet, also M - N. Nach Start der Induktion wird ein temporallogischer Schritt ausgeführt. Durch diesen wird der Wert von N um eins erhöht und somit der Wert des Induktionsterms um eins verringert. Damit kann die Induktion geschlossen werden. Zu beweisen ist noch, dass die Sequenz, die nach Ausführung des Schritts entsteht im Wesentlichen gleich der Sequenz vor Ausführung des Schritts ist. Dazu wird der gleiche Test verwendet, wie bei Anwenden der Sequenzierung, wie in Abschnitt 4.3.2 beschrieben. Da sich Systembeschreibung und Umgebungsannahme durch den Schritt nicht verändert haben, ist dieser Test erfolgreich und die Aussage somit gezeigt.

4.3.4 Abstraktion

Durch die Kompositionalität der Parallel-Operatoren ist es möglich, Formeln unterhalb der Parallel-Operatoren zu ersetzen. Dieser Vorgang wird Abstraktion genannt. Kann bewiesen

werden, dass für eine temporallogische Formel φ gilt, dass $\varphi \rightarrow \psi$, dann folgt aus der Kompositionalität automatisch, dass bei einer Systembeschreibung im Antezedenten φ durch ψ ersetzt werden kann. Beispielsweise kann dann eine Systembeschreibung $\varphi \parallel_s \alpha$ ersetzt werden durch $\psi \parallel_s \alpha$. Ziel dieser Ersetzung ist eine Vereinfachung und Verkürzung des Beweises.

In Abschnitt 4.3.3 wird das Beispiel des einfachen Zählers eingeführt, der die Variable N unendlich inkrementiert. Wird die Systembeschreibung dieses Zählers parallel zu einem beliebigen anderen Programm ausgeführt, so bleibt die Eigenschaft, dass N über jede beliebige Schranke wächst, erhalten, wenn sich das zum Zähler parallele Programm auf bestimmte Art und Weise verhält. Das Programm darf die Werte der Variablen N und M nicht verändern. Die Abstraktion soll nun dazu verwendet werden, die nachstehende Aussage zu beweisen:

$$\Box\,(\mathtt{N'} = \mathtt{N} + 1 \wedge \lceil \mathtt{N} \rceil \wedge \neg\ \mathbf{last})\|_s\,\alpha, \Box\,(\mathtt{N''} = \mathtt{N'} \wedge \mathtt{M''} = \mathtt{M'}) \vdash \Diamond\,\mathtt{N} > \mathtt{M}$$

Dazu wird für das Programm α bewiesen, dass $\alpha \rightarrow \Box\,(\mathtt{M'} = \mathtt{M} \wedge \mathtt{N'} = \mathtt{N})$ gilt. Kann diese Aussage gezeigt werden, dann kann in der Beweisverpflichtung α ersetzt werden:

$$\begin{array}{l} \Box\,(\mathtt{N'} = \mathtt{N} + 1 \wedge \lceil \mathtt{N} \rceil \wedge \neg\ \mathbf{last})\|_s\,\Box\,(\mathtt{M'} = \mathtt{M} \wedge \mathtt{N'} = \mathtt{N}), \\ \Box\,(\mathtt{N''} = \mathtt{N'} \wedge \mathtt{M''} = \mathtt{M'}) \\ \vdash \Diamond\,\mathtt{N} > \mathtt{M} \end{array}$$

Das Programm α muss in seiner Form nicht statisch sein. Das heißt, nach Ausführung eines Schritts kann das Programm abgeändert worden sein in das Programm α'. In diesem Fall gibt es in der ursprünglichen Beweisverpflichtung keine Möglichkeit die Induktion zu schließen, da sich die Systembeschreibung durch den Schritt verändert hat. Im Gegensatz dazu bleibt die abstrakte Systembeschreibung immer gleich.

Zu Beachten ist bei der Abstraktion, dass diese Komplexität nicht „verschwinden“ lässt, sondern nur auslagert. Im Beispiel kann im Beweis mit dem abstrahierten Programm zwar innerhalb eines Schrittes Induktion angewendet und der Beweis damit geschlossen werden. Der Beweis der Abstraktion $\alpha \rightarrow \varphi$ dagegen kann beliebig kompliziert werden. Daher ist abzuwägen, wann die Technik sinnvoll eingesetzt werden kann. Dies ist häufig dann der Fall, wenn zwei komplexe Programme parallel ausgeführt werden und das für die Korrektheit der Formel relevante Verhalten der Programme einfach ist. In solchen Fällen kann Abstraktion verwendet werden, damit sich die Komplexität der beiden parallelen Programme nicht gegenseitig verstärkt.

5 Die Brustkrebs-Fallstudie

Die entwickelten Methoden zur Qualitätssicherung von Asbru-Leitlinien werden in dieser Arbeit auf eine Leitlinie zur Behandlung von Brustkrebs [7, 55, 59] angewendet. Die ursprüngliche Form der Leitlinie ist ein Dokument in natürlichsprachigem, holländischem Text. Im Rahmen des EU-Projekts Protocure II wurde dieses Dokument zunächst in die englische Sprache übersetzt. Aus dieser Übersetzung wurde ein Asbru-Modell der Leitlinie erstellt. Dieses Modell wird zusammen mit dem Formalisierungsprozess in [48] vorgestellt. Die Repräsentationssprachen, die für die einzelnen Schritte dieses Formalisierungsprozesses genutzt werden, werden in [7] beschrieben. Dieser Abschnitt erläutert die verschiedenen Kapitel der Leitlinie hinsichtlich ihres medizinischen Inhaltes. Weiterhin werden die formalen Asbru-Modelle der Kapitel beschrieben.

Die untersuchte Brustkrebsfallstudie untergliedert sich in acht Kapitel. Diese beschreiben die Behandlung von Brustkrebs in unterschiedlichen Krankheitsstadien sowie verschiedene Aspekte die behandlungsbegleitend berücksichtigt werden sollen. Nach einer allgemeinen Einführung in die Behandlung von Brustkrebs sowie die Umstände der Erstellung der Leitlinie wird in Kapitel 1 die Behandlung von DCIS-Brustkrebs sowie von operabelem invasivem Brustkrebs beschrieben. Dieses Kapitel der Leitlinie wird in Abschnitt 5.2 beschrieben. Daran anschließend beschreibt Abschnitt 5.3 das zweite Kapitel der Leitlinie. Dieses erläutert Aspekte der behandlungsbegleitende Therapie, also beispielsweise einer Chemotherapie, die den chirurgischen Eingriff zur Tumorentferung unterstützt. Kapitel drei der Fallstudie beschreibt die Behandlung von lokal begrenztem Brustkrebs in fortgeschrittenem Stadium. Dieses Kapitel wird in Abschnitt 5.4 beschrieben. Darauf folgend wird in Abschnitt 5.5 Kapitel vier der Fallstudie beschrieben. Dieses enthält Informationen zu Nachsorge-Untersuchungen. Kapitel fünf beschreibt lokal wieder auftretenden Brustkrebs. Dieses Kapitel wird in Abschnitt 5.6 beschrieben. Kapitel sechs wird in Abschnitt 5.7 dokumentiert und beschreibt die Behandlung von metastasierendem Brustkrebs.

Kapitel sieben und Kapitel acht der Brustkrebs Fallstudie enthalten Informationen über psychosoziale Aspekte, beispielsweise die psychologische Betreuung der Patienten während der Behandlung und die Organisation der Behandlung. Diese Aspekte werden nicht formal untersucht. Diese Entscheidung wurde auf Basis einer Empfehlung der Leitlinienersteller des niederländischen Instituts für Qualitätssicherung im Gesundheitswesen, CBO, getroffen.

In den Asbru-Modellen der hier vorgestellten Fallstudie werden die Namen der Asbru-Pläne verkürzt dargestellt. Beispielsweise wird der Plan `dealing with axilla` im Asbru-Modell schlicht mit der Abkürzung `dwa` bezeichnet. Bei der Vorstellung der Pläne in den nächsten Abschnitten werden die Namen der Pläne und die Abkürzungen unter denen die Pläne im jeweiligen Modell des Kapitels zu finden sind wie folgt angegeben: `dealing with axilla/dwa`

5.1 Graphische Darstellung der Planhierarchien

Graphisch werden die Pläne durch Bäume repräsentiert dargestellt. Dabei entspricht ein Knoten einem Asbru Plan, die Vater-Kind Beziehung im Baum entspricht der Super-/Subplan Relation der Pläne. Ein Beispiel ist in Abbildung 5.1 dargestellt. Die Obere Hälfte einer Planrepräsentation enthält den abgekürzten Namen des Plans, hier etwa `chap1` als Kurzform von `chapter1`. Die untere Hälfte der Planrepräsentation enthält die drei wichtigsten

Aspekte der Plandefinition, den Plankontrolltyp, die Wartebedingung und die Angabe, welche Bedingungen im Plan gesetzt sind.

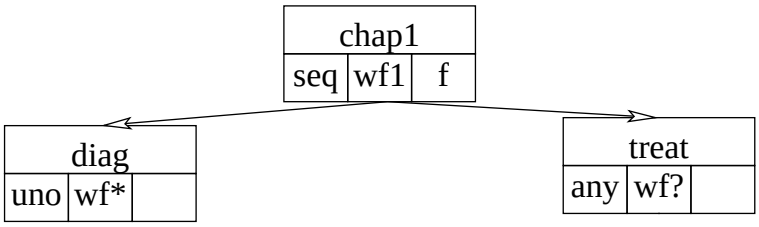


Abbildung 5.1: Graphische Darstellung von Asbru-Plänen

Im Beispiel ist zu erkennen, dass das linke der drei unteren Rechtecke der `chap1` Definition mit seq markiert ist. Hierbei handelt es sich um den Kontrolltyp von `chap1`. seq ist eine Abkürzung für `sequential`. Analog dazu werden Abkürzungen für die übrigen Plantypen definiert:

seq	:≡	`sequential`
uno	:≡	`unordered`
any	:≡	`anyorder`
par	:≡	`parallel`
ift	:≡	`ifThenElse`
ona	:≡	`onabort`
cyc	:≡	`cyclical`
use	:≡	`user`
ask	:≡	`ask`

Das zweite Rechteck beschreibt die Wartebedingung des Plans. Angegeben ist hierbei, ob der Plan auf alle seine Unterpläne wartet, was durch Angabe von wf* kenntlich gemacht wird. Alternativ kann der Plan auch nur auf einen Unterplan warten. Dieser Fall wird graphisch durch Angabe von wf1 dargestellt. In allen anderen Fällen wird der Plan mit wf? markiert. Im Beispiel in Abbildung 5.1 ist zu sehen, dass der Plan `chap1` nur auf einen seiner Unterpläne wartet.

Zuletzt wird angegeben, welche Bedingungen in einem Plan definiert sind. Prinzipiell kennen Asbru-Pläne sechs Bedingungen, **filter**, **setup**, **suspend**, **reactivate**, **abort** und **complete**. In der Fallstudie werden lediglich **filter**-, **abort**- und **complete**-Bedingungen verwendet. Wenn diese bei einem Plan definiert sind, so ist der Anfangsbuchstabe der Bedingung im letzten der Rechtecke angegeben. Im Fall von `chap1` ist beispielsweise eine **filter**-Bedingung definiert, deswegen wird im Beispiel für diesen Plan ein f im letzten Rechteck angegeben.

5.2 DCIS und operabeler, invasiver Brustkrebs

5.2.1 Medizinischer Inhalt des Kapitels

DCIS steht als Abkürzung für die Bezeichnung „Ductual Carcinoma in Situ“ und beschreibt eine krankhafte Wucherung in den Milchgängen der weiblichen Brust. Eine solche Wucherung ist per Definition („in situ“) stets ortsfest und hat die Grenze des Milchgangs noch nicht durchbrochen. DCIS gilt als Vorstufe des invasiven Brustkrebses, jedoch ist derzeit ungeklärt,

wie wahrscheinlich eine Weiterentwicklung von unbehandeltem DCIS zu invasivem Brustkrebs im Allgemeinen ist.

Ein scheinbares Paradox ergibt sich bei der Behandlung von invasivem Brustkrebs und DCIS dadurch, dass das Vorstadium DCIS stets mit Mastektomie behandelt werden soll, also der vollständigen Entfernung der Brust. Gleichzeitig zeigen Studien, dass die Behandlung des Folgestadiums, also invasiven Brustkrebses, mit brusterhaltender Therapie statt der kompletten Brustentfernung keinen statistisch signifikanten Einfluss auf die Langzeit-Überlebenserwartung hat.

Dieses Paradox erklärt sich zumindest teilweise daraus, dass die eigentliche Größe des krankhaften Gewebes bei DCIS nicht exakt bestimmt werden kann. Durch bildgebende Verfahren, wie etwa Ultraschall kann bei DCIS Patienten sogenannter Mikrokalk nachgewiesen werden. Dieser besteht aus Überresten abgestorbenener Zellen und gibt einen Hinweis auf die Ausbreitung von DCIS. Eine Entfernung allen Gewebes, in dem Mikrokalk nachgewiesen werden kann, entspricht typischerweise einer kompletten Brustentfernung. Erst nach der Entfernung des Gewebes kann dann die exakte Ausbreitung der Wucherung bestimmt werden. Im Gegensatz dazu zeichnen sich Tumore des invasiven Brustkrebses auf Mammographiebildern, also speziellen Röntgenaufnahmen der Brust, deutlicher ab und können so besser abgegrenzt werden.

Bei invasivem Brustkrebs kann auf eine komplette Entfernung der Brust verzichtet werden, wenn nach einer Teilentfernung der Brust die Schnittränder des entnommenen Gewebes tumorfrei sind, also in dem entnommenen Gewebe der Tumor komplett von gesundem Gewebe umschlossen ist. Diese Art der Operation nennt sich brusterhaltende Operation bzw. brusterhaltende Therapie. Diese darf nur in Betracht gezogen werden, wenn eine Mammographie keine Hinweise auf Krebsherde außerhalb des Haupttumors ergibt und der Tumor auf einen Quadranten der Brust begrenzt ist. Neben diesen Forderungen existieren noch eine Reihe weitere Kontraindikationen für die brusterhaltende Therapie. Diese Kontraindikationen können bei etwa einem Viertel der Patientinnen nachgewiesen werden. Damit erfüllen Dreiviertel der Patientinnen die Vorbedingungen, die die Durchführung der brusterhaltenden Therapie voraussetzt. Untersuchungen zeigen, dass das psychische Wohlbefinden der Patienten nach einer brusterhaltenden Therapie maßgeblich dadurch bestimmt wird, ob diese die brusterhaltende Therapie wünschen. Im Falle des entsprechenden Patientenwunsches wird daher stets die komplette Brustentfernung durchgeführt. Untersuchungen zeigen bei Abwesenheit der Kontraindikationen keine statistisch signifikanten Unterschiede in der Langzeit-Überlebensrate zwischen Patienten, die mit brusterhaltender Therapie behandelt wurden und solchen, denen die Brust komplett abgenommen wurde.

5.2.2 Asbru Modell des ersten Kapitels

Das Asbru-Modell des ersten Kapitels ist in Abbildung 5.2 dargestellt. Oberster Plan des Kapitels ist der Plan `chapter 1/chap1`. Dieser startet zunächst die Erstellung einer Diagnose mittels Ultraschall und Mammographie. Mit den Ergebnissen dieser Diagnose wird die Behandlung gestartet. Im Vorfeld der Behandlung soll der Patient informiert (Plan `patient information/pin`) werden sowie ein interdisziplinären Treffen der behandelnden Ärzte stattfinden (Plan `multi disciplinary decisions on BCT/mdBCT`), um die Aspekte brusterhaltender Therapie dieses Falls zu diskutieren. Die eigentliche Behandlung besteht entweder aus brusterhaltender Therapie (Plan `breast conserving therapy/bct`) oder aus Mastektomie (Plan `mastectomy/mast`). Brusterhaltende Therapie verlangt, dass der Tumor komplett entfernt wurde, das heißt, die Schnittränder müssen tumorfrei sein. Andernfalls soll die brusterhaltende Therapie abgebrochen werden. Strahlentherapie (Plan `dcis radiotherapy/dcis`) ist ein zwingender Bestandteil der brusterhaltenden Thera-

pie. Alternativ zur brusterhaltenden Therapie kann, bei Vorliegen von Kontraindikationen oder dem entsprechenden Patientenwunsch, eine komplette Brustentfernung durchgeführt werden.

Nach Abschluss der Operation kann unabhängig von der konkreten Methodik der Operation eine Strahlentherapie angeordnet werden (Plan `locoregional postoperative radiotherapy/lrpr`). Weiterhin sollen die Lymphknoten der Achsel untersucht werden (Plan `dealing with axilla/dwa`). Dazu werden Methoden angegeben, die Metastasen in den Lymphknoten aufspüren sollen. Außerdem werden dort Operationen zur Entfernung befallener Achselknoten beschrieben.

5.3 Behandlungsbegleitende systematische Therapie

5.3.1 Medizinischer Inhalt des Kapitels

Behandlungsbegleitende, systematische Therapie besteht aus Chemotherapie oder endokriner Therapie, also Hormontherapie. Diese Formen der Therapie werden angewendet um Metastasen entgegen zu wirken, die sich unerkannt neben dem Haupttumor gebildet haben können. Die Auftretenswahrscheinlichkeit solcher Metastasen (und damit auch die Sinnhaftigkeit systematischer Therapie) hängt von verschiedenen Faktoren wie etwa Tumorgröße und Lymphknoten-Befall aber auch dem Alter des Patienten ab.

Untersuchungen zeigen, dass gerade bei Befall der Lymphknoten mit Krebs die Chemotherapie entscheidende Verbesserungen im Bezug auf die Langzeitüberlebenserwartung hat. Auch bei krebsfreien Lymphknoten trägt eine begleitende Chemotherapie zu einer Erhöhung der Lebenserwartung bei, jedoch ist bei dieser Patientengruppe der Überlebensvorteil nicht ganz so ausgeprägt. Untersuchungen zeigen, dass diese Effekte abhängig sind vom Alter der Patientin. Der Leitlinie liegen beispielsweise keine Studien zugrunde, die positive Effekte systematischer Therapie bei über 70-Jährigen belegen.

Endokrine Therapie verfolgt das Ziel, Tumoren die Wachstumsgrundlage zu entziehen, die auf die Anwesenheit der Hormone Östrogen und Progesteron mit Wachstum reagieren (sog. ER+/PgR+-Tumore). Für Patientinnen, die sich noch nicht in der Menopause befinden kann dieses Ziel auch durch eine Entfernung der Gebärmutter erreicht werden. In der Menopause können Medikamente gegeben werden, die Östrogen- und Progesteronrezeptoren blockieren und dadurch ein Andocken von Hormonen an die Tumorzellen verhindern.

Abhängig von der Art des Tumors und der Physiologie der Patientin wird in Kapitel zwei der Fallstudie anhand von Tabellen angegeben, welche Kombination von Chemotherapie und endokriner Therapie für einen spezifischen Patienten empfohlen wird. Speziell wird dabei auf den Fall von schwangeren Patienten eingegangen. Bei diesen muss das Wohl der Mutter gegen das des Kindes abgewogen werden und gegebenenfalls die systematische Therapie verzögert werden, um das ungeborene Leben nicht zu gefährden. Dazu werden Therapien wann immer möglich zumindest im ersten Drittel der Schwangerschaft verzögert. Mit geeigneten Medikamenten ist im zweiten und dritten Drittel der Schwangerschaft eine Chemotherapie für das Kind relativ gefahrlos möglich.

5.3.2 Asbru Modell des zweiten Kapitels

Das zweite Kapitel der Fallstudie untergliedert sich im Asbru-Modell, das in Abbildung 5.3 dargestellt ist, in einen Teil dessen Pläne allein für die Auswahl einer geeigneten Therapie zuständig sind und einen Teil, der die Therapie in Abhängigkeit dieser Entscheidung beschreibt. Letztlich wird die tatsächlich auszuführende Therapie durch drei Pläne beschreiben, einen zur Einleitung der Chemotherapie, welcher `adjuvant chemotherapy/adch` heißt, einen zur Durchführung der endokrinen Therapie

namens `adjuvant endocrine therapy for premenopausal status/atps` sowie einen Plan, der beschreibt, dass keine begleitende Therapie gegeben werden soll. Dieser Plan heißt `no treatment should be given/ntsbg`.

Zur Erläuterung soll ein Durchlauf durch diesen Entscheidungsbaum angegeben werden. Zunächst wird auf oberster Ebene durch die **filter**-Bedingungen der Pläne `treatment for pregnant/trpr` und `treatment for N-positive and negative patients/tNpn` die Patientengruppen aufgeteilt, abhängig davon, ob die Patientin schwanger ist oder nicht. Ist die Patientin nicht schwanger, so wird Plan `treatment for N positive and negative patients/tNpn` ausgeführt. Die Ausführung dieses Plans bedeutet eine erneute Aufteilung der Patienten. In diesem Fall wird unterschieden, ob die Lymphknoten der Patienten mit Krebs befallen sind oder nicht. Diese Unterscheidung wird mit Hilfe der **filter**-Bedingungen der Pläne `treatment for N negative patients at grade level/TNngl` und `treatment for N positive patients/tNp` durchgeführt. Sind die Lymphknoten nicht befallen, kommt der Plan `treatment for N negative patients at grade level/tNp` zur Anwendung. Dieser teilt die Patienten in Gruppen, die durch das Stadium des Krebsbefalls ausgezeichnet sind. Im Fall eines relativ frühen Stadiums des Krebses wird der Plan `grades I II treatment/g12t` ausgeführt. Dieser entscheidet anhand von weiteren Randbedingungen, ob überhaupt eine Behandlung durchgeführt werden soll. Falls ja, kommt der Plan `treatment for N negative patients at menopausal status level/tNnms` zur Anwendung. Dieser unterscheidet zwischen Patienten in der Menopause und solchen, die die Menopause noch nicht erreicht haben. Abhängig davon werden dann die Chemotherapie und die Hormontherapie gestartet, wobei innerhalb der Behandlungsgruppen jeweils verschiedene Medikamente zur Behandlung bereitstehen.

5.4 Lokal fortgeschrittener Brustkrebs

5.4.1 Medizinischer Inhalt des Kapitels

In den 50er Jahren des letzten Jahrhunderts wurden Kriterien definiert, unter denen eine Behandlung von fortgeschrittenem, aber noch lokal begrenztem Brustkrebs nicht mehr als sinnvoll erachtet wurde, da die Prognose sehr schlecht war. Beispiele für Tumore, die diese Kriterien erfüllen sind solche Tumore, die sich bereits so weit ausgebreitet haben, dass sich Entzündungen der Haut, Ödeme der Haut, Eitergeschwulste oder ähnliches zeigen. In diesen Fällen brachte früher selbst eine Brustentfernung keine Verbesserung der Langzeit-Überlebenswahrscheinlichkeit. Mit dem Aufkommen der Chemotherapie in den 70er Jahren hat sich dies geändert.

Kapitel drei der Leitlinie beschreibt die Behandlung von Brustkrebs, der nach den ursprünglichen Kriterien als nicht mehr operabel angesehen wird. Die verbesserte Operationstechnik sowie das Aufkommen von systematischer Therapie führt dazu, dass die Behandlung in diesen Fällen eine signifikante Verbesserung der Überlebenswahrscheinlichkeit bringt. Dazu wird die systematische Therapie (also Chemo- und gegebenenfalls Endokrine-Therapie) angewendet, um bereits vor der chirurgischen Tumorentfernung die Tumorgröße zu reduzieren. Dadurch wird erreicht, dass der Tumor besser kontrollierbar wird, was sich positiv auf das Operationsergebnis auswirkt.

Im Allgemeinen besteht die Behandlung daher aus einer initialen systematischen Therapie, gefolgt von einer Behandlung die der von operablem invasivem Brustkrebs entspricht. In diesem Fall wird die systematische Therapie als nah-behandlungsbegleitend bezeichnet.

5.4.2 Asbru Modell des dritten Kapitels

Das Asbru-Modell des dritten Kapitels wird in Abbildung 5.4 dargestellt. Bei der Behandlung von fortgeschrittenem Brustkrebs wird zunächst vom Plan `locally advanced breastcancer/load` untersucht, ob Kontraindikationen gegen eine nah-behandlungsbegleitende Chemotherapie vorliegen. In Abhängigkeit davon wird eine Behandlung gestartet, die mit einer initialen Chemotherapie beginnt (Plan `intensive treatment/intm`) oder diese auslässt (Plan `less intensive treatment/lintm`). Nach Abschluss der Chemotherapie wird eine chirurgische Tumorentfernung durch den Plan `locoregional treatment/lotr` durchgeführt. Diese kann von einer Strahlentherapie begleitet werden (Plan `locoregional rediotherapy manual/lrm`). Abschließend wird der Patient mit einer Hormontherapie behandelt, sofern dies medizinisch sinnvoll ist. Dies wird durch den Plan `adjuvant hormonal therapy/aht` modelliert.

5.5 Nachsorge-Untersuchungen

5.5.1 Medizinischer Inhalt des Kapitels

Nachdem die eigentliche Behandlung einer Brustkrebs-Erkrankung abgeschlossen ist, sollten Nachsorge-Untersuchungen durchgeführt werden. Verschiedene Indizien deuten daraufhin, dass in manchen Fällen bei der Behandlung von Brustkrebs ein zweiter primärer Tumor existiert, der in der Diagnose nicht erfasst und in der Folge nicht chirurgisch entfernt wird. Außerdem kann sich unabhängig davon mit zeitlicher Verzögerung erneut ein Tumor bilden.

Etwa ein Drittel der wiederkehrenden Krebsfälle sind von Metastasen begleitet, die frühzeitig erkannt und behandelt werden sollen. Die Leitlinie empfiehlt dazu in Abhängigkeit des Alters jährliche bzw. zwei-jährliche Mammographien. Spezielle, komplexe Nachuntersuchungen über die Mammographie hinaus, beispielsweise zum Aufspüren von Metastasen haben dagegen im Allgemeinen keinen positiven Effekt auf die Überlebenswahrscheinlichkeit.

Frauen, die langfristig den Wirkstoff Tamoxifen als Teil der systematischen Therapie einnehmen, sollen bei der Nachsorge speziell auf Karzinome des Endometriums untersucht werden, da diese bei einer Tamoxifen-Behandlung, die länger als fünf Jahre andauert, mit leicht erhöhter Wahrscheinlichkeit auftreten können.

Abgesehen von der Krebsnachsorge sollte bei den Nachuntersuchungen auch auf die physiologischen und psychologischen Auswirkungen der Behandlung geachtet werden. Durch die chirurgische Entfernung von Gewebe kann es zu Spannungen und Schmerzen bei der Bewegung der Arme kommen. Auch auf andere Effekte der Behandlung, die das Wohlbefinden einschränken, sollte geachtet werden.

Die Leitlinie empfiehlt zusätzlich zu den regelmäßigen Mammographien weitere Untersuchungen. Im ersten Jahr nach der Behandlung wird eine Nachsorgeuntersuchung pro Quartal vorgeschlagen, im zweiten Jahr halbjährliche Untersuchungen und danach jährliche Untersuchungen.

5.5.2 Asbru Modell des vierten Kapitels

Die Nachsorgeuntersuchungen bestehen aus fünf verschiedenen Komponenten, die in Abbildung 5.5 visualisiert sind. Als erstes soll der Patient unmittelbar nach seiner initialen Behandlung darüber informiert werden, warum Nachsorgeuntersuchungen wichtig sind und welche Untersuchungen wann anstehen. Diese Informationsvermittlung wird durch den Asbru-Plan `follow-up information/fui` modelliert. Weiterhin soll bei Nachsorgeuntersuchungen stets geprüft werden, ob die initiale Diagnose noch Gültigkeit

hat, oder ob, beispielsweise durch Auftreten von Metastasen oder einem neuen, primären Tumor die ursprüngliche Diagnose geändert werden muss. Dafür zuständig ist der Plan `determine diagnosis change/ddcy`.

Wichtiger Aspekt der Nachsorge-Untersuchungen ist die Durchführung von Mammographien. In Abhängigkeit des Alters sollen diese unterschiedlich oft ausgeführt werden. Die Durchführung dieser Untersuchungen wird durch den Plan `mammographie programme/mapro` geregelt. Unabhängig von den Mammographien sollen allgemeine Nachsorge-Untersuchungen durchgeführt werden. Dies wird durch den Plan `follow-up programme/fup` umgesetzt.

Wenn sehr ausgeprägte psychologische Probleme bei einem Patienten auftreten, die in Beziehung gesetzt werden können zum plötzlichen Einsetzen der „Wechseljahre“ bedingt durch Anti-Östrogenbehandlung, dann kann eine Hormonersatztherapie (also Östrogengabe) verordnet werden. Dies ist jedoch im Allgemeinen nicht zu empfehlen.

5.6 Lokal wiederkehrender Brustkrebs

5.6.1 Medizinischer Inhalt des Kapitels

Bei lokal wiederkehrendem Brustkrebs wird aufgrund der relative guten Prognose eine erneute Behandlung mit dem Ziel der Entfernung des Tumors durchgeführt. In diesen Fällen wird, falls möglich, der Tumor zunächst chirurgisch entfernt und in der Folge das Gewebe mit hochdosierter Strahlentherapie behandelt. Wurde das befallene Gewebe allerdings im Vorfeld bereits durch Strahlentherapie geschädigt, so soll statt hochdosierter Strahlentherapie niedriger dosierte Strahlentherapie zum Einsatz kommen, die mit hyperthermischer Behandlung – also Hitze Therapie – verbunden werden soll.

Eine begleitende Hormontherapie bei entsprechenden (ER+/PgR+) Tumoren kann die Überlebenswahrscheinlichkeit erhöhen, wohingegen keine Studie einen positiven Beitrag von Chemotherapie festellen kann.

5.6.2 Asbru Modell des fünften Kapitels

Das Asbru Modell des fünften Kapitels wird in Abbildung 5.6 dargestellt und besteht aus drei Komponenten. Die erste wird vom Plan `diagnostic procedures group/dpg` gesteuert und beinhaltet das Erstellen einer genauen Diagnose des wieder-aufgetretenen Brustkrebses. Eine zweite Komponente erfragt den Typ des wiederauftretenden Tumors, also ob er nur lokal beschränkt ist oder Metastasen bildet. Die dritte Komponente umfasst die eigentliche Behandlung.

Die Komplexität der Behandlungskomponente ergibt sich daraus, dass eine Vielzahl möglicher Diagnosen existiert, beispielsweise ob ein Auftreten verbunden ist mit Metastasen oder nicht. Auch ist die Behandlung abhängig vom Vorgehen, das beim ersten Auftreten des Brustkrebses gewählt wurde. Beispielsweise kommt hier die bereits erwähnte Unterscheidung zum Tragen, ob Gewebe bereits bestrahlt wurde und auf diesem Grund nicht mehr mit voller Dosis bestrahlt werden darf.

5.7 Metastasierender Brustkrebs

5.7.1 Medizinischer Inhalt des Kapitels

Im Fall von Brustkrebs, der entfernte Metastasen bildet, also beispielsweise im Knochenmark, in der Lunge oder in der Leber, ist davon auszugehen, dass dieser unheilbar ist. Entsprechend konzentriert sich die Behandlung in diesen Fällen darauf, das Wohlbefinden des Patienten zu

steigern bzw. auf möglichst hohem Niveau zu halten. Dazu kann Hormontherapie in Form von Tamoxifen und anderen Wirkstoffen bei bestimmten Charakteristika der Metastasen verwendet werden. Alternativ ist eine Behandlung mit Chemotherapie denkbar, um den Verlauf zu verzögern. Strahlentherapie wird angewendet, um einzelne Metastasen kurzfristig in der Größe zu reduzieren und so den davon ausgelösten Schmerz zu lindern.

Die Behandlung von metastasierendem Brustkrebs verlangt intensive psychologische Betreuung des Patienten. Da die Patienten mit der Tatsache konfrontiert werden, dass sie unheilbar krank sind, lastet enormer Druck auf ihnen. Die Leitlinie erwähnt spezielle, zusätzliche Stressfaktoren, beispielsweise die psychologischen Folgen von Kurzatmigkeit, die durch Metastasen ausgelöst wird.

5.7.2 Asbru Modell des sechsten Kapitels

Das Asbru Modell der Behandlung von metastasierendem Brustkrebs wird in Abbildung 5.7 dargestellt. Es untergliedert sich in eine Diagnose-Komponente und eine Behandlungs-Komponente. In der Diagnose-Komponente, die durch den Plan `diagnostic procedures group/dpg` repräsentiert wird, werden die unterschiedlichen Diagnoseverfahren, also beispielsweise Ultraschall- und Röntgenuntersuchungen modelliert. Weiterhin wird dort angegeben, wie diese Untersuchungen zu verwenden sind, um Metastasen in den einzelnen Körperteilen aufzuspüren. Beispielsweise soll der Plan `heart imaging/heim` untersuchen, ob das Herz durch Tumore in seiner Funktion beeinträchtigt wird oder werden könnte. Diese Untersuchung kann mittels einer Ultraschalluntersuchung durchgeführt werden, die durch den Plan `ultrasound/ultra` modelliert wird. Damit kann Krebsbefall des Herzens nachgewiesen werden. Die Leitlinie lässt zu, dass diese Untersuchung alternativ mit einer Röntgenaufnahme des Brustkorbs durchgeführt werden kann. Diese wird durch den Plan `chest x-ray/chx` modelliert. Nach Abschluss der Diagnose wird die Behandlung gestartet. Diese richtet sich in erster Linie nach den Beschwerden des Patienten. Die einzig präventive Maßnahme, die im Fall von metastasierendem Brustkrebs durchgeführt wird, ist die Hormon-Therapie (Plan `hormone therapy/hothe`). Diese wird in verschiedenen Stufen angewendet, so dass immer dann, wenn eine Behandlung nicht mehr anschlägt, die nächststärkere automatisch gestartet wird.

Abgesehen davon werden nur Behandlungen ausgeführt, um eine unmittelbare Verbesserung der Lebenssituation zu erreichen, wie etwa eine Strahlentherapie, um eine Metastase kurzfristig in ihrer Größe zu reduzieren oder die Gabe von schmerzstillenden Medikamenten.

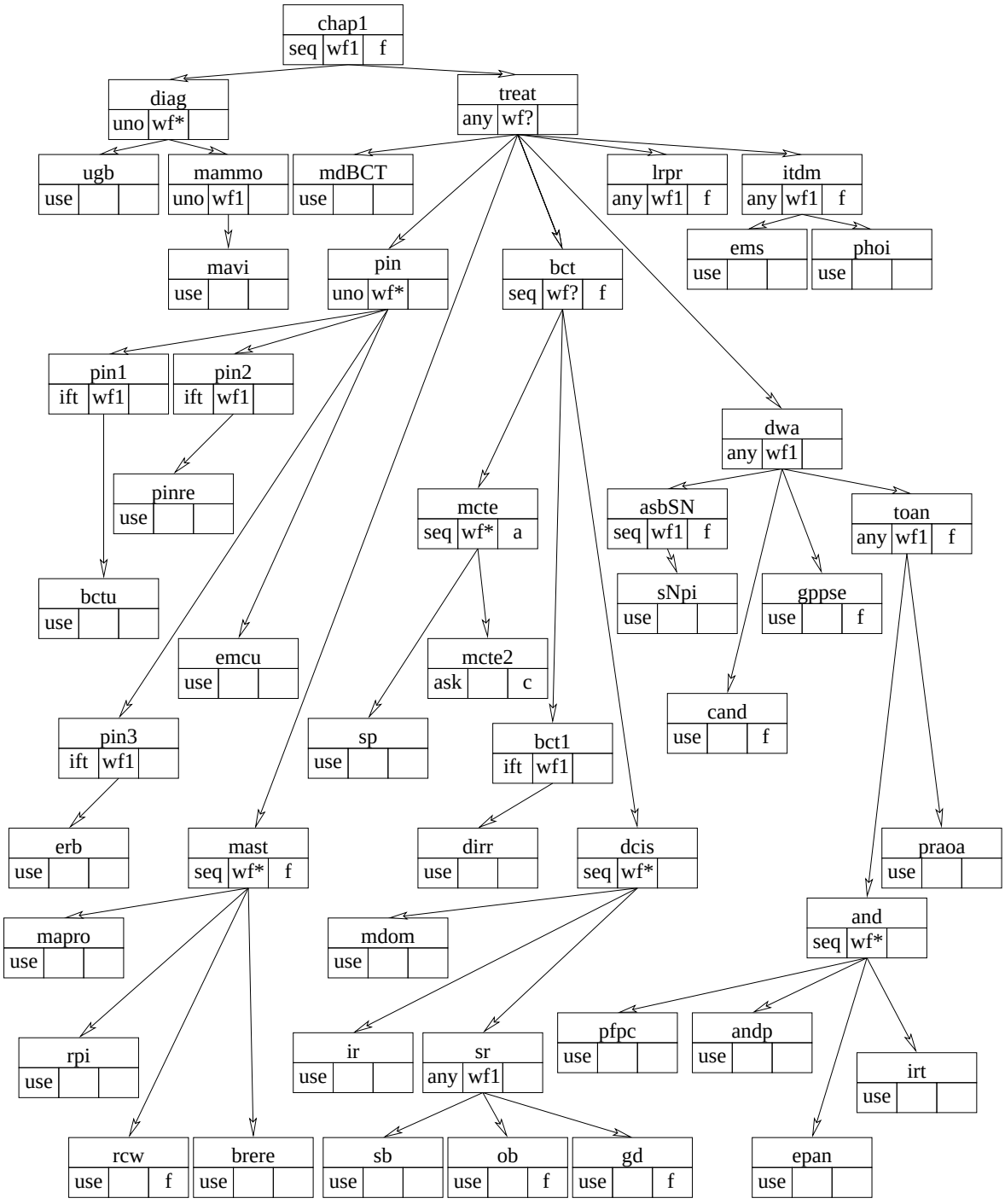


Abbildung 5.2: Asbru Modell des ersten Kapitels

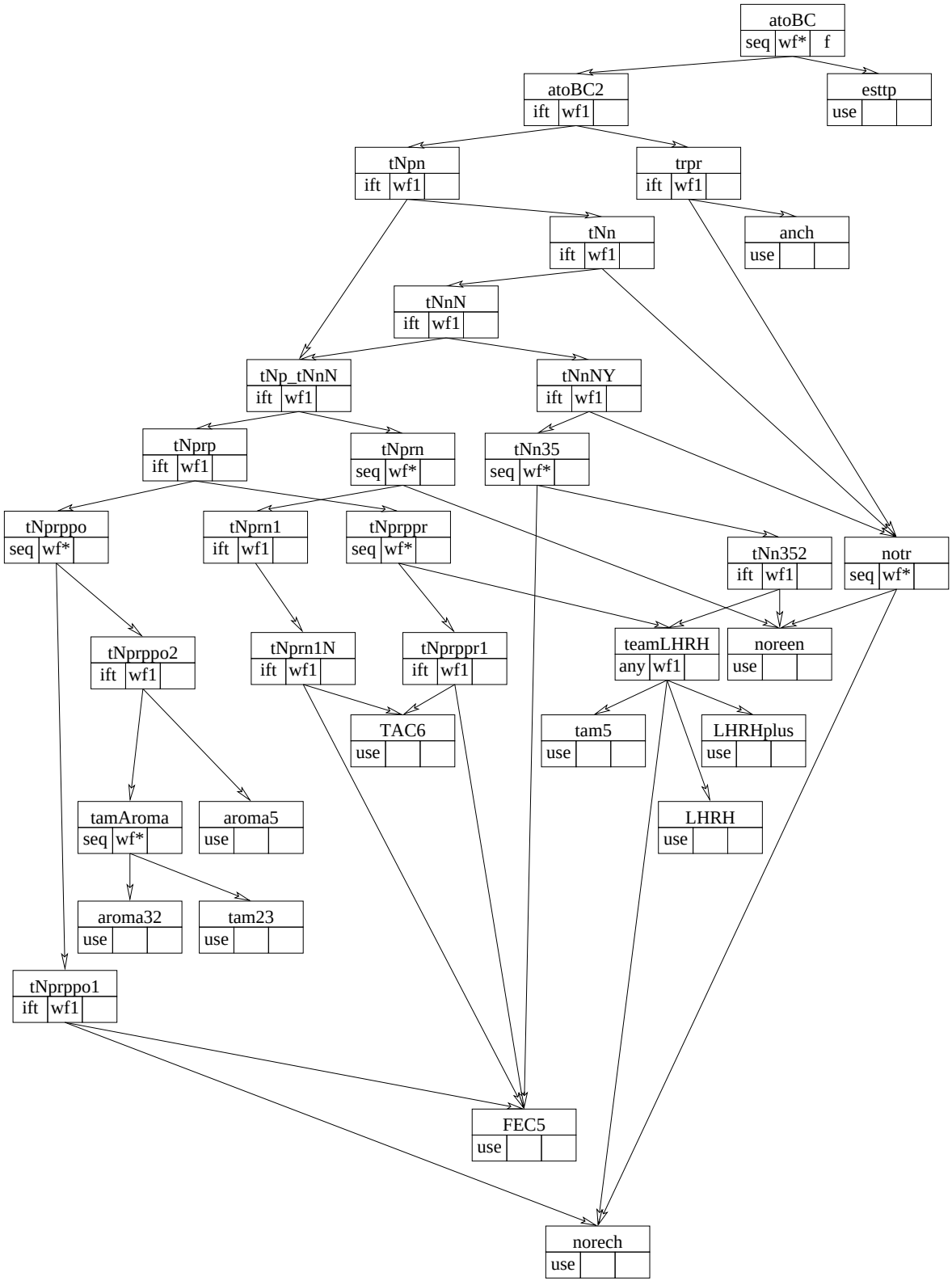


Abbildung 5.3: Asbru Modell des zweiten Kapitels

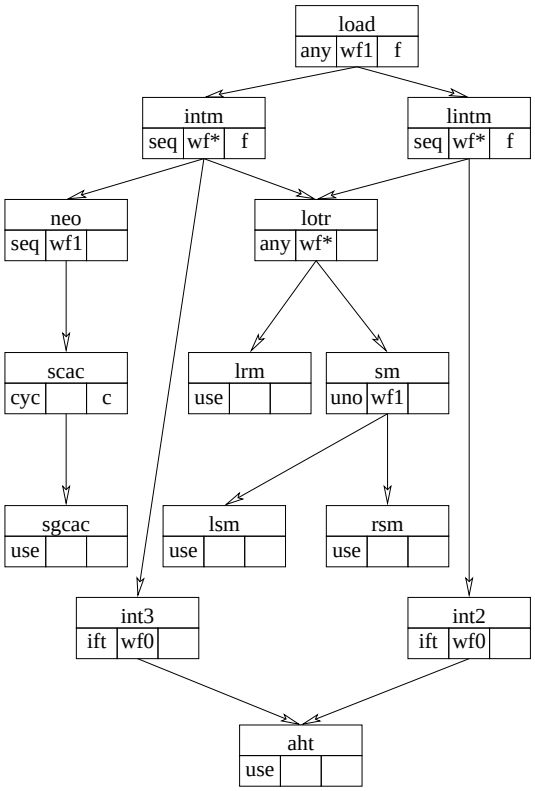


Abbildung 5.4: Asbru Modell des dritten Kapitels

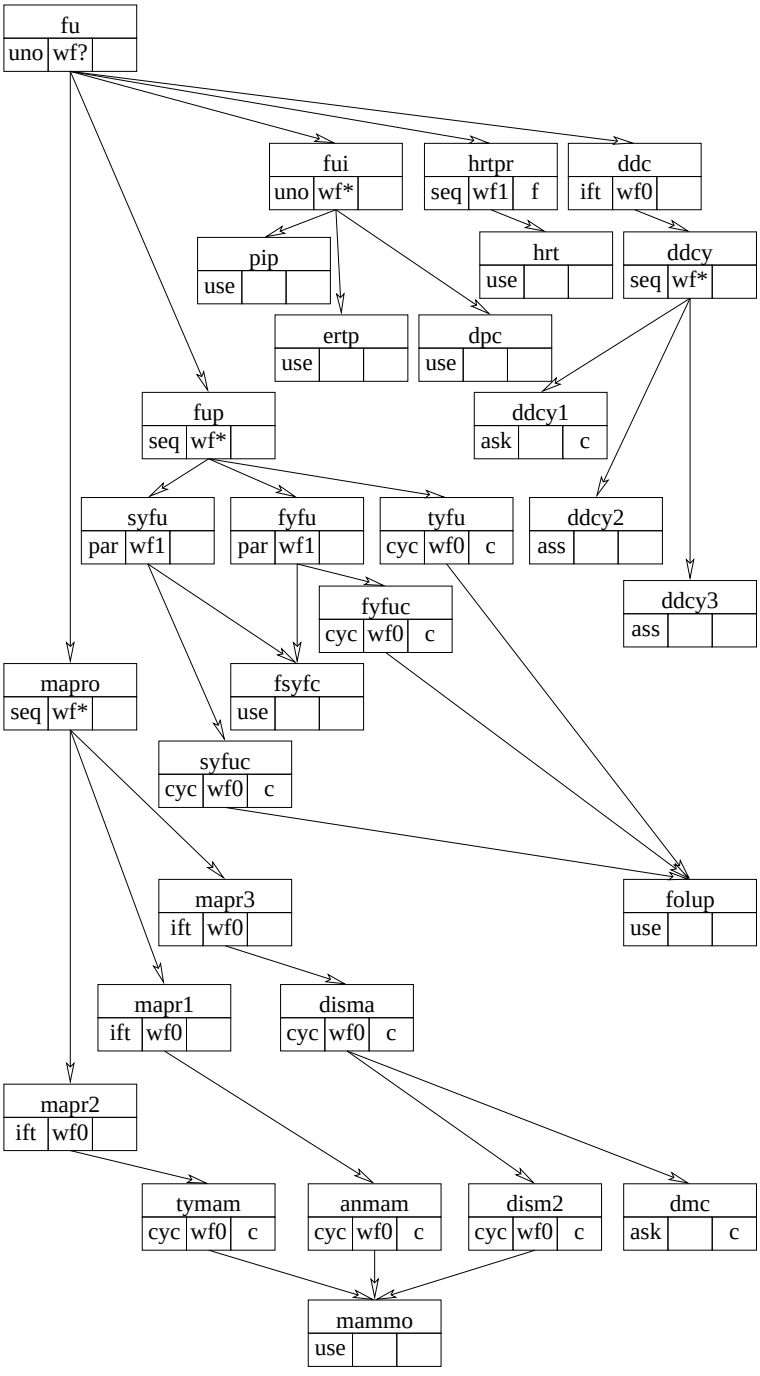


Abbildung 5.5: Asbru Modell des vierten Kapitels

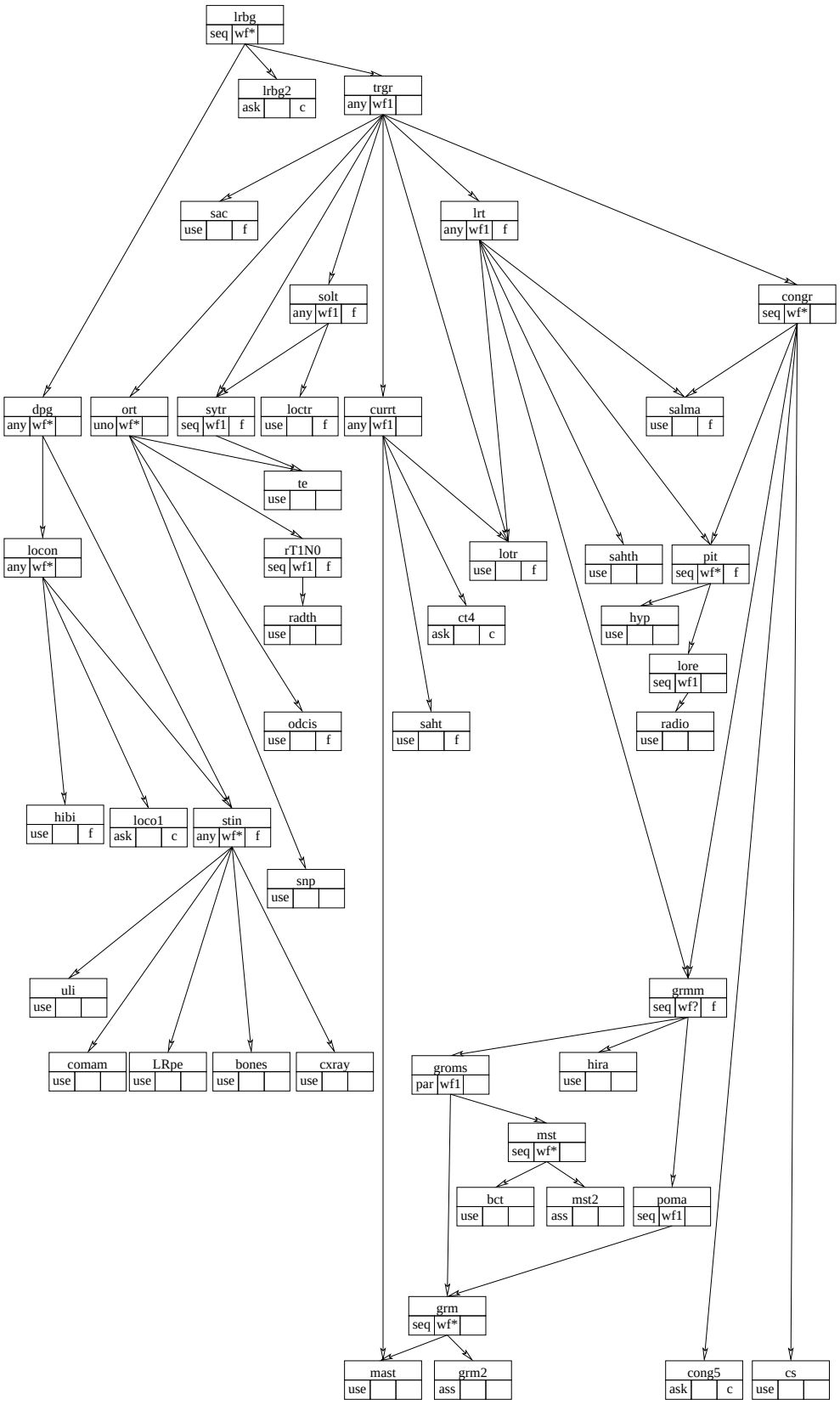


Abbildung 5.6: Asbru Modell des fünften Kapitels

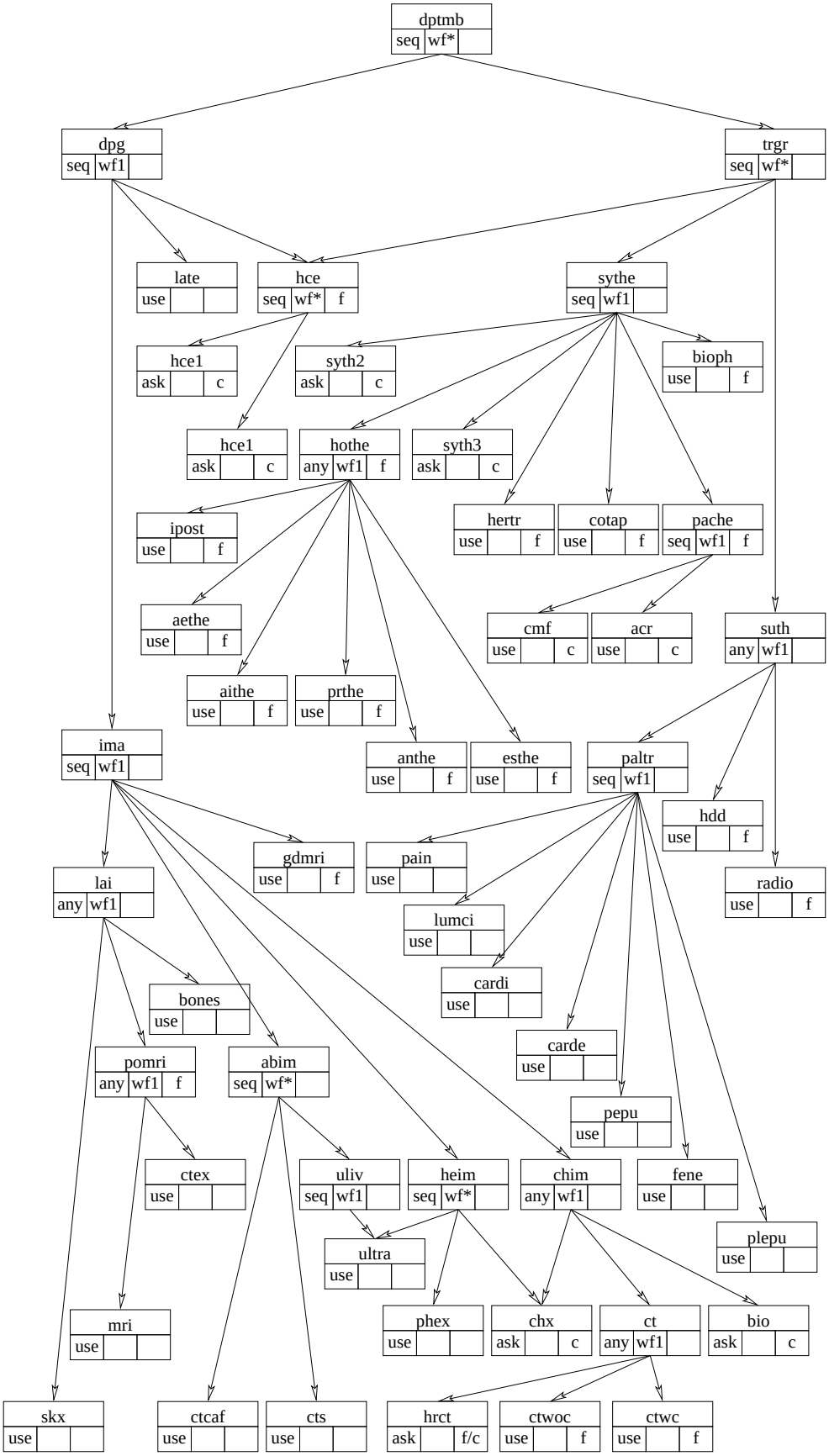


Abbildung 5.7: Asbru Modell des sechsten Kapitels

Teil II

Syntax und Semantik von Asbru

6 Syntax von Asbru

Im Folgenden wird die Syntax von Asbru vorgestellt. Dazu wird ein Top-Down Ansatz gewählt. Das bedeutet, dass jeweils die übergeordneten Konzepte vor den untergeordneten Konzepten vorgestellt werden. Begonnen wird dabei mit der Datenstruktur, die einen Asbru-Plan selbst darstellt. Den Abschluss bilden dann die einfachsten Komponenten.

Die Konzepte sollen anhand eines Beispiels motiviert werden. Dies beschreibt die Behandlung einer Erkältung.

6.1 Ein einfaches Beispiel

Anhand eines einfachen Beispiels zur Behandlung von Erkältungen werden die Konzepte von Asbru erläutert. Die Planhierarchie dieses Behandlungsplans ist in Abbildung 6.1 dargestellt. Die Behandlung der Erkältung gliedert sich in drei Komponenten, Bettruhe, die Gabe von Schmerzmitteln und Antibiotika. Prinzipiell soll ein erkälteter Patient zunächst das Bett hüten und auf Besserung warten. Treten jedoch Kopf- oder Gliederschmerzen, so soll neben der Bettruhe ein Analgetikum eingenommen werden. Wenn die Erkältung von Fieber begleitet ist, und dieses trotz Einnahme des Analgetikums weiter besteht, sollen Antibiotika eingenommen werden.

Die Behandlung mit Antibiotika gliedert sich wiederum in drei Teile. Zunächst soll anhand einer Diagnose festgestellt werden, ob die fieberauslösenden Keime resistent gegen klassisches Penicillin sind. In diesem Fall soll statt des Penicillins ein Breitband-Antibiotikum namens Amoxicillin gegeben werden. Sind die Keime nicht resistent, so soll Penicillin gegeben werden.

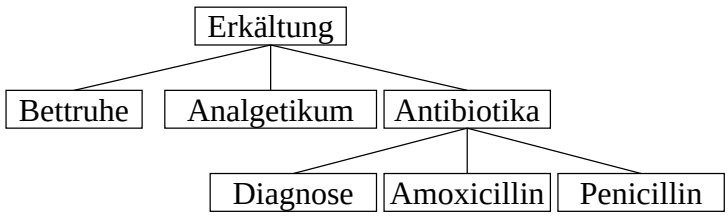


Abbildung 6.1: Behandlung einer Erkältung

6.2 Asbru-Plan Datenstruktur (asbru-plan)

Im Folgenden werden die Komponenten eines Asbru-Planes in der Reihenfolge aufgeführt, in der sie in der Plandefinition vorkommen:

Typ Der Plantyp bestimmt die Ausführungsmodalitäten des Planes. Darunter fallen hauptsächlich der Umgang mit Unterplänen, d.h. in welcher Reihenfolge diese gestartet und aktiviert werden sollen. Weiterhin kann der Plantyp auch Informationen darüber enthalten, wie lange ein Plan aktiv bleibt bevor er sich selbst beendet.

Im Beispiel wird der Plan zur Erkältung verlangen, dass alle Unterpläne gleichzeitig laufen sollen. Der Unterplan zur Gabe von Penicillin sollte so gestaltet sein, dass nur einer der Pläne jeweils gleichzeitig ausgeführt werden soll.

Planliste Die Planliste definiert die Liste aller Unterpläne eines Plans. In der Regel werden nach der Aktivierung eines Plans wenigstens einige Pläne aus dessen Planliste gestartet. Die genaue Anzahl der zu startenden Pläne kann nur zur Laufzeit ermittelt werden, da äußere Faktoren diese beeinflussen. Der Ausführungsmodus der Pläne aus der Liste, also ob die Unterpläne gleichzeitig oder nacheinander ausgeführt und aktiviert werden, hängt vom Plantyp ab.

Im Beispiel enthält die Unterplanliste des Plans `Erkältung` die Plannamen `Bettruhe`, `Analgetikum` sowie `Antibiotika`,

Bedingungen Bedingungen geben an, wie die Planausführung an bestimmten, neuralgischen Punkten fortgesetzt werden soll. Allgemein können Bedingungen zur Folge haben, dass die Ausführung des Plans verzögert wird, wenn der Plan auf einen bestimmten Zustand des Patienten warten soll. Ebenso können Bedingungen zum Abbruch eines Plans führen und dazu, dass der Plan selbst in den Zustand erfolgreicher Beendigung setzen kann. Die Auswertung von Bedingungen erfolgt teilweise nur über Zustandshistorien geschehen. Dies ist der Fall, wenn eine Bedingung so formuliert wird, dass die Auswertung vergangene Zeitpunkte umfasst. In diesem Fall wird die Auswertung eine Bedingung so oft wiederholt, bis die Bedingung entweder erfüllt ist, oder nachweisbar ist, dass sie auch in der Zukunft nicht mehr erfüllt werden kann.

Im Erkältungsbeispiel kommen verschiedene Bedingungen zum Einsatz. Im Fall des Antibiotika Plans soll anhand der Resistenz der Keime entschieden werden, welche Behandlung gestartet wird und die Antibiotika-Behandlung selbst soll nur angewendet werden, wenn Fieber mindestens drei Tage lang andauert.

Wartebedingung Zur Beendigung eines Plans können neben Bedingungen an den Zustand des Patienten auch Bedingungen an den Zustand der Unterpläne formuliert werden. Dazu dient die Wartebedingung. Diese spezifiziert Mengen von Unterplänen, die erfolgreich abgeschlossen werden müssen, damit ein Plan sich selbst erfolgreich beenden darf. Gegebenenfalls verlangt die Wartebedingung den Start weiterer Unterpläne, bevor sie einen Abschluss des Plans zulässt.

Im Beispiel sollte die Wartebedingung so formuliert sein, dass der Plan zur Gabe von Antibiotika auf keinen Fall abgebrochen werden darf. Andernfalls könnten durch den Abbruch der Behandlung resistente Keime entstehen.

Marker Zwei boolesche Marker geben weitere Informationen darüber, wie der Plan mit Unterplänen umgeht. Ein Marker gibt Auskunft, ob Unterpläne eines Plans neu gestartet werden sollen, wenn diese abgebrochen wurden. Der andere Marker definiert das Verhalten im Terminierungsfall und spezifiziert, ob der Plan vor seiner Terminierung auf die Beendigung bereits aktiver Unterplänen warten muss. Der Unterschied zur Wartebedingung ist, dass hier lediglich bereits laufende Pläne beendet werden müssen, um die Bedingung zu erfüllen. Im Fall der Wartebedingung kann es erforderlich sein, weitere Pläne zu starten, bevor die Wartebedingung erfüllt werden kann.

Die vorgestellten Komponenten werden in der Datenstruktur **asbru-plan** gespeichert. Diese ist wie folgt definiert:

asbru-plan plan ::= mk-asbru-plan(**plan-type**, `//Plantyp`
plan-list, `//Liste der Unterpläne`
asbru-condition, `//Filter-Bedingung`
asbru-condition, `//Setup-Bedingung`
asbru-condition, `//Suspend-Bedingung`
asbru-condition, `//Reactivate-Bedingung`
asbru-condition, `//Abort-Bedingung`
asbru-condition, `//Complete-Bedingung`
bool, `//wait-for optional Marker`
bool, `//retry-aborted Marker`
wait-for`//Wartebedingung`);

Dabei definiert der Datentyp **plan-type** den Plantypen. Der Datentyp wird in Abschnitt 6.3 eingeführt und genauer beschrieben. **plan-list** ist die Liste der Unterpläne, die in Abschnitt 6.5 beschrieben wird. Die sechs **asbru-condition** beschreiben nacheinander die **filter**-, **setup**-, **suspend**-, **reactivate**-, **abort**- und **complete**-Bedingungen. Dabei beschreiben die **filter**- und die **setup**-Bedingungen Voraussetzungen, unter denen der Plan anwendbar ist, die **suspend**- und die **reactivate**-Bedingungen beschreiben Situationen, in denen die Ausführung des Plans temporär unterbrochen oder wieder aufgenommen werden soll und die **abort**- sowie die **complete**-Bedingungen beschreiben Zustände, in denen der Plan terminieren soll und unterscheiden dabei zwischen dem Abbruch eines Plans und der erfolgreichen Beendigung. Die beiden **bool** beschreiben die Marker, die entscheiden, ob abgebrochene Unterpläne neu gestartet werden sollen und ob auf bereits gestartete Unterpläne gewartet wird, wenn der Plan ansonsten erfolgreich beendet werden könnte.

Sei **plan** ein **asbru-plan** mit nachstehender Definition:

plan = mk-asbru-plan($\mathbf{ptype}, \mathbf{PL}$,
$\mathbf{acond}_1, \mathbf{acond}_2, \mathbf{acond}_3$,
$\mathbf{acond}_4, \mathbf{acond}_5, \mathbf{acond}_6$,
$\mathbf{bv}_1, \mathbf{bv}_2, \mathbf{wf}$);

dann sind nachstehende postfix Selektoren definiert:

..type	:	**asbru-plan** $\mapsto$ **plan-type**
..subplans	:	**asbru-plan** $\mapsto$ **plan-list**
..filter	:	**asbru-plan** $\mapsto$ **asbru-condition**
..setup	:	**asbru-plan** $\mapsto$ **asbru-condition**
..suspend	:	**asbru-plan** $\mapsto$ **asbru-condition**
..reactivate	:	**asbru-plan** $\mapsto$ **asbru-condition**
..abort	:	**asbru-plan** $\mapsto$ **asbru-condition**
..complete	:	**asbru-plan** $\mapsto$ **asbru-condition**
..wait-for-optional	:	**asbru-plan** $\mapsto$ **bool**
..retry-aborted	:	**asbru-plan** $\mapsto$ **bool**
..wait-for	:	**asbru-plan** $\mapsto$ **wait-for**

Diese dienen der Selektion der einzelnen Daten aus einem **asbru-plan** und werden im Folgenden verwendet. Mit der oben definierten Datenstruktur **plan** vom Typ **asbru-plan** soll nun die Semantik der Selektoren definiert werden, indem das Ergebnis der Anwendung der Selektoren auf diese Datenstruktur angegeben wird.

plan.type	=	**ptype**
plan.subplans	=	**PL**
plan.filter	=	$\mathbf{acond}_1$
plan.setup	=	$\mathbf{acond}_2$
plan.suspend	=	$\mathbf{acond}_3$
plan.reactivate	=	$\mathbf{acond}_4$
plan.abort	=	$\mathbf{acond}_5$
plan.complete	=	$\mathbf{acond}_6$
plan.wait-for-optional	=	$\mathbf{bv}_1$
plan.retry-aborted	=	$\mathbf{bv}_2$
plan.wait-for	=	**wf**

Um Asbru-Pläne ihrer jeweiligen Definition zuordnen zu können, wird eine partiell definierte Funktion `asbru` verwendet. Diese ordnet Plannamen vom Typ **plan-name**, die in Abschnitt 6.6 beschrieben werden, den jeweiligen Plandefinitionen zu.

Für jede Fallstudie müssen diejenigen Stellen der `asbru` Funktion spezifiziert werden, die auf Pläne verweisen, die in der jeweiligen Fallstudie vorkommen.

`asbru` : **plan-name** $\mapsto$ **asbru-plan**

6.3 Die Plantypen Datenstruktur (plan-type)

Im Allgemeinen hat ein Asbru Plan einen oder mehrere Unterpläne. Diese Unterpläne müssen vom kontrollierenden Plan über Signale gesteuert werden. Es muss dazu für jeden Plan festgelegt werden, wie die Unterpläne genau gesteuert werden, beispielsweise, ob alle Pläne gleichzeitig oder einzeln nacheinander gestartet werden sollen. Um diese Festlegung zu treffen, ist für jeden Plan ein sogenannter Plantyp (**plan-type**) spezifiziert. Dieser Plantyp bestimmt den Ausführungsmodus der Unterpläne eines Plans. Festgelegt wird durch den Plantyp, wann die Unterpläne eines Plans gestartet werden wann sie aktiviert werden sollen. Einzelne Plantypen existieren dabei für die sequentielle Ausführung aller Unterpläne oder die gleichzeitige Ausführung.

plan-type ptype	::=	„`sequential`“
	|	„`anyorder`“
	|	„`parallel`“
	|	„`unordered`“
	|	`ifThenElse`
	|	„`onabort`“
	|	`cyclical`
	|	„`user`“
	|	`ask`

Die wichtigsten Plantypen sind `sequential`, `anyorder`, `parallel` und `unordered`. Von diesen Plänen sind `sequential` und `anyorder` sequentiell, was bedeutet, dass maximal einer der Unterpläne gleichzeitig in der `Execution`-Phase sein kann. Ein `sequential` kontrollierter Plan startet seine Unterpläne in der durch die Unterplanliste angegebene Reihenfolge (=deterministisch), während ein `anyorder` Plan jeweils einen zufällig ausgewählten Unterplan aktiviert (=indeterministisch). `parallel` und `unordered` kontrollierte Pläne starten jeweils alle Unterpläne auf einmal, wobei `parallel` die Unterpläne an definierten Punkten synchronisiert, während `unordered` keinen weiteren Synchronisierungspunkt der Unterpläne festlegt.

Die `ifThenElse`, `onabort`, und `cyclical` Plankontrolltypen beschreiben die Kontrolle von einem bzw. zwei Unterplänen. Dabei entscheidet der `ifThenElse` Kontrolltyp, dessen Syntax in Abschnitt 6.3.1 dargestellt wird, anhand des aktuellen Zustands, welchen der beiden Unterpläne er ausführt. Der `onabort` Kontrolltyp hingegen definiert zwei Unterpläne, wobei der zweite gestartet wird, wenn der erste Unterplan seine Auswertung abbrechen muss. Der `cyclical` Kontrolltyp hingegen kennt nur einen Unterplan, der wiederholt ausgeführt wird, bis eine festgeschriebene Anzahl von Durchläufen erreicht wird. Seine Syntax wird in Abschnitt 6.3.3 beschrieben.

Spezialtypen sind `user` sowie `ask` kontrollierte Pläne. Diese haben keine Unterpläne und stellen somit die Endpunkte der Asbru Hierarchie dar. Dabei symbolisieren `user` kontrollierte Pläne die Anwendung medizinischer Eingriffe, sind also als Signal an die Umgebung zu verstehen medizinische Behandlungen durchzuführen. Umgekehrt sind `ask` Pläne ebenso ein Signal an die Umgebung, allerdings um Messungen durchzuführen, die für den weiteren Ablauf des Behandlungsplanes notwendig sind. Da diese Plantypen für Signale an die Umgebung stehen, kontrollieren sie keine Unterpläne.

Im Beispiel kommen mehrere Plantypen vor. Beispielsweise ist der `Diagnose`-Plan ein `ask` Plan, da er Daten vom Arzt erfragt und dem System bereitstellt. `Bettruhe` ist hingegen ein `user`-Plan, da er eine atomare Behandlung beschreibt. `Erkältung` soll die Behandlungspläne unabhängig voneinander starten. Daher handelt es sich hierbei um einen `unordered` kontrollierten Plan. Der `Antibiotika`-Plan soll nur genau einen Plan gleichzeitig starten, jedoch in beliebiger Reihenfolge. Daher ist dieser Plan `anyorder`-kontrolliert.

6.3.1 Entscheidungs-Plantyp (`ifThenElse`)

Die Handlungen des `ifThenElse` Kontrolltyps sind abhängig von einem Konditional, das als Teil der Datenstruktur des Kontrolltypen spezifiziert wird. Daher ist der Plantyp `ifThenElse` in dieser Grammatik kein Terminalsymbol, sondern beschreibt selbst wieder eine Datenstruktur, die das Konditional mitführt, anhand dessen entschieden wird, welcher Unterplan ausgeführt wird.

`ifThenElseite` ::= mk-ite(**conditional**)

Der hierbei verwendet Datentyp **conditional** wird in Abschnitt 6.9 vorgestellt. Er entspricht einer Bedingung, die an den aktuellen Zustand gestellt wird. Ein Beispiel für eine solche Bedingung ist die Frage, ob der Patient gegenwärtig an Fieber leidet.

Um die Bedingung der `ifThenElse` Kontrolle zu selektieren, existiert eine Funktion, die die Bedingung aus dem Datentyp extrahiert. Diese hat nachstehende Syntax:

.conditional : `ifThenElse` $\mapsto$ **conditional**

Die Semantik dieser Funktion wird mit nachstehender Gleichung festgelegt.

mk-ite(**cond**)**.conditional** = **cond**

Ein `ifThenElse` kontrollierter Plan hat einen oder zwei Unterpläne. Bei der Ausführung des Planes wird nach der Aktivierung des Plans einmalig überprüft, ob das Konditional wahr ist oder nicht. Im ersten Fall wird der erste Unterplan gestartet. Ist das Konditional nicht wahr, so wird der zweite Unterplan gestartet, sofern er existiert. Hat ein `ifThenElse` kontrollierter Plan nur einen Unterplan und das Konditional ist bei der Auswertung nicht wahr, so beendet der Plan seine Ausführung.

6.3.2 Dateneingabe-Plantyp (ask)

Ein `ask` kontrollierter Plan stellt ein Signal an die Umgebung dar. Dieses Signal bedeutet, dass das System Daten anfordert. Das Asbru-System ist dabei als eine Art von Software zu verstehen, die den Arzt bei seinen Handlungen unterstützt und anleitet. Diese Unterstützung beschränkt sich auf die Auswertung von Bedingungen sowie Anweisungen, wie der Arzt zu verfahren hat. Das System kann nicht direkt mit dem Patienten interagieren. Somit obliegt auch die Entnahme von Blut oder das Messen eines Pulses dem Arzt, der mittels `ask` Plänen auf fehlende Werte aufmerksam gemacht werden soll.

Ein `ask` kontrollierter Plan muss dazu spezifizieren, welchen Wert das System vom Arzt erfragt. Zu diesem Zweck wird der Kontrolltyp parametrisiert. Der Parameter ist eine Zeichenkette, die den einzulesenden Wert beschreiben muss.

`askaskObj` ::= mk-ask(**string**)

Dabei wird die Zeichenkette, die den einzulesenden Parameter beschreibt mittels einer Funktion selektiert:

.paramName : `ask` $\mapsto$ **string**

mk-ask(**SK**)**.paramName** = **SK**

6.3.3 Zyklischer-Plantyp (cyclical)

Ein zyklischer Plankontrolltyp hat genau einen Unterplan. Dieser wird wiederholt ausgeführt, bis eine im Datentyp spezifizierte Anzahl erfolgreicher Beendigungen dieses Unterplans erreicht wird. Ist dies der Fall, wird der übergeordnete Plan beendet.

In der Spezifikation dieses Plantyps werden mehrere Parameter angegeben. Diese geben ein Startintervall an, in dem der Unterplan gestartet werden kann. Damit spätere Iterationen der Ausführung des Unterplans möglich sind, wird eine Wiederholfrequenz angegeben, die dieses Startintervall verschiebt. Soll beispielsweise eine Chemotherapie am jeweils ersten eines Quartals stattfinden, so wird das Startintervall so festgelegt, dass es am 1.1. um 8 Uhr morgens beginnt und um 16 Uhr endet. Die Frequenz wird in diesem Fall drei Monate festgelegt.

Um diese Spezifikation technisch möglich zu machen, wird der `cyclical` Kontrolltyp mit vier Werten parametrisiert. Ein Parameter beschreibt das eigentliche Startintervall, wozu eine Wert vom Typ **abstract-time-annotation** verwendet wird. Ein ganzzahliger Parameter gibt die Wiederholfrequenz an und eine natürliche Zahl speichert die Zahl der notwendigen Wiederholungen. Zusätzlich wird aus technischen Gründen noch ein Parameter angegeben, der das Startintervall sowie alle Wiederholungen desselben um einen fixen Zeitwert verschiebt.

Diese Datentypen werden in einer Datenstruktur vom Typ `cyclical` gespeichert, deren Syntax wie nachfolgend spezifiziert ist:

`cyclicalcyclicalObj` ::= mk-cyc(**abstract-time-annotation**, **int**, **nat**, **int**)

Es werden Selektoren definiert, um die jeweiligen Werte aus dem `cyclical` Objekt zu selektieren. Deren Syntax wie folgt festgelegt:

.ata : `cyclical` $\mapsto$ **abstract-time-annotation**
.offset : `cyclical` $\mapsto$ **int**
.frequncy : `cyclical` $\mapsto$ **nat**
.rounds : `cyclical` $\mapsto$ **int**

Um die Semantik dieser Selektoren zu definieren, werden nachstehende Semantik Definitionen verwendet. Dazu sei `cyclicalObj` ein Objekt vom Typ `cyclical` mit `cyclicalObj` = mk-cyc(**ata**, $\mathbf{i}_1$, **n**, $\mathbf{i}_2$) dann gilt:

`cyclicalObj.ata`	=	**ata**
`cyclicalObj.offset`	=	$\mathbf{i}_1$
`cyclicalObj.frequncy`	=	**n**
`cyclicalObj.rounds`	=	$\mathbf{i}_2$

Das Beispiel kann erweitert werden, so dass es einen zyklischen Plan enthält. Die Modellierung der Gabe der beiden Antibiotika kann dahingehend erweitert werden, dass statt einem `user`-kontrollierten Plan ein `cyclical`-kontrollierter Plan verwendet wird. Dieser spezifiziert dann, dass über einen Zeitraum von zehn Tagen alle acht Stunden eine Tablette Amoxicillin bzw. alle zwölf Stunden eine Tablette Penicillin genommen werden muss.

6.4 Abstrakte Zeit-Annotationen (abstract-time-annotation)

Eine abstrakte Zeit-Annotation (**abstract-time-annotation**) ist ein Datentyp zur Formalisierung zeitlicher Aspekte. Der Datentyp speichert einen Anfangszeitraum, einen Endzeitraum sowie ein Intervall, das alle zugelassenen Längen beschreibt. Eine **abstract-time-annotation** spezifiziert damit Mengen von Intervallen. Alle Intervalle dieser Menge haben Anfangspunkte, Endpunkte und Längen, die der Spezifikation durch die **abstract-time-annotation** entsprechen müssen. Um maximale Flexibilität bei der Spezifikation zu erreichen wird ein Referenzzeitpunkt in der **abstract-time-annotation** angegeben, zu dem die Start- und Endzeiträume relativ angegeben sind.

Dieser Referenzzeitpunkt darf in Form einer abstrakten Uhrzeit angegeben sein und kann somit nicht nur absolute Zeiten speichern, sondern auch Planzustandswechsel anderer Pläne beschreiben sowie angeben, dass die Zeit-Annotation relativ zum aktuellen Zeitpunkt sein soll.

Eine abstrakte Zeit-Annotation kann beispielsweise eingesetzt werden, um Therapieerfolg zu überwachen. Dieser könnte beispielsweise angegeben sein, als eine Senkung von Fieber für mindestens zwölf Stunden beginnend drei bis vier Stunden nach einer Medikamentengabe. Der Referenzzeitpunkt ist in diesem Beispiel der Moment der Medikamentengabe und gesucht sind alle Intervalle, die drei bis vier Stunden nach diesem Referenzzeitpunkt beginnen und mindestens zwölf Stunden andauern.

Um solche Spezifikationen zu ermöglichen sind abstrakte Zeit-Annotationen als sieben-Tupel spezifiziert. Die beiden ersten spezifizierten Werte geben das relative Startintervall an, also in welchem Zeitraum die gesuchten Intervalle beginnen müssen. Die nächsten beiden Werte spezifizieren das Endintervall und das dritte Wertepaar spezifiziert die Länge. Der letzte Wert gib den Referenzpunkt an.

Einen Sonderfall der abstrakten Zeit-Annotationen bilden die **now_ata**-Zeit-Annotationen. Diese sind quasi Auswertungsneutral, was bedeutet, dass eine **now_ata** praktisch den aktuellen Zeitpunkt isoliert betrachtet. Dies wird relevant, wenn Zeit-Annotationen als Teil von Bedingungen betrachtet werden und die Semantik der Bedingungen festgelegt wird. Die **now_ata**-Zeit-Annotationen werden immer dann verwendet, wenn ein Plan beliebig lange auf ein Ereignis warten soll. Zum Beispiel könnte ein Plan zur automatischen Defilibration bei Kammerflimmern so modelliert sein, dass seine **setup**-Bedingung fordert, dass Kammerflimmern vorliegt und dies mit einer **now_ata** verknüpft wird.

abstract-time-annotationata ::= mk-ta(**int-bottom**, **int-bottom**,
int-bottom, **int-bottom**,
int, **int-bottom**,
abstract-asbru-clock)
| **now_ata**

Um mit abstrakten Zeit-Annotationen arbeiten zu können, werden einige Selektoren für die Komponenten der **abstract-time-annotation**s definiert. Diese werden hier aufgeführt. Die Selektoren haben dabei selbsterklärende, jedoch abgekürzte Namen. Das Kürzel „ess“ steht für den „**e**arliest **s**tarting **s**hift“, also den frühesten relativen Startzeitpunkt. Dessen Gegenstück ist der „lss“ (latest starting shift), der späteste relativen Startzeitpunkt. Analog dazu sind „efs“ und „lfs“ als frühste bzw. späteste relative Endzeitpunkte benannt. Die Abkürzungen „minDu“ und „maxDu“ stehen für die „minimum duration“ und „maximum duration“. Zuletzt steht der „refPoint“ für den „reference point“, der den Referenzzeitpunkt angibt.

.ess : **abstract-time-annotation** ↦ **int-bottom**
.lss : **abstract-time-annotation** ↦ **int-bottom**
.efs : **abstract-time-annotation** ↦ **int-bottom**
.lfs : **abstract-time-annotation** ↦ **int-bottom**
.minDu : **abstract-time-annotation** ↦ **int**
.maxDu : **abstract-time-annotation** ↦ **int-bottom**
.refPoint : **abstract-time-annotation** ↦ **abstract-asbru-clock**

Gegeben sei eine **abstract-time-annotation ata** mit **ata** = (**bi**$_1$, **bi**$_2$, **bi**$_3$, **bi**$_4$, **i**, **bi**$_5$, **aac**). Die Semantik der oben aufgeführten Selektoren wird durch nachstehende Gleichungen definiert:

ata.ess = **bi**$_1$
ata.lss = **bi**$_2$
ata.efs = **bi**$_3$
ata.lfs = **bi**$_4$
ata.minDu = **i**
ata.maxDu = **bi**$_5$
ata.refPoint = **aac**

Abstrakte Zeit-Annotationen werden im Beispiel verwendet, um Planabhängigkeiten zu modellieren. Beispielsweise darf der `Amoxicillin`-Plan nur gestartet werden, wenn die `Diagnose` abgeschlossen wurde und feststeht, dass die Keime resistent gegen Penicillin sind. Auch darf der `Antibiotika`-Plan selbst nur gestartet werden, wenn Fieber für mindestens drei Tage nachgewiesen werden kann. Auch dies wird mittels abstrakter Zeit-Annotationen modelliert.

6.5 Unterplanlisten (plan-list)

Die Planliste dient in Asbru dazu, die Menge und Reihenfolge der Unterpläne eines Planes zu beschreiben. Die Reihenfolge der Unterpläne ist beispielsweise für die `sequential` Kontrolle relevant, die diese auswertet. Andere Plankontrolltypen, wie etwa `anyorder` berücksichtigen die Reihenfolge nicht, sondern beachten nur die Menge der angegebenen Pläne. Die Planlisten sind bei wohlgeformten Asbru-Plänen duplikatfrei. Die in Abschnitt 9.2 und 9.3 angegebenen Regeln zur Ausführung von Asbru stellen die Semantik von Asbru-Planhierarchien nur dann korrekt dar, wenn die Unterplanlisten duplikatfrei sind.

plan-list PL ::= „[]“
| **plan-name** + **plan-list**

Für die Listen existieren Selektoren, um die einzelnen Plannamen der Listen selektieren zu können. Standardfunktionen für Listen **.first** und **.rest** werden definiert, die den Zugriff auf das erste Listenelement bzw. den Listenrest erlauben. Syntax und Semantik dieser Selektoren ist nachstehend definiert:

.first : **plan-list** $\mapsto$ **plan-name**
.rest : **plan-list** $\mapsto$ **plan-list**

(**SK** + **PL**)**.first** = **SK**
(**SK** + **PL**)**.rest** = **PL**

6.6 Plannamen (plan-name)

Der Planname **plan-name** steht für die unterschiedlichen Namen der Pläne. Da die Plannamen mittels der `asbru` Funktion auf Plandefinitionen abgebildet werden, dürfen unterschiedliche Pläne in der gleichen Fallstudie nicht gleich benannt sein.

plan-name SK

6.7 Asbru-Bedingungen (asbru-condition)

Eine **asbru-condition** umfasst eine Bedingung (**condition**), die in Abschnitt 6.8 vorgestellt wird, sowie Möglichkeiten für die Benutzerinteraktion. Eine medizinische Bedingung im Sinne einer **condition** beschreibt einen bestimmten Zustand, beispielsweise einen Patienten, der seit mehr als drei Tagen mittleres Fieber hat. Das medizinische Personal soll manchmal in der Lage sein, bestimmte Bedingungen zu übersteuern. Damit könnte ein Bedingung mit Zustimmung des Arztes als wahr ausgewertet werden, obwohl das Kriterium nicht erfüllt ist. Umgekehrt sollen auch erfüllte Bedingungen bis zu einer Zustimmung des medizinischen Personals blockiert werden können.

Um dies zu erreichen, existieren in dem **asbru-condition** Datentyp zwei Marker. Einer dieser Marker legt fest, ob es prinzipiell möglich ist, die Auswertung der Bedingung zu übersteuern. Der andere Marker bestimmt, ob ein verzögern der Bedingung erlaubt ist. Damit hat der Datentyp die nachstehende Form:

asbru-condition acond ::= mk-acond(**condition**, **bool**, **bool**)

Um die einzelnen Konstrukte des Datentyps selektieren zu können, sind folgende Funktionen definiert:

..condition : **asbru-condition** $\mapsto$ **condition**
..confirm : **asbru-condition** $\mapsto$ **bool**
..override : **asbru-condition** $\mapsto$ **bool**

Sei **acond** eine **asbru-condition** mit **acond** = mk-acond(**cond**, $\mathbf{bv}_1$, $\mathbf{bv}_2$), wobei $\mathbf{bv}_1$ und $\mathbf{bv}_2$ boolesche Marker sind. Die Semantik der Selektoren wird damit durch nachstehende Gleichungen definiert:

acond .condition = **cond**
acond .confirm = $\mathbf{bv}_1$
acond .override = $\mathbf{bv}_2$

Dabei selektiert der Term **acond .confirm** den Marker, der festlegt, ob eine manuelle Bestätigung des Arztes vorliegen muss, also ob die Bedingung verzögerbar ist. **acond .override** bestimmt, ob die Bedingung übersteuert werden kann, die Bedingung also mit entsprechender Bestätigung des Arztes immer zu wahr auswerten kann.

Im Beispiel wird der Start der `Antibiotika`-Plan durch eine Asbru-Bedingung in seinem Startverhalten beeinflusst. Der Plan soll nur starten, wenn für mindestens drei Tage Fieber nachgewiesen werden kann oder, wenn der Arzt die Erkältung als so schlimm ansieht, dass sofort Antibiotika gegeben werden müssen. Dies kann der Fall sein, wenn der Patient die Erkältung verschleppt hat und in sehr schlechtem Zustand den Arzt aufsucht. Um dies zu modellieren wird der Marker zur Übersteuerung der Bedingung verwendet.

6.8 Bedingungen (condition)

Eine Bedingung (**condition**) beschreibt eine medizinische Bedingung, beispielsweise Blutdruck von höchstens 120 mm/hg. Manchmal muss eine Bedingung auch an zeitliche Aspekte geknüpft sein, etwa wenn niedriger Blutdruck für einen Mindestzeitraum wahr sein muss. Um dies spezifizieren zu können, bestehen Bedingungen aus einem Konditional (**conditional**, siehe 6.9), das die eigentliche Bedingung spezifiziert. Optional kann dazu eine **abstract-time-annotation** angegeben werden. Dieser Datentyp bietet eine Möglichkeit, Bedingungen an die Zeit zu formulieren. Ist eine **abstract-time-annotation** angegeben, so wird das Konditional der Bedingung nicht über dem aktuellen Zustand ausgewertet, sondern über dem von der **abstract-time-annotation** vorgegebenem Intervall. Mehrere Bedingungen können durch logische Kombination mittels „und“ und „oder“ verknüpft werden.

condition cond ::= mk-cond(**conditional**)
| mk-cond(**conditional**, **abstract-time-annotation**)
| **condition** _and **condition**
| **condition** _or **condition**

Die Semantik von Bedingungen wird durch die Prädikate **satisfied** und **satisfiable** definiert. Bedingungen können so definiert sein, dass sie nicht über punktuelle Zustände sondern über Zeiträume auswerten. Damit entsteht die Möglichkeit, dass eine Auswertung einer Bedingung noch nicht möglich ist, da die Bedingung über einem Zeitraum ausgewertet werden soll, der noch in der Zukunft liegt. Soll beispielsweise vier Stunden nach Behandlungsbeginn eine Besserung eintreten, kann dies zwei Stunden nach Behandlungsbeginn noch nicht ausgewertet werden. Das Prädikat **satisfied** berechnet, ob eine Bedingung erfüllt ist, das Prädikat **satisfiable** berechnet, ob eine Möglichkeit besteht, die Bedingung in der Zukunft zu erfüllen.

6.9 Konditional (conditional)

Ein **conditional** formalisiert eine einzelne Bedingung, die von zeitlichen Aspekten entkoppelt ist. Ein Beispiel dafür ist die Spezifikation, dass der Patient Fieber hat, das nicht höher als 39 Grad Celsius sein soll.

conditional cond

6.10 Der Zahlentyp int-bottom (int-bottom)

Asbru erlaubt zeitliche Aspekte in der **abstract-time-annotation** undefiniert zu lassen. Beispielsweise kann in einer abstrakten Zeit-Annotation das Ende des Ereignisses unspezifiziert gelassen werden, solange nur eine bestimmte Dauer eingehalten wird. Um diese Anforderung effizient umzusetzen, wird eine Erweiterung der ganzen Zahlen spezifiziert, die neben den konkreten Zahlenwerten auch das Element „⊥“ enthält.

int-bottom bi ::= **int**
| „⊥“

Auf den **int-bottom** werden die nachstehenden Rechenfunktionen definiert:

$. + . \;:\; \textbf{int-bottom} \times \textbf{int-bottom} \mapsto \textbf{int-bottom};$
$. - . \;:\; \textbf{int-bottom} \times \textbf{int-bottom} \mapsto \textbf{int-bottom};$
$. * . \;:\; \textbf{nat} \times \textbf{int-bottom} \mapsto \textbf{int-bottom};$
$. < . \;:\; \textbf{int-bottom} \times \textbf{int-bottom} \mapsto \textbf{bool};$
$. > . \;:\; \textbf{int-bottom} \times \textbf{int-bottom} \mapsto \textbf{bool};$
$. \leq . \;:\; \textbf{int-bottom} \times \textbf{int-bottom} \mapsto \textbf{bool};$
$. \geq . \;:\; \textbf{int-bottom} \times \textbf{int-bottom} \mapsto \textbf{bool};$

Die Semantik dieser Funktionen wird zurückgeführt auf die Standardsemantik der ganzen Zahlen, soweit dies möglich ist. Für die Addition, Subtraktion und Multiplikation in Verbindung mit dem ⊥ Element gelten nachstehende Definitionen:

$\bot + \textbf{bi} = \bot;$
$\textbf{bi} + \bot = \bot;$
$\bot - \textbf{bi} = \bot;$
$\textbf{bi} - \bot = \bot;$
$\bot * \textbf{bi} = \bot;$
$\textbf{bi} * \bot = \bot;$

Die Semantik der Vergleichsoperatoren ist undefiniert, sobald ⊥ Elemente beteiligt sind.

6.11 Abstrakte Referenzpunkte (abstract-asbru-clock)

Eine abstrakte Uhr (**abstract-asbru-clock**) beschreibt ein Konzept, das es erlaubt, Planzustandsübergänge zu referenzieren. Damit ist es möglich, statisch den Zeitpunkt eines Ereignisses zu spezifizieren, der erst zur Laufzeit bekannt wird. Die abstrakte Uhr erlaubt dazu, neben der Angabe einer absoluten Uhrzeit, die Konstrukte **enter** und **leave** zu verwenden. Diese Konstrukte sind parametrisiert und erhalten als Parameter einen Plannamen **plan-name** sowie einen Planzustand **plan-state**.

Ein **enter** Konstrukt steht für den Zeitpunkt, zu dem der angegebene Plan in den angegebenen Planzustand wechselt. Umgekehrt beschreibt das **leave** Konstrukt den Zeitpunkt, zu dem der spezifizierte Planzustand verlassen wurde. Ein letztes Konstrukt ***now*** beschreibt den jeweils aktuellen Zeitpunkt. Bei der Auswertung wird dieser durch die aktuelle Uhrzeit ersetzt werden.

abstract-asbru-clock aac ::= **int-bottom**
| **enter**(**instance-name**, **plan-state**)
| **leave**(**instance-name**, **plan-state**)
| ***now***;

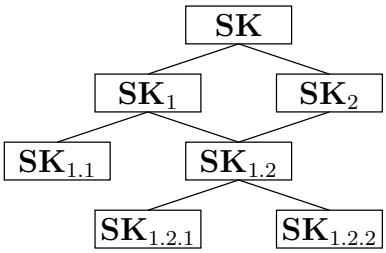


Abbildung 6.2: Notwendigkeit von Planinstanzen

6.12 Planinstanznamen (instance-name)

Asbru Planhierarchien, die in dieser Arbeit betrachtet werden, lassen sich als zyklenfreie Graphen darstellen, deren Knoten die einzelnen Pläne, und deren Kanten die Plan/Unterplan Beziehungen darstellen. Es kann vorkommen, dass ein Plan von mehreren übergeordneten Plänen gestartet wird. Dies wird in Abbildung 6.2 dargestellt. Dies kann mehrere Probleme verursachen. Angenommen der Plan mit dem Plannamen $\mathbf{SK}_1$ startet den Unterplan $\mathbf{SK}_{1.2}$ zuerst. Dann wird der Plan $\mathbf{SK}_2$ mit einer Situation konfrontiert, in der einer seiner Unterpläne gestartet wurde, obwohl er dies nicht selbst veranlasst hat. Ein solcher Fall ist jedoch nicht vorgesehen und das daraus resultierende Verhalten unspezifiziert. Durch diese Randfälle ist es nicht ausreichend, alleine Plannamen zu betrachten. Im Beispiel ist es notwendig, dass mehrere Instanzen des Planes $\mathbf{SK}_{1.2}$ unterschieden werden können und jede dieser Instanzen die Information in sich trägt, von wem sie gestartet wurde. Dadurch wird vermieden, dass Konflikte durch parallel laufende Instanzen entstehen.

Um dies zu erreichen werden Planinstanznamen eingeführt. Planinstanznamen speichern den Plannamen eines Plans. Gleichzeitig wird im Planinstanznamen auch der Plan gespeichert der für den Aufruf des Plans verantwortlich ist. Das Beispiel zeigt dabei, dass es dazu nicht reicht, nur ein Paar von Plannamen zu verwenden. Der Plan $\mathbf{SK}_{1.2}$ beispielsweise kann durch zwei übergeordnete Pläne aufgerufen werden. Seine Unterpläne brauchen also Informationen darüber von welcher Planinstanz von $\mathbf{SK}_{1.2}$ aus sie aufgerufen wurden.

Zunächst wird nun der Typ der Instanznamen angegeben, im Anschluss daran werden dann Funktionen definiert, um mit Planinstanznamen umgehen zu können.

Der Typ der Planinstanznamen wird wie folgt definiert:

instance-name iname

Um einen Planinstanznamen generieren zu können, wird eine Funktion verwendet, die den Plannamen des Plans sowie den Instanznamen des übergeordneten Plans zu einem neuen Planinstanznamen umwandelt.

nameGen : **plan-name** $\times$ **instance-name** $\mapsto$ **instance-name**

Nicht jeder Plan wird von einem anderen Plan aufgerufen. In jeder Fallstudie gibt es einen obersten Plan, der von niemandem aufgerufen wird. Bei diesem Plan kann der obige Konstruktor für den Instanznamen nicht verwendet werden. Daher gibt es einen zweiten Konstruktor, der den Instanznamen eines Plans generiert, der keinen übergeordneten Plan hat:

superGen : **plan-name** $\mapsto$ **instance-name**

Aus einem Planinstanznamen soll der ursprüngliche Planname extrahiert werden können. Dies ist beispielsweise notwendig um mittels der `asbru` Funktion auf die Plandefinition zu-

greifen zu können. Dazu wird die nachstehende Funktion definiert, die Instanznamen auf Plannamen abbildet:

nameSel : **instance-name** $\mapsto$ **plan-name**

Der definierte Namensselektor sollte sich „vernünftig“ verhalten. Wird beispielsweise mittels der **nameGen** Funktion ein Instanzname **iname** für einen Plan mit dem Namen **SK** erzeugt, so sollte die Anwendung von **nameSel** auf diesen Instanznamen wieder den ursprünglichen Plannamen **SK** zurückliefern. Weiterhin soll es auch möglich sein, den Namen eines Planes zu selektieren, der keinen übergeordneten Plan hat. Auch in diesem Fall soll sich der Selektor „vernünftig“ verhalten. Dies wird nachstehend formal als Forderung an diese Funktionen gefasst:

$$\begin{aligned} \textbf{nameSel}(\textbf{nameGen}(\textbf{SK}, \textbf{iname})) &= \textbf{SK} \\ \textbf{nameSel}(\textbf{superGen}(\textbf{SK})) &= \textbf{SK} \end{aligned}$$

Für die Ausführung eines Plans ist der Zustand des übergeordneten Plans wichtig. Beispielsweise muss sich ein Plan abbrechen, wenn er feststellen kann, dass sein übergeordneter Plan nicht mehr läuft. Um dies feststellen zu können, muss der Plan in der Lage sein, den Instanznamen des übergeordneten Plans herauszufinden. Dazu wird eine Funktion definiert, die aus dem Instanznamen eines Plans den Instanznamen des übergeordneten Plans herausselektiert. Diese heißt **parSel**:

parSel : **instance-name** $\mapsto$ **instance-name**

Analog zu der **nameSel** Funktion muss auch für die **parSel** Funktion eine Plausibilitätsbedingung gelten. Wird ein Instanzname aus dem Plannamen **SK** und dem Instanznamen des übergeordneten Plans **iname** erzeugt und auf das Ergebnis dann die **parSel** Funktion angewendet, so soll der ursprüngliche Instanzname zurückgeliefert werden. Dies wird formal durch nachstehende Gleichung gefasst:

$$\textbf{parSel}(\textbf{nameGen}(\textbf{SK}, \textbf{iname})) = \textbf{iname}$$

Im Beispiel ist der Instanzname des `Amoxicillin` Planes:
nameGen(`Amoxicillin`, **nameGen**(`Antibiotika`, **superGen**(`Erkältung`))).

6.13 Planzustände (plan-state)

Asbru-Pläne folgen einem Zustandsübergangsmodell, das in Abschnitt 3.1 erläutert wird. In diesem Modell werden Asbru-Pläne nach ihrem Start zunächst in eine `Selection`-Phase versetzt. In dieser wird die prinzipielle Anwendbarkeit des Plans überprüft. Nach Abschluss der Evaluierung kann der Plan in die `Execution`-Phase versetzt werden. Der Plan kann außerdem terminieren und dabei einen von drei Terminierungszuständen einnehmen.

Alle möglichen Zustände, die ein Plan erreichen kann, werden vom Datentyp **plan-state** zusammengefasst.

```
plan-state PS ::= inactive
                | considered
                | possible
                | ready
                | rejected
                | activated
                | suspended
                | aborted
                | completed
```

Von diesen Planzuständen beschreibt `inactive` den Zustand vor Starten des Plans. `considered`, `possible` und `ready` beschreiben in dieser Reihenfolge die zu durchlaufenden Zustände der `Selection`-Phase. `activated` und `suspended` sind die Zustände `Execution`-Phase und `rejected`, `aborted` und `completed` die Terminierungszustände.

6.14 Wartebedingungen (wait-for)

Die Wartebedingung beschreibt Mengen von Mengen von Unterplänen, auf die ein Plan warten muss, bevor er seine Ausführung erfolgreich beenden darf. Der Plan darf sich beenden, sobald mindestens aus einer der spezifizierten Mengen alle Pläne beendet wurden.

wait-for wf ::= **bwf**(**plan-name**)
| „**wait-for-none**“
| **wait-for-n**(**nat**; **plan-list**)
| **wait-for** _and **wait-for**
| **wait-for** _or **wait-for**
| **wait-for** _xor **wait-for**
| _not **wait-for**;

Ein Basiskonstrukt der Wartebedingungen ist das **bwf**. Dieses gilt dann als erfüllt, wenn der mittels des Plannamen spezifizierte Unterplan erfolgreich beendet wurde. Das andere Basiskonstrukt der Wartebedingungen ist das **wait-for-none**, welches immer erfüllt ist.

Am häufigsten wird das **wait-for-n** verwendet. Dieses erlaubt eine List von Plannamen zusammen mit einer Zahl von Plänen anzugeben. Ein solches **wait-for-n** ist erfüllt, wenn aus der angegebenen Menge von Plänen mindestens soviele erfolgreich beendet wurden, wie von der angegebenen Zahl gefordert.

Die logische Konjunktion mehrere **wait-for** mittels _and, _xor und _or entspricht der intuitiven Bedeutung der Junktoren. Im Fall des _and Junktors müssen beide beteiligten **wait-for** erfüllt sein, damit die Kombination erfüllt ist. Im Fall des _or Junktors reicht es, wenn eines der **wait-for** erfüllt ist und im Fall von _xor muss genau eines erfüllt sein.

Besondere Beachtung verdient die Negation von Wartebedingungen. Eine negierte Wartebedingung gibt an, dass die nicht negierte Wartebedingung gerade nicht erfüllt sein darf. Damit ist es möglich, Pläne zu spezifizieren, deren erfolgreiche Beendigung einen Fehlerzustand anzeigt und dies in die Wartebedingung einzubeziehen. Beispielsweise kann damit ein Plan spezifiziert werden, der erfolgreich beendet wird, wenn der Patient einen kritisch niedrigen Blutdruck hat. In einer negativen Wartebindung eingebunden, würde eine Beendigung dieses Plans den Abbruch des übergeordneten Plans auslösen.

Die Semantik des **wait-for** Konstruktes wird über Prädikate **satisfied** und **satisfiable** spezifiziert. Damit ist es möglich, frühzeitig zu festzustellen, wenn eine Wartebedingung nicht mehr erfüllt werden kann, weil beispielsweise zu viele notwendige Pläne zurückgewiesen wurden. In diesem Fall ist es möglich, sofort mit einem Planabbruch zu reagieren.

Ein **wait-for** Konstrukt ist erfüllt, wenn alle notwendigen Pläne erfolgreich beendet wurden und alle Pläne eines negierten **wait-for** nicht erfolgreich beendet wurden. Ein **wait-for** Konstrukt ist erfüllbar, wenn alle notwendigen Pläne noch laufen oder bereits erfolgreich beendet wurden und alle negierten Pläne noch laufen oder abgebrochen wurden.

Im Beispiel muss sichergestellt werden, dass der `Antibiotikum`-Plan nicht abgebrochen werden darf. Das heißt, dieser Plan muss so gestaltet sein, dass, wenn innerhalb einer gewissen Wartezeit kein Fieber festgestellt wird, der Plan automatisch beendet wird. Der `Erkältung`-Plan muss darauf warten, bis der Plan beendet, entweder weil kein Fieber vorliegt oder weil die Behandlung erfolgreich beendet wurde. Dazu kann in der Wartebedingung des `Erkältung`-Plans der `Antibiotikum`-Plan als notwendiger Plan spezifiziert werden.

7 Semantik des Zustands von Asbru Systemen

In Kapitel 6 wurde die Syntax von Asbru-Plänen vorgestellt. Diese beschreibt eine Art statischer Sicht auf Asbru-Pläne. Auch wird von der Syntax die Asbru-Planhierarchie beschrieben. Dieser Begriff beschreibt die hierarchische Ordnung, die sich für eine Menge zusammengehöriger Pläne aus der Plan-Unterplan Relation heraus ergibt. Ziel dieses Teils der Arbeit ist es, die Semantik von Asbru zu definieren. Diese wird durch Zustandsübergänge beschrieben. Um die Semantik festlegen zu können, sind mehrere Schritte notwendig. Zuerst wird in diesem Kapitel der Begriff des Zustands der Asbru-Hierarchie eingeführt und die Syntax und Semantik dieses Zustands beschrieben. Das darauffolgende Kapitel 8 beschreibt die Semantik der Auswertung von Asbru Bedingungen. Mit diesen Grundlagen ist es dann möglich, die Zustandsübergänge der Asbru-Pläne formal zu definieren, was in Kapitel 9 getan wird. Diese Definitionen zusammen erlauben die Ablaufsemantik von Asbru zu definieren. Dies geschieht in Kapitel 10.

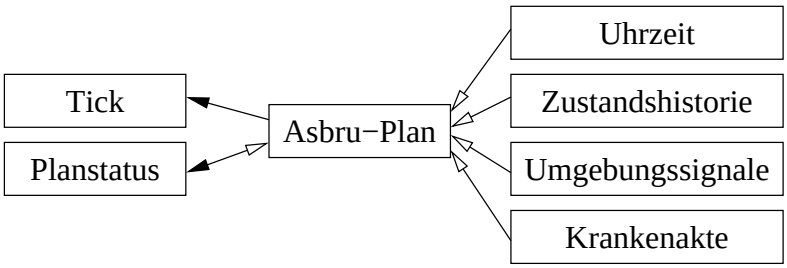


Abbildung 7.1: Kommunikation in Asbru

In diesem Kapitel wird der Zustand von Asbru beschrieben und die dazu notwendigen Datenstrukturen definiert. Bevor diese im Einzelnen geschieht, sollen die Datenstrukturen zunächst in Relation zu einem Plan und seiner Ausführung gesetzt werden. Abbildung 7.1 zeigt einen Plan und die Variablen des Zustands, auf die er zugreift. Es gibt sechs verschiedene Zustandskomponenten, die für einen Plan bei der Ausführung relevant sind. In der Grafik sind die Eingabevariablen des Plans auf der rechten Seite vom Plan aufgeführt, die Ein-/Ausgabevariablen auf der linken Seite. Die verschiedenen Komponenten werden zunächst kurz beschrieben, um einen Überblick zu geben, bevor sie im Anschluss detailliert erläutert werden.

Das Zeitmodell von Asbru besteht aus Mikro- und Makro-Schritten. Mikro-Schritte bezeichnen Aktionen der Asbru-Pläne, die so kurz dauern, dass während dieser Aktionen keine messbare Zeit vergeht. Details dieses Modells und der Uhren in Asbru werden in Abschnitt 7.1 beschrieben. Die **Tick** Komponente des Zustands dient einem Plan dazu, bekannt zu geben, ob die von ihm während eines Schritts durchgeführten Handlungen zu einem Mikro-Schritt führen sollen oder nicht. Im Fall, dass der Plan aktive Handlungen ausführt, gilt der Schritt als Mikro-Schritt, wodurch sich die Uhrzeit nicht ändert, andernfalls wächst die Uhrzeit. Die Uhrzeit des Gesamtsystems wird von den Plänen ausschließlich gelesen. Sie wird von Asbru-Plänen zur Auswertung von Bedingungen benötigt.

Die Umgebung – also das medizinische Personal – kann die Auswertung von Bedingungen beeinflussen, in dem sie Umgebungssignale an den Plan sendet. Diese Signale beeinflussen

zum Einen die Auswertung der Asbru-Bedingungen (Kapitel 6.7), zum Anderen aber auch die Reihenfolge, in denen die Pläne Aktivierungssignale senden. Diese Signale und die zugehörige Zustandskomponente werden in Abschnitt 7.2 beschrieben.

Ein Plan macht seine Ausführung unter anderem von seinem und dem Planstatus der übergeordneten Pläne sowie Unterpläne abhängig. Der Planstatus enthält Informationen über den Plan, beispielsweise in welchem der Planzustände (`inactive`, `considered`,...) sich der Plan befindet oder ob das Aktivierungssignal `E` vorliegt, dass ein Plan für seinen Übergang vom Zustand `ready` in den Zustand `activated` benötigt. Bestimmte Zustandsübergänge eines Plans sind dazu an Bedingungen an den Planzustand übergeordneter Pläne gekoppelt, beispielsweise, ob der übergeordnete Plan in einem terminierten Zustand ist. Die Details dieser Zustandskomponente werden in Abschnitt 7.3 beschrieben.

Asbru-Bedingungen, die die Auswertung von Plänen beeinflussen, können mit zeitlichen Bezügen versehen sein. Damit ist es möglich, dass eine Bedingung nicht nur vom aktuellen Zustand abhängt („Patient hat Fieber“) sondern auch von vergangenen Zuständen („Patient hatte fünf Stunden nach Einnahme des Medikaments Fieber“). Diese zeitlichen Bezüge müssen ausgewertet werden können („wann war die Einnahme des Medikaments“). Diese Auswertung wird übersetzt in Planzustandwechsel („Das Medikament wurde gegeben, als Plan `X` in den Zustand `activated` gewechselt hat“). Um diese Planzustandswechsel reale Uhrzeiten zuordnen zu können, müssen alle Planzustandswechsel stets protokolliert werden. Dies wird mittels der Zustandshistorie getan, die in Abschnitt 7.4 beschrieben wird.

Analog dazu, dass vergangene Planzustandswechsel gespeichert werden müssen, muss auch der Zustand des Patienten gespeichert werden. Das heißt, es muss eine Art elektronischer Krankenakte geführt werden, die nicht nur den gegenwärtigen sondern auch die Vergangenen Zustände des Patienten speichert. Diese wird in Abschnitt 7.5 vorgestellt.

Mit all diesen Komponenten zusammen wird der globale Zustand von Asbru in Abschnitt 7.6 definiert werden.

7.1 Zeitliche Aspekte in Asbru

Ein Asbru-System beschreibt ein Interaktionssystem. Die Pläne der Asbru-Hierarchie reagieren auf den gegenwärtigen Zustand des Patienten und senden Signale an das medizinische Personal, welche Interaktionen von diesem durchgeführt werden sollen. Diese Interaktionen und die Reaktionen darauf verlangen, dass in die Semantik von Asbru das Konzept der Unterscheidung zwischen dem System und einer Umgebung eingebaut wird. Dieses Zeitmodell soll im Folgenden beschrieben werden.

Das Asbru Zeitmodell unterscheidet zwischen Mikro- und Makro-Schritten. Mikro-Schritte finden dann statt, wenn die Asbru-Hierarchie interne Berechnungen und Zustandsübergänge durchführt. Ein Beispiel dafür ist das Starten eines Unterplans durch einen Plan. Während eines solchen Mikro-Schritts vergeht keine messbare Zeit. Diese Annahme entspricht der sogenannten „strong synchronous hypothesis“ [3]. Diese geht davon aus, dass ein Computersystem beliebig viel schneller agiert als die physische Umgebung. Mit dieser Annahme wird dem Asbru System erlaubt, beliebig viele interne Schritte zur Berechnung durchzuführen. Pläne können in ihrer Ausführung blockieren. Beispielsweise kann ein Plan im Zustand `considered` seine **filter**-Bedingung auswerten und dabei feststellen, dass der Wahrheitswert dieser Bedingung vom Plan verlangt, auf eine Zustandsänderung zu warten. In diesem Fall ist der Plan blockiert. Ein übergeordneter Plan kann dadurch ebenfalls blockiert werden, beispielsweise, wenn er darauf wartet, dass der bereits blockierte Plan seine Ausführung beendet. Allgemein ist ein Plan immer dann blockiert, wenn alle für ihn möglichen Transitionen, die seinen Zustand verändern, blockiert sind und er auf Eingaben anderer Pläne oder allgemein auf Zustandsänderungen wartet. Sind alle Pläne einer Asbru-Hierarchie blockiert, so ist diese stabil.

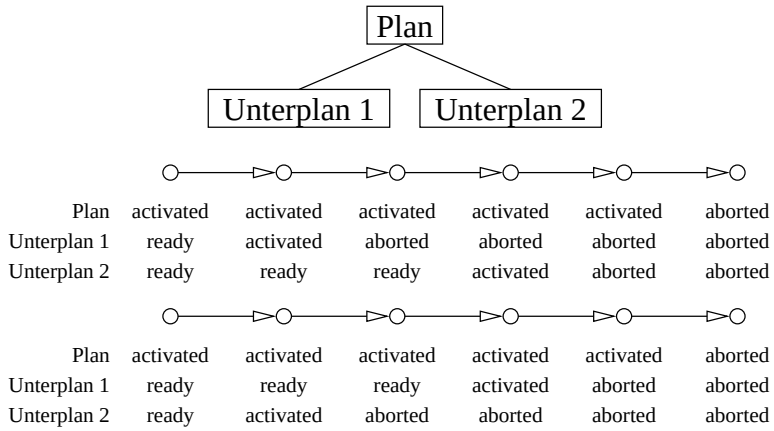


Abbildung 7.2: Ununterscheidbare Zustandsabfolgen

Erst wenn die Hierarchie stabil ist, ist es zulässig, dass die Umgebung einen Schritt ausführt. In diesem Fall findet ein Makro-Schritt statt. Während eines Makro-Schritts vergeht Zeit, der Patientenzustand sowie die Signale an die Planhierarchie können geändert werden. Jedoch ist es nicht zulässig, dass während eines Makro-Schritts der Zustand der Asbru-Pläne verändert wird.

Nach außen hin vergeht keine Zeit während eines Mikro-Schritts, also beispielsweise während des Planzustandswechsels eines Plans von `possible` nach `ready`. Vergehen zwischen zwei Makro-Schritten mehrere Mikro-Schritte, so kann ein externer Beobachter keine zeitliche Reihenfolge zwischen einzelne Planzustandsänderungen bestimmen. Dies soll anhand eines Beispiels erläutert werden, das in Abbildung 7.2 dargestellt ist.

Als Beispiel zur Verdeutlichung dient hier der `anyorder` kontrollierte Plan `Plan` mit seinen zwei Unterplänen. Dieser Plan aktiviert unter geeigneten Voraussetzungen nacheinander seine beiden Unterpläne jedoch in beliebiger Reihenfolge. Das heißt, dass beide in der Abbildung aufgeführten Abläufe aufgrund dieses Indeterminismus möglich sind. Alle in der Grafik aufgeführten Schritte sind Mikro-Schritte, was daran erkennbar ist, dass jeweils mindestens ein Plan der Hierarchie nicht blockiert ist. Dadurch dass alle Mikro-Schritte unendlich kurz sind, wird ein externer Beobachter am Ende beider Abläufe jeweils die Information bekommen, dass die beiden Unterpläne jeweils unendlich kurz in den Zuständen `ready`, `activated` und `aborted` waren und in letzterem Zustand auch noch sind. Dadurch werden die beiden Abläufe für den externen Beobachter ununterscheidbar. Dies ist gewollt. Da Mikro-Schritte nicht gezählt werden sollen, reicht zur Erfassung der Zeit ein Makro-Schrittzähler als Uhr aus. Dabei wird in jedem Makro-Schritt für jeden Plan in einer Liste protokolliert, welche Planzustände der Plan seit dem letzten Makro-Schritt durchlaufen hat. Diese Daten werden in der Datenstruktur der Asbru-Zustands-Historie gespeichert, die in Abschnitt 7.4 spezifiziert wird.

Definition

Die Uhrzeit in Asbru wird gespeichert durch eine ganze Zahl (**int**) mit der Standardsemantik ganzer Zahlen.

Uhrzeiten werden in Asbru häufig als relative Werte angegeben. Beispielsweise können Behandlungspläne den Zustand des Patienten vor Behandlungsbeginn abfragen, soweit die-

ser bekannt ist. Damit ist es nicht möglich, eine natürliche Zahl als Uhrzeit zu verwenden, weil es damit nicht möglich ist, beliebig weit vor den Planstart in der Vergangenheit Werte auszulesen.

7.2 Umgebungssignale

Das medizinische Personal kommuniziert mit dem Asbru System mit Hilfe von Signalen, die dem System gesendet werden. Dabei hat das medizinische Personal zwei grundsätzlich unterschiedliche Möglichkeiten, das Verhalten des Systems zu beeinflussen. Zum Einen kann die Auswertung der sechs Asbru-Bedingungen (also **filter**-Bedingung, **setup**-Bedingung usw.) jedes Plans beeinflusst werden. Zum Anderen können `anyorder`-Pläne in ihrer Ausführung durch Signale beeinflusst werden. Dieser Plankontrolltyp wurde in Abschnitt 6.3 vorgestellt. Er steuert seine Unterpläne prinzipiell zufällig an. Das medizinische Personal hat eine Möglichkeit, mittels Signalen diesen Indeterminismus zu übersteuern. Dadurch aktiviert ein `anyorder` kontrollierter Plan nicht einen beliebigen Unterplan sondern genau den vom medizinischen Personal angegebenen.

Im Asbru Zustand werden alle Signale, die die Umgebung dem System senden kann, in einer Variablen gebündelt. Diese Variable ist eine prädikatenlogisch spezifizierte dynamische Funktion, die Planinstanznamen abbildet auf Sammlungen dieser Signale. Diese Signal-Sammlungen werden im Folgenden spezifiziert werden.

Jede der sechs Bedingungen eines Asbru Plans kann auf zwei verschiedene Arten vom medizinischen Personal in ihrer Auswertung beeinflusst werden. Zum Einen kann das Personal verlangen, dass eine Bedingung übersteuert wird, was bedeutet, dass diese Bedingung ausgewertet wird, als wäre sie erfüllt – unabhängig vom tatsächlichen Wahrheitswert. Die zweite Möglichkeit ist, dass die Auswertung der Bedingung verzögert wird, dass also eine eigentlich erfüllte Bedingung so behandelt wird, als wäre sie zwar erfüllbar, jedoch noch nicht erfüllt. Ein Beispiel für die Notwendigkeit solcher Signale ist die Behandlung von Kindern. Es gibt Fälle, in denen keine Medikamente existieren, für die Studien die uneingeschränkte Unbedenklichkeit Kindern gegenüber bestätigen, da Studien an Kindern im Allgemeinen als ethisch sehr kritisch angesehen werden [2]. In solchen Fällen muss der Arzt eine Risikoabwägung durchführen, um zu entscheiden, ob in einem konkreten Fall ein Medikament gegeben werden soll. Dies kann in einer Asbru-formalisierten Leitlinie so formalisiert sein, dass der Arzt eine Bedingung zum Starten des Plans aktiv übersteuern muss, damit der entsprechende Plan aktiviert wird.

Booltupel (bool-tupel)

Insgesamt ergeben sich durch die sechs Bedingungen und je zwei Interaktionsmöglichkeiten pro Bedingung zwölf Signale, die jedem Asbru-Plan gesendet werden können. Hinzu kommt noch ein zusätzliches Signal, das angibt ob eine Präferenz für diesen Plan in Bezug auf die `anyorder` Kontrolle getroffen wurde. Diese 13 Signale werden in zwei Gruppen von jeweils sechs Signalen plus einem zusätzlichen Signal angeordnet. Die zwölf Signale zur Kontrolle der Bedingungen werden zur Speicherung im Zustand in zwei Gruppen zu je sechs Signalen angeordnet. Jeweils eine Gruppe enthält die Signale für die Verzögerung bzw. die Übersteuerung aller Bedingungen. Eine solche Signalgruppe von sechs Signalen wird in der Datenstruktur Booltupel gespeichert.

Definition

Ein **bool-tupel** ist ein sechs-Tupel, **bool-tupel** = (**bool** × **bool** × **bool** × **bool** × **bool** × **bool**).

Für den Zugriff auf die **bool**-Werte eines **bool-tupel** sind die sechs nachstehenden Zugriffsfunktionen definiert:

..filter	:	**bool-tupel** $\mapsto$ **bool**
..setup	:	**bool-tupel** $\mapsto$ **bool**
..suspend	:	**bool-tupel** $\mapsto$ **bool**
..reactivate	:	**bool-tupel** $\mapsto$ **bool**
..abort	:	**bool-tupel** $\mapsto$ **bool**
..complete	:	**bool-tupel** $\mapsto$ **bool**

Sei **BT** ein **bool-tupel** mit $\mathbf{BT} = (\mathbf{bv}_1, \mathbf{bv}_2, \mathbf{bv}_3, \mathbf{bv}_4, \mathbf{bv}_5, \mathbf{bv}_6)$. Dann ist die Semantik der Selektoren der **bool-tupel** definiert durch die nachstehenden Gleichungen:

BT.filter	=	$\mathbf{bv}_1$
BT.setup	=	$\mathbf{bv}_2$
BT.suspend	=	$\mathbf{bv}_3$
BT.reactivate	=	$\mathbf{bv}_4$
BT.abort	=	$\mathbf{bv}_5$
BT.complete	=	$\mathbf{bv}_6$

Jedes der sechs Signale wird einer Bedingung des Asbru-Plans zugeordnet. Daher werden die Selektoren der Signale bereits nach der jeweiligen Bedingung benannt, auf die sie später Einfluss nehmen. Beispielsweise wird das Signal zum Übersteuern bzw. Verzögern der **filter**-Bedingung aus einem Booltupel mit dem Selektor **.filter** selektiert.

Für die restlichen Signale gilt dies analog.

Booltupel-Paare (environment-aggregation-entry)

Nachdem jeweils sechs Signale nun zu einem **bool-tupel** gebündelt werden können, müssen als nächstes zwei **bool-tupel**s zu einem Paar kombiniert werden. Dies ist notwendig, damit alle zwölf Signale an einem Punkt zentral gespeichert werden können. Dazu wird eine Datenstruktur namens **environment-aggregation-entry** definiert. Diese soll nicht nur zwei **bool-tupel** Datenstrukturen speichern, sondern zusätzlich noch das **bool**, in dem spezifiziert wird, ob ein Plan präferiert werden soll, wenn er Unterplan eines `anyorder` kontrollierten Plans ist.

Definition

Ein **environment-aggregation-entry** ist ein Tripel, **environment-aggregation-entry** = (**bool-tupel** $\times$ **bool-tupel** $\times$ **bool**).

Zum Zugriff auf die Inhalte eines **environment-aggregation-entry** sind die nachfolgenden Selektoren definiert:

..override	:	**environment-aggregation-entry** $\mapsto$ **bool-tupel**
..confirm	:	**environment-aggregation-entry** $\mapsto$ **bool-tupel**
..prefer	:	**environment-aggregation-entry** $\mapsto$ **bool**

Sei **EAE** ein **environment-aggregation-entry** mit $\mathbf{EAE} = (\mathbf{BT}_1, \mathbf{BT}_2, \mathbf{bv})$. Dann ist die Semantik dieser Funktionen definiert durch:

$$\begin{array}{lcl} \textbf{EAE.override} & = & \textbf{BT}_1 \\ \textbf{EAE.confirm} & = & \textbf{BT}_2 \\ \textbf{EAE.prefer} & = & \textbf{bv} \end{array}$$

Um auf die Komponenten der **environment-aggregation-entry** zugreifen zu können, sind Zugriffsfunktionen definiert. Der Selektor, der das erste **bool-tupel** selektiert, wird **.override** genannt. Die in dem selektierten **bool-tupel** gespeicherten Signale bestimmen, ob die Auswertung der zugehörigen Bedingung übersteuert werden kann. Der Selektor **.confirm** selektiert die Signale, die für eine Verzögerung der Auswertung zuständig sind. Der letzte Selektor, **.prefer**, liefert das Signal zurück, dass über eine Bevorzugung des Plans bei `anyorder` Ausführung entscheidet.

Umgebungssignal-Sammlung (environment-aggregation)

Da für jeden Plan ein eigener Satz von 13 Umgebungssignalen benötigt wird, muss eine Möglichkeit gefunden werden, die Signale aller Pläne zu bündeln. Andernfalls wäre die Syntax des Zustands, der all diese Signale speichern muss, abhängig von der Zahl der laufenden Pläne. Eine **environment-aggregation** dient dazu, die Umgebungssignale aller Pläne in einer Variablen zu bündeln. Technisch gesehen wird zur Speicherung eine dynamische Funktion verwendet. Diese soll im Folgenden algebraisch spezifiziert werden. Dazu wird zunächst die Syntax der **environment-aggregation** festgelegt als eine Funktion, die von Planinstanznamen auf **environment-aggregation-entry** abbildet:

Definition

Die Umgebungssignal-Sammlung ist eine algebraisch spezifizierte Funktion, die Planinstanznamen auf Umgebungssignalsammlungen abbildet.

$$\textbf{environment-aggregation} \ : \ \textbf{instance-name} \mapsto \textbf{environment-aggregation-entry}$$

Zur Aktualisierung und zur Wertebestimmung der Funktion werden nachstehende Funktionen definiert:

$$\begin{array}{lcl} \textbf{.[.]} & : & \textbf{environment-aggregation} \times \textbf{instance-name} \\ & & \times \textbf{environment-aggregation-entry} \mapsto \textbf{environment-aggregation} \\ \textbf{.[.]} & : & \textbf{environment-aggregation} \times \textbf{instance-name} \\ & & \mapsto \textbf{environment-aggregation-entry} \end{array}$$

Die Semantik dieser Funktionen wird wie folgt definiert.
Sei **EAGG** ein **environment-aggregation**, weiterhin seien $\textbf{iname}_1$ und $\textbf{iname}_2$ vom Typ **instance-name** sowie **EAE** vom Typ **environment-aggregation-entry**. Dann gilt:

$$\begin{array}{lcl} \textbf{EAGG}[\textbf{iname}_1, \textbf{EAE}][\textbf{iname}_1] & = & \textbf{EAE} \\ \textbf{EAGG}[\textbf{iname}_1, \textbf{EAE}][\textbf{iname}_2] & = & \textbf{EAGG}[\textbf{iname}_2], \text{wenn } \textbf{iname}_2 \neq \textbf{iname}_1 \end{array}$$

Die Funktionswerte der **environment-aggregation** Funktion können mittels expliziter Updates spezifiziert werden. Dies ist beispielsweise notwendig, um für einen Asbru-Plan festzulegen, welche Werte die Signale an ihn gegenwärtig haben. Um dies zu ermöglichen werden

zwei Funktionen spezifiziert, eine, die es erlaubt, eine **environment-aggregation** zu aktualisieren und eine, die es erlaubt aktuelle Werte aus einer **environment-aggregation** auszulesen.

Diese Funktionen werden in sogenannter Mixfix-Notation angegeben. Diese drückt aus, dass die zu aktualisierende Funktion als erstes aufgeführt wird, danach werden in Klammern die Argumente übergeben. Die Semantik der angegebenen Funktionen ist wie folgt. Wird in einer **environment-aggregation** Funktion der Eintrag zum Instanznamen **instance-name** gesucht, so werden die Aktualisierungen der Funktion sequentiell durchsucht, bis eine entsprechende Aktualisierung gefunden werden kann.

Damit ist die Definition der Datenstruktur abgeschlossen, mit der die Signale aller Pläne im Zustand von Asbru abgespeichert werden können.

7.3 Hierarchiezustand

Die Umgebungssignale bieten eine Möglichkeit, zentralisiert auf alle Signale zuzugreifen, die von der Umgebung an die Planhierarchie gesendet werden. Analog dazu wird in diesem Abschnitt eine Möglichkeit angegeben, mit der es möglich ist, den internen Zustand der Planhierarchie zu verwalten. Der innere Zustand aller Asbru-Pläne wird im Folgenden auch Hierarchiezustand genannt. Dieser beschreibt drei Aspekte jedes einzelnen Plans. Zum Einen wird angegeben, welchen Planzustand der Plan gegenwärtig innehat, also, dass er beispielsweise im Zustand `possible` ist. Als zweites wird in diesem Hierarchiezustand gespeichert, ob der Plan bereits von seinem übergeordneten Plan das Aktivierungssignal `E` erhalten hat, welches er benötigt, um vom Zustand `ready` in den Zustand `activated` wechseln zu können (vgl. Abbildung 3.5 in Abschnitt 3.2). Die dritte zu speichernde Komponente betrifft den zyklischen Plantyp. Wie in Abschnitt 6.3.3 beschrieben führt ein zyklischer Plan seinen Unterplan wiederholt aus, bis eine ausreichende Anzahl von Wiederholungen erreicht wurde. Die Zahl der bereits absolvierten Wiederholungen ist auch im Hierarchiezustand gespeichert.

Zunächst soll nun ein Datentyp spezifiziert werden, der diese drei Werte aufnimmt um danach eine dynamische Funktion anzugeben, die einen Planinstanznamen auf einen solchen Tripel-Datentyp abbildet.

7.3.1 Planstatus (plan-status)

Der Planstatus **plan-status** ist ein Datentyp mit dem der Zustand eines Plans gespeichert wird. Im Planstatus wird gespeichert, welchen Planzustand der Plan gegenwärtig hat, also ob er beispielsweise im Zustand `possible` ist. Weiterhin wird gespeichert, ob sein Aktivierungssignal `E` gesendet wurde. Im Falle zyklischer Pläne wird außerdem gespeichert, wieviele Wiederholungen der Ausführung des Unterplans bereits abgeschlossen wurden.

Definition

Ein Planstatus **plan-status** ist ein Tripel, **plan-status** = (**plan-state** × **bool** × **nat**). Um auf die Komponenten eines Planzustand zugreifen zu können, werden die nachstehenden Selektoren definiert:

..state	:	**plan-status** ↦ **plan-state**
..activate	:	**plan-status** ↦ **bool**
..rounds	:	**plan-status** ↦ **nat**

Sei θ ein **plan-status**, **PS** vom Typ **plan-state**, **bv** vom Typ **bool** und **n** vom Typ **nat**. Weiter soll gelten θ = (**PS**, **bv**, **n**). Dann ist die Semantik der Selektoren wie folgt definiert:

θ**.state** = **PS**
θ**.activate** = **bv**
θ**.rounds** = **n**

Um auf die einzelnen Komponenten des Planstatus zugreifen zu können sind Selektoren definiert. Der **.state** Selektor selektiert den aktuellen Planzustand, der **.activate** Selektor das Aktivierungssignal E und die Anwendung des **.rounds** Selektors auf einen **plan-status** ergibt die Zahl der bereits erfolgreich abgeschlossenen Wiederholungen des zyklischen Plans.

Zur vereinfachten Handhabung des Planstatus werden Aktualisierungsfunktionen definiert. Diese sind in Mixfix Notation angegeben und erlauben jeweils eine der drei Komponenten eines Planzustands zu aktualisieren.

Definition

Die Aktualisierungsfunktionen des Planzustands sind wie folgt definiert:

. [.] : **plan-status** × **plan-state** ↦ **plan-status**
. [.] : **plan-status** × **nat** ↦ **plan-status**
. [.] : **plan-status** × **bool** ↦ **plan-status**

Sei θ ein **plan-status**, sowie **PS** und $\mathbf{PS}_1$ Planzustände. Außerdem sollen **bv** und $\mathbf{bv}_1$ vom Typ **bool** sein und **n** sowie $\mathbf{n}_1$ vom Typ **nat**. Mit $\theta = (\mathbf{PS}, \mathbf{bv}, \mathbf{n})$ ist die Semantik der Aktualisierungsfunktionen wie folgt definiert:

$\theta[\mathbf{PS}_1] = (\mathbf{PS}_1, \mathbf{bv}, \mathbf{n})$
$\theta[\mathbf{n}_1] = (\mathbf{PS}, \mathbf{bv}, \mathbf{n}_1)$
$\theta[\mathbf{bv}_1] = (\mathbf{PS}, \mathbf{bv}_1, \mathbf{n})$

Die erste angegebene Aktualisierungsfunktion ersetzt die Planzustands-Komponente des Planstatus, die zweite Funktion ersetzt den zyklischen Rundenzähler und die dritte Komponente aktualisiert das Aktivierungssignal im Planzustand.

7.3.2 Zustand der Asbru-Hierarchie (hierarchy-state)

Ein Planstatus **plan-status** speichert den Status eines einzelnen Asbru-Plans. Asbru-Plan-Hierarchien können beliebig groß werden. Damit der Zustand von Asbru unabhängig von der Größe der Plan-Hierarchien spezifiziert werden kann, muss ein Konstrukt verwendet werden, um die verschiedenen Planstatus Variablen zusammenzufassen und zentral zugreifbar zu machen. Wie auch im Fall der Umgebungssignale soll dazu eine algebraisch spezifizierte dynamische Funktion verwendet werden.

Analog zur Sammlung der Umgebungssignale in einer **environment-aggregation** sind Funktionen definiert, die eine Aktualisierung von Hierarchiezustandsvariablen erlauben sowie Daten aus diesen selektieren.

Definition

Ein Hierarchiezustand **hierarchy-state** ist eine dynamische Funktion, die von Instanznamen nach Planstatus abbildet:

$$\textbf{hierarchy-state} \;:\; \textbf{instance-name} \mapsto \textbf{plan-status}$$

Für einen Hierarchiezustand sind folgende Aktualisierungs- und Selektionsfunktionen definiert:

$$\begin{array}{lcl} \textbf{.}\,[\,\textbf{.}\,] & : & \textbf{hierarchy-state} \times \textbf{instance-name} \times \textbf{plan-status} \mapsto \textbf{hierarchy-state} \\ \textbf{.}\,[\,\textbf{.}\,] & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{plan-status} \end{array}$$

Deren Semantik ist wie folgt spezifiziert: Sei **HS** ein **hierarchy-state** und $\textbf{iname}_1$ sowie $\textbf{iname}_2$ Planinstanznamen **instance-name**, sowie θ ein **plan-status**. Dann gilt:

$$\begin{array}{lcl} \textbf{HS}[\textbf{iname}_1, \theta][\textbf{iname}_1] & = & \theta \\ \textbf{HS}[\textbf{iname}_1, \theta][\textbf{iname}_2] & = & \textbf{HS}[\textbf{iname}_2], \text{wenn } \textbf{iname}_2 \neq \textbf{iname}_1 \end{array}$$

Es gibt eine Aktualisierungsfunktion, die bei gegebenem Hierarchiezustand einen Planstatus für einen Instanznamen einfügt. Weiterhin gibt es eine Zugriffsfunktion, die den Wert des Planstatus einer bestimmten Planinstanz selektiert.

Wird eine Aktualisierung eines Hierarchiezustands für eine Planinstanz durchgeführt, so ergibt die Anwendung der Selektorfunktion mit diesem Planinstanznamen den ursprünglich eingegebenen Wert. Aktualisierungen anderer Planinstanzen werden vom Selektor verworfen.

7.4 Asbru-Zustands-Historie

Der Hierarchie-Zustand speichert für jeden Plan den aktuellen Planzustand. Dies erlaubt dem Plan, lokal Entscheidungen zu treffen. Beispielsweise kann ein Plan feststellen, dass er, falls er gegenwärtig im Zustand `ready` ist, auf das Aktivierungssignal seines übergeordneten Plans warten muss und sonst keine Handlungen durchführen kann, da dieses noch nicht gesendet wurde.

Allerdings ist es mit dem Hierarchie-Zustand nicht möglich, in die Vergangenheit zu sehen. Beispielsweise bedeutet dies, dass damit die Auswertung abstrakter Uhren unmöglich ist. Um diese auswerten zu können, muss der Zeitpunkt bestimmt werden, an dem ein bestimmter Planzustandswechsel eines Plans stattgefunden hat. Dazu müssen die vergangenen Planzustandsübergänge dieses Plans bekannt sein. Diese werden in der Asbru-Zustands-Historie gespeichert.

Um diese Informationen zu speichern, werden zunächst die Planzustandslisten definiert . Diese können Folgen von Planzuständen speichern. Mit den Einträgen in die Asbru-Zustands-Historie (**asbru-state-history-entry**) wird danach ein Datentyp definiert, mit dem die Planzustandslisten zu Uhrzeiten zugeordnet werden können. Zuletzt wird dann der Datentyp der Asbru-Zustands-Historie definiert, der die Zuordnung von Instanznamen zu Einträgen in die Asbru-Zustands-Historie übernimmt.

7.4.1 Planzustandslisten (plan-state-list)

Planzustandslisten enthalten eine Liste von Planzuständen, also Einträge vom Typ **plan-state**. Wie in Abschnitt 7.1 beschrieben, kennt Asbru ein Zeitkonzept von Mikro- und Makro-Schritten. Dabei wird jeweils während der Makro-Schritte ein Protokoll mitgeschrieben, in dem festgehalten wird, welche Zustände die einzelnen Pläne in den

Mikro-Schritten seit dem letzten Makro-Schritt durchlaufen haben. Um solche Protokolle speichern zu können, werden Planzustandslisten verwendet. Diese Datenstrukturen werden als algebraische Listen definiert:

Definition

Eine Planzustandsliste **plan-state-list** ist eine Liste von Planzuständen **plan-state**.

plan-state-list = []
| . ⇒ .(. .previous : **plan-state-list**; . .last : **plan-state**)

In dieser Datenstruktur sollen alle Planzustände gespeichert werden, die zwischen zwei Makro-Schritten durchlaufen werden. Anstelle des für Listen typischen Append-Operators verwenden Planzustandslisten den ⇒ Operator.

Eine mögliche Planzustandsliste, die beschreibt, dass ein Plan zwischen zwei Makro-Schritten zunächst gestartet wurde und dann die komplette `Selection`-Phase bis zum Zustand `ready` durchlaufen hat, sieht wie folgt aus:

(`inactive` ⇒ `considered` ⇒ `possible` ⇒ `ready`)

Für Planzustandslisten gibt es ein Prädikat **dff**. Dieses beschreibt, dass Planzustandslisten duplikatfolgenfrei sind, was bedeutet, dass keine zwei gleichen Planzustände direkt aufeinander folgen. Eine Planzustandsliste der Form

(... ⇒ `considered` ⇒ `considered` ⇒ ...)

wäre also nicht duplikatfolgenfrei. Die Protokolle, die von den Planzustandsübergängen angefertigt werden, sollen die Eigenschaft der Duplikatfolgenfreiheit erfüllen.

Definition

dff ist ein Prädikat, das als Argument eine Planzustandsliste erhält:

dff : **plan-state-list** ↦ **bool**

Sei **PSX** eine **plan-state-list**, und seien **PS** und $\mathbf{PS}_1$ jeweils vom Typ **plan-state**, dann formalisieren die nachstehenden Gleichungen die Semantik von **dff**.

dff([])
dff([] ⇒ **PS**)
dff(**PSX** ⇒ $\mathbf{PS}_1$ ⇒ **PS**)↔**PS** ≠ $\mathbf{PS}_1$ ∧ **dff**(**PSX** ⇒ $\mathbf{PS}_1$)

Mit dieser Definition der Duplikatfolgenfreiheit kann nun ein neuer Anfüge-Operator definiert werden, der bei Einfügeoperationen die Duplikatfolgenfreiheit erhalten soll:

. + . : **plan-state-list** × **plan-state** ↦ **plan-state-list**

Die Semantik des + Operators wird mit Hilfe der Planzustände $\mathbf{PS}_1$ und $\mathbf{PS}_2$ sowie der Planzustandsliste **PSX** formal spezifiziert:

$$
\begin{array}{lcl}
[] + \mathbf{PS}_1 & = & [] \Rightarrow \mathbf{PS}_1 \\
(\mathbf{PSX} \Rightarrow \mathbf{PS}_1) + \mathbf{PS}_1 & = & (\mathbf{PSX} \Rightarrow \mathbf{PS}_1) \\
(\mathbf{PSX} \Rightarrow \mathbf{PS}_1) + \mathbf{PS}_2 & = & \mathbf{PSX} \Rightarrow \mathbf{PS}_1 \Rightarrow \mathbf{PS}_2, \text{wenn } \mathbf{PS}_1 \neq \mathbf{PS}_2
\end{array}
$$

In der Semantik gilt, dass leere und ein-elementige Planzustandslisten stets duplikatfolgenfrei sind. Bei mehrelementigen Listen gilt die duplikatfolgenfreiheit genau dann, wenn sich die jeweils letzten beiden eingefügten Planzustände unterscheiden und die Listenreste ebenfalls duplikatfolgenfrei sind.

Der + Operator soll ein in eine Planzustandsliste einzufügende Element nur dann an die Liste anhängen, wenn das letzte Element nicht mit dem einzufügenden identisch ist. Andernfalls wird das Element verworfen. Ein Element wird stets in eine leere Liste eingefügt. In eine nicht-leere Liste wird ein Element genau dann eingefügt, wenn die Liste nicht auf ein identisches Element endet.

7.4.2 Einträge der Asbru-Zustands-Historie (asbru-state-history-entry)

Die Asbru-Zustands-Historie soll für jede Planinstanz und jeden Makro-Schritt speichern, welche Planzustände seit dem letzten Makro-Schritt durchlaufen wurden. Dies erfordert eine zweidimensionale Speichermöglichkeit, also beispielsweise eine Funktion, die zwei Parameter (nämlich Planinstanznamen und Uhrzeit) auf einen Wert (in dem Fall eine Planzustandsliste) abbildet, oder aber zwei ineinander verschachtelte Funktionen. In Asbru wird das Problem über verschachtelte Funktionen gelöst, so dass die Asbru-Zustands-Historie Planinstanznamen abbildet auf Funktionen, die von Zeitpunkten auf Planzustandslisten abbilden. Diese Funktionen werden als Einträge in die Asbru-Zustands-Historie bezeichnet und in diesem Abschnitt beschrieben.

Definition

Einträge in die Asbru-Zustands-Historie sind definiert als Funktionen, die eine Uhrzeit, also einen **int**-Wert, auf eine Planzustandsliste abbilden:

asbru-state-history-entry : **int** $\mapsto$ **plan-state-list**

Für **asbru-state-history-entry** werden Aktualisierungs- und Selektionsfunktionen wie folgt definiert:

$$
\begin{array}{lcl}
\mathbf{.[.]} & : & \mathbf{asbru\text{-}state\text{-}history\text{-}entry} \times \mathbf{int} \times \mathbf{plan\text{-}state\text{-}list} \\
 & & \mapsto \mathbf{asbru\text{-}state\text{-}history\text{-}entry} \\
\mathbf{.[.]} & : & \mathbf{asbru\text{-}state\text{-}history\text{-}entry} \times \mathbf{int} \mapsto \mathbf{plan\text{-}state\text{-}list} \\
\mathbf{.[.]} & : & \mathbf{asbru\text{-}state\text{-}history\text{-}entry} \times \mathbf{int} \times \mathbf{plan\text{-}state} \\
 & & \mapsto \mathbf{asbru\text{-}state\text{-}history\text{-}entry}
\end{array}
$$

Seien $\mathbf{AC}$ und $\mathbf{AC}_1$ vom Typ **int**, $\mathbf{PSX}$ vom Typ **plan-state-list**, $\mathbf{PS}$ vom Typ **plan-state** und **ASHE** ein Eintrag in eine Asbru-Zustands-Historie. Dann ist damit die Semantik dieser Funktionen wie folgt definiert:

$$
\begin{array}{lcl}
\mathbf{ASHE}[\mathbf{AC}, \mathbf{PSX}][\mathbf{AC}] & = & \mathbf{PSX} \\
\mathbf{ASHE}[\mathbf{AC}, \mathbf{PSX}][\mathbf{AC}_1] & = & \mathbf{ASHE}[\mathbf{AC}_1], \text{wenn } \mathbf{AC} \neq \mathbf{AC}_1 \\
\mathbf{ASHE}[\mathbf{AC}, \mathbf{PS}] & = & \mathbf{ASHE}[\mathbf{AC}, \mathbf{ASHE}[\mathbf{AC}] + \mathbf{PS}]
\end{array}
$$

Ein Eintrag in eine Asbru-Zustands-Historie **asbru-state-history-entry** ist eine algebraisch spezifizierte, dynamische Funktion, die Zeitpunkte auf Planzustandslisten abbildet. Die Spezifikation entspricht damit im Wesentlichen der der Definitionen von Umgebungssignalsammlung und Hierarchie-Zustand, nur dass in diesem Fall Zeitpunkte auf Planzustandslisten abgebildet werden, während beispielsweise beim Hierarchie-Zustand Planinstanznamen auf Planstatus Variablen abgebildet werden.

Es sind für diese dynamische Funktion Aktualisierungsfunktionen sowie Selektoren definiert. Die Semantik der ersten Aktualisierungsfunktion sowie des Selektors wird wechselseitig definiert. Soll aus einem Eintrag in die Asbru-Zustands-Historie der Wert zu einer bestimmten Uhrzeit selektiert werden, so wird dazu die letzte Aktualisierung des Eintrags in die Asbru-Zustands-Historie überprüft. Ist der Zeitpunkt dieser Aktualisierung gleich dem Zeitpunkt der Selektion, so ist das Ergebnis der Planzustandslisten-Wert der Aktualisierung. Andernfalls wird diese Aktualisierung verworfen und der Selektor wird rekursiv auf den Rest-Eintrag in die Asbru-Zustands-Historie angewendet.

Die zweite hier spezifizierte Aktualisierungsfunktion nimmt eine Sonderrolle ein. Diese Aktualisierung soll nicht eine Planzustandsliste durch eine andere ersetzen, sondern vielmehr um einen neuen Planzustand ergänzen. Dabei wird das anzufügende Element mit dem Additionsoperator + der Planzustandslisten angefügt, der die Duplikatfolgenfreiheit erhält.

Bei Anwendung dieses Operators wird eine Ergänzung der Planzustandsliste vorgenommen. Das heißt, die zu der angegebenen Zeit gehörige Planzustandsliste wird aus dem Eintrag der Asbru-Zustands-Historie zunächst selektiert, dann mittels des „+“ Operators verändert und dann wieder in den Eintrag zurückgeschrieben. Die Ersetzung von Planzustandslisten erfolgt dabei wie erwartet. Der Term $\mathbf{ASHE}[\mathbf{AC}, \mathbf{ASHE}[\mathbf{AC}] + \mathbf{PS}]$ ist dabei so zu lesen, dass zunächst der alte Wert des Eintrags selektiert wird ($\mathbf{ASHE}[\mathbf{AC}]$). Dieser Eintrag wird dann erweitert um den einzutragenden Planzustand ($\mathbf{ASHE}[\mathbf{AC}]+\mathbf{PS}$). Das Ergebnis ersetzt dann die ursprüngliche Planzustandsliste.

7.4.3 Asbru-Zustands-Historie (asbru-state-history)

Die Einträge in die Asbru-Zustands-Historie speichern für einen Plan den gesamten Verlauf des Planzustands. Wie schon bei Umgebungssignalsammlung und Hierarchiezustand muss nun eine Bündelung all dieser Einträge erfolgen, so dass der Zustand der Asbru-Hierarchie unabhängig von der Zahl der laufenden Asbru-Pläne bleiben kann. Dazu wird die Datenstruktur der Asbru-Zustands-Historie definiert.

Definition

Eine Planzustandshistorie **asbru-state-history** ist algebraisch definierte dynamische Funktion, die Planinstanznamen auf Einträge in die Asbru-Zustands-Historie abbildet:

asbru-state-history : **instance-name** $\mapsto$ **asbru-state-history-entry**;

Im Folgenden werden Funktionen definiert, um die Asbru-Zustands-Historie zu aktualisieren bzw. um Einträge auslesen zu können:

. [**.**] : **asbru-state-history** $\times$ **instance-name** $\times$ **asbru-state-history-entry**
$\mapsto$ **asbru-state-history**
. [**.**] : **asbru-state-history** $\times$ **instance-name**
$\mapsto$ **asbru-state-history-entry**

Sei **ASH** vom Typ **asbru-state-history**, **ASHE** vom Typ **asbru-state-history-entry**

uns **iname**$_1$ sowie **iname**$_2$ vom Typ **instance-name**. Dann wird die Semantik der Funktionen wie folgt spezifiziert:

$$\begin{aligned} \mathbf{ASH}[\mathbf{iname}_1, \mathbf{ASHE}][\mathbf{iname}_1] &= \mathbf{ASHE} \\ \mathbf{ASH}[\mathbf{iname}_1, \mathbf{ASHE}][\mathbf{iname}_2] &= \mathbf{ASH}[\mathbf{iname}_2], \text{wenn } \mathbf{iname}_2 \neq \mathbf{iname}_1 \end{aligned}$$

Die Definition der Semantik dieser Funktionen kann nur wechselseitig angegeben werden. Das heißt, der Selektor wird angewendet auf eine Asbru-Zustands-Historie, die mittels der Aktualisierungsfunktion aktualisiert wurde. Wurde die Aktualisierung für den gleichen Planinstanznamen durchgeführt, der vom Selektor selektiert wird, so ist der Rückgabewert der **ASHE** Wert der Aktualisierungsfunktion. Andernfalls wird der Selektor rekursiv auf die nicht-aktualisierte **ASH** angewendet.

Mit der **asbru-state-history** ist es möglich den kompletten Verlauf der Planzustände eines Asbru-Plans abzufragen. Auch möglich ist es, zu eruieren, zwischen welchen Makro-Schritten ein Planzustandswechsel stattgefunden hat bzw. welche Planzustände ein Plan durchlaufen hat.

7.5 Krankenakte

Die Krankenakte speichert den Verlauf des Zustands des Patienten ab. Da die in der Krankenakte zu speichernden Werte von der Fallstudie abhängen, wird zunächst ein allgemeiner Datentyp **data-value** spezifiziert, der später mit den korrekten Einträgen aktualisiert werden muss. Dann wird eine dynamische Funktion **patient-data** spezifiziert, die Zeichenketten auf **data-value** Einträge abbildet und zuletzt die Krankenakte, die Uhrzeiten auf **patient-data** abbildet.

7.5.1 Krankenakten-Datentyp (data-value)

Asbru erlaubt beliebige Datentypen als Einträge in die Krankenakte. Beispielsweise können Blutwerte in Gleitkommazahlen, das Alter eines Patienten als natürliche Zahl oder genetische Prädispositionen als Wahrheitswerte gespeichert werden. Darüberhinaus erlaubt Asbru die Definition symbolischer Werte. Beispielsweise kann die Körpertemperatur in symbolischen Werten, wie etwa „normal“, „leichtes Fieber“ und „hohes Fieber“ angegeben werden.

Speziell die symbolischen Werte sind abhängig von der betrachteten Fallstudie. Damit ist es nicht möglich, unabhägig von der Fallstudie alle möglichen Wertebereiche der Krankenakteneinträge anzugeben. Daher ist der Krankenakten-Datentyp, der diese Wertebereiche angibt, als unspezifizierter, allgemeiner Datentyp definiert. Um Fallstudien in Asbru zu spezifizieren, muss dieser Datentyp zunächst mit den in der Fallstudie verwendeten Datentypen aktualisiert werden.

Definition

Es gibt einen Krankenakten-Datentyp **data-value**:

data-value

7.5.2 Patientenakten-Eintrag (patient-data)

Eine Patientenakte enthält zu jedem Zeitpunkt mehrere verschiedene Einträge, beispielsweise den Blutdruck (angegeben als natürliche Zahl), dazu die Körpertemperatur (angegeben als Gleitkommazahl) und den Puls (angegeben als symbolischer Wert). Eine solche Sammlung von Werten soll gleichzeitig in eine elektronische Patientenakte eingefügt werden können. Daher wird ein Eintrag in die Krankenakte als eine algebraisch definierte dynamische Funktion spezifiziert, die Zeichenketten auf Datenwerte abbildet.

Definition

Ein Patientenakten-Eintrag bildet Zeichenketten auf Datenwerte ab:

patient-data : **string** $\mapsto$ **data-value**

Ist beispielsweise **PD** ein **patient-data**, der die oben angegebenen Werte speichern soll, dann kann etwa mittels **PD**(„Blutdruck“) auf den Blutdruck des Patienten zugegriffen werden.

7.5.3 Patientenakte (PDH)

Eine Patientenakte ordnet Patientenakten-Einträge zeitlich, das heißt, es ist damit möglich, unter Angabe einer Uhrzeit (als **int**-Wert) den zugehörigen Patientenakten-Eintrag zu erhalten.

Definition

Die Patientenakte ist eine algebraisch spezifizierte dynamische Funktion, die Uhrzeiten auf Patientenakten-Einträge abbildet:

patient-data-history : **int** $\mapsto$ **patient-data**

Auf die Patientenakte wird innerhalb von Asbru nur lesend zugegriffen. Da die Aktualisierungen von der Umgebung durchgeführt werden, ist es nicht notwendig Aktualisierungfunktionen zu definieren.

7.6 Zustand von Asbru

Definition

Ein Zustand von Asbru ist ein Sechs-Tupel, **state** = (**asbru-state-history**, **hierarchy-state**, **patient-data-history**, **environment-aggregation**, **bool**, **int**)
Auf Zuständen sind die folgenden Selektoren definiert:

.ASH : **state** $\mapsto$ **asbru-state-history**
.HS : **state** $\mapsto$ **hierarchy-state**
.PDH : **state** $\mapsto$ **patient-data-history**
.EAGG : **state** $\mapsto$ **environment-aggregation**
.Tick : **state** $\mapsto$ **bool**
.AC : **state** $\mapsto$ **int**

Gegeben sei ein Zustand σ mit $\sigma = (\mathbf{ASH}, \mathbf{HS}, \mathbf{PDH}, \mathbf{EAGG}, \mathbf{Tick}, \mathbf{AC})$. Dann ist die Semantik der Selektoren wie folgt definiert:

σ.ASH = **ASH**
σ.HS = **HS**
σ.PDH = **PDH**
σ.EAGG = **EAGG**
σ.Tick = **Tick**
σ.AC = **AC**

Der Zustand eines Asbru Systems enthält eine Variable vom Typ **asbru-state-history**, die die vergangenen Planzustände aller Pläne speichert und somit erlaubt zu evaluieren, wann die einzelnen Pläne welche Zustände durchlaufen haben. Ferner ist im Asbru-Zustand eine Variable vom Typ Hierarchie-Status **hierarchy-state** enthalten, die die internen Planstatus der Asbru-Hierarchie speichert. Neben diesen Variablen, die für den internen Gebrauch der Asbru-Hierarchie sind, existieren noch Variablen zur Kommunikation mit der Umgebung.

Als erstes ist in diesem Zusammenhang die Krankenakte **patient-data-history** zu nennen, die sowohl den aktuellen als auch die vergangenen Zustandsinformationen des Patienten speichert. Weiterhin Teil dieses Aspekts des Zustands ist die Umgebungssignalsammlung **environment-aggregation**, die alle Eingaben der Umgebung sammelt und dem System zur Verfügung stellt. Zur Kommunikation der Aktivität des Systems dient die **Tick** Komponente des Zustands. Mit dieser Komponente geben die Asbru-Pläne bekannt, ob ein Makro-Schritt durchgeführt werden darf oder nicht. Abschließend enthält der Zustand noch eine Uhr in der **AC**-Komponente, die speichert, zu welcher Uhrzeit dieser Zustand zuzuordnen ist.

8 Semantik von Bedingungen in Asbru

Asbru kennt zwei Arten von Bedingungen. Dies sind zum Einen Asbru-Bedingungen, die spezifizieren welche äußeren Umstände gelten müssen, damit bestimmte Zustandsübergänge möglich sind. Zum Anderen kennt Asbru Wartebedingungen, die spezifizieren, auf welche Unterpläne ein Plan warten muss, um sich erfolgreich beenden zu können. Die Semantik beider Arten von Bedingungen wird in diesem Abschnitt vorgestellt, angefangen mit der Semantik der Asbru-Bedingungen, die in Abschnitt 8.1 vorgestellt werden. Im Anschluss wird die Semantik der Wartebedingungen in Abschnitt 8.2 beschrieben.

8.1 Auswertung von Asbru-Bedingungen

In diesem Abschnitt wird erläutert wie Asbru-Bedingungen ausgewertet werden. Der erste Unterabschnitt 8.1.1 fasst kurz die Eigenschaften von Asbru-Bedingungen zusammen und erläutert diese an einem Beispiel, welches für den restlichen Abschnitt als laufendes Beispiel dient. Der darauffolgende Unterabschnitt 8.1.2 erläutert warum Asbru-Bedingungen nicht zu jedem Zeitpunkt ausgewertet werden können. Damit wird die Einführung der Auswertung mittels zweier Prädikate **satisfied** und **satisfiable** motiviert. Im Anschluss daran wird der Begriff der Signalflanken in Abschnitt 8.1.3 erläutert und beschrieben, inwiefern diese bei der Auswertung der Bedingungen berücksichtigt werden müssen. Die bis dahin aufgeführten Konzepte werden dann in Abschnitt 8.1.4 zusammengefasst, um informell die Erfüllbarkeit von Bedingungen zu beschreiben. Danach erläutert Abschnitt 8.1.5, wie das medizinische Personal die Auswertung von Bedingungen beeinflussen kann. Damit sind alle relevanten Komplikationen bei der Auswertung beschrieben und es folgt eine formale Definition der Erfüllbarkeit der Asbru-Bedingungen in Abschnitt 8.1.6.

8.1.1 Einführung an einem Beispiel

Asbru-Bedingungen formulieren eine prädikatenlogische Forderung an einen Zustand. Als Beispiel dafür soll die Bedingung „Patient hat starkes Fieber“ dienen. Eine solche Bedingung kann entweder für sich alleine stehen oder mit einem zeitlichen Bezug versehen sein. Dieser zeitliche Bezug kann viele Formen haben. Einige Beispiele dafür sind „Fieber in den letzten zwei Stunden“, „Fieber für mindestens zwei Stunden Dauer in den letzten zwei Tagen“ oder „Fieber höchstens vier Stunden nach Einnahme des Medikaments“. Damit ist es möglich, die Auswertung von Bedingungen in zeitliche Abhängigkeit zu bestimmten Ereignissen zu setzen, etwa der Einnahme eines Medikaments oder dem Abschluss einer Behandlung. Die Zeitpunkte zu denen die spezifizierten Ereignisse auftreten werden Referenzzeitpunkte oder Referenzpunkte genannt. Solche Referenzzeitpunkte werden in Asbru typischerweise in Relation zu Planzustandsübergängen von Asbru-Plänen gesetzt. Zusätzlich zu diesen Referenzpunkten werden Fristen angegeben, die zeitliche Korridore definieren, in denen das gewünschte Verhalten beobachtet werden muss. Zur genaueren Unterscheidung wird im Folgenden bei Bedingungen, die mit einem zeitlichen Bezug spezifiziert sind, die prädikatenlogische Forderung als Konditional bezeichnet, die Kombination aus dem Konditional und dem zeitlichen Bezug als Bedingung.

Ein Beispiel, bei dem eine solche zeitliche Spezifikation notwendig ist, ist die Erfolgskontrolle von medizinischen Interventionen. Angenommen, ein fiebersenkendes Medikament wird

verabreicht. Von diesem ist bekannt, dass es frühestens vier Stunden nach der Einnahme zu wirken beginnt, spätestens aber sechs Stunden danach. Ist bei einem Patienten nach sechs Stunden noch kein Fieberrückgang eingetreten, so ist das Mittel bei diesem Patienten unwirksam und muss durch ein anderes Medikament ersetzt werden. Illustriert wird dies in Abbildung 8.1. Die Auswertung dieser Bedingung findet relativ zum Start der Intervention statt, also relativ zu der Einnahme des Medikaments. Somit ist dieser Zeitpunkt der Referenzpunkt. Die Fristen im Beispiel sind vier bzw. sechs Stunden nach Planstart, zwischen denen der Fieberrückgang beobachtet werden soll. Einige der möglichen erfüllenden Korridore sind in der Grafik eingezeichnet. Es ist ausreichend, wenn während der gesamten Dauer eines einzigen Korridors das Konditional der Bedingung wahr ist. Im Laufe dieses Kapitels wird davon ausgegangen, dass das Fieber-Beispiel in Form einer Asbru-Hierarchie modelliert ist, bei dem die Einnahme des Medikaments mit dem Start des zugehörigen Behandlungsplans zusammenfällt. Weitere Details dieser Hierarchie sind für die Betrachtung dieses Beispiels nicht wichtig.

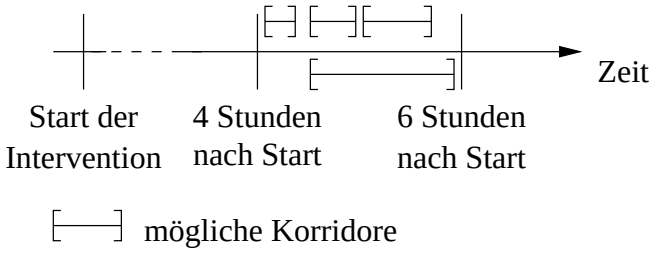


Abbildung 8.1: Einige Korridore des Beispiels

8.1.2 Zeiträume der Unentscheidbarkeit

Das in Abschnitt 8.1.1 eingeführte Beispiel zeigt, dass die Auswertung von Bedingungen nicht zu jedem Zeitpunkt möglich ist. So kann beispielsweise zwei Stunden nach Beginn der Intervention noch nicht entschieden werden, ob die Behandlung wirksam sein wird, denn diese Frage kann frühestens vier Stunden nach Planstart beantwortet werden. Kann vier Stunden nach der Medikamentengabe ein Rückgang des Fiebers gemessen werden, so folgt daraus, dass die Medikamentengabe erfolgreich war. Sinkt das Fieber jedoch zunächst nicht, so kann über die Wirksamkeit der Therapie noch keine Aussage getroffen werden. Erst sechs Stunden nach der Einnahme gilt das Medikament nach Definition als wirkungslos, wenn kein Fieberrückgang festgestellt werden kann.

Zusammenfassen lässt sich diese Betrachtung des Beispiels so, dass die Auswertung der Bedingung in mehrere Abschnitte unterteilt werden kann. Dies soll am Beispiel erläutert werden. Zunächst ist der zeitliche Abschnitt vor dem Eintreffen des Referenzereignisses, also dem Start der Intervention zu nennen. Damit ist der Zeitpunkt gemeint, zu dem der Referenzzeitpunkt ausgewertet werden kann. In diesem Zeitraum ist noch nicht einmal bekannt, mit welcher Uhrzeit der Referenzzeitpunkt „Einnahme des Medikaments“ verknüpft werden kann. Selbst Bedingungen, die in die Vergangenheit blicken („zwei Stunden in den zwei Tagen vor Behandlungsbeginn“) können nicht ausgewertet werden, sofern die Zuordnung des Referenzzeitpunkts zu einer konkreten Uhrzeit nicht bekannt ist. Kann der Referenzzeitpunkt ausgewertet werden, so ist der nächste logische Abschnitt der, in dem keine Auswertung möglich ist, obwohl der Referenzpunkt einer konkreten Zeit zugeordnet werden kann. Im Beispiel sind das die vier Stunden unmittelbar nach Behandlungsbeginn. Unabhängig vom Zustand kann in dieser Zeit nur ausgesagt werden, dass noch eine Möglichkeit existiert, dass die Bedingung wahr werden kann. Jedoch ist diese Möglichkeit noch nicht Realität geworden. Asbru bezeichnet

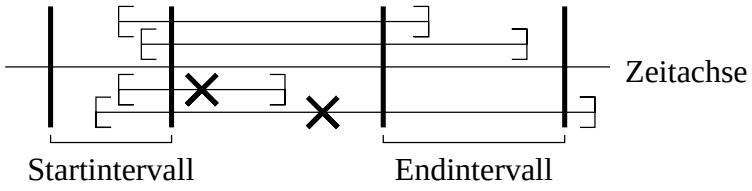


Abbildung 8.2: Beispiel für Korridore und Zeit-Annotationen

eine Bedingung in diesem Auswertungszustand als „erfüllbar“. Im Zeitraum zwischen vier und sechs Stunden nach Behandlungsbeginn ist die Bedingung noch immer erfüllbar. Tritt hier allerdings ein Fieberrückgang ein, so gilt die Bedingung als „erfüllt“. Tritt dagegen bis einschließlich sechs Stunden nach Einnahme des Medikaments keine Besserung ein, so ist die Bedingung nicht mehr erfüllbar (und auch nicht erfüllt).

Um diese Unsicherheiten bei der Auswertung zu erfassen, erfolgt die Auswertung der Richtigkeit der Bedingungen mit Hilfe zweier Prädikate, **satisfied** und **satisfiable**. Ersteres entscheidet, ob eine Bedingung nach der Nomenklatur des letzten Abschnitts erfüllt ist, letzteres, ob sie erfüllbar ist. Ein wenig formaler betrachtet bedeutet eine erfüllte Bedingung, dass ein Korridor in der Vergangenheit existiert, in dem das Konditional der Bedingung kontinuierlich wahr ist. Am Beispiel betrachtet könnte ein externer Beobachter acht Stunden nach Planstart das Krankenblatt eines Patienten ansehen und dabei feststellen, dass in einem Zeitraum von 30 Minuten Länge, der fünf Stunden nach Einnahme des Medikaments begann, das Fieber gesunken ist. Somit ist die Bedingung erfüllt. Das Prädikat **satisfiable** hingegen entscheidet, ob ein Korridor existiert, der ganz oder teilweise in der Zukunft liegt und die zeitliche Spezifikation erfüllt. Für die Erfüllbarkeit wird weiterhin verlangt, dass in den Teilen des Korridors, die in der Vergangenheit liegen, das Konditional erfüllt sein muss. Wird die Erfüllbarkeit der Bedingung im Beispiel zwei Stunden nachdem das Medikament gegeben wurde geprüft, so muss ein entsprechender Korridor gefunden werden, der die zeitliche Spezifikation erfüllt und der ganz oder teilweise in der Zukunft liegt. Beginnt diese Korridor 5 Stunden nach Beginn der Intervention und hat eine Länge von 20 Minuten, so erfüllt er die zeitliche Spezifikation. Da der Korridor komplett in der Zukunft liegt, ist die Bedingung damit erfüllbar.

Die zeitlichen Abhängigkeiten einer Bedingung werden in Asbru mittels der Datenstruktur **abstract-time-annotation** (Abschnitt 6.4) spezifiziert. Mit dieser können alle für die Menge der Korridore relevanten Angaben, wie Höchstdauer oder Startzeitpunkt, unabhängig voneinander spezifiziert werden oder unspezifiziert bleiben. Ein unspezifizierter Wert entspricht dabei der Aussage, dass der Wert nicht relevant ist. Beispielsweise wäre für die Wirkung des fiebersenkenden Mittels eine Höchstdauer der Wirkung irrelevant. Die gültigen Korridore, die eine Bedingung erfüllbar bzw. erfüllt machen, können mit Datenstrukturen vom Typ **abstract-time-annotation** durch Spezifikation von Startzeiträumen, Zielzeiträumen und Längenkorridoren eingeschränkt werden. Ausschließlich solche Korridore können eine Bedingung erfüllen, die in dem spezifizierten Startzeitraum beginnen, dem Endzeitraum enden und deren Dauer innerhalb des Längenkorridors liegt. Werden alle Zeit-Werte unspezifiziert gelassen, so entspricht dies der Forderung, dass ein vollständig beliebiger Korridor existiert, in dem die Bedingung gilt. Daraus ergibt sich eine mengentheoretische Sicht auf diese Datentypen: Eine **abstract-time-annotation** steht stellvertretend für eine (unendliche) Menge von Korridoren. Erfüllt ein Korridor alle Bedingungen der Zeit-Annotation, so ist er ein Element dieser Zeit-Annotation. Einige Beispiele finden sich in Abbildung 8.2. Dabei sind im oberen Teil der Grafik zwei Beispiele für Korridore angegeben, die die zeitliche Spezifikation der Zeit-Annotation erfüllen, im unteren Bereich der Grafik sind zwei nicht erfüllende Beispiele gezeichnet. Die beiden unteren Beispiele sind nicht in der Zeit-Annotation enthalten, da das

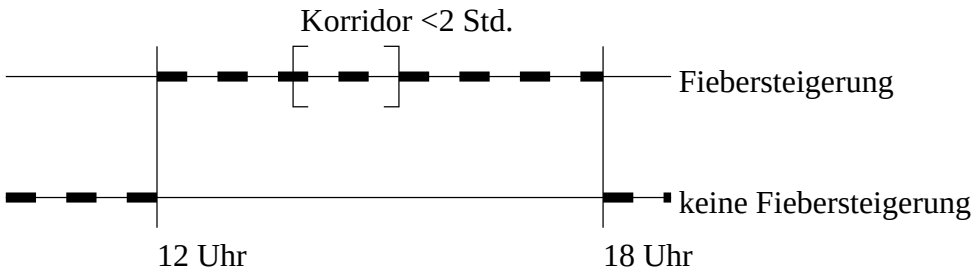


Abbildung 8.3: Beispiel für die Notwendigkeit von Signalflanken

erste Beispiel zu früh, das zweite zu spät endet.

Das Fieber-Beispiel aus Abschnitt 8.1.1 verlangt, dass ein Fieberrückgang zwischen vier und sechs Stunden nach Planstart festgestellt werden muss. Angenommen, der Planinstanzname des Plans zur Einnahme des fiebersenkenden Medikaments sei **iname**. Dann kann die Zeit-Annotation zu dem Beispiel geschrieben werden als:

$$([\perp, +\text{hour}(6)][+\text{hour}(4), \perp][0, \perp], \textbf{enter}(\textbf{iname}, \texttt{activated}))$$

Das heißt, gesucht werden Korridore, die bis sechs Stunden nach Planstart beginnen (spätester Startzeitpunkt), nicht früher als vier Stunden nach Planstart enden (frühester Endzeitpunkt) und deren Länge beliebig ist.

8.1.3 Umgang mit Signalflanken

Eine Komplikation bei der Untersuchung der Erfüllbarkeit von Bedingungen sind Signalflanken. Teilweise verlangen Bedingungen, dass ein bestimmtes Verhalten zeitlich beschränkt ist. Das Fieber-Beispiel lässt sich dahingehend erweitern. Angenommen es stehen mehrere verschiedene, unterschiedlich starke Medikamente zur Verfügung und die nötige Stärke des zu verabreichenden Medikaments soll bestimmt werden. Dafür ist nicht nur die aktuelle Höhe des Fiebers wichtig, sondern auch die Länge des Anstiegs der Temperatur. Je länger der Anstieg andauert, umso bedenklicher ist der Zustand des Patienten. Dauert der Anstieg nur kurz und bleibt das Fieber dann auf konstantem Niveau wird eine weniger aggressive Therapie angewendet, als wenn das Fieber über einen längeren Zeitraum ansteigt. In Asbru wird nun formuliert, dass das schwache Medikament nur verabreicht werden darf, wenn der Anstieg höchstens zwei Stunden gedauert hat. Andernfalls soll das stärkere Medikament gegeben werden. Die Bedingung, die überprüft, ob das schwächere Medikament gegeben wird, soll nicht zu wahr auswerten, wenn der Fieberanstieg länger als zwei Stunden gedauert hat.

Mit den bisher vorgestellten Mechanismen ist dies nicht möglich, da offensichtlich ein Korridor gefunden werden kann, der nicht länger als zwei Stunden ist und in dem das Fieber angestiegen ist. Dies ist in Abbildung 8.3 visualisiert. Obwohl der Fieberanstieg sich über insgesamt sechs Stunden hinzieht, kann ein kürzerer Korridor gefunden werden, der die Bedingung wahr macht. Daher wird der Begriff der Signalflanken eingeführt. Eine Signalflanke beschreibt einen Zustandswechsel, im Beispiel um 12 Uhr, wenn das Fieber beginnt zu steigen bzw. um 18 Uhr, wenn der Fieberanstieg beendet wird. Die Signalflanke um 12 Uhr wird als positive oder steigende Signalflanke bezeichnet, da das Konditional wahr wird. Um 18 Uhr dagegen findet sich eine fallende oder negative Signalflanke, da das Konditional hier falsch wird. Die Zeit-Annotation einer Bedingung kann verlangen, dass bei der Auswertung der Asbru-Bedingung eine steigende Signalflanke zu Beginn des erfüllenden Korridors vorhanden sein muss, ebenso kann spezifiziert werden, dass eine fallende Signalflanke am Ende des Korridors vorhanden sein muss oder das an beiden Enden Flanken existieren müssen. Ein Korridor, der

mit einer fallenden Signalflanke beendet wird, wird als maximal im Ende bezeichnet. Analog wird ein Korridor, der mit einer steigenden Signalflanke beginnt als maximal im Anfang bezeichnet. Ein Korridor, der maximal im Ende und maximal im Anfang ist, heißt maximal.

8.1.4 Berechnung der Erfülltheit von Asbru-Bedingungen

Um zu berechnen, ob eine Bedingung erfüllt ist, muss man zunächst unterscheiden, ob die Bedingung mit einer Zeit-Annotation versehen ist oder nicht. Falls keine Zeit-Annotation angegeben ist, muss lediglich das Konditional der Bedingung ausgewertet werden. Ist dieses wahr, so gilt die Bedingung als erfüllbar und als erfüllt, anderenfalls ist sie keines von beiden.

Sollte eine **abstract-time-annotation** angegeben worden sein, so muss zunächst die Menge der möglichen Korridore berechnet werden, die zu der Zeit-Annotation gehört. Um dies zu erreichen, muss zunächst der Referenzzeitpunkt der Zeit-Annotation ausgewertet werden. Sofern er ein Ereignis wie etwa einen Planstart oder eine Planterminierung referenziert, muss anhand des aktuellen Zustandes ermittelt werden, ob eine Zuordnung des Ereignisses zu einer konkreten Zeit möglich ist. Ist dies möglich, so kann die Menge aller Korridore berechnet werden. Die Berechnung von Erfüllbarkeit und Erfülltheit wird dann im Folgenden beschrieben. Kann der Referenzpunkt nicht berechnet werden, weil etwa der referenzierte Planzustandswechsel noch nicht stattgefunden hat, gilt die Bedingung als erfüllbar aber nicht als erfüllt.

Eine Zeit-annotierte Bedingung genügt in einem Zustand dem Prädikat **satisfied**, ist also erfüllt, wenn einer der Korridore, die Teil der Zeit-Annotation sind, komplett in der Vergangenheit liegt. Über die gesamte Länge dieses Korridors muss das Konditional der Bedingung wahr sein. Gegebenenfalls müssen Flanken, die die Zeit-Annotation spezifiziert, berücksichtigt werden. Im Beispiel der Fieberbehandlung aus Abschnitt 8.1.1 bedeutet dies, dass ein beliebig langer Korridor gefunden werden muss, der vier bis sechs Stunden nach Gabe des fiebersenkenden Medikaments beginnt oder endet. Während des ganzen Korridors (der beliebig kurz sein darf), muss das Fieber gesunken sein.

Eine Zeit-annotierte Bedingung ist in einem Zustand erfüllbar, d.h. das Prädikat **satisfiable** wertet wahr aus, wenn für einen der Korridore, die Elemente der Zeit-Annotation sind, gilt, dass für jeden Zeitpunkt dieses Korridors das Konditional der Bedingung über dem Zustand zu wahr auswertet oder der Zeitpunkt in der Zukunft des aktuellen Zustands liegt. Gleiches gilt für Flanken. Diese müssen nachgewiesen werden, sofern sie von der Zeit-Annotation der Bedingung gefordert werden und der Zeitpunkt der Flanke nicht in der Zukunft liegt. Im Beispiel kann man beispielsweise zum Zeitpunkt, zwei Stunden nach Planstart, die Erfüllbarkeit der Bedingung nachweisen. Dafür wird ein Korridor gewählt, der vier Stunden nach Medikamentengabe beginnt und eine halbe Stunde lang ist. Da jeder einzelne Punkt dieses Korridors in der Zukunft liegt, ist die Bedingung erfüllbar.

Ist der Referenzpunkt der Bedingung nicht ermittelbar, beispielsweise, weil das spezifizierte Ereignis noch nicht eingetreten ist, so gilt die Bedingung als erfüllbar, jedoch nicht als erfüllt.

8.1.5 Benutzereinwirkung auf die Auswertung von Bedingungen

Asbru Bedingungen (**asbru-condition**) erweitern Bedingungen um die Möglichkeit zur Einbeziehung der externen Signale, die in Abschnitt 7.2 beschrieben werden. Das medizinische Personal kann mit diesen Signalen die Auswertung von Bedingungen verzögern oder übersteuern. Um diese Einflussnahme nur gezielt zuzulassen, besteht eine Asbru-Bedingung (Abschnitt 6.7) neben einer Bedingung aus zwei **bool**-Werten. Diese Marker spezifizieren, ob die Auswertung der Asbru-Bedingung von den Umgebungssignalen beeinflusst werden kann. Einer der Marker wird mittels des Selektors **.override** angesprochen. Dieser Marker gibt an, ob die Bedingung vom medizinischen Personal übersteuert werden kann. Erlaubt die Bedingung ein Übersteuern und sendet das medizinische Personal zusätzlich das Übersteuerungs-

signal, dann wertet das Prädikat **satisfied** die Asbru-Bedingung zu wahr aus. Dies erfolgt unabhängig von der Auswertung der Bedingung, die der Asbru-Bedingung zugrunde liegt. Ist in einer Asbru-Bedingung der **.override** Marker gesetzt, so ist die Bedingung immer erfüllbar. Dies soll am konkreten Beispiel erläutert werden. Dazu wird die **filter**-Bedingung des Plans `Fieber Behandlung` betrachtet. Dabei handelt es sich um eine Asbru-Bedingung, die die Anwendung des Plans immer dann erlaubt, wenn mindestens moderates Fieber vorliegt. Diese Asbru-Bedingung ist unabhängig vom Zustand immer erfüllbar, wenn gilt, dass `asbru`(`Fieber Behandlung`)**.filter .override** den Wert **true** ergibt. Weiterhin ist diese Bedingung erfüllt, wenn obiges gilt und zusätzlich vom medizinischen Personal ein override Signal gesendet wird. Dieses Signal wird in der Umgebungssignal-Sammlung **EAGG** gespeichert. Auf die den Plan betreffende Signalsammlung wird mittels des Terms

EAGG[`Fieber Behandlung`]

zugegriffen. Ergebnis dieses Terms ist die Sammlung der 13 verschiedenen Signale, über die ein Plan gesteuert werden kann. Aus diesen 13 Signalen muss als nächstes die Signalgruppe der override Signale ausgewählt werden.

EAGG[`Fieber Behandlung`]**.override**

Ergebnis dieses Aufrufs ist ein Wert vom Typ **bool-tupel**, das die sechs override Signale für alle Bedingungen des Plans enthält. Von diesen muss zuletzt das Signal für die **filter**-Bedingung selektiert werden:

EAGG[`Fieber Behandlung`]**.override.filter**

Dieses Signal muss den Wert **true** haben.

Ist der mittels **.confirm** selektierte Marker einer Asbru-Bedingung gesetzt, so kann die Auswertung der Asbru Bedingung verzögert werden. Durch ein entsprechendes externes Signal wird bei gesetztem **.confirm** Marker das Prädikat **satisfied** zu falsch auswertet. Die Auswertung von **satisfiable** wird durch dieses Flag nicht verändert.

8.1.6 Formale Erfüllbarkeit von Asbru Bedingungen

Ziel dieses Abschnitts ist es, eine formale Definition der Erfüllbarkeit und der Erfülltheit von Asbru-Bedingungen zu geben. Zentraler Aspekt dabei ist die Betrachtung zeitlicher Aspekte und die damit verbundene Frage, ob die zeitliche Spezifikation der Bedingung eingehalten werden kann oder nicht. Um diese Auswertung zu ermöglichen soll wie folgt verfahren werden. Zunächst wird der Begriff des Intervalls bzw. des abstrakten Intervalls eingeführt. Ein Intervall soll als Datenstruktur einen Korridor repräsentieren.

Danach wird die formale Semantik der Auswertung der abstrakten Referenzpunkte, also des **abstract-asbru-clock** Datentyps, angegeben. Diese abstrakten Referenzpunkte erlauben neben der Angabe einer konkreten Uhrzeit auch, Planzustandswechsel anderer Pläne zu referenzieren. Diese auszuwerten ist notwendig, um mit abstrakten Zeit-Annotationen umgehen zu können, da diese solche abstrakte Referenzpunkte verwenden. Die Auswertung der abstrakten Zeit-Annotationen ist notwendig um die Menge der von einer solchen Zeit-Annotation definierten Korridore bestimmen zu können. Nachdem eine formale Definition für diese Menge angegeben worden ist, wird definiert, ob bei gegebenem Zustand eine Asbru-Bedingung erfüllt bzw. erfüllbar ist.

Semantik von Konditionalen (conditional)

Konditionale sind die einfachsten Bedingungstypen, die in Asbru definiert sind. Konditionale speichern eine nicht zeitbezogene Bedingung, etwa „der Patient hat Fieber“.

Definition

Ein Konditional **conditional** ist ein Prädikat, das die Information einer Krankenakte und einer Uhrzeit zu einem Wahrheitswert auswertet. Damit ist ein **conditional** eine Implementierung der in Abschnitt 6.9 vorgestellten Datenstruktur.

conditional : **patient-data-history** $\times$ **int** $\mapsto$ **bool**

Ein **conditional cond** kann über einem Zustand σ ausgewertet werden:

$[\![\mathbf{cond}]\!]_\sigma \leftrightarrow \mathbf{cond}(\sigma.\mathrm{PDH}, \sigma.\mathrm{AC})$

Intervalle (abstract-interval und interval)

Ist es in Asbru notwendig, eine Bedingung mit einem zeitlichen Bezug zu versehen, so wird dazu eine Zeit-Annotation (siehe Abschnitt 6.4) verwendet. Um diese besser handhaben zu können, wird zunächst das Konzept des **interval** bzw. des **abstract-interval** eingeführt. Diese Konzepte implementieren Korridore, wie sie im Beispiel 8.1.1 vorgestellt werden.

interval und **abstract-interval** unterscheiden sich darin, dass ein **interval** lediglich ein Intervall zwischen zwei ganzen Zahlen beschreibt, während ein **abstract-interval** auch $\perp$-Elemente als Intervallenden zulässt. $\perp$-Elemente repräsentieren undefinierte Werte, die beispielsweise dadurch auftreten können, dass bestimmte zeitliche Aspekte bei Zeit-Annotationen unspezifiziert gelassen werden.

Definition

Ein **abstract-interval** ist ein Paar aus zwei **int-bottom** Zahlen, **abstract-interval** = (**int-bottom** $\times$ **int-bottom**).

Zum Zugriff auf den Start- bzw. den Endwert eines **abstract-interval** Konstrukts werden Selektoren definiert:

..start : **abstract-interval** $\mapsto$ **abstract-asbru-clock**
..end : **abstract-interval** $\mapsto$ **abstract-asbru-clock**

Die Semantik der Selektoren wird formal wie folgt spezifiziert:

$(\mathbf{bi}_1, \mathbf{bi}_2)\mathbf{.start} = \mathbf{bi}_1$
$(\mathbf{bi}_1, \mathbf{bi}_2)\mathbf{.end} = \mathbf{bi}_2$

Abstrakte Intervalle dienen der Aufnahme von Zeitintervallen im Zusammenhang mit Zeit-Annotationen. Eine Zeit-Annotation wird beschrieben durch die Spezifikation eines Startintervalls, eines Zielintervalls sowie eines Längenintervalls. Diese drei Intervalle können durch die **abstract-interval** Datenstruktur repräsentiert werden. Die Selektoren, die auf

abstract-interval Datentypen definiert sind, heißen **.start** und **.end**. **.start** selektiert den ersten Wert des Intervalls, also den Startpunkt. **.end** selektiert den letzten Wert des Intervalls, also den Endpunkt.

Analog dazu wird die Definition der nicht-abstrakten Intervalle **interval** angegeben. Verwendet werden die **interval** unter anderem dafür, die zeitlichen Korridore zu spezifizieren, die in der Menge der von einer Zeit-Annotation beschriebenen Korridore enthalten sind.

Definition

Ein **interval** ist ein Paar ganzer Zahlen, **interval** = (**int** × **int**).

Wie für die abstrakten Korridore **abstract-interval** werden Selektoren für die beiden Komponenten definiert.

$$\begin{array}{lcl} \textbf{..start} & : & \textbf{interval} \mapsto \textbf{int} \\ \textbf{..end} & : & \textbf{interval} \mapsto \textbf{int} \end{array}$$

Seien $\mathbf{AC}_1$[1] und $\mathbf{AC}_2$ ganze Zahlen, dann wird damit die Semantik der Selektoren wie folgt spezifiziert:

$$\begin{array}{lcl} (\mathbf{AC}_1, \mathbf{AC}_2)\textbf{.start} & = & \mathbf{AC}_1 \\ (\mathbf{AC}_1, \mathbf{AC}_2)\textbf{.end} & = & \mathbf{AC}_2 \end{array}$$

Damit werden Intervalle analog zu den abstrakten Intervallen spezifiziert.

Semantik der abstrakten Referenzpunkte (abstract-asbru-clock)

Um die Auswertung von Zeit-annotierten Bedingungen durchführen zu können, muss es möglich sein, die Menge der durch eine Zeit-Annotation definierten Intervalle zu bestimmen. Für das Beispiel der Behandlung von Fieber in Abschnitt 8.1.1 wurde die Zeit-Annotation

$$([\perp, +\text{hour}(6)][+\text{hour}(4), \perp][0, \perp], \textbf{enter}(\textbf{iname}, \texttt{activated}))$$

formuliert. Um auswerten zu können, ob ein Korridor in der Menge der von dieser Zeit-Annotationen zugelassenen Korridoren liegt, muss zunächst der Referenzpunkt ausgewertet werden. Alle anderen Werte der Zeit-Annotation sind relativ zu diesem angegeben. Zu absoluten Werten werden diese durch die Addition mit dem **abstract-asbru-clock** Objekt, im Beispiel **enter**(**iname**, `activated`).

Ein Objekt vom Typ **abstract-asbru-clock** (siehe Abschnitt 6.11) beschreibt allgemein einen abstrakten Referenzzeitpunkt. Im einfachsten Fall handelt es sich bei einem **abstract-asbru-clock** Objekt um eine konkrete Uhrzeit, jedoch sind als gültige Werte auch die Referenzierung von Planzustandswechseln – wie im Beispiel – und der wandelnde Referenzpunkt ***now*** zugelassen. **abstract-asbru-clock** Objekte werden als Referenzpunkte für Zeit-Annotationen in Asbru-Bedingungen verwendet. Im Beispiel zur Behandlung von Fieber wird eine Bedingung angegeben, die vier bis sechs Stunden nach Planstart gültig sein soll. Offensichtlich kann bei der Erstellung der Leitlinie keine absolute Zeit als Referenzpunkt angegeben werden, da die Ausführungszeit des entsprechenden Plans bei der Erstellung der Leitlinie unbekannt ist. Stattdessen wird als Referenzpunkt der Moment gewählt, in dem der Plan zur Verabreichung des Medikaments seinen Planzustand von `ready` auf `activated` ändert. Die Semantik des dafür vorgesehenen syntaktischen Konstrukts **enter** soll in diesem Abschnitt definiert werden, genauso wie die Semantik

des inversen Konstruktes **leave**. **leave** beschreibt einen Zeitpunkt, zu dem ein Plan einen bestimmten Zustand verlässt. Das heißt, im Beispiel kann der Zeitpunkt, in dem der Plan vom Zustand `ready` nach `activated` wechselt beschrieben werden als **enter**(**iname**, `activated`) oder als **leave**(**iname**, `ready`).

Für die semantische Auswertung der **enter** und **leave** Strukturen muss ein Zustand vorliegen, aus dessen Asbru-Zustands-Historie abgelesen werden kann, ob und wann der entsprechende Planzustandswechsel stattgefunden hat. Die **abstract-asbru-clock** Objekte müssen anhand dieses Zustands (bzw. dessen Asbru-Zustands-Historie) ausgewertet werden. Eine solche Auswertung über dem Zustand σ wird syntaktisch mit der Formel $[\![\mathbf{aac}]\!]_\sigma$ aufgeschrieben. Beschreibt das **abstract-asbru-clock** Objekt einen Planzustandswechsel, so müssen bei der Auswertung zwei Fälle unterscheiden werden. Im einen Fall hat der referenzierte Planzustandswechsel bereits in der Vergangenheit des aktuellen Zustands stattgefunden, im anderen Fall nicht. Hat der Zustandswechsel stattgefunden, so soll $[\![\mathbf{aac}]\!]_\sigma$ den Zeitpunkt dieses Zustandswechsels zurückliefern. Falls der Zustandswechsel noch nicht stattgefunden hat, soll $[\![\mathbf{aac}]\!]_\sigma$ zu $\perp$ auswerten.

Falls der referenzierte Planzustandswechsel mehrfach stattgefunden hat, so soll jeweils der Zeitpunkt der letzten solchen Planzustandsänderung als Ergebnis der Auswertung stehen.

Um die Auswertung von **enter** und **leave** durchführen zu können, müssen Hilfsprädikate eingeführt werden. Diese heißen **enterP** und **leaveP**. Während **enter** und **leave** bei der Auswertung berechnen, zu welcher Zeit eine Planinstanz einen bestimmten Zustandswechsel zum letzten mal durchgeführt hat, berechnen **enterP** und **leaveP** *ob* ein Planzustandswechsel zu der angegeben Zeit stattgefunden hat. Es gilt also:

$$[\![\mathbf{enter}(\mathbf{iname}, \theta)]\!]_\sigma = \mathbf{AC} \rightarrow [\![\mathbf{enterP}(\mathbf{iname}, \theta, \mathbf{AC})]\!]_\sigma$$

Definition

enterP ist ein Konstrukt bestehend aus Planinstanznamen, Planzustand und Uhrzeit, **enterP**(**instance-name** $\times$ **plan-state** $\times$ **int**).

Ein **enterP** Konstrukt kann über einem Zustand ausgewertet werden, $[\![\mathbf{enterP}(...)]\!]_\sigma$. Dazu gilt folgende Definition, wenn **iname** ein Planinstanzname, **PS** ein Planzustand und **AC** eine Uhrzeit ist:

$$\begin{aligned}[\![\mathbf{enterP}(\mathbf{iname}, \mathbf{PS}, \mathbf{AC})]\!]_\sigma \quad \leftrightarrow \quad & \exists\, \mathbf{PSX}_1, \mathbf{PSX}_2, \mathbf{PS}_1. \\ & \quad \sigma.\mathrm{ASH}[\mathbf{iname}][\mathbf{AC}] \\ & \quad = \mathbf{PSX}_1 + \mathbf{PS}_1 + \mathbf{PS} + \mathbf{PSX}_2 \\ & \wedge\, \mathbf{PS}_1 \neq \mathbf{PS} \\ & \wedge\, \mathbf{AC} \leq \sigma.\mathrm{AC}\end{aligned}$$

Das Konstrukt **enterP** beschreibt einen Planzustandswechsel in den angegebenen Planzustand zur angegebenen Zeit. Bei Auswertung von **enterP**(**iname**, **PS**, **AC**) über einem Zustand σ ($[\![\mathbf{enterP}(\mathbf{iname}, \mathbf{PS}, \mathbf{AC})]\!]_\sigma$) wird überprüft, ob die Planinstanz, die durch den Namen **iname** benannt wird zum Zeitpunkt **AC** einen Zustandsübergang von einem beliebigem Zustand in den Zustand **PS** durchgeführt hat. Dies geschieht wie nachfolgend beschrieben.

Aus dem Zustand σ wird mittels .ASH die Asbru-Zustands-Historien Komponente selektiert. Diese ist eine dynamische Funktion, die Plannamen abbildet auf Einträge in die Asbru-Zustands-Historie. Das bedeutet, der Term

$$\sigma.\text{ASH}[\textbf{iname}]$$

selektiert das gesamte protokollierte Zustandsübergangsverhalten des Planes **iname**. Um festzustellen, ob ein Zustandsübergang zum Zeitpunkt **AC** stattgefunden hat, wird die Planzustandsliste dieses Zeitpunktes aus dem Eintrag selektiert.

$$\sigma.\text{ASH}[\textbf{iname}][\textbf{AC}]$$

Wenn der Plan **iname** zum Zeitpunkt **AC** in den Zustand **PS** gewechselt ist, so ist das gleichbedeutend damit, dass in der Planzustandsliste, die zu diesem Plan und dieser Zeit gehört der Planzustand **PS** enthalten ist. Weiterhin muss ein *Wechsel* in diesen Zustand stattgefunden haben, also eine Zerlegung der Planzustandsliste existieren, so dass der Zustand **PS** darin vorkommt und der Zustand unmittelbar vor **PS** ungleich diesem ist.

Die Definition soll nun nochmals anhand des Beispiels betrachtet werden. Zur Erinnerung, die Zeit-Annotation des Beispiels ist

$$([\bot, +\text{hour}(6)][+\text{hour}(4), \bot][0, \bot], \textbf{enter}(\textbf{iname}, \texttt{activated}))$$

Eine mögliche Zustandshistorie für den Plan **iname** hat die nachfolgende Form:

$$\begin{array}{lcl}
\textbf{ASHE}[0] & = & ([] \Rightarrow \texttt{inactive}) \\
\textbf{ASHE}[1] & = & ([] \Rightarrow \texttt{inactive} \Rightarrow \texttt{considered} \Rightarrow \texttt{possible} \Rightarrow \texttt{ready}) \\
\textbf{ASHE}[2] & = & ([] \Rightarrow \texttt{ready} \Rightarrow \texttt{activated}) \\
\textbf{ASHE}[3] & = & ([] \Rightarrow \texttt{activated}) \\
\textbf{ASHE}[4] & = & ([] \Rightarrow \texttt{activated} \Rightarrow \texttt{suspended}) \\
\textbf{ASHE}[5] & = & ([] \Rightarrow \texttt{suspended} \Rightarrow \texttt{activated})
\end{array}$$

Das heißt, der Plan war zum Zeitpunkt 0 im Zustand `inactive`. Im Zustand 1 wurde die `Selection`-Phase durchlaufen, im Zustand 2 erfolgte der Übergang in die `Execution`-Phase, im Zustand 4 wurde der Plan unterbrochen um dann im Zustand 5 wieder in den Zustand `activated` zu wechseln. $\textbf{enterP}(\textbf{iname}, \texttt{activated}, \textbf{AC})$ soll den Wert **true** bei der Auswertung erhalten, wenn **AC** für eine Zeit steht, zu der der Plan **iname** den Planzustand `activated` betreten hat. Im Beispiel gilt dies für $\textbf{AC} = 2$ und $\textbf{AC} = 5$. Zu diesen Zeitpunkten hat der Planzustand von einem beliebigen nicht-`activated` Zustand zum `activated`-Zustand gewechselt. Im Zustand 3 sowie im Zustand 4 war der Plan zwar im Zustand `activated`, hat aber nicht dorthin gewechselt. Daher wertet $\textbf{enterP}(\textbf{iname}, \texttt{activated}, 3)$ zu **false** aus.

Definition

Analog zu **enterP** ist das **leaveP** Konstrukt definiert mit **leaveP**(**instance-name** $\times$ **plan-state** $\times$ **int**).

Auch dessen Semantik ist analog zum **enterP** Konstrukt definiert:

$$\begin{array}{lcl}
[\![\textbf{leaveP}(\textbf{iname}, \textbf{PS}, \textbf{AC})]\!]_\sigma & \leftrightarrow & \exists\, \textbf{PSX}_1, \textbf{PSX}_2, \textbf{PS}_1. \\
 & & \quad \sigma.\text{ASH}[\textbf{iname}][\textbf{AC}] \\
 & & \quad = \textbf{PSX}_1 + \textbf{PS} + \textbf{PS}_1 + \textbf{PSX}_2 \\
 & & \wedge\, \textbf{PS}_1 \neq \textbf{PS} \\
 & & \wedge\, \textbf{AC} \leq \sigma.\text{AC}
\end{array}$$

Der Unterschied zwischen der Semantikdefinition von **enterP** und **leaveP** ist, dass im Fall von **enterP** gefordert wird, dass der betreffende Zustand betreten wird ($\mathbf{PS}_1 + \mathbf{PS}$), während bei leaveP der Zustand verlassen wird ($\mathbf{PS} + \mathbf{PS}_1$).

Nun soll die Auswertung abstrakter Referenzpunkte definiert werden. Dazu wird zunächst eine Auswertungsregel für den Referenzpunkt ***now*** angegeben, danach wird die Auswertung von **enter** und **leave** Konstrukten angegeben.

Definition

Gegeben sei ein Zustand σ. Dann wird der abstrakte Referenzpunkt ***now*** wie folgt ausgewertet:

$$[\![\textbf{*now*}]\!]_\sigma \quad = \quad \sigma.\text{AC}$$

Die Auswertung des Abstrakten Referenzpunkte **enter** stützt sich auf das eben definierte Konstrukt **enterP** ab.

$$\begin{array}{lcl}
[\![\textbf{enter}(\textbf{iname}, \textbf{PS})]\!]_\sigma & = & \bot, \text{wenn kein } \mathbf{AC} \text{ existiert, so dass} [\![\textbf{enterP}(\textbf{iname}, \textbf{PS}, \textbf{AC})]\!]_\sigma \\
[\![\textbf{enter}(\textbf{iname}, \textbf{PS})]\!]_\sigma & = & \mathbf{AC}, \text{wenn} \quad [\![\textbf{enterP}(\textbf{iname}, \textbf{PS}, \textbf{AC})]\!]_\sigma \\
 & & \quad \text{und es existiert kein } \mathbf{AC}_1, \text{ so dass} \\
 & & \quad\quad [\![\textbf{enterP}(\textbf{iname}, \textbf{PS}, \mathbf{AC}_1)]\!]_\sigma \\
 & & \quad \text{und } \mathbf{AC} < \mathbf{AC}_1 \\
 & & \quad \text{und } \mathbf{AC} \leq \sigma.\text{AC}
\end{array}$$

Analog dazu ist die Semantik des **leave** Konstrukts definiert:

$$\begin{array}{lcl}
[\![\textbf{leave}(\textbf{iname}, \textbf{PS})]\!]_\sigma & = & \bot, \text{wenn kein } \mathbf{AC} \text{ existiert, so dass} [\![\textbf{leaveP}(\textbf{iname}, \textbf{PS}, \textbf{AC})]\!]_\sigma \\
[\![\textbf{leave}(\textbf{iname}, \textbf{PS})]\!]_\sigma & = & \mathbf{AC}, \text{wenn} \quad [\![\textbf{leaveP}(\textbf{iname}, \textbf{PS}, \textbf{AC})]\!]_\sigma \\
 & & \quad \text{und es existiert kein } \mathbf{AC}_1, \text{ so dass} \\
 & & \quad\quad [\![\textbf{leaveP}(\textbf{iname}, \textbf{PS}, \mathbf{AC}_1)]\!]_\sigma \\
 & & \quad \text{und } \mathbf{AC} < \mathbf{AC}_1 \\
 & & \quad \text{und } \mathbf{AC} \leq \sigma.\text{AC}
\end{array}$$

Ein ***now*** Konstrukt wird bei der Auswertung stets durch die jeweilige Uhrzeit des Zustands ersetzt, die mittels des .AC Selektors aus dem Zustand selektiert wird.

Ein **enter** Konstrukt soll durch die Auswertung zum letzten (d.h. neuesten) Zeitpunkt ausgewertet werden, zu dem ein Plan den angegebenen Zustand erreicht hat.

Zur Erläuterung soll noch einmal das Beispiel von eben herangezogen werden. Der abstrakte Referenzpunkt **enter**(**iname**, `activated`) soll ausgewertet werden, wobei folgende Zustandshistorie für den Plan **iname** gilt:

$$\begin{array}{lcl}
\textbf{ASHE}[0] & = & ([] \Rightarrow \texttt{inactive}) \\
\textbf{ASHE}[1] & = & ([] \Rightarrow \texttt{inactive} \Rightarrow \texttt{considered} \Rightarrow \texttt{possible} \Rightarrow \texttt{ready}) \\
\textbf{ASHE}[2] & = & ([] \Rightarrow \texttt{ready} \Rightarrow \texttt{activated}) \\
\textbf{ASHE}[3] & = & ([] \Rightarrow \texttt{activated}) \\
\textbf{ASHE}[4] & = & ([] \Rightarrow \texttt{activated} \Rightarrow \texttt{suspended}) \\
\textbf{ASHE}[5] & = & ([] \Rightarrow \texttt{suspended} \Rightarrow \texttt{activated})
\end{array}$$

Wie bereits oben erläutert wertet **enterP** für die Zeitpunkte 2 und 5 zu **true** aus. Dementsprechend gibt es mindestens einen Zeitpunkt, zu dem **enterP** nicht zu **false** auswertet. Damit

wird die Auswertung von **enter** nicht zum $\perp$ Element geschehen. Stattdessen wird nun nach einem Zeitpunkt gesucht, zu dem **enterP** zu **true** auswertet (2 oder 5) und dieser Zeitpunkt muss der späteste sein, zu dem die Auswertung zu **true** erfolgt. Damit fällt der Zeitpunkt 2 aus und **enter** wertet zu 5 aus.

Die Definition des **leave** Konstruktes erfolgt analog zum **enter** Konstrukt. Das bedeutet, die Auswertung stützt sich auf **leaveP** ab, erfolgt aber davon abgesehen identisch.

Semantik von Zeit-Annotationen (abstract-time-annotation)

Bisher wurden für Zeit-Annotationen nur Selektoren definiert, die die relativen Start- und Endzeitpunkte sowie den Referenzpunkt selektieren. Das heißt, der Selektor, der den „earliest starting shift“ einer Zeit-Annotation zurückliefert, bestimmt nicht den absoluten frühesten Startzeitpunkt sondern nur die Verschiebung dieses frühesten Startzeitpunktes zum Referenzpunkt. Nachdem nun die Auswertung abstrakter Referenzzeitpunkte definiert ist, können nun Funktionen definiert werden, die die Auswertung des Referenzpunktes sowie dessen Addition zu den relativen Start- und Endpunkten einer Zeit-Annotation zusammenfassen.

Definition

Für Zeit-Annotationen sind die nachstehenden Selektoren definiert:

$$
\begin{array}{lcl}
\textbf{.est} & : & \textbf{abstract-time-annotation} \mapsto \textbf{int-bottom} \\
\textbf{.lst} & : & \textbf{abstract-time-annotation} \mapsto \textbf{int-bottom} \\
\textbf{.eft} & : & \textbf{abstract-time-annotation} \mapsto \textbf{int-bottom} \\
\textbf{.lft} & : & \textbf{abstract-time-annotation} \mapsto \textbf{int-bottom} \\
\textbf{.sti} & : & \textbf{abstract-time-annotation} \mapsto \textbf{abstract-interval} \\
\textbf{.fti} & : & \textbf{abstract-time-annotation} \mapsto \textbf{abstract-interval} \\
\textbf{.dti} & : & \textbf{abstract-time-annotation} \mapsto \textbf{abstract-interval}
\end{array}
$$

Deren Semantik wird wie folgt definiert:

$$
\begin{array}{lcl}
[\![\textbf{ata.est}]\!]_\sigma & = & \textbf{ata.ess} + [\![\textbf{ata.refPoint}]\!]_\sigma \\
[\![\textbf{ata.lst}]\!]_\sigma & = & \textbf{ata.lss} + [\![\textbf{ata.refPoint}]\!]_\sigma \\
[\![\textbf{ata.eft}]\!]_\sigma & = & \textbf{ata.efs} + [\![\textbf{ata.refPoint}]\!]_\sigma \\
[\![\textbf{ata.lft}]\!]_\sigma & = & \textbf{ata.lfs} + [\![\textbf{ata.refPoint}]\!]_\sigma \\
[\![\textbf{ata.sti}]\!]_\sigma & = & ([\![\textbf{ata.est}]\!]_\sigma \times [\![\textbf{ata.lst}]\!]_\sigma) \\
[\![\textbf{ata.fti}]\!]_\sigma & = & ([\![\textbf{ata.eft}]\!]_\sigma \times [\![\textbf{ata.lft}]\!]_\sigma) \\
[\![\textbf{ata.dti}]\!]_\sigma & = & ([\![\textbf{ata.minDu}]\!]_\sigma \times [\![\textbf{ata.maxDu}]\!]_\sigma)
\end{array}
$$

Die Semantik der oben definierten Selektoren zur Bestimmung der absoluten Zeitgrenzen einer Zeit-Annotation ist angegeben jeweils durch die Addition des zugehörigen relativen Wertes mit dem Referenzpunkt. Der Selektor **.est** beispielsweise selektiert den frühesten Startzeitpunkt (englisch **e**arliest **s**tarting **t**ime) im Gegensatz zu dem bereits bekannten Selektor **.ess**, der den relativen frühesten Startzeitpunkt selektiert. Der früheste Startzeitpunkt berechnet sich aus der durch den **.ess** Selektor bestimmten relativen Verschiebung des frühesten Startzeitpunkt gegen den Referenzpunkt sowie dem Referenzpunkt, also informell est = ess + refPoint.

Analog dazu sind die übrigen drei Selektoren benannt. Der **.lst** Selektor bestimmt den spätesten Startzeitpunkt (latest starting time), durch Addition des relativen spätesten Start-

zeitpunktes **.lss** sowie dem Referenzpunkt. Die **.eft** Funktion bestimmt den frühesten Endzeitpunkt und **.lft** den spätesten Endzeitpunkt.

Neben den vier Selektoren für die absoluten Start- und Endzeitpunkte werden zusätzlich noch drei Funktionen definiert, die die Start-, End- und Dauerintervalle selektieren. Diese Intervalle werden für die Definition der Erfüllbarkeit und Erfülltheit von Asbru-Bedingungen benötigt. Dabei selektiert **.sti** das Startintervall, also das Paar aus **.est** und **.lst**. In diesem Intervall muss ein Korridor starten, damit er Element der Zeit-Annotation sein kann. Die Funktion **.fti** liefert das Endintervall, also das aus **.eft** und **.lft** gebildete Intervall, in dem ein Korridor enden muss, um Element einer Zeit-Annotation sein zu können. Zuletzt wird noch der Selektor für das Dauer-Intervall, **.dti**, angegeben. Dieser liefert ein Paar aus minimaler und maximaler Dauer zurück und erlaubt so zu bestimmen, ob ein Korridor die richtige Länge hat um Element einer Zeit-Annotation zu sein.

Ziel dieses Kapitels ist es, die Semantik der Erfüllbarkeit und der Erfülltheit von Bedingungen anzugeben. Diese wurde zunächst informell erläutert. Danach wurde die formale Definition der semantische Auswertung der abstrakten Referenzpunkte von Zeit-Annotationen angegeben. Mit dieser Auswertung ist es möglich, die absoluten Start- und Endzeitpunkte von Zeit-Annotationen zu berechnen. Mit Hilfe dieser Funktionen die Start-, End- und Dauerintervalle einer Zeit-Annotation berechnet werden.

Damit die Erfüllbarkeit einer Bedingung nachgewiesen werden kann, muss die Existenz eines Korridors gezeigt werden, innerhalb dessen das Konditional der Bedingung wahr ist. Der Korridor selbst muss in der Menge der von der Zeit-Annotation der Bedingung erlaubten Korridore liegen. Diese Menge beschreibt alle Korridore, die im Startintervall der Zeit-Annotation beginnen, im Endintervall enden und deren Längen im Längenintervall der Zeit-Annotation liegen.

Die Zeit-Annotation des Beispiels aus 8.1.1 verlangt, dass ein Fieberrückgang vier bis sechs Stunden nach Beginn der Einnahme der fiebersenkenden Medikamente gemessen werden muss. Zulässig sind damit alle Korridore, die zwischen vier und sechs Stunden nach Beginn der Einnahme starten und enden. Diese Korridore müssen nicht mit Flanken abgeschlossen sein. Wie dieser Sachverhalt zu formalisieren ist, wird im Folgenden beschrieben.

Um festzustellen, ob eine Uhrzeit Element eines abstrakten Korridors ist, wird eine Elementbeziehung definiert.

Definition

Es wird ein Prädikat $\in$ definiert, dass eine Beziehung zwischen Zeiten und Intervallen herstellt.

$$. \in . \;:\; \mathbf{int} \times \mathbf{abstract\text{-}interval} \mapsto \mathbf{bool}$$

Die Semantik dieser Elementbeziehung ist wie folgt definiert:

$$\begin{aligned} \mathbf{AC} \in \mathbf{Ia} \quad \leftrightarrow \quad & \mathbf{Ia.start} = \bot \\ & \vee\ \mathbf{Ia.start} \leq \mathbf{AC} \\ \wedge \quad & \mathbf{Ia.end} = \bot \\ & \vee\ \mathbf{AC} \leq \mathbf{Ia.end} \end{aligned}$$

Informell kann das $\bot$-Element von abstrakten Korridoren als ein Platzhalter für plus- bzw. minus-Unendlich angesehen werden. In der üblichen Definition der Vergleichsoperatoren zwischen ganzen Zahlen und Unendlichkeitswerten gilt, dass jede Zahl immer kleiner ist als

Unendlich und immer größer ist als minus Unendlich. Im Fall der abstrakten Korridore wird angenommen, dass ein $\perp$-Element an erster Stelle (also mittels **.start** selektiert) für den Wert minus-Unendlich steht, an zweiter Stelle für den Wert Unendlich.

Bei einer Überprüfung, ob ein ganzzahliger Wert Element eines abstrakten Korridors ist, muss überprüft werden, ob der Wert größer ist als das Startelement des Korridors und kleiner als das Endelement des Korridors. Ist eines dieser Elemente der $\perp$-Wert und steht somit für die Unendlichkeit, erübrigt sich diese Überprüfung.

Mit diesen Definitionen ist es nun möglich, die eigentlich gewünschte Beziehung zwischen Korridoren und Zeit-Annotationen herzustellen. Damit soll formalisierbar werden, wann ein Korridor in der Menge der von einer Zeit-Annotation zugelassenen Korridore liegt.

Definition

Es wird eine Elementbeziehung zwischen Intervallen und abstrakten Zeit-Annotationen definiert:

$$. \in . \quad : \quad \textbf{interval} \times \textbf{abstract-time-annotation} \mapsto \textbf{bool}$$

Zwischen einem **interval I** mit $\mathbf{I} = (\mathbf{AC}_1 \times \mathbf{AC}_2)$ und einer **abstract-time-annotation ata** wird die Semantik der Elementbeziehung wie nachstehend definiert:

$$\begin{aligned} [\![\mathbf{I} \in \mathbf{ata}]\!]_\sigma \quad \leftrightarrow \quad & \mathbf{I.start} \in [\![\mathbf{ata.sti}]\!]_\sigma \\ & \wedge\, \mathbf{I.end} \in [\![\mathbf{ata.fti}]\!]_\sigma \\ & \wedge\, (\mathbf{I.end} - \mathbf{I.start}) \in [\![\mathbf{ata.dti}]\!]_\sigma \\ & \wedge\, [\![\mathbf{ata.refPoint}]\!]_\sigma \neq \perp \end{aligned}$$

Ein (nicht-abstrakter) Korridor kann Element einer Zeit-Annotation sein. Dazu ist es notwendig, dass der Startpunkt des Korridors im Startintervall der Zeit-Annotation liegt, der Endpunkt des Korridors im Endintervall der Zeit-Annotation und die Länge des Korridors muss im Längenintervall der Zeit-Annotation liegen. Diese Auswertung ist nur möglich, wenn der ausgewertete Referenzpunkt der Zeit-Annotation ungleich dem $\perp$-Element ist.

Noch einmal zurück zum Beispiel. Dessen Zeit-Annotation ist wie folgt definiert:

$$([\perp, +\text{hour}(6)][+\text{hour}(4), \perp][0, \perp], \textbf{enter}(\textbf{iname}, \texttt{activated}))$$

Es wurde bereits bestimmt, dass der Referenzpunkt zu „5“ auswertet. Geprüft werden soll, ob ein bestimmter Korridor Element dieser Zeit-Annotation ist. Der Korridor, der hier untersucht werden soll ist der, der 5 Stunden nach Planstart beginnt und 5 Stunden und 20 Minuten nach Planstart endet. Es wird also überprüft ob

$$(5 + \text{hour}(5) \times 5 + \text{hour}(5) + \text{minute}(20)) \in \textbf{ata}$$

Dazu müssen die Bedingungen der $\in$ Beziehungen einzeln geprüft werden. Zunächst muss geprüft werden, ob der Startpunkt des Intervalls im Startintervall der Zeit-Annotation liegt, also:

$$5 + \text{hour}(5) \in [\perp \; + 5, \text{hour}(6) + 4]$$

Es muss also geprüft werden, ob folgende Ungleichungen gelten:

$$\perp + 5 \leq 5 + \text{hour}(5) \text{ oder } \perp + 5 = \perp$$

und

$$5 + \text{hour}(5) \leq \text{hour}(6) + 5 \text{ oder } \text{hour}(6) + 5 = \bot$$

Der erste Teil der Gleichung kann positiv entschieden werden, da $\bot + 5 = \bot$ und damit die Disjunktion erfüllt ist. Für den zweiten Teil muss geprüft werden, ob der Wert 5 + hour(5) kleiner ist als 5 + hour(6). Dies ist der Fall, da hour eine Funktion ist, die von **int** nach **int** abbildet und deren steigende Monotonie bekannt ist.

Damit konnte nachgewiesen werden, dass der Anfang des gewählten Intervalls im Anfangsintervall der Zeit-Annotation liegt. Zu zeigen bleibt, dass das Ende des gewählten Intervalls im Endintervall der Zeit-Annotation liegt, dass die Dauer korrekt gewählt wurde und der Referenzpunkt der Zeit-Annotation nicht zum $\bot$-Element ausgewertet wird.

Geringfügig anders werden Zeit-Annotationen ausgewertet, die im Kontext zyklischer Pläne verwendet werden. Zyklische Pläne erlauben, ihren Unterplan in regelmäßig wiederkehrenden Startfenstern zu starten, bis der Unterplan ausreichend oft ausgeführt wurde. Das Startintervall wird über eine reguläre Zeit-Annotation angegeben. Zusätzlich kennt die zyklische Kontrolle aber noch einen Offset sowie eine Frequenz, die dieses Startintervall verschieben können.

Die Definition der Elementbeziehung einer Zeit und der zyklischen Kontrolle stützt sich daher auf die bereits definierten Elementbeziehungen ab, verschiebt aber das Intervall um die angegebenen Offsets. Der Einfachheit halber wird diese Verschiebung nicht auf der Zeit-Annotation durchgeführt. Stattdessen wird eine inverse Rechenoperation auf der zu testenden Zeit ausgeführt, d.h. statt auf beide Komponenten des Startintervalls der Zeit-Annotation Frequenz und Offset aufzuaddieren, werden diese von der zu testenden Zeit abgezogen.

Definition

Ein Zeitpunkt kann Element einer zyklischen Zeit-Annotation sein:

$$. \in . \ : \ \textbf{int} \times \texttt{cyclical} \mapsto \textbf{bool}$$

Ein Zeitpunkt ist Element einer zyklischen Zeit-Annotation, $\mathbf{AC} \in \texttt{cyclicalObj}$, wenn gilt:

$$\begin{aligned} &[\![\mathbf{AC} \in \texttt{cyclicalObj}]\!]_\sigma \\ \leftrightarrow\ &\exists\, \mathbf{n}.[\![\ \mathbf{AC} - (\mathbf{n} * \texttt{cyclicalObj.frequncy} \\ &\quad + \texttt{cyclicalObj.offset}) \in \texttt{cyclicalObj.ata.sti}]\!]_\sigma \end{aligned}$$

Das bedeutet, dass der Zeitpunkt, für den ermittelt werden soll, ob er Element des Startintervalls ist, durch Frequenz und Offset verschoben wird. Durch diese Verschiebung kann er entweder als zu einem der Startintervalle passend erkannt werden oder es wird festgestellt, dass er nicht in einem Startintervall liegt.

Semantik von Bedingungen (condition)

Nachdem nun formal definiert ist, wann ein Korridor Element einer Zeit-Annotation ist, können als nächstes die Prädikate **satisfied** und **satisfiable** für Bedingungen definiert werden. Diese prüfen, ob eine (Zeit-annotierte) Bedingung in einem gegebenen Zustand erfüllt ist, bzw. ob die Bedingung in der Zukunft erfüllt werden kann. Dies ist bei der Planausführung wichtig. So kann beispielsweise ein Plan nur dann die `Selection`-Phase verlassen, wenn nacheinander die **filter**- und die **setup**-Bedingung erfüllt sind. Wird die Erfüllbarkeit der

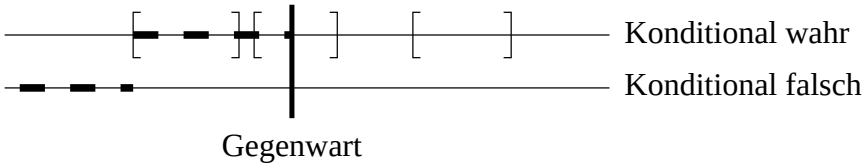


Abbildung 8.4: Darstellung der Erfüllbarkeit

filter-Bedingung im Zustand `considered` überprüft und die Bedingung ist erfüllt, wechselt der Plan in den Zustand `possible`. Ist die Bedingung nicht erfüllt aber erfüllbar so verändert der Plan seinen Zustand nicht und wartet auf eine Zustandsänderung. Ist die Bedingung nicht mehr erfüllbar, so bricht der Plan seine Auswertung ab und setzt sich in den Zustand `rejected`. Analog werden die anderen Bedingungen des Asbru-Plans an den entsprechenden Punkten ausgewertet. Dies wird detailliert in Abschnitt 3.1 erläutert.

Um die intendierte Semantik der Bedingungen korrekt abzubilden, soll eine erfüllbare Bedingung stets eine mögliche Zukunft existieren, die diese erfüllt. Diese Zukunft muss technisch möglich sein, nicht jedoch medizinisch. Beispielsweise ist bei einem Patienten, dessen Mandeln entfernt wurden offensichtlich, dass dieser keine Mandelentzündung mehr bekommen kann. Eine Bedingung, die verlangt, dass der Patient in den nächsten drei Monaten eine Mandelentzündung bekommt ist dennoch im Sinne der Erfüllbarkeit von Asbru erfüllbar. Erfüllbar ist in dieser Hinsicht ein syntaktisches Kriterium, das unabhängig von medizinischen Realitäten auswertet, ob die spezifizierten Korridore noch einhaltbar sind.

Grafisch wird die Erfüllbarkeit in Abbildung 8.4 dargestellt. Dort sind drei Intervalle angezeichnet, das erste in der Vergangenheit, das zweite teilweise in der Vergangenheit und teilweise in der Zukunft, das dritte komplett in der Zukunft. Angenommen die drei Intervalle sind jeweils unabhängig voneinander Element einer Zeit-Annotation. Für das erste Intervall gilt, dass während der gesamten Länge das Konditional der Bedingung wahr ist. Somit folgt, dass die Bedingung sowohl erfüllbar als auch erfüllt ist (sofern keine Signalflanken an den Randpunkten benötigt werden). Für das zweite Intervall gilt, dass zu allen Zeitpunkten dieses Intervalls in der Vergangenheit das Konditional wahr ist. Somit könnte dieses Intervall bei geeigneter Fortführung des Zustands die Bedingung erfüllen und es gilt in diesem Fall, dass die Bedingung erfüllbar ist. Das dritte Intervall liegt komplett in der Zukunft. Damit gibt es, bei geeigneter Fortsetzung des Zustands, eine Möglichkeit, dass während des gesamten Intervalls das Konditional wahr sein wird. Daher ist auch diese Bedingung erfüllbar aber nicht erfüllt.

Zunächst soll die Definition der Syntax der Prädikate sowie die Semantik der einfachen Basisfälle angegeben werden.

Definition

Die Syntax der Prädikate **satisfied** und **satisfiable** wird wie folgt definiert:

$$\begin{array}{lcl} \textbf{satisfied} & : & \textbf{condition} \times \textbf{state} \mapsto \textbf{bool} \\ \textbf{satisfiable} & : & \textbf{condition} \times \textbf{state} \mapsto \textbf{bool} \end{array}$$

$$\begin{array}{lcl} \textbf{satisfied}(\text{mk-cond}(\textbf{cond}), \sigma) & \leftrightarrow & [\![\textbf{cond}]\!]_\sigma \\ \textbf{satisfiable}(\text{mk-cond}(\textbf{cond}), \sigma) & \leftrightarrow & \textbf{satisfied}(\text{mk-cond}(\textbf{cond}), \sigma) \end{array}$$

Die oben aufgeführten einfachen Basisfälle zur Definition der Semantik beschreiben Bedingungen, die nicht Zeit-annotiert sind, also ausschließlich Konditionale beinhalten. In diesen Fällen wird die Auswertung des **satisfied** Prädikates auf die Evaluierung des Konditionals zurückgeführt. Solche Bedingungen sind genau dann erfüllbar, wenn sie erfüllt sind und entsprechend wird das Prädikat **satisfiable** definiert.

Komplexer ist die Definition der Erfüllbarkeit Zeit-annotierter Bedingungen.

Definition

$$
\begin{aligned}
&\textbf{satisfied}(\text{mk-cond}(\textbf{cond}, \textbf{ata}), \sigma) \\
\leftrightarrow\ &\exists\ \mathbf{I}.\quad [\![\mathbf{I} \in \mathbf{ata}]\!]_\sigma \\
&\quad \wedge\ \forall\ \mathbf{AC}_1.\quad \mathbf{AC}_1 \in \mathbf{I} \\
&\qquad\qquad \rightarrow \textbf{satisfied}(\text{mk-cond}(\textbf{cond}), (\textbf{ASH}, \textbf{HS}, \textbf{PDH}, \textbf{EAGG}, \textbf{Tick}, \mathbf{AC}_1)) \\
&\quad \wedge\quad [\![\mathbf{ata.ess}]\!]_\sigma \neq \bot \vee [\![\mathbf{ata.maxDu}]\!]_\sigma \neq \bot \\
&\qquad \rightarrow \neg\ \textbf{satisfied}(\text{mk-cond}(\textbf{cond}), (\textbf{ASH}, \textbf{HS}, \textbf{PDH}, \textbf{EAGG}, \textbf{Tick}, \mathbf{I.start} - 1)) \\
&\quad \wedge\quad [\![\mathbf{ata.lfs}]\!]_\sigma \neq \bot \vee [\![\mathbf{ata.maxDu}]\!]_\sigma \neq \bot \\
&\qquad \rightarrow\quad \neg\ \textbf{satisfied}(\text{mk-cond}(\textbf{cond}), (\textbf{ASH}, \textbf{HS}, \textbf{PDH}, \textbf{EAGG}, \textbf{Tick}, \mathbf{I.end} + 1)) \\
&\qquad\ \wedge\ \mathbf{AC} \geq \mathbf{I.end} + 1 \\
&\quad \wedge\ \mathbf{AC} \geq \mathbf{I.end}
\end{aligned}
$$

Erfüllt ist eine Zeit-annotierte Bedingung, wenn ein Korridor existiert, der Element der Zeit-Annotation der Bedingung ist:

$$\exists\ \mathbf{I}.[\![\mathbf{I} \in \mathbf{ata}]\!]_\sigma$$

Für diesen Korridor gilt, dass zu allen Zeitpunkten des Korridors das Konditional erfüllt ist:

$$
\begin{aligned}
&\exists\ \mathbf{I}.\quad [\![\mathbf{I} \in \mathbf{ata}]\!]_\sigma \\
&\quad \wedge\ \forall\ \mathbf{AC}_1.\quad \mathbf{AC}_1 \in \mathbf{I} \\
&\qquad\qquad \rightarrow \textbf{satisfied}(\text{mk-cond}(\textbf{cond}), (\textbf{ASH}, \textbf{HS}, \textbf{PDH}, \textbf{EAGG}, \textbf{Tick}, \mathbf{AC}_1))
\end{aligned}
$$

Wenn die Zeit-Annotation Flanken verlangt, zum Beispiel, weil ein frühester Startzeitpunkt oder eine Maximaldauer spezifiziert sind, dann muss der Korridor entsprechend von negativen Konditionalauswertungen begrenzt sein.

$$
\begin{aligned}
&[\![\mathbf{ata.ess}]\!]_\sigma \neq \bot \vee [\![\mathbf{ata.maxDu}]\!]_\sigma \neq \bot \\
&\rightarrow \neg\ \textbf{satisfied}(\text{mk-cond}(\textbf{cond}), (\textbf{ASH}, \textbf{HS}, \textbf{PDH}, \textbf{EAGG}, \textbf{Tick}, \mathbf{I.start} - 1))
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathbf{ata.lfs}]\!]_\sigma \neq \bot \vee [\![\mathbf{ata.maxDu}]\!]_\sigma \neq \bot \\
&\rightarrow \neg\ \textbf{satisfied}(\text{mk-cond}(\textbf{cond}), (\textbf{ASH}, \textbf{HS}, \textbf{PDH}, \textbf{EAGG}, \textbf{Tick}, \mathbf{I.end} + 1))
\end{aligned}
$$

Im Fall negativer Flanken am Ende des Korridors muss verhindert werden, dass eine Auswertung in der Zukunft erfolgt, also ein Zustand abgefragt wird, der noch nicht eingetreten ist:

$$
\begin{aligned}
&[\![\mathbf{ata.lfs}]\!]_\sigma \neq \bot \vee [\![\mathbf{ata.maxDu}]\!]_\sigma \neq \bot \\
&\rightarrow\quad \neg\ \textbf{satisfied}(\text{mk-cond}(\textbf{cond}), (\textbf{ASH}, \textbf{HS}, \textbf{PDH}, \textbf{EAGG}, \textbf{Tick}, \mathbf{I.end} + 1)) \\
&\quad \wedge\ \mathbf{AC} \geq \mathbf{I.end} + 1
\end{aligned}
$$

Außerdem muss der Korridor komplett in der Vergangenheit liegen.

$$\mathbf{AC} \geq \mathbf{I.end}$$

Definition

$$
\begin{aligned}
&\mathbf{satisfiable}(\text{mk-cond}(\mathbf{cond}, \mathbf{ata}), \sigma) \\
\leftrightarrow\ &\exists\, \mathbf{I}.\quad [\![\mathbf{I} \in \mathbf{ata}]\!]_\sigma \\
&\quad \wedge \forall\, \mathbf{AC}_1.\quad \mathbf{AC}_1 \in \mathbf{I} \\
&\qquad\qquad \rightarrow\quad \mathbf{satisfied}(\text{mk-cond}(\mathbf{cond}), (\mathbf{ASH}, \mathbf{HS}, \mathbf{PDH}, \mathbf{EAGG}, \mathbf{Tick}, \mathbf{AC}_1)) \\
&\qquad\qquad\quad \vee\ \mathbf{AC}_1 > \mathbf{AC} \\
&\quad \wedge\quad [\![\text{TA.ess}]\!]_\sigma \neq \bot \vee [\![\text{TA.maxDu}]\!]_\sigma \neq \bot \\
&\qquad \rightarrow\quad \neg\ \mathbf{satisfied}(\text{mk-cond}(\mathbf{cond}), (\mathbf{ASH}, \mathbf{HS}, \mathbf{PDH}, \mathbf{EAGG}, \mathbf{Tick}, \mathbf{I.start} - 1)) \\
&\qquad\quad \vee\ \mathbf{I.start} - 1 > \mathbf{AC} \\
&\quad \wedge\quad [\![\text{TA.lfs}]\!]_\sigma \neq \bot \vee [\![\text{TA.maxDu}]\!]_\sigma \neq \bot \\
&\qquad \rightarrow\quad \neg\ \mathbf{satisfied}(\text{mk-cond}(\mathbf{cond}), (\mathbf{ASH}, \mathbf{HS}, \mathbf{PDH}, \mathbf{EAGG}, \mathbf{Tick}, \mathbf{I.end} + 1)) \\
&\qquad\quad \vee\ \mathbf{I.end} + 1 > \mathbf{AC} \\
\vee\ &[\![\mathbf{ata.refPoint}]\!]_\sigma = \bot
\end{aligned}
$$

Die Erfüllbarkeit wird ähnlich definiert wie die Erfülltheit. Ziel der Erfüllbarkeit ist es, festzustellen, ob eine Bedingung in der Zukunft erfüllt werden kann, oder es bereits ist. Das bedeutet, dass eine erfüllte Bedingung automatisch erfüllbar ist. Spezifiziert ist die Erfüllbarkeit analog zur Erfülltheit, jedoch mit einem wichtigen Unterschied. Bei der Erfülltheit wird verlangt, dass ein Korridor gefunden wird, der komplett in der Vergangenheit liegt, das Konditional während diesem wahr ist und Flanken vorliegen. Bei der Erfüllbarkeit werden Bedingungen an das Konditional lediglich für die Zeitpunkte angegeben, die in der Vergangenheit liegen. Für Zeitpunkte des gewählten Korridors, die in der Zukunft liegen, wird nichts gefordert.

Eine Bedingung ist immer erfüllbar, wenn der Referenzpunkt der Zeit-Annotation zum $\bot$-Element auswertet.

Aufbauend auf diesen Definitionen wird die Semantik der Verknüpfung von Bedingungen rekursiv definiert:

Definition

Die Semantik der Auswertung logisch verknüpfter Bedingungen ist wie folgt definiert:

$$
\begin{aligned}
\mathbf{satisfied}(\mathbf{cond}_1 \ \texttt{\_and}\ \mathbf{cond}_2, \sigma) &\leftrightarrow \mathbf{satisfied}(\mathbf{cond}_1, \sigma) \wedge \mathbf{satisfied}(\mathbf{cond}_2, \sigma) \\
\mathbf{satisfiable}(\mathbf{cond}_1 \ \texttt{\_and}\ \mathbf{cond}_2, \sigma) &\leftrightarrow \mathbf{satisfiable}(\mathbf{cond}_1, \sigma) \wedge \mathbf{satisfiable}(\mathbf{cond}_2, \sigma) \\
\mathbf{satisfied}(\mathbf{cond}_1 \ \texttt{\_or}\ \mathbf{cond}_2, \sigma) &\leftrightarrow \mathbf{satisfied}(\mathbf{cond}_1, \sigma) \vee \mathbf{satisfied}(\mathbf{cond}_2, \sigma) \\
\mathbf{satisfiable}(\mathbf{cond}_1 \ \texttt{\_or}\ \mathbf{cond}_2, \sigma) &\leftrightarrow \mathbf{satisfiable}(\mathbf{cond}_1, \sigma) \vee \mathbf{satisfiable}(\mathbf{cond}_2, \sigma)
\end{aligned}
$$

Die Erfüllbarkeit und Erfülltheit der mit logischen Junktoren verknüpften Bedingungen wird rekursiv definiert. Zwei Bedingungen können mittels der Junktoren `_and` bzw. `_or` verknüpft werden. Zwei Bedingungen die `_and` verknüpft sind, sind also genau dann erfüllt,

wenn die beiden Einzelbedingungen erfüllt sind, analoges gilt für die Erfüllbarkeit und den `_or` Operator.

Semantik von Asbru-Bedingungen

Asbru-Bedingungen erlauben die Einbeziehung der Umgebung in die Auswertung von Bedingungen. Die Umgebung, sprich, das medizinische Personal, hat die Möglichkeit, die Auswertung von Asbru-Bedingungen so zu beeinflussen, das für erfüllte Bedingungen das Prädikat **satisfied** nicht zu wahr auswertet. Auch gibt es die Möglichkeit, den umgekehrten Schritt zu gehen und für eine nicht erfüllte Bedingung dennoch das Prädikat **satisfied** zu wahr auszuwerten.

Definition

Für Asbru-Bedingungen werden Prädikate **satisfied** und **satisfiable** definiert:

$$\begin{array}{lcl} \textbf{satisfied} & : & \textbf{asbru-condition} \times \textbf{state} \times \textbf{bool} \times \textbf{bool} \mapsto \textbf{bool} \\ \textbf{satisfiable} & : & \textbf{asbru-condition} \times \textbf{state} \mapsto \textbf{bool} \end{array}$$

Deren Semantik ist wie folgt spezifiziert:

$$\begin{array}{ll} & \textbf{satisfied}(\textbf{acond}, \sigma, \textbf{bv}_1, \textbf{bv}_2) \\ \leftrightarrow & \quad \textbf{satisfied}(\textbf{acond.condition}, \sigma) \\ & \quad \wedge \neg\, \textbf{acond.confirm} \vee \textbf{bv}_1 \\ & \vee\, \textbf{bv}_2 \wedge \textbf{acond.override} \\ & \textbf{satisfiable}(\textbf{acond}, \sigma) \\ \leftrightarrow & \quad \textbf{satisfiable}(\textbf{acond.condition}, \sigma) \\ & \vee\, \textbf{acond.override} \end{array}$$

Die Erfüllbarkeit und Erfülltheit von Asbru-Bedingungen wird mit den Prädikaten **satisfied** und **satisfiable** definiert. Für diese Auswertung müssen die Umgebungssignale, die bereits im Zustand gespeichert sind, noch einmal getrennt übergeben werden. Bei der Auswertung einer Asbru-Bedingung ist weder der Planinstanzname des zugehörigen Plans noch der aktuelle Signal-Zustand bekannt. Das bedeutet, dass es bei der Auswertung nicht möglich ist, die entsprechenden Signale aus der **environment-aggregation** Datenstruktur zu selektieren. Aus diesem Grund werden die entsprechenden Signale gesondert übergeben. Ist eine Bedingung sowohl übersteuerbar als auch verzögerbar und werden durch die Umgebung beide Signale gesetzt, so erhält das Signal zur Übersteuerung höhere Priorität.

8.2 Auswertung von Wartebedingungen

Die Behandlung von Brustkrebs im DCIS Stadium besteht, wie in Abschnitt 3.1 beschrieben, darin, nach einer anfänglichen Information für den Patienten entweder eine Brusterhaltende Therapie mit einer Untersuchung der Achselknoten durchzuführen oder eine Brustentfernung. Der Plan, der die Ausführung dieser Behandlungen steuert, kann also erst abgeschlossen werden, wenn die Brustentfernung oder die brusterhaltende Operation in Verbindung mit der Untersuchung der Achselknoten durchgeführt wurden. Diese Forderung an den Zustand der Unterpläne des Behandlungsplans wird in einem **wait-for** Konstrukt gespeichert.

Die Wartebedingung **wait-for** eines Planes definiert notwendige Kriterien für die erfolgreiche Beendigung des Planes. Diese Kriterien richten sich an das Terminierungsverhalten der

jeweiligen Unterpläne des Plans. Dabei beschreibt ein **wait-for** Konstrukt, welche Teilmenge der Unterpläne eines Plans erfolgreich abgeschlossen sein muss, damit sich der Plan beenden kann.

Die Semantik der Wartebedingung wird wie die der Asbru-Bedingungen durch Prädikate **satisfied** und **satisfiable** definiert. Dabei überprüft **satisfiable**, ob der gegenwärtige Zustand theoretisch erlaubt, dass das **wait-for** Konstrukt in der Zukunft erfüllt werden kann. **satisfied** überprüft, ob alle relevanten Pläne erfolgreich beendet wurden. Die Überprüfungsmöglichkeit **satisfiable** wird verwendet, damit ein Plan nicht unendlich lange auf die Erfüllung seiner Wartebedingung wartet, obwohl bereits klar ist, dass diese nicht mehr erfüllbar ist.

Die Definition des syntaktischen **wait-for** Konstrukts erfolgt statisch in der Planhierarchie. Damit ist bei der Spezifikation des **wait-for** unbekannt, welchen Instanznamen der wartende Plan sowie dessen Unterpläne haben werden. Bei der Auswertung eines **wait-for** muss somit der Instanzname des wartenden Plans als Argument mit übergeben werden, damit die Instanznamen der Unterpläne berechnet werden können.

Definition

Die Prädikate **satisfied** und **satisfiable** bilden Wartekonstrukte und Zustandsinformationen auf Wahrheitswerte ab:

$$\begin{array}{lcl} \textbf{satisfied} & : & \textbf{wait-for} \times \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\ \textbf{satisfiable} & : & \textbf{wait-for} \times \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \end{array}$$

Ein **wait-for** darf nur Pläne enthalten, die direkte Unterpläne des Planes sind, in dem das **wait-for** enthalten ist. Um zu bewerten, ob ein **wait-for** erfüllbar ist, ist zu berücksichtigen, dass Unterpläne von Plänen mit gesetztem retry-aborted flag möglicherweise neu gestartet werden, falls der Plan in den Zustand `aborted` kommt.

Definition

Die Semantik der Basisfälle der Auswertung von Wartekonstrukten ist wie folgt definiert:

$$\begin{array}{l} \textbf{satisfied}(\textbf{bwf}(\textbf{SK}), \textbf{HS}, \textbf{iname}) \\ \leftrightarrow \textbf{isCompleted}(\textbf{HS}, \textbf{nameGen}(\textbf{SK}, \textbf{iname})) \end{array}$$

$$\begin{array}{l} \textbf{satisfiable}(\textbf{bwf}(\textbf{SK}), \textbf{HS}, \textbf{iname}) \\ \leftrightarrow \quad \neg\, \textbf{isRejected}(\textbf{HS}, \textbf{nameGen}(\textbf{SK}, \textbf{iname})) \\ \quad \wedge\, (\textbf{isAborted}(\textbf{HS}, \textbf{nameGen}(\textbf{SK}, \textbf{iname})) \rightarrow \texttt{asbru}(\textbf{nameSel}(\textbf{iname})).\textbf{retry-aborted}) \end{array}$$

$$\begin{array}{l} \textbf{satisfied}(\textbf{wait-for-none}, \textbf{HS}, \textbf{iname}) \\ \leftrightarrow \textbf{true} \end{array}$$

$$\begin{array}{l} \textbf{satisfiable}(\textbf{wait-for-none}, \textbf{HS}, \textbf{iname}) \\ \leftrightarrow \textbf{true} \end{array}$$

Basis der Auswertung allgemeiner **wait-for** stellt die Auswertung der Basiskonstrukte **bwf** bzw. **wait-for-none** dar, auf die alle anderen **wait-for** zurückgeführt werden können. Ein **bwf** Konstrukt ist erfüllt, wenn der darin angegebene Plan erfolgreich abgeschlossen wurde,

also im Zustand `completed` ist. Erfüllbar ist das Konstrukt, wenn der Plan nicht zurückgewiesen wurde und nicht abgebrochen wurde oder zwar abgebrochen wurde, aber der Plan neu gestartet werden kann. Zurückgewiesene Pläne sind im Zustand `rejected`, abgebrochene im Zustand `aborted`. Die Möglichkeit zum Neustart wird mit Hilfe der Plandefinition des angegebenen übergeordneten Plans überprüft.

Definition

Die Semantik der Auswertung der **wait-for-n** Wartekonstrukte ist wie folgt definiert:

$$\begin{aligned}
&\textbf{satisfied}(\textbf{wait-for-n}(\mathbf{n}+1, []), \mathbf{HS}, \mathbf{iname}) \\
\leftrightarrow\ &\textbf{false}
\end{aligned}$$

$$\begin{aligned}
&\textbf{satisfiable}(\textbf{wait-for-n}(\mathbf{n}+1, []), \mathbf{HS}, \mathbf{iname}) \\
\leftrightarrow\ &\textbf{false}
\end{aligned}$$

$$\begin{aligned}
&\textbf{satisfied}(\textbf{wait-for-n}(0, \mathbf{PL}), \mathbf{HS}, \mathbf{iname}) \\
\leftrightarrow\ &\textbf{satisfied}(\textbf{wait-for-none}, \mathbf{HS})
\end{aligned}$$

$$\begin{aligned}
&\textbf{satisfiable}(\textbf{wait-for-n}(0, \mathbf{PL}), \mathbf{HS}, \mathbf{iname}) \\
\leftrightarrow\ &\textbf{satisfiable}(\textbf{wait-for-none}, \mathbf{HS}, \mathbf{iname})
\end{aligned}$$

$$\begin{aligned}
&\textbf{satisfied}(\textbf{wait-for-n}(\mathbf{n}+1, \mathbf{SK}+\mathbf{PL}), \mathbf{HS}, \mathbf{iname}) \\
\leftrightarrow\ &\quad \textbf{satisfied}(\textbf{bwf}(\mathbf{SK}), \mathbf{HS}, \mathbf{iname}) \\
&\quad \wedge\ \textbf{satisfied}(\textbf{wait-for-n}(\mathbf{n}, \mathbf{PL}), \mathbf{HS}, \mathbf{iname}) \\
&\vee\ \neg\ \textbf{satisfied}(\textbf{bwf}(\mathbf{SK}), \mathbf{HS}, \mathbf{iname}) \\
&\quad \wedge\ \textbf{satisfied}(\textbf{wait-for-n}(\mathbf{n}+1, \mathbf{PL}), \mathbf{HS}, \mathbf{iname})
\end{aligned}$$

$$\begin{aligned}
&\textbf{satisfiable}(\textbf{wait-for-n}(\mathbf{n}+1, \mathbf{SK}+\mathbf{PL}), \mathbf{HS}, \mathbf{iname}) \\
\leftrightarrow\ &\quad \textbf{satisfiable}(\textbf{bwf}(\mathbf{SK}), \mathbf{HS}, \mathbf{iname}) \\
&\quad \wedge\ \textbf{satisfiable}(\textbf{wait-for-n}(\mathbf{n}, \mathbf{PL}), \mathbf{HS}, \mathbf{iname}) \\
&\vee\ \neg\ \textbf{satisfiable}(\textbf{bwf}(\mathbf{SK}), \mathbf{HS}, \mathbf{iname}) \\
&\quad \wedge\ \textbf{satisfiable}(\textbf{wait-for-n}(\mathbf{n}+1, \mathbf{PL}), \mathbf{HS}, \mathbf{iname})
\end{aligned}$$

Komplexere Wartekonstrukte werden auf die Auswertungen der Basiskonstrukte zurückgeführt. Das **wait-for-n** Konstrukt enthält eine Planliste, die untersucht wird sowie eine Zahl, die die Menge der zu erfüllenden Pläne angibt. Beispielsweise beschreibt **wait-for-n**(1, `chemotherapy + hormone therapy '`) ein Wartekonstrukt, bei dem lediglich einer der beiden Pläne `chemotherapy` und `hormone therapy` erfolgreich abgeschlossen sein muss, damit die Wartebedingung erfüllt ist. Erfüllbarkeit und Erfülltheit von **wait-for-n** wird rekursiv berechnet. Dazu wird der erste Plan der Planliste, die im **wait-for-n** Konstrukt enthalten ist gemäß den Regeln des **bwf** Konstrukts auf Erfülltheit bzw. Erfüllbarkeit überprüft. Im Beispiel wird also zunächst die Erfüllbarkeit von **bwf**(`chemotherapy`) überprüft. Entsprechend dem Ergebnis dieser Prüfung wird das **wait-for-n** Konstrukt rekursiv mit der Restliste aufgerufen. Die Zahl der notwendigen Pläne wird unverändert übernommen, wenn der gerade betrachtete Plan nicht erfüllbar bzw. erfüllt war, ansonsten wird die Zahl um eins veringert. Im Beispiel ist der rekursive Aufruf **wait-for-n**(1, `hormone therapy '`), wenn die Wartebedingung **bwf**(`chemotherapy`) nicht erfüllbar ist, also der Plan abgebrochen oder zurückgewiesen wurde. Andernfalls ist der rekursive Aufruf **wait-for-n**(0, `hormone therapy '`). Ein analoges Beispiel kann für die Erfülltheit konstruiert werden.

Gilt für ein **wait-for-n**, dass die Planliste leer ist, jedoch noch auf mindestens einen Plan gewartet werden muss, so werten Erfüllbarkeit und Erfülltheit beide zu **false** aus. Ist dagegen

die Zahl der Pläne, auf die gewartet werden soll 0, so sind Erfüllbarkeit und Erfülltheit beide **true**.

Definition

Die logische Verknüpfung zweier Wartekonstrukte mittels _and bzw. _or ist wie folgt definiert:

$$\begin{aligned}
&\textbf{satisfied}(\textbf{wf}_1 \texttt{ _and } \textbf{wf}_2, \textbf{HS}, \textbf{iname}) \\
\leftrightarrow\ &\textbf{satisfied}(\textbf{wf}_1, \textbf{HS}, \textbf{iname}) \\
&\wedge \textbf{satisfied}(\textbf{wf}_2, \textbf{HS}, \textbf{iname})
\end{aligned}$$

$$\begin{aligned}
&\textbf{satisfiable}(\textbf{wf}_1 \texttt{ _and } \textbf{wf}_2, \textbf{HS}, \textbf{iname}) \\
\leftrightarrow\ &\textbf{satisfiable}(\textbf{wf}_1, \textbf{HS}, \textbf{iname}) \\
&\wedge \textbf{satisfiable}(\textbf{wf}_2, \textbf{HS}, \textbf{iname})
\end{aligned}$$

$$\begin{aligned}
&\textbf{satisfied}(\textbf{wf}_1 \texttt{ _or } \textbf{wf}_2, \textbf{HS}, \textbf{iname}) \\
\leftrightarrow\ &\textbf{satisfied}(\textbf{wf}_1, \textbf{HS}, \textbf{iname}) \\
&\vee \textbf{satisfied}(\textbf{wf}_2, \textbf{HS}, \textbf{iname})
\end{aligned}$$

$$\begin{aligned}
&\textbf{satisfiable}(\textbf{wf}_1 \texttt{ _or } \textbf{wf}_2, \textbf{HS}, \textbf{iname}) \\
\leftrightarrow\ &\textbf{satisfiable}(\textbf{wf}_1, \textbf{HS}, \textbf{iname}) \\
&\vee \textbf{satisfiable}(\textbf{wf}_2, \textbf{HS}, \textbf{iname})
\end{aligned}$$

Weiterhin wird in diesem Zusammenhang die Semantik von **satisfied** und **satisfiable** im Kontext der verknüpften **wait-for** angegeben. Diese bestehen aus jeweils zwei mit _and oder _or verknüpften **wait-for**. Zwei mit _and verknüpfte **wait-for** sind erfüllt bzw. erfüllbar, wenn die jeweils einzelnen **wait-for** erfüllt bzw. erfüllbar sind. Gleiches gilt für die Verknüpfung mit _or, bei der dann jeweils mindestens eins der Konstrukte erfüllbar bzw. erfüllt sein müssen.

Definition

Die Auswertung negierter Wartebedingungen wird wie folgt spezifiziert:

$$\begin{aligned}
&\textbf{satisfied}(\texttt{_not } \textbf{wf}_1, \textbf{HS}, \textbf{iname}) \\
\leftrightarrow\ &\neg\ \textbf{satisfied}(\textbf{wf}_1, \textbf{HS}, \textbf{iname})
\end{aligned}$$

$$\begin{aligned}
&\textbf{satisfiable}(\texttt{_not } \textbf{wf}_1, \textbf{HS}, \textbf{iname}) \\
\leftrightarrow\ &\neg\ \textbf{satisfied}(\textbf{wf}_1, \textbf{HS}, \textbf{iname})
\end{aligned}$$

Es existiert ein negiertes **wait-for**. Ein solches negiertes **wait-for** ist erfüllt, solange das nicht negierte **wait-for** nicht erfüllt ist. Gleiches gilt für die Erfüllbarkeit, bei der das negierte **wait-for** so lange erfüllbar ist, solange das nicht negierte **wait-for** nicht erfüllt ist.

Definition

Die Auswertung zweier mittels exklusivem Oder verknüpfter Wartebedingungen ist wie folgt definiert:

$$\begin{aligned}
&\textbf{satisfied}(\mathbf{wf}_1 \texttt{_xor} \mathbf{wf}_2, \mathbf{HS}, \mathbf{iname}) \\
\leftrightarrow\ &\textbf{satisfied}(\texttt{_not}(\mathbf{wf}_1 \texttt{_and} \mathbf{wf}_2), \mathbf{HS}, \mathbf{iname}) \\
\wedge\ &\textbf{satisfied}(\mathbf{wf}_1 \texttt{_or} \mathbf{wf}_2, \mathbf{HS}, \mathbf{iname})
\end{aligned}$$

$$\begin{aligned}
&\textbf{satisfiable}(\mathbf{wf}_1 \texttt{_xor} \mathbf{wf}_2, \mathbf{HS}, \mathbf{iname}) \\
\leftrightarrow\ &\textbf{satisfiable}(\texttt{_not}(\mathbf{wf}_1 \texttt{_and} \mathbf{wf}_2), \mathbf{HS}, \mathbf{iname}) \\
\wedge\ &\textbf{satisfiable}(\mathbf{wf}_1 \texttt{_or} \mathbf{wf}_2, \mathbf{HS}, \mathbf{iname})
\end{aligned}$$

Werden zwei **wait-for** mittels des `_xor` Operators verknüpft, so wird diese Verknüpfung analog zu dem logischen xor behandelt. Das heißt, dass zwei xor verknüpfte **wait-for** erfüllt sind, wenn genau eines der **wait-for** erfüllt ist. Für die Erfüllbarkeit gilt analog, dass zwei xor verknüpfte **wait-for** erfüllbar sind, wenn genau eines der **wait-for** erfüllbar ist. Für die Formalisierung des Begriffs „genau eines“ werden die **wait-for** Verknüpfungen `_and`, `_or` und `_not` verwendet.

9 Semantik des Asbru-Zustandsübergangssystems

In Kapitel 7 wurde der Zustand von Asbru-Systemen definiert, im darauf folgenden Kapitel 8 wurde festgelegt, wie die Erfüllbarkeit und Erfülltheit von Asbru-Bedingungen ausgewertet werden kann. Mit diesen Vorarbeiten ist es möglich, Zustandsübergänge von Asbru-Systemen formal zu beschreiben. Dazu wird in diesem Kapitel zunächst in Abschnitt 9.1 die Syntax der Asbru-Übergangssysteme beschrieben. In Abschnitt 9.2 werden die Zustandsübergänge einzelner Asbru-Pläne definiert. Darauf folgt in Abschnitt 9.3 die Beschreibung, wie diese Zustandsübergänge einzelner Pläne zusammengefügt werden können, so dass Zustandsübergänge des Gesamtsystems entstehen. Weiterhin zum Verständnis des Kapitels relevant sind die in Anhang C angegebenen Definitionen der Prädikate, die zur Definition der Zustandsübergangsregeln verwendet werden.

9.1 Syntax des Asbru-Zustandsübergangssystems (asbru-system)

Definition

Ein Asbru-System besteht aus dem leeren Programm oder einer Menge von parallel laufenden Asbru-Plänen. Syntaktisch werden Asbru System mittels folgender Grammatik erzeugt:

asbru-system	→	**asbru-system** $\|\|_s$ **asbru-system**,
	|	**instance-name**
	|	ε

Die erste Zeile der Definition beschreibt den Fall, in dem ein Asbru-System aus mehr als Asbru-Plan besteht. In diesem Fall werden die einzelnen Komponenten durch den $\|_s$ Operator verknüpft. Diese Zeile kann rekursiv angewendet werden, so dass beliebig große Systeme erzeugt werden können. Die zweite und dritte Zeile beschreibt jeweils einen Basisfall. Zeile zwei den, in dem ein Asbru-System aus genau einem Asbru-Plan besteht, der durch seinen Instanznamen identifiziert wird. Zeile drei beschreibt den Terminierungsfall. Das ε Konstrukt steht für einen terminierten Plan, der keine Handlungen mehr ausführen kann. Im Folgenden werden daher für den Fall, dass das System aus dem ε Konstrukt besteht, keine Zustandsübergangsregeln angegeben.

9.2 Zustandsübergänge für einzelne Pläne

In diesem Kapitel wird die Semantik der Ausführung von Asbru-Plänen mittels Regeln in der SOS-Form (Structured Operational Semantics) angegeben. Die Form der SOS-Regeln, wie sie in diesem Abschnitt verwendet wird, ist in [5] definiert. Um Lesbarkeit und Verständnis zu fördern, werden die Zustandsübergänge der Pläne zunächst durch Statecharts grob

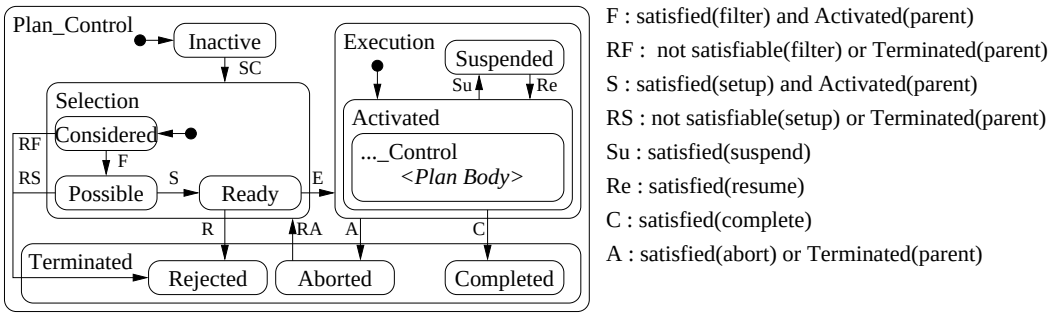


Abbildung 9.1: allgemeines Ausführungsmodell Asbru

skizziert, bevor die formale Definition der Übergänge durch SOS-Regeln definiert wird. Zu beachten ist dabei, dass einzelne Aspekte der Semantik durch die Statecharts möglicherweise nicht ausreichend genau beschrieben werden. Im Zweifel sind daher immer die SOS-Regeln maßgeblich.

9.2.1 Informelle Beschreibung der Ausführung von Asbru

In der Abbildung 9.1 wird der allgemeine Ablauf der Ausführung von Asbru-Plänen dargestellt, der allen Asbru-Plänen gemein ist. Prinzipiell beginnt die Ausführung immer im Zustand `inactive` und wechselt nach dem Start des Plans in die `Selection`-Phase. In dieser wird die Anwendbarkeit des Planes überprüft, beispielsweise, ob der Plan für ein konkretes Krankheitsstadium, eine bestimmte Altersgruppe oder bei bestimmten Vorerkrankungen geeignet ist. Zu diesem Zweck wird die Gültigkeit zweier Asbru-Bedingungen mit den Prädikaten **satisfied** und **satisfiable** überprüft. Diese Asbru-Bedingungen sind die **filter**- bzw. die **setup**-Bedingung. Technisch unterscheidet sich die Auswertung beider Bedingungen nicht. Das bedeutet, es wird in beiden Fällen überprüft, ob die Bedingung erfüllt, nicht erfüllt aber erfüllbar oder nicht mehr erfüllbar ist. Bei der Auswertung werden in der Asbru-Bedingung angegebene Zeitaspekte und von der Umgebung gesendete Signale berücksichtigt. Details dieser Auswertung sind in Abschnitt 8 beschrieben.

Von den Entwicklern von Asbru ist ein konzeptueller Unterschied beider Bedingungen vorgesehen. So soll die **filter**-Bedingung ausschließlich dazu verwendet werden, statische, also unveränderliche Bedingungen zu prüfen, wie etwa, ob der Patient das richtige Geschlecht oder die richtige Blutgruppe hat. Bedingungen, die über die Zeit variieren, etwa Herzfrequenz, Körpertemperatur oder Blutzuckerwerte sollen hingegen mit der **setup**-Bedingung geprüft werden. In der Praxis wird diese Trennung nicht immer beachtet. Aus theoretischer Sicht ist diese Unterscheidung nicht relevant, da sie keine Auswirkung auf die Auswertung hat.

Tritt ein Plan in die `Selection`-Phase ein, so wird er in den Zustand `considered` versetzt. In jedem Zustand der `Selection`-Phase gibt es drei Transitionen. Eine, die den Plan in den Zustand `rejected` versetzt, die zweite, die den Plan in den „nächsten“ Zustand fortschreiten lässt und eine dritte, die den Zustand des Plans unverändert lässt. Diese drei Transitionen sind jeweils mit Vorbedingungen versehen, die wechselseitig ausschließend sind. Dabei sind die Transitionen, die zu einer Zurückweisung des Plans führen in der Hinsicht am höchsten priorisiert, dass deren Vorbedingung am weitesten gefasst ist. In Abbildung 9.1 sind die Transitionen, die zu einer Zurückweisung führen, mit **R**, **RF** bzw. **RS** beschriftet. Die Vorbedingungen der Transitionen, die ein Fortschreiten in den nächsten Zustand erlauben, sind am zweithöchsten priorisiert, wieder in dem Sinne, dass die Vorbedingungen weit gefasst sind. Zuletzt folgen die Transitionen, die den Zustand unverändert lassen.

Dies bedeutet beispielsweise im Zustand `considered`, dass die **filter**-Bedingung sowie der Zustand des übergeordneten Plans überprüft wird und abhängig vom Ergebnis der Plan zurückgewiesen wird, in den Zustand `possible` wechselt oder im Zustand `considered` bleibt. Verlassen wird die `Selection`-Phase zugunsten der `Execution`-Phase, wenn der Plan im Zustand `ready` fortschreiten soll. Dies geschieht, wenn der Plan von seinem übergeordneten Plan durch das Aktivierungssignal `E` aktiviert wurde. Ist dies geschehen, wechselt der Plan in den Zustand `activated`. In diesem Zustand startet und aktiviert der Plan abhängig von seinem Plantyp seine Unterpläne.

Wie auch in der `Selection`-Phase gibt es in der `Execution`-Phase Transitionen unterschiedlicher Priorität. Am höchsten priorisiert ist eine Transition, die für einen Abbruch der Auswertung de Plans sorgt. Nur wenn die Vorbedingung dieser Transition nicht erfüllt ist, wird als nächstes geprüft, ob der Plan beendet werden kann bzw. ob die reguläre Planausführung ausgeführt werden soll. Die reguläre Planausführung beschreibt das Starten und Aktivieren der Unterpläne.

Um das Verständnis für die Notation der SOS-Regeln zu erhöhen, soll hier eine kurze Einführung in die verwendete Syntax gegeben werden. Die Regeln werden nach folgendem Schema aufgeschrieben:

$$\frac{\text{Precondition}}{\mathbf{sys} \overset{\sigma,\sigma'}{\longrightarrow} \mathbf{sys}'}$$

Dabei ist die eigentliche Transition zu lesen als Übergang eines Paares aus **asbru-system** und **state** in ein ebensolches Paar. Das heißt, obige Übergangsregel kann auch wie folgt aufgeschrieben werden:

$$(\mathbf{sys} \times \sigma) \longrightarrow (\mathbf{sys}' \times \sigma')$$

Diese Transition kann nur ausgeführt werden, wenn die Vorbedingung „Precondition“ erfüllt ist. Die verwendete Syntax lehnt sich an die von Balser [8] und Cau [17] verwendete an.

Die Vorbedingung sind logische Verknüpfungen prädikatenlogischer Formeln. Für diese Verknüpfungen ist die Klammerung zu beachten. Diese wird im Allgemeinen durch die Einrückungen der Formeln festgelegt. Klammern werden implizit um Formelblöcke gesetzt, die die gleiche Einrücktiefe haben. Beispielsweise soll nachstehender Formelblock

$$\begin{array}{ll} & \text{precondition}_1 \\ \wedge & \quad \text{precondition}_2 \\ & \rightarrow \text{precondition}_3 \end{array}$$

wie folgt gelesen werden:

$$\text{precondition}_1 \wedge (\text{precondition}_2 \rightarrow \text{precondition}_3)$$

Die formale Definition der in den Vorbedingungen verwendeten Prädikate sowie der Modifikationsfunktionen des Zustands findet sich im Anhang C.

9.2.2 Zustandsübergänge von `considered`

Es gibt drei Transitionen, die vom Zustand `considered` aus genommen werden können. Relevant dafür ist das Auswertungsergebnis der **filter**-Bedingung sowie die Terminierung des übergeordneten Plans. Die **filter**-Bedingung wird wie in Kapitel 8 beschrieben ausgewertet.

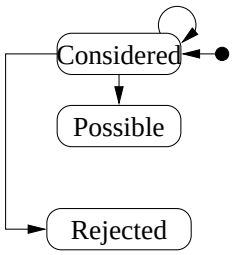


Abbildung 9.2: Zustandsübergänge ausgehend von dem Zustand `considered`

Abhängig von dieser Auswertung und der eventuellen Terminierung des übergeordneten Plans kann sich der Zustand des Plans entweder zu `possible` oder zu `rejected` verändern, oder aber in `considered` stehen bleiben.

Der Plan verbleibt im Zustand `considered`, wenn die **filter**-Bedingung derzeit nicht erfüllt, jedoch noch erfüllbar ist und der übergeordnete Plan nicht terminiert ist. Ist die Bedingung nicht mehr erfüllbar oder terminiert der übergeordnete Plan, so wechselt der Planzustand nach `rejected`. Ist die Bedingung zum Auswertungszeitpunkt erfüllt und ist der übergeordnete Plan nicht terminiert, so wechselt der Planzustand nach `possible`.

Transition `considered` ⇒ `rejected`

Ein Plan im Zustand `considered` (**isConsidered**) geht über in den Zustand `rejected` und stellt damit seine Bearbeitung ein, wenn eine von zwei möglichen Bedingungen wahr ist. Zum Einen geschieht dies, wenn die **filter**-Bedingung nicht erfüllbar ist, was durch das Prädikat **shouldRejectFilter** geprüft wird. Zum Anderen soll der Plan zurückgewiesen werden, wenn der übergeordnete Plan seine Bearbeitung eingestellt hat und terminiert ist. Dies wird durch das Prädikat **parent_terminated** geprüft.

Der Zustand wird durch diese Transition dahingehend abgeändert, dass der Planzustand des Plans auf `rejected` gesetzt wird. Weiterhin soll der Plan selbst seine Ausführung einstellen und terminieren und zuletzt soll die **Tick** Komponente des Zustands auf den Wert **false** gesetzt und somit ein Makro-Schritt verhindert werden.

$$\frac{\begin{array}{l}\quad\textbf{isConsidered}(\textbf{HS},\textbf{iname})\\ \wedge\quad\textbf{shouldRejectFilter}(\sigma,\textbf{iname})\\ \quad\vee\ \textbf{parent\_terminated}(\textbf{HS},\textbf{iname})\end{array}}{\textbf{iname}\quad \sigma,\textbf{Tick}(\sigma[\textbf{iname},\ \texttt{rejected}])\ \xrightarrow{}\ \varepsilon}$$

Listing 9.1: Transition `considered` ⇒ `rejected`

Die Klammerung der Vorbedingung ist in dieser SOS Regel wie folgt zu lesen:

$$\begin{array}{l}\quad\textbf{isConsidered}(\textbf{HS},\textbf{iname})\\ \wedge\ (\textbf{shouldRejectFilter}(\sigma,\textbf{iname})\ \vee\ \textbf{parent\_terminated}(\textbf{HS},\textbf{iname}))\end{array}$$

Transition `considered` ⇒ `possible`

Diese Transition verlangt, dass ein Plan, der im `considered`-Zustand ist in den Zustand `possible` wechselt. Dazu müssen drei Prädikate der Vorbedingung wahr sein. Das erste, das

verlangt, dass der Plan im Zustand `considered` ist (**isConsidered**), das zweite verlangt, dass der übergeordnete Plan dieses Plans nicht terminiert ist (**parent_terminated**). Das dritte Prädikat verlangt, dass die **filter**-Bedingung erfüllt ist (**shouldProceedFilter**).

Durch die Anwendung dieser SOS Regel wird das Asbru-System nicht verändert. Allerdings wird der Planzustand des Plans auf den Wert `possible` und die **Tick** Komponente des Zustands auf den Wert **false** geändert. Mit letzterem wird ein Makro-Schritt verhindert.

$$\frac{\begin{array}{c}\textbf{isConsidered}(\textbf{HS},\textbf{iname})\\ \wedge\ \neg\ \textbf{parent\_terminated}(\textbf{HS},\textbf{iname})\\ \wedge\ \textbf{shouldProceedFilter}(\sigma,\textbf{iname})\end{array}}{\textbf{iname}\quad \sigma,\textbf{Tick}(\sigma[\textbf{iname}, \texttt{possible}]) \quad \textbf{iname}}$$

Listing 9.2: Transition `considered` ⇒ `possible`

Transition `considered` ⇒ `considered`

Mittels dieser SOS-Regel wird der Stillstand eines Plans erzwungen. Die Vorbedingung verlangt, dass der betreffende Plan im Zustand `considered` ist (**isConsidered**) und dass der übergeordnete Plan nicht terminiert (**parent_terminated**), und die **filter**-Bedingung erfüllbar ist (**shouldRejectFilter**). Dies ist die Negation der Vorbedingung der Terminierungsregel 9.1. Gleichzeitig beschreibt die vierte Zeile der Vorbedingung, die erzwingt, dass die **filter**-Bedingung nicht erfüllt ist (**shouldProceedFilter**), die Negation der Vorbedingung der Transition 9.2. Damit ist diese Regel nur anwendbar, wenn die Anwendung aller anderen Regeln des `considered`-Zustands ausgeschlossen ist.

Die Regel ändert den Zustand nicht ab. Zwar wird der Planzustand des laufenden Plans auf den Wert `considered` gesetzt, jedoch erzwingt die Vorbedingung, dass der Planzustand bereits auf diesen Wert gesetzt war. Insofern hat die Aktualisierung keine Änderung des Zustands zur Folge.

$$\frac{\begin{array}{c}\textbf{isConsidered}(\textbf{HS},\textbf{iname})\\ \wedge\ \neg\ \textbf{parent\_terminated}(\textbf{HS},\textbf{iname})\\ \wedge\ \neg\ \textbf{shouldRejectFilter}(\sigma,\textbf{iname})\\ \wedge\ \neg\ \textbf{shouldProceedFilter}(\sigma,\textbf{iname})\end{array}}{\textbf{iname}\quad \sigma,\sigma[\textbf{iname}, \texttt{considered}] \quad \textbf{iname}}$$

Listing 9.3: Transition `considered` ⇒ `considered`

9.2.3 Zustandsübergänge von `possible`

Der Zustand `possible` ist dem Zustand `considered` sehr ähnlich, abgesehen davon, dass in diesem Zustand statt der **filter**-Bedingung die **setup**-Bedingung ausgewertet wird. Auch im Zustand `possible` hängt von der Auswertung der Bedingung und dem Terminierungsverhalten des übergeordneten Plans ab, ob ein Zustandsübergang zu `ready` oder zu `rejected`

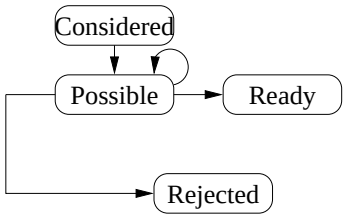


Abbildung 9.3: Zustandsübergänge ausgehend von dem Zustand `possible`

durchgeführt wird oder ob die Plankontrolle im Zustand `possible` stehen bleibt. Dies wird in Abbildung 9.3 visualisiert.

Transition `possible` ⇒ `rejected`

Falls ein Plan im Zustand `possible` ist (**isPossible**), soll er zurückgewiesen werden, wenn die **setup**-Bedingung nicht mehr erfüllbar oder der übergeordnete Plan terminiert ist. Ersteres wird durch das Prädikat **shouldRejectSetup** beschrieben, letzteres durch das Prädikat **parent_terminated**.

In beiden Fällen, in denen diese Regel anwendbar ist, soll der Zustand des Plans auf den Wert `rejected` geändert werden. Außerdem soll die **Tick** Komponente des Zustands auf den Wert **false** gesetzt und so ein Mikro-Schritt erzwungen werden.

$$\frac{\begin{array}{l}\mathbf{isPossible}(\mathbf{HS},\mathbf{iname}) \\ \wedge\ \ \mathbf{shouldRejectSetup}(\sigma,\mathbf{iname}) \\ \vee\ \mathbf{parent\_terminated}(\mathbf{HS},\mathbf{iname})\end{array}}{\mathbf{iname}\ \ \sigma,\mathbf{Tick}(\sigma[\mathbf{iname},\ \mathtt{rejected}])\ \ \varepsilon}$$

Listing 9.4: Transition `possible` ⇒ `rejected`

Transition `possible` ⇒ `ready`

Nachstehende Transition ist für einen Plan im Zustand `possible` (**isPossible**) anwendbar, wenn der übergeordnete Plan nicht terminiert (**parent_terminated**) und die **setup**-Bedingung erfüllt ist (**shouldProceedSetup**).

Durch Anwendung dieser Regel wird der Zustand des Plans auf `ready` geändert. Weiterhin wird die **Tick** Komponente des Zustands auf den Wert **false** gesetzt und so ein Makro-Schritt verhindert.

$$\frac{\begin{array}{l}\mathbf{isPossible}(\mathbf{HS},\mathbf{iname}) \\ \wedge\ \neg\ \mathbf{parent\_terminated}(\mathbf{HS},\mathbf{iname}) \\ \wedge\ \mathbf{shouldProceedSetup}(\sigma,\mathbf{iname})\end{array}}{\mathbf{iname}\ \ \sigma,\mathbf{Tick}(\sigma[\mathbf{iname},\ \mathtt{ready}])\ \ \mathbf{iname}}$$

Listing 9.5: Transition `possible` ⇒ `ready`

Transition `possible` ⇒ `possible`

Wie auch bei Plänen im Zustand `considered` gibt es bei Plänen im Zustand `possible` (**isPossible**) die Möglichkeit, dass diese weder abbrechen noch weiter fortschreiten. Dies ist immer dann der Fall, wenn keine der anderen Regeln des Zustands anwendbar ist, also wenn der übergeordnete Plan nicht terminiert ist (**parent_terminated**), außerdem die **setup**-Bedingung nicht erfüllt (**shouldProceedSetup**) und nicht erfüllbar (**shouldRejectSetup**) ist. In diesem Fall wird der Zustand durch die Transition nicht verändert.

$$\frac{\begin{array}{l}\textbf{isPossible}(\textbf{HS},\textbf{iname})\\ \wedge\ \neg\ \textbf{parent\_terminated}(\textbf{HS},\textbf{iname})\\ \wedge\ \neg\ \textbf{shouldProceedSetup}(\sigma,\textbf{iname})\\ \wedge\ \neg\ \textbf{shouldRejectSetup}(\sigma,\textbf{iname})\end{array}}{\textbf{iname}\quad \sigma,\sigma[\textbf{iname},\ \texttt{possible}] \longrightarrow \quad \textbf{iname}}$$

Listing 9.6: Transition `possible` ⇒ `possible`

9.2.4 Zustandsübergänge von `ready`

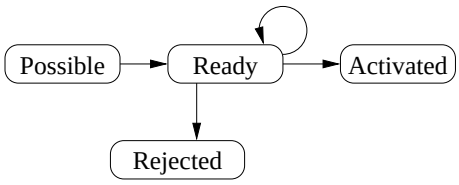


Abbildung 9.4: Zustandsübergänge ausgehend von dem Zustand `ready`

Der Zustand `ready` wird zur Synchronisation der Pläne eingeführt. Einige Plankontrolltypen, wie etwa der `parallel` und der `anyorder` Kontrolltyp benötigen einen Synchronisationspunkt. An diesem Synchronisationspunkt sammeln sich alle Unterpläne, die die `Selection`-Phase erfolgreich durchlaufen haben und danach auf das Aktivierungssignal `E` warten.

Soll ein Plan im Zustand `ready` in die `Execution`-Phase wechseln, so muss ihm der übergeordnete Plan das Aktivierungssignal `E` schicken. Dieses konsumiert der Plan beim Wechsel in die `Execution`-Phase, setzt es also zurück. Falls der übergeordnete Plan eines Plans terminiert, der sich im Zustand `ready` befindet, bricht auch dieser Plan seine Auswertung ab und setzt sich in den Zustand `rejected`. Dies wird in Abbildung 9.4 dargestellt.

Transition `ready` ⇒ `rejected`

Befindet sich ein Plan im Zustand `ready` (**isReady**), so bricht er die Ausführung ab und setzt sich in den Zustand `rejected`, wenn sein übergeordneter Plan terminiert (**parent_terminated**).

Die Terminierung des Plans verlangt, dass der Zustand des Plans auf den Wert `rejected` gesetzt und das den Plan repräsentierende Asbru-System zu ε abgeändert wird. Aufgrund dieser Änderung des Zustands wird die **Tick** Komponente des Zustands auf den Wert **false** gesetzt, wodurch ein Makro-Schritt verhindert wird.

$$\frac{\begin{array}{l}\mathbf{isReady}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \mathbf{parent\_terminated}(\mathbf{HS},\mathbf{iname})\end{array}}{\mathbf{iname}\ \ \sigma,\mathbf{Tick}(\sigma[\mathbf{iname},\ \mathtt{rejected}])\ \ \varepsilon}$$

Listing 9.7: Transition `ready` ⇒ `rejected`

Transition `ready` ⇒ `activated`

Ein Plan, der im Zustand `ready` ist (**isReady**) und dessen übergeordneter Plan nicht terminiert ist (**parent_terminated**) soll diese Transition ausführen, sofern das Aktivierungssignal des Plans gesetzt ist (**isActSig**).

Durch Anwendung dieser Regel werden die Zustände aller Unterpläne auf den Wert `inactive` zurückgesetzt. Dazu wird die Liste aller Unterpläne des Plans in der Variablen **PL** gespeichert. Die Belegung der **PL**-Variablen geschieht über die letzte Zeile der Vorbedingung der SOS-Regel.

Die Zustandsänderung, die durch diese Regel beschrieben wird, verlangt, dass der betrachtete Plan seinen Zustand auf `activated` setzt. Weiterhin wird der Rundenzähler der zyklischen Pläne für diesen Plan auf den Wert 0 gesetzt und zuletzt wird das Aktivierungssignal zurückgesetzt bzw. konsumiert. Zusätzlich werden alle Planzustände der Pläne der Unterplanliste auf den Wert `inactive` gesetzt. Aufgrund dieser Zustandsänderung soll ein Mikro-Schritt im Anschluss an diese Regel erzwungen werden, was durch das Setzen der **Tick** Komponente des Zustands geschieht.

$$\frac{\begin{array}{l}\mathbf{isReady}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \neg\ \mathbf{parent\_terminated}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \mathbf{isActSig}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \mathbf{PL} = \mathtt{asbru}(\mathbf{nameSel}(\mathbf{iname}))\mathbf{.subplans}\end{array}}{\mathbf{iname}\ \ \sigma,\mathbf{Tick}(\sigma[\mathbf{iname},\ \mathtt{activated}][\mathbf{PL},\ \mathbf{iname},\ \mathtt{inactive}][\mathbf{iname},\ 0][\mathbf{iname}]_{nac})\ \ \mathbf{iname}}$$

Listing 9.8: Transition `ready` ⇒ `activated`

Transition `ready` ⇒ `ready`

Ist das Aktivierungssignal (**isActSig**) eines Plans im Zustand `ready` (**isReady**) noch nicht gesetzt und der übergeordnete Plan nicht abgebrochen (**parent_terminated**), so bleibt der Plan im Zustand `ready`. In dieser Situation erlaubt der Plan einen Makro-Schritt, da der Plan blockiert ist.

$$\frac{\begin{array}{l}\texttt{isReady}(\texttt{HS},\texttt{iname})\\ \wedge\,\neg\ \texttt{parent\_terminated}(\texttt{HS},\texttt{iname})\\ \wedge\,\neg\ \texttt{isActSig}(\texttt{HS},\texttt{iname})\end{array}}{\texttt{iname}\ \overset{\sigma,\sigma[\texttt{iname},\ \texttt{ready}]}{\longrightarrow}\ \texttt{iname}}$$

Listing 9.9: Transition `ready` ⇒ `ready`

9.2.5 Allgemeine Zustandsübergänge von `activated`

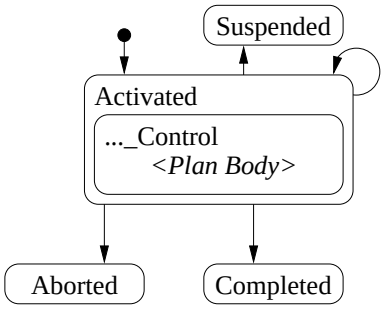


Abbildung 9.5: Zustandsübergänge ausgehend von dem Zustand `activated`

Der `activated` Zustand erlaubt eine sehr große Menge verschiedener Transitionen bedingt durch das unterschiedliche Verhalten der einzelnen Plantypen. Daher gliedert sich die Vorstellung der Zustandsübergangsregeln in zwei Teile, dem Ersten in dem diejenigen Transitionen vorgestellt werden, die von allen Plankontrolltypen gleichermaßen berücksichtigt werden und danach dem Zweiten, der die Regeln beinhaltet, die jeweils nur für einen Plankontrolltyp gelten.

Die Regeln, die das plankontrolltypenspezifische Verhalten spezifizieren haben eine geringere Priorität als die allgemeinen Regeln. Das heißt, zunächst überprüft jeder Plan unabhängig von seinem Typ, ob er die Auswertung abbrechen, beenden oder unterbrechen soll und erst wenn all diese Fälle verneint werden, kommen die spezifischen Regeln zur Anwendung. Damit bei diesen Regeln nicht in die Vorbedingung aufgenommen werden muss, dass kein Abbruch-, Beendigungs- und Unterbrechungsfall vorliegt, werden die spezifischen Regeln mit dem Zusatz SOS_{con} gekennzeichnet. Damit wird diese Überprüfung implizit bei jeder Regelanwendung durchgeführt, da die SOS_{con}-Regeln nur angewendet werden, wenn kein Terminierungs- oder Unterbrechungsfall vorliegt.

In diesem Abschnitt werden zunächst die allgemeinen Zustandsübergänge ausgehend vom Zustand `activated` beschrieben. Diese sind in Abbildung 9.5 dargestellt. Anschließend werden in Abschnitt 9.2.6 die Zustandsübergänge aufgeführt, die den Planzustand `suspended` voraussetzen und die danachfolgenden Abschnitte führen jeweils alle `con` Regeln auf, die die Semantik eines spezifischen Plantypen beschreiben.

Transition `activated` ⇒ `aborted`

Ein Plan in der `Execution`-Phase kann aus mehreren Gründen seinen Zustand nach `aborted` ändern. Der erste Grund dafür ist die Erfüllung der **abort**-Bedingung, der Zweite eine Terminierung des Superplanes und der Dritte eine nicht mehr gegebene Erfüllbarkeit der Warte-Bedingung.

Dabei wird die **abort**-Bedingung wie in Kapitel 8 beschrieben ausgewertet. Die Warte-Bedingung spezifiziert, wie in Abschnitt 8.2 angegeben, Mengen von Unterplänen, die abgeschlossen sein müssen, bevor der übergeordnete Plan seine Bearbeitung abschließen darf. Sollte feststellbar sein, dass diese Bedingung nicht mehr erfüllt werden kann, so bricht die Ausführung ab und der Plan wechselt in den Zustand `aborted`.

Die Auswertung der **abort**-Bedingung, der Wartebedingung und der Terminierung des übergeordneten Planes wird in dem Prädikat **shouldAbort** gekapselt. Dieses ist wahr, wenn eine der drei Abbruchbedingungen wahr ist.

Ein Plan, der im Zustand `activated` (**isActivated**) oder `suspended` (**isSuspended**) ist, bricht ab, wenn das Prädikat **shouldAbort** für diesen Plan zu wahr auswertet. In diesem Fall setzt der Plan seinen Planzustand um auf `aborted`, lässt sein Asbru-System terminieren (ε) und verhindert einen Makro-Schritt durch Setzen der **Tick** Komponente des Zustands auf den Wert **false**.

$$\frac{\begin{array}{r}\textbf{isActivated}(\textbf{HS},\textbf{iname})\\ \vee\ \textbf{isSuspended}(\textbf{HS},\textbf{iname})\\ \wedge\ \textbf{shouldAbort}(\sigma,\textbf{iname})\end{array}}{\textbf{iname}\quad \sigma,\textbf{Tick}(\sigma[\textbf{iname}, \xrightarrow{} \texttt{aborted}])\quad \varepsilon}$$

Listing 9.10: Transition `activated` ⇒ `aborted`

Transition `activated` ⇒ `completed`

Ist ein Plan im Zustand `activated` (**isActivated**), so kann er in den Zustand `completed` wechseln, sofern das Abbruchkriterium **shouldAbort** nicht wahr ist. Dazu ist erforderlich, dass die **complete**-Bedingung, sowie die Wartebedingung bezüglich der Unterpläne erfüllt sind. Diese Bedingungen werden in dem Prädikat **shouldComplete** gebündelt.

Sind diese Vorbedingungen erfüllt, ändert der Plan seinen Zustand auf den Wert `completed`, terminiert das zu ihm gehörige Asbru-System und verhindert mit einer Umsetzung der **Tick** Komponente des Zustands einen Makro-Schritt.

$$\frac{\begin{array}{r}\textbf{isActivated}(\textbf{HS},\textbf{iname})\\ \wedge\ \neg\ \textbf{shouldAbort}(\sigma,\textbf{iname})\\ \wedge\ \textbf{shouldComplete}(\sigma,\textbf{iname})\end{array}}{\textbf{iname}\quad \sigma,\textbf{Tick}(\sigma[\textbf{iname}, \xrightarrow{} \texttt{completed}])\quad \varepsilon}$$

Listing 9.11: Transition `activated` ⇒ `completed`

Transition `activated` ⇒ `suspended`

Ist ein Plan im Zustand `activated` (**isActivated**), und trifft weder die Vorbedingung zum Abbruch des Plans (**shouldAbort**) noch die Vorbedingung zur Beendigung (**shouldComplete**) zu, so wird überprüft, ob der Plan seine Ausführung unterbrechen

soll. Dies ist immer dann der Fall, wenn die **suspend**-Bedingung erfüllt ist oder der übergeordnete Plan dieses Plans sich im Zustand `suspended` befindet. Die Abfrage dieser Bedingungen wird im Prädikat **shouldSuspend** gekapselt.

Soll der Plan seine Ausführung unterbrechen, so ändert der Plan seinen Planzustand nach `suspended` und erzwingt einen Mikro-Schritt durch Setzen der **Tick** Komponente des Zustands auf den Wert **false**.

$$\frac{\begin{array}{l}\mathbf{isActivated}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \neg\ \mathbf{shouldAbort}(\sigma, \mathbf{iname}) \\ \wedge\ \neg\ \mathbf{shouldComplete}(\sigma, \mathbf{iname}) \\ \wedge\ \mathbf{shouldSuspend}(\sigma, \mathbf{iname})\end{array}}{\mathbf{iname}\ \ \sigma, \mathbf{Tick}(\sigma[\mathbf{iname}, \underrightarrow{\ \ } \texttt{suspended}])\ \ \mathbf{iname}}$$

Listing 9.12: Transition `activated` ⇒ `suspended`

Transition `activated` ⇒ `con`

Trifft keine der Vorbedingungen der oben aufgeführten Regeln des `activated` Zustands zu, so sollen die plankontrolltypenspezifischen Regeln angewendet werden. Dazu muss der Plan im Zustand `activated` (**isActivated**) sein. Es darf außerdem kein Grund zum Abbrechen (**shouldAbort**), zum Beenden (**shouldComplete**) oder zum Unterbrechen des Plans (**shouldSuspend**) vorliegen.

In diesem Fall wird der Zustandsübergang des Plans dadurch definiert, dass eine der `con` Regeln angewendet wird und das Ergebnis dieser Regelanwendung als Ergebnis des Zustandsübergangs dieser Regel verwendet wird. Auch das aus Anwendung dieser Regel resultierende Asbru-System wird durch den `con` Übergang bestimmt.

$$\frac{\begin{array}{l}\mathbf{isActivated}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \neg\ \mathbf{shouldAbort}(\sigma, \mathbf{iname}) \\ \wedge\ \neg\ \mathbf{shouldComplete}(\sigma, \mathbf{iname}) \\ \wedge\ \neg\ \mathbf{shouldSuspend}(\sigma, \mathbf{iname}) \\ \wedge\ \mathbf{iname}\ \overset{\sigma, \sigma'}{\longrightarrow}_{\texttt{Con}}\ \mathbf{sys}\end{array}}{\mathbf{iname}\ \overset{\sigma, \sigma'}{\longrightarrow}\ \mathbf{sys}}$$

Listing 9.13: Transition `activated` ⇒ `con`

9.2.6 Allgemeine Zustandsübergänge von `suspended`

Der Zustand `suspended` beschreibt einen Planzustand, in dem der Plan seine Ausführung temporär unterbricht. Das bedeutet, dass er keine weiteren Unterpläne startet oder aktiviert und implizit die bereits aktiven Unterpläne durch Planzustandspropagation auch in den `suspended` Zustand versetzt. Ein Plan in diesem Zustand kann bis zu seiner Reaktivierung außer einem Abbruch keine Zustandsänderungen ausführen.

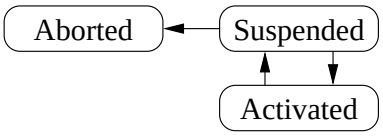


Abbildung 9.6: Zustandsübergänge ausgehend von dem Zustand `suspended`

Transition `suspended` ⇒ `aborted`

Ein Plan im Zustand `suspended` (**isSuspended**) prüft die gleichen Abbruch-Bedingungen (**shouldAbort**) wie ein Plan im Zustand `activated`. Das bedeutet, auch hier wird getestet, ob der übergeordnete Plan abgebrochen wurde, die **abort**-Bedingung erfüllt wurde oder die Warte-Bedingung des Plans nicht mehr erfüllbar ist. Diese Prüfung ist gekapselt in dem Prädikat **shouldAbort**. Ist dieses wahr, so bricht der Plan die Ausführung ab.

In diesem Fall setzt der Plan seinen Planzustand um auf `aborted`, lässt sein Asbru-System terminieren (ε) und verhindert einen Makro-Schritt durch Setzen der **Tick** Komponente des Zustands auf den Wert **false**.

$$\frac{\begin{array}{r}\mathbf{isActivated}(\mathbf{HS},\mathbf{iname})\\ \vee\ \mathbf{isSuspended}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \mathbf{shouldAbort}(\sigma,\mathbf{iname})\end{array}}{\mathbf{iname}\quad \sigma,\mathbf{Tick}(\sigma[\mathbf{iname},\ \texttt{aborted}])\ \xrightarrow{}_{\varepsilon}}$$

Listing 9.14: Transition `suspended` ⇒ `aborted`

Transition `suspended` ⇒ `activated`

Ein Plan im Zustand `suspended` (**isSuspended**), der seine Ausführung nicht abbrechen soll (**shouldAbort**), soll zurück in den Zustand `activated` wechseln, sofern der übergeordnete Plan (wieder) im Zustand `activated` ist und die **reactivate**-Bedingung erfüllt ist. Diese beiden Tests sind im Prädikat **shouldReactivate** gekapselt.

Sind diese Bedingungen erfüllt, soll der Plan seine Ausführung wieder aufnehmen und dazu seinen Zustand auf den Wert `activated` zurücksetzen. Weiterhin setzt der Plan in diesem Schritt die **Tick**-Komponente des Zustands auf den Wert **false** und verhindert so einen Makro-Schritt.

$$\frac{\begin{array}{r}\mathbf{isSuspended}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \neg\ \mathbf{shouldAbort}(\sigma,\mathbf{iname})\\ \wedge\ \mathbf{shouldReactivate}(\sigma,\mathbf{iname})\end{array}}{\mathbf{iname}\quad \sigma,\mathbf{Tick}(\sigma[\mathbf{iname},\ \texttt{activated}])\ \xrightarrow{}\ \mathbf{iname}}$$

Listing 9.15: Transition `suspended` ⇒ `activated`

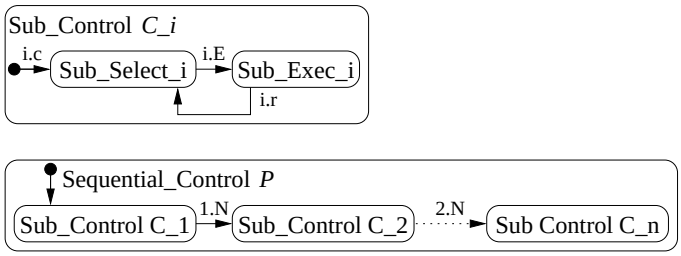


Abbildung 9.7: Zustandsübergänge des `sequential` Kontrolltyps

Transition `suspended` ⇒ `suspended`

Ein Plan im Zustand `suspended` (**isSuspended**) soll in diesem bleiben, wenn er weder seine Auswertung abbrechen (**shouldAbort**) soll noch zurück in den Zustand `activated` wechseln soll (**shouldReactivate**).

Der Plan verbleibt dann im Zustand `suspended`. Da der Planzustand des Plans bereits vor der Ausführung auf dem Wert `suspended` war, bedeutet dies, dass sich durch Anwendung dieser Regel keine Zustandsänderung ergibt. Daher ist nach Anwendung dieser Regel ein Makro-Schritt möglich.

$$\frac{\begin{array}{l}\textbf{isSuspended}(\textbf{HS}, \textbf{iname}) \\ \wedge\ \neg\ \textbf{shouldAbort}(\sigma, \textbf{iname}) \\ \wedge\ \neg\ \textbf{shouldReactivate}(\sigma, \textbf{iname})\end{array}}{\textbf{iname}\ \ \sigma, \sigma[\textbf{iname}, \xrightarrow{} \texttt{suspended}]\ \ \textbf{iname}}$$

Listing 9.16: Transition `suspended` ⇒ `suspended`

9.2.7 Zustandsübergänge der `sequential` Kontrolle

Aufgabe der `sequential` kontrollierten Pläne ist die sequentielle Abarbeitung der Unterpläne unter Berücksichtigung der Reihenfolge, in der diese angegeben sind. Dies wird in Abbildung 9.7 dargestellt.

In dieser Abbildung ist zu sehen, dass die Pläne nacheinander jeweils komplett abgearbeitet werden. Das bedeutet, dass die Pläne der Reihe nach zunächst in die `Selection`-Phase versetzt und dann aktiviert werden. Dazu wird der i-te Plan gestartet (`c`-Signal) und aktiviert(`E`-Signal), sobald die Ausführung des i-1-ten Plans beendet wurde. In der Semantik wird das Signal `c` implementiert durch das Starten des jeweiligen Unterplans. Bevor der jeweils nächste Unterplan gestartet wird, erhält ein abgebrochener Plan gegebenenfalls das Signal zur Reaktivierung. Dies ist davon abhängig, ob der `sequential`-kontrollierte Plan seine Unterpläne grundsätzlich bei Abbruch neu startet oder nicht.

Passiver Übergang

Ein Sequentieller Plan (**isSequential**) soll sich passiv verhalten, wenn er einen Unterplan hat, der in der `Execution`-Phase (**hasExecutedSub**), oder in der `Selection`-Phase ist (**hasSelectedSub**) oder wenn er keinen Unterplan hat, der in einem startbaren Zustand ist

(**hasStartableSub**). Letzteres ist immer dann der Fall, wenn kein Unterplan im Zustand `inactive` ist. Außerdem darf kein Unterplan im Zustand `aborted` sein oder der Plan darf Unterpläne im Zustand `aborted` nicht neu starten.

Ist dies der Fall, verhält sich die `sequential`-Kontrolle passiv und ändert den Zustand nicht. Entsprechend wird ein Makro-Schritt nicht verhindert.

$$\frac{\begin{array}{l}\quad \mathbf{isSequential}(\mathbf{iname}) \\ \wedge \quad \mathbf{hasExecutedSub}(\mathbf{HS}, \mathbf{iname}) \\ \quad \vee \mathbf{hasSelectedSub}(\mathbf{HS}, \mathbf{iname}) \\ \quad \vee \neg\ \mathbf{hasStartableSub}(\mathbf{HS}, \mathbf{iname})\end{array}}{\mathbf{iname}\ \sigma, \sigma[\mathbf{iname}, \xrightarrow{} \texttt{activated}]_{\texttt{Con}}\ \ \mathbf{iname}}$$

Listing 9.17: Passiver Übergang

Aktivierungsfall

Eine `sequential`-kontrollierter Plan (**isSequential**), der keine Unterpläne in der `Execution`-Phase (**hasExecutedSub**) und keine Unterpläne in der `Selection`-Phase (**hasSelectedSub**) hat, soll, sofern ein startbarer Plan existiert (**hasStartableSub**) den ersten startbaren Plan auswählen (**selectFirstStartable**), starten und aktivieren. Ein Plan hat einen startbaren Unterplan, wenn ein Unterplan im Zustand `inactive` existiert, oder aber ein Unterplan im Zustand `aborted` ist und der Plan abgebrochene Unterpläne neu startet. Der Variablen $\mathbf{iname}_1$ wird der erste Unterplan zugewiesen, der startbar ist.

Ist diese Regel anwendbar, so wird der durch den Instanznamen $\mathbf{iname}_1$ bezeichnete Plan parallel zum Plan **iname** ausgeführt. Weiterhin bleibt der Planzustand von **iname** unverändert auf dem Wert `activated`, der Planzustand des ausgewählten Unterplans wird auf den Wert `considered` gesetzt und der Unterplan wird aktiviert. Die **Tick**-Komponente des Zustands wird auf den Wert **false** gesetzt und damit ein Mikro-Schritt erzwungen.

$$\frac{\begin{array}{l}\quad \mathbf{isSequential}(\mathbf{iname}) \\ \wedge \neg\ \mathbf{hasExecutedSub}(\mathbf{HS}, \mathbf{iname}) \\ \wedge \neg\ \mathbf{hasSelectedSub}(\mathbf{HS}, \mathbf{iname}) \\ \wedge \mathbf{hasStartableSub}(\mathbf{HS}, \mathbf{iname}) \\ \wedge \mathbf{iname}_1 = \mathbf{nameGen}(\mathbf{selectFirstStartable}(\mathbf{HS}, \mathbf{iname}), \mathbf{iname})\end{array}}{\mathbf{iname}\ \sigma, \mathbf{Tick}(\sigma[\mathbf{iname}, \xrightarrow{} \texttt{activated}][\mathbf{iname}_1, \texttt{considered}][\mathbf{iname}_1]_{ac})_{\texttt{Con}}\ \ \mathbf{iname} \|_s\ \mathbf{iname}_1}$$

Listing 9.18: Aktivierungsfall

9.2.8 Zustandsübergänge der `anyorder` Kontrolle

Pläne, die vom Kontrolltypen `anyorder` gesteuert werden, aktivieren, wie auch die `sequential` Kontrolle, maximal einen Unterplan auf einmal. Dabei wird allerdings nicht die durch die Unterplanliste vorgegebene Ordnung der Pläne beachtet. Um bei der

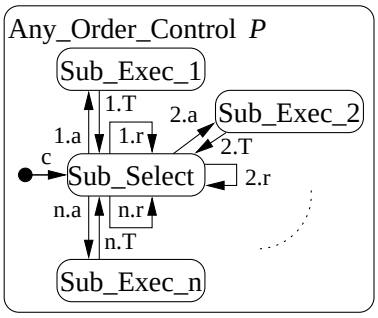


Abbildung 9.8: Zustandsübergänge des `anyorder` Kontrolltyps

Aktivierung nur die wirklich anwendbaren Pläne zu berücksichtigen, werden trotz der Sequentialität der Aktivierungssignale zunächst alle Unterpläne gleichzeitig gestartet. Nachdem Unterpläne den Zustand `ready` erreichen, wird ihre Ausführung angehalten und nur jeweils ein Unterplan kann zu einem Zeitpunkt in der `Execution`-Phase sein oder aktiviert werden. Dies wird graphisch dargestellt in Abbildung 9.8.

Dort werden zunächst alle Pläne gestartet und dann jeweils ein Plan indeterministisch aktiviert. Verlässt der Unterplan die `Execution`-Phase, so wird ein anderer Plan des Zustandes `ready` ausgewählt und aktiviert. Dabei ist zu beachten, dass gegebenenfalls Pläne im Zustand `aborted` neu gestartet werden, was durch Transition $i.r$ dargestellt wird. Diese Transition hat eine höhere Priorität als die Aktivierung von Unterplänen, so dass vor der Aktivierung von Unterplänen zunächst immer abgebrochene Unterpläne neu gestartet werden sollen.

Aktivierung von Unterplänen

Ein `anyorder` kontrollierter Plan (**isAnyorder**) startet alle (neu)startbaren Unterpläne (**selectAllStartable**), sofern welche existieren (**hasStartableSub**). Die Zuweisung an die Planliste **PL** sowie das Prädikat **planNames** in der Vorbedingung stellen sicher, dass alle startbaren Pläne ausgewählt wurden und das neu gestartete Asbru-System genau die Planinstanzen startet, die (neu) gestartet werden müssen.

Ist diese Regel anwendbar, so werden alle startbaren Unterpläne ausgewählt, parallel zum laufenden Plan gestartet und deren Zustand auf den Wert `considered` gesetzt. Ein Makro-Schritt wird dadurch verhindert.

$$\frac{\begin{array}{l}\mathbf{isAnyorder}(\mathbf{iname})\\ \wedge\ \mathbf{hasStartableSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \mathbf{PL} = \mathbf{selectAllStartable}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \mathbf{planNames}(\mathbf{PL},\mathbf{iname},\mathbf{sys})\end{array}}{\mathbf{iname}\ \sigma,\mathbf{Tick}(\sigma[\mathbf{iname},\ \mathtt{activated}]\overrightarrow{[}\mathbf{PL},\ \mathbf{iname},\ \mathtt{considered}])_{\mathtt{Con}}\ \ \mathbf{iname}\|_s\ \mathbf{sys}}$$

Listing 9.19: Aktivierung von Unterplänen

Aktivierter Unterplan

Hat ein `anyorder`-kontrollierter Plan (**isAnyorder**) keinen startbaren Unterplan (**hasStartableSub**) und ist mindestens ein Unterplan des Plans in der `Execution`-Phase,

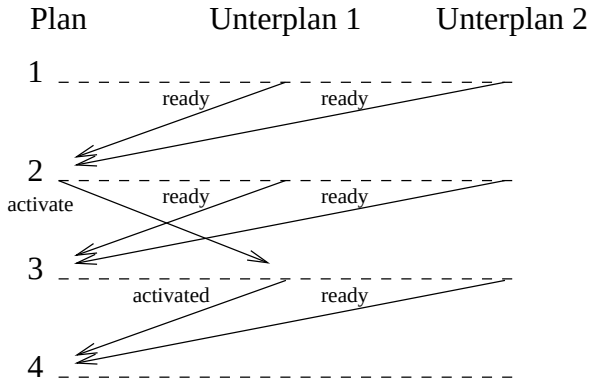


Abbildung 9.9: Verzögerung bei der Auswertung von Signalen

dann soll sich der Plan passiv verhalten.

Bei der Aktivierung von Unterplänen ergibt sich eine zeitliche Verzögerung zwischen dem Absenden des Aktivierungssignals und der Reaktion der Unterplans auf dieses Signal. Dieser Sachverhalt ist in Abbildung 9.9 dargestellt. Angenommen eine Reihe von Plänen gelangen gleichzeitig im Schritt 1 in den Zustand `ready`. Der übergeordnete Plan wird dies erst im nächsten Zustand festellen können und darauf ein Aktivierungssignal senden, welches der aktivierte Unterplan aber erst in Schritt 3 empfangen kann. Die Reaktion auf diese Signal kann der übergeordnete Plan erst im Schritt 4 feststellen, nachdem er das Signal in Schritt 2 versendet hat. Er muss also im Schritt 3 nicht nur prüfen, ob ein Plan in der `Execution`-Phase vorliegt, sondern auch, ob er bereits ein Signal verschickt hat, welches in diesem Schritt bearbeitet wird. Diese Prüfung wird durch das Prädikats **hasExecOrSigReadySub** durchgeführt.

Ist diese Regel anwendbar, so bleibt der Zustand unverändert. Es wird auch kein Makro-Schritt verhindert.

$$\frac{\begin{array}{l}\textbf{isAnyorder}(\textbf{iname}) \\ \wedge\, \neg\ \textbf{hasStartableSub}(\textbf{HS}, \textbf{iname}) \\ \wedge\, \textbf{hasExecOrSigReadySub}(\textbf{HS}, \textbf{iname})\end{array}}{\textbf{iname} \quad {}^{\sigma, \sigma[\textbf{iname},\ \texttt{activated}]}_{\longrightarrow_{\texttt{Con}}} \quad \textbf{iname}}$$

Listing 9.20: Aktivierter Unterplan

Bereiter Unterplan

Diese Regel beschreibt den Fall, in dem ein `anyorder`-kontrollierter Plan (**isAnyorder**) weder startbare Unterpläne hat (**hasStartableSub**), noch Unterpläne, die in der `Execution`-Phase sind oder bereits aktiviert wurden (**hasExecOrSigReadySub**). Existieren dann Unterpläne im Zustand `ready` (**hasReadySub**), so soll einer dieser bereiten Pläne aktiviert werden. Der Name des zu startenden Plans, der mit der Funktion **selectPrivilegedSub** bestimmt wird, wird an den Instanznamen $\textbf{iname}_1$ gebunden. Die Funktion **selectPrivilegedSub** wählt in Abhängigkeit der Umgebungssignale einen Unterplan aus, der vom medizinischen Personal präferiert wird.

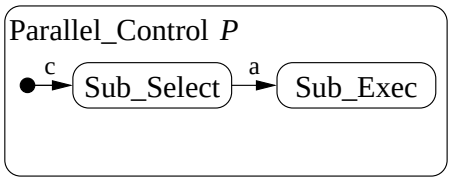


Abbildung 9.10: Zustandsübergänge des `parallel` Kontrolltyps

Durch die Regel wird das Aktivierungssignal des ausgewählten Plans $\mathbf{iname}_1$ gesetzt. Weiterhin wird der **Tick**-Komponente des Zustands der Wert **false** zugewiesen und so ein Makro-Schritt verhindert.

$$\frac{\begin{array}{l}\mathbf{isAnyorder}(\mathbf{iname}) \\ \wedge \neg\ \mathbf{hasStartableSub}(\mathbf{HS}, \mathbf{iname}) \\ \wedge \neg\ \mathbf{hasExecOrSigReadySub}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \mathbf{hasReadySub}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \mathbf{iname}_1 = \mathbf{nameGen}(\mathbf{selectPrivilegedSub}(\mathbf{HS}, \mathbf{EAGG}, \mathbf{iname}), \mathbf{iname})\end{array}}{\mathbf{iname} \quad \sigma, \mathbf{Tick}(\sigma[\mathbf{iname},\ \texttt{activated}][\mathbf{iname}_1]_{ac}) \quad \xrightarrow[\texttt{Con}]{} \quad \mathbf{iname}}$$

Listing 9.21: Bereiter Unterplan

Kein Unterplan

Die letzte Regel des `anyorder`-Kontrolltypen (**isAnyorder**) ist anwendbar, wenn der Plan keinen startbaren (**hasStartableSub**) Plan, keinen in der `Execution`-Phase (**hasExecutedSub**) und keinen im Zustand `ready` hat (**hasReadySub**).

In diesem Fall soll der Plan sich passiv verhalten und den Zustand nicht verändern. Dabei wird die Möglichkeit eines Makro-Schrittes zugelassen.

$$\frac{\begin{array}{l}\mathbf{isAnyorder}(\mathbf{iname}) \\ \wedge \neg\ \mathbf{hasStartableSub}(\mathbf{HS}, \mathbf{iname}) \\ \wedge \neg\ \mathbf{hasExecutedSub}(\mathbf{HS}, \mathbf{iname}) \\ \wedge \neg\ \mathbf{hasReadySub}(\mathbf{HS}, \mathbf{iname})\end{array}}{\mathbf{iname} \quad \sigma, \sigma[\mathbf{iname},\ \texttt{activated}] \quad \xrightarrow[\texttt{Con}]{} \quad \mathbf{iname}}$$

Listing 9.22: Kein Unterplan

9.2.9 Zustandsübergänge der `parallel` Kontrolle

Der `parallel`-Kontrolltyp beschreibt eine Ausführung, bei der alle Unterpläne synchronisiert gestartet und aktiviert werden. Dies wird dargestellt in Abbildung 9.10. Die `parallel` Kontrolle startet abgebrochene Unterpläne unabhängig von der Spezifikation des **.retry-aborted** Flags des `parallel`-Plans nicht neu.

Laufende Pläne

Ein `parallel` kontrollierter Plan (**isParallel**) der Unterpläne in der `Execution`-Phase hat oder solche die im Zustand `ready` sind und bereits aktiviert wurden (**hasExecOrSigReadySub**), verhält sich passiv.

In einem solchen Schritt ändert den Zustand nicht. Daher erlaubt die Regel die Durchführung eines Makro-Schritts.

$$\frac{\begin{array}{l}\textbf{isParallel}(\textbf{iname}) \\ \wedge\ \textbf{hasExecOrSigReadySub}(\textbf{HS}, \textbf{iname})\end{array}}{\textbf{iname}\ \ \sigma, \sigma[\textbf{iname}, \texttt{activated}]_{\texttt{Con}}\ \ \textbf{iname}}$$

Listing 9.23: Laufende Pläne

Aktivierung der Unterpläne

Ein `parallel`-kontrollierter Plan (**isParallel**), der keine Pläne hat, die in der `Execution`-Phase sind bzw. im Zustand `ready` bereits das Aktivierungssignal erhalten haben (**hasExecOrSigReadySub**) und dessen Unterpläne alle im Zustand `ready` oder im Zustand `rejected` sind (**allSubsReadyOrRejected**) soll alle Unterpläne auswählen, die im Zustand `ready` sind (**selectAllReady**) und diese aktivieren, sofern mindestens ein Unterplan in der `Selection`-Phase existiert (**hasSelectedSub**).

Für die Aktivierung wird das Ergebnis des Aufrufs der **selectAllReady** Funktion an die Plannamensliste **PL** gebunden. Alle Pläne in dieser Liste werden aktiviert. Aufgrund der Zustandsänderung soll ein Mikro-Schritt erzwungen werden, weswegen die **Tick**-Komponente des Zustands auf den Wert **false** gesetzt wird.

$$\frac{\begin{array}{l}\textbf{isParallel}(\textbf{iname}) \\ \wedge\ \neg\ \textbf{hasExecOrSigReadySub}(\textbf{HS}, \textbf{iname}) \\ \wedge\ \textbf{allSubsReadyOrRejected}(\textbf{HS}, \textbf{iname}) \\ \wedge\ \textbf{PL} = \textbf{selectAllReady}(\textbf{HS}, \textbf{iname}) \\ \wedge\ \textbf{hasSelectedSub}(\textbf{HS}, \textbf{iname})\end{array}}{\textbf{iname}\ \ \sigma, \textbf{Tick}(\sigma[\textbf{iname}, \texttt{activated}][\textbf{PL}, \textbf{iname}]_{ac})_{\texttt{Con}}\ \ \textbf{iname}}$$

Listing 9.24: Aktivierung der Unterpläne

Abwarten der Bereitschaft der Unterpläne

Sind die Unterpläne eines `parallel`-kontrollierten Plans (**isParallel**) noch nicht in der `Execution`-Phase (**hasExecutedSub**) und sind auch noch nicht alle Unterpläne im Zustand `ready` oder im Zustand `rejected` (**allSubsReadyOrRejected**), dann soll der Plan abwarten, falls einige Unterpläne in der `Selection`-Phase sind (**hasSelectedSub**).

Durch das passive Abwarten wird der Zustand nicht verändert. Dadurch ist die Ausführung eines Makro-Schrittes möglich.

$$\frac{\begin{array}{l}\mathbf{isParallel}(\mathbf{iname})\\ \wedge\ \neg\ \mathbf{hasExecutedSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \neg\ \mathbf{allSubsReadyOrRejected}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \mathbf{hasSelectedSub}(\mathbf{HS},\mathbf{iname})\end{array}}{\mathbf{iname}\ \xrightarrow{\sigma,\sigma[\mathbf{iname},\ \mathtt{activated}]}_{\mathtt{Con}}\ \mathbf{iname}}$$

Listing 9.25: Abwarten der Bereitschaft der Unterpläne

Starten der Unterpläne

Hat ein `parallel`-kontrollierter Plan (**isParallel**) weder Unterpläne in der `Execution`-Phase (**hasExecutedSub**) noch in der `Selection`-Phase (**hasSelectedSub**), dafür aber Pläne im Zustand `inactive` (**hasInactiveSub**), so sollen diese gestartet werden. Dazu werden alle Unterpläne des Plans an die Planlistenvariable **PL** gebunden und ein Asbru-System gestartet, dass genau die Pläne aus der Liste **PL** enthält (**planNames**).

Alle Pläne der Planliste werden in den Zustand `considered` versetzt und im Folgenden parallel zum Plan ausgeführt. Eine Zuweisung des Wertes **false** an die **Tick** Komponente des Zustands erzwingt einen Mikro-Schritt

$$\frac{\begin{array}{l}\mathbf{isParallel}(\mathbf{iname})\\ \wedge\ \neg\ \mathbf{hasExecutedSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \neg\ \mathbf{hasSelectedSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \mathbf{hasInactiveSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \mathbf{PL} = \mathtt{asbru}(\mathbf{nameSel}(\mathbf{iname}))\mathbf{.subplans}\\ \wedge\ \mathbf{planNames}(\mathbf{PL},\mathbf{iname},\mathbf{sys})\end{array}}{\mathbf{iname}\ \xrightarrow{\sigma,\mathbf{Tick}(\sigma[\mathbf{iname},\ \mathtt{activated}][\mathbf{PL},\ \mathbf{iname},\ \mathtt{considered}])}_{\mathtt{Con}}\ \mathbf{iname}\|_s\ \mathbf{sys}}$$

Listing 9.26: Starten der Unterpläne

Kein Unterplan

Hat ein `parallel`-kontrollierter Plan (**isParallel**) weder Pläne in der `Execution`-Phase (**hasExecutedSub**), noch Pläne in der `Selection`-Phase (**hasSelectedSub**) oder Pläne im Zustand `inactive` (**hasInactiveSub**), so wartet er passiv ab.

Dabei wird der Zustand nicht verändert und Makro-Schritte sind möglich.

$$\frac{\begin{array}{l}\mathbf{isParallel}(\mathbf{iname})\\ \wedge\ \neg\ \mathbf{hasExecutedSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \neg\ \mathbf{hasSelectedSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \neg\ \mathbf{hasInactiveSub}(\mathbf{HS},\mathbf{iname})\end{array}}{\mathbf{iname}\ \xrightarrow{\sigma,\sigma[\mathbf{iname},\ \mathtt{activated}]}_{\mathtt{Con}}\ \mathbf{iname}}$$

Listing 9.27: Kein Unterplan

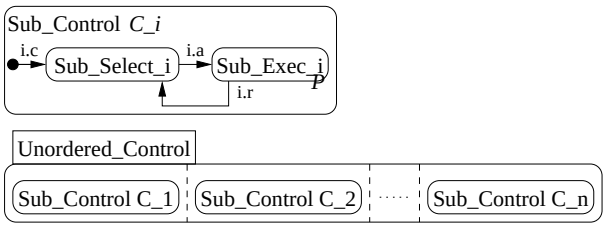


Abbildung 9.11: Zustandsübergänge des `unordered` Kontrolltyps

9.2.10 Zustandsübergänge der `unordered` Kontrolle

Die `unordered` Kontrolle verlangt, dass alle Pläne so schnell wie möglich gestartet und aktiviert werden sollen. Das bedeutet, dass keine Synchronisation der Unterpläne im Zustand `ready` notwendig ist. Damit kann allen Plänen, die gestartet werden, sofort das Aktivierungssignal mitgegeben werden. Damit ergeben sich zur Umsetzung der Semantik nur zwei SOS-Regeln. Die Eine beschreibt den Start der Unterpläne und das Senden der Aktivierungssignale, die zweite Regel beschreibt die Passivität, falls keine Aktivierung notwendig ist. Dies wird in Abbildung 9.11 dargestellt.

Starten von Unterplänen

Ein `unordered` kontrollierter Plan (**isUnordered**), der startbare Unterpläne hat (**hasStartableSub**) soll diese starten. Dazu wird die Menge aller startbaren Pläne mit der Funktion **selectAllStartable** ausgewählt und an die Variable **PL** gebunden. Dann wird ein Asbru-System gewählt, dass genau die Pläne der Planliste **PL** enthält (**planNames**).

Dieses System wird dann parallel zum Plan ausgeführt. Alle Pläne aus der Liste **PL** werden in den Zustand `considered` versetzt. Außerdem werden diese Pläne aktiviert. Ein Makro-Schritt wird durch setzen der **Tick**-Komponente des Zustands auf den Wert **false** verhindert.

$$\frac{\begin{array}{l}\mathbf{isUnordered}(\mathbf{iname}) \\ \wedge\ \mathbf{hasStartableSub}(\mathbf{HS},\mathbf{iname}) \\ \wedge\ \mathbf{PL} = \mathbf{selectAllStartable}(\mathbf{HS},\mathbf{iname}) \\ \wedge\ \mathbf{planNames}(\mathbf{PL},\mathbf{iname},\mathbf{sys})\end{array}}{\mathbf{iname}\ \ \sigma,\mathbf{Tick}(\sigma[\mathbf{iname},\ \mathtt{activated}][\mathbf{PL},\ \mathbf{iname},\ \mathtt{considered}][\mathbf{PL},\mathbf{iname}]_{ac})\ \xrightarrow{}_{\mathtt{Con}}\ \mathbf{iname}\|_s\ \mathbf{sys}}$$

Listing 9.28: Starten von Unterplänen

Kein startbarer Unterplan

Hat ein `unordered` kontrollierter Plan (**isUnordered**) keinen startbaren Unterplan (**hasStartableSub**), so soll er passiv abwarten. Dadurch wird der Zustand nicht verändert und ein Makro-Schritt nicht ausgeschlossen.

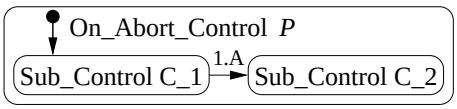


Abbildung 9.12: Zustandsübergänge des `onabort` Kontrolltyps

$$\frac{\begin{array}{c}\mathbf{isUnordered}(\mathbf{iname}) \\ \wedge\, \neg\ \mathbf{hasStartableSub}(\mathbf{HS}, \mathbf{iname})\end{array}}{\mathbf{iname}\ \ \sigma, \sigma[\mathbf{iname},\ \mathtt{activated}]_{\mathtt{Con}}\ \ \mathbf{iname}}$$

Listing 9.29: Kein startbarer Unterplan

9.2.11 Zustandsübergänge der `onabort` Kontrolle

Ein `onabort`-Plan steuert die Ausführung von genau zwei Unterplänen. Der erste Unterplan wird ausgeführt, als wäre der übergeordnete Plan vom Typ `sequential`. Bricht der erste Unterplan seine Bearbeitung ab und ändert seinen Zustand zum Wert `aborted` ab, wird der zweite Plan gestartet. Das Flag, das angibt, ob ein Plan bei Abbruch neu gestartet werden soll, wird ausschließlich für den zweiten Plan ausgewertet. Dies wird in Abbildung 9.12 dargestellt, wobei die Bedingung zum Ausführen der Transition *1.A* ist, dass der von C_1 kontrollierte Plan abgebrochen wurde. Ein Neustart des ersten Planes ist ausgeschlossen.

Aktivierung des ersten Plans

Bei einem `onabort`-kontrolliertem Plan (**isOnAbort**) wird der erste Unterplan gestartet und aktiviert, sofern dieser im Zustand `inactive` ist (**isInactive**). Der Instanzname des ersten Unterplans (**selectFirstSub**) wird an die Variable $\mathbf{iname}_1$ gebunden und dann diese Planinstanz parallel zum Plan ausgeführt. Dazu wird der Planzustand dieses Plans auf den Wert `considered` gesetzt und der Plan aktiviert.

Durch Umsetzen der **Tick**-Komponente des Zustands wird durch diese Regel ein Makro-Schritt verhindert.

$$\frac{\begin{array}{c}\mathbf{isOnAbort}(\mathbf{iname}) \\ \wedge\, \mathbf{isInactive}(\mathbf{HS}, \mathbf{iname}_1) \\ \wedge\, \mathbf{iname}_1 = \mathbf{nameGen}(\mathbf{selectFirstSub}(\mathbf{iname}), \mathbf{iname})\end{array}}{\mathbf{iname}\ \ \sigma, \mathbf{Tick}(\sigma[\mathbf{iname},\ \mathtt{activated}][\mathbf{iname}_1,\ \mathtt{considered}][\mathbf{iname}_1]_{ac})_{\mathtt{Con}}\ \ \mathbf{iname}\|_s\ \mathbf{iname}_1}$$

Listing 9.30: Aktivierung des ersten Plans

Aktivierung des zweiten Plans

Wurde bei einem `onabort`-kontrolliertem Plan (**isOnAbort**) der erste Unterplan abgebrochen (**isAborted**) und ist der zweite Unterplan start- oder neustartbar (**isStartable**), so

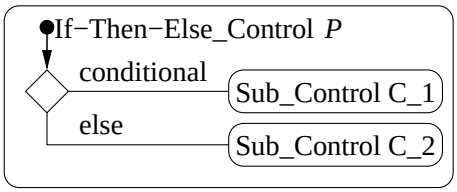


Abbildung 9.13: Zustandsübergänge des `ifThenElse` Kontrolltyps

wird der Instanzname des zweiten Unterplans (**selectSecondSub**) an die Variable $\mathbf{iname}_1$ gebunden.

Dieser Plan wird dann aktiviert und gestartet. Der Zustand des Unterplans wird auf den Wert `considered` gesetzt. Durch Umsetzen der **Tick**-Komponente des Zustands wird durch Anwendung dieser Regel ein Makro-Schritt verhindert.

$$\frac{\begin{array}{l}\quad \mathbf{isOnAbort}(\mathbf{iname}) \\ \wedge\ \mathbf{isAborted}(\mathbf{HS}, \mathbf{selectFirstSub}(\mathbf{iname})) \\ \wedge\ \mathbf{isStartable}(\mathbf{HS}, \mathbf{selectSecondSub}(\mathbf{iname})) \\ \wedge\ \mathbf{iname}_1 = \mathbf{nameGen}(\mathbf{selectSecondSub}(\sigma, \mathbf{iname}), \mathbf{iname})\end{array}}{\mathbf{iname}\ \ \sigma, \mathtt{Tick}(\sigma[\mathbf{iname},\ \mathtt{activated}][\mathbf{iname}_1,\ \mathtt{considered}][\mathbf{iname}_1]_{ac})\ \ {}_{\mathtt{Con}}\ \ \mathbf{iname}\|_s\ \mathbf{iname}_1}$$

Listing 9.31: Aktivierung des zweiten Plans

Kein startbarer Unterplan

Ist der erste Unterplan (**selectFirstSub**) eins `onabort`-kontrolliert Plans (**isOnAbort**) nicht im Zustand `inactive` (**isInactive**) bzw. der erste Unterplan nicht abgebrochen (**isAborted**) oder der zweite Unterplan (**selectSecondSub**) nicht startbar (**isStartable**), so kann die `onabort` Kontrolle keine Änderung des Zustandes mehr bewirken und bleibt passiv. Dadurch ist ein Makro-Schritt im Anschluss an die Abarbeitung dieses Falles möglich.

$$\frac{\begin{array}{l}\quad \mathbf{isOnAbort}(\mathbf{iname}) \\ \wedge\ \neg\ \mathbf{isInactive}(\mathbf{HS}, \mathbf{selectFirstSub}(\mathbf{iname})) \\ \wedge\quad \neg\ \mathbf{isAborted}(\mathbf{HS}, \mathbf{selectFirstSub}(\mathbf{iname})) \\ \quad \vee\ \neg\ \mathbf{isStartable}(\mathbf{HS}, \mathbf{selectSecondSub}(\mathbf{iname}))\end{array}}{\mathbf{iname}\ \ \sigma, \sigma[\mathbf{iname}, \underset{\longrightarrow}{\ }\mathtt{activated}]_{\mathtt{Con}}\ \ \mathbf{iname}}$$

Listing 9.32: Kein startbarer Unterplan

9.2.12 Zustandsübergänge der `ifThenElse` Kontrolle

Der `ifThenElse` Kontrolltyp überprüft vor dem Start eines Unterplans den Wahrheitswert eines Konditionals und startet daraufhin in Abhängigkeit des Wahrheitswertes des Konditio-

nals einen der möglichen Unterpläne. Ein `ifThenElse` kontrollierter Plan muss genau einen oder genau zwei Unterpläne haben. Ist kein zweiter Unterplan spezifiziert, so verhält sich der `ifThenElse` Kontrolltyp wie ein wenn-dann Konstrukt und schließt die Bearbeitung erfolgreich ab, wenn das Konditional bei der Auswertung falsch ist.

Um festzustellen, ob das Konditional bereits ausgewertet wurde, prüft die `ifThenElse` Kontrolle, ob bereits einer der Pläne den Zustand `inactive` verlassen hat. Dies ist gleichbedeutend damit, dass bereits in einem vorhergehenden Schritt das Konditional ausgewertet wurde und dementsprechend Pläne aktiviert wurden.

Positive Auswertung des Konditionals

Sind alle Unterpläne eines `ifThenElse`-kontrollierten Plans (**isITE**) im Zustand `inactive` (**allSubsInactive**) und wertet das Konditional des Plantyps positiv aus (**ITEP**), so soll der erste Unterplan (**selectFirstSub**) des Plans gestartet und aktiviert werden. Dazu wird dessen Name an die Variable $\mathbf{iname}_1$ gebunden, parallel zum Plan ausgeführt, der Zustand auf `considered` gesetzt und das Aktivierungssignal gesendet. Weiterhin soll die **Tick**-Komponente des Zustands auf den Wert **false** gesetzt werden und so ein Mikro-Schritt erzwungen.

$$\frac{\begin{array}{l}\mathbf{isITE}(\mathbf{iname})\\ \wedge\ \mathbf{allSubsInactive}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \mathbf{ITEP}(\sigma,\mathbf{iname})\\ \wedge\ \mathbf{iname}_1 = \mathbf{nameGen}(\mathbf{selectFirstSub}(\mathbf{iname}),\mathbf{iname})\end{array}}{\mathbf{iname}\ \xrightarrow{\sigma,\mathbf{Tick}(\sigma[\mathbf{iname},\ \mathtt{activated}][\mathbf{iname}_1,\ \mathtt{considered}][\mathbf{iname}_1]_{ac})}_{\mathtt{Con}}\ \mathbf{iname}\|_s\ \mathbf{iname}_1}$$

Listing 9.33: Positive Auswertung des Konditionals

Negative Auswertung des Konditionals mit zweitem Unterplan

Sind alle Unterpläne eines `ifThenElse`-kontrollierten Plans (**isITE**) im Zustand `inactive` (**allSubsInactive**) und wertet das Konditional des Plantyps negativ aus (**ITEP**), so soll, falls ein zweiter Unterplan existiert (**secondSubPresent**), dieser (**selectSecondSub**) gestartet und aktiviert werden. Dazu wird dessen Name an die Variable $\mathbf{iname}_1$ gebunden, parallel zum Plan ausgeführt, der Zustand auf `considered` gesetzt und das Aktivierungssignal gesendet. Weiterhin soll die **Tick**-Komponente des Zustands auf den Wert **false** gesetzt werden und so ein Mikro-Schritt erzwungen.

$$\frac{\begin{array}{l}\mathbf{isITE}(\mathbf{iname})\\ \wedge\ \mathbf{allSubsInactive}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \neg\ \mathbf{ITEP}(\sigma,\mathbf{iname})\\ \wedge\ \mathbf{secondSubPresent}(\mathbf{iname})\\ \wedge\ \mathbf{iname}_1 = \mathbf{nameGen}(\mathbf{selectSecondSub}(\mathbf{iname}),\mathbf{iname})\end{array}}{\mathbf{iname}\ \xrightarrow{\sigma,\mathbf{Tick}(\sigma[\mathbf{iname},\ \mathtt{activated}][\mathbf{iname}_1,\ \mathtt{considered}][\mathbf{iname}_1]_{ac})}_{\mathtt{Con}}\ \mathbf{iname}\|_s\mathbf{iname}_1}$$

Listing 9.34: Negative Auswertung des Konditionals mit zweitem Unterplan

Negative Auswertung des Konditionals ohne zweitem Unterplan

Sind alle Unterpläne eines `ifThenElse`-kontrollierten Plans (**isITE**) im Zustand `inactive` (**allSubsInactive**) und wertet das Konditional des Plantyps negativ aus (**ITEP**), so soll sich der Plan auf den Zustand `completed` setzen und die Ausführung einstellen, falls kein zweiter Unterplan existiert (**secondSubPresent**). In diesem Fall soll kein Makro-Schritt möglich sein, weswegen die **Tick**-Komponente des Zustands auf den Wert **false** gesetzt wird.

$$\frac{\begin{array}{l}\quad \mathbf{isITE}(\mathbf{iname}) \\ \wedge\ \mathbf{allSubsInactive}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \neg\ \mathbf{ITEP}(\sigma, \mathbf{iname}) \\ \wedge\ \mathbf{noSecondSubPresent}(\mathbf{iname})\end{array}}{\mathbf{iname}\ \ \sigma, \mathbf{Tick}(\sigma[\mathbf{iname}, \xrightarrow{}\ \texttt{completed}])_{\texttt{Con}}\ \ \varepsilon}$$

Listing 9.35: Negative Auswertung des Konditionals ohne zweitem Unterplan

Aktivierter Unterplan

Wurden bereits Unterpläne eines `ifThenElse`-Plantyps (**isITE**) gestartet (**allSubsInactive**) und befinden sich Unterpläne in der `Execution`-Phase (**hasExecutedSub**) oder in der `Selection`-Phase (**hasSelectedSub**) bzw. existiert kein Unterplan, der abgebrochen wurde und neu gestartet werden soll (**hasRestartableSub**), dann soll der Plan passiv abwarten. In diesem Fall sind Makro-Schritte möglich.

$$\frac{\begin{array}{l}\quad \mathbf{isITE}(\mathbf{iname}) \\ \wedge\ \neg\ \mathbf{allSubsInactive}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \ \mathbf{hasExecutedSub}(\mathbf{HS}, \mathbf{iname}) \\ \vee\ \mathbf{hasSelectedSub}(\mathbf{HS}, \mathbf{iname}) \\ \vee\ \neg\ \mathbf{hasRestartableSub}(\mathbf{HS}, \mathbf{iname})\end{array}}{\mathbf{iname}\ \ \sigma, \sigma[\mathbf{iname}, \xrightarrow{}\ \texttt{activated}]_{\texttt{Con}}\ \ \mathbf{iname}}$$

Listing 9.36: Aktivierter Unterplan

Neustart eines Unterplans

Sind nicht alle Unterpläne eines `ifThenElse`-Plans (**isITE**) im Zustand `inactive` (**allSubsInactive**) und existieren keine Unterpläne in `Execution`-Phase (**hasExecutedSub**) oder `Selection`-Phase (**hasSelectedSub**), dafür aber Pläne, die abgebrochen wurden und neu gestartet werden sollen (**hasRestartableSub**), so wird der Instanzname des ersten neuzustartenden Plans (**selectFirstRestartable**) an die Variable $\mathbf{iname}_1$ gebunden. Dieser Plan wird dann gestartet, aktiviert und parallel zum Plan ausgeführt. Der Zustand des Unterplans wird auf den Wert `considered`. Durch Umsetzen der **Tick**-Komponente des Zustands wird ein Mikro-Schritt erzwungen.

$$\frac{\begin{array}{l}\mathbf{isITE}(\mathbf{iname})\\ \wedge\,\neg\ \mathbf{allSubsInactive}(\mathbf{HS},\mathbf{iname})\\ \wedge\,\neg\ \mathbf{hasExecutedSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\,\neg\ \mathbf{hasSelectedSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \mathbf{hasRestartableSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \mathbf{iname}_1 = \mathbf{nameGen}(\mathbf{selectFirstRestartable}(\mathbf{iname}),\mathbf{iname})\end{array}}{\mathbf{iname}\ {}^{\sigma,\mathbf{Tick}(\sigma[\mathbf{iname},\ \mathtt{activated}][\mathbf{iname}_1,\ \mathtt{considered}])}{\longrightarrow}_{\mathtt{Con}}\ \mathbf{iname}\|_s\ \mathbf{iname}_1}$$

Listing 9.37: Neustart eines Unterplans

9.2.13 Zustandsübergänge der ask Kontrolle

Die ask Kontrolle nimmt unter den verschiedenen Kontrolltypen eine Sonderrolle ein. Ein ask kontrollierter Plan (**isASK**) startet keine Unterpläne. Stattdessen dient ein solcher Plan lediglich als Signal an die Umgebung des Asbru Systems, etwa einen Arzt, der auf die Anforderung des Systems nach Daten mit einer Messung und Dateneingabe reagieren kann oder auch nicht. Ein ask kontrollierter Plan sollte in seiner Abschlussbedingung spezifizieren, wie lange das Signal an die Umgebung aktiv bleibt. Daher muss der ask Kontrolltyp selbst nur passiv abwarten. Somit steht die Abarbeitung eines aktiven ask Planes auch keinem Makro-Schritt im Wege.

$$\frac{\mathbf{isASK}(\mathbf{iname})}{\mathbf{iname}\ {}^{\sigma,\sigma[\mathbf{iname},\ \mathtt{activated}]}{\longrightarrow}_{\mathtt{Con}}\ \mathbf{iname}}$$

Listing 9.38: Zustandsübergang der ask Kontrolle

9.2.14 Zustandsübergänge der user Kontrolle

Die user-Kontrolle (**isUser**) beschreibt, wie die ask-Kontrolle, die Ausführung von Plänen, die die Blätter in der Asbru Hierarchie darstellen. user-Pläne haben keine eigenen Unterpläne und verhalten sich daher in ihrer Ausführung ähnlich den ask Plänen rein passiv und warten lediglich auf die Erfüllung der jeweiligen **complete**-Bedingung. Wie die ask Pläne stellen user Pläne Signale des Asbru-Systems an die Umgebung dar. Während jedoch die ask kontrollierten Pläne das medizinische Personal zu einer Messung bzw. der Eingabe von Patientenwerten auffordern, signalisieren die user kontrollierten Pläne dem medizinischen Personal, dass eine bestimmte medizinische Interaktion durchgeführt werden soll. Da user kontrollierte Pläne sich jedoch technisch rein passiv verhalten, steht die Abarbeitung eines aktiven user Planes keinem Makro-Schritt im Wege.

$$\frac{\mathbf{isUser}(\mathbf{iname})}{\mathbf{iname}\ {}^{\sigma,\sigma[\mathbf{iname},\ \mathtt{activated}]}{\longrightarrow}_{\mathtt{Con}}\ \mathbf{iname}}$$

Listing 9.39: Zustandsübergang der user Kontrolle

9.2.15 Zustandsübergänge der `cyclical` Kontrolle

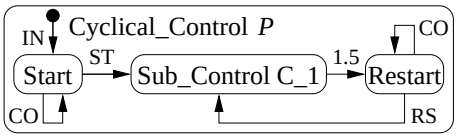


Abbildung 9.14: Zustandsübergänge des `cyclical`-Kontrolltyps

Die `cyclical` Kontrolle hat genau einen Unterplan. Sie definiert ein zyklisch wiederkehrendes Startinterval, in dem der Unterplan gestartet werden darf. Der Unterplan wird während eines Startintervalls gestartet, sofern er in einem terminierten Zustand oder im Zustand `inactive` ist. Der Kontrolltyp definiert zusätzlich eine Anzahl von erforderlichen erfolgreichen Durchläufen, die der Unterplan erreichen muss, damit der übergeordnete Plan abgeschlossen werden darf. Der übergeordnete Plan überprüft bei Terminierung des Unterplanes, ob dieser abgeschlossen wurde. Falls ja, wird ein Rundenzähler hochgesetzt und mit der Zahl der erforderlichen Durchläufe verglichen. In Abhängigkeit von der Auswertung dieser Überprüfung setzt sich der übergeordnete Plan auf den Zustand `completed` oder startet einen neuen Durchlauf. Dieses Verhalten wird durch Abbildung 9.14 visualisiert.

Start des Unterplans

Ist der Unterplan eines `cyclical`-Plans (**isCyclical**) im Zustand `inactive` (**isInactive**), `rejected` (**isRejected**) oder `aborted` (**isAborted**), so darf er ohne weitere Prüfungen (neu) gestartet werden, sobald das Startintervall erreicht ist (**isInTimeFrame**). Dazu wird der Name des Unterplans (**selectFirstSub**) an die Variable $\mathbf{iname}_1$ gebunden. Der Unterplan wird gleichzeitig gestartet und aktiviert. Außerdem wird der Planzustand des Plans auf den Wert `considered` gesetzt. Durch Umsetzen der **Tick**-Komponente des Zustands wird ein Mikro-Schritt erzwungen.

$$\frac{\begin{array}{l}\mathbf{isCyclical}(\mathbf{iname}) \\ \wedge\ \ \mathbf{isInactive}(\mathbf{HS}, \mathbf{iname}_1) \\ \quad \vee\ \mathbf{isRejected}(\mathbf{HS}, \mathbf{iname}_1) \\ \quad \vee\ \mathbf{isAborted}(\mathbf{HS}, \mathbf{iname}_1) \\ \wedge\ \mathbf{isInTimeFrame}(\sigma, \mathbf{iname}) \\ \wedge\ \mathbf{iname}_1 = \mathbf{nameGen}(\mathbf{selectFirstSub}(\mathbf{iname}), \mathbf{iname})\end{array}}{\mathbf{iname} \xrightarrow{\sigma, \mathbf{Tick}(\sigma[\mathbf{iname},\ \texttt{activated}][\mathbf{iname}_1,\ \texttt{considered}][\mathbf{iname}_1]_{ac})}_{\texttt{Con}} \mathbf{iname} \|_s \mathbf{iname}_1}$$

Listing 9.40: Start des Unterplans

Beendigung des letzten Durchlaufes

Wurde der Unterplan (**selectFirstSub**) eines `cyclical`-Plans (**isCyclical**) abgeschlossen (**isCompleted**), so muss der übergeordnete Plan überprüfen, ob die Zahl der notwendigen erfolgreichen Durchläufe des Unterplans damit erreicht wurde (**hasCompletedRounds**). Ist dies der Fall, so kann der übergeordnete Plan ebenfalls seine Bearbeitung abschließen. Der

Zustand des Unterplanes wird dabei nicht weiter verändert. Ein Mikro-Schritt wird durch Umsetzen der **Tick**-Komponente des Zustands erzwungen.

$$\frac{\begin{array}{l}\mathbf{isCyclical}(\mathbf{iname})\\ \wedge\ \mathbf{isCompleted}(\mathbf{HS}, \mathbf{selectFirstSub}(\mathbf{iname}))\\ \wedge\ \mathbf{hasCompletedRounds}(\mathbf{HS}, \mathbf{iname})\end{array}}{\mathbf{iname}\ \ \sigma, \mathbf{Tick}(\sigma[\mathbf{iname}, \texttt{completed}])\ \ \mathbf{iname}}\ \texttt{Con}$$

Listing 9.41: Beendigung des letzten Durchlaufes

Neustart nach einem erfolgreichen Durchlauf

Wurde der Unterplan (**selectFirstSub**) eines `cyclical` Plans (**isCyclical**) abgeschlossen (**isCompleted**), ohne dass dabei die notwendige Zahl der Durchläufe erreicht wurde (**hasCompletedRounds**), so muss der übergeordnete Plan den Unterplan neu starten. Dabei ist das wiederkehrende Startintervall zu beachten (**isInTimeFrame**), innerhalb dessen der Unterplan neu gestartet werden kann. Beim Neustart wird der Rundenzähler des übergeordneten Plans hochgezählt, der die Anzahl der erfolgreichen Durchläufe des Unterplans speichert. Weiterhin wird der Planzustand des Unterplans auf den Wert `considered` gesetzt, der Unterplan aktiviert und parallel zum Plan ausgeführt. Ein Umsetzen der **Tick** Komponente verhindert einen Makro-Schritt.

$$\frac{\begin{array}{l}\mathbf{isCyclical}(\mathbf{iname})\\ \wedge\ \mathbf{iname}_1 = \mathbf{nameGen}(\mathbf{selectFirstSub}(\mathbf{iname}), \mathbf{iname})\\ \wedge\ \mathbf{isCompleted}(\mathbf{HS}, \mathbf{iname}_1)\\ \wedge\ \neg\ \mathbf{hasCompletedRounds}(\mathbf{HS}, \mathbf{iname})\\ \wedge\ \mathbf{isInTimeFrame}(\sigma, \mathbf{iname})\end{array}}{\begin{array}{l}\mathbf{iname}\ \ \sigma, \mathbf{Tick}(\sigma\ [\mathbf{iname}, \texttt{activated}][\mathbf{iname}_1, \texttt{considered}]\\ \qquad\qquad [\mathbf{iname}, \sigma[\mathbf{iname}].\mathbf{rounds} + 1])\\ \qquad\qquad\qquad\longrightarrow\qquad\qquad\qquad \texttt{Con}\ \ \mathbf{iname}\|_s\ \mathbf{iname}_1\end{array}}$$

Listing 9.42: Neustart nach einem erfolgreichen Durchlauf

Passives Abwarten

Es gibt mehrere Bedingungen, unter denen ein `cyclical`-Plan (**isCyclical**) passiv auf ein Zustandsänderung durch Umgebung oder seinen Unterplan wartet. Diese Bedingungen werden mit der letzten Regel geprüft.

Der `cyclical`-Plan verhält sich passiv, wenn der Unterplan zwar in terminierten (**isTerminated**) oder inaktivem (**isInactive**) Zustand ist, aber das Startintervall derzeit nicht erreicht ist (**isInTimeFrame**). Außerdem muss gelten, dass, falls der Unterplan beendet wurde (**isCompleted**), die notwendige Zahl von Durchläufen noch nicht erreicht wurde (**hasCompletedRounds**).

Davon abgesehen würde der Plan stets passiv bleiben, wenn der Unterplan gerade in der `Selection`-Phase oder der `Execution`-Phase ist. Dies wird in der Vorbedingung implizit

modelliert. Wird der Unterplan ausgeführt, so können die Vorbedingungen der Implikationen nicht erfüllt werden. Somit sind die Implikationen in diesem Fall automatisch wahr.

In all diesen Fällen würde der übergeordnete Plan nur passiv abwarten. Dabei wird die Ausführung eines Makro-Schrittes nicht verhindert.

$$\frac{\begin{array}{l}\quad \mathbf{isCyclical}(\mathbf{iname}) \\ \wedge \quad \mathbf{isTerminated}(\mathbf{HS}, \mathbf{selectFirstSub}(\mathbf{iname})) \\ \quad\ \vee \mathbf{isInactive}(\mathbf{HS}, \mathbf{selectFirstSub}(\mathbf{iname})) \\ \rightarrow \neg\ \mathbf{isInTimeFrame}(\sigma, \mathbf{iname}) \\ \wedge \quad \mathbf{isCompleted}(\mathbf{HS}, \mathbf{selectFirstSub}(\mathbf{iname})) \\ \rightarrow \neg\ \mathbf{hasCompletedRounds}(\mathbf{HS}, \mathbf{iname})\end{array}}{\mathbf{iname}\ \overset{\sigma, \sigma[\mathbf{iname}, \underrightarrow{\ } \mathtt{activated}]}{\longrightarrow}_{\mathtt{Con}}\ \mathbf{iname}}$$

Listing 9.43: Passives Abwarten

9.3 Zustandsübergänge für Planhierarchien

Werden komplette Planhierarchien ausgeführt, so geschieht dies durch die Berechnung der jeweiligen einzelnen Planzustandsübergänge und ein Zusammenschalten der Ergebnis durch nachstehende Regel. Dabei werden die Ergebnisse mit dem Prädikat sync zusammengeführt. Dieses überprüft, ob die Zustandsübergänge der Planhierarchien kompatibel zueinander sind. Dies ist der Fall, wenn keine Schreibkonflikte vorliegen.

$$\frac{\begin{array}{c}\mathbf{sys}_1\ \overset{\sigma, \sigma_1}{\longrightarrow}\ \mathbf{sys}_1' \\ \wedge\ \mathbf{sys}_2\ \overset{\sigma, \sigma_2}{\longrightarrow}\ \mathbf{sys}_2' \\ \wedge\ \mathtt{sync}(\sigma, \sigma_1, \sigma_2, \sigma')\end{array}}{\mathbf{sys}_1 \|_s\ \mathbf{sys}_2\ \overset{\sigma, \sigma'}{\longrightarrow}\ \mathbf{sys}_1{}'\ \|_s\ \mathbf{sys}_2{}'}$$

Dabei ist der Spezialfall zu beachten, in dem eine Komponente des Asbru-Systems terminiert. Das terminierte Programm kann keine weiteren Zustandsübergange mehr ausführen. Daher wird es aus dem auszuführenden System entfernt.

$$\frac{\mathbf{sys}\ \overset{\sigma, \sigma'}{\longrightarrow}\ \mathbf{sys}'}{\mathbf{sys} \|_s\ \varepsilon\ \overset{\sigma, \sigma'}{\longrightarrow}\ \mathbf{sys}'}$$

Dies gilt auch für den Fall, in dem die erste Komponente terminiert ist:

$$\frac{\texttt{sys} \xrightarrow{\sigma, \sigma'} \texttt{sys}'}{\varepsilon \|_s \texttt{sys} \xrightarrow{\sigma, \sigma'} \texttt{sys}'}$$

10 Ablaufsemantik von Asbru

Im Folgenden wird die Semantik von Asbru als Ablaufsemantik definiert. Dazu wird zunächst in Abschnitt 10.1 der Begriff des Ablaufs definiert. Danach wird in Abschnitt 10.2 der Modellierungsbegriff beschrieben, der die Semantik von Asbru festlegt. Im Folgenden soll dann noch der Spezialfall geschlossener Asbru-Systeme beschrieben werden. Dazu wird in Abschnitt 10.3 die Mikro- bzw. Makro-Schritt Problematik noch einmal aufgegriffen und eine formale Definition gegeben. In dem darauf folgenden Abschnitt 10.4 wird beschreibt, welche Verhaltensmuster es für die Umgebung eines Asbru-Systems gibt. Darauf folgt eine Definition der initialen Zustände in Abschnitt 10.5. Aufbauend auf diesen Ergebnissen wird dann in Abschnitt 10.6 die Ablaufsemantik von geschlossenen Asbru-Systemen definiert.

10.1 Der Ablaufbegriff

Da die Semantik von Asbru in Form von Abläufen definiert ist, wird zunächst eine Definition von Abläufen gegeben. Die Semantik von Asbru wird dann im Folgenden beschrieben als eine Menge von Abläufen.

Definition

Ein Ablauf ist eine endliche oder unendliche Folge von Zuständen. Sei $\overline{\mathbf{n}}$ aus der Menge der natürlichen Zahlen vereinigt mit Unendlich. Dann lässt sich ein Ablauf schreiben als

$$\Sigma = (\sigma_0, \sigma'_0, \sigma''_0, \sigma'_1 ..., \sigma'_{\overline{\mathbf{n}}-1}, \sigma''_{\overline{\mathbf{n}}-1})$$

Dabei besteht ein Ablauf Σ aus einem initialen Zustand σ_0 sowie einer Abfolge von Transitionen $(\sigma'_i,\ \sigma''_i)_{i=0}^{\overline{\mathbf{n}}-1}$ In einem Ablauf gilt stets für alle zweigestrichenen Zustand σ''_i dass $\sigma''_i = \sigma_{i+1}$. Die Länge eines Ablaufs wird bestimmt durch $\mid \Sigma \mid$. Die Länge eines Ablaufs $\mid (\sigma_0, \sigma'_0, \sigma''_0, \sigma'_1 ..., \sigma'_{\overline{\mathbf{n}}-1}, \sigma''_{\overline{\mathbf{n}}-1}) \mid$ ist $\overline{\mathbf{n}}$.

Ein Ablauf ist also eine möglicherweise unendliche Abfolge von Transitionen. Transitionen sind Paare von Zuständen. Diese paarweise Anordnung entspricht der, der TL-Intervalle. Wie dort werden die Transitionen zwischen eingestrichenen und zweigestrichenen Zuständen durch die Umgebung kontrolliert und die Übergänge zwischen ungestrichenen und eingestrichenen Zuständen durch das System. Die Intuition eines Asbru-Ablaufes ist, dass die Übergänge des Systems durch die SOS-Regeln aus Abschnitt 9 beschrieben werden. Umgebungsübergänge sollen die Handlungen des Arztes sowie die Auswirkungen dieser Handlungen auf den Patienten modellieren. Was dies genau bedeutet, wird in Abschnitt 10.3 beschrieben.

Definition

Um auf den i-ten Zustand eines Ablaufes zuzugreifen, wird der Term $\Sigma(i)$ verwendet. Sei Σ ein Ablauf mit $\Sigma = (\sigma_0, \sigma'_0, \sigma''_0, \sigma'_1 ..., \sigma'_{\overline{\mathbf{n}}-1}, \sigma''_{\overline{\mathbf{n}}-1})$. Dann ist $\Sigma(i) = \sigma_i$, weiterhin ist $\Sigma(i)' = \sigma'_i$ und $\Sigma(i)'' = \Sigma(i+1)$, falls $i < \overline{\mathbf{n}}$, andernfalls ist $\Sigma(i) = \Sigma(i)' = \sigma_{\overline{\mathbf{n}}}$

Ein Schritt eines Ablaufs bezeichnet die beiden Transitionen vom ungestrichenen zum eingestrichenen und vom eingestrichenen zum zweigestrichenen Zustand. Der i-te Schritt ist definiert als $(\Sigma(i), \Sigma(i)', \Sigma(i)'')$.

10.2 Modellierung durch Asbru-Systeme

Definition

Ein Ablauf Σ modelliert eine Asbru-System **sys**, geschrieben $\Sigma \models$ **sys**, wenn gilt, dass alle Zustandsübergänge zwischen ungestrichenen und eingestrichenen Variablen durch die SOS Regeln gebildet wurden. Dazu muss es eine Reihe von Asbru-Systemen $(\mathbf{sys}_i)_{i=0}^{|\Sigma|}$ geben. Für das erste Asbru-System, $\mathbf{sys}_0$ muss gelten, dass $\mathbf{sys}_0 = \mathbf{sys}$. Weiterhin gilt für die Übergänge, dass die Transitionen

$$\mathbf{sys}_i \xrightarrow{\Sigma(\mathrm{i}),\Sigma(\mathrm{i})'} \mathbf{sys}_{i+1}$$

jeweils durch die SOS Regeln gebildet werden können.

Ein Ablauf modelliert also ein Asbru-System, wenn die jeweiligen Systemübergänge des Ablaufs genau diejenigen sind, die vom Asbru-System bzw. den SOS Regeln vorgeschrieben werden. Das bedeutet aber auch, dass in diesem Fall noch keine Bedingung an die Umgebung gestellt wird. Dies wird im Folgenden gemacht.

10.3 Mikro- und Makro-Schritt

Mikro- und Makro-Übergänge beschreiben Einschränkungen an Zustandsübergänge, die jeweils für System- und Umgebungsübergang getrennt definiert sind. Die Unterscheidung zwischen Mikro- und Makro-Übergängen wird anhand der **Tick** Komponente des gestrichenen Zustands einer Ablauf-Komponente getroffen:

Definition

Ein Schritt, bestehend aus drei Zuständen, $(\sigma, \sigma',\ \sigma'')$ heißt Makro-Schritt, wenn gilt $\sigma'.\mathrm{Tick} = \mathbf{true}$. Der Schritt heißt Mikro-Schritt, wenn gilt, dass $\sigma'.\mathrm{Tick} = \mathbf{false}$.

10.4 Umgebungsübergang

Definition

Ein Umgebungsübergang ist ein Übergang zwischen einfach und zweifach gestrichenen Variablen, der folgenden Einschränkungen unterliegt:

Asbru-Zustands-Historie Die aktuellen Planzustände des Hierarchiezustands werden aus diesem in die Asbru-Zustands-Historie übertragen. Dazu wird für alle Planinstanznamen **iname** der aktuelle Planzustand an den entsprechenden Eintrag der Asbru-Zustands-Historie angehängt:

$$\sigma''.\text{ASH}[\textbf{iname}] = \sigma'.\text{ASH}[\textbf{iname}][\sigma''.\text{AC}, \sigma'.\text{HS}[\textbf{iname}].\textbf{state}]$$

Diese Gleichung wird nun schrittweise erklärt. Zunächst wird der „alte“ Eintrag in die Asbru-Zustands-Historie des betroffenen Plans **iname** selektiert:

$$\sigma'.\text{ASH}[\textbf{iname}]$$

Dieser Eintrag in die Asbru-Zustands-Historie wird dann aktualisiert, in dem die Aktualisierungsfunktion **asbru-state-history-entry**[**int**, **PS**] verwendet wird. Als Eintrag in die Asbru-Zustands-Historie wird der oben selektierte Eintrag gewählt. Der Zeitpunkt, zu dem die Aktualisierung stattfinden soll, ist der Zeitstempel des zweigestrichenen Zustands ($\sigma''.\text{AC}$) und der Wert der Aktualisierung ist der aktuelle Planzustand des Plans **iname** ($\sigma'.\text{HS}[\textbf{iname}].\textbf{state}$).

Das Ergebnis der oben aufgeführten Gleichung ist daher, dass die Asbru-Zustands-Historie des neuen Zustands in allen vergangenen Zeitpunkten identisch ist zu der des alten Zustands. Lediglich im aktuellen Zeitpunkt wird eine Aktualisierung um den Planzustand durchgeführt, der in der Hierarchiezustandskomponente des Zustands enthalten ist.

Hierarchie-Zustand Der Hierarchiezustand bleibt bei Umgebungsübergängen stets unverändert:

$$\sigma_1.\text{HS} = \sigma.\text{HS}$$

Krankenakte Innerhalb von Mikro-Schritten muss die Krankenakte unverändert bleiben, während Makro-Schritten darf sie sich ändern:

$$\sigma.\text{Tick} = \textbf{false} \Rightarrow \sigma_1.\text{PDH} = \sigma.\text{PDH}$$

Umgebungssignalsammlung Innerhalb von Mikro-Schritten muss die Signalsammlung unverändert bleiben, während Makro-Schritten darf sie sich ändern:

$$\sigma.\text{Tick} = \textbf{false} \Rightarrow \sigma_1.\text{EAGG} = \sigma.\text{EAGG}$$

Tick-Komponente Die Tick-Komponente wird während eines Umgebungsüberganges auf den Wert **true** gesetzt.

$$\sigma_1.\text{Tick} = \textbf{true}$$

Uhrzeit Während Mikro-Schritten bleibt die Uhrzeit unverändert. Während Makro-Schritten wird sie hochgezählt:

$$\begin{aligned} &\sigma_1.\text{Tick} = \textbf{true} \Rightarrow \sigma_1.\text{AC} = \sigma.\text{AC} + 1 \\ \wedge\ &\sigma_1.\text{Tick} = \textbf{false} \Rightarrow \sigma_1.\text{AC} = \sigma.\text{AC} \end{aligned}$$

Zusammengefasst müssen zwei „Arten“ von Umgebung unterschieden werden. Zum Einen gibt es quasi ein System über dem Asbru-System, das das Mitschreiben der Asbru-Zustands-Historien-Einträge übernimmt. Zum Anderen existiert die wirkliche Umgebung, die Veränderungen des Patientenverhaltens, Signalwechsel und ein Fortschreiben der Uhr modelliert.

Im Abschnitt 7.1 wird das Zeitmodell von Asbru beschrieben. Dort wird erläutert, wie das Mikro- und Makro-Schritt System von Asbru zu verstehen ist. Da das Asbru-System als beliebig schnell angenommen werden kann, vergeht zwischen zwei Mikro-Schritten praktisch keine Zeit. Daher kann die physische Umgebung zwischen zwei Mikro-Schritten den Zustand nicht beeinflussen. Ein Makro-Schritt findet immer dann statt, wenn das System keine Handlungen mehr durchführen kann und auf Zustandsänderungen der Umgebung warten muss.

Entsprechend sind die Bedingungen an einen Umgebungsübergang formuliert. Während eines Mikro-Schrittes kann die Umgebung das System nicht beeinflussen. Lediglich das „System über dem Asbru-System“, dass das Fortschreiben der Asbru-Zustands-Historie übernimmt, kommt dann zum Zug und protokolliert die aktuellen Zustände mit. Weiterhin wird die **Tick** Komponente des Zustands, die Mikro- bzw. Makro-Schritte anzeigt zurückgesetzt. Während eines Makro-Schrittes hat die Umgebung mehr Möglichkeiten zur Einflussnahme. In diesem Fall dürfen die Umgebungs-Datentypen, also die Signalsammlung und die Krankenakte, beliebig verändert werden dürfen. Weiterhin muss in diesem Fall die Uhr weiterlaufen.

10.5 Initiale Zustände

Definition

Ein Zustand σ heißt initialer Zustand, wenn für alle Planinstanznamen **iname** und alle Uhrzeiten **AC** gilt:

- $\mathbf{AC} \neq \sigma.\mathrm{AC} \Rightarrow \sigma.\mathrm{ASH}[\mathbf{iname}][\mathbf{AC}] = ([])$
- $\sigma.\mathrm{HS}[\mathbf{iname}].\mathbf{rounds} = 0$
- $\sigma.\mathrm{Tick} \Leftrightarrow \mathbf{true}$

Weiterhin gilt für alle Planinstanzen **iname** mit Ausnahme einer Planinstanz $\mathbf{iname}_1$:

- $\sigma.\mathrm{ASH}[\mathbf{iname}][\sigma.\mathrm{AC}] = ([] \Rightarrow \texttt{inactive})$
- $\sigma.\mathrm{HS}[\mathbf{iname}].\mathbf{activate} \Leftrightarrow \mathbf{false}$
- $\sigma.\mathrm{HS}[\mathbf{iname}].\mathbf{state} = \texttt{inactive}$

Für $\mathbf{iname}_1$ gilt stattdessen:

- $\sigma.\mathrm{ASH}[\mathbf{iname}_1][\sigma.\mathrm{AC}] = ([] \Rightarrow \texttt{inactive} \Rightarrow \texttt{considered})$
- $\sigma.\mathrm{HS}[\mathbf{iname}_1].\mathbf{activate} \Leftrightarrow \mathbf{true}$
- $\sigma.\mathrm{HS}[\mathbf{iname}_1].\mathbf{state} = \texttt{considered}$
- Es existiert ein **SK**, so dass $\mathbf{iname}_1 = \mathbf{superGen}(\mathbf{SK})$

Der Instanzname $\mathbf{iname}_1$ beschreibt den initialen Plan. Der initiale Plan ist der Top-Level Plan in der auszuführenden Hierarchie, das heißt, dieser Plan hat keinen übergeordneten Plan. Er startet im Zustand `considered`. Weiterhin wurde bereits das Aktivierungssignal gesendet. Damit kann dieser Plan in die `Execution`-Phase übertreten, sobald er den Zustand `ready` erreicht.

Ein initialer Zustand soll das System in einem definierten Ausgangszustand starten lassen. Dabei sind alle Planzustände auf den Wert `inactive` gesetzt, dass heißt, es gibt nur eine Planinstanz die ausgeführt werden kann. Keiner der Pläne außer dem initialen Plan hat das Aktivierungssignal gesetzt und für alle Pläne gilt, dass die Asbru-Zustands-Historie keine Einträge außer solche mit dem `inactive` Wert enthält. Ausnahme hierbei ist wieder der initiale Plan, dessen `considered` Zustand in der Asbru-Zustands-Historie verzeichnet ist. Auch sind alle Rundenzähler auf den Wert 0 gesetzt und es gibt initial kein Signal, das einem Makro-Schritt widerspricht.

10.6 Geschlossene Asbru-Abläufe

Definition

Ein geschlossener Asbru-Ablauf ist ein Ablauf Σ für den gilt, dass es ein Asbru-System **sys** gibt, so dass $\Sigma \models$ **sys**. Dazu muss gelten, dass $\Sigma(0)$ ein initialer Zustand ist. Zusätzlich gilt für alle Übergänge zwischen eingestrichenen und zweigestrichenen Zuständen in Σ, dass diese Umgebungsübergänge sind.

Geschlossene Abläufe beschreiben die stärkste, aussagenunabhängige Einschränkung, die an Abläufe von Asbru-Systemen gestellt werden kann. Eine Aussage ist dann korrekt, wenn sie in allen geschlossenen Asbru-Abläufen eines Asbru-Systems korrekt ist.

Möglich ist allerdings auch der Beweis von Aussagen über nicht-geschlossenen Asbru-Abläufen. Wenn beispielsweise in einem Beweis keine Zeit-Annotationen vorkommen, so ist es nicht nötig, die Asbru-Zustands-Historie im Sinne der geschlossenen Abläufe mitzuschreiben. Eine solche Generalisierung vergrößert die Menge der Abläufe, erhält aber gleichzeitig alle Asbru-Abläufe. Wenn eine Aussage über einem solchermaßen generalisiertem System bewiesen werden kann, folgt daraus automatisch, dass sie auch über der Menge aller geschlossenen Abläufe gilt.

11 Validierung der Asbru-Semantik

Die Komplexität der Semantik-Definition von Asbru macht diese anfällig für Fehler. Aus diesem Grund wurde Aufwand investiert, um die Semantik zu validieren.

Es gibt verschiedene Möglichkeiten, um eine Ausführungs-Semantik wie die von Asbru zu validieren. Eine einfache Möglichkeit ist die wiederholte Anwendung der SOS-Regeln auf ein Asbru-System um zu eruieren, ob der Ablauf, der durch die Anwendung der SOS-Regeln entsteht, der Erwartung entspricht. Diese Form der Validierung wurde anhand von abstrakten Beispielen durchgeführt, um die SOS-Regeln für die einzelnen Plantypen zu testen. Ebenso wurden aber auch Vorarbeiten aus alten Fallstudien wie der Gelbfieber-Fallstudie verwendet, um zu prüfen, ob die Ergebnisse der Ausführung der formalen Semantik den bereits bekannten Ergebnissen entsprechen.

In diesem Kapitel werden drei unterschiedliche Aspekte der Validierung beschrieben, die darüber hinaus bei der Asbru-Semantik zum Einsatz kommen. Der erste Aspekt beschreibt den Einsatz von formaler Verifikation, um die intendierte Semantik einzelner Prädikate nachzuweisen. Dies wird in Abschnitt 11.1 beschrieben. Danach folgt in Abschnitt 11.2 eine Beschreibung, wie eine gezielte Modifikation der Semantik zum Auffinden eines nicht-trivialen Fehlers führen kann. Zuletzt wird in Abschnitt 11.3 beschrieben, wie die Implementierung der Semantik in ein Beweisunterstützungssystem zur Validierung herangezogen werden kann.

11.1 Validierung von Zeit-Annotationen

Dieser Abschnitt fasst die in [63] publizierten Ergebnisse zusammen und passt diese Ergebnisse auf die aktuelle Version der formalen Semantik von Asbru an.

Zum Umgang mit Asbru-Zeit-Annotationen gibt es zwei Prädikate. Ziel dieser Prädikate ist es, allgemein und abstrakt Eigenschaften bezüglich der von Zeit-Annotationen definierten Mengen an Intervallen zu beschreiben.

Eine Zeit-Annotation beschreibt eine Menge von Intervallen. Eine Zeit-Annotation ist *legal*, das heißt, das Prädikat **legal** wertet zu wahr aus, wenn die von der Zeit-Annotation beschriebene Menge an Intervallen nicht leer ist. Eine Zeit-Annotation ist *normal*, das heißt, das Prädikat **normal** wertet zu wahr aus, wenn die Zeit-Annotation minimal ist. Der Begriff der Minimalität wird Anhand der Abbildung 11.1 beschrieben.

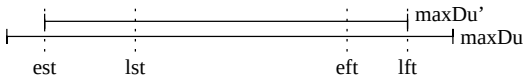


Abbildung 11.1: Beispiel einer nicht-minimalen Zeit-Annotation

In dieser Abbildung ist zu erkennen, dass die Maximallänge der Zeitannotation (maxDu) so angegeben wurde, dass sie größer ist als die Differenz von spätestem Endzeitpunkt und frühestem Startzeitpunkt. Das bedeutet, dass es kein Intervall geben kann, dessen Länge gleich der Maximallänge und das Gleichzeitig Element der Zeit-Annotation ist. In dieser Hinsicht ist diese Zeitannotation nicht minimal. Die Zeichnung gibt eine alternative Maximallänge maxdu' an. Diese verändert die Menge der von der Zeit-Annotation beschriebenen Intervalle nicht,

genügt aber dem Minimalitäts-Begriff. Dies ist der Fall da ein Element der Zeit-Annotation gefunden werden kann, dass die Länge maxdu' hat.

Die Prädikate **normal** und **legal** werden in [63] eingeführt und genauer erläutert. Definiert werden sie durch die folgenden Gleichungen:

Das Prädikat **legal** formalisiert, dass die Menge der durch eine Zeit-Annotation **ata** beschriebenen Intervalle nicht leer ist. Ein erster Versuch der Definition der Semantik dieses Prädikats lautet wie folgt:

$$\begin{aligned}\mathbf{legal}(\mathbf{ata}) \quad \leftrightarrow \quad & \mathbf{ata.ess} + \mathbf{ata.minDu} \leq \mathbf{ata.lfs} \\ & \wedge\, \mathbf{ata.efs} \leq \mathbf{ata.lss} + \mathbf{ata.maxDu}\end{aligned}$$

Das Prädikat **normal** formalisiert, dass eine Zeitannotation minimal ist. Dazu wird zunächst folgende Formalisierung gewählt:

$$\begin{aligned}\mathbf{normal}(\mathbf{ata}) \quad \leftrightarrow \quad & \mathbf{ata.efs} \leq \mathbf{ata.lss} + \mathbf{ata.minDu} \\ & \wedge\, \mathbf{ata.efs} \leq \mathbf{ata.ess} + \mathbf{ata.maxDu} \\ & \wedge\, \mathbf{ata.lfs} \leq \mathbf{ata.lss} + \mathbf{ata.maxDu} \\ & \wedge\, \mathbf{ata.maxDu} + \mathbf{ata.ess} \leq \mathbf{ata.lfs} \\ & \wedge\, \mathbf{ata.minDu} + \mathbf{ata.ess} \leq \mathbf{ata.efs} \\ & \wedge\, \mathbf{ata.minDu} + \mathbf{ata.lss} \leq \mathbf{ata.lfs} \\ & \wedge\, 0 \leq \mathbf{ata.minDu}\end{aligned}$$

Das Prädikat **legal** verlangt laut dieser Definition, dass ein Intervall, dessen Startpunkt auf den frühesten Startzeitpunkt der Zeit-Annotation fällt nicht über den spätesten Endzeitpunkt hinaus andauern darf, sofern seine Länge gleich der minimalen Dauer ist. Informell bedeutet diese Forderung, dass Start- und Endzeiträume so gewählt sind, dass die minimale Dauer „hineinpasst". Eine analoge Forderung wird für die maximale Dauer aufgestellt.

Das Prädikat **normal** soll definieren, dass es keine „zu groß" bzw. „zu klein" gewählten Zeiten in den Intervallen der Zeit-Annotation gibt. Die erste Gleichung der Definition des **normal** Prädikats beispielsweise verlangt, dass der früheste Endzeitpunkt kleiner ist, als die Addition des spätesten Startzeitpunkts und der minimalen Dauer. Wäre diese Ungleichung nicht erfüllt, könnte kein Intervall Element der Zeit-Annotation sein, das genau die Länge der minimalen Dauer hat, da dieses Intervall zu früh enden oder zu spät starten würde. Damit ist die minimale Dauer überflüssig.

Die zweite Zeile der Definition verlangt, dass der früheste Startzeitpunkt sinnvoll spezifiziert ist. Ist die Ungleichung nicht erfüllt, so kann kein Intervall vom frühesten Startzeitpunkt aus gefunden werden, das nicht länger als die Maximaldauer ist und dennoch das Zielintervall erreicht, also nach dem frühesten Endzeitpunkt endet. Die dritte Zeile gibt an, dass der späteste Endzeitpunkt sinnvoll gewählt wurde. Die Zeilen vier, fünf und sechs geben an, dass jeweils maximale Dauer, früheste Endzeitpunkt und spätester Startzeitpunkt sinnvoll gewählt wurden.

Diese Definitionen werden als Ausgangspunkt für die Validierung verwendet. Dazu soll gezeigt werden, dass die intendierte Semantik durch diese Definitionen getroffen wird, das heißt, dass **normal**-Zeit-Annotationen minimal sind und **legal**-Zeit-Annotationen mindestens ein Intervall enthalten. Für die Validierungsbeweise soll die Möglichkeit außer Acht gelassen werden, dass der Referenzpunkt der Zeit-Annotation zum Wert $\perp$ auswertet. Dazu soll jeweils ein Zustand σ gewählt werden, mit dem der Referenzpunkt der Zeitannotation zu einem konkreten Wert ausgewertet wird. Die Aussagen werden damit formal wie folgt gefasst:

Eine Zeit-Annotation **ata** ist **legal**, genau dann wenn ein **interval I** existiert, so dass gilt $[\![\mathbf{I} \in \mathbf{ata}]\!]_\sigma$.

Seien $\mathbf{i}_1$, $\mathbf{i}_2$ und $\mathbf{i}_3$ vom Typ **int**. Eine Zeit-Annotation **ata** ist **normal**, genau dann wenn die folgenden Bedingungen gelten:

- Für alle $\mathbf{i}_1$ mit $\mathbf{i}_1 \in [\![\mathbf{ata.sti}]\!]_\sigma$ gibt es $\mathbf{i}_2$, so dass $[\![(\mathbf{i}_1, \mathbf{i}_2) \in \mathbf{ata}]\!]_\sigma$
- Für alle $\mathbf{i}_1$ mit $\mathbf{i}_1 \in [\![\mathbf{ata.fti}]\!]_\sigma$ gibt es $\mathbf{i}_2$, so dass $[\![(\mathbf{i}_2, \mathbf{i}_1) \in \mathbf{ata}]\!]_\sigma$
- Für alle $\mathbf{i}_1$ mit $\mathbf{i}_1 \in \mathbf{ata.dti}$ gibt es $\mathbf{i}_2$, $\mathbf{i}_3$, so dass $[\![(\mathbf{i}_2, \mathbf{i}_3) \in \mathbf{ata}]\!]_\sigma$ und $\mathbf{i}_3$ - $\mathbf{i}_2 = \mathbf{i}_1$

Der Beweis beider Aussagen schlägt unter Verwendung obiger Definitionen fehl, so dass weder gezeigt werden kann, dass das Prädikat **legal** seine Intention erfüllt, noch kann dies für das Prädikat **normal** gezeigt werden. Die Ursache dafür liegt darin, dass die relativen Start- und Endzeitpunkte der Zeitannotationen auch den Wert $\perp$ enthalten dürfen. Da Werte von Ungleichungen, die ein $\perp$ Element vergleichen, nicht definiert sind, muss die Formalisierung der Prädikate daraufhin geändert werden:

$$
\begin{aligned}
\mathbf{legal}(\mathbf{ata}) \quad \leftrightarrow \quad & \mathbf{ata.ess} = \perp \vee \mathbf{ata.lss} = \perp \vee \mathbf{ata.ess} \leq \mathbf{ata.lss} \\
& \wedge \mathbf{ata.efs} = \perp \vee \mathbf{ata.lfs} = \perp \vee \mathbf{ata.efs} \leq \mathbf{ata.lfs} \\
& \wedge \mathbf{ata.maxDu} = \perp \vee \mathbf{ata.minDu} \leq \mathbf{ata.maxDu} \\
& \wedge \quad \mathbf{ata.efs} = \perp \vee \mathbf{ata.lss} = \perp \vee \mathbf{ata.maxDu} = \perp \\
& \quad \vee \mathbf{ata.efs} \leq \mathbf{ata.maxDu} + \mathbf{ata.lss} \\
& \wedge \mathbf{ata.lfs} = \perp \vee \mathbf{ata.ess} = \perp \vee \mathbf{ata.minDu} + \mathbf{ata.ess} \leq \mathbf{ata.lfs}
\end{aligned}
$$

Abgesehen davon, dass in jedem Fall die Behandlung der $\perp$ Elemente in die Definition miteinbezogen wurde, werden zusätzlich drei Ungleichungen mit eingefügt. Es wird verlangt, dass die drei Intervall-Komponenten der Zeit-Annotation, also Startintervall, Endintervall und Dauerintervall jeweils „vernünftig“ definiert sind. Diese Aspekte wurden in der ursprünglichen Definition nicht berücksichtigt.

Das Prädikat **normal** formalisiert, dass eine Zeitannotation minimal ist:

$$
\begin{aligned}
\mathbf{normal}(\mathbf{ata}) \quad \leftrightarrow \quad & \mathbf{ata.efs} = \perp \vee \mathbf{ata.lss} = \perp \\
& \quad \vee \mathbf{ata.efs} \leq \mathbf{ata.lss} + \mathbf{ata.minDu} \\
& \wedge \quad \mathbf{ata.efs} = \perp \vee \mathbf{ata.maxDu} = \perp \\
& \quad \vee \mathbf{ata.efs} \leq \mathbf{ata.ess} + \mathbf{ata.maxDu} \\
& \wedge \mathbf{ata.lss} = \perp \vee \mathbf{ata.maxDu} = \perp \vee \mathbf{ata.lfs} \leq \mathbf{ata.lss} + \mathbf{ata.maxDu} \\
& \wedge \mathbf{ata.lfs} = \perp \vee \mathbf{ata.ess} = \perp \vee \mathbf{ata.maxDu} + \mathbf{ata.ess} \leq \mathbf{ata.lfs} \\
& \wedge \mathbf{ata.ess} = \perp \vee \mathbf{ata.minDu} + \mathbf{ata.ess} \leq \mathbf{ata.efs} \\
& \wedge \mathbf{ata.lfs} = \perp \vee \mathbf{ata.minDu} + \mathbf{ata.lss} \leq \mathbf{ata.lfs} \\
& \wedge 0 \leq \mathbf{ata.minDu} \\
& \wedge \mathbf{ata.maxDu} = \perp \rightarrow \mathbf{ata.ess} = \perp \vee \mathbf{ata.lfs} = \perp \\
& \wedge \mathbf{ata.efs} = \perp \rightarrow \mathbf{ata.ess} = \perp \\
& \wedge \mathbf{ata.lss} = \perp \rightarrow \mathbf{ata.lfs} = \perp \\
& \wedge \mathbf{ata.ess} = \perp \rightarrow \mathbf{ata.efs} = \perp \vee \mathbf{ata.maxDu} = \perp \\
& \wedge \quad \mathbf{ata.ess} = \perp \vee \mathbf{ata.efs} = \perp \\
& \quad \rightarrow \mathbf{ata.minDu} \leq \mathbf{ata.maxDu} \vee \mathbf{ata.maxDu} = \perp \\
& \wedge \mathbf{ata.lfs} = \perp \rightarrow \mathbf{ata.lss} = \perp \vee \mathbf{ata.maxDu} = \perp
\end{aligned}
$$

Die ersten sieben Zeilen dieser Definition entsprechen mit Ausnahme der Berücksichtigung der $\perp$ Elemente der ursprünglichen Definition. Danach folgen Abhängigkeitsbedingungen bezüglich der $\perp$ Elemente. Ist beispielsweise die maximale Dauer nicht spezifiziert (und kann damit als unendlich interpretiert werden), so müssen gleichzeitig frühester Startzeitpunkt oder spätester Endzeitpunkt den Wert $\perp$ haben. Sind sowohl frühester Startzeitpunkt als auch spätester Endzeitpunkt mit konkreten Werten belegt, so kann daraus die maximal mögliche Gesamtlänge der die Zeit-Annotation erfüllenden Intervalle berechnet werden. Diese muss dann als maximale Dauer angegeben werden um eine minimale Zeit-Annotation zu erhalten.

Ähnlich kann man für frühesten Endzeitpunkt sowie frühesten Startzeitpunkt argumentieren. Ist der früheste Endzeitpunkt nicht spezifiziert, also interpretierbar als minus unendlich, dann muss der früheste Startzeitpunkt ebenfalls undefiniert sein. Andernfalls könnte der früheste Endzeitpunkt bei der Auswertung auf den Wert gelegt werden, der sich aus Addition des frühesten Startzeitpunktes und der minimalen Dauer ergibt. Analog dazu kann man die Notwendigkeit der restlichen Gleichungen ableiten.

Somit konnte durch die Validierung dieser Aussagen offenkundig gemacht werden, inwiefern die Definitionen der Prädikate nicht den Anforderungen entsprechen. Vorher ist dies nicht aufgefallen, da statt der $\perp$-Werte immer mit hypothetischen unendliche-Werten gerechnet wurde. Für diese sind die Gleichungen jedoch richtig.

11.2 Modellwechsel als Methode zur Validierung

Teilweise lassen sich Fehler in der Semantik dadurch aufspüren, dass versuchsweise ein Teil des einer Definition zugrundeliegenden Modells durch ein anderes ausgetauscht wird. Beispielsweise soll in Asbru sichergestellt sein, dass die komplexen Definitionen der Erfüllbarkeit und der Erfülltheit richtig getroffen wurde. Ein Teil dieser Fragestellung ist die Festlegung, dass die Semantikdefinition nicht verlangen darf, dass die Prädikate über Zustände der Zukunft ausgewertet werden.

Dies lässt sich prinzipiell durch eine Aussage formalisieren, die verlangt, dass die Auswertung der Erfüllbarkeit und der Erfülltheit unabhängig davon sind, wie sich ein Zustand in der Zukunft verhält. Formal lässt sich dies wie folgt fassen:

$$\textbf{satisfied}(\textbf{acond}, \sigma_1) \leftrightarrow \textbf{satisfied}(\textbf{acond}, \sigma_2)$$

wenn σ_1 und σ_2 in folgender Relation zueinander stehen:
Für alle Planinstanznamen **iname** und alle Zeitpunkte **AC** mit $\textbf{AC} \leq \sigma_1.\text{AC}$ gilt:

$$\begin{array}{lcl}
\sigma_1.\text{ASH}[\textbf{iname}][\textbf{AC}] & = & \sigma_2.\text{ASH}[\textbf{iname}][\textbf{AC}] \\
\sigma_1.\text{HS} & = & \sigma_2.\text{HS} \\
\sigma_1.\text{PDH}[\textbf{AC}] & = & \sigma_2.\text{PDH}[\textbf{AC}] \\
\sigma_1.\text{EAGG} & = & \sigma_2.\text{EAGG} \\
\sigma_1.\text{Tick} & = & \sigma_2.\text{Tick} \\
\sigma_1.\text{AC} & = & \sigma_2.\text{AC}
\end{array}$$

Diese Forderung bedeutet, dass alle nicht-Historienvariablen, wie beispielsweise der Hierarchiezustand, identisch sein müssen und alle Historienvariablen in allen Feldern, die die Vergangenheit betreffen, identisch sein müssen. Fragt eine formale Definition nach Werten aus der Zukunft, so ist es nicht mehr möglich, die Äquivalenz zwischen den Erfülltheitsprädikaten im Allgemeinen zu zeigen.

Der Versuch diese Äquivalenz zu beweisen schlägt fehl, weil die Konditionale, die die Basis der Bedingungen bilden, komplette Krankenakten und Zeitpunkte auf Wahrheitswerte abbilden. Es kann also nicht sichergestellt werden, dass die Konditionale, die Funktionen von jeweils einer Krankenakte und einer Uhrzeit nach booleschen Werten sind, nicht bereits ungünstig formuliert sind und Abfragen an zukünftige Zustände durchführen.

Um dennoch die Äquivalenz beweisen zu können, wird das zugrundeliegende Modell geändert. Dazu werden die Konditionale so umgestellt, dass sie statt über einer Krankenakte nur über einen Krankenakteneintrag auswerten. Die Definition der Auswertung von Konditionalen wird entsprechend umgestellt, dass jeweils der passende Krankenakteneintrag ausgewählt und zur Auswertung übergeben wird. Das heißt, die Definition zur Auswertung von Konditionalen

$$[\![\mathbf{cond}]\!]_\sigma \leftrightarrow \mathbf{cond}(\sigma.\mathrm{PDH}, \sigma.\mathrm{AC})$$

wird abgeändert zu:

$$[\![\mathbf{cond}]\!]_\sigma \leftrightarrow \mathbf{cond}(\sigma.\mathrm{PDH}[\sigma.\mathrm{AC}], \sigma.\mathrm{AC})$$

Damit ist sichergestellt, dass sich ein Konditional „vernünftig“ verhält und nicht in die Zukunft auswertet. Mit dieser Änderung wird der Beweis der Äquivalenz der beiden Erfülltheitsbedingungen erneut angegangen. Dieser schlägt wieder fehl, da die Definition der Erfülltheit nicht korrekt ist. In der ursprünglichen Version der Erfülltheitsbedingungen wurde bei der Auswertung der Signalflanken am Ende des Korridors nicht überprüft, ob diese möglicherweise in der Zukunft liegt. Dadurch konnte hier das Konditional zu einem Zeitpunkt ausgewertet werden, für den noch keine Daten vorlagen und damit „in die Zukunft gesehen“ werden. Dieser Fehler konnte nur aufgrund des Modellwechsels formal behandelt und erkannt werden. Nach der Korrektur dieses Fehlers gelingt der Beweis.

11.3 Validierung durch unabhängige Spezifikation

Als letzter Aspekt der Validierung der Semantik von Asbru wird hier die mehrfache, unabhängige Spezifikation der Semantik aufgeführt.

Um eine Vorstellung von den Details der Auswertung zu erhalten, wurde die Ausführungssemantik zunächst nur rudimentär in Form der State-Chart Notation spezifiziert. Basierend auf diesen Rohentwürfen wurde eine Implementierung durch rekursive Prozeduren in das Beweissystem KIV durchgeführt. Mit dieser Umsetzung war es möglich, die Ausführung von Grenzfällen zu simulieren und die Ergebnisse dieser Simulation mit den Spracherstellern von Asbru abzugleichen. Aufgrund der Rückmeldung wurden die SOS-Regeln definiert und diese dann unabhängig von den ursprünglich implementierten Prozeduren komplett neu in das Verifikationswerkzeug implementiert.

Damit stehen zwei weitgehend voneinander unabhängige Modelle der Asbru Semantik zur Verfügung. Einerseits handelt es sich dabei um die SOS-Regeln der Semantik, andererseits um die Implementierung durch parallele Prozeduren. Um Beziehungen zwischen diesen Modellen herzustellen wird ein Beweis der Korrektheit der Einbettung in Kapitel 13 vorgestellt. Dieser beweist die semantische Äquivalenz beider Modelle modulo einer bijektiven Abbildungsfunktion.

Bei Durchführung dieses Beweises werden Inkonsistenzen zwischen den Modellen offenkundig. Diese können zum Teil auf Fehler in den SOS-Regeln, zum Teil auf Fehler in den parallelen Programmen zurückgeführt werden. Ausgangspunkt dafür, Fehler entdecken zu können, sind die unterschiedlichen Herangehensweisen. Beispielsweise muss auf Implementierungsseite speziell auf Beweiseffizienz der Definitionen geachtet werden, während auf Seiten der SOS-Regeln eine möglichst einfach zu verstehende Definition Priorität hat. Weiterhin ergeben sich Unterschiede dadurch, dass die SOS-Regeln sich wechseleseitig ausschließende Vorbedingungen haben müssen um deterministisch zu sein, wohingegen eine sequentielle Komponente eines parallelen Programms automatisch durch die **itlif**-Programmkonstrukte deterministische Form hat.

Teil III

Einbettung von Asbru in KIV

12 Implementierung der SOS-Regeln

Für die Einbettung von Asbru in KIV sind zwei verschiedene Aspekte zu betrachten. Zum Einen müssen die statischen Komponenten eine Entsprechung in der KIV Implementierung finden, also die Datentypen, des Asbru-Zustands sowie der Datentypen von Asbru. Zum Anderen ist es erforderlich, die dynamischen Aspekte, also die Implementierung der SOS-Regeln zu betrachten. Die Implementierung der Zustands- und Datentypen wird in Anhang A beschrieben. In diesem Abschnitt wird aufbauend auf diesen Datentypen die Implementierung der SOS-Regeln beschrieben.

Die durch die SOS-Regeln definierte Semantik erlaubt, Korrektheitsbeweise händisch durchzuführen. Für effizientes Beweisen in einem System, das so komplex ist wie Asbru, ist jedoch automatisierte Beweisunterstützung erforderlich.

Für die Beweisunterstützung zum Nachweis medizinischer Aussagen über Asbru-Systeme wurde eine Implementierung für das interaktive Beweisunterstützungssystem KIV gewählt. Dazu wurden die SOS-Regeln in Form paralleler Programme implementiert. Im Folgenden soll diese Implementierung vorgestellt. Die Korrektheit der Implementierung wird dann im Kapitel 13 gezeigt.

Die Komplexität der Asbrupläne macht eine effiziente Implementierung durch eine globale Zustandstransformationsregel - wie sie etwa von ASMs bekannt ist - nahezu unmöglich. Stattdessen wurde der Weg gewählt, die SOS-Regeln zu gruppieren und jeweils eine Gruppe von Regeln durch eine Prozedur zu implementieren. Es werden beispielsweise Prozeduren definiert, die jeweils einem der Planzustände `considered` oder `possible` entsprechen. Jede dieser Prozeduren bildet genau das Verhalten ab, das durch die SOS-Regeln definiert ist, die von diesem Zustand ausgehen. Beispielsweise sind im Zustand `considered` alle Regeln anwendbar, die entweder keine Voraussetzung an den Planzustand treffen oder fordern, dass das Prädikat **isConsidered** oder das Prädikat **isSelected** wahr ist. Die Implementierung dieser Regeln wird dann in einer Prozedur `considered#` gebündelt, wobei diese Prozedur je nach Regelanwendung im Anschluss an die Ausführung terminiert, wieder die Prozedur `considered#` oder die Prozedur `possible#` aufruft.

Eine weitergehende Unterteilung der Implementierung wird im Zustand `activated` durchgeführt. Dort implementiert jeweils eine Prozedur alle Zustandsübergänge eines einzelnen Plankontrolltyps. Eine Prozedur `body#` ruft entsprechend des Kontrolltyps die korrekte Prozedur auf.

12.1 Technische Prozeduren

Die Implementierung von Asbru verlangt die Einführung von drei Prozeduren, die keine direkte Entsprechung in den SOS-Regeln haben. Eine Prozedur `inactive#` beschreibt das Verhalten eines Plans im Zustand `inactive`, der gerade gestartet wurde und deswegen seinen Zustand ohne weitere Prüfungen auf `considered` ändert. Eine Prozedur `inactive-rec#` ist ein rekursiver Aufruf der `inactive#`-Prozedur. Dieser ist notwendig, um den parallelen Planstart beliebiger Unterplanmengen zu implementieren. Die `considered-rec#`-Prozedur beschriebt das Verhalten beliebig vieler Pläne im Zustand `considered` und entspricht damit dem Asbru-System, das nach dem Start mehrerer Unterpläne in den SOS-Regeln entsteht.

12.1.1 Die `inactive#`-Prozedur

Die `inactive#`-Prozedur ändert den Planzustand ohne weitere Prüfungen ab auf den Wert `considered` und ruft entsprechend im Anschluss daran die `considered#`-Prozedur auf.

```
inactive#(iname; var Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
begin
  AS[iname] := considered;
  considered#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
end;
```

Listing 12.44: `inactive#`-Prozedur

12.1.2 Die `inactive-rec#`-Prozedur

Die `inactive-rec#`-Prozedur implementiert den Start von Mengen von Unterplänen. Dazu erhält die `inactive-rec#`-Prozedur als Parameter die Liste von Plänen, die gestartet werden soll. Für diese wird rekursiv jeweils eine `inactive#`-Prozedur parallel zur `inactive-rec#`-Prozedur, parametrisiert mit der Restliste der zu startenden Pläne, gestartet

```
inactive-rec#(strx, iname; var Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
begin
  itlif strx = [] then skip
  else itlif strx.rest = [] then
         inactive#(nameGen(strx .first, iname); Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
       else
         begin
           inactive#(nameGen(strx .first, iname); Tick, EAGG, PDH, ASH, AS, AC, PC, RC) ||s
           inactive-rec#(skx .rest, iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
         end
end;
```

Listing 12.45: `inactive-rec#`-Prozedur

12.1.3 Die `considered-rec#`-Prozedur

Die `considered-rec#`-Prozedur ist das Analogon der `inactive-rec#`-Prozedur bezogen auf den Zustand `considered`. Diese Prozedur bildet also eine Menge von Plänen ab, die sich im Zustand `considered` befinden.

```
considered-rec#(strx, iname; var Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
begin
  itlif strx = [] then [PL laststep]
  else itlif strx.rest = [] then
         considered#(nameGen(strx .first, iname); Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
       else
         begin
           considered#(nameGen(strx .first, iname);
                       Tick, EAGG, PDH, ASH, AS, AC, PC, RC) ||s
           considered-rec#(strx .rest, iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
         end
end;
```

Listing 12.46: `considered-rec#`-Prozedur

12.2 Zustandsübergänge ausgehend von der Selection-Phase

12.2.1 Die considered#-Prozedur

Alle Zustandsübergänge, die vom Zustand considered ausgehen, werden von der Prozedur considered# abgebildet, die in Listing 12.47 dargestellt wird. Relevant für die Übergänge vom Zustand considered aus ist die Betrachtung der Regel 9.1, die den Abbruchfall eines Plans im Zustand considered beschreibt, sowie die Regeln 9.2 und 9.3, die den Stillstand im Zustand considered sowie das Fortschreiten in den Zustand possible beschreiben. Ausschließlich diese drei Regeln sind anwendbar, wenn ein Plan im Zustand considered ist.

```
considered#(iname; var Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
begin
   itlif   shouldRejectFilter(iname, EAGG, PDH, ASH, AC)
         ∨ parent_terminated(AS, iname) then
     AS[iname] := rejected,
     Tick      := false
  else itlif (shouldProceedFilter(iname, EAGG, PDH, ASH, AC)) then
    begin
      AS[iname] := possible,
      Tick      := false;
      possible#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
    end
  else
    begin
      AS[iname] := considered;
      considered#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
    end
end;
```

Listing 12.47: considered#-Prozedur

Im Programmtext können die Zustandsübergänge den entsprechenden Regeln einfach zugeordnet werden. Die erste Auswahlbedingung überprüft die Anwendbarkeit der Regel 9.1, indem auf die Terminierung des Superplans sowie auf die nicht-Erfüllbarkeit der **filter**-Bedingung getestet wird. Ist eines von beiden gegeben, so ist die Terminierungsregel anwendbar. In diesem Fall wird im Programm der Zustand auf rejected gesetzt und die Abarbeitung eingestellt.

Sollte keine Abbruchsituation vorliegen, so wird die Anwendung der Regel 9.2 überprüft. Dazu wird in der Prozedur überprüft, ob die **filter**-Bedingung erfüllt ist. Analog zu der SOS-Regel wird in diesem Fall der Zustand des Planes auf possible gesetzt. In diesem Fall wird die possible#-Prozedur aufgerufen.

Sollte die Vorbedingung dieser SOS-Regeln nicht erfüllt sein, so wird das Analogon zu Regel 9.3 ausgeführt. Dieser Fall wird in der Implementierung durch einen **else** Zweig dargestellt, da die Vorbedingungen der SOS-Regeln wechselseitig ausschließend sind. In diesem Fall wird keine Zustandsänderung durch den Plan ausgelöst, das heißt, der Plan wartet passiv auf eine Zustandsänderung durch seine Umgebung. Daher ist dies auch der einzige Fall, in dem kein Tick Ereignis verhindert wird, also die Tick Variable nicht auf den Wert **false** gesetzt wird. Damit wird ein Makro-Schritt nicht verhindert.

12.2.2 Die possible#-Prozedur

Nach Erfüllen der **filter**-Bedingung im Zustand considered wechselt die Plankontrolle in den Zustand possible. Das mögliche Verhalten in diesem Zustand wird durch die Prozedur possible# implementiert. Diese bildet die Zustandsübergänge ab, welche durch die SOS-Regeln 9.4 sowie 9.6 und 9.5 definiert werden.

```
possible#(iname; var Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
begin
   itlif   shouldRejectSetup(iname, EAGG, PDH, ASH, AC)
         ∨ parent_terminated(AS, iname) then
     AS[iname] := rejected,
     Tick      := false
  else itlif (shouldProceedSetup(iname, EAGG, PDH, ASH, AC)) then
    begin
      AS[iname] := ready,
      Tick      := false;
      ready#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
    end
  else
    begin
      AS[iname] := possible;
      possible#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
    end
end;
```

Listing 12.48: Possible#-Prozedur

Zunächst wird auch in der `possible#`-Prozedur überprüft, ob ein Abbruchkriterium erfüllt ist. Grundlage dafür sind die SOS-Regel 9.4. Im Falle der Anwendbarkeit wird der Zustand des Planes auf `rejected` gesetzt und die Planausführung beendet.

Soll der Plan nicht terminieren, so wird die Anwendbarkeit der Regel 9.5 überprüft, die auf die Erfülltheit der **setup**-Bedingung testet. Sollte diese erfüllt sein, so wird der Planzustand dieses Plans auf den Wert `ready` geändert. In der Folge wird die `ready#` Prozedur gestartet.

Kann keine dieser Regeln angewendet werden, so ist die letzte verbliebene Regel 9.6 automatisch aufgrund der wechselseitig ausschließenden Vorbedingungen anwendbar. Diese verlangt, dass sich der Zustand nicht ändert, womit der Plan im Zustand `possible` verbleibt und wieder die `possible#`-Prozedur aufgerufen wird.

12.2.3 Die `ready#`-Prozedur

Nach Erfüllen der **setup**-Bedingung im Zustand `possible` wechselt die Plankontrolle in den Zustand `ready`. Um die in diesem Zustand möglichen Zustandsübergänge abzubilden, wird die Prozedur `ready#` verwendet, die in Abbildung 12.49 dargestellt ist.

```
ready#(iname; var Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
begin
  itlif parent_terminated(AS, iname) then
    AS[iname] := rejected,
    Tick      := false
  else itlif (PC[iname]) then
    begin
      AS   := AS[iname, activated][asbru(nameSel(iname)) .subplans, iname, inactive],
      PC   := PC[iname, false][asbru(nameSel(iname)) .subplans, iname, false],
      RC   := RC[iname, 0],
      Tick := false;
      activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
    end
  else
    begin
      AS[iname] := ready;
      ready#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
    end
end;
```

Listing 12.49: `ready#`-Prozedur

Im Zustand `ready` sind drei verschiedene Zustandsübergänge möglich, die durch die Regeln 9.7, 9.8 und 9.9 beschrieben werden. Diese Regeln werden durch die `ready#`-Prozedur implementiert.

Die erste **itlif** Abfrage gilt der Anwendbarkeit der Regel 9.7. Diese ist immer dann anwendbar, wenn der übergeordnete Plan terminiert ist. Ist dies nicht gegeben, so wird mittels des zweiten **itlif** geprüft, ob das Aktivierungssignal vorliegt. Falls dies der Fall ist, geht der Plan in den Zustand `activated` über. In dem Fall, in dem der Plan weder abbrechen soll noch in den Zustand `activated` fortschreitet, verbleibt er im Zustand `ready`.

Soll der Plan in den Zustand `activated` übergehen, so wird die `activated#`-Prozedur aufgerufen. Weiterhin werden in diesem Schritt die Planzustandswerte der Unterpläne des Plans auf den Wert `inactive`, die Aktivierungssignale der Unterpläne auf den Wert **false** und der Rundenzähler des Plans auf 0 gesetzt. Zuletzt wird dann das Aktivierungssignal des Plans selbst „konsumiert“, also auf den Wert **false** gesetzt

Wie auch in den vorigen Fällen ist ausschließlich in dem Stillstandsfall ein Tick Ereignis nicht ausgeschlossen. Alle anderen Transitionen verbieten ein globales Tick, da jeweils eine Zustandsänderung stattfindet.

12.3 Zustandsübergänge ausgehend von der Execution-Phase

12.3.1 Die activated#-Prozedur

Wird der Zustand `ready` mit Erhalt eines Aktivierungssignales verlassen, so erreicht die Plankontrolle den Zustand `activated`. Die implementierende Prozedur `activated#` orientiert sich in diesem Zustand an der Hierarchisierung der SOS-Regeln. Das bedeutet, dass in diesem Zustand zunächst abgefragt wird, ob ein Grund vorliegt, die Ausführung abzubrechen, zu beenden oder zu unterbrechen. Falls keiner dieser Gründe vorliegt, wird eine niedriger priorisierte Prozedur aufzurufen, die die plantypenspezifischen Zustandsübergänge implementiert. Dieses Vorgehen ist analog zum Vorgehen der SOS-Regeln beim Aufruf der `con`-Regeln.

```
activated#(iname; var Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
begin
  itlif shouldAbort(iname, EAGG, PDH, ASH, AS, AC) then
      AS[iname] := aborted,
      Tick      := false
  else itlif shouldComplete(iname, EAGG, PDH, ASH, AS, AC) then
    AS[iname] := completed,
    Tick      := false
  else itlif shouldSuspend(iname, EAGG, PDH, ASH, AS, AC) then
    begin
      AS[iname] := suspended,
      Tick := false;
      suspended#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
     end
  else
    body#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
end;
```

Listing 12.50: Activated#-Prozedur

Zunächst wird von der `activated#`-Prozedur, die in Abbildung 12.50 dargestellt ist, überprüft, ob ein Grund zum Abbruch der Ausführung vorliegt. Es gibt drei verschiedene Gründe, aufgrund derer die Auswertung laufender Pläne abgebrochen werden kann. Der erste Grund ist die Terminierung des Superplans. Der zweite Grund ist die Erfüllung der **abort**-Bedingung und der dritte Grund zum Abbruch der Ausführung ist die fehlende Erfüllbarkeit der Wartebedingung **wait-for**. In all diesen Fällen soll der Plan abbrechen. Aus Gründen der Übersichtlichkeit werden die drei Abbruch-Gründe mittels eines einzigen Prädikats namens **shouldAbort** im ersten **itlif** der Prozedur geprüft.

Bei negativer Auswertung des **shouldAbort**-Prädikats wird getestet, ob der Plan seine Ausführung abschließen soll. Dazu werden die durch Wartebedingung **wait-for** sowie die

complete-Bedingung gestellten Anforderungen überprüft. Sind diese Anforderungen beide erfüllt, wird die Ausführung der Prozedur abgeschlossen und dabei der Planzustand auf `completed` gesetzt. Diese Überprüfung wird in einem Prädikat namens **shouldComplete** gekapselt.

Liegt keiner der beiden Terminierungsfälle vor, so wird mittels der dritten **itlif**-Abfrage überprüft, ob der Plan seine Ausführung unterbrechen soll. Dazu muss die **suspend**-Bedingung wahr sein, oder der Superplan des aktuell betrachteten Planes im `suspended` Zustand sein. In diesen Fällen wechselt der Planzustand zu `suspended`. Die Abfrage wird durch das Prädikat **shouldSuspend** gekapselt.

Liegt keiner dieser drei Fälle vor, wird die `body#`-Prozedur ausgeführt. Diese steht stellvertretend für alle `con`-SOS-Regeln, die die für die einzelnen Kontrolltypen spezifischen Zustandsübergänge enthält.

12.3.2 Die `suspended#`-Prozedur

Erreicht ein Plan den Zustand `suspended`, wird seine Ausführung unterbrochen. Dies bedeutet, dass keine weiteren Unterpläne gestartet oder aktiviert werden und ein Abschließen des Planes ausgeschlossen ist. Nach wie vor, wird allerdings die Abbruchbedingung des Planes in Form des Prädikates **shouldAbort** ausgewertet. Falls dieses nicht erfüllt ist, wird kontrolliert, ob der Plan zurück in den Zustand `activated` wechseln kann.

```
suspended#(iname; var Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
begin
  itlif shouldAbort(iname, EAGG, PDH, ASH, AS, AC)  then
      AS[iname] := aborted,
      Tick      := false
  else itlif shouldReactivate(iname, EAGG, PDH, ASH, AS, AC) then
    begin
      AS[iname] := activated,
      Tick      := false;
      activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
     end
  else
    begin
      AS[iname] := suspended;
      suspended#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
    end
end;
```

Listing 12.51: `suspended#`-Prozedur

Das Verhalten dieses Zustandes wird von der `suspended#`-Prozedur abgebildet, die in Abbildung 12.51 dargestellt ist. In dieser wird mit der ersten **itlif**-Abfrage überprüft, ob der Abbruchfall eingetreten ist. Falls dies geschehen ist, so wird der Planzustand auf `aborted` gesetzt und die Planausführung eingestellt.

Liegt kein Abbruchfall vor, so wird als nächstes überprüft, ob eine Reaktivierung des Plans möglich ist. Dazu muss der Superplan des betrachteten Planes im aktiven Zustand und außerdem die **reactivate**-Bedingung erfüllt sein. Diese beiden Bedingungen werden im Prädikat **shouldReactivate** gekapselt. Soll der Plan reaktiviert werden, so wird der Zustand auf den Wert `activated` gesetzt und danach wieder die `activated#`-Prozedur aufgerufen.

Kann keine Reaktivierung stattfinden, so bleibt der Zustand `suspended` erhalten. Nur in diesem Fall ist ein Makro-Schritt möglich.

12.3.3 Die `body#`-Prozedur

Die SOS-Regeln, die die niedriger priorisierten `con` Übergänge festlegen, sind alle spezifisch für einzelne Plankontrolltypen. Daher ist in den Vorbedingungen dieser Regeln stets eine

Abfrage auf den korrekten Plantypen enthalten. Dadurch sind die Vorbedingungen für unterschiedliche Plankontrolltypen automatisch wechselseitig ausschließend. Um die Übersichtlichkeit der Prozeduren zu verbessern, werden die einzelnen Plankontrolltypen in unterschiedlichen Prozeduren implementiert. Die body#-Prozedur, die in Abbildung 12.52 abgebildet ist, führt die Unterscheidung nach den Plankontrolltypen durch und ruft abhängig von diesem jeweils die Prozedur auf, die dann alle zu diesem Kontrolltypen gehörigen SOS-Regeln implementiert.

Ausnahme hierzu sind die ask und user Kontrolltypen. Da diese Plantypen sich bei der Ausführung stets passiv verhalten und keine Handlungen ausführen, werden diese Plantypen nicht durch eigenen Prozeduren implementiert. Stattdessen wird deren passives Abwarten durch den **else** Fall der body#-Prozedur implementiert.

```
body#(iname; var Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
begin
  itlif isSequential(iname) then
    sequential#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
  else itlif isAnyorder(iname) then
    anyorder#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
  else itlif isParallel(iname) then
    parallel#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
  else itlif isUnordered(iname) then
    unordered#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
  else itlif isOnAbort(iname) then
    onabort#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
  else itlif isITE(iname) then
    ite#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
  else itlif isCyclical(iname) then
    cyclical#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
  else
    begin
      AS[iname] := activated;
      activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
    end
end;
```

Listing 12.52: body#-Prozedur

12.4 Prozeduren der Plankontrolltypen

12.4.1 Die sequential#-Prozedur

Die sequential#-Prozedur bildet die Kontrollübergänge ab, die für die sequentielle Plankontrolle spezifisch sind. Dabei unterscheidet die sequentielle Plankontrolle bei der Ausführung zwei unterschiedliche Fälle.

```
sequential#(iname; var Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
begin
  itlif   hasExecutedSub(AS, iname)
        ∨ hasSelectedSub(AS, iname)
        ∨ ¬ hasStartableSub(AS, iname) then
    begin
      AS[iname] := activated;
      activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
    end
  else
    begin
          inactive#(nameGen(selectFirstStartable(AS, iname), iname);
                    Tick, EAGG, PDH, ASH, AS, AC, RC) ‖s
          begin
            AS[iname] := activated,
            Tick := false,
            PC[nameGen(selectFirstStartable(AS, iname), iname)] := true;
```

```
            activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
          end

    end
end;
```

Listing 12.53: `sequential#`-Prozedur

Der erste Fall setzt voraus, dass keine Handlung seitens des Plans notwendig ist. Dies wird daran festgemacht, dass bereits ein Unterplan läuft oder kein startbarer Plan mehr existiert. Da die sequentielle Kontrolle nur höchstens einen Unterplan gleichzeitig in `Execution`- oder `Selection`-Phase hat, soll kein Unterplan gestartet werden, wenn bereits festgestellt werden kann, dass ein Unterplan in einer dieser Phasen existiert. Offensichtlich ist es außerdem nicht möglich, einen Unterplan zu starten, wenn kein Unterplan mehr existiert, der gestartet werden kann. Dies ist immer dann der Fall, wenn kein Unterplan mehr im Zustand `inactive` ist und gleichzeitig kein Plan existiert, der neu gestartet werden soll.

Der zweite Fall tritt immer dann ein, wenn derzeit kein Unterplan in `Selection`- oder `Execution`-Phase existiert, gleichzeitig aber noch mindestens ein Unterplan startbereit ist. In diesem Fall soll der erste startbare Plan ausgewählt und aktiviert werden. Dabei wird die Prozedur `inactive#` für den ausgewählten Unterplan parallel zum laufenden Plan gestartet. Damit wird erreicht, dass der Zustand des Unterplanes bereits im laufenden Schritt auf `considered` wechselt, wie von den SOS-Regeln gefordert wird.

12.4.2 Die `anyorder#`-Prozedur

Der `anyorder` Kontrolltyp startet zunächst alle seine Unterpläne gleichzeitig und wählt dann jeweils unter den Unterplänen, die im Zustand `ready` sind einen aus, der aktiviert wird. Dabei werden abgebrochene Unterpläne gegebenenfalls neu gestartet. Die Transition, die abgebrochene Pläne neu startet, hat gegenüber den anderen Plänen die höchste Priorität.

```
anyorder#(iname; var Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
begin
  itlif hasStartableSub(AS, iname) then
    begin
      inactive-rec#(selectAllStartable(AS, iname), iname;
                    Tick, EAGG, PDH, ASH, AS, AC, RC) ||s
      begin
        AS[iname] := activated,
        Tick      := false;
        activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
      end
    end
  else itlif hasExecOrSigReadySub(AS, PC, iname) then
    begin
      AS[iname] := activated;
      activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
    end
  else itlif hasReadySub(AS, iname) then
    begin
      AS[iname] := activated,
      PC[nameGen(selectPrivilegedSub(EAGG, AS, iname), iname)] := true,
      Tick := false;
      activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
    end
  else
    begin
      AS[iname] := activated;
      activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
    end
end;
```

Listing 12.54: `anyorder#`-Prozedur

Wenn keine startbaren Pläne vorhanden sind, wird überprüft, ob ein Unterplan in der `Execution`-Phase ist oder ein Unterplan in der `Selection`-Phase bereits das Aktivierungssignal erhalten hat. In diesen Fällen wartet die Plankontrolle eine Zustandsänderung ab.

Ist dagegen noch kein solcher Plan vorhanden, gleichzeitig aber mindestens einer, der den Zustand `ready` erreicht hat, so wird unter allen Unterplänen des `ready`-Zustandes nach Vorgabe des externen `prefer` Signales einer ausgewählt, der dann aktiviert wird.

In jedem anderen Fall, wenn also kein einziger Unterplan mehr läuft oder startbar ist, oder kein Plan den Zustand `ready` erreicht hat, wird der Plan nur abwarten.

12.4.3 Die `parallel#`-Prozedur

Die `parallel#`-Prozedur startet alle Unterpläne gleichzeitig und wartet dann so lange ab, bis diese jeweils entweder zurückgewiesen wurden oder den Zustand `ready` erreicht haben. In diesem Fall werden alle Unterpläne gleichzeitig aktiviert.

```
parallel#(iname; var Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
begin
  itlif hasExecOrSigReadySub(AS, PC, iname) then
    begin
      AS[iname] := activated;
      activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
    end
  else itlif (  allSubsReadyOrRejected(AS, iname)
              ∧ hasSelectedSub(AS, iname)) then
    begin
      PC        := PC[selectAllReady(AS, iname), iname, true],
      AS[iname] := activated,
      Tick      := false;
      activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
    end
  else itlif hasSelectedSub(AS, iname) then
    begin
      AS[iname] := activated;
      activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
    end
  else itlif hasInactiveSub(AS, iname) then
    begin
      inactive-rec#(asbru(nameSel(iname)) .subplans; Tick, EAGG, PDH, ASH, AS, AC, RC) ‖s
      begin
        AS[iname] := activated,
        Tick := false;
        activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
      end
    end
  else
    begin
      AS[iname] := activated;
      activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
    end
end;
```

Listing 12.55: `parallel#`-Prozedur

Zunächst wird dazu überprüft, ob solche Unterpläne existieren, die aktiviert wurden und im Zustand `ready` sind oder solche Unterpläne, die bereits in der `Execution`-Phase sind. In diesem Fall wird wartet der Plan ab. Gibt es hingegen noch keine aktivierten Unterpläne, so ist zu prüfen, ob alle Pläne, die nicht in Ihrer `Selection`-Phase zurückgewiesen wurden, den Zustand `ready` erreicht haben. Sollte dies der Fall sein, so werden diese Unterpläne aktiviert.

Sind die Unterpläne weder aktiviert worden noch alle im Zustand `ready`, so wird als nächstes getestet, ob die Unterpläne bereits in die `Selection`-Phase versetzt wurden. Ist dies

der Fall, so muss der Kontrolltyp abwarten, da noch nicht alle Unterpläne für den Übergang in die `Execution`-Phase bereit sind.

Fällt auch dieser Test negativ aus, so bedeutet das, dass noch keine Pläne in der `Selection`-Phase existieren. Sind in diesem Fall Unterpläne im Zustand `inactive` vorhanden, so werden diese gestartet.

Zuletzt wird noch der Randfall betrachtet, der keinem der anderen Fälle entspricht, beispielsweise, weil alle Unterpläne abgebrochen wurden. In diesem Fall ist nichts weiter zu tun, der Plan wartet in diesem Fall einfach ab.

12.4.4 Die `unordered#`-Prozedur

Der `unordered`-Kontrolltyp erlaubt eine fast beliebige Ausführung der Unterpläne. Diese werden gleichzeitig gestartet und aktiviert, sofern sie in einem (neu)startbaren Zustand sind.

```
unordered#(iname; var Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
begin
  itlif hasStartableSub(AS, iname) then
    begin
       inactive-rec#(selectAllStartable(AS, iname), iname;
                     Tick, EAGG, PDH, ASH, AS, AC, RC) ||s
       begin
         AS[iname] := activated,
         PC        := PC[selectAllStartable(AS, iname), iname, true],
         Tick      := false;
         activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
       end
    end
  else
    begin
      AS[iname] := activated;
      activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
    end
end;
```

Listing 12.56: `unordered#`-Prozedur

Analog zu diesen Vorgaben der SOS-Regeln sieht die Implementierung in Form der `unordered#`-Prozedur aus. Hier wird zunächst überprüft, ob startbare Pläne existieren, entweder weil sie noch im Zustand `inactive` sind oder weil der Plan abgebrochene Pläne neu starten soll. All diese startbaren Pläne werden gestartet und gleichzeitig das Aktivierungssignal geschickt.

Existieren keine startbaren Pläne, so wartet der Kontrolltyp passiv ab.

12.4.5 Die `onabort#`-Prozedur

Aufgabe des `onabort`-Kontrolltypen ist es, einen Plan auszuführen und im Falle des Abbruches dieses Planes einen Ersatzplan zu starten. Damit kennt die `onabort`-Kontrolle drei verschiedene Handlungsmöglichkeiten.

```
onabort#(iname; var Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
begin
  itlif isInactive(AS, nameGen(selectFirstSub(iname), iname)) then
    begin
      inactive#(nameGen(selectFirstSub(iname), iname); Tick, EAGG, PDH, ASH, AS, AC, RC) ||s
      begin
        AS[iname] := activated,
        PC        := PC[selectFirstSub(iname), true],
        Tick      := false;
        activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
      end
    end
```

```
  else itlif   isStartable(AS, nameGen(selectSecondSub(iname), iname),
                           asbru(nameSel(iname)) .retry-aborted))
             ∧ isAborted(AS, nameGen(selectFirstSub(iname), iname)) then
    begin
      inactive#(nameGen(selectSecondSub(iname), iname);
                Tick, EAGG, PDH, ASH, AS, AC, RC) ||s
      begin
        AS[iname] := activated,
        PC        := PC[nameGen(selectSecondSub(iname), iname), true],
        Tick := false;
        activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
      end
    end
  else
    begin
      AS[iname] := activated;
      activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
    end
end;
```

Listing 12.57: Onabort#-Prozedur

Initial ist der erste Unterplan im Zustand `inactive`. In diesem Fall wird dieser Plan gestartet und aktiviert. Sollte der erste Unterplan bereits abgebrochen worden sein, wird überprüft, ob der zweite Unterplan (neu)startbar ist. Ist dies gegeben, so wird der zweite Unterplan gestartet und aktiviert.

In allen anderen Fällen, etwa wenn ein Unterplan noch aktiv ist oder der erste Unterplan in den Zustand `rejected` versetzt wurde, wartet der Plankontrolltyp ab.

12.4.6 Die `ite#`-Prozedur

Der `ifThenElse`-Kontrolltyp soll bei seinem ersten Aufruf entscheiden, ob ein Konditional wahr ist oder nicht. In Abhängigkeit des Ergebnisses dieser Prüfung soll danach entweder der erste oder der zweite Unterplan gestartet und aktiviert werden.

```
ite#(iname; var Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
begin
  itlif allSubsInactive(AS, iname) then
    begin
      itlif asbru(nameSel(iname)) .type .conditional(PDH, AC) then
        begin
          inactive#(nameGen(selectFirstSub(iname), iname);
                    Tick, EAGG, PDH, ASH, AS, AC, RC) ||s
          begin
            AS[iname] := activated,
            PC        := PC[nameGen(selectFirstSub(iname), iname), true],
            Tick      := false;
            activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
          end
        end
      else itlif secondSubPresent(iname) then
        begin
          inactive#(nameGen(selectSecondSub(iname), iname);
                    Tick, EAGG, PDH, ASH, AS, AC, RC) ||s
          begin
            AS[iname] := activated,
            PC        := PC[nameGen(selectSecondSub(iname), iname), true],
            Tick      := false;
            activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
          end
        end
      else
        begin
          AS[iname] := completed,
          Tick := false
        end
```

```
    end
  else itlif   hasSelectedSub(AS, iname)
             ∨ hasExecutedSub(AS, iname)
             ∨ ¬ hasRestartableSub(AS, iname) then
    begin
      AS[iname] := activated;
      activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
    end
  else
    begin
      inactive#(nameGen(pickFirstRestartable(AS, iname), iname);
                Tick, EAGG, PDH, ASH, AS, AC, RC) ||s
      begin
        AS[iname] := activated,
        PC        := PC[nameGen(pickFirstRestartable(AS, iname), iname), true],
        Tick      := false;
        activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
      end
    end
end;
```

Listing 12.58: ite#-Prozedur

Um festzustellen, ob ein `ifThenElse`-kontrollierte Plan bereits sein Konditional ausgewertet hat, wird der Zustand der Unterpläne herangezogen. Einer diese Pläne wird durch die Auswertung des Konditionals und den danach folgenden Start aus dem Zustand `inactive` in die `Selection`-Phase versetzt. Da der Zustand `inactive` danach nicht mehr vom Unterplan erreicht wird, kann der übergeordnete Plan mit der Überprüfung, ob alle Unterpläne im Zustand `inactive` sind feststellen, ob er sein Konditional bereits ausgewertet hat.

Bei der positiven Auswertung des Konditionals wird der erste Unterplan gestartet, bei negativer Auswertung wird der zweite Unterplan gestartet, sofern dieser existiert. Existiert kein zweiter Unterplan, so wird die Ausführung des `ifThenElse`-Plans beendet und der Plan versetzt sich in den Zustand `completed`.

Muss das Konditional nicht ausgewertet werden, so wird geprüft, ob der bereits gestartete Unterplan noch läuft oder zurückgewiesen wurde. In diesen Fällen verhält sich ein `ifThenElse`-kontrollierter Plan passiv.

Ist der Unterplan in einem neu-startbaren Zustand, führt die **ITEP**-Prozedur einen Neustart des Plans und dessen Aktivierung durch.

12.4.7 Die `cyclical#`-Prozedur

Zyklische Pläne dienen zur mehrfachen, iterativen Ausführung eines Unterplans. Dabei spezifiziert ein zyklischer Plan ein wiederkehrendes Zeitintervall, in dem der Unterplan gestartet werden kann. Eine zyklische Plankontrolle spezifiziert weiterhin, wieviele erfolgreiche Durchläufe des Unterplans notwendig sind, damit der übergeordnete Plan seine Ausführung beenden kann. Die `cyclical#` Prozedur, die in Listing 12.59 dargestellt ist, bildet dieses Verhalten ab.

```
cyclical#(iname; var Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
begin
  itlif   (  isInactive(AS, nameGen(selectFirstSub(iname), iname))
           ∨ isRejected(AS, nameGen(selectFirstSub(iname), iname))
           ∨ isAborted(AS, nameGen(selectFirstSub(iname), iname)))
        ∧ isInTimeFrame(ASH, AC, asbru(nameSel(iname)) .type .cta) then
    begin
      inactive#(selectFirstSub(iname); Tick, EAGG, PDH, ASH, AS, AC, RC) ||s
      begin
        AS[iname] := activated,
        PC        := PC[nameGen(selectFirstSub(iname), iname), true],
        Tick      := false;
        activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
```

```
      end
    end
  else itlif (  isCompleted(AS, nameGen(selectFirstSub(iname), iname))
              ∨ hasCompletedRounds(iname, RC)) then
        begin
          AS[iname] := completed,
          Tick := false;
        end
  else itlif (  isCompleted(AS, nameGen(selectFirstSub(iname), iname))
              ∧ isInTimeFrame(ASH, AC, asbru(nameSel(iname)) .type .cta)) then
        begin
          inactive#(nameGen(selectFirstSub(iname), iname);
                    Tick, EAGG, PDH, ASH, AS, AC, RC) ||s
          begin
            AS[iname] := activated,
            PC        := PC[nameGen(selectFirstSub(iname), iname), true],
            RC        := RC[iname, RC[iname] + 1],
            Tick      := false;
            activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
          end
        end
  else begin
      AS[iname] := activated;
      activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)
  end
end;
```

Listing 12.59: Cyclical#-Prozedur

Zunächst wird dazu geprüft, ob der Unterplan im Zustand `rejected`, `aborted` oder `inactive` ist und der aktuelle Zeitpunkt in einer Iteration des Startintervalls liegt. In diesem Fall wird der Unterplan gestartet.

Andernfalls wird getestet, ob der Zustand des Unterplans `completed` ist und gleichzeitig mit diesem letzten Durchlauf die Zahl der notwendigen Iterationen erreicht wurde. Dann darf sich der `cyclical`-kontrollierte Plan beenden.

Darf er dies nicht, weil die notwendige Zahl an Durchläufen noch nicht erreicht wurde, so soll der Unterplan erneut gestartet werden, dabei allerdings der Zähler der erfolgreichen Durchläufe hochgesetzt werden. Dieser erneute Start der Unterplans ist nur möglich, wenn damit das Startintervall eingehalten wird.

Ist der aktuelle Zeitpunkt in keinem Startintervall oder läuft der Unterplan noch, dann wartet die `cyclical`-Plankontrolle ab.

13 Korrektheit der Implementierung

Die Implementierung der Asbru-Semantik in das interaktive Verifikationswerkzeug KIV erlaubt es, computerunterstützte Beweise zu führen um die Korrektheit von Aussagen über medizinischen Leitlinien zu verifizieren, die in Asbru modelliert sind. Dabei ist zu beachten, dass die Ergebnisse der interaktiven Verifikation auch denen entsprechen müssen, die die Semantik fordert. Angenommen, eine Aussage soll gezeigt werden, die verlangt, dass Chemotherapie im Falle von Brustkrebs bei über 70 Jährigen nicht angewendet wird. Dann soll eine computergestützte Verifikation für diese Aussage den gleichen Wahrheitswert ermitteln, wie es eine händische Anwendung der SOS Regeln auf Papier tun würde. Um dies zu erreichen soll in diesem Kapitel ein Nachweis geführt werden, dass die Implementierung sich exakt so verhält wie die Semantik.

Die Semantik von Asbru ist in Form von Abläufen gegeben. Im KIV-System wird die Semantik durch parallele, temporallogische Programme implementiert. Die Semantik dieser Temporallogik ist ebenfalls als Ablaufsemantik definiert. Ziel dieses Kapitels ist der Nachweis, dass die Abläufe der Asbru-Semantik und der in KIV implementierten parallelen Programme isomorph zueinander sind. Zu diesem Zweck wird argumentiert, dass die jeweils ausgeführten Schritte der Semantik und der Implementierung von isomorphen Zuständen in isomorphe Zustände führen. Außerdem wird eine bijektive Abbildung von Zuständen der Semantik in Zustände des implementierten Systems angegeben, so dass jeder Zustand des einen Ablaufs in einen des jeweils anderen Ablaufs abgebildet werden kann.

13.1 Beweisidee

Um die Isomorphie der Abläufe nachzuweisen wird das Beweisprinzip des kommutierenden Diagramms verwendet. Dazu wird eine Isomorphie der Zustände der Semantik und der Implementierung angegeben. Ist σ ein Zustand der Semantik, dann wird der dazu isomorphe Zustand ρ der Implementierung gesucht. Im Anschluss werden die Zustandsübergänge der Semantik, die vom Zustand σ ausgehen gesucht und deren Endzustände berechnet. Gleichzeitig werden alle Zustandsübergänge der Implementierung berechnet, die vom Zustand ρ (der isomorph zu σ ist) ausgehen. Zu beweisen ist dann, dass die jeweiligen Ergebnisse wieder isomorph zueinander sind. Grafisch wird diese Idee in Abbildung 13.1 dargestellt. Ausgehend von einem Zustand auf der Semantik-Ebene σ wird mittels einer Abbildungsfunktion der isomorphe Zustand der Implementierung gesucht. Beide Zustände werden als Eingabe in einen Ausführungsschritt genommen. Dann muss nachgewiesen werden, dass die jeweiligen Ergebnisse dieser Ausführung wieder isomorph zueinander sind.

Mit diesem Beweis kann gezeigt werden, dass die Implementierung und die Semantik modulo der noch anzugebenden bijektiven Abbildung identische Schritte durchführen. Da Abläufe der Semantik und der Implementierung jeweils Abfolgen solcher Zustandsübergänge sind, folgt aus dem oben skizzierten Beweis dass die kompletten Abläufe der Semantik und der Implementierung isomorph zueinander sind.

Um diesen Beweis durchzuführen sind nachstehende Schritte notwendig. Zunächst wird eine Abbildung des abstrakten Plannamen Datentyps **plan-name** bzw **instance-name** der Semantik in konkrete Datentypen der Implementierung angegeben. Dazu wird eine bijektive Abbildungsfunktion spezifiziert, die Plannamen der einen Ebene in die andere übersetzen kann. Im Anschluss daran wird eine Bijektion zwischen Zuständen der Semantik und denen

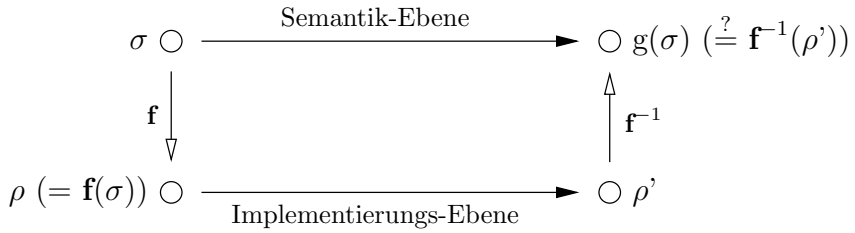


Abbildung 13.1: Kommutierendes Diagramm Implementierung/Semantik

der Implementierung angegeben, so dass der Begriff der isomorphen Zustände definiert werden kann. Zuletzt wird nachgewiesen, dass Zustandsübergänge der Implementierung denen der Semantik entsprechen. Ein Nebenergebnis dieses Nachweises ist, dass die Semantik (und damit auch die Implementierung) von Asbru deterministisch ist. Damit gilt dann die Korrektheit der Einbettung als bewiesen.

13.2 Abbildung der Plannamen und Instanznamen

Voraussetzung für den Beweis der Korrektheit der Einbettung der Asbru-Semantik ist eine Abbildung der abstrakten Plan- und Planinstanznamenstypen der Semantik auf die konkreten Datentypen der Implementierung. Plannamen werden implementiert durch beliebige Zeichenketten. Instanznamen werden abgebildet durch den Datentyp der Instanznamen der Implementierung.

Für die Abbildung zwischen Semantik und Implementierung wird die Funktion **f** definiert, die die Plannamen der Semantik in die Implementierung abbildet. Dazu invers ist die Funktion $\mathbf{f}^{-1}$, die die Zeichenketten der Implementierung auf Plannamen abbildet. Gleichzeitig werden die Funktion **f2** und die zugehörige inverse Funktion $\mathbf{f2}^{-1}$ definiert, die die Abbildung von Planinstanznamen übernehmen.

Die Syntax der dazu notwendigen Funktionen **f** und **f2** ist wie folgt definiert:

f	:	**plan-name** $\mapsto$ `string`
f2	:	**instance-name** $\mapsto$ `instance-name`
$\mathbf{f}^{-1}$	:	`string` $\mapsto$ **plan-name**
$\mathbf{f2}^{-1}$	:	`instance-name` $\mapsto$ **instance-name**

Da der Typ der Plannamen **plan-name** ein unspezifizierter Datentyp ist, kann für die Abbildungsfunktionen für diesen Typ nur zweierlei gefordert werden. Einerseits muss die Anwendung von Abbildung und inverser Abbildung nacheinander dazu führen, dass das Ergebnis der ursprünglichen Eingabe entspricht. Ist also **SK** ein **plan-name**, dann gilt:

$$\mathbf{f}^{-1}(\mathbf{f}(\mathbf{SK})) = \mathbf{SK}$$

Die zweite Forderung ist die Bijektivität der Abbildung. Diese Forderung bedeutet, dass die Funktion nicht zwei unterschiedliche Plannamen auf die gleiche Zeichenkette abbildet. Sind $\mathbf{SK}_1$ und $\mathbf{SK}_2$ zwei Plannamen, so soll **f** diese genau dann auf identische Zeichenketten abbilden, wenn die Plannamen gleich sind:

$$\mathbf{f}(\mathbf{SK}_1) = \mathbf{f}(\mathbf{SK}_2) \leftrightarrow \mathbf{SK}_1 = \mathbf{SK}_2$$

Die Abbildung von Instanznamen der Semantik auf Instanznamen der KIV-Ebene wird unter Zuhilfenahme der Abbildung von Plannamen auf Zeichenketten spezifiziert. Ein Instanzname kann interpretiert werden als eine Liste von Plannamen. Dies ergibt sich aus dem

Aufbau eines Instanznamens mittels der **nameGen** Konstruktoren. Diese können als Listen-Additionsoperator angesehen werden. Die Plannamen in der „Liste“ geben den kompletten Aufrufpfad an, beginnend mit dem Plannamen des eigentlichen Plans über den übergeordneten Plan bis hin zum Wurzelplan. In der Implementierung werden analog spezifizierte Instanznamen verwendet:

$$\mathbf{f2}(\mathbf{nameGen}(\mathbf{SK}, \mathbf{iname})) = \mathbf{nameGen}(\mathbf{f}(\mathbf{SK}), \mathbf{f2}(\mathbf{iname}))$$

Dazu muss noch die Übersetzungsregel für den Superplan einer Hierarchie angegeben werden. Dieser hat keinen übergeordneten Plan. Daher kann sein Instanzname auch nicht aus seinem Plannamen und dem Instanznamen des (nicht-existierenden) Überplans gebildet werden. Stattdessen wird dazu das **superGen** Konstrukt der KIV-Ebene verwendet:

$$\mathbf{f2}(\mathbf{superGen}(\mathbf{SK})) = \mathbf{superGen}(\mathbf{f}(\mathbf{SK}))$$

Die Umwandlung eines Instanznamens der KIV-Ebene in einen Planinstanznamen der Semantik erfolgt rekursiv:

$$\mathbf{f2}^{-1}(\mathbf{nameGen}(\mathsf{sk}, \mathbf{iname})) = \mathbf{nameGen}(\mathbf{f}^{-1}(\mathsf{sk}), \mathbf{f2}^{-1}(\mathbf{iname}))$$

Analog dazu wird die Inversion der Instanznamen der Top-Level Pläne durchgeführt:

$$\mathbf{f2}^{-1}(\mathbf{superGen}(\mathsf{sk}_1)) = \mathbf{superGen}(\mathbf{f}^{-1}(\mathsf{sk}_1))$$

Damit wird vollständig beschrieben, wie Plan- und Instanznamen der Semantik und der Implementierung ineinander umgerechnet werden können.

13.3 Zustandsabbildung

In diesem Abschnitt soll eine Bijektion zwischen Zuständen der Asbru Semantik und der Implementierung angegeben werden. Mit dieser wird der Begriff der isomorphen Zustände definiert werden. Damit werden die vertikalen Abbildungen in Abbildung 13.1 definiert, so dass im Anschluss daran die Zustandsübergänge untersucht werden können.

Die Algebren, die zur Auswertung der Zustände von Implementierung und Semantik herangezogen werden sind identisch. Die Datentypen auf Semantik- und KIV-Ebene sind frei erzeugte Datentypen und Aufzählungstypen, einmal mittels EBNF erzeugt und einmal über algebraische KIV Spezifikationen. Abgesehen von diesen Typen werden nur die prädikatenlogisch spezifizierten dynamischen Funktionen verwendet, die auf Semantik- und KIV-Ebene monomorph zueinander spezifiziert sind.

Lediglich im Falle des Hierarchiezustands **hierarchy-state** gilt, dass dieser keine direkte Entsprechung in der KIV-Ebene hat. Der Hierarchiezustand in der Semantik ist eine Funktion, die Planinstanznamen **iname** auf Planstati θ abbildet. Ein Planstatus wiederum ist ein Tripel, das aus einem Planzustand, einem booleschen Flag sowie einer natürlichen Zahl besteht. In der Syntax-Ebene wird diese Funktion aus Effizienzgründen durch drei Funktionen ersetzt. Eine bildet einen Instanznamen ab auf einen Planzustand, die Zweite bildet den Instanznamen ab auf ein boolesches Flag, die Dritte bildet Instanznamen auf natürliche Zahlen ab. Die häufigen Zugriffe auf die einzelnen Komponenten des Planstatus sowie die Aktualisierungen sind durch die Implementierung in drei getrennten Funktionen leichter durchführbar.

Es muss also eine Funktion die auf ein Tripel von Werten abbildet ersetzt werden durch ein Tripel von Funktionen. Dies soll im Folgenden formal geschehen.

Sei σ ein Zustand der Semantik, für den gilt $\sigma = (\mathbf{ASH}, \mathbf{HS}, \mathbf{PDH}, \mathbf{EAGG}, \mathbf{Tick}, \mathbf{AC})$. Aus diesem Zustand können die Komponenten mit den definierten Selektorfunktionen herausselektiert werden. Auf der syntaktischen Ebene werden Variablenbelegungen ρ verwendet.

Die Belegung einer Variablen AS in der Belegung ρ kann mit der Formel ρ(AS) selektiert werden.

Im Fall des Hierarchiezustandes **hierarchy-state**, der eine Funktion von Planinstanznamen nach Planstatus ist, wird ein semantischer Lookup definiert. Dieser hat die gleiche Syntax wie der Selektor, der aus einer **hierarchy-state** Variablen eine Planzustandskomponente selektiert:

$$\mathbf{HS}[\mathbf{iname}]$$

Diese definierte Funktion selektiert den Wert des Planstatus, der zu der Komponente **iname** in **HS** gespeichert ist. Analog dazu wird der semantische Lookup definiert:

$$\sigma.\mathrm{HS}[\mathbf{iname}]$$

Die neu definierte Funktion selektiert den Wert der in der **HS** Komponente des Zustands unter dem Schlüssel **iname** gespeichert ist.

Damit ist es nun möglich, den Isomorphismus zu definieren. Ein Zustand der Semantik σ und eine Variablenbelegung ρ sind isomorph, wenn folgende Bedingungen gelten. Zunächst müssen die Variablenbelegungen der Datentypen, die in Semantik und Implementierung identisch sind gleich sein. Dazu muss für alle Planinstanznamen **iname**, alle Zeichenketten **SK** und alle Uhrzeiten **AC** gelten:

$$\begin{array}{lcl}
\sigma.\mathrm{ASH}[\mathbf{AC}][\mathbf{iname}] & = & \rho(\mathrm{ASH}[\mathbf{AC}][\mathbf{f2}(\mathbf{iname})]) \\
\sigma.\mathrm{PDH}[\mathbf{SK}] & = & \rho(\mathrm{PDH}[\mathbf{SK}]) \\
\sigma.\mathrm{EAGG}[\mathbf{iname}] & = & \rho(\mathrm{EAGG}[\mathbf{f2}(\mathbf{iname})]) \\
\sigma.\mathrm{Tick} & = & \rho(\mathrm{Tick}) \\
\sigma.\mathrm{AC} & = & \rho(\mathrm{AC})
\end{array}$$

Zusätzlich müssen der Hierarchiestatus auf Semantikseite in definierter Beziehung mit Asbru-Zustand, Plankommunikation und Rundenzähler auf der Implementierungsseite stehen. Konkret müssen für alle Plannamen die Werte des semantischen Planstatus auf die drei Funktionen der Implementierung abgebildet werden. Diese Funktionen sind erstens der Asbru-Zustand (**asbru-state**), der Planinstanznamen auf Planzustände abbildet, zweitens der Rundenzähler (**round-state**), der die Zahl der erfolgreichen Iterationen zyklischer Pläne speichert, sowie drittens die Plankommunikations-Variable **plan-com**, die die Aktivierungssignale aller Pläne speichert. Für diese Funktionen müssen für alle Planinstanznamen **iname** die nachfolgenden Bedingungen gelten:

$$\begin{array}{lcl}
\sigma.\mathrm{HS}[\mathbf{iname}].\mathbf{state} & = & \rho(\mathrm{AS})[\mathbf{f2}(\mathbf{iname})] \\
\sigma.\mathrm{HS}[\mathbf{iname}].\mathbf{rounds} & = & \rho(\mathrm{roundvar})[\mathbf{f2}(\mathbf{iname})] \\
\sigma.\mathrm{HS}[\mathbf{iname}].\mathbf{activate} & = & \rho(\mathrm{PC})[\mathbf{f2}(\mathbf{iname})]
\end{array}$$

Sind all diese Bedingungen wahr, so gelten die Zustände als isomorph zueinander. Es bleibt nun zu zeigen, dass die Abarbeitung von Ausführungsregeln in der Semantik und der Implementierung von isomorphen Zuständen wieder in isomorphe Zustände führen.

13.4 Korrektheit der Prädikate

Die SOS-Regeln der Semantik verwenden in der Vorbedingung Prädikate, um den Ausgangszustand von dem aus eine Transition startet zu definieren. Aus Gründen der Klarheit sollen gleich benannte Prädikate mit identischer Semantik auch in der Implementierung als Vorbedingung zu der Ausführung der Asbru-Prozeduren verwendet werden. Aus Effizienzgründen sind diese Prädikate allerdings in der Implementierung teilweise anders implementiert als

in der Semantik. Ein Beispiel dafür ist das Prädikat **hasInactiveSub**. Auf Semantik-Seite wurde dieses Prädikat deklarativ spezifiziert. Es verwendet einen Existenzquantor um zu spezifizieren, dass das Prädikat wahr ist, genau dann wenn ein Unterplan existiert, der im Zustand `inactive` ist. Dies ist die natürlichste Möglichkeit zu formalisieren, dass ein inaktiver Unterplan existiert. Jedoch sind Existenzquantoren in der Implementierung hinderlich, da es schwierig ist, existenzquantifizierte Formeln mit den automatisierungs-Konzepten des Theorembeweisers zu handhaben. Daher wird implementierungsseitig eine prozedurale Spezifikation des Prädikats angegeben. Diese prüft rekursiv alle Unterpläne ob diese im Zustand `inactive` sind und liefert beim ersten Treffer den Wahrheitswert **true** zurück. Es ist nicht offensichtlich, dass die zwei Prädikatdefinitionen semantisch gleichwertig sind. Dazu zunächst die Definition des Prädikats auf Semantikebene:

$$\begin{aligned} &\mathbf{hasInactiveSub}(\mathbf{HS}, \mathbf{iname}) \\ \leftrightarrow\ &\exists\ \mathbf{SK}.\quad \mathbf{SK} \in \texttt{asbru}(\mathbf{nameSel}(\mathbf{iname}))\mathbf{.subplans} \\ &\qquad \wedge\ \mathbf{isInactive}(\mathbf{HS}, \mathbf{nameGen}(\mathbf{SK}, \mathbf{iname})) \end{aligned}$$

Die Implementierung dieses Prädikates hat die folgende Form:

$$\begin{aligned} &\mathbf{hasInactiveSub}(\mathbf{AS}, \texttt{iname}) \\ \leftrightarrow\ &\mathbf{hasInactiveSub}(\mathbf{AS}, \texttt{asbru}(\mathbf{nameSel}(\texttt{iname}))\mathbf{.subplans}, \texttt{iname}) \end{aligned}$$

$$\neg\ \mathbf{hasInactiveSub}(\mathbf{AS}, [], \texttt{iname})$$

$$\begin{aligned} &\mathbf{hasInactiveSub}(\mathbf{AS}, \mathbf{SK}_1 + \mathbf{STRX}, \texttt{iname}) \\ \leftrightarrow\ &\mathbf{isInactive}(\mathbf{AS}, \mathbf{nameGen}(\mathbf{SK}_1, \texttt{iname}) \vee \mathbf{hasInactiveSub}(\mathbf{AS}, \mathbf{STRX}, \texttt{iname})) \end{aligned}$$

Die Definition auf Implementierungsebene ist prozedural. Das eigentliche Prädikat hat eine Typisierung die ähnlich zu dem der Semantik ist, nur dass statt einem Hierarchie-Zustand ein Asbru-Zustand als Parameter angegeben wird. Um die Spezifikation zu implementieren, soll die Liste der Unterpläne rekursiv durchlaufen werden. Um dieses rekursive Durchlaufen zu ermöglichen, wird ein Hilfsprädikat verwendet. Dieses hat als Typisierung den Asbru-Zustand, dann die Liste der Unterpläne, die noch durchlaufen werden muss und zuletzt den Instanznamen des aufrufenden Plans. Durch letzteren können die Instanznamen der Unterpläne erzeugt und deren Zustand rekursiv geprüft werden.

Da die Äquivalenz dieser Formalisierungen nicht trivial ersichtlich ist, wurde ein Beweis geführt um diese nachzuweisen. Um Computerunterstützung für den Beweis nutzen zu können, wurde dieser Implementierungsseitig geführt. Da das Prädikat **hasInactiveSub** Semantik-seitig nur auf die Status Komponente des Hierarchie-Status zugreift, ist es nicht notwendig, die Plankommunikations-Komponente bzw. den Rundenzähler in die Übersetzung mit einzubeziehen. Damit bleibt folgende Formel als Beweisverpflichtung stehen:

$$\begin{aligned} &\mathbf{hasInactiveSub}(\mathbf{AS}, \texttt{iname}) \\ \leftrightarrow\ &\exists\ \mathbf{SK}_1.\quad \mathbf{SK}_1 \in \texttt{asbru}(\mathbf{nameSel}(\texttt{iname}))\mathbf{.subplans} \\ &\qquad \wedge\ \mathbf{isInactive}(\mathbf{AS}, \mathbf{nameGen}(\mathbf{SK}_1, \texttt{iname})) \end{aligned}$$

Mit dem Beweis dieser Aussage wird gezeigt, dass die Prädikate sich trotz unterschiedlicher Spezifikation in Semantik und Implementierung gleich verhalten. Entsprechende bewiesene Aussagen sind für alle in der Semantik verwendeten Prädikate formuliert worden, die in Abschnitt C definiert wurden. Die Beweise dieser Prädikate wurden mit Hilfe von KIV geführt.

13.5 Abbildung der Prozeduren

Dieses Kapitel soll die Korrektheit der Einbettung von Asbru in das Verifikationswerkzeug KIV vorstellen. Dazu wurde zunächst eine Abbildung der Plannamen der Semantik auf Plannamen in der Implementierung angegeben. Im Anschluss wurden Abbildungen zwischen den

Zuständen der Implementierung und der Semantik angegeben, so dass eine Aussage darüber getroffen werden kann, wann ein Zustand der Semantik isomorph zu einem Zustand der Implementierung ist. Mit dieser Isomorphie wurde gezeigt, dass Prädikate der Semantik und der Implementierung semantisch gleichwertig sind. Der letzte Schritt, der vor dem Korrektheitsbeweis noch offen ist, ist die Abbildung der Asbru-Systeme der Semantik auf parallele Programme der Implementierung. Diese Abbildung wird in diesem Abschnitt beschrieben.

In der Implementierung existieren verschiedene Prozeduren, die jeweils eine Teilfunktionalität der SOS-Regeln der Semantik implementieren. Beispielsweise werden alle Zustandsübergänge, die im Zustand `considered` ausgeführt werden können, durch die Prozedur `considered#` beschrieben. Wechselt der Zustand des Plans nach `possible`, so muss im Folgenden die Prozedur `possible#` ausgeführt werden.

Für diese Abbildung soll die Funktion **f** verwendet werden, und diese soll ein Paar aus semantischem Zustand und Asbru-System abbilden auf eine paralleles Programm. Dazu wird die Syntax der **f** Funktion genau so angegeben, nämlich dass eine Eingabe bestehend aus Semantischem Zustand und Asbru-System auf ein paralleles Programm abgebildet wird:

$$\mathbf{f} : \sigma \times \textbf{asbru-system} \mapsto \textbf{parprog}$$

Da ein Asbru-System rekursiv aufgebaut ist, muss auch die Funktion **f** rekursiv spezifiziert werden. Zunächst werden die Basisfälle beschrieben, in denen ein Zustand und eine einzelne Planinstanz in ein paralleles Programm überführt werden soll. Dazu wird angenommen, dass der Zustand der Semantik σ isomorph ist zum Zustand ρ der Implementierung.

$$\begin{array}{lcl}
\textbf{isConsidered}(\sigma, \textbf{iname}) & \rightarrow & \mathbf{f}(\sigma, \textbf{iname}) = \texttt{considered\#}(...) \\
\textbf{isPossible}(\sigma, \textbf{iname}) & \rightarrow & \mathbf{f}(\sigma, \textbf{iname}) = \texttt{possible\#}(...) \\
\textbf{isReady}(\sigma, \textbf{iname}) & \rightarrow & \mathbf{f}(\sigma, \textbf{iname}) = \texttt{ready\#}(...) \\
\textbf{isActivated}(\sigma, \textbf{iname}) & \rightarrow & \mathbf{f}(\sigma, \textbf{iname}) = \texttt{activated\#}(...) \\
\textbf{isSuspended}(\sigma, \textbf{iname}) & \rightarrow & \mathbf{f}(\sigma, \textbf{iname}) = \texttt{suspended\#}(...)
\end{array}$$

In dieser Abbildung werden die Zustände `inactive`, `rejected`, `aborted` und `completed` nicht berücksichtigt. Der Grund dafür liegt darin, dass von diesen Zuständen aus weder in der Semantik noch in der Implementierung Zustandsübergänge geschehen können. Dazu können in Abläufen der Asbru-Semantik, die in den definierten Ausgangszuständen starten, Kombinationen nicht auftreten, in denen eine Planinstanz einen dieser vier Zustände erreicht und noch Teil des Asbru-Systems ist.

In dieser Abbildungsmatrix werden die Funktionsparameter der Funktionen aus Gründen der Lesbarkeit nicht aufgeführt. Für alle oben genannten Abbildungen gilt jedoch, dass der Methodenaufruf mit Parametern durchgeführt wird. Beispielhaft soll dies an der `considered` Prozedur gezeigt werden, wobei die gleichen Methodensignaturen für alle Prozeduren gelten:

$$\texttt{considered\#}(\ \mathbf{f}(\textbf{iname}); \rho(\texttt{Tick}), \rho(\texttt{EAGG}), \rho(\texttt{PDH}), \rho(\texttt{ASH}), \rho(\texttt{AS}), \rho(\texttt{AC}), \rho(\texttt{PC}), \rho(\texttt{roundvar}))$$

Das heißt, die Prozeduren der Implementierung erhalten als ersten Parameter den Instanznamen des Plans. Danach wird der boolesche Wert **Tick** übergeben. Über diesen kommuniziert die Prozedur nach außen, ob der in einem Schritt durchgeführte Zustandsübergangswechsel aktiv war oder nicht. Dies hat Bedeutung für die Frage, ob im Anschluss an diesen Schritt ein Makro-Schritt durchgeführt werden kann oder nicht. Nach der **Tick** Variablen wird die Umgebungssignalsammlung übergeben. In dieser werden alle Umgebungssignale gesammelt. Umgebungssignale kommen vom medizinischen Personal und beschreiben beispielsweise welchen Unterplan eines `anyorder` kontrollierten Plans ein Arzt starten möchte. Auch

das Übersteuern oder Verzögern der Auswertung von Asbru-Bedingungen wird über diese Signalaggregation gesteuert. Nach dieser Signalsammlung ist der nächste Parameter die elektronische Krankenakte des Patienten. Der nächste Parameter ist die Asbru-Zustands-Historie. Diese speichert eine Ansammlung aller Asbru-Zustände der Vergangenheit und erlaubt somit, festzustellen, ob eine Planinstanz bereits einen bestimmten Zustand durchlaufen hat. Nach der Asbru-Zustands-Historie ist der nächste Parameter der Asbru-Zustand der die aktuelle Information über die Planzustände aller Pläne der Hierarchie enthält. Nach diesem erhält die Prozedur als nächsten Parameter die Uhrzeit, gefolgt von der Plankommunikations-Information. Diese ist eine Sammlung sämtlicher Aktivierungssignale aller Pläne der Plan-Hierarchie. Den letzten Parameter bildet der Rundenzähler, der die Anzahl der erfolgreichen Durchläufe der zyklischen Pläne speichert.

Nachdem nun die Abbildung einzelner Planinstanzen auf parallele Prozeduren durchgeführt wurde, soll als nächstes die Abbildung von Asbru-Systemen beschrieben werden. Dabei werden die parallelen Asbru-Systeme auf parallele Prozeduren abgebildet. Das Terminierungssymbol der Asbru-Systeme (ε) wird auf das temporallogische Konstrukt **last** abgebildet, da beide keine Zustandsübergänge mehr ausführen können.

$$\begin{array}{lcl} \mathbf{f}(\sigma, \mathbf{sys}_1 \|_s \mathbf{sys}_2) & = & \mathbf{f}(\sigma, \mathbf{sys}_1) \|_s \mathbf{f}(\sigma, \mathbf{sys}_2) \\ \mathbf{f}(\sigma, \varepsilon) & = & \mathbf{last} \end{array}$$

Damit können alle Asbru-Systeme auf parallele Prozeduren abgebildet werden. Die Umkehrung dieser Abbildung ist wie folgt definiert:

$$\begin{array}{lcl} \mathbf{f}^{-1}(\texttt{proc\#(iname;...)}) & = & \mathbf{f}^{-1}(\texttt{iname}) \\ \mathbf{f}^{-1}(\mathbf{parproc}_1 \|_s \mathbf{parproc}_2) & = & \mathbf{f}^{-1}(\mathbf{parproc}_1) \|_s \mathbf{f}^{-1}(\mathbf{parproc}_2) \\ \mathbf{f}^{-1}(\mathbf{last}) & = & \varepsilon \end{array}$$

Somit können nun Prozeduren und Asbru-Systeme aufeinander abgebildet werden. Damit ist das letzte Teilstück der Abbildungen zwischen Semantik und Implementierung angegeben. Somit kann nun mit dem eigentlichen Beweis der Korrektheit der Implementierung fortgefahren werden.

13.6 Korrektheit der Implementierung

Bisher wurden in diesem Kapitel die Grundlagen für den Korrektheitsbeweis der Implementierung gelegt. Zunächst wurde angegeben, wie Plannamen und Planinstanznamen abgebildet werden können. Als nächstes wurde der Isomorphiebegriff zwischen Zuständen der Semantik und Zuständen der Implementierung definiert. Daraufhin konnte nachgewiesen werden, dass die in Semantik und Implementierung jeweils gleich benannten Prädikate semantisch äquivalent sind und zuletzt wurde eine Abbildung zwischen Prozeduren der Implementierung und Asbru-Systemen der Semantik angegeben. Nun muss nachgewiesen werden, dass die Schritte der Implementierung denen der Semantik entsprechen. Dieser Nachweis wird im Verifikationswerkzeug KIV geführt.

Beispielhaft soll dieser Nachweis nun anhand von drei typischen Zustandsübergängen beschrieben werden. Das erste Beispiel ist eine der Terminierungsregeln in der `Selection`-Phase, das zweite Beispiel ist der Übergang zwischen den höher priorisierten und den weniger hoch priorisierten SOS-Regeln im `activated`-Zustand und als drittes soll ein Zustandsübergang des sequentiellen Kontrolltyps gezeigt werden. Die übrigen Beweise sind in Abschnitt B beschrieben.

13.6.1 Zustandsübergang von `considered` nach `rejected`

Der Zustandsübergang, der hier betrachtet werden soll beschreibt die Zurückweisung einer Planinstanz im Zustand `considered`. Ist in diesem Zustand die **filter**-Bedingung nicht erfüllbar, so wechselt die Planinstanz in den Zustand `rejected` und terminiert. Dies wird in der Regel 9.1 beschrieben. Die zu beweisende Sequenz in KIV hat folgende Form:

$$
\begin{array}{l}
\quad \mathbf{isConsidered}(\mathbf{AS}, \texttt{iname}), \\
\quad\ \mathbf{shouldRejectFilter}(\texttt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AC}) \\
\quad \vee\ \mathbf{parent\_terminated}(\mathbf{AS}, \texttt{iname}), \\
\quad [\texttt{considered\#}(\texttt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})] \\
\vdash \quad \lceil \mathbf{AS},\ \mathbf{Tick} \rceil \\
\quad \wedge \neg\ \mathbf{Tick}' \\
\quad \wedge\ \mathbf{AS}' = \mathbf{AS}[\texttt{iname}, \texttt{rejected}] \\
\quad \wedge \circ\ \mathbf{last};
\end{array}
$$

Die zugehörige SOS-Regel, die den Übergang in der Semantik beschreibt, sieht wie folgt aus:

$$
\frac{\begin{array}{l}\quad\ \mathbf{isConsidered}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\quad \mathbf{shouldRejectFilter}(\sigma, \mathbf{iname}) \\ \quad \vee\ \mathbf{parent\_terminated}(\mathbf{HS}, \mathbf{iname})\end{array}}{\mathbf{iname} \quad \sigma, \mathbf{Tick}(\sigma[\mathbf{iname}, \texttt{rejected}]) \quad \varepsilon}
$$

Die Vorbedingung dieser Beweisverpflichtung ist gegeben durch die Prädikate **isConsidered** und **shouldRejectFilter**. Diese beschreiben den Zustand, in dem die Anwendung der `considered#` Prozedur zum beschriebenen Ergebnis kommen muss. Die Vorbedingung entspricht dabei genau der Vorbedingung der Transitionsregel.

Die Abbildungsfunktion für Prozeduren verlangt, dass eine Planinstanz, deren Plan-Zustand `considered` ist, auf die `considered#` Prozedur abgebildet werden soll. Dies ist hier geschehen. Zu zeigen ist nun, dass beide Regeln bei der Abarbeitung zu einem identischen Ergebnis kommen. Die Semantik geht von expliziten Zustandsübergängen aus, das heißt, in den SOS-Regeln der Semantik wird angegeben, wie der neue Zustand aus dem alten berechnet wird. Dadurch wird automatisch festgelegt, welche Komponenten des Zustands sich nicht ändern. Dies muss auf TL-Seite explizit gemacht werden. Dazu wird das Frame-Konstrukt verwendet. Die Formel $\lceil \mathbf{AS}, \mathbf{Tick} \rceil$ beschreibt einen Zustandsübergang, bei dem ausschließlich die Variablen **AS** sowie **Tick** verändert werden. Dies entspricht auch der Forderung der SOS-Regel, die alle Komponenten des Zustands außer dem Hierarchiezustand **HS** und der **Tick**-Komponente unverändert lässt. Der neue Zustand, der von der SOS-Regel gefordert wird, verlangt, dass die **Tick** Komponente des Zustands auf den Wert **false** gesetzt wird. Dies wird ebenso in der Beweisverpflichtung durch die Formel $\neg\ \mathbf{Tick}'$ verlangt. Die SOS-Regel gibt an, dass der Zustand des Plans auf `rejected` geändert werden soll. Eine analoge Forderung enthält die Beweisverpflichtung durch die Formel $\mathbf{AS}' = \mathbf{AS}[\texttt{iname}, \texttt{rejected}]$. Diese Formel verlangt, dass der neue Asbru-Zustand aus dem alten so generiert wird, dass das den Plan betreffende Feld auf den Wert `rejected` geändert wird und alles weitere unverändert bleibt. Die letzte Forderung der SOS-Regel ist, dass das Asbru-System nach Ausführung der Regel terminieren soll. Die Beweisverpflichtung

sieht genau dies vor, indem für den nächsten Schritt das System **last** verlangt wird. Gemäß der Abbildungsregel für Asbru-Systeme entspricht dies dem ε System.

Mit dem Beweis dieser Regel ist somit der Nachweis erbracht, dass die SOS-Regel korrekt in die Implementierung eingebettet wurde. Dieser Beweis wird ebenso wie alle weiteren Beweise der `Selection`-Phase im Beweissystem KIV geführt.

13.6.2 Fortschreiten in der `Execution`-Phase

Befindet sich ein Plan in der `Execution`-Phase im Zustand `activated`, so sind vier SOS-Regeln für die weitere Ausführung zuständig. Die erste dieser Regeln ist anwendbar, wenn der Plan abbrechen soll, die zweite, wenn der Plan sich erfolgreich beenden kann und nicht abbrechen muss. Die dritte Regel ist dann anwendbar, wenn der Plan nicht terminieren, dafür aber seine Ausführung unterbrechen soll. Die letzte Regel greift, wenn der keine der vorgenannten Bedingungen gilt und die eigentliche Ausführung stattfinden soll. In den SOS-Regeln bezeichnet diese letzte Regel den Übergang der übergeordneten, normal priorisierten SOS-Regeln zu den niedriger priorisierten, plankontrollspezifischen Regeln. Die Korrektheit dieser Regel, die das Fortschreiten im Zustand `activated` beschreibt, soll nun untersucht werden. Dazu sollen die Beweisverpflichtung in KIV und die SOS-Regel zunächst aufgeführt und erklärt werden. Zunächst wird noch einmal die SOS-Regel 9.13 dargestellt.

$$\frac{\begin{array}{l}\mathbf{isActivated}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \neg\ \mathbf{shouldAbort}(\sigma, \mathbf{iname}) \\ \wedge\ \neg\ \mathbf{shouldComplete}(\sigma, \mathbf{iname}) \\ \wedge\ \neg\ \mathbf{shouldSuspend}(\sigma, \mathbf{iname}) \\ \wedge\ \mathbf{iname}\ \xrightarrow{\sigma, \sigma'}_{\mathtt{Con}}\ \mathbf{sys}\end{array}}{\mathbf{iname}\ \xrightarrow{\sigma, \sigma'}\ \mathbf{sys}}$$

Die Regel beschreibt vier Bedingungen an den Zustand, unter denen der Zustandsübergang des Systems von den niedriger priorisierten `con` Regeln bestimmt werden kann. Die Vorbedingungen und deren Abbildung auf die Prozeduren der Implementierung werden im Anschluss an die Darstellung der Beweisverpflichtung gegeben.

$$\begin{array}{l}\quad \mathbf{isActivated}(\mathbf{AS}, \mathtt{iname}), \\ \quad \neg\ \mathbf{shouldAbort}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AC}), \\ \quad \neg\ \mathbf{shouldComplete}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AC}), \\ \quad \neg\ \mathbf{shouldSuspend}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AC}), \\ \quad [\mathtt{activated\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})] \\ \vdash\ [\mathtt{body\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})];\end{array}$$

Die Beweisverpflichtung verlangt als Voraussetzung, dass vier Eigenschaften des Zustands gelten. Zunächst muss der Plan im Zustand `activated` sein.. Dies korrespondiert zur ersten Voraussetzung der SOS-Regel, die das Gleiche verlangt. An zweiter Stelle steht sowohl in der SOS-Regel als auch in der Beweisverpflichtung die Voraussetzung, dass kein Grund für den Abbruch des Plans vorliegen darf und an dritter Stelle wird verlangt, dass der Plan auch nicht erfolgreich abschließen kann. Diese Vorbedingungen sind semantisch gleichwertig. Dies gilt auch für die vierte Vorbedingung, nach der der betrachtete Plan seine Ausführung nicht

unterbrechen soll. Gemäß der Übersetzungsregel wird in diesem Fall aus dem Instanznamen der SOS-Regel die `activated#` Prozedur erzeugt.

Die SOS-Regel verlangt, dass unter diesen Umständen der Zustandsübergang durch die `con` Übergangsregeln kontrolliert wird. Auf Implementierungsseite ist zu zeigen, dass diese SOS-Regeln durch die Prozedur `body#` implementiert werden. Dies wird im folgenden bewiesen werden. Sobald dieser Beweis erbracht wurde, folgt daraus, dass die SOS-Regel korrekt implementiert wurden.

13.6.3 Starten eines Unterplans der `sequential` Plankontrolle

Als letztes Beispiel soll das Starten eines Unterplans der sequentiellen Plankontrolle detaillierter beschrieben werden. Ein sequentiell kontrollierter Plan soll einen Unterplan starten, wenn kein anderer Unterplan läuft und ein startbarer Unterplan existiert. In diesem Fall soll der erste mögliche startbare Unterplan gestartet und aktiviert werden. Dies kann in der SOS-Regel entsprechend in der Vorbedingung abgelesen werden.

$$\frac{\begin{array}{l}\mathbf{isSequential}(\mathbf{iname})\\ \wedge\ \neg\ \mathbf{hasExecutedSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \neg\ \mathbf{hasSelectedSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \mathbf{hasStartableSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \mathbf{iname}_1 = \mathbf{nameGen}(\mathbf{selectFirstStartable}(\mathbf{HS},\mathbf{iname}),\mathbf{iname})\end{array}}{\mathbf{iname}\ \ {}^{\sigma,\mathbf{Tick}(\sigma[\mathbf{iname},\ \mathtt{activated}][\underrightarrow{\mathbf{iname}_1},\ \mathtt{considered}][\mathbf{iname}_1]_{ac})}_{\mathtt{Con}}\ \ \mathbf{iname}\|_s\ \mathbf{iname}_1}$$

Die erste Vorbedingung verlangt, dass der Plan von einer sequentiellen Plankontrolle kontrolliert wird. Die nächsten beiden Bedingungen beschreiben den Umstand, dass derzeit kein anderer Unterplan des Plans läuft. Dies wird in zwei Stufen geprüft. In der ersten Stufe wird sichergestellt, dass kein Unterplan existiert, der ausgeführt wird. In der zweiten Überprüfung wird getestet, dass kein Unterplan in der `Selection`-Phase ist. Gilt dann noch die nächste Bedingung, nämlich, dass ein startbarer Unterplan existiert, so soll der erste startbare Unterplan selektiert und ausgeführt werden.

Diese Ausführung verlangt, dass der Plan seinen Zustand bei `activated` belässt, der Zustand des zu startenden Plans auf `considered` geändert wird und dessen Aktivierungssignal gesetzt wird. Im folgenden Schritt soll das Asbru-System aus der Parallelausführung des Plans sowie des gerade gestarteten Unterplans bestehen.

$$\begin{array}{l}\mathbf{isSequential}(\mathtt{iname}),\\ \neg\ \mathbf{hasExecutedSub}(\mathbf{AS},\mathtt{iname}),\\ \neg\ \mathbf{hasSelectedSub}(\mathbf{AS},\mathtt{iname})),\\ \mathbf{hasStartableSub}(\mathbf{AS},\mathtt{iname})),\\ \mathtt{sk} = \mathbf{selectFirstStartable}(\mathbf{AS},\mathtt{iname})),\\ [\mathtt{body\#}(\mathtt{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})]\\ \vdash\ \ \lceil\mathbf{AS},\ \mathbf{PC},\ \mathbf{Tick}\rceil\\ \wedge\ \neg\ \mathbf{Tick}'\\ \wedge\ \mathbf{AS}' = \mathbf{AS}\ [\mathtt{iname},\mathtt{activated}]\\ \qquad\qquad [\mathbf{nameGen}(\mathtt{sk},\mathtt{iname}),\mathtt{considered}]\\ \wedge\ \mathbf{PC}' = \mathbf{PC}\ [\mathbf{nameGen}(\mathtt{sk},\mathtt{iname}),\mathrm{true}]\\ \wedge\ \circ\ [\ \ \mathtt{activated\#}(\ \mathtt{iname};\\ \qquad\qquad\qquad\qquad \mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})\\ \qquad \|_s\ \mathtt{considered\#}(\ \mathbf{nameGen}(\mathbf{SK}_1,\mathtt{iname});\\ \qquad\qquad\qquad\qquad \mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})];\end{array}$$

Die Beweisverpflichtung verlangt, dass der Zustandsübergang, der durch die SOS-Regel vorgegeben ist, durch Implementierung korrekt wiedergegeben wird. Zunächst wird analog zur SOS-Regel gefordert, dass der fragliche Plan von einer sequentiellen Plankontrolle gesteuert wird. Wie bei der SOS Regel beschreiben die nächsten beiden Aspekte der Vorbedingung, dass kein Unterplan vorliegt, der in der `Execution`-Phase ist, ebensowenig wie ein Plan, der in der `Selection`-Phase ist. Die vierte Vorbedingung der Beweisverpflichtung verlangt die Existenz eines startbaren Unterplans und wie in der SOS-Regel wird der Name des zu startenden Unterplans mittels der **selectFirstStartable** Funktion festgelegt. Als letztes wird im Antezedent dann das System beschrieben. Dabei handelt es sich um die `body#` Prozedur die alle `con` Regeln abbildet.

Die Beweisverpflichtung stellt Anforderungen an den Zustand, der nach Ausführung eines Schritts gelten soll. Die SOS-Regel verlangt eine genau definierte Zustandsänderung, bei der die **Tick** Variable nach Abschluss des Schrittes auf den Zustand **false** gesetzt sein soll. Außerdem soll der laufende Plan seinen Zustand auf `activated` setzen, der neu gestartete Plan soll sich nach dem Schritt im Zustand `considered` befinden und das Aktivierungssignal für den Unterplan soll gesetzt werden. Diese Bedingungen werden der Reihe nach durch die Beweisverpflichtung gefordert. Zunächst wird verlangt, dass sich kein Variablenwert des Zustands verändert außer den drei genannten, dem Asbru-Zustand, der Plan Kommunikationsvariablen und der Tick Variablen. Die zweite Unterformel der Beweisverpflichtung verlangt, dass die **Tick** Variable den Wert **false** annimmt. Der Asbru-Zustand soll lediglich an den Stellen des Plans und seines Unterplans geändert werden und dort die Zustände `activated` bzw. `considered` annehmen. Bezüglich der Plankommunikation soll ausschließlich das Aktivierungssignal des Unterplans gesetzt werden. Zuletzt soll das neue System die Form `activated#`$\|_s$`considered#` haben. Dies ist dadurch bedingt, dass der Plan im Zustand `activated` und der Unterplan im Zustand `considered` ist.

13.7 Zusammenfassung

Mit diesen Beweisen wird die Korrektheit der Implementierung der Asbru-Semantik gezeigt. Es wurde eine bijektive Abbildung zwischen den Zuständen der Semantik und der Implementierung angegeben. Mit Hilfe dieser Abbildung konnte nachgewiesen werden, dass die Prädikate der Semantik und der Implementierung semantisch äquivalent sind. Zuletzt wurde für die Beweisregeln gezeigt, dass diese von isomorphen Zuständen auf isomorphe Zustände abbilden.

Für die letzte Stufe des Korrektheitsbeweises wurde der Theorembeweiser KIV zur Beweisunterstützung eingesetzt. Der Beweis der hier abgebildeten Sequenzen zur Korrektheit der Zustandsübergänge wurde mit der TL-Komponente des Beweisers geführt. Die Mehrzahl der Aussagen kann dabei mit einer einzigen interaktiven Regelanwendung - nämlich der Ausführung eines TL-Schritts geschlossen werden. Lediglich bei den Korrektheitsnachweisen der `con` Übergängen müssen manuell Regeln zur Vereinfachung angewendet werden. Der aufwendigste Beweis in diesem Zusammenhang ist der des passiven Übergangs der zyklischen Plankontrolle. Für diesen Beweis müssen 15 Interaktionen durchgeführt werden, dennoch bleibt die für den Beweis notwendige Bearbeitungszeit bei unter zwei Minuten.

Mit dem Ergebnis der Verifikation kann gefolgert werden, dass der Wahrheitswert von Aussagen unverändert bleibt, unabhängig davon, ob sie in der Semantik durch manuelle Anwendung der SOS-Regeln oder in der Implementierung durch maschinell unterstützten Beweis mit dem interaktiven Verifikationssystem KIV durchgeführt werden.

Teil IV

Formale Verifikation medizinischer Aussagen

14 Formalisierung von Aussagen mit GDL

Um Eigenschaften über Asbru Systeme zu formulieren wird die sogenannte „Goal Definition Language“ GDL verwendet. In [67] wird neben der formalen Sprache GDL selbst ein Prozess zur Formalisierung natürlichsprachlicher Aussagen unter Verwendung der GDL vorgestellt. Ergebnis dieses Prozesses sind temporallogische Formeln. Temporallogische Formeln beschreiben, genau wie die Asbru-Abläufe Mengen von Zustandsintervallen. Bewiesen werden soll nun mittels symbolischer Ausführung, dass eine Teilmengenbeziehung zwischen den Intervallen der Formel und denen des Systems gilt. Zunächst wird in Abschnitt 14.1 die Syntax der GDL vorgestellt. Im Anschluss daran wird der Formalisierungsprozess in Abschnitt 14.2 beschrieben.

14.1 Die GDL Syntax

Grundlage der GDL-formulierbaren Aussagen sind die sogenannten Asbru-Intentionen, die beispielsweise in [45, 64] beschrieben werden. Asbru erlaubt, für Pläne Intentionen festzulegen. Diese Intentionen beschreiben, welches Ziel der Plan mit seiner Ausführung verfolgt. Beispielsweise kann Aspirin verordnet werden, um Kopfschmerzen zu lindern, genauso aber kann es verordnet werden um die Blutgerinnung herabzusetzen. Welches dieser Ziele durch die Behandlung erreicht werden soll, geht aus der Verordnung alleine nicht hervor. Die Intentionen wurden in Asbru eingeführt, um dem Arzt diese Information zu vermitteln. Damit kann dieser gegebenenfalls von der vorgeschlagenen Behandlung abweichen und stattdessen eine durchführen, die die gleichen Ziele verfolgt. Intentionen in Asbru formulieren Bedingungen, die während eines Planablaufes („intermediate-state“ bzw. „intermediate-action“) oder am Ende eines Planablaufes („overall-state“ bzw. „overall-action“) ausgewertet werden. Diese Bedingungen sollen entweder gelten („maintain“), vermieden („avoid“) oder erreicht („achieve“) werden .

Intentionen können nun als Grundlage für die Verifikation von Aussagen verwendet werden. Dazu wird für einen Plan formuliert, welchen Zweck er erfüllen soll und dann nachgewiesen, dass dieser Zweck durch die Planausführung erreicht wird. Um solche Aussagen formal fassen zu können wird die GDL verwendet. Diese basiert auf den Intentionen von Asbru, erweitert diese jedoch um eine formale Semantik sowie um Konstrukte, die speziell für den Korrektheitsnachweis benötigt werden. Die Struktur von GDL Aussagen hat die Form

Goal *Name des Beweisziels*
  Precondition
    Vorbedingung der Aussage
  Time-Specification
    From
      Startzeitpunkt
    Zeitbegrenzung
  Beobachtungsmodus
    Beobachtungsformel

Jede mittels GDL formulierte Aussage erhält einen Namen, der nach dem **Goal** Schlüsselwort angegeben wird. Darauf folgt das Schlüsselwort **Precondition**. Mit diesem wird die Vor-

bedingung eingeleitet, unter der die Aussage gelten muss. Beispielsweise könnte eine Aussage nur für schwangere Patienten gelten. In diesem Fall ist diese Bedingung Teil der Vorbedingung. Mit dem Schlüsselwort **Time-specification** wird die Definition des Beobachtungszeitraums eingeleitet. Dieser Zeitraum hat einen Startpunkt, der durch das **From** Feld angegeben wird, sowie einen Endpunkt. Der Endpunkt kann entweder absolut angegeben werden oder relativ zum Startzeitpunkt. Durch das Schlüsselwort **Until** wird die Definition eines absoluten Endzeitpunkts eingeleitet, durch das Schlüsselwort **Duration** die eines relativen Endzeitpunkts. Zusätzlich gibt es die Möglichkeit, zu definieren, ob die Bedingung gelten muss, wenn der Endpunkt nicht erreicht werden kann, beispielsweise weil die Behandlung vor dem zeitlichen Erreichen des Endpunktes beendet wird. Dies wird durch die Ergänzung mittels des Schlüsselwortes **Open** erreicht. Mit **Until** spezifizierte Aussagen müssen nur bewiesen werden, wenn das Endereignis eintritt, **Open-until** müssen immer gezeigt werden. Analog gilt dies für **Duration** und **Open-duration**.

Der Beobachtungsmodus erlaubt die Definition, zu welchen Zeitpunkten innerhalb des Intervalls das Verhalten beobachtet werden muss. Beispielsweise kann der Modus **Maintain-during-period** verwendet werden, um zu spezifizieren, dass die Beobachtungsformel während des gesamten, spezifizierten Intervalls wahr ausgewertet werden muss. Das Gegenteil dazu stellt der Modus **Avoid-during-period** dar, der verlangt, dass die Aussage stets als falsch ausgewertet wird. Der Modus **Observe-during-period** verlangt, dass das Beobachtungsmuster während des Intervalls beobachtet werden muss. Es ist möglich, dabei detailliert zu spezifizieren, dass diese Beobachtung mindestens, höchstens oder genauso oft wie angegeben gemacht werden muss. Weitere Spezifikationen sind möglich und sind in [67] beschrieben. Dort wird auch eine formale Semantik für diese Konstrukte definiert.

Diese erlaubt es, eine semantikerhaltende Übersetzung der GDL-Aussagen in temporallogische Aussagen durchzuführen. Dieser Schritt ist prinzipiell automatisierbar.

Als nächstes soll nun eine Möglichkeit angegeben werden, wie informelle, medizinische Aussagen in GDL Beweisverpflichtungen übersetzt werden können. Dazu wird ein fünf-stufiger Prozess verwendet.

14.2 Formalisierung von Aussagen

Dieser Abschnitt beschreibt die schrittweise Formalisierung medizinischer Aussagen, wie sie in [67] vorgestellt wird. Wichtig ist dabei die Erkenntnis, dass die medizinischen Aussagen zur Qualitätssicherung von Leitlinien in der Regel aus vielen verschiedenen Quellen kommen. Dadurch ist die Formulierung der Aussagen in der Regel sehr heterogen und muss zunächst linguistisch vereinheitlicht und vereinfacht werden. Dies wird in Abschnitt 14.2.1 beschrieben. Danach soll der natürlichsprachliche Text in eine definierte syntaktische Struktur gebracht werden. Dies wird in Abschnitt 14.2.2 beschrieben. Daraufhin wird aus der Aussage eine GDL Aussage generiert. Dieser Prozess heißt Formalisierung und wird in Abschnitt 14.2.3 vorgestellt. Im Anschluss daran wird die GDL Aussage an die Sprache der Leitlinie angepasst, was durch Abschnitt 14.2.4 beschrieben wird. Abschließend muss dann die angepasste GDL Aussage in eine temporallogische Formel übersetzt werden. Dies wird in Abschnitt 14.2.5 behandelt.

Zur Verdeutlichung des Vorgangs wird eine Aussage aus der Brustkrebsfallstudie verwendet. Der Übersetzungsprozess ist dabei aus [48] entnommen. Dabei handelt es sich um die sogenannte Property 4. Diese lautet wie folgt:

> Appropriate use of chemotherapy and hormone therapy (tamoxifen) in premenopausal women, node(+), hormone receptor(+) breast cancer.

14.2.1 Reduktion

In der Reduktionsphase werden überflüssige und irreführende Füllwörter entfernt. Gleichzeitig werden unnötige Zeitbezüge entfernt. Allgemein sollen Daten, die keine Information tragen entfernt werden. Im Beispiel wird der unpräzise Begriff „appropriate use“ konkretisiert:

> Premenopausal women with breast cancer, that is node positive and has positive hormone receptors, must be treated with chemotherapy and hormone therapy (tamoxifen).

Allgemein basieren viele Aussagen auf Qualitätsindikatoren. Diese Qualitätsindikatoren sollen helfen, einschätzen zu können, welche Qualität die Behandlung im Krankenhaus hat. Das Krankenhaus veröffentlicht dazu statistische Daten, etwa, welcher Anteil der Patienten nach einer Operation Komplikationen erleidet, wie hoch eine Rückfallquote ist oder ähnliches. Diese statistischen Daten werden häufig auf bestimmte Zeiträume normiert, also beispielsweise der Anteil an Patienten mit Komplikationen im letzten Jahr. In diesem Fall ist die eigentliche Qualitätsaussage vom Zeitraum unabhängig. Auch wenn für praktische Belange nur ein Zeitraum betrachtet wird, so sollte aus theoretischen Erwägungen heraus die Zahl absolut gesehen so klein wie möglich sein. Lautet die Aussage also „Die Zahl der Patienten mit Komplikationen in den letzten zwölf Monaten soll möglichst gering sein“, so kann der zeitliche Bezug für die Korrektheitsverifikation weggelassen werden. Auch dies wird in der Reduktionsphase getan.

14.2.2 Normalisierung

Die Normalisierung von Aussagen soll erreichen, dass die bereits reduzierten Aussagen strukturiert werden. Diese Strukturierung erzwingt möglicherweise eine weitere Konkretisierung unpräziser Formulierungen. Die Strukturierung umfasst eine Unterteilung der Aussage in drei Komponenten. Eine Vorbedingung, die angibt, für welche Patientengruppe die Aussage erfüllt sein soll, ein Zeitraum, in dem Verhalten ausgewertet werden soll sowie eine Beschreibung des zu beobachtenden Verhaltens. Diese drei Komponenten werden wie folgt aufgeschrieben:

> G[Vorbedingung der Patientengruppe]
> S[Startzeitpunkt des Korridors]
> E[Endzeitpunkt des Korridors]
> B[Gesuchtes Verhalten]

Die Buchstaben vor den Klammern stehen dabei für *G*roup (Patientengruppe), *S*tart (Anfang des Beobachtungszeitraums), *E*nd (Ende des Beobachtungszeitraums) und *B*ehaviour (zu beobachtendes Verhalten).

Die erste Klammer umfasst also die Vorbedingung. In der Beispielaussage ist das die genaue Beschreibung der Krebsart sowie der Tatsache, dass die Patientinnen pre-menopausal sein müssen. Die zweite und dritte Klammer beschreiben den Beobachtungszeitraum. Nachdem die Aussage keinen Zeitraum zur Beobachtung des Verhaltens angibt, wird der größtmögliche sinnvolle Zeitraum, nämlich vom Start bis zum Ende der Behandlung gewählt. Die vierte Klammer beschreibt das zu beobachtende Verhalten. Dieses ist, dass eine Behandlung mit Chemotherapie und Hormontherapie erfolgen muss. Damit wird die Aussage normalisiert wie folgt aufgeschrieben:

> G[Premenopausal women with breast cancer, that is node positive and has positive hormone receptors]

S[from the start of care]
E[until the end of care]
B[must be treated with chemotherapy and hormone therapy (tamoxifen)]

Die Vorbedingung ist optional, das heißt, wenn keine Vorbedingung vorliegt, darf das Feld leer bleiben. Alle anderen Felder sind obligatorisch und müssen ausgefüllt werden.

14.2.3 Formalisierung

Die Formalisierung beschreibt die Übersetzung der normalisierten Aussage in die GDL-Syntax. Die GDL-Syntax umfasst dabei zwei gleichwertige Repräsentationsformen, die XML-Form sowie die sogenannte *Präsentationssyntax*. Diese Formen sind einfach ineinander übersetzbar und dienen den unterschiedlichen Ansprüchen der Lesbarkeit durch Menschen und durch Maschinen. Die Repäsentationssyntax hat dabei die bereits oben vorgestellte Form:

Goal *Name des Beweisziels*
  Precondition
    Vorbedingung der Aussage
  Time-Specification
    From
      Startzeitpunkt
    Zeitbegrenzung
  Beobachtungsmodus
    Beobachtungsformel

Die XML-Form ersetzt die Schlüsselwörter und optischen Einrückungen durch XML-Elemente. Sie wird hier nicht weiter betrachtet.

Die Formalisierung der Aussage führt zu folgendem Ergebnis:

Goal Property 4
  Precondition
    premenopausal women AND node positive AND
    hormone receptor positive AND breast cancer
  Time-Specification
    From start of care
    Until end of care
  Observe-during-period
    chemotherapy and hormone therapy (tamoxifen)

Bis zu diesem Schritt hat sich an der Formulierung der Aussage noch nicht viel geändert. Überflüssige Füllwörter und unnötige Zeitannotationen werden während Reduktion und Normalisierung entfernt, davon abgesehen bleibt der Wortlaut der ursprünglichen Aussage aber erhalten. Dies ist notwendig, damit Experten der medizinischen Domäne diesen Formalisierungsprozess begleiten können und damit sicherstellen, dass die letztendlich bewiesene Aussage inhaltlich gleich zu der Ursprungsaussage ist.

Im nächsten Schritt wird die Aussage an das Vokabular der Leitlinie angepasst.

14.2.4 Anpassung

Der Anpassungsschritt bei der Formalisierung von Aussagen meint die Anpassung des Vokabulars der Aussage an das der betrachteten Leitlinie. Zu diesem Zweck wird GDL zunächst modellspezifisch erweitert, also ein GDL-Asbru definiert, das beispielsweise auf die Planstruktur und Planaktivierungen abgestimmt ist. In der durch die Anpassung erzeugte GDL-Asbru Aussage muss dann festgelegt werden, welche Pläne mit ihren Zustandswechseln den Anforderungen der Aussage entsprechen, das heißt, welcher Plan entspricht der `chemotherapie` und welchem Planzustandswechsel entspricht die Beobachtung einer Behandlung.

GDL-Asbru kennt zu diesem Zweck das Schlüsselwort **Transition**, welches einen Planzustandsübergang in Asbru markiert. Die Syntax orientiert sich an der von abstrakten Asbru-Uhren, es wird also angegeben, um welchen Plan und Planzustand es geht und dazu, ob dieser betreten oder verlassen werden soll.

Die Beispielaussage verlangt die Anwendung von Chemotherapie. Um festzustellen, ob die Leitlinie, wie gefordert, Chemotherapie verordnet, muss untersucht werden, welche Asbru-Pläne einer Anwendung der Chemotherapie entsprechen. Im Beispiel wird festgestellt, dass der gesuchte Plan der Plan mit dem Namen `cmf-chemotherapy` ist. Auch muss festgelegt werden, dass die Chemotherapie dann als angewendet gilt, wenn der Plan in den Zustand `activated` wechselt. Analog dazu müssen die Vorbedingungen an die Variablen der Asbru-Leitlinie angepasst werden. Damit wird die Aussage wie folgt angepasst an das Modell formuliert:

Goal Property 4
  Precondition
    menopausal-state = premenopausal AND
    BC-nodes = positive AND
    hormone-receptor-levels = positive
  Time-Specification
    From start
    Until end
  Observe-during-period
    Transition `cmf-chemotherapy` enter `activated` AND
    `tamoxifen-therapy` enter `activated`

Die Vorbedingung für diese Aussage ist also, dass die Variable „menopausal-state“ auf dem Wert „premenopausal“ gesetzt sein muss, die Variable „BC-nodes“, die angibt, ob die Lymphknoten der Achsel mit Krebs befallen sind, muss auf den Wert „positive“ gesetzt sein und die Variable „hormone-receptor-levels“ muss auf den Wert „positive“ gesetzt sein. Start- und Endpunkte werden auf „start“ und „end“ gesetzt. Die Bedingung wird nun auf die Pläne der Hierarchie angepasst, das heißt, die Chemotherapie wird durch den Plan `cmf-chemotherapy` beschrieben, die Hormonbehandlung mit Tamoxifen durch den Plan `tamoxifen-therapy`.

Es deutet sich an, dass durch eine Modellierung der Aussage, die bereits so nah am Asbru-Modell ist, eine automatische Übersetzung möglich ist. Gleichzeitig kann ein Domänenexperte durch die relativ kleinen Einzelschritte, die zwischen den einzelnen Stufen der Übersetzung liegen, das Ergebnis verstehen. Dadurch ist es möglich, an dieser Stelle eine Aussage anzugeben, die sowohl vom Mediziner verstanden, als auch vom System interpretiert werden kann.

Für letzteres muss noch eine (automatische) Übersetzung angegeben werden. Derzeit existiert kein solcher automatischer GDL-Übersetzer. Jedoch werden Übersetzungsregeln definiert, so dass die Übersetzung sich streng an einem vorgegebenem Schema orientiert.

14.2.5 Übersetzung

Für die Übersetzung der GDL-Asbru Aussagen in eine Beweisverpflichtung muss zunächst die Syntax der Bedingungen festgelegt werden. Die Vorbedingung der Aussage lautet:

menopausal-state = premenopausal AND
BC-nodes = positive AND
hormone-receptor-levels = positive

Diese Vorbedingung muss an die Variablen des Asbru-Systems angeglichen werden. Es handelt sich dabei um Bedingungen an den Zustand des Patienten. Der Patientenzustand wird in der Krankenakte **PDH** gespeichert. Auf diese wird so zugegriffen, dass zunächst der Zeitpunkt angegeben wird, dessen Krankenakteneintrag selektiert werden soll und dann das Datenfeld der Akte, das von Belang ist. Eine globale Variable **AC** speichert die aktuelle Uhrzeit, die im Fall der Bestimmung der Vorbedingung relevant ist. Als Feldname, der selektiert werden soll, wird der „Variablenbezeichner“ aus der GDL-Asbru Bedingung verwendet. Das heißt, die Vorbedingung wird wie folgt übersetzt:

$\mathbf{PDH}[\mathbf{AC}][\text{„menopausal-state“}] = \text{premenopausal}$
$\wedge\ \mathbf{PDH}[\mathbf{AC}][\text{„BC-nodes“}] = \text{positive}$
$\wedge\ \mathbf{PDH}[\mathbf{AC}][\text{„hormone-receptor-levels“}] = \text{positive}$

Es wird also aus der Krankenakte **PDH** der durch die aktuelle Uhrzeit **AC** bezeichnete Eintrag selektiert. In diesem Eintrag wird ausgewertet, ob das Feld mit der Bezeichnung „menopausal-state“ den Wert premenopausal hat. Analog erfolgt die Auswertung der anderen Teile der Vorbedingung.

Das Startereignis der Aussage ist einfach die Konstante **start**. Diese bezeichnet den erstmöglichen Zeitpunkt bei Betrachtung der Planausführung, also den ersten TL-Schritt. Zu übersetzen ist noch das Verhalten, also die Zeilen

Transition cmf-chemotherapy enter activated AND
tamoxifen-therapy enter activated

des GDL-Asbru Beispiels. Damit wird die Transition beschreiben, in der die Pläne der Chemotherapie und der Hormontherapie in den Zustand `activated` wechseln. Diesen Zustandswechsel liest GDL-Asbru aus dem Asbru-Zustand **AS** aus. Da es sich um eine Transition handelt, wird geprüft ob der Plan von einem beliebigen nicht-`activated`-Planzustand in den `activated`-Planzustand wechselt. Dies wird wie folgt formalisiert:

$$\mathbf{AS}[\texttt{cmf-chemotherapy}] \neq \texttt{activated} \wedge \mathbf{AS}'[\texttt{cmf-chemotherapy}] = \texttt{activated}$$

Bei dieser Aussage ist zu beachten, dass sie als ITL+ Aussage formuliert ist. Als solche können darin ungestrichene, einfach- und zweifach gestrichene Variablen vorkommen. Ungestrichene Variablen beschreiben den Zustand, der zu Anfang eines temporallogischen Schrittes gilt, also den Ausgangszustand. Einfach gestrichene Variablen die Werte nach dem System- und vor dem Umgebungsschritt und die zweigestrichenen Variablen die Werte nach dem Umgebungsschritt. Die Transition zwischen ungestrichenen und einfach gestrichenen Variablen wird durch das System kontrolliert – also die Asbru-Pläne. Die Transition zwischen einfach und zweifach gestrichenen Variablen wird durch die Umgebung kontrolliert. In diesem Fall

ist die Formel wahr, wenn das Asbru-System den Zustand des Plans `cmf-chemotherapy` auf den Wert `activated` ändert.

Übersetzungsregeln der GDL-Asbru Aussagen in die ITL+-Temporallogik, die vom interaktiven Verifikationswerkzeug KIV verstanden wird, sollen möglichst generisch sein. Das heißt, unabhängig vom Beobachtungsmuster der Aussage, also der Frage, wann und wie oft das Verhalten gezeigt werden muss. GDL-Asbru unterstützt verschiedene Beobachtungsmuster, beispielsweise zum Nachweis, dass ein bestimmtes Verhalten immer, nie oder manchmal im Beobachtungszeitraum auftritt. Insofern liegt es nahe, eine generische Modellierung des Beobachtungszeitraums zu wählen. Diese Modellierung wird durch eine Umgebungsannahme umgesetzt.

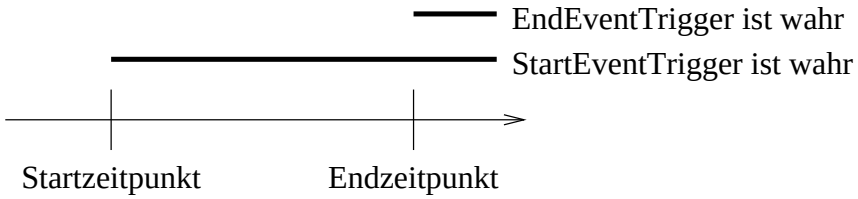


Abbildung 14.1: Start- und EndEventTrigger im zeitlichen Verlauf

Die Umgebung ist dafür verantwortlich, festzustellen, ob ein Zustand innerhalb oder außerhalb des Beobachtungszeitraums liegt. Dazu werden Variablen `StartEventTrigger` und `EndEventTrigger` verwendet. Die Variable `StartEventTrigger` ist immer dann wahr, wenn der Startzeitpunkt bereits überschritten wurde, die Variable `EndEventTrigger`, wenn der Endzeitpunkt überschritten wurde. Dies ist in Abbildung 14.1 dargestellt. Der Startzeitpunkt dieser Abbildung entspricht dem Startzeitpunkt der Aussage, der Endzeitpunkt entspricht dem Endzeitpunkt der Aussage. Die Umgebung überwacht das Eintreten dieser Ereignisse und setzt entsprechend die Trigger-Variablen auf den Wert **true** um. Für das Startereignis lässt sich dies wie folgt formulieren:

$$\Box \ \neg \texttt{StartEventTrigger} \wedge \mathit{Vorbedingung}'' \wedge \mathit{Startereignis} \supset \texttt{StartEventTrigger''};\texttt{StartEventTrigger''} \leftrightarrow \texttt{StartEventTrigger}$$

Es gilt immer, dass die Startereignisvariable auf den Wert **true** gesetzt wird, wenn das Startereignis eintritt, dieses vorher noch nicht eingetreten ist und die Vorbedingung der Aussage gilt. Zu beachten ist dabei, dass nach der Definition der GDL die Vorbedingung unmittelbar nach dem Setzen des Startereignisses gelten muss, also im ersten Systemschritt nach diesem. Daher wird in der Formel verlangt, dass die Vorbedingung im zweifach gestrichenen Zustand gilt. Sind die Bedingungen nicht erfüllt, so bleibt die Startereignisvariable unverändert. Analog dazu wird die Endereignisvariable gesetzt:

$$\Box \ \texttt{StartEventTrigger} \wedge \mathit{Endereignis} \supset \texttt{EndEventTrigger''};\texttt{EndEventTrigger''} \leftrightarrow \texttt{EndEventTrigger}$$

Nun kann in Abhängigkeit des Beobachtungsmusters eine Übersetzung angegeben werden. Beispielsweise verlangt das **avoid-during-period** Muster, dass die Bedingung während des Beobachtungszeitraums nie wahr sein darf. Dies wird mit einer temporallogische Formel formalisiert, die verlangt, dass immer, wenn der Beobachtungszeitraum begonnen hat aber noch nicht beendet wurde, die Bedingung nicht wahr sein darf:

$$\Box \ \texttt{StartEventTrigger} \ \rightarrow \texttt{EndEventTrigger} \vee \neg \ \mathit{Bedingung}$$

Ein weiteres Beobachtungsmuster ist `observe-during-period`. Dieses prüft, ob eine Bedingung während des Intervalls irgendwann einmal aufgetreten ist. Dies entspricht auch dem Muster der Beispielaussage. Um dieses Muster zu formalisieren, wird eine Hilfsvariable eingeführt. Diese heißt `Observed`. Die Variable muss initial mit dem Wert **false** belegt sein und dann, wenn innerhalb der Beobachtungszeitraums das gewünschte Ereignis eintritt auf den Wert **true** umgesetzt werden. Allgemein wird dies wie folgt angegeben:

$$\Box\ \texttt{StartEventTrigger} \wedge \neg\ \texttt{EndEventTrigger} \wedge \textit{Bedingung} \supset \texttt{Observed''};\texttt{Observed''} \leftrightarrow \texttt{Observed}$$

Die Formel wird so interpretiert, dass die Umgebung den Variablenwert von `Observed` auf **true** ändern soll, falls im Beobachtungszeitraum das gewünschte Verhalten auftritt. Passiert dies nicht, so soll der Wert von der Umgebung auf den gesetzt werden, der zu Anfang des Schrittes galt. Da das Asbru System den Wert der `Observed` Variablen unverändert lässt, bedeutet dies faktisch, dass die Umgebung den Wert ebenfalls unverändert lässt.

Mit dieser Hilfsvariablen kann nun verlangt werden, dass am Ende des Beobachtungsintervalls das gewünschte Ereignis beobachtet wurde. Gezeigt werden muss also folgende Formel:

$$\Box\ \neg\ \texttt{EndEventTrigger} \wedge \texttt{EndEventTrigger''} \rightarrow \texttt{Observed''}$$

Wichtig ist nun die Einordnung, wie diese Komponenten beim Aussagenbeweis zusammenspielen. Das Setzen der Startereignisvariablen und Endereignisvariablen sowie der Beobachtungsvariablen `Observed` gehören zu der sogenannten Umgebungsannahme. Es wird also angenommen, dass die Umgebung diese Variablen stets korrekt setzt. Daher werden diese Formeln in den Antezedenten der Sequenz aufgenommen. Dagegen ist zu beweisen, dass die Beobachtungsvariable am Ende des Beobachtungsintervalls wahr ist. Insofern muss diese Formel im Sukzedenten aufgeschrieben werden. Die Übersetzung der Beispielaussage lautet also wie folgt:

$$\Box\ \neg\ \texttt{StartEventTrigger} \wedge \textit{Vorbedingung"} \wedge \textit{Startereignis} \supset \texttt{StartEventTrigger''};\texttt{StartEventTrigger''} \leftrightarrow \texttt{StartEventTrigger}$$

$$\Box\ \texttt{StartEventTrigger} \wedge \textit{Endereignis} \supset \texttt{EndEventTrigger''};\texttt{EndEventTrigger''} \leftrightarrow \texttt{EndEventTrigger}$$

$$\Box\ \texttt{StartEventTrigger} \wedge \neg\ \texttt{EndEventTrigger} \wedge \textit{Bedingung} \supset \texttt{Observed''};\texttt{Observed''} \leftrightarrow \texttt{Observed}$$

$$\vdash \Box\ \neg\ \texttt{EndEventTrigger} \wedge \texttt{EndEventTrigger''} \rightarrow \texttt{Observed''}$$

Dabei werden Vorbedingung, Startereignis, Endereignis sowie Bedingung gemäß den oben angegebenen Regeln übersetzt. Details zum gesamten Übersetzungprozess sind in [67] aufgeschrieben.

15 Symbolische Ausführung von Asbru

Die grundlegende Beweistechnik zur Verifikation von Aussagen über Asbru-Systeme ist die symbolische Ausführung. In Abschnitt 12 wird beschrieben, wie die Semantik von Asbru mittels paralleler Programme in das interaktive Beweiswerkzeug KIV eingebettet wird. Der Abschnitt 13 weist nach, dass die durch Ausführung der parallelen Programme entstehenden Abläufe isomorph sind zu den Asbru-Abläufen, die durch die Anwendung der Asbru-Semantik Regeln vorgegeben werden.

Dieser Abschnitt beschreibt, wie die symbolische Ausführung paralleler Asbru-Prozeduren durchgeführt wird. Dazu wird zunächst in Abschnitt 15.1 beschrieben, wie Aussagen aufgebaut sind und gelesen werden. Darauf folgt in Abschnitt 15.2 eine Erläuterung, welche Ergebnisse nach der symbolischen Ausführung von Asbru-Prozeduren zu erwarten sind und wie die Beweistechnik der Induktion anzuwenden ist.

15.1 Aussagenformulierung

Temporallogische Aussagen bestehen aus mehreren inhaltlich unterschiedlichen Komponenten. In diesem Abschnitt werden die einzelnen Komponenten im Hinblick auf ihre Notwendigkeit bei der Aussagenformulierung und der Ausführung der Beweise erklärt. In diesem Abschnitt werden die in Abschnitt 14 beschriebenen GDL-Aussagen verwendet. Diese werden hier in Bezug zum auszuführenden Asbru-System gesetzt, für das sie gelten sollen.

Beweise temporallogischer Aussagen werden in der hier verwendeten ITL+-Logik über Intervalle geführt. Intervalle enthalten jeweils einen initialen Zustand σ_0, sowie eine potentiell unendliche Folge von Zustandsübergängen σ_i', σ_i'' wobei jeweils der zweifach gestrichene Zustand σ_i'' interpretiert wird, als der ungestrichene Zustand σ_{i+1}. Eine Aussage besteht typischerweise aus vier verschiedenen Komponenten. Die erste Komponente ist eine Beschreibung des aktuellen Zustands σ_0. Dabei handelt es sich um prädikatenlogische Formeln, die Werte der ungestrichenen Variablen beschreiben. Diese werden in Abschnitt 15.1.1 beschrieben. Eine zweite Komponente beschreibt den Systemübergang, also wie sich der Zustand σ_i' bei gegebenem σ_i berechnet. Dieser Zustandsübergang wird bei den hier betrachteten Aussagen ausschließlich durch Asbru-Systeme definiert. Eine Beschreibung von Systemkomponenten befindet sich in Abschnitt 15.1.2. Die dritte Komponente beschreibt die Zustandsübergänge zwischen einfach gestrichenen σ_i' und zweifach gestrichenen σ_i''. Diese Komponente wird Umgebungsannahme genannt und in Abschnitt 15.1.3 beschrieben. Die vierte und letzte Komponente ist die Beweisverpflichtung. Diese stellt Anforderungen an die Zustandsübergänge. Diese Anforderungen können an ungestrichene, einfach- oder doppeltgestrichene Zustände formuliert sein. Die Beweisverpflichtung wird in Abschnitt 15.1.4 erläutert.

15.1.1 Zustandsbeschreibung

Bei der Zustandsbeschreibung handelt es sich um prädikatenlogische Beschreibungen des aktuellen Zustands, also des ersten Zustands des Intervalls. Eine Sequenz $\Gamma \vdash \Delta$ kann nur dann von einem Intervall $\mathbf{I}$ modelliert werden ($\mathbf{I} \models \Gamma \vdash \Delta$), wenn der initiale Zustand $\mathbf{I}(0)$ die Zustandsbeschreibung erfüllt.

Typischerweise gibt es zwei verschiedene Aspekte, die in der Zustandsbeschreibung betrachtet werden. Zum Einen wird der aktuelle Zustand der für Asbru relevant ist beschrieben. Zum

Anderen werden in der Zustandsbeschreibung die Werte aller GDL-Hilfsvariablen (wie etwa dem `StartEventTrigger`) festgehalten.

Für Asbru sind die Belegung folgender Variablen relevant. Als erstes die Belegung der **Tick** Variablen, in der gespeichert wird, ob ein Makro-Schritt durchgeführt werden kann oder nicht. Als nächstes die Werte der Umgebungssignalsammlung **EAGG**, in der die Kommunikation des medizinischen Personals mit dem System gespeichert wird. Weiterhin Teil des Zustands des Asbru-Systeme sind die Krankenakte des Patienten **PDH** sowie die Asbru-Zustands-Historie **ASH**. In der Krankenakte sind die medizinisch relevanten Werte des Patienten gespeichert, in der Asbru-Zustands-Historie die vergangenen Planzustandsübergänge des Asbru-Systems. Im Asbru-Zustand **AS** werden die aktuellen Planzustände aller Asbru-Pläne gespeichert, in der Plankommunikationsvariablen **PC** die Aktivierungssignale. In der Rundenzähler-Variablen **RC** wird die Zahl der erfolgreichen Durchläufe zyklischer Pläne festgehalten.

Die Belegungen all dieser Variablen werden in der Zustandsbeschreibung festgehalten. Dabei ist es zulässig und teilweise sogar notwendig, einzelne Werte unspezifiziert zu lassen. Beispielsweise ist im Allgemeinen nicht vorgegeben, welche Signale das medizinische Personal an das System sendet. Entsprechend müssen diese **EAGG**-Werte unspezifiziert bleiben. Aus Effizienzgründen werden diverse Variablen in der Regel unspezifiziert gelassen. Beispielsweise muss der Asbru-Zustand in der Regel nur für alle aktuell laufenden Pläne und deren Unterpläne spezifiziert werden. Für Pläne die (noch) nicht ausgeführt werden, kann der Planzustand unspezifiziert bleiben. Das erhöht die Lesbarkeit der Sequenz und steigert die Ausführungseffizienz. Analoges gilt für die Asbru-Zustands-Historie. Diese muss nur für die Pläne mitgeschrieben werden, die in einem **enter** bzw. **leave** Konstrukt einer abstrakten Asbru-Uhr einer Asbru-Bedingung referenziert werden.

Zusätzlich zu den Variablen, die für die Ausführung von Asbru relevant sind, müssen auch noch die Variablenbelegungen berücksichtigt werden, die für die Spezifikation von GDL-Aussagen notwendig sind. Dabei handelt es sich speziell um den `StartEventTrigger`, den `EndEventTrigger` sowie gegebenenfalls die Variable `Observed`.

Eine initiale Zustandsbeschreibung könnte die folgende Form haben:

```
Tick,
PDH[AC][BC-nodes] = negative,
PC = init,
¬ StartEventTrigger,
¬ EndEventTrigger,
¬ Observed
```

In diesem Beispiel ist die **Tick** Komponente auf den Wert **true** gesetzt. Alle Pläne, die aktive Schritte durchführen, setzen diese Komponente auf den Wert **false**. Aufgrund der Spezifikation des `sync` Prädikats, dass die Berechnungsergebnisse mehrerer Pläne zu einem zusammenführt, reicht ein Plan aus, der die **Tick** Variable auf den Wert **false** setzt, damit der Wert der Variablen nach der Zusamenführung **false** ist. Ist jedoch nach der Systemtransition **Tick** mit dem Wert **false** belegt, dann bedeutet dies, dass ein Makro-Schritt verhindert wird.

Die Umgebungssignale in diesem Beispiel bleiben unspezifiziert, dies führt dazu, dass bei Abfrage dieser Signale eine Fallunterscheidung durchgeführt werden muss. Nur so kann aber sichergestellt werden, dass tatsächlich alle möglichen Arztentscheidungen berücksichtigt werden. Die Krankenakte wird partiell festgelegt. Das heißt, dass das „BC-nodes“ Feld des aktuellen Zeitpunkts **AC** der Krankenakte **PDH** mit dem Wert „negative“ belegt wird. Alle anderen Einträge der Krankenakte bleiben unspezifiziert. Ebenso unspezifiziert bleiben die Asbru-Zustands-Historie sowie der Asbru-Zustand und die Zeit. Der Asbru-Zustand wird während der Ausführung automatisch richtig festgelegt, das heißt, der initial inaktive Top-Level Plan wird in seinem ersten Schritt seinen Zustand automatisch auf den korrekten Wert

`considered` setzen. Bevor er in die aktive Phase wechselt, wird er die Zustände seiner Unterpläne auf den Wert `inactive` setzen. Auf die Weise wird stets der Teil des Asbru-Zustands konsistent mit den Planzuständen der Pläne gemacht, der für die Planausführung relevant ist. Gleiches gilt für die Plankommunikationsvariable. Diese kann aber initial auf die Konstante „init“ gesetzt werden. Diese Konstante liefert bei Abfragen der Aktivierungssignale stets das Ergebnis **false** zurück. Die **RC** Variable kann unspezifiziert gelassen werden, da Pläne, die in die `Execution`-Phase gelangen automatisch ihren Rundenzähler auf den korrekten Wert 0 setzten. Das heißt, der relevante Teil des Rundenzählers wird stets konsistent gehalten.

Nachdem der Zustand der Asbru-Hierarchie beschrieben wurde, muss als nächstes noch der für die GDL-Asbru Aussagen relevante Teil des Zustands beschrieben werden. Dieser besteht im Beispiel aus den drei booleschen Variablen `StartEventTrigger`, `EndEventTrigger` und `Observed`. Diese ersten beiden Variablen beschreiben das Intervall, in dem das gewünschte Verhalten beobachtet werden muss. Der `StartEventTrigger` wird wahr, wenn der Startzeitpunkt dieses Beobachtungsintervalls vergangen ist, der `EndEventTrigger` wird wahr, wenn der Endzeitpunkt dieses Intervalls vergangen ist. Auf diese Weise kann anhand der Variablen bestimmt werden, ob der gegenwärtige Zeitpunkt vor, nach oder innerhalb des gesuchten Intervalls ist. `Observed` wird dann auf den Wert **true** gesetzt, wenn das gewünschte Verhalten innerhalb des Beobachtungsintervalls auftritt. Details über die Bedeutung dieser Variablen finden sich in Abschnitt 14.

15.1.2 Systembeschreibung

Die Systembeschreibung einer temporallogischen Aussage beinhaltet die Beschreibung aller Systemtransitionen, also der Zustandsübergänge zwischen ungestrichenen und einfach gestrichenen Zuständen. Die Übergänge können entweder mit temporallogischen Formeln oder mit parallelen Programmen beschrieben werden. Im Fall der symbolischen Ausführung von Asbru werden die in Abschnitt 12 angegebenen Programme verwendet. Dazu wird im initialen Zustand eine Prozedur verwendet, die dem Top-Level Plan der zu betrachtenden Hierarchie entspricht. Im Beispiel soll eine Aussage über das erste Kapitel der Brustkrebsfallstudie geführt werden. Dieser Plan heißt `chapter1` und hat als Top-Level Plan keine weiteren übergeordneten Pläne. Entsprechend wird die Prozedur wie folgt verwendet:

```
[inactive#(superGen('chapter1'); Tick, EAGG, PDH, ASH, AS, AC, PC, RC)]
```

Diese Prozedur ruft nach Abarbeitung eines Schrittes die `considered#` Prozedur auf, welche dann ihrerseits zu gegebener Zeit die Prozeduren, die die Unterpläne repräsentieren, parallel zu sich selbst startet. Daher entspricht diese Systembeschreibung genau der kompletten relevanten Planhierarchie.

15.1.3 Umgebungsannahme

Die Umgebungsannahme beschreibt die Umgebungsübergänge, also die Relation zwischen einfach gestrichenen und zweifach gestrichenen Zuständen. Umgebungsannahmen werden mit Hilfe temporallogischer Formeln festgelegt. Dabei ist bei der Formulierung der temporallogischen Formeln ein Unterschied zu den parallelen Programmen zu beachten. Dazu soll als Beispiel das Programm $X := 3$ dienen. Dieses Programm kann durch folgende temporallogische Formel ausgedrückt werden:

$$X' = 3 \wedge \circ\mathbf{last} \wedge \lceil \mathrm{X} \rceil$$

Dabei hat das erste Glied der Konjunktion die Funktion X im einfach gestrichenen Zustand auf den Wert 3 zu setzen. Das nächste Glied verlangt, dass die temporallogische Formel nach

einem Schritt terminiert. Das dritte Glied der Konjunktion ist eine „unendliche“ Formel. Diese verlangt, dass der Wert aller Variablen, die nicht X sind unverändert bleibt. Dadurch gewinnt ein Programm eine gewisse Form der Exklusivität. Ein Programm verlangt, dass genau die definierten Änderungen durchgeführt werden aber nicht mehr.

Ein Analogon zu dieser unendlichen Formel, das entsprechende Aussagen für den Umgebungsübergang zwischen einfach und zweifach gestrichenen Zuständen macht, existiert nicht. Daher muss bei der Umgebungsannahme explizit gemacht werden, welche endlichen Mengen an Variablen sich auf eine genau definierte Weise ändern sollen. Für alle nicht angegebenen Variablen gilt, dass diese sich beliebig ändern können. Dieser Ansatz hat den Vorteil, dass nicht ein zentrales Programm alle Variablenübergänge definiert. Stattdessen kann und muss die Umgebungsannahme dezentral für die einzelnen Variablen formuliert werden.

Dies ist insofern sehr hilfreich, als die Umgebungsannahme in Asbru meist aus drei unterschiedlichen Aspekten zusammengesetzt wird. Der erste Aspekt beschreibt dabei die Forderungen des Asbru-Systems. Dieses verlangt beispielsweise, dass die internen Variablen des Systems, wie die **AS**-Variable, von der Umgebung nicht verändert werden. Gleichzeitig wird aber von der Umgebung verlangt, dass die Asbru-Zustands-Historie in jedem Mikro- wie Makro-Schritt korrekt fortgeschrieben wird. Diese beiden Forderungen können in KIV wie folgt als Umgebungsannahme spezifiziert werden:

```
□ AS'' = AS'
□ ASH'' = ASH'[AS']
```

Weiterhin muss die Umgebung in jedem Fall die **Tick**-Variable auf den Wert **true** setzen. Da die Programme, die Asbru implementieren diese Variable nur auf den Wert **false** setzen, wenn sie einen Mikro-Schritt wünschen, nicht aber auf **true**, wenn sie dies nicht tun, könnte ohne dieses Verhalten nach einmaligem Umsetzen von **Tick** kein Makro-Schritt mehr durchgeführt werden. In KIV wird diese Umgebungsannahme wie folgt formalisiert:

```
□ Tick''
```

Ein zweiter Aspekt, der von der Umgebungsannahme betrachtet wird, sind Zustandsänderungen, die während Makro-Schritten durch die physische Umgebung ausgelöst werden. Immer, wenn ein Makro-Schritt geschieht, kann sich beispielsweise die Krankenakte **PDH** ändern. Das bedeutet, dass in Mikro-Schritten verlangt wird, dass sich diese nicht ändert. Dies kann wie folgt formalisiert werden:

```
□ ¬ Tick' → PDH'' = PDH'
```

In Makro-Schritten kann durch die zu beweisende Aussage festgelegt sein, wie sich die Werte der Variablen verändern können. Verlangt die Aussage beispielsweise, dass der Plan `radiotherapy` als Unterplan eines `anyorder`-Plans ab dem Moment vom medizinischen Personal präferiert wird, ab dem der Plan zur Brusterhaltenden Therapie in die `Execution`-Phase gewechselt hat, kann dies wie folgt formalisiert werden:

```
□ (    AS'[nameGen('bct', nameGen('treatment', superGen('chapter1')))] = activated
   → □ EAGG''[nameGen('radiotherapy', nameGen('treatment', superGen('chapter1')))].prefer)
```

Der dritte Aspekt der Umgebungsannahme wird durch die GDL-Ziele vorgegeben. Diese regeln über temporallogische Formeln das Setzen der Variablen, die festhalten, ob das erforderliche Verhalten beobachtet wurde und ob das Startereignis oder das Endereignis stattgefunden haben. Diese Formeln werden im Abschnitt 14 beschrieben.

15.1.4 Beweisverpflichtung

Die vierte und letzte Komponente, die zu einer Aussage gehört ist die Beweisverpflichtung. Durch die Zustandsbeschreibung wird der ungestrichene Zustand definiert. Die Systembeschreibung legt fest, wie der Übergang von diesem in den einfach gestrichenen Zustand zu erfolgen hat. Den Übergang vom einfach gestrichenen zum zweifach gestrichenen Zustand definiert die Umgebungsannahme. Die Beweisverpflichtung kann alle drei Zustände in Beziehung zueinander setzen. Zu beweisen ist, dass diese temporallogische Formel von den anderen drei Komponenten erfüllt wird.

Die Aussagen selber werden mittels GDL formalisiert. Bei einer **observe-during-period** Aussage, wie sie in Abschnitt 14 vorgestellt wird, ist die Beweisverpflichtung der Aussage die nachstehende Formel:

```
□ ¬ EndEventTrigger ∧ EndEventTrigger'' → Observed''
```

15.1.5 Sequenz

Die eben vorgestellten Komponenten werden in diesem Abschnitt zu einer Sequenz zusammengesetzt. Diese Sequenz ist ein Beispiel für ein Beweisziel, das im interaktiven Verifikationssystem KIV bewiesen werden kann.

```
(: Zustandsbeschreibung :)
Tick,
PDH[AC]['BC-nodes'] = negative,
PC = init,
¬ StartEventTrigger,
¬ EndEventTrigger,
¬ Observed

(: Systembeschreibung :)
[inactive#(superGen('chapter1'); Tick, EAGG, PDH, ASH, AS, AC, PC, RC)]

(: Umgebungsannahme :)
 (: Nicht Beeinflussung interner Variablen :)
 □ AS'' = AS' ∧ PC'' = PC'
 □ Tick''

 (: Korrektes Schreiben der Signale und der Krankenakte :)
 □ (    AS'[nameGen('bct', nameGen('treatment', superGen('chapter1')))] = activated
    → □ EA''[nameGen('radiotherapy', nameGen('treatment', superGen('chapter1')))].prefer)
 □ ¬ Tick' → PDH'' = PDH'

 (: Setzen der GDL-Variablen :)
 □     ¬ StartEventTrigger ∧ PDH''[AC]['BC-nodes'] = negative
     ∧ AS[superGen('treatment')] = activated
   ⊃ StartEventTrigger''; StartEventTrigger'' ↔ StartEventTrigger
 □   StartEventTrigger ∧ AS[superGen('treatment')] = completed
   ⊃ EndEventTrigger''; EndEventTrigger'' ↔ EndEventTrigger
 □     StartEventTrigger ∧ ¬ EndEventTrigger
     ∧ AS[nameGen('radiotherapy', nameGen('treatment', superGen('chapter1')))] = activated
   ⊃ Observed''; Observed'' ↔ Observed

⊢
(: Beweisverpflichtung :)
□ ¬ EndEventTrigger ∧ EndEventTrigger'' → Observed''
```

Der erste Abschnitt in der Sequenz beschreibt den aktuellen Zustand, sowohl des Asbru-Systems als auch der GDL-Variablen. Danach folgt die Beschreibung des Asbru-Systems, gefolgt von der Umgebungsannahme. Diese gliedert sich in die drei Teile, nicht-Beeinflussung des Systems durch die Umgebung, korrektes Umsetzen der Umgebungssignale und der Krankenakte sowie das korrekte Setzen der GDL-Variablen. Die letzte Komponente der Sequenz ist die Beweisverpflichtung.

15.2 Symbolische Ausführung

Die in Abschnitt 15.1 vorgestellten Sequenzen sollen im folgenden symbolisch ausgeführt werden. Dazu wird die symbolische Ausführung paralleler Programme verwendet. In Abschnitt 15.2.1 wird die Ausführung der in Abschnitt 15.1 vorgestellten Sequenz durchgeführt. Darauf folgend beschreibt der Abschnitt 15.2.2 die Anwendung der temporallogischen Induktion.

15.2.1 Ein erster Schritt

Als Beispiel soll die Aussage aus Abschnitt 15.1 ausgeführt werden. Diese sieht wie folgt aus:

```
(: Zustandsbeschreibung :)
Tick,
PDH[AC]['BC-nodes'] = negative,
PC = init,
¬ StartEventTrigger,
¬ EndEventTrigger,
¬ Observed

(: Systembeschreibung :)
[inactive#(superGen('chapter1'); Tick, EAGG, PDH, ASH, AS, AC, PC, RC)]

(: Umgebungsannahme :)
 (: Nicht Beeinflussung interner Variablen :)
 □ AS'' = AS' ∧ PC'' = PC'
 □ Tick''

 (: Korrektes Schreiben der Signale und der Krankenakte :)
 □ (    AS'[nameGen('bct', nameGen('treatment', superGen('chapter1')))] = activated
    → □ EA''[nameGen('radiotherapy', nameGen('treatment', superGen('chapter1')))].prefer)
 □ ¬ Tick' → PDH'' = PDH'

 (: Setzen der GDL-Variablen :)
 □     ¬ StartEventTrigger ∧ PDH''[AC]['BC-nodes'] = negative
     ∧ AS[superGen('treatment')] = activated
   ⊃ StartEventTrigger''; StartEventTrigger'' ↔ StartEventTrigger
 □   StartEventTrigger ∧ AS[superGen('treatment')] = completed
   ⊃ EndEventTrigger''; EndEventTrigger'' ↔ EndEventTrigger
 □     StartEventTrigger ∧ ¬ EndEventTrigger
     ∧ AS[nameGen('radiotherapy', nameGen('treatment', superGen('chapter1')))] = activated
   ⊃ Observed''; Observed'' ↔ Observed

⊢
(: Beweisverpflichtung :)
□ ¬ EndEventTrigger ∧ EndEventTrigger'' → Observed''
```

Wird in dieser Situation ein symbolischer Schritt ausgeführt, so ist die Ergebnisaussage wie folgt:

```
(: Zustandsbeschreibung :)
Tick,
PDH[AC][BC-nodes] = negative,
PC = init,
¬ StartEventTrigger,
¬ EndEventTrigger,
¬ Observed,
AS[superGen('chapter1')] = considered

(: Systembeschreibung :)
[considered#(superGen('chapter1'); Tick, EAGG, PDH, ASH, AS, AC, PC, RC)]

(: Umgebungsannahme :)
 (: Nicht Beeinflussung interner Variablen :)
 □ AS'' = AS' ∧ PC'' = PC'

 (: Korrektes Schreiben der Signale und der Krankenakte :)
```

```
□ (    AS'[nameGen('bct', nameGen('treatment', superGen('chapter1')))] = activated
   → □ EA''[nameGen('radiotherapy', nameGen('treatment', superGen('chapter1')))].prefer)
□ ¬ Tick' → PDH'' = PDH'


(: Setzen der GDL-Variablen :)
□     ¬ StartEventTrigger ∧ PDH''[AC]['BC-nodes'] = negative
    ∧ AS[superGen('treatment')] = activated
  ⊃ StartEventTrigger''; StartEventTrigger'' ↔ StartEventTrigger
□   StartEventTrigger ∧ AS[superGen('treatment')] = completed
  ⊃ EndEventTrigger''; EndEventTrigger'' ↔ EndEventTrigger
□     StartEventTrigger ∧ ¬ EndEventTrigger
    ∧ AS[nameGen('radiotherapy', nameGen('treatment', superGen('chapter1')))] = activated
  ⊃ Observed''; Observed'' ↔ Observed

⊢
(: Beweisverpflichtung :)
□ ¬ EndEventTrigger ∧ EndEventTrigger'' → Observed''
```

Die Änderungen, die sich durch den Schritt ergeben, beschränken sich auf die Systembeschreibung und die Zustandsbeschreibung. Durch den Schritt wurde der Planzustand von `inactive` auf `considered` geändert. Entsprechend besteht die Systembeschreibung nicht länger aus der `inactive#`-Prozedur, sondern aus der `considered#` Prozedur.

15.2.2 Induktion

In Fällen, in denen ein Zyklus im Beweis auftritt, also eine Sequenz, die in dieser Form bereits vorher aufgetreten ist, kann die Beweistechnik der temporallogische Induktion verwendet werden, um das Beweisziel zu schließen. Beispielsweise ist das der Fall, wenn der Plan „chapter1“ im Zustand `considered` ist. Dann wertet der Plan seine **filter**-Bedingung aus. Diese verlangt, dass eine bestimmte Diagnose vorliegt. Falls dies nicht der Fall ist, wartet der Plan im Zustand `considered` ab, ob sich die Diagnose noch ändert. Dies kann beliebig lange dauern, weswegen temporallogische Induktion angewendet werden muss, um dieses Ziel zu schließen. Um temporallogische Induktion starten zu können, wird ein Induktionsterm benötigt. In KIV fest eingebaut ist die Möglichkeit, über Eventually-Aussagen im Antezedenten Induktion zu führen. Dabei ist der Induktionsterm die Zahl der Schritte bis zum Eintreten des Ereignisses der Eventually-Aussage. Aufgrund der Dualität zwischen Eventually und Always kann eine Induktion auch über Always-Formeln im Sukzedenten gestartet werden. In anderen Fällen muss ein Induktionsterm von Hand angegeben werden.

Durch Starten der Induktion verändert sich die Sequenz. Zu zeigen ist, dass eine Aussage immer gilt. Statt dies zu zeigen, kann alternativ angenommen werden, dass die negierte Aussage irgendwann wahr wird. Wenn gezeigt werden kann, dass die negierte Aussage nie wahr wird, ist damit die ursprüngliche Aussage bewiesen. Formal wird in KIV die Formel

```
□ ¬ EndEventTrigger ∧ EndEventTrigger'' → Observed''
```

umgeschrieben zu folgender Formel:

```
N' = N'' + 1 until ¬ (¬ EndEventTrigger ∧ EndEventTrigger'' → Observed'')
```

Das heißt, eine unspezifizierte, natürliche Zahl N wird so lange weiter erniedrigt, bis die Negation der ursprünglichen Aussage wahr ist. Da N eine natürliche Zahl ist, ist die Induktion damit wohlfundiert. Damit wird die Sequenz aus Abschnitt 15.2.1, in der Plan `chapter1` im Zustand `considered` ist, wie folgt umgeschrieben:

```
(: Zustandsbeschreibung :)
Tick,
PDH[AC]['BC-nodes'] = negative,
```

```
PC = init,
¬ StartEventTrigger,
¬ EndEventTrigger,
¬ Observed,
AS[superGen('chapter1')] = considered

(: Systembeschreibung :)
[considered#(superGen('chapter1'); Tick, EAGG, PDH, ASH, AS, AC, PC, RC)]

(: Umgebungsannahme :)
 (: Nicht Beeinflussung interner Variablen :)
 □ AS'' = AS' ∧ PC'' = PC'

 (: Korrektes Schreiben der Signale und der Krankenakte :)
 □ (    AS'[nameGen('bct', nameGen('treatment', superGen('chapter1')))] = activated
    → □ EA''[nameGen('radiotherapy', nameGen('treatment', superGen('chapter1')))].prefer)
 □ ¬ Tick' → PDH'' = PDH'

 (: Setzen der GDL-Variablen :)
 □     ¬ StartEventTrigger ∧ PDH''[AC]['BC-nodes'] = negative
     ∧ AS[superGen('treatment')] = activated
   ⊃ StartEventTrigger''; StartEventTrigger'' ↔ StartEventTrigger
 □   StartEventTrigger ∧ AS[superGen('treatment')] = completed
   ⊃ EndEventTrigger''; EndEventTrigger'' ↔ EndEventTrigger
 □     StartEventTrigger ∧ ¬ EndEventTrigger
     ∧ AS[nameGen('radiotherapy', nameGen('treatment', superGen('chapter1')))] = activated
   ⊃ Observed''; Observed'' ↔ Observed

 (: Induktionsformel :)
 N' = N'' + 1 until ¬ (¬ EndEventTrigger ∧ EndEventTrigger'' → Observed'')
 INDHYP-nat(N)

⊢
```

Der Term INDHYP-nat(N) enthält dabei die eigentliche Induktionshypothese. Die Aussage kann nun wie gewohnt ausgeführt werden. Nach der Ausführung des symbolischen Schrittes entstehen zwei Beweisziele. Ein Beweisziel beschreibt eine Situation, in der der Planzustand von „chapter1“ den Wert `possible` angenommen hat, der andere, in der Planzustand noch immer auf den Wert `considered` gesetzt ist. Im ersten Fall muss der Beweis weiter mit symbolischer Ausführung geführt werden, im zweiten Fall muss die Induktion angewendet werden.

Die Anwendung der Induktion schlägt jedoch fehl. Dadurch, dass der Plan „chapter1“ bei einem Warten auf die Erfüllung der **filter**-Bedingung keinen aktiven Schritt durchführt, kommt es zur Ausführung eines Makro-Schritts. Dadurch kann sich die Krankenakte beliebig verändern wodurch das Wissen, dass die Lymphknoten der Achsel beim Patienten nicht durch Metastasen befallen sind (PDH[AC][BC-nodes] = negative), verloren geht. Es gibt zwei Möglichkeiten, mit diesem Umstand umzugehen. Zum Einen kann die Sequenz um eine Annahme erweitert werden, dass der Status der Lymphknoten auch über Makro-Schritte hinweg nicht verändert wird. Dies kann formuliert werden als:

```
□  PDH[AC]['BC-nodes'] = negative
```

Ist diese Annahme medizinisch nicht zu rechtfertigen, so muss vor der symbolischen Ausführung die Sequenz generalisiert werden, das heißt, die Information, die durch den Schritt verloren geht, wird vor der Schrittausführung verworfen. Damit ist die Induktion anwendbar. Ist allerdings die Generalisierung zu stark, das heißt, ist die verworfene Information im weiteren Beweis wichtig, so führt dies zu einem nicht beweisbaren Ziel.

16 Beweisdekomposition

In diesem Abschnitt wird eine Erweiterung der Beweistechnik der symbolischen Ausführung von Asbru beschrieben. Diese Beweistechnik heißt Beweisdekomposition und ist geeignet, Beweise über große Planhierarchien zu führen, die mit der symbolischen Ausführung alleine nicht mehr behandelbar sind. Abschnitt 16.1 führt in die Probleme der symbolischen Ausführung und das Konzept der Lösung ein. Danach wird in Abschnitt 16.2 die Beweistechnik selbst beschrieben.

16.1 Motivation der Beweisdekomposition

Zunächst sollen die Probleme der symbolischen Ausführung im Zusammenhang mit großen Planhierarchien beschrieben werden. Im Anschluss daran wird die Idee der Beweisdekomposition anhand eines Beispiels skizziert.

16.1.1 Effizienz der symbolischen Ausführung

Die symbolische Ausführung, wie sie in Abschnitt 15 vorgestellt wird, kann theoretische mit beliebig großen Planhierarchien umgehen. In der Praxis gibt es jedoch zwei Faktoren, die die Größe der Planhierarchien einschränken, die mit symbolischer Ausführung effizient behandelbar sind. Der erste Faktor ist die Ausführungseffizienz. Diese nimmt mit der Größe paralleler Systeme ab. Der zweite Faktor ist der Indeterminismus der Ausführung von Plänen.

Die Ausführungseffizienz sinkt mit der Größe des parallelen Systems, beispielsweise dadurch, dass für jede einzelne Komponente des Systems temporäre Berechnungsergebnisse bestimmt und gespeichert werden müssen. Danach müssen diese Ergebnisse mit dem `sync` Operator zusammengefasst werden. Neben der gesteigerten Komplexität der eigentlichen Berechnung ergibt sich eine Verlangsamung dadurch, dass die temporären Zwischenergebnisse bei der Simplifikation der Sequenz berücksichtigt werden müssen und diese so indirekt verlangsamen.

Der Indeterminismus der Prozeduren ergibt sich wiederum aus zwei verschiedenen Aspekten. Werden beispielsweise zwei Unterpläne gleichzeitig gestartet, wobei bei beiden Unterplänen jeweils aufgrund einer unspezifizierten Krankenakte die Erfüllbarkeit der **filter**-Bedingungen nicht entschieden werden kann, müssen möglicherweise neun verschiedene Fälle berücksichtigt werden. Der erste Unterplan kann sich in drei verschiedenen Zuständen befinden je nachdem, ob die **filter**-Bedingung nicht erfüllbar, erfüllbar oder erfüllt ist. Für den zweiten Plan ergeben sich gleichen drei Fälle und die parallele Ausführung zwingt dann zur Betrachtung aller drei mal drei Fälle. Die Zahl der zu betrachtenden Fälle wächst also exponentiell mit der Zahl der Pläne. Eine weitere Quelle des Indeterminismus sind die `anyorder`-Kontrolltypen. Diese aktivieren jeweils einen Unterplan basierend auf einem extern vorgegebenem Signal. Im Bezug auf die Verifikation muss dieses Signal als indeterministisch angesehen werden.

16.1.2 Umgehung des Indeterminismus

In vielen Fällen ist es nicht notwendig, den Indeterminismus, der durch die Ausführung entsteht im Detail zu betrachten. Als Beispiel wird die Planhierarchie aus Abbildung 16.1 ver-

wendet. Der dort angegebene Plan `Plan` hat die Unterpläne `U-Plan 1` bis `U-Plan 3`, die mittels `anyorder` Kontrolle ausgeführt werden sollen. Gezeigt werden soll, dass `Plan` immer beendet wird, sofern nicht sein übergeordneter Plan terminiert. `Plan` soll dabei auf alle seine Unterpläne warten.

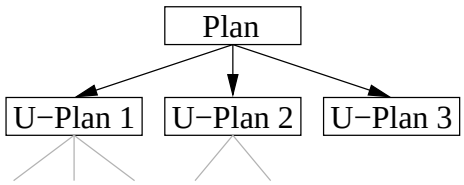


Abbildung 16.1: Beispielhierarchie

Angenommen, die Unterpläne haben alle **filter**- und **setup**- Bedingungen, die immer erfüllbar sind, jedoch ist die Krankenakte unterspezifiziert, wodurch nicht entscheidbar ist, ob die Bedingungen erfüllt sind. Sind alle Pläne im Zustand `considered`, so müssen nach Ausführung eines Schritts acht verschiedene Fälle unterschieden werden, da jeder der Pläne unabhängig von den anderen in den Zustand `possible` fortschreiten kann oder im Zustand `considered` bleiben kann.

Der hier entstehende Indeterminismus soll nun besser handhabbar werden. Angenommen die drei Unterpläne haben keine **abort**-Bedingung und keine Wartebedingung. Das bedeutet, dass diese Pläne, wenn sie einmal gestartet wurden, unendlich lange in `Selection`- und `Execution`-Phase verbleiben oder irgendwann in den Zustand `completed` wechseln, sofern der Plan `Plan` nicht terminiert.

Ziel der Beweisdekomposition ist es das für eine Aussage relevante Verhalten eines Plans zu identifizieren. In diesem Fall ist das für die Beendigung von `Plan` relevante Verhalten der Unterpläne, dass diese ihre Bearbeitung nicht abbrechen und auch nicht zurückgewiesen werden. Dann wird nachgewiesen, dass die Pläne dieses identifizierte Verhalten auch wirklich zeigen. Dies wird in Abbildung 16.2 veranschaulicht. Es sollen also temporallogische Formeln φ_1 bis φ_3 gefunden werden, die das identifizierte Verhalten beschreiben und für die gilt, dass sie eine Abstraktion des Verhaltens der Unterpläne sind.

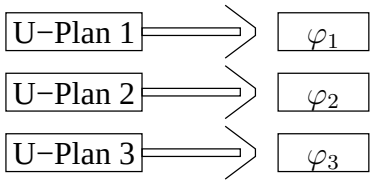


Abbildung 16.2: Abstraktion der Unterpläne

In einem zweiten Schritt kann eine temporallogische Formel φ gesucht werden, die das gemeinsame Verhalten von φ_1 bis φ_3 ausdrückt. Dann wird entweder durch symbolische Ausführung oder mit Hilfe eines Kompositionalitätstheorems nachgewiesen, dass φ eine korrekte Abstraktion für die Zusammenschaltung von φ_1, φ_2 und φ_3 ist, wie in Abbildung 16.3 gezeigt.

Durch diese Technik wird von dem exakten Verhalten der Unterpläne so abstrahiert, dass der Indeterminismus in getrennte Beweise ausgelagert werden kann. Weiterhin wird die Zahl der parallel laufenden Komponenten reduziert und so erreicht, dass die Ausführungseffizienz steigt. Mit dieser Technik ist somit es möglich, Beweise über nahezu beliebig große Hierarchien zu führen.

Abbildung 16.3: Zusammenfassung der Abstraktionen

16.2 Beweisdekomposition

In diesem Abschnitt werden die formalen Aspekte der Beweisdekomposition beschrieben. Dazu wird zunächst in Abschnitt 16.2.1 der Rely-Guarantee-Ansatz vorgestellt. Die Formulierung von Rely-Guarantee-Formeln wird in Abschnitt 16.2.2 beschrieben. In Abschnitt 16.2.3 wird erläutert, wie diese Formeln bewiesen und zusammengeführt werden können. Abschnitt 16.2.4 schließt dann die Beschreibung der Beweisdekomposition mit einer Betrachtung ab, wie diese Formeln angewendet werden könne.

16.2.1 Der Rely-Guarantee-Ansatz

Der Rely-Guarantee-Ansatz beschreibt eine Möglichkeit, wie reaktive Systeme, die in einer Umgebung agieren, abstrahiert werden können. Der Ansatz wird beispielsweise in [4, 47, 38, 17] vorgestellt. Um die Abstraktion durchführen zu können, werden für dieses eine Annahme (Rely) und eine Garantie (Guarantee) formuliert. Der Grundgedanke ist, dass das System das in der Garantie beschriebene Verhalten aufrecht erhält, wenn die Umgebung im Gegenzug die Annahme nicht verletzt.

Damit ist es möglich, das garantierte Verhalten des Systems vom Verhalten der Umgebung abhängig zu machen. Ein Beispiel dafür ist, wenn der Plan garantieren soll, dass seine Ausführung nicht abgebrochen wird. Dazu muss allerdings im Gegenzug die Umgebung garantieren, dass der übergeordnete Plan nicht terminiert. Terminiert der übergeordnete Plan, so darf der Unterplan abbrechen oder zurückgewiesen werden. Angenommen der übergeordnete Plan heißt P_1, der Unterplan $P_{1.1}$. Dann ist eine naive Möglichkeit dieses Verhalten zu formalisieren wie folgt:

$$\Box \neg \textbf{isTerminated}(\text{AS}[P_1]) \rightarrow \Box \neg \textbf{isAborted}(\text{AS}[P_{1.1}])$$

Wird die Aussage so formalisiert ist sie nur begrenzt nützlich. Hintergrund hierfür ist die Vorbedingung $\Box \neg$ **isTerminated**(AS[P_1]). Dies ist in der Regel nicht beweisbar. In der Regel terminieren Asbru-Pläne nach einer endlichen Zahl von Schritten. Es ist nicht davon auszugehen, dass eine medizinische Behandlung, allein aus biologischen Gründen, unendlich lang andauern kann. Die Formulierung der Aussage lässt zu, dass das vom Plan garantierte Verhalten bereits von Anfang an nicht eingehalten werden muss, wenn irgendwann die Umgebung die Annahme verletzt.

Ziel der Formulierung von Rely-Guarantee-Aussagen muss es sein, dass das abstrahierte System die Garantie erst dann verletzen darf, wenn die Umgebung ihrerseits die Annahme verletzt hat und das System dies feststellen konnte. Andernfalls werden die abstrahierten Systeme „hellseherisch“. Dies wird in Abbildung 16.4 dargestellt.

Die Abbildung zeigt drei Abläufe zur Illustration von Rely-Guarantee. Die Abläufe bestehen aus Transitionen, wobei Transitionen von durchgezogenen zu einem gestrichelten Zuständen Systemschritte sind, die anderen Transitionen Umgebungsschritte. Der erste Ablauf beschreibt den Fall, in dem die Annahme nie verletzt wird. Dann soll die Garantie auch nie verletzt werden. Der zweite Ablauf zeigt das Phänomen „hellseherischer“ Systeme. Da im zweiten Schritt die Umgebung die Annahme verletzt, kann der Plan sich bereits im ersten Schritt beliebig verhalten. Das interessante an diesem Fall ist, dass die Verletzung der

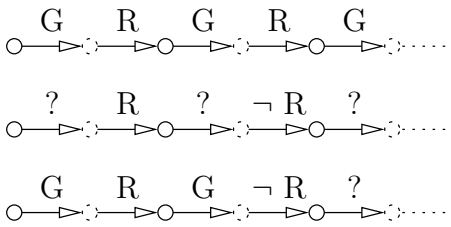


Abbildung 16.4: Garantieverletzung bei Rely-Guarantee

Annahme hier möglicherweise überhaupt erst entsteht, weil die Garantie nicht eingehalten wurde. Dies soll am Beispiel erläutert werden.

Angenommen die Annahme an die Umgebung ist, dass der übergeordnete Plan eines Plans niemals abbricht. Unter dieser Annahme soll der untergeordnete Plan ebenfalls nicht abgebrochen werden. Für das Beispiel soll weiter angenommen werden, dass der übergeordnete Plan in seiner Wartebedingung zwingend auf den zu abstrahierenden Unterplan wartet. Kann dieser nicht beendet werden, bricht der übergeordnete Plan ab. In diesem Szenario kann das Paradoxon auftreten, dass der untergeordnete Plan seine Garantie verletzt, weil die Umgebung die Annahme für den übergeordneten Plan später einmal verletzen wird. Die Verletzung der Annahme geht aber in diesem Fall direkt darauf zurück, dass die Garantie zuerst verletzt wurde.

Aus diesem Grund ist eine Formulierung der Rely-Guarantee-Formeln so wünschenswert, wie dies im dritten Ablauf in Abbildung 16.4 gezeigt wird. In diesem Fall wird das System gezwungen, die Garantie so lange einzuhalten, bis die Umgebung die Annahme verletzt hat und das System dies auch feststellen konnte. Wird die Annahme also im zweiten Schritt verletzt, so kann das System dies erst zu Beginn des dritten Schritts feststellen. Erst dann darf das System sich beliebig verhalten.

16.2.2 Rely-Guarantee-Aussagen

Die Formulierung von Rely-Guarantee-Aussagen wird hier mit dem **unless** Operator durchgeführt. Eine Formel φ **unless** ψ ist wahr, genau dann, wenn φ so lange gilt, bis mindestens einmal ψ wahr geworden ist. Dabei muss ψ niemals eintreten, das heißt, es kann auch immer φ gelten.

Sei G die Garantie eines Systems und R die Annahme an die Umgebung, dann wird die Rely-Guarantee-Aussage wie folgt formalisiert. G **unless** G $\wedge$ $\neg$ R. In Worten, die Garantie G gilt solange, bis noch ein letztes mal G gilt und die Annahme R verletzt wird. Die Tatsache, dass die Garantie G in dem Schritt noch einmal gelten muss, in dem die Annahme R verletzt wurde, beruht auf dem System/Umgebungskonzept von ITL+. Jeder Schritt besteht aus zwei Teilschritten, dem Systemschritt und dem Umgebungsschritt. Die Abfolge ist, dass zunächst ein Systemschritt stattfindet, nachdem dann ein Umgebungsschritt durchgeführt wird. Die Annahme an die Umgebung wird logischerweise so formuliert, dass sie Aussagen über den Umgebungsschritt macht, während die Garantie so formuliert wird, dass sie Aussagen über den Systemschritt macht. Das bedeutet, dass der Plan die Verletzung der Annahme erst einen Schritt später feststellen kann. Diese Argumentation kann in Abbildung 16.4 nachvollzogen werden.

Mit diesen Grundlagen können nun Garantien und Annahmen formuliert werden. Dabei handelt es sich um einen kreativen Schritt, der nicht allgemein beschrieben werden kann. Es hat sich jedoch im Laufe der Anwendung der Verfahren auf die Fallstudie ein Vorgehensmuster als besonders nützlich herausgestellt. Nach diesem Vorgehen wird die Garantie formuliert als eine Konjunktion von Implikationen. Vorbedingungen der Implikationen sind ungestrichene

Variablenzustände, Nachbedingungen einfach-gestrichene Variablenzustände. Dies soll nun anhand von Beispielen gezeigt werden.

Den Anfang macht dabei das oben aufgeführte Beispiel, bei dem ein Plan nicht abbrechen soll, solange der übergeordnete Plan nicht abbricht. In das erwähnte Formulierungsmuster übersetzt ist die Garantie des Plans also, dass der Planzustand in der einfach gestrichenen Asbru-Zustandsvariablen nicht abgebrochen ist, wenn er in der nicht-gestrichenen Asbru-Zustandsvariablen nicht abgebrochen ist. Die Garantie ist also, dass der Plan nicht auf den `aborted` Zustand wechselt. Formal:

$$\mathrm{AS}[\mathbf{iname}] \neq \mathrm{aborted} \rightarrow \mathrm{AS'}[\mathbf{iname}] \neq \mathrm{aborted}$$

Eine andere Garantie ist, dass ein Plan niemals in die `Execution`-Phase wechselt, wenn er nicht sein Aktivierungssignal erhalten hat. Auch dies lässt sich mit der oben beschriebenen Methodik formalisieren:

$$\neg\ \mathbf{isExecuted}(\mathrm{AS}[\mathrm{iname}]) \wedge \neg\ \mathrm{PC}[\mathrm{iname}] \rightarrow \neg\ \mathbf{isExecuted}(\mathrm{AS'}[\mathrm{iname}])$$

Ein letztes Beispiel, dass hier noch vorgestellt wird beschreibt die Aktivierung eines Plans über mehrere Ebenen. Angenommen, es ist wichtig, dass ein Plan so lange nicht aktiviert wird, wie der übergeordnete Plan selbst noch nicht aktiviert worden ist. Dies ist deswegen eine vernünftige Forderung, da nur der übergeordnete Plan das Aktivierungssignal senden kann. Ist dieser noch nicht in der `Execution`-Phase, kann der untergeordnete Plan nicht gestartet und nicht aktiviert werden. Als Garantie kann dies wie folgt formuliert werden:

$$\begin{array}{l}\neg\ \mathbf{isExecuted}(\mathrm{AS}[\mathrm{iname}]) \wedge \neg\ \mathrm{PC}[\mathrm{iname}] \rightarrow \neg\ \mathbf{isExecuted}(\mathrm{AS'}[\mathrm{iname}]) \\ \neg\ \mathbf{isExecuted}(\mathrm{AS}[\mathrm{iname}]) \wedge \neg\ \mathrm{PC}[\mathbf{nameGen}(\mathbf{SK}_1, \mathrm{iname})] \rightarrow \neg\ \mathrm{PC'}[\mathbf{nameGen}(\mathbf{SK}_1, \mathrm{iname})]\end{array}$$

Die erste Zeile dieser Garantie stellt sicher, dass der übergeordnete Plan nur in die `Execution`-Phase kommen kann, wenn er sein Aktivierungssignal erhalten hat. Die zweite Zeile dann stellt sicher, dass der Plan kein Aktivierungssignal an seinen Unterplan verschickt, wenn er selbst nicht in der `Execution`-Phase ist.

Die Annahmen werden über einfach und zweifach gestrichene Variablen formuliert. Typischerweise besteht eine Umgebungsannahme aus Formeln, die sicherstellen, dass die Umgebung nicht die Planzustandsfelder der betrachteten Pläne ändert. In dem zuletzt beschrieben Aktivierungsbeispiel wäre also ein Teil der Annahme, das die `plan` und **`nameGen(`$\mathbf{SK}_1$`, iname)`** Felder der **AS**-Variablen unverändert bleiben:

$$\mathrm{AS''}[\mathrm{iname}] = \mathrm{AS'}[\mathrm{iname}] \wedge \mathrm{AS''}[\mathbf{nameGen}(\mathbf{SK}_1, \mathrm{iname})] = \mathrm{AS'}[\mathbf{nameGen}(\mathbf{SK}_1, \mathrm{iname})]$$

Eine weitere häufige Komponente der Annahme ist, dass durch die Umgebung nicht die Plankommunikationsvariable in den relevanten Feldern verändert wird. Im Beispiel wird dies wie folgt formalisiert:

$$\mathrm{PC''}[\mathbf{nameGen}(\mathbf{SK}_1, \mathrm{iname})] = \mathrm{PC'}[\mathbf{nameGen}(\mathbf{SK}_1, \mathrm{iname})]$$

Damit kann eine Gesamtaussage formuliert werden, die folgende Form hat:

$$\begin{array}{l}(\ \ (\neg\ \mathbf{isExecuted}(\mathrm{AS}[\mathrm{iname}]) \wedge \neg\ \mathrm{PC}[\mathrm{iname}] \rightarrow \neg\ \mathbf{isExecuted}(\mathrm{AS'}[\mathrm{iname}])) \\ \quad \wedge (\neg\ \mathbf{isExecuted}(\mathrm{AS}[\mathrm{iname}]) \wedge \neg\ \mathrm{PC}[\mathbf{nameGen}(\mathbf{SK}_1, \mathrm{iname})] \rightarrow \neg\ \mathrm{PC'}[\mathbf{nameGen}(\mathbf{SK}_1, \mathrm{iname})])) \\ \mathbf{unless} \\ \quad (\qquad (\neg\ \mathbf{isExecuted}(\mathrm{AS}[\mathrm{iname}]) \wedge \neg\ \mathrm{PC}[\mathrm{iname}] \rightarrow \neg\ \mathbf{isExecuted}(\mathrm{AS'}[\mathrm{iname}])) \\ \quad\ \wedge (\neg\ \mathbf{isExecuted}(\mathrm{AS}[\mathrm{iname}]) \wedge \neg\ \mathrm{PC}[\mathbf{nameGen}(\mathbf{SK}_1, \mathrm{iname})] \rightarrow \neg\ \mathrm{PC'}[\mathbf{nameGen}(\mathbf{SK}_1, \mathrm{iname})]) \\ \quad\ \wedge \neg\ (\ \ \mathrm{AS''}[\mathrm{iname}] = \mathrm{AS'}[\mathrm{iname}] \\ \qquad\ \wedge \mathrm{AS''}[\mathbf{nameGen}(\mathbf{SK}_1, \mathrm{iname})] = \mathrm{AS'}[\mathbf{nameGen}(\mathbf{SK}_1, \mathrm{iname})] \\ \qquad\ \wedge \mathrm{PC''}[\mathbf{nameGen}(\mathbf{SK}_1, \mathrm{iname})] = \mathrm{PC'}[\mathbf{nameGen}(\mathbf{SK}_1, \mathrm{iname})]))\end{array}$$

Mit dieser Aussage wird ein Teil des Verhaltens der betrachteten Pläne `iname` und **`nameGen(SK`**$_1$**`, iname)`** beschrieben, von großen Teilen jedoch abstrahiert. Beispielsweise gehen Informationen über zeitliche Abfolgen weitgehend verloren. Würde der Plan `iname` die `Selection`-Phase überspringen und direkt vom Zustand `inactive` in den Zustand `completed` wechseln, würde dies nicht dieser Aussage widersprechen.

Bevor diese Aussage jedoch tatsächlich verwendet werden kann, muss die Garantie noch erweitert werden. Die abstrahierten Pläne haben mit dieser Formel die Möglichkeit, den Asbru-Zustand, die Patientendaten oder die Plankommunikationsvariable beinahe vollkommen beliebig zu ändern. Damit können die abstrahierten Pläne auch die Planzustandsfelder anderer Pläne beschreiben und so bei der Ausführung Schreibkonflikte auslösen. Daher wird jeder Garantie noch eine Komponente hinzugefügt, die für alle Abstraktionen gleichermaßen erstellt wird:

```
  ⌈AS, PC, Tick⌉
∧ frame_as(AS, AS', (iname + nameGen(SK1, iname)))
∧ frame_pc(PC, PC', subplans)
```

Diese Formel garantiert, dass lediglich die Asbru-Zustands-Variable, und die Plankommunikationsvariable der betrachteten Plänen sowie die Tick-Komponente verändert werden. Der zweite Teil der Formel beschreibt genauer, welche Aspekte der Asbru-Zustandsvariablen verändert werden dürfen, nämlich nur die des obersten betrachteten Plans und aller seiner Unterpläne. Die Plankommunikationsvariable darf nur verändert werden an den Stellen aller Unterpläne des obersten betrachteten Plans.

Damit lässt sich die vorhin vorgestellte Aussage vervollständigen:

```
  (  (¬ isExecuted(AS[iname]) ∧ ¬ PC[iname] → ¬ isExecuted(AS'[iname]))
   ∧ (¬ isExecuted(AS[iname]) ∧ ¬ PC[nameGen(SK1, iname)] → ¬ PC'[nameGen(SK1, iname)])
   ∧ ⌈AS, PC, Tick⌉
   ∧ frame_as(AS, AS', iname + nameGen(SK1, iname))
   ∧ frame_pc(PC, PC', nameGen(SK1, iname)))
unless (  (¬ isExecuted(AS[iname]) ∧ ¬ PC[iname] → ¬ isExecuted(AS'[iname]))
       ∧ (  ¬ isExecuted(AS[iname]) ∧ ¬ PC[nameGen(SK1, iname)]
          →¬ PC'[nameGen(SK1, iname)])
       ∧ ⌈AS, PC, Tick⌉
       ∧ frame_as(AS, AS', iname + nameGen(SK1, iname))
       ∧ frame_pc(PC, PC', nameGen(SK1, iname))
       ∧ ¬ (  AS''[iname] = AS'[iname]
            ∧ AS''[nameGen(SK1, iname)] = AS'[nameGen(SK1, iname)]
            ∧ PC''[nameGen(SK1, iname)] = PC'[nameGen(SK1, iname)]))
```

16.2.3 Beweis und Kombination von Rely-Guarantee-Aussagen

Der Beweis von Rely-Guarantee-Aussagen erfolgt genau wie der Beweis von medizinischen Aussagen, das heißt, es wird dazu entweder die symbolische Ausführung oder rekursiv die Beweisdekomposition angewendet. Besonderheiten können sich durch eingeschränkte Annahmen an die Umgebung ergeben. Beispielsweise sind Aussagen über Pläne möglich, bei denen die Umgebungsannahme keine Forderung an die Umgebung stellt, die Planzustandskomponenten der betrachteten Pläne unverändert zu lassen. Da ein Plan in der `Execution`-Phase seine Ausführung vom Zustand seiner Unterpläne abhängig macht, wird das Verhalten des Plans nahezu beliebig, wenn die Zustände der Unterpläne von der Umgebung geändert werden können.

Hierzu ist das System/Umgebungsmodell von ITL+ zu beachten. Die Unterpläne setzen ihre eigenen Planzustände zwar korrekt in der einfach gestrichenen **AS**-Variablen, jedoch lesen die Pläne die Planzustände aus der von der Umgebung modifizierten, zweifach-gestrichenen **AS**-Variablen (die die nicht-gestrichene **AS** Variable des nächsten Zustands ist). Wenn die

Umgebung diese beliebig verändern darf, ist undefiniert, welches Ergebnis gelesen wird und entsprechend ist auch das Verhalten der Pläne beliebig. Solange dieses Verhalten ausreicht um die eigentliche Aussage zu beweisen, führt eine Verkleinerung der Annahme lediglich zu einer effizienteren Ausführung der Rely-Guarantee-Aussagen und damit zu einer effizienteren Beweisführung.

Um die Zahl der im System existierenden $\|_s$ Operatoren zu verringern, können Rely-Guarantee-Aussagen miteinander kombiniert werden. Werden in einem Beweis zwei Aussagen φ_1 und φ_2 verwendet, so ist das Ziel, eine Aussage φ zu finden, so dass einerseits $\varphi_1 \|_s \varphi_2 \rightarrow \varphi$ gilt, also φ eine korrekte Abstraktion der beiden einzelnen Komponenten ist, und gleichzeitig die Gesamtaussage beweisbar bleibt.

Sind die beiden Aussagen von der in Abschnitt 16.2.1 beschriebenen Form, lässt sich die Kombination der Aussagen automatisieren. Dazu wird vereinfachend angenommen, dass die Aussagen φ_1 und φ_2 jeweils eine Implikation in der Garantie aufführen. Sei also φ_1 wie folgt:

```
  (  (ψ1.1 → ψ1.2)
   ∧ ⌈AS, PC, Tick⌉
   ∧ frame_as(AS, AS', planlist1.1)
   ∧ frame_pc(PC, PC', planlist1.2))
unless (  (ψ1.1 → ψ1.2)
       ∧ ⌈AS, PC, Tick⌉
       ∧ frame_as(AS, AS', planlist1.1)
       ∧ frame_pc(PC, PC', planlist1.2))
       ∧ ¬ (R1)
```

φ_2 sei wie folgt definiert:

```
  (  (ψ2.1 → ψ2.2)
   ∧ ⌈AS, PC, Tick⌉
   ∧ frame_as(AS, AS', planlist2.1)
   ∧ frame_pc(PC, PC', planlist2.2))
unless (  (ψ2.1 → ψ2.2)
       ∧ ⌈AS, PC, Tick⌉
       ∧ frame_as(AS, AS', planlist2.1)
       ∧ frame_pc(PC, PC', planlist2.2))
       ∧ ¬ (R2)
```

Dann kann daraus in der Regel eine Gesamtaussage φ generiert werden, indem folgendes Schema benutzt wird:

```
  (  (ψ1.1 → ψ1.2)
   ∧ (ψ2.1 → ψ2.2)
   ∧ ⌈AS, PC, Tick⌉
   ∧ frame_as(AS, AS', union(planlist1.1, planlist2.1))
   ∧ frame_pc(PC, PC', union(planlist1.2, planlist2.2)))
unless (  (ψ1.1 → ψ1.2)
       ∧ (ψ2.1 → ψ2.2)
       ∧ ⌈AS, PC, Tick⌉
       ∧ frame_as(AS, AS', union(planlist1.1, planlist2.1))
       ∧ frame_pc(PC, PC', union(planlist1.2, planlist2.2)))
       ∧ ¬ (R2 ∧ R2)
```

Das Schema sieht vor, dass alle Implikationen der beiden Garantien mittels Konjunktionen verknüpft werden und so den ersten Teil der globalen Garantie bilden. Der zweite Teil der Garantie beschreibt, welche Variablen bzw. Variablenfelder nicht durch die abstrahierten Systeme verändert werden. Hierbei ist es notwendig, die Mengen der zu verändernden Pläne zu vereinigen. Das bedeutet, das abstrahierte Gesamtsystem darf alle Planzustands-Felder und Plankommunikationsvariablen-Felder verändern, die eines der Teilsysteme verändern durfte. Die Annahmen der Teilsysteme werden mittels Konjunktion miteinander verbunden. Das heißt, das Gesamtsystem darf dann von seinen Garantien abweichen, wenn eine der Annahmen der Teilsysteme verletzt wurde. Dieser Aspekt der Zusammenführung wirkt abschwächend,

das heißt, φ ist aufgrund dieser Zusammenlegungen der Garantien echt schwächer als die Teilsysteme φ_1 und φ_2. Jedoch wird diese Abschwächung nur dann relevant, wenn eine der Annahmen der Teilsysteme verletzt wird. Unter der Annahme, dass keine der Annahmen verletzt wird, ist die zusammengeführte Formel φ in der Regel äquivalent zu $\varphi_1 \parallel_s \varphi_2$.

Nachdem die Gesamtformel φ aufgestellt wurde, muss noch bewiesen werden, dass diese eine Abstraktion von $\varphi_1 \parallel_s \varphi_2$ ist. Dazu kann entweder symbolische Ausführung verwendet werden oder ein Modularisierungstheorem. Diese Beweistechnik wird in [11] vorgestellt.

16.2.4 Anwendung der Beweisdekomposition

Der Vorgang zur Verwendung der Beweisdekomposition ist immer derselbe. Bei einem Beweis, der mittels symbolischer Ausführung durchgeführt wird, wird offensichtlich, dass das System zu groß ist, um den Beweis effizient mit der symbolischen Ausführung abschließen zu können. In diesem Fall fällt die Wahl auf die Anwendung der Beweisdekomposition.

Die Planhierarchie wird daraufhin untersucht, ob und wenn ja, welche Pläne die Korrektheit der Aussage beeinflussen können. Häufig stellt sich bei dieser Untersuchung heraus, dass die überwiegende Zahl der Pläne der Hierarchie keinen Einfluss auf die Korrektheit der zu zeigenden Aussage hat. Solche Pläne werden als `passive` Pläne bezeichnet. Für diese wird bei der Suche nach Rely-Guarantee-Aussagen eine Standardabstraktion gewählt, die lediglich verlangt, dass der passive Plan und seine Unterpläne lediglich die zu ihnen gehörigen Felder in der **AS**-Variablen und der **PC**-Variablen sowie die **Tick**-Komponente des Zustands beschreiben. Diese Garantie gilt immer, da die Pläne technisch keine Möglichkeit haben, andere Felder zu ändern. In diesem Fall ist die Annahme an das System daher leer, die Aussage gilt unabhängig vom Verhalten der Umgebung.

Als nächstes wird in einem kreativen Schritt das relevante Verhalten der Pläne bestimmt, die nicht passiv sind. Ein typischer Fall für solches Verhalten ist, dass ein Plan nicht in die `Execution`-Phase wechseln kann, solange er oder seine übergeordneten Pläne nicht aktiviert wurden. Ein anderes typisches Beispiel ist, dass ein Plan unter bestimmten Umständen weder seine Bearbeitung abbricht noch zurückgewiesen wird und nach seinem Start entweder unendlich lange läuft oder schlussendlich seine Ausführung beendet.

Wenn für alle Pläne die entsprechenden Abstraktionsaussagen formuliert wurden, werden diese zu einer einzigen Aussage zusammengefasst. Diese Aussage wird dann in den ursprünglichen Beweis so eingebaut, dass statt der Vielzahl der Asbru-Pläne nur noch eine Abstraktionsaussage ausgeführt werden muss. Nach Einbau der Abstraktionsaussage wird als nächstes der Zustand soweit generalisiert wie nötig, so dass nach Ausführung eines temporallogischen Schritts Induktion anwendbar ist.

Am Beispiel wird diese Technik bei der Beschreibung der Fallstudienbeweise ab Kapitel 21 demonstriert.

17 Nachweis von Gegenbeispielen

Bei Durchführung eines Korrektheitsbeweises mit Asbru kann sich die Situation ergeben, dass nach einem temporallogischen Schritt eine unbeweisbare Sequenz entsteht. In diesem Fall ist es wünschenswert, feststellen zu können, ob die Sequenz nur aufgrund fehlerhaft angewendeter abschwächender Regeln entstanden ist oder die ursprüngliche Aussage nicht beweisbar ist. Die Probleme, die dabei entstehen, werden in Abschnitt 17.1 beschrieben. Die dafür einzusetzenden Lösungen werden in Abschnitt 17.2 beschrieben. Dieser Abschnitt fasst die Ergebnisse aus [34] zusammen.

17.1 Problembeschreibung

Bei der symbolischen Ausführung von Asbru werden abschwächende Regeln verwendet. Zum Einen geschieht dies automatisch dadurch, dass Heuristiken Daten über alte Zustände verwerfen, zum Anderen werden bei der Beweisdekomposition abschwächende Generalisierungen verwendet. In der Regel werden die Heuristiken nur Daten verwerfen, die keine Information mehr tragen, jedoch kann es in Ausnahmefällen auch vorkommen, dass Informationen verworfen werden. Dadurch kann eine Sequenz unbeweisbar werden. Genauso kann dies bei der Dekomposition von Beweisen passieren, wenn die Abstraktion wichtige Aspekte des Ablaufs eines Plans nicht berücksichtigt.

Aus diesem Grund kann aus der Unbeweisbarkeit einer Sequenz innerhalb eines Korrektheitsbeweises in der Regel nicht gefolgert werden, dass die ursprüngliche Aussage fehlerhaft war. Demzufolge muss nachgewiesen werden, dass die unbeweisbare Aussage tatsächlich fehlerhaft ist. Große Planhierarchien erlauben es allerdings meist nicht, ohne die abschwächenden Regeln den Beweis zu wiederholen und zu prüfen, ob die fragliche Sequenz dann noch immer nicht beweisbar ist.

17.2 Lösungsansätze

Das Problem unbeweisbarer Sequenzen wird über den Nachweis von Gegenbeispielen angegangen. Das heißt, falls die ursprüngliche Aussage unbeweisbar sein sollte, wird gezeigt, dass die negierte Aussage beweisbar ist. Als Beispiel soll die Beweisverpflichtung einer **observe-during-period** Aussage verwendet werden. Diese hat die Form

$$\Box \neg \texttt{EndEventTrigger} \wedge \texttt{EndEventTrigger''} \rightarrow \texttt{Observed''}$$

Es muss also nachgewiesen werden, dass die `Observed` Variable immer dann den Wert **true** angenommen hat, wenn das Endereignis auftritt. Wird diese Aussage nach den allgemeinen Regeln zur Negation von Aussagen behandelt, ergibt sich als Negat davon

$$\Diamond \neg \texttt{EndEventTrigger} \wedge \texttt{EndEventTrigger''} \wedge \texttt{Observed''}$$

Diese Aussage zu beweisen kann zu zwei Problemen führen. Das erste Problem hat seine Ursache darin, dass gezeigt werden muss, dass das Endereignis auch wirklich eintritt. Details und die Lösung zu dieses Problems wird in Abschnitt 17.2.1 beschrieben. Das zweite Problem liegt darin, dass die ursprüngliche Aussage möglicherweise nicht unter allen Umständen falsch ist. Details und Lösung zu diesem Problem werden in Abschnitt 17.2.2 beschrieben.

17.2.1 Erzwingen der Terminierung

Asbru-Aussagenbeweise verwenden häufig die Induktion, um mit dem Problem potentiell unendlicher Zyklen umzugehen. Solche Zyklen können immer dann auftreten, wenn eine Asbru-Bedingung so formuliert ist, dass sie prinzipiell unendlich lange erfüllbar, dabei aber nie erfüllt ist. In diesem Fall kann nicht garantiert werden, dass der Zyklus jemals beendet wird. Der Beweis der Lebendigkeitsaussage

```
◇ ¬ EndEventTrigger ∧ EndEventTrigger'' ∧ Observed''
```

ist damit unmöglich, da durch die Zyklen eben gerade nicht bewiesen werden kann, dass ein Endereignis überhaupt eintritt.

Entsprechend müssen bei Beweis solcher Aussagen unendliche Zyklen verboten werden. Typischerweise müssen dafür drei Bedingungen überprüft werden. Dabei handelt es sich um die **filter**- und die **setup**-Bedingung, speziell wenn sie mit einer **now_ata**-Zeit-Annotation verknüpft sind. Weiterhin muss die **complete**-Bedingung von `user`-kontrollierten Plänen geprüft werden.

Im Fall der **filter**- und der **setup**-Bedingungen muss durch entsprechende Forderungen an die Krankenakte sichergestellt werden, dass die Bedingungen als erfüllt oder nicht erfüllbar ausgewertet werden können. Beispielsweise prüft der oberste Plan des ersten Kapitels der Fallstudie in seiner **filter**-Bedingung, ob eine bestimmte Diagnose vorliegt. Diese Prüfung wird unendlich lange wiederholt, bis sie positiv abgeschlossen werden kann. Entsprechend muss zum Beweis des Gegenbeispiels eine zusätzliche Umgebungsannahme formuliert werden, die diese Diagnose immer dann wahr macht, wenn die Bedingung ausgewertet wird. Dies kann wie folgt formalisiert werden:

```
□ AS[superGen('chapter1')] = considered → PDH[AC]['diagnosis'] = DCIS
```

Damit wird erreicht, dass die Auswertung der **filter**-Bedingung in einem Schritt abgeschlossen werden kann. Solche, dem Gegenbeispiel hinzugefügten, Umgebungsannahmen müssen allerdings nach Beweis des Gegenbeispiels den begutachtenden Ärzten vorgelegt werden, damit diese beurteilen können, ob die geforderten Einschränkungen medizinisch zulässig sind.

Ein weiterer Aspekt, der zu unendlichen Zyklen führen kann, ist die Auswertung der **complete**-Bedingung bei `user`-kontrollierten Plänen. Typischerweise sind diese Bedingungen so spezifiziert, dass sie vom medizinischen Personal verzögert werden können. Das heißt, ein solcher Plan wird so lange laufen, bis das **.confirm** Signal der **complete**-Bedingung gesendet wird. Für die Berechnung von Gegenbeispielen wird das Auftreten dieses Signals durch eine Umgebungsannahme erzwungen. Soll beispielsweise erreicht werden, dass stets das **.override** Signal des Plans 'mammography' gesetzt wird, sobald dieser ausgeführt wird, kann dies mit folgender Umgebungsannahme geschehen:

```
□   AS[nameGen('mammography', nameGen('diagnosis', superGen('chapter1')))] = activated
  → ◇ EAGG[nameGen('mammography',
                   nameGen('diagnosis', superGen('chapter1')))] .confirm .complete
```

In diesem Fall wird garantiert, dass nach Aktivierung des Plans irgendwann das entsprechende Signal gesendet wird. Eine Induktion ist damit möglich über die Zahl der Schritte, die bis zum Senden des Signals vergehen. Diese Annahme muss prinzipiell nicht gerechtfertigt werden. Asbru soll eine reale Patientenbehandlung darstellen und als solches ist es unmöglich, dass eine medizinische Intervention unendlich lang andauern soll.

Alternativ dazu kann eine stärkere Annahme getroffen werden. Diese verzichtet auf die Annahme, dass das Signal irgendwann gesendet wird sondern verlangt stattdessen, dass das Signal sofort gesetzt wird. Formalisiert wird dies wie folgt:

```
□    AS[nameGen('mammography', nameGen('diagnosis', superGen('chapter1')))] = activated
  → EAGG[nameGen('mammography',
                 nameGen('diagnosis', superGen('chapter1')))] .confirm .complete
```

In diesem Fall muss in Zusammenarbeit mit Domänenexperten geklärt werden, ob die vereinfachende Annahme zulässig ist.

17.2.2 Einschränkungen der Aussage

Auch wenn das Modell so modifiziert wurde, dass keine unendlichen Abläufe mehr möglich sind, kann es dennoch sein, dass die negierte Aussage nicht beweisbar ist. Dies ist immer dann der Fall, wenn die ursprüngliche Aussage in einigen Fällen gilt, jedoch nicht in allen. In diesem Fall muss das Modell soweit eingeschränkt werden, dass nur noch Fälle betrachtet werden, in denen die Aussage nicht gilt. Es gibt zwei Kategorien von Einschränkungen. Die eine schränkt die Gruppe der betrachteten Patienten ein, die andere die Menge der Abläufe des Systems. Dies wird im Folgenden vorgestellt.

Einschränken der Patientengruppen

Einschränkungen der betrachteten Patientengruppe werden immer dann nötig, wenn Fallunterscheidungen, etwa bei Bedingungen oder dem `ifThenElse`-Kontrolltyp die Korrektheit der Aussage beeinflussen. Beispielsweise entscheidet eine `ifThenElse`-Kontrolle welcher Unterplan gestartet wird. Diese Entscheidung basiert auf der Zahl der mit Krebs befallenen Lymphknoten der Achsel. Ist kein Lymphknoten befallen, so wird der erste Unterplan ausgeführt. In diesem Fall wird die Aussage erfüllt. Sind dagegen Lymphknoten von Krebs befallen, wird der zweite Unterplan ausgeführt. In diesem Fall wird die Aussage verletzt und die Negation der Aussage wird wahr. Für die gesamte Patientengruppe, also alle Patienten sowohl mit als auch ohne befallene Lymphknoten ist weder die Aussage noch deren Negat wahr. Es kann allerdings gezeigt werden, dass bei einer eingeschränkten Patientengruppe, nämlich der mit befallenen Lymphknoten, die Aussage verletzt wird. Dieser eingeschränkte Fall kann bewiesen werden, in dem das Modell um die nachstehende Umgebungsannahme erweitert wird:

```
□ PDH[AC]['BC-nodes'] = positive
```

Wenn mit dieser Zusatzannahme die ursprüngliche Aussage falsifiziert wird, kann das Ergebnis mit Domänenexperten der Medizin diskutiert werden und eingegrenzt werden, woher diese Anomalie kommt.

Einschränken der Abläufe

Teilweise reicht ein Einschränken der Patientengruppe nicht aus, um eine Aussage falsifizieren zu können. Viele Fallstudien enthalten `anyorder`-kontrollierter Pläne. Diese wählen zu startende Unterpläne auf Basis eines vom Arzt zu senden Signals aus. Für Belange der Verifikation muss dieses Signal jedoch als zufällig angesehen werden. Eine Aussage kann nun gerade dadurch verletzt werden, dass die Signale in einer bestimmten Reihenfolge eintreffen.

Für die Falsifizierung von Aussagen muss also gegebenenfalls auch eine Umgebungsannahme formuliert werden, die eine Reihenfolge der Signale vorgibt. Beispielsweise kann mit nachstehender Formel erreicht werden, das stets der Plan der Brusterhaltenden Therapie den Vorzug vor dem der Mastektomie erhält:

```
□   EAGG[nameGen('bct', nameGen('treatment', superGen('chapter1')))].prefer
  ∧ ¬ EAGG[nameGen('bct', nameGen('treatment', superGen('chapter1')))].prefer
```

Mit den hier vorgestellten Techniken ist es möglich, mittels geeigneter Umgebungsannahmen einen genauen Ablauf der Planhierarchie vorzugeben. Werden die Terminierungsbedingungen entsprechend gewählt, der Indeterminismus der `anyorder`-Kontrollen begrenzt und die Patientengruppen entsprechend eingeschränkt, ist es möglich, Gegenbeispiele vollautomatisch zu beweisen. Dieses Ergebnis aus [34] erlaubt es, auf einfache Weise Gegenbeispiele zu berechnen und den Domänen-Experten zur Begutachtung vorzulegen.

18 Automatische Modell-Prüfung

Ein zur interaktiven Verifikation verschiedenes Herangehen zur Verifikation medizinischer Behandlungsrichtlinien ist die Anwendung automatischer Modell-Prüfungs-Techniken. Diese werden in [24] vorgestellt. Um diese Technik anwenden zu können, wird ein abstrahiertes Modell der Leitlinie aus dem Asbru-Modell erstellt. Aussagen können in diesem Modell automatisiert verifiziert werden. Die Abstraktion der Asbru-Modells wird in Abschnitt 18.1 beschrieben, die Verifikation der Aussagen in Abschnitt 18.2.

18.1 Generierung des abstrahierten Modells

Modellprüfer versuchen den Zustandsraum eines Modells vollautomatisch zu generieren und die Korrektheit einer Aussage in diesem Zustandsraum nachzuweisen. Dadurch, dass während der Generierung des Zustandsraums keine Interaktion möglich ist, muss der Modellprüfer mittels Heuristiken automatisch mit Situationen umgehen, in denen die Zustandsräume sehr groß oder sogar unendlich sind. Es gibt inzwischen Modellprüfer, die mit eingeschränkten Klassen unendlicher Zustandsräumen umgehen können, zum Beispiel [21, 33]. Diese Modellprüfer unterliegen jedoch in der Regel an anderer Stelle so starker Einschränkungen, dass ein Einsatz für die Verifikation von Asbru nicht möglich ist.

Das bedeutet, dass automatische Modellprüfung in Asbru nur möglich ist, wenn das Asbru-Modell abstrahiert wird. Eine entsprechende Technik zur automatischen Generierung einer Abstraktion von Asbru Modellen wird in [24] vorgestellt. Bei dieser Übersetzung werden alle Referenzen auf komplexe Datentypen entfernt, wie sie sich beispielsweise in Asbru-Bedingungen finden. Asbru-Bedingungen werden durch indeterministische Prädikate abstrahiert. Der zentrale Asbru-Zustand wird durch dezentrale Planzustands-Variablen implementiert. Damit werden die Quellen für unendliche Datentypen wie die Krankenakte und die Asbru-Zustands-Historie aus dem Modell eliminiert.

Durch diese Abstraktion wird eine große Zahl zusätzlicher Abläufe zugelassen. Daher muss bei der Verifikation von Aussagen stets untersucht werden, in wie weit diese Abstraktionen das Verifikationsergebnis beeinflussen. Es kann durch diese Überapproximation dazu kommen, dass falsch-negative Ergebnisse auftreten. Da es sich bei der Abstraktion allerdings um eine echte Überapproximation handelt, können falsch positive Ergebnisse ausgeschlossen werden.

18.2 Verifikation von Aussagen

Die automatische Modellprüfung verifiziert Aussagen über einem anderen Modell als die interaktive Verifikation. Speziell der Wechsel des Modells von der linearen ITL-Logik in die verzweigende CTL-Logik bedeutet, dass bei der Modellprüfung Aussagen auf andere Art und Weise formalisiert werden können. Diese Unterschiede werden in Abschnitt 18.2.1 beschrieben. Ergebnisse, die durch die Anwendung von Modell-Prüfung berechnet werden können, werden in Abschnitt 18.2.2 beschrieben. Abschnitt 18.2.3 schließt die Betrachtung der Modellprüfung mit einer Interpretation der dadurch erreichbaren Ergebnisse ab.

18.2.1 Aussagen in CTL

Die Modellprüfung von Asbru-Fallstudien wird mit dem Modellprüfer SMV [43] durchgeführt. Dabei kommen Aussagen zum Einsatz, die in der Temporallogik CTL [18, 25] formuliert sind. CTL ist eine Verzweigungs-Temporallogik, wohingegen ITL eine lineare Logik ist. Die Unterschiede zwischen beiden Logiken werden in Abbildung 18.1 visualisiert.

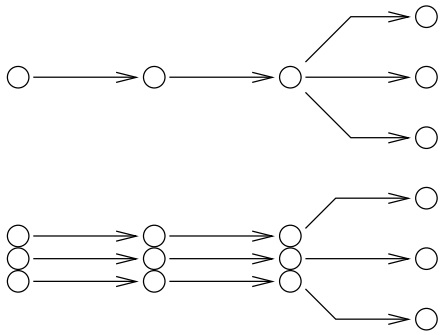


Abbildung 18.1: Vergleich zwischen verzweigenden und linearen Logiken

Der obere Teil der Grafik beschreibt die Sichtweise einer Verzweigungslogik. Bei dieser wird eine Formel über einem Zustand und allen möglichen Folgezuständen ausgewertet. Lineare Logiken wie ITL werden durch den unteren Teil der Abbildung dargestellt. Diese werden über einzelne, lineare Intervalle ausgewertet. Deutlich wird der Unterschied bei der Betrachtung von Fallunterscheidungen, die zu mehreren möglichen Folgezuständen führen. CTL führt dabei eine Fallunterscheidung innerhalb eines Zustands durch und betrachtet alle einzelnen Fälle jeweils als mögliche Folgezustände dieses einen Zustands. In ITL wird eine Fallunterscheidung dagegen dahingehend interpretiert, dass von vorneherein mehrere verschiedene Intervalle untersucht wurden, die jeweils die einzelnen Fälle betrachten. In jedem Intervall sind dabei bereits initial die Ergebnisse der Fallunterscheidungen festgelegt.

Diese unterschiedliche Sichtweise führt dazu, dass es in CTL möglich ist, Aussagen zu formulieren, die verlangen, dass eine Formel nur in mindestens einem der möglichen Folgeabläufe wahr sein muss. Eine solche Aussage ist in ITL nicht möglich. Dort muss a priori die Menge der Intervalle festgelegt werden, über welchen die Aussage gezeigt werden soll. Neben diesen Erreichbarkeits-Aussagen erlaubt CTL auch die Formulierung von Aussagen, die verlangen, dass eine Formel in allen möglichen Folgeabläufen gilt. Solche Formeln entsprechen weitgehend denen, die in ITL formulierbar sind.

Umgekehrt ist es in ITL möglich, Aussagen über das aktuelle Intervall zu formulieren. Eine Formel, die aussagt, dass ein Prozess in einem Ablauf fair behandelt wird ist in ITL direkt zu formulieren, während dies in CTL so nicht möglich ist.

18.2.2 Ergebnisse der Verifikation

Der Modellprüfer SMV geht bei der Evaluierung der Ergebnisse anders vor, als das interaktive Beweiswerkzeug KIV. Der Modellprüfer erzeugt zunächst den Zustandraum des Modells. Dazu wird nicht die symbolische Ausführung verwendet. Stattdessen werden automatisch Techniken eingesetzt, die den Zustandsraum so weit wie möglich abstrahieren und verkleinern. Danach wird getestet, ob die zu beweisende Aussage in diesem Zustandsraum wahr ist. Im Folgenden soll die Rückmeldung, die ein Anwender vom Modellprüfung SMV bzw. KIV erhält beschrieben und eingeordnet werden.

In KIV wird der Zustandsraum durch symbolische Ausführung erzeugt. Der Zustandsraum wird in Form eines Beweisbaums mitgeschrieben. Alle Abstraktionen des Zustandsraums wer-

den händisch eingeführt. Dadurch ist es bei einer Verletzung der Aussage relativ leicht, den Überblick zu behalten und festzustellen, welche Bedingungen die Aussagenverletzung ausgelöst haben. Ist die Aussage nicht verletzt, so ist bekannt, wie der Zustandsraum aussieht. Dadurch kann bereits während der Verifikation eine Art Plausibilitätsprüfung durchgeführt werden, ob sich System und Aussage wie erwartet verhalten.

Bei der Betrachtung der automatischen Modellprüfung von CTL Aussagen ist zu unterscheiden, welche Art von Aussage bewiesen werden soll. Zum Einen gibt es die Aussagen, die für jeweils einen Folgeablauf verlangen, dass die Bedingung wahr ist, zum Anderen gibt es Aussagen, die dies für alle Folgeablauf verlangen.

Kann eine Aussage bewiesen werden, die nur in jeweils einem Folgeablauf wahr sein muss, so gibt der Modellprüfer als Ergebnis einen Ablauf an, in dem diese Aussage erfüllt ist. Dieser Ablauf entspricht – bei gleichem zugrunde liegendem Modell – weitgehend dem, der durch symbolische Ausführung entsteht. Die Interpretations-Möglichkeiten sind somit ähnlich wie die bei der interaktiven Verifikation. Der Verlauf kann nachvollzogen und auf Plausibilität geprüft werden. Ist eine solche Aussage allerdings verletzt, dann gibt der Modellprüfer außer dieser Tatsache keine Rückmeldung. Offensichtlich kann kein Pfad gefunden werden, der die Aussage wahr macht. Da alle Ausführungspfade die Aussage verletzen gibt es kein sinnvolles „Gegenbeispiel“ dass zur Illustration angegeben werden könnte. Insofern bleibt bei einer Verletzung einer solchen Aussage nicht die Möglichkeit, einzugrenzen warum die Aussage nicht beweisbar ist.

Aussagen, die in jeweils allen Folgeablauf wahr sein müssen, verhalten sich genau invers zu den oben beschriebenen. Das heißt, im Falle der Korrektheit der Aussage wird lediglich dieses Faktum ausgegeben, jedoch keine weitere Erläuterung dazu und im Besonderen kein möglicher, erfüllender Ablauf. Im Falle der Aussagenverletzung wird ein Ablauf angegeben, der die Aussage verletzt. Die Interpretations-Möglichkeiten sind damit gerade bei der Korrektheit einer solchen Aussage stark eingeschränkt.

In der Brustkrebs Fallstudie war es beispielsweise so, dass ein Fehler in einem Behandlungsplan verhindert hat, das dessen Unterpläne jemals ausgeführt werden konnten. Eine Aussage, die eine bestimmte Abfolge zweier Behandlungen verboten hat, war in diesem Modell trivialerweise korrekt, da keine der Behandlungen jemals ausgeführt wurde. Ein solcher Fehler ist mit den eingeschränkten Rückgabewerten des Modellprüfers nur schwer feststellbar.

18.2.3 Interpretation der Ergebnisse

Die Modellprüfung erlaubt als „push-button“-Technologie einem breiteren Personenkreis den Zugang zu Verifikationstechniken als beispielsweise die interaktive Verifikation. Speziell dadurch, dass nicht nur die Aussagen-Prüfung, sondern auch die Erzeugung des zugrundeliegenden Modells aus dem Asbru-Modell automatisch durchgeführt wird, ist die Technik sehr einfach anwendbar. Damit eignet sie sich besonders zur schnellen und unproblematischen Untersuchung von Leitlinien.

Zu beachten sind dabei aber die Einschränkungen, die mit dieser Technik einhergehen. Zum Einen arbeitet die automatische Modellprüfung mit einem abstrahierten Modell der Asbru-Leitlinie. Das bedeutet zwangsläufig, dass es bei der Verifikation dazu kommen kann, dass Aussagen fälschlicherweise als verletzt angesehen werden, obwohl sie dies in Wirklichkeit nicht sind. Daher müssen die Gegenbeispiele, die als Ergebnis der Modellprüfung zurückgegeben werden, daraufhin untersucht werden, ob diese ein Fehlverhalten des Systems beschreiben oder auf eine zu starke Abstraktion des Modells zurückzuführen sind. Zum Anderen bestehen teilweise nur eingeschränkte Möglichkeiten, das Ergebnis einer Verifikation auf Plausibilität zu prüfen. Damit bestehen nur eingeschränkte Möglichkeiten, strukturelle Fehler im Modell zu finden, die nicht die Aussage verletzen.

Damit wird die Modellprüfung im Rahmen dieser Arbeit als eine Technik zur Validierung

von Asbru-Modellen angesehen. Der Aufwand zur Verfeinerung des abstrahierten Modells zur automatischen Verifikation einer Aussage muss abgewogen werden gegen den Aufwand, der durch die interaktive Verifikation entsteht. Dabei zeigt sich, dass der Aufwand für die Abstraktion unverhältnismäßig ansteigt, wenn die Korrektheit einer Aussage von komplexen Datentypen einer Fallstudie und Zeit-Annotationen abhängt. Eine Ausnahme zu dieser Regel bildet die eingeschränkte Klasse der sogenannten strukturellen Aussagen, die in Abschnitt 20.2 vorgestellt werden.

Strukturelle Aussagen können als allgemeine Qualitäts-Indikatoren von Asbru-Modellen aufgefasst werden. Sie prüfen allgemeine Aussagen, wie etwa die Erreichbarkeit aller Asbru-Pläne einer Asbru-Hierarchie. Eine Verletzung einer solchen strukturellen Aussage gibt einen Hinweis auf eine mögliche Anomalie des Asbru-Modells, ist jedoch keineswegs als Beweis eines Fehlers der Leitlinie zu verstehen. Insofern ist der Aufwand, der durch die interaktive Verifikation entsteht, für die Verifikation dieser Aussagen nicht zu rechtfertigen. Die Modellprüfung wird jedoch aufgrund des geringen Aufwands erfolgreich dazu eingesetzt, auf diese Weise das Modell der Leitlinie zu prüfen und gegebenenfalls auf Komponenten des Modells zu verweisen, die Anomalien enthalten könnten.

19 Formalisierung des Patientenverhaltens

Der Begriff Patientenmodell beschreibt eine Modellierung der Veränderungen des Patientenzustands unter Berücksichtigung verordneter Medikamente und Behandlungen. In diesem Sinne des Wortes sind in Leitlinien meist keine Patientenmodelle enthalten. Die Ursachen dafür sind vielfältig. Zum Beispiel kann die Reaktion eines Patienten auf eine bestimmte Behandlung abhängig davon sein, wie gut diese Behandlung durchgeführt wurde. Dies kann aber von einer Leitlinie nicht erfasst werden. Ein anderes Beispiel, warum der Effekt einer Behandlung in der Regel nicht vorhergesagt werden kann ist, dass viele Medikamente bei unterschiedlichen Menschen unterschiedlich anschlagen. Dabei spielen nicht nur physiologische Faktoren eine Rolle sondern auch psychologische.

Im Allgemeinen können mit den hier vorgestellten Methoden daher nur sogenannte Prozessaussagen qualitätsgesichert werden, also solche Aussagen, die eine bestimmte Abfolge von Behandlungen vorschreiben, bestimmte Behandlungen verbieten oder zwingend erfordern. Das Gegenstück zu Prozessaussagen sind Ergebnisaussagen. Ein Beispiel dafür ist die Aussage, dass alle durch eine Leitlinie behandelte Patienten nach zwei Wochen beschwerdefrei sind. Aufgrund des fehlenden Patientenmodells sind solche Aussagen in der Regel nicht beweisbar. Es allerdings Fälle, in denen eine Modellierung von Patientenverhalten wichtige Einblicke in die Qualität der Leitlinie gibt. Beispielsweise kann die Modellierung eines „idealen“ Patienten, der stets auf definierte Weise auf Behandlungen reagiert, Einblick geben, ob die Leitlinie diesen minimalinvasiv behandelt. Auch gibt es Fälle, in denen ein Effekt der Leitlinie auf das Befinden des Patienten nachweisbar ist. Beispielsweise ist offensichtlich, dass bei einem Patienten, der an lokal begrenztem Brustkrebs leidet, dieser nicht länger existiert, wenn die komplette Brust entfernt wird. Die Modellierung von Patientenmodellen wird in [36, 32, 37] dokumentiert. Das dort vorgeschlagene Vorgehen wird in diesem Abschnitt beschrieben.

Zunächst wird ein Prädikat `knowledge` definiert, dass zwei Krankenakten-Einträge in ein Verhältnis zueinander setzt. In dieses Prädikat wird das Patientenmodell mit eingearbeitet. Dazu werden Axiome über dieses Prädikats definiert. Ein einfaches Beispiel ist die Modellierung des Alters des Patienten, angegeben in Jahren. Dieses Alter kann nie kleiner werden, allerdings gleichbleiben oder größer werden. Angenommen, das Zeitmodell ist entsprechend feingranular, so folgt, dass das Alter zwischen zwei Schritten immer gleich bleibt oder um genau eins anwächst. Dies lässt sich mit zwei Krankenakten-Einträgen `pre` und `post` wie folgt modellieren:

```
knowledge(pre, post) → post['age'] = pre['age'] ∨ post['age'] = pre['age'] + 1
```

Dabei ist `pre` der Zustand des Patienten vor Ausführung des Schrittes, `post` der Zustand des Patienten danach. Es gilt, dass das Alter in beiden Zustandsbeschreibungen entweder gleich ist oder in `post` um eins größer als in `pre`.

Es hat sich in Versuchen herausgestellt, dass es in diesem Zusammenhang sinnvoll ist, die Medikamente, die ein Patient erhält als Teil der Krankenakte zu modellieren. Dazu wird ein Eintrag `drugs` in die Krankenakte verwendet. Dieser enthält eine Menge aller gegenwärtig verabreichten Medikamente. Eine Umgebungsannahme sorgt dafür, dass diese Menge stets aktuell gehalten wird. Beispielsweise wird durch die nachstehende Umgebungsannahme sichergestellt, dass bei einer Ausführung des „dcis“ Plans stets das Medikament „CMF“ in der Medikamentenliste enthalten ist:

```
□      AS[nameGen('dcis', nameGen('bct', nameGen('treatment', superGen('chapter1'))))]
     = activated
   → 'CMF' ∈ PDH[AC]['drugs']
```

Das Patientenmodell kann dahingehend erweitert werden, dass die Tumorgröße unter Einfluss des „CMF“ Medikaments zurückgeht:

```
knowledge(pre, post) → ('CMF' ∈ pre['drugs'] → post['tumoursize'] ≤ pre['tumoursize'])
```

Analog lässt sich eine Aussage für die Hormontherapie formulieren. Dabei ist allerdings zu beachten, dass die Tumorgröße nur dann beeinflusst wird, wenn der Tumor Hormonrezeptorpositiv ist:

```
   knowledge(pre, post)
→     pre['pgR'] = positive ∨ pre['eR'] = positive
   → 'Tamoxifen' ∈ pre['drugs'] → post['tumoursize'] ≤ pre['tumoursize']
```

Diese Beispiele sollen zwei Aspekte der Patientenmodellierung verdeutlichen. Zum Einen ist es möglich, einzelne Wissenskomponenten unabhängig voneinander zu spezifizieren. Da das Prädikat `knowledge` lediglich impliziert, dass bestimmte Relationen zwischen den `pre` und `post` Zuständen gelten, können beliebig viele solcher Definitionen festgelegt werden, solange diese nicht widersprüchlich sind. Zum Anderen zwingt diese Form der Modellierung nicht, ein vollständiges Patientenmodell zu generieren sondern erlaubt, bekannte Aspekte zu spezifizieren und Unbekannte offen zu lassen. Speziell ist es möglich, partielle Änderungen zu definieren, also, dass ein Tumor kleiner wird, gleichzeitig aber nicht genau festzulegen, wie groß die Größenabnahme ist.

Wenn auf diese Weise das Patientenmodell festgelegt worden ist, kann als nächstes dieses mit dem Mikro- und Makro-Schritt Modell von Asbru verbunden werden. Dazu wird eine Umgebungsannahme definiert, die beschreibt, dass dich die Krankenakte während Mikro-Schritten nicht und während Makro-Schritten nur auf die vorgegebene Weise ändert:

```
□   Tick → knowledge(PDH[AC], pde) ∧ PDH'' = PDH[AC'', pde]
  ∧ ¬ Tick → PDH'' = PDH
```

Im Fall eines Tick soll ein Makro-Schritt durchgeführt werden. In diesem Schritt kann sich die Krankenakte ändern. Um diese Änderung zu berechnen, wird mittels des `knowledge` Prädikats ein neuer Krankenakten-Eintrag `pde` generiert. Nachdem dieser berechnet ist, kann die neue Krankenakte berechnet werden, indem die alte Krankenakte als Basis genommen wird und zur aktuellen Uhrzeit `AC''` um den berechneten Krankenakten-Eintrag ergänzt wird. Findet dagegen nur ein Mikro-Schritt statt, so bleibt die Krankenakte unverändert.

Diese Technik wurde auf das Beispiel der Behandlung von Diabetes angewendet. Dabei wurde die Interaktion zwischen Leitlinie und Medikamentenliste implementiert, also die Medikamentenliste automatisch auf Basis der laufenden Asbru-Pläne aktualisiert. Gleichzeitig wurde die Änderung des Zustands des Patienten durch die Einnahme der Medikamente modelliert. Das Ergebnis dieser Untersuchung war, dass die betrachtete Leitlinie immer dann eine minimalinvasive Behandlung garantiert, wenn der behandelnde Arzt den gegebenen Medikamenten „lang genug“ Zeit lässt, zu wirken. Die Arbeit wird im Detail in [36] beschrieben.

20 Beschreibung der Aussagentypen

In diesem Abschnitt werden verschiedene Klassen von Aussagen vorgestellt, die über Leitlinien formuliert werden können. Dazu wird diskutiert, wie diese Aussagen zu beweisen sind und welche Beweistechniken dafür zum Einsatz kommen. Die Aussagen werden dazu in vier Klassen eingeteilt. Die erste Klasse beschreibt syntaktische Aussagen über die Struktur von Asbru-Modellen. Diese werden in Abschnitt 20.1 vorgestellt. In Abschnitt 20.2 werden strukturelle Eigenschaften diskutiert und im Anschluss daran in Abschnitt 20.3 die medizinischen Aussagen.

20.1 Syntax der Asbru-Modelle

Eine erste Klasse von Aussagen prüft syntaktische und statische Eigenschaften des Asbru Modells. Die syntaktischen Aussagen entsprechen Prüfungen, ob die XML Repräsentation korrektes und gültiges XML darstellt. Um solche Prüfungen vorzunehmen, können XML-Parser wie Xerces[1] oder XML Spy[2] verwendet werden, die prüfen können, ob die XML-Modelle den Asbru-DTD Beschreibungen genügen.

Auf der Ebene dieser Asbru Modelle können auch weitere Aussagen überprüft werden. Beispielsweise darf eine Asbru-Planhierarchie keine Zyklen enthalten. Das bedeutet, dass ein Plan nicht in seiner eigenen Unterplanliste oder den Unterplanlisten seiner Unterpläne enthalten sein darf. Solche Prüfungen können durch Werkzeuge wie AsbruView[3] durch geführt werden.

Fehler, die durch diese Prüfung gefunden werden, haben ihren Ursprung üblicherweise nicht im ursprünglichen Modell sondern resultieren aus Fehlern des Übersetzungsprozesses.

20.2 Strukturelle Eigenschaften

Strukturelle Eigenschaften werden in [22, 23] definiert, um die ein Leitliniendokument zu validieren. Ziel ist es, eine Reihe von Aussagen zu definieren, die in der Regel von einer Leitlinie erfüllt sein sollen. Ein Beispiel für eine solche Aussage ist die Erreichbarkeit aller Pläne. Das bedeutet, dass jeder Plan die Möglichkeit erhält, in seine `Execution`-Phase einzutreten. Beispielsweise kann diese Aussage verletzt werden, wenn ein übergeordneter Plan zwar Unterpläne hat, jedoch diese niemals startet, weil er immer sofort nach seiner Aktivierung seine Ausführung beenden kann. Ein solcher Fall legt die Interpretation nahe, dass das Modell der Leitlinie Anomalien enthält, da bestimmte Behandlungen zwar definiert, jedoch niemals ausgeführt werden.

Nicht alle durch [23] definierten Aussagen weisen so eindeutig auf Fehler im Modell hin. Beispielsweise ist eine der dort definierten Aussagen, dass jeder Plan die Möglichkeit haben soll, seine Ausführung zu beenden. Dabei kann es durchaus erwünscht sein, dass ein Plan nicht beendet wird, der beispielsweise eine Überwachung des Patienten während der Operation durchführt. Stattdessen soll dieser Plan so lange laufen wie die eigentliche Operation und darf danach abgebrochen werden. Auch verlangt eine der Eigenschaften, dass kein Plan unendlich

[1] http://xerces.apache.org, besucht am 5.6.08
[2] www.Altova.com/XMLSpy, besucht am 5.6.08
[3] http://www.asgaard.tuwien.ac.at/asbruview/, besucht am 5.6.08

lange läuft. Dies ist zwar einerseits eine vernünftige Forderung im Hinblick auf die Endlichkeit einer medizinischen Intervention, andererseits aber verlangt der Beweis der Endlichkeit eine Reihe von zusätzlichen Annahmen. Diese sollen möglicherweise nicht gemacht werden, um das Asbru-Modell nicht zu sehr einzuschränken.

Damit ist die Aussage, dass ein Asbru-Modell hoher Qualität alle Aussagen von [23] erfüllen muss so nicht haltbar. Stattdessen sollten diese Aussagen als Anhaltspunkt dienen, welche Teile des Asbru-Modells daraufhin überprüft werden sollten, ob sie Anomalien enthalten könnten.

Bei einem durchschnittlichen Kapitel einer Leitlinie, das aus etwa 30 bis 40 Plänen besteht, sind etwa 600 solcher strukturellen Aussagen zu prüfen, wie in [10] beschrieben. Eine solche Zahl an Aussagen ist mit der interaktiven Verifikation nicht mehr handhabbar. Der Aufwand für die Verifikation steht auch in keinem Verhältnis zum Ergebnis, insofern als eine verletzte Aussage nicht unbedingt einen Fehler in der Leitlinie anzeigt, sondern allenfalls ein Hinweis auf einen möglichen Fehler. Daher werden solche Aussagen mit der automatischen Modell-Prüfung verifiziert, wie in Abschnitt 18 beschrieben.

Da die Verifikation dieser Aussagen nur Hinweise auf einen Fehler geben soll, ist die Möglichkeit, dass eine Aussage durch Modell-Prüfung fälschlicherweise als verletzt angesehen werden kann nicht von großer Bedeutung. Dies resultiert aus der Tatsache, dass bei strukturellen Aussagen die Ergebnisse der Verifikation ohnehin interpretiert werden müssen.

20.3 Medizinische Aussagen

Die in [23] definierten Aussagen sind vom Inhalt der modellierten Leitlinie unabhängig. Wie der Name bereits andeutet, handelt es sich um Aussagen, die die Struktur einer Leitlinie prüfen, nicht den Inhalt. Aussagen, die spezifisch den Inhalt einer Leitlinie prüfen werden als medizinische Aussagen bezeichnet. Dabei werden zwei Klassen medizinischer Aussagen unterschieden. Zum Einen die Klasse der Prozessaussage, die in Abschnitt 20.3.1 beschrieben wird, zum Anderen die Klasse der Ergebnisaussagen, die in Abschnitt 20.3.2 erläutert wird.

20.3.1 Prozessaussagen

Prozessaussagen beschreiben eine Ordnung der Asbru-Pläne, die erreicht oder vermieden werden soll. Verschiedene Formulierungen für Prozessaussagen sind möglich und sollen anhand von Beispielen kurz erläutert werden.

Beispielsweise verlangt die Behandlung von Brustkrebs, dass im Fall der brusterhaltenden Therapie diese von einer Strahlentherapie begleitet werden muss. Auch eine Reihenfolge wird angegeben, so wird verlangt, dass die Strahlentherapie immer nach der Operation angewendet wird. Ein anderes Beispiel ist, dass Frauen, deren Lymphknoten durch eine Operation entfernt wurden, keinesfalls Strahlentherapieerhalten dürfen. Eine solche Ordnung der Behandlungen wird als Prozessaussage bezeichnet.

Auch Teil der Prozessaussagen sind solche Aussagen, die auf Patientengruppen eingeschränkt sind. Ein Beispiel dafür ist, dass es Patientengruppen gibt, an denen aufgrund der Diagnose keine brusterhaltende Therapie durchgeführt werden darf. Dies ist der Fall bei allen Patienten, deren Tumor eine gewisse Ausdehnung innerhalb der Brust überschritten hat.

Allen diesen Prozessaussagen ist gemein, dass sie Behandlungsabfolgen beschreiben, die unter bestimmten Umständen erlaubt, verboten oder verpflichtend sind.

20.3.2 Ergebnisaussagen

Ergebnisaussagen unterscheiden sich von Prozessaussagen dahingehend, dass sie Forderungen an den Zustand des Patienten formulieren. Typische Prozessaussagen sind etwa, dass Patien-

ten nach einer gewissen Zeit beschwerdefrei sind oder dass die Langzeit-Überlebensrate einen Schwellwert übersteigt.

In der Regel zeichnen sich Ergebnisaussagen dadurch aus, dass die Leitlinie nur einen begrenzten Einfluss auf die Korrektheit hat. Beispielsweise ist die Wiederauftretensrate von Brustkrebs abhängig davon, wie gut eine Operation durchgeführt wird, welchem Stress der Patient in der Genesungsphase ausgesetzt ist oder ob bestimmte genetische Faktoren vorliegen. Diese Faktoren sind allerdings nur Indizien, die Hinweise auf den weiteren Verlauf der Krankheit geben können. Damit ist die Betrachtung des Ergebnisses von der Leitlinie weitgehend losgelöst.

Stattdessen ist es für die Verifikation solcher Aussagen notwendig, ein Patientenmodell zu erstellen, dass die Reaktion des Patienten auf die Behandlung abbildet. Ist dieses Patientenmodell genau genug, ist es möglich, Ergebnisaussagen zu verifizieren. Das größte Problem, dass diesem Ansatz entgegensteht, ist, dass in der Regel die genauen Wirkmechanismen von Behandlungen unbekannt sind. Zusätzlich ist zu beachten, dass die Effekte einer Behandlung manchmal überhaupt nicht sicher vorhergesehen werden können, da es beispielsweise paradox wirkende Medikamente gibt. Darunter werden Medikamente verstanden, die in der Regel eine bestimmte Wirkung haben, in Extremfällen, beispielsweise bei Unverträglichkeit oder falscher Dosierung, jedoch eine exakt gegenteilige Wirkung entfalten.

Auf solch unsicherer Grundlage ein Patientenmodell zu erstellen, dass konsistent mit der Realität ist und gleichzeitig stark genug, um Aussagen zu beweisen ist nur in eingeschränkten Fällen möglich.

20.4 Bewertung des Ansatzes

Der in dieser Arbeit vorgestellte Ansatz zur Qualitätssicherung von Aussagen ist primär für die Qualitätssicherung von Prozessaussagen entworfen und geplant worden. In diesem Zusammenhang wurden diverse Fallstudien erfolgreich behandelt worden.

Um mit den speziellen Anforderungen der strukturellen Aussagen umgehen zu können, wird die Technik der automatischen Modellprüfung eingeführt, die genau zu den Anforderungen dieser Validierungsaussagen passt.

Experimente zum Umgang mit Ergebnisaussagen haben ergeben, dass die Probleme, die im Umgang mit solchen Aussagen bestehen keine technische Probleme sind, sondern allein auf der Problematik basieren, dass das medizinische Wissen häufig zu vage ist, um eine erfolgreiche Verifikation durchführen zu können. In begrenzten medizinischen Domänen, wie etwa der Behandlung von Diabetes existiert genug medizinisches Wissen, so dass auch diese Technik anwendbar ist.

Teil V

Verifikation der Fallstudie

21 Aussage 20

Am Beispiel von Aussage 20 wird das Vorgehen bei der modularen Verifikation medizinischer Aussagen im Detail beschrieben. Bei den nachfolgenden Aussagen werden nur noch Beweisidee bzw. Abstraktionen genauer dargestellt, da die technischen Details dort bereits vorausgesetzt werden.

21.1 Inhalt der Aussage und Formalisierung

Aussage 20 verlangt, dass nach einer chirurgischen Entfernung der Lymphknoten der Achsel des Patienten keine Strahlentherapie angewendet wird. Wörtlich lautet die Aussage: „After Axillary clearance the axilla should not normally be irradiated“. Diese Aussage fällt in die Kategorie der „avoid during period“ Aussagen. Die Aussage wird hier gemäß der in Abschnitt 14 vorgestellten Technik formalisiert. Die Anpassung der Aussage, wie sie in [61] vorgestellt wird, wird hier nach Rücksprache mit Medizinern geringfügig modifiziert. Statt des Starts des Planes `axilla-staging-by-SN`, wie bei [61] angegeben, wird hier der Start des Plans `complete-axillary-node-dissection` (abgekürzt als `cand`) als Auslöser des Startereignisses gewählt. Nach Eintritt des Startereignisses soll keine Strahlentherapie mehr angewendet werden. Diese wird durch den Plan `primary-radiotherapy-of-axilla` (abgekürzt als `praoa`) repräsentiert.

Für eine weitergehende Formalisierung der Aussage muss die Planhierarchie, die für die Korrektheit der Aussage ausschlaggebend ist, betrachtet werden. Diese ist in Abbildung 21.1 dargestellt.

Aus dieser Grafik geht hervor, dass der erste Plan, der beide relevanten Pläne als Unterpläne hat, der Plan `dealing-with-axilla` (abgekürzt als `dwa`) ist. Damit wird die Beendigung dieses Plans als Endereignis gewählt. Entsprechend werden die Übergänge der Variablen, die Start- und Endereignis speichern durch die Umgebung kontrolliert. Das Startereignis tritt ein, sobald der Plan `cand` erfolgreich beendet wird:

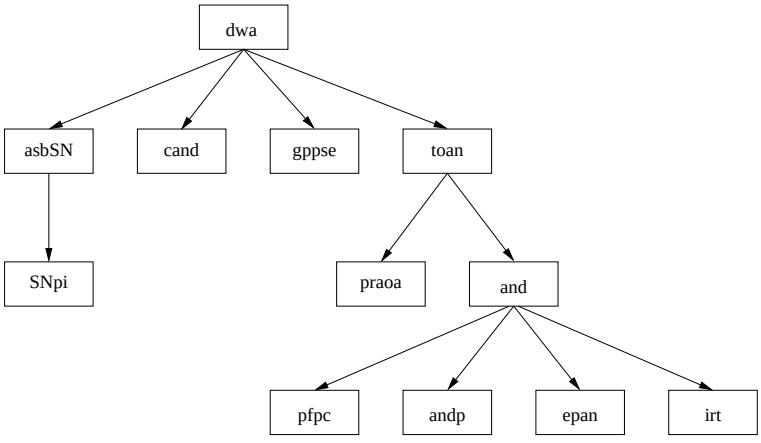


Abbildung 21.1: Relevante Planhierarchie für Aussage 20

```
□ ( (  AS[nameGen('cand', nameGen('dwa', iname))] ≠ completed
     ∧ AS'[nameGen('cand', nameGen('dwa', iname))] = completed)
   ⊃ StartEventTrigger''
   ; (StartEventTrigger' ↔ StartEventTrigger''))
```

Das Ende des Beobachtungsintervalls ist in der Aussage offen gelassen. Da es nach einer Terminierung des Planes `dwa` nicht länger möglich ist, den Plan `praoa` zu starten, wird diese Terminierung als Endzeitpunkt des Intervalls festgelegt. Dies wird wie folgt formalisiert:

```
□ ( (  StartEventTrigger ∧ ¬ isTerminated(AS, nameGen('dwa', iname))
     ∧ isTerminated(AS', nameGen('dwa', iname)))
   ⊃ EndEventTrigger''
   ; (EndEventTrigger' ↔ EndEventTrigger''))
```

Das Verhalten, das während des Beobachtungsintervalls nicht auftreten darf ist eine Aktivierung des Plans `praoa`. Ein solches Auftreten wird über die Variable `Observed` festgehalten, die immer dann auf den Wert **true** gesetzt wird, wenn die Aussage während des Beobachtungsintervalls verletzt wird. Formal wird dies wie folgt beschrieben:

```
□ ( (  StartEventTrigger
     ∧ (  AS[nameGen('praoa', nameGen('dwa', iname))] ≠ activated
        ∧ AS'[nameGen('praoa', nameGen('dwa', iname))] = activated))
   ⊃ Observed''
   ; (Observed' ↔ Observed''))
```

Zu zeigen ist dann, dass bei Eintreten des Endereignisses die `Observed` Variable den Wert **false** hat:

```
□ (¬ EndEventTrigger ∧ EndEventTrigger'' → Observed'')
```

21.2 Beweis der Aussage

In den Beweis der Korrektheit dieser Aussage sind insgesamt zwölf Pläne eingebunden. Davon können bis zu sieben Pläne gleichzeitig ausgeführt werden. Besonders die Tatsache, dass `dwa` ein `anyorder` kontrollierter Plan ist, bedeutet, dass seine Unterpläne unabhängig voneinander durch die `Selection`-Phase gehen können. Bedingt durch die existierenden **filter**-Bedingungen dieser Unterpläne muss eine große Zahl von Abläufen betrachtet werden.

Um zu vermeiden, diese einzeln zu betrachten, wird bei Aussage 20 die Beweisdekomposition eingesetzt. Ziel dieser ist es, das relevante Verhalten eines Plans in Form einer Rely-Guarantee Aussage zu kapseln und vom konkreten Verhalten des Plans zu abstrahieren. Dadurch wird beispielsweise davon abstrahiert, dass der Plan `asbSN` unendlich lange im `considered` Zustand verbleiben kann.

In diesem Sinne werden nun für die vier beteiligten Unterpläne von `dwa` geeignete Abstraktionsaussagen angegeben, so dass die Verifikation von Aussage 20 möglich wird. Den Anfang machen dabei die sogenannten passiven Pläne `asbSN` und `gppse`. Als passiv wird ein Plan dann bezeichnet, wenn seine Ausführung keinen Einfluss auf die Korrektheit der Aussage hat. Im Fall von Aussage 20 kann die Ausführung dieser beiden Pläne weder den Wahrheitswert des `StartEventTrigger` noch des `EndEventTrigger` oder der `Observed`-Variablen direkt beeinflussen.

Daher müssen für diese Pläne nur recht einfache Abstraktionen definiert und bewiesen werden. Beispielsweise muss gezeigt werden, dass der Plan `gppse` nur die ihm zugeordneten Felder des Asbru-Zustands verändert. Würde die Abstraktion des Plans `gppse` beliebiges Schreiben in der Asbru-Zustandsvariablen zulassen, so könnte `gppse` den Zustand des Plans

praoa verändern und so indirekt auf die Korrektheit der Aussage Einfluss nehmen. Die Abstraktion trifft keine Aussage darüber, wie die Zustandswechsel der passiven Pläne erfolgen. Im Fall von gppse wird die Aussage formal wie folgt gefasst:

```
AS' = AS[gppse AS'[nameGen('gppse', nameGen('and', iname))]]
```

Informell bedeutet dies, dass der einfach gestrichene und ungestrichene Zustand der AS Variablen gleich sind mit Ausnahme des Feldes gppse. Für dieses Feld trifft die Formel keine Aussage, es darf also seinen Wert beliebig verändern. Eine analoge Forderung wird für die Plan-Kommunikationsvariable PC aufgestellt:

```
PC' = PC[gppse PC'[nameGen('gppse', nameGen('and', iname))]]
```

Aufgrund der Beschaffenheit der anyorder Kontrolle werden alle Pläne, die in einem startbaren Zustand sind immer neu gestartet. Im vorliegenden Fall gilt, dass der Plan dwa seine untergeordneten Pläne nicht neu startet, wenn diese abgebrochen wurden. Somit sind die startbaren Unterpläne genau die, die im Zustand inactive sind. Daher wird durch eine weitere Annahme verhindert, dass der Plan gppse jemals seinen Zustand in den Wert inactive wechselt:

```
   AS[nameGen('gppse', nameGen('and', iname))] ≠ inactive
→ AS'[nameGen('gppse', nameGen('and', iname))] ≠ inactive
```

Wenn also initial der Plan nicht im Zustand inactive ist, so wird dies durch die Ausführung des Plans aufrecht erhalten.

Zuletzt muss noch eine Komponente ergänzt werden, die sagt, dass außer der Asbru-Zustandsvariablen und der Plankommunikationsvariablen keine Variablen geändert werden. Dazu wird die Rahmen-Annahme verwendet:

```
⌈AS, PC, RC, Tick⌉
```

Da der Plan diese Aussage unabhängig von einer Umgebungsannahme garantiert, kann zur Formulierung statt der **unless** Aussage eine □ Aussage verwendet werden. Im Fall der Umgebungsannahme **true** sind diese Formeln äquivalent, jedoch ist die Ausführung von □-Formeln deutlich effizienter als die von **unless**-Formeln.

Insgesamt lautet die Abstraktion für den Plan gppse mit all diesen Komponenten also:

```
□ (   ⌈AS, PC, RC, Tick⌉
   ∧ AS' = AS[gppse AS'[nameGen('gppse', nameGen('and', iname))]]
   ∧ PC' = PC[gppse PC'[nameGen('gppse', nameGen('and', iname))]]
   ∧    AS[nameGen('gppse', nameGen('and', iname))] ≠ inactive
     → AS'[nameGen('gppse', nameGen('and', iname))] ≠ inactive)
```

Auch der Plan asbSN ist ein passiver Plan. Für diesen soll eine analoge Abstraktion wie für gppse verwendet werden. Da asbSN jedoch einen Unterplan hat, kann nicht nachgewiesen werden, dass dieser Plan nur seinen eigenen Zustand ändert. Der Grund dafür liegt darin, dass asbSN die Prozedur starten kann, die den Unterplan kontrolliert und somit indirekt dessen Zustand verändert.

Um die Formulierung der Abstraktionen für große, passive Planhierarchien zu vereinfachen, werden ein Prädikat sowie eine Funktion definiert, die eine einfache Formulierung der Passivität von Planhierarchien erlauben. Die Funktion subs generiert die Liste aller Unterpläne eines angegebenen Plans. Dazu wird die Asbru-Definition des angegebenen Plans verwendet. Die korrekte Bildung der Instanznamen wird beachtet. Im Beispiel gilt:

```
  subs(nameGen('asbSN',nameGen('and', iname)))
=   nameGen('asbSN', nameGen('and',iname))
  + nameGen('SNpi', nameGen('asbSN', nameGen('and',iname)))
```

Das Prädikat `frame` ist eine kompakte Aufschreibung dafür, welche Aspekte zweier Zustandsvariablen sich nicht verändern. Beispielsweise wird das Prädikat wie folgt angewendet, um auszusagen, dass der Plan `asbSN` lediglich die Felder von sich und seinem Unterplan im Asbru-Zustand ändert:

```
frame(AS, AS', subs(nameGen('asbSN', nameGen('and', iname))))
```

Allgemein werden dem `frame`-Prädikat zwei Zustandsvariablen und eine Liste von Instanznamen übergeben, woraus dann eine Formel generiert wird, die aussagt, dass die beiden Zustandsvariablen überall gleich sind, außer an den Stellen, die durch die Planinstanznamensliste bezeichnet sind.

Analog zu der Abstraktionsaussage des Plans `gppse` kann der Rest der Aussage erzeugt werden. Damit ergibt sich die gesamte Abstraktionsaussage für den Plan `asbSN` wie folgt:

```
□ (   ⌈AS, PC, RC, Tick⌉
   ∧ frame(AS, AS', subs(nameGen('asbSN', nameGen('and', iname))))
   ∧ frame(PC, PC', subs(nameGen('asbSN', nameGen('and', iname))))
   ∧    AS[nameGen('asbSN', nameGen('and', iname))] ≠ inactive
     → AS'[nameGen('asbSN', nameGen('and', iname))] ≠ inactive)
```

Werden nun in der Sequenz die beiden Programme, die die Pläne `asbSN` und `gppse` beschreiben durch die Abstraktionsaussagen ersetzt, so wird vom genauen Verhalten der Programme abstrahiert. Dadurch reduziert sich die Zahl der explizit zu betrachtenden Pfade.

Um die Effizienz der Ausführung noch weiter zu erhöhen, sollen die Abstraktionen zusammengefasst und der Parallel-Operator, der sie verbindet, eliminiert werden. Dadurch werden technische Probleme, wie etwa der Nachweis, dass die beiden Programme untereinander keine Schreibkonflikte erzeugen, in einen getrennten Beweis ausgelagert. Die Gesamtaussage, die eine Abstraktion beider passiver Pläne ist, lautet wie folgt:

```
□ (   ⌈AS, PC, RC, Tick⌉
   ∧ frame(AS, AS', nameGen('gppse', nameGen('and', iname))
                    subs(nameGen('asbSN', nameGen('and', iname))))
   ∧ frame(PC, PC', nameGen('gppse', nameGen('and', iname))
                    subs(nameGen('asbSN', nameGen('and', iname))))
   ∧    AS[nameGen('asbSN', nameGen('and', iname))] ≠ inactive
     → AS'[nameGen('asbSN', nameGen('and', iname))] ≠ inactive)
   ∧    AS[nameGen('gppse', nameGen('and', iname))] ≠ inactive
     → AS'[nameGen('gppse', nameGen('and', iname))] ≠ inactive)
```

Wenn die beiden Pläne unabhängig voneinander garantieren, keine Zustandsfelder außerhalb der eigenen Planhierarchie zu verändern, so kann daraus gefolgert werden, dass die Pläne zusammengenommen diese Garantie aufrecht erhalten. Weiter gilt, dass die Pläne, wenn sie individuell garantieren, niemals in den Zustand `inactive` zu wechseln, dies auch als Kombination garantieren können, solange sie untereinander keine Schreibkonflikte verursachen.

Als nächstes werden die Abstraktionsaussagen für die nicht-passiven Unterpläne definiert und dann bewiesen. Die Aussagen aktiver Unterpläne bestehen jeweils aus zwei Komponenten. Zum Einen bestehen sie aus einer Komponente die analog zu den Abstraktionen für passive Pläne gebildet wird, zum Anderen aus einer Komponente, die planspezifisch erdacht werden muss. Die erste Komponente sorgt dafür, dass keine Schreibkonflikte entstehen können, während die zweite Komponente das abstrahierte Verhalten des Plans so weit einschränkt, wie für den Beweis der Aussage nötig.

Das heißt, dass zunächst für die aktiven Pläne `toan` und `cand` analoge Aussagen definiert werden, diese sind

```
□ (   ⌈AS, PC, RC, Tick⌉
   ∧ frame(AS, AS', subs(nameGen('toan', nameGen('and', iname))))
   ∧ frame(PC, PC', subs(nameGen('toan', nameGen('and', iname))))
   ∧    AS[nameGen('toan', nameGen('and', iname))] ≠ inactive
     → AS'[nameGen('toan', nameGen('and', iname))] ≠ inactive)
```

für den Plan `toan` sowie

```
□ (   ⌈AS, PC, RC, Tick⌉
   ∧ AS' = AS[cand AS'[nameGen('cand', nameGen('and', iname))]]
   ∧ PC' = PC[cand PC'[nameGen('cand', nameGen('and', iname))]]
   ∧    AS[nameGen('cand', nameGen('and', iname))] ≠ inactive
     → AS'[nameGen('cand', nameGen('and', iname))] ≠ inactive)
```

für den Plan `cand`. Zusätzlich zu diesen Aussagen muss das für den Beweis der Aussage 20 relevante Verhalten der Pläne extrahiert werden.

In der gegebenen Hierarchie gilt folgendes: Der übergeordnete Plan `dwa` ist ein `anyorder` Plan. Das bedeutet, dass von den Unterplänen nur jeweils einer gleichzeitig in die `Execution`-Phase gelangen kann, sofern die Unterpläne die Aktivierungssignale respektieren. Gleichzeitig können weder der Plan `cand` noch der Plan `toan` abgebrochen werden, solange `dwa` nicht terminiert. Dies ist darin begründet, dass keiner der Pläne eine **abort**-Bedingung hat und auch die Unterpläne von `toan` nicht abgebrochen werden können.

Wenn `dwa` einen der kritischen Unterpläne aktiviert, dann wird dieser so lange ausgeführt, bis er seine Bearbeitung beendet. Dann allerdings kann sich der Plan `dwa` auch sofort beenden und aktiviert keine weiteren Unterpläne. Nur einer der Unterpläne kann also in einen der Zustände `activated` bzw. `completed` kommen, weswegen die Aussage beweisbar ist.

Bei der Formalisierung dieses Verhaltens in Form der Abstraktionsaussagen ist zu beachten, dass diese Abstraktionsaussagen nach Möglichkeit keine Forderung nach Fortschritt enthalten sollten. Das heißt, eine Aussage, die verlangt, dass der Plan `cand` vom Zustand `ready` in den Zustand `activated` wechselt, wenn sein Aktivierungssignal vorliegt sollte vermieden werden. Der Grund dafür wird später erläutert. Stattdessen wird die Aussage so formuliert, dass `cand` aus dem Zustand `ready` nur in den Zustand `ready` oder den Zustand `activated` wechseln kann. Gleichzeitig wird das Verhalten so eingeschränkt, dass der Plan nicht den Zustand `activated` erreichen kann, wenn nicht das Aktivierungssignal gesendet wird. Auf diese Weise kann der Plan nicht zum Wechsel in die `Execution`-Phase gezwungen werden. Dies spielt beim Beweis der Aussage 20 jedoch keine Rolle und kann daher zugelassen werden.

Insgesamt besteht die spezifische Abstraktion für `cand` aus vier Komponenten. Die erste Komponente sagt aus, dass der Plan, wenn er aktiviert ist entweder aktiviert bleibt oder in den Zustand `completed` wechselt, solange `dwa` nicht in einem terminierten Zustand ist. Die zweite Komponente verlangt, dass der Plan, sofern er in der `Selection`-Phase ist, in dieser bleibt, solange `dwa` nicht terminiert und auch kein Aktivierungssignal gesendet wird. Die dritte Komponente sagt, dass der Plan vom Zustand `ready` aus im Zustand `ready` verbleibt oder in den Zustand `activated` wechselt, sofern `dwa` nicht terminiert ist. Die letzte Komponente beschreibt, dass der Plan bei Terminierung von `dwa` nicht in den Zustand `completed` wechseln kann. Formal wird das durch nachstehende temporallogische Formel beschrieben:

```
(  (  AS[nameGen('cand', iname)] = activated ∧ ¬ isTerminated(AS, iname)
    → AS'[nameGen('cand', iname)] = activated ∨ AS'[nameGen('cand', iname)] = completed)
    ∧ (    isSelected(AS, nameGen('cand', iname)) ∧ ¬ PC[nameGen('cand', iname)]
         ∧ ¬ isTerminated(AS, iname)
       → isSelected(AS', nameGen('cand', iname)))
    ∧ ( isReady(AS, nameGen('cand', iname)) ∧ ¬ isTerminated(AS, iname)
       → isReady(AS', nameGen('cand', iname)) ∨ isActivated(AS', nameGen('cand', iname)))
    ∧ ( AS[nameGen('cand', iname)] ≠ completed ∧ isTerminated(AS, iname)
       → AS'[nameGen('cand', iname)] ≠ completed))
```

```
unless ( ( AS[nameGen(‘cand’, iname)] = activated ∧ ¬ isTerminated(AS, iname)
             →   AS’[nameGen(‘cand’, iname)] = activated
                ∨ AS’[nameGen(‘cand’, iname)] = completed)
         ∧ (    isSelected(AS, nameGen(‘cand’, iname)) ∧ ¬ PC[nameGen(‘cand’, iname)]
              ∧ ¬ isTerminated(AS, iname)
             → isSelected(AS’, nameGen(‘cand’, iname)))
         ∧ ( isReady(AS, nameGen(‘cand’, iname)) ∧ ¬ isTerminated(AS, iname)
             →   isReady(AS’, nameGen(‘cand’, iname))
                ∨ isActivated(AS’, nameGen(‘cand’, iname)))
         ∧ ( AS[nameGen(‘cand’, iname)] ≠ completed ∧ isTerminated(AS, iname)
             → AS’[nameGen(‘cand’, iname)] ≠ completed)
         ∧ ¬ ( AS’’[nameGen(‘cand’, iname)] = AS’[nameGen(‘cand’, iname)]
              ∧ ¬ isSuspended(AS’’, iname)));
```

Die Formel sagt aus, dass die oben beschriebenen Komponenten so lange garantiert werden, wie die Umgebung sicherstellt, dass das `cand` zugeordnete Feld der Asbru-Zustandsvariablen nicht verändert wird. Außerdem darf der `cand` übergeordnete Plan nicht in seiner Ausführung unterbrochen werden. Bei der letzten Forderung handelt es sich um einen technischen Kniff. Da im kompletten Modell von Kapitel 1 der Fallstudie keine **suspend**-Bedingung vorkommt, ist es unmöglich, dass einer der Pläne dieses Kapitels in seiner Ausführung unterbrochen wird. Durch die Bedingung an die Umgebung werden also alle real vorkommenden Abläufe untersucht, jedoch dabei die ohnehin nicht existente Komplikation der Unterbrechung von Plänen ausgelassen.

Analog zu der Abstraktion von `cand` wird nun eine Abstraktion für den Plan `toan` definiert. Diese entspricht weitgehend der für den Plan `cand`, jedoch erweitert um eine Aussage zu seinem Unterplan `praoa`. Insgesamt besteht die Abstraktion aus fünf Teilkomponenten. Die Erste entspricht der von `cand`, in der festgelegt wird, dass der Plan nicht abbrechen kann. Das heißt, es wird sichergestellt, dass `toan`, sobald einmal der `activated` Zustand erreicht wurde, nur `activated` bleiben kann oder in den Zustand `completed` wechseln, zumindest solange `dwa` nicht terminiert. Auch die zweite Komponente wird analog übernommen und besagt, dass der Plan aus der `Selection`-Phase ohne Terminierung von `dwa` oder das Senden eines Aktivierungssignals nicht herauskommt. Die dritte Aussage ist ebenfalls analog und besagt, dass `toan` aus dem Zustand `ready` nur in den Zustand `ready` oder den Zustand `activated` wechseln kann, solange `dwa` nicht terminiert. Die vierte Komponente wird gegenüber der Abstraktion des `cand` Plans neu eingefügt und beschreibt das Verhältnis zwischen `toan` und seinem Unterplan `praoa`. `praoa` kann nach dieser Forderung nicht in den Zustand `activated` wechseln, wenn nicht bereits `toan` im Zustand `activated` ist. Wie bei der Aussage von `cand` beschreibt die letzte Komponente das Verhalten von `toan` im Fall der Terminierung von `dwa`. In diesem Fall gilt, dass `toan` nicht in den Zustand `activated` wechseln kann, wenn der Plan nicht bereits in diesem Zustand war.

Formal wird dies durch die nachstehende Aussage beschrieben:

```
  (  AS[nameGen(‘toan’, iname)] = activated ∧ ¬ isTerminated(AS, iname)
   → AS’[nameGen(‘toan’, iname)] = activated ∨ AS’[nameGen(‘toan’, iname)] = completed)
∧ (    isSelected(AS, nameGen(‘toan’, iname)) ∧ ¬ PC[nameGen(‘toan’, iname)]
     ∧ ¬ isTerminated(AS, iname)
   → isSelected(AS’, nameGen(‘toan’, iname)))
∧ ( isReady(AS, nameGen(‘toan’, iname)) ∧ ¬ isTerminated(AS, iname)
   → isReady(AS’, nameGen(‘toan’, iname)) ∨ isActivated(AS’, nameGen(‘toan’, iname)))
∧ (    AS[nameGen(‘praoa’, nameGen(‘toan’, iname))] ≠ activated
     ∧ AS[nameGen(‘toan’, iname)] ≠ activated
   → AS’[nameGen(‘praoa’, nameGen(‘toan’, iname))] ≠ activated)
∧ ( AS[nameGen(‘toan’, iname)] ≠ activated ∧ isTerminated(AS, iname)
   → AS’[nameGen(‘toan’, iname)] ≠ activated))
```

Bei dieser Aussage ist aus Gründen der Lesbarkeit nur die Garantie des Systems abgebildet. Das System verlangt als Umgebungsannahme, dass die Umgebung die Planzustände von

`toan` sowie dessen Unterplänen `and` und `praoa` unverändert lässt. Formal wird dies wie folgt aufgeschrieben:

```
   AS'[nameGen('toan', iname)] = AS''[nameGen('toan', iname)]
∧   AS'[nameGen('praoa', nameGen('toan', iname))]
  = AS''[nameGen('praoa', nameGen('toan', iname))]
∧   AS'[nameGen('and', nameGen('toan', iname))]
  = AS''[nameGen('and', nameGen('toan', iname))]
∧ ¬ isSuspended(AS'', iname);
```

In einem nächsten Schritt sollen nun alle temporallogischen Abstraktionen in eine einzige temporallogische Formel vereinigt werden, die keine Parallel-Operatoren enthält. Nach Anwendungen der Abstraktionen und der Vereinigung der Abstraktionen von `gppse` und `asbSN` ergibt sich folgende Sequenz:

```
   □ φ(gppse, asbSN)
‖s   □ φ(cand)
   ∧ G(cand) unless (G(cand) ∧ ¬ R(cand))
‖s   □ φ(toan)
   ∧ G(toan) unless (G(toan) ∧ ¬ R(toan))
```

Dabei ist $\varphi(...)$ die oben beschriebene Standard-Abstraktion für passive Pläne und G(..) bzw R(...) sind die oben beschriebenen Abstraktionen für die aktiven Pläne. Ziel ist es nun, alle Parallel-Operatoren zu eliminieren und am Ende eine einzige „geschlossene“ Formel der Form

```
   □ φ(gppse, asbSN, cand, toan)
∧ G(cand, toan) unless (G(cand, toan) ∧ ¬ R(cand, toan))
```

zu erlangen. Die Always Formeln werden dabei analog zur Vereinigung der Abstraktionen der passiven Pläne durchgeführt. Das heißt, die Gesamtaussage garantiert, keine Felder des Asbru-Zustands zu verändern außer denen, die von mindestens einer der beteiligten Komponenten verändert werden. Analog wird diese Forderung für die Plankommunikationsvariable aufgestellt. Garantiert eine der Teilkomponenten, dass der Zustand `inactive` niemals erreicht werden wird, so garantiert dies auch die Gesamtaussage.

Die Zusammenführung von Rely-Guarantee Aussagen wird ähnlich durchgeführt. Bei den im Rahmen dieser Fallstudie verwendeten Rely-Guarantee Aussagen gilt immer, dass diese durch einfache Konjunktion der einzelnen Garantien bzw. Relies durchgeführt werden können. Im Fall von Aussage 20 heißt das, dass G(`cand`, `toan`) = G(`cand`) ∧ G(`toan`) sowie R(`cand`, `toan`) = R(`cand`) ∧ R(`toan`). Damit der Beweis möglich ist, dass die Komponenten auch die Gesamtaussage erhalten, müssen jedoch für die einzelnen Garantien Bedingungen gelten.

Zum Einen sollten Garantien keine temporallogischen Operatoren enthalten, sondern, sofern möglich, immer rein prädikatenlogisch formuliert sein. Zum Anderen muss gelten, dass die einzelnen Garantien keinen Fortschritt beinhalten. Bei der Zusammenführung der Aussagen, also dem Beweis, dass die einzelnen Garantien die Gesamtgarantie erhalten, ist zu beachten, dass die beiden Komponenten unabhängig voneinander terminieren können. Das bedeutet, dass jede der Garantien für sich genommen sicherstellen muss, dass die Gesamtgarantie gilt. Konkret sind drei Beweisverpflichtungen zu zeigen. Zum Ersten, dass die beiden Komponenten zusammen die globale Garantie erhalten:

```
     □ φ(cand)
   ∧ G(cand) unless (G(cand) ∧ ¬ R(cand))
‖s   □ φ(toan)
   ∧ G(toan) unless (G(toan) ∧ ¬ R(toan))
⊢ G(cand, toan) unless (G(cand, toan) ∧ ¬ R(cand, toan))
```

Zum Zweiten, dass die erste Komponente die Gesamtgarantie erhält:

```
    □ φ(cand)
  ∧ G(cand) unless (G(cand) ∧ ¬ R(cand))
⊢ G(cand, toan) unless (G(cand, toan) ∧ ¬ R(cand, toan))
```

Zum Dritten, dass auch die zweite Komponente die Gesamtgarantie erhält:

```
    □ φ(toan)
  ∧ G(toan) unless (G(toan) ∧ ¬ R(toan))
⊢ G(cand, toan) unless (G(cand, toan) ∧ ¬ R(cand, toan))
```

Würde nun die Garantie des Gesamtsystems verlangen, dass der Plan `toan` irgendwann in die `Execution`-Phase eintritt, so müsste auch die Garantie von `cand` alleine dieses eintreten in die `Execution`-Phase sicherstellen können. Da aber `cand` garantiert, den Zustand von `toan` nicht zu verändern, kann dies nicht wahr sein.

Es gibt Fälle, in denen Fortschrittsforderungen in der Garantie notwendig sind. In diesen Fällen muss die Garantie allerdings auch das Terminierungsverhalten beschreiben, beispielsweise, dass `cand` nur terminiert, wenn er dabei gleichzeitig den Zustand `completed` erreicht. Dies ist aber nur möglich, wenn die Garantie den temporallogischen Operator **last** enthält, was vermieden werden sollte, falls möglich.

Mit den vier Abstraktionen der Unterpläne des Plans `dwa` wird auf diesem Weg eine Gesamtabstraktion generiert. Diese wird dann parallel zu `dwa` ausgeführt. Der Zustand muss dabei abstrahiert werden, so dass sichergestellt ist, dass keiner der Unterpläne im Zustand `inactive` ist (sonst würde er durch `dwa` nochmals gestartet werden), die relevanten Unterpläne `cand` und `toan` in der `Selection`-Phase sind und keiner der beiden Pläne ein Aktivierungssignal erhalten hat. Dann wird ein temporallogischer Schritt ausgeführt. Nach diesem Schritt sind im Wesentlichen vier Fälle zu unterscheiden.

Als Erstes muss der Fall betrachtet werden, in dem `dwa` keine Änderung an den Signalen von `toan` und `cand` durchgeführt hat. Die Abstraktionen garantieren, dass beide Pläne dann noch immer in der `Selection`-Phase sind und damit weder das Startereignis noch eine Aussagenverletzung stattgefunden hat. Dabei können durchaus Aktivierungssignale an `gppse` oder `asbSN` gesendet werden, da dies keinen Einfluss auf die Korrektheit der Aussage hat. Dieser Fall wird durch temporallogische Induktion geschlossen.

Der Zweite zu betrachtende Fall ist der in dem `toan` sein Aktivierungssignal erhalten hat. In diesem Fall ist bekannt, dass `toan` im Zustand `ready` ist, da er sonst das Signal nicht gesendet bekommen hätte. Die `anyorder` Kontrolle von `dwa` garantiert nun, dass keine weiteren Signale versendet werden, bis das Signal von `toan` konsumiert wurde. Die Abstraktion garantiert nun, dass `toan` solange im Zustand `ready` bleibt, bis er in den Zustand `activated` wechselt. Dann garantiert die Abstraktion, dass er so lange im Zustand `activated` bleibt, bis er in den Zustand `completed` wechselt, woraufhin `dwa` seinen Zustand ebenfalls nach `completed` ändert und keine weiteren Signale sendet. Damit kann dieser Fall abgeschlossen werden. In dem Spezialfall, in dem `toan` sein Signal konsumiert ohne in die `Execution`-Phase zu wechseln, kann Induktion angewendet werden.

Der Dritte zu betrachtende Fall ist der, in dem `cand` sein Aktivierungssignal gesendet wird. Dieser Fall verläuft analog zu dem obigen und kann aus den gleichen Gründen erfolgreich geschlossen werden.

Als letztes ist noch der Fall zu betrachten, in dem `dwa` terminiert, die Abstraktion der Unterpläne jedoch noch weiterläuft. In diesem Fall kann durch die Bedingungen bei Terminierung in den Garantien der Komponenten gezeigt werden, dass keine Aussagenverletzung stattfinden kann, wenn sie noch nicht bis dahin stattgefunden hat. Damit kann auch dieser Fall erfolgreich geschlossen werden, womit die Aussage bewiesen ist.

21.3 Besonderheiten der Aussage

Die Aussage kann als typischer Vertreter medizinischer Aussagen gelten. Aufgrund der relevanten Details des Aktivierungsverhaltens der Unterpläne ist sie als eher überdurchschnittlich komplex einzuordnen. Relevant ist das Aktivierungsverhalten eines `anyorder` kontrollierten Plans sowie die Abbildung des Verhaltens zweier Unterpläne. Davon ist einer kein direkter Unterplan des `anyorder` Plans sondern ein Unter-Unterplan.

Der Beweisaufwand für diese Aussage liegt bei etwa zwei Vollzeit-Tagen. Zu beachten ist dabei, dass dies nur den Netto-Beweisaufwand beschreibt. Dieser wird gerechnet für die Beweise der korrekten Abstraktionen, die Kompositionalitätsbeweise der Abstraktionen sowie dem eigentlichen Korrektheitsbeweis. Nicht mit einbezogen ist die Zeit, die für das Finden der korrekten Abstraktionen notwendig ist. Insgesamt benötigen alle Beweise zusammen knapp 900 Interaktionen und insgesamt etwa 2400 Beweisschritte.

22 Aussage 35

Aussage 35 ist ein Beispiel dafür, wie man Schwächen eines Leitlinien-Modells durch zusätzliche Annahmen ausgleichen kann und trotz dieser Schwächen die Korrektheit einer Aussage zu zeigen.

22.1 Inhalt der Aussage und Formalisierung

Aussage 35 verlangt, dass Patienten mit Brustkrebs die Wahl zwischen brusterhaltender Therapie und der kompletten Brustentfernung freigestellt sein soll. Dabei sollen mögliche Kontraindikatoren allerdings berücksichtigt werden. Im Original lautet die Aussage „Women with Stage I or Stage II breast cancer should be offered a choice of modified radical mastectomy or breast conserving surgery, unless contraindications to breast conserving therapy are present."

Nach Konsultationen mit den Medizinern wird die Anpassung der Aussage, wie sie in Dokument [61] angegeben ist, komplett verworfen. Stattdessen wird die brusterhaltende Therapie an den Plan `bct` (kurz für „breast conserving therapy") gebunden, die Mastectomy an den Plan `mast` (kurz für „mastectomy") und die Wahl des Patienten an den Parameter „parameter-patient-prefers-bct".

Die Aussage wird als „avoid during period" Aussage formuliert, wobei vermieden werden soll, dass der Plan `bct` aktiviert wird, wenn der Patient dies nicht wünscht und der Plan `mast` nicht aktiviert wird, wenn der Patient dies nicht wünscht.

Dies bedeutet, dass das Startereignis, also die `StartEventTrigger` Variable auf den Wert **true** gesetzt werden soll, wenn der Plan `treatment` in den Zustand `activated` wechselt. Die `EndEventTrigger` Variable soll auf den Wert **true** gesetzt werden, wenn der Plan `treatment` in den Zustand `completed` wechselt.

Die Aussage gilt als verletzt, wenn bei gesetztem „parameter-patient-prefers-bct" Flag der Plan `mast` aktiviert wird oder bei nicht-gesetztem Flag der Plan `bct` aktiviert wird. Dies wird formal wie folgt gefasst:

```
□  ( ( (  StartEventTrigger ∧ PDH[AC]['parameter-patient-prefers-bct'] .val
       ∧ AS[nameGen('mast', nameGen('treat', iname))] = activated)
     ∨ (StartEventTrigger ∧ ¬ PDH[AC]['parameter-patient-prefers-bct'] .val
       ∧ AS[nameGen('bct', nameGen('treat', iname))] = activated))
   ⊃ Observed''; (Observed' ↔ Observed'')),
```

Dabei wird eine Aussagenverletzung gleichgesetzt mit dem Setzen der `Observed` Variablen auf den Wert **true**. Zu beweisen ist, dass immer dann, wenn der `EndEventTrigger` auf den Wert **true** gesetzt wird, die Beobachtungsvariable `Observed` auf den Wert **false** gesetzt ist.

Bei der Formalisierung der Aussage wurde der Teil der Aussage, der über die Kontraindikationen redet, bewusst ausgelassen.

Der Beweis muss über die komplette Planhierarchie von Kapitel 1 geführt werden, die unterhalb des Plans `treatment` (abgekürzt `treat`) liegt. Diese ist in Abbildung 22.1 dargestellt.

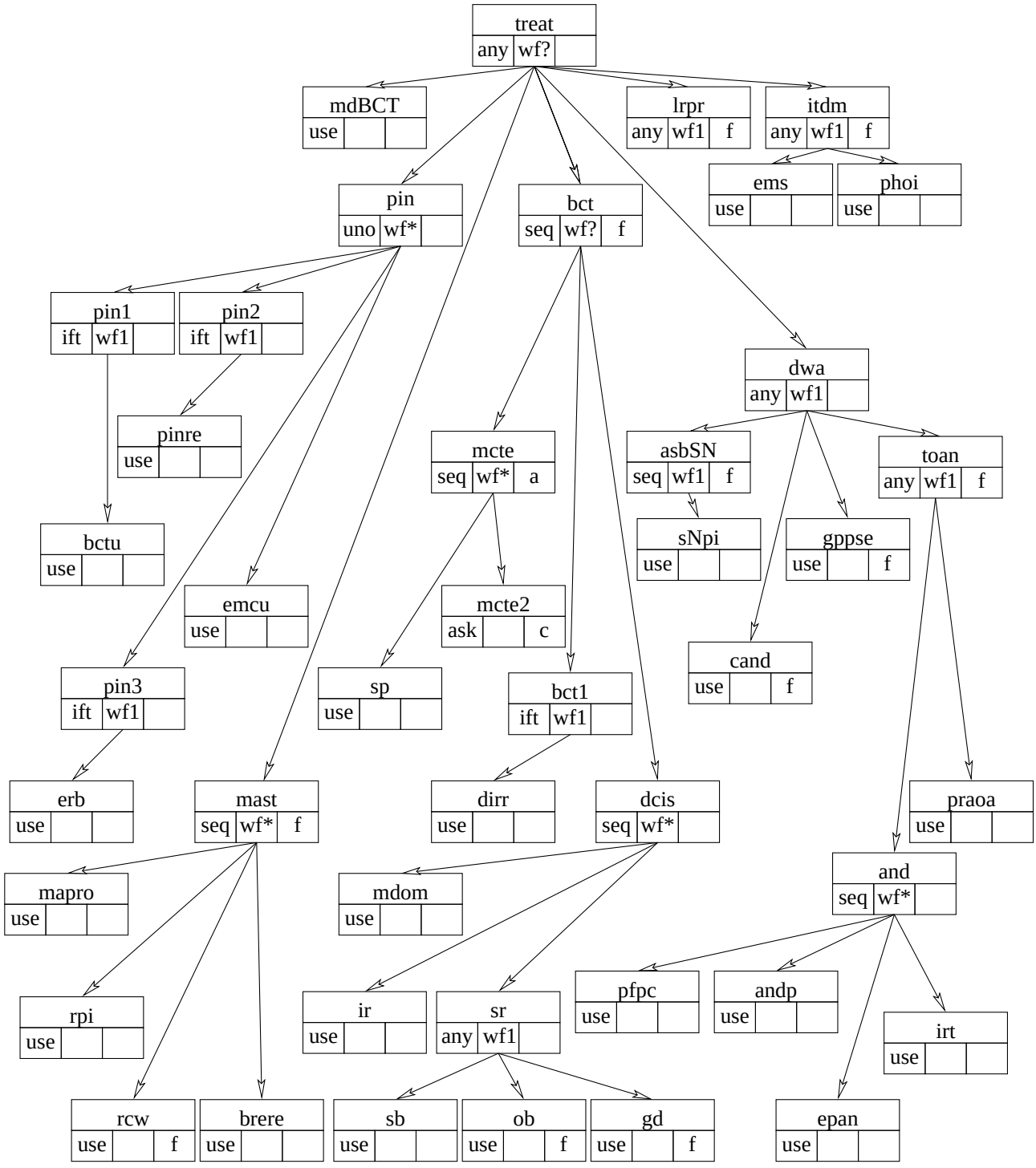


Abbildung 22.1: Relevante Planhierarchie für Aussage 35

22.2 Beweis der Aussage

Der Beweis der Aussage gestaltet sich zunächst als schwierig. Grund hierfür ist der Plan `mast`. Dieser enthält in seiner **filter**-Bedingung zwar korrekterweise eine Bedingung an den Parameter, der die Präferenz des Patienten festlegt, jedoch zusätzlich noch eine Forderung an einen Plan namens `reex`. Das Problem damit ist, dass der Plan `reex` in der gesamten Fallstudie nicht vorkommt.

Eine Anfrage bei den Medizinern ergibt die Auskunft, dass der Plan `reex` Teil der brusterhaltenden Therapie sein sollte, jedoch nicht im Modell von Kapitel 1 der Fallstudie enthalten ist. Dieser Plan wird aufgerufen, wenn die eigentliche brusterhaltende Therapie fehlschlägt und soll die fehlgeschlagene Therapie ersetzen.

Mit diesem Wissen ist davon auszugehen, dass der Plan `reex` nicht aktiviert wird, solange nicht die brusterhaltende Therapie fehlschlägt. Entsprechend wird eine Umgebungsannahme formuliert, die sagt, dass der Plan nicht aktiviert wird, solange `bct` nicht in den Zustand `aborted` wechselt.

In diesem Fall hat das Hintergrundwissen der Mediziner geholfen, einen Fehler im Modell zu umgehen. Nichtsdestotrotz ist die Abwesenheit des `reex` Planes eine Anomalie, die beseitigt werden sollte.

Der eigentliche Beweis der Aussage gestaltet sich nach Klärung dieses Sachverhalts als relativ einfach. Der Plan `bct` wird ohnehin nur gestartet, wenn der Patient dies wünscht und insofern wird eine Abstraktion dieses Plans gewählt, nach der der Plan den Zustand `considered` nicht verlassen kann, solange der Parameter „parameter-patient-prefers-bct“ nicht auf den Wert **true** gesetzt ist. Entscheidend ist nun noch die zweite Komponente der Aussage. Dabei ist zu zeigen, dass eine analoge Forderung auch für den Plan `mast` gilt. Dabei ist die Nebenforderung der **filter**-Bedingung des Plans `mast` zu beachten. Diese kann wahr werden, wenn der Plan `bct` seine Abarbeitung abbricht. Es kann allerdings im Modell gezeigt werden, dass dies nicht der Fall sein kann. `bct` selbst hat keine **abort**-Bedingung und seine notwendigen Unterpläne haben ebenfalls keine Möglichkeit abzubrechen. Daher kann bewiesen werden, dass `bct` stets nach der Aktivierung irgendwann in den Zustand `completed` übergeht.

Diese Abstraktion für `bct` muss über eine Hierarchie von insgesamt 13 Plänen bei einer Tiefe der Hierarchie von vier Stufen nachgewiesen werden. Insgesamt lautet die Garantie der Abstraktion für `bct` wie folgt:

```
   (  AS[nameGen('bct', iname)] ≠ rejected
    → AS'[nameGen('bct', iname)] ≠ rejected)
∧ ( AS[nameGen('bct', iname)] ≠ aborted
    → AS'[nameGen('bct', iname)] ≠ aborted)
∧ (   AS[nameGen('bct', iname)] = considered
     ∧ ¬ (PDH[AC]['parameter-patient-prefers-bct']) .val
    → AS'[nameGen('bct', iname)] = considered)
```

Die beiden oberen Implikationen sagen aus, dass `bct` nicht abbricht und nicht zurückgewiesen wird, die dritte Implikation garantiert, dass `bct` nicht den Zustand `considered` verlässt, wenn der Patient keine brusterhaltende Therapie wünscht. Diese Garantie erhält `bct` solange aufrecht, wie die Umgebung nicht die Zustände der Unterpläne ändert, der übergeordnete Plan terminiert oder der übergeordnete Plan seine Ausführung unterbricht.

Die Aussage des Plans `mast` ist geringfügig einfacher, da hier nicht garantiert werden muss, dass der Plan nicht abbricht. Die Garantie der Abstraktion lautet wie folgt:

```
    AS[nameGen('mast', iname)] = considered
  ∧ (PDH[AC]['parameter-patient-prefers-bct']) .val
  ∧ reduce(leave(superGen('reex'), activated), ASH, AC) = clock(⊥)
→ AS'[nameGen('mast', iname)] = considered)
```

Der Plan garantiert also nur, nicht den Zustand `considered` zu verlassen, solange der Patient keine brusterhaltende Therapie wünscht und gleichzeitig der Plan `reex` nicht aktiviert wurde und dann den Zustand `activated` verlassen hat. Diese Garantie hält der Plan solange aufrecht, wie der übergeordnete Plan seine Ausführung nicht unterbricht oder abbricht und so lange die Umgebung den Planzustand von `mast` nicht ändert.

Mit diesen Abstraktionen sowie den Standard-Abstraktionen für passive Pläne für die Unterpläne von `treat` kann die Verifikation erfolgreich durchgeführt werden. Dabei wird die gleiche Technik wie beim Beweis der Aussage 20 angewendet. Das heißt, zunächst werden Standardabstraktionen für alle passiven Pläne bewiesen und dann die Abstraktionen sowie die Standardaussagen für die Pläne `mast` und `bct`. Die Abstraktionen werden dann mit Modularisierungsbeweisen verknüpft, so dass am Ende die Prozedur, die den Plan `treat` repräsentiert parallel zu einer temporallogischen Formel ausgeführt wird. Insgesamt sind dabei 44 verschiedene Pläne zu berücksichtigen.

22.3 Besonderheiten der Aussage

Die Aussage ist insofern herausragend, als sie eine extrem große Zahl von Plänen umfasst. Dies macht sich bei den Modularisierungsbeweisen der Abstraktionen deutlich bemerkbar. Der Aufbau der Sequenzen durch KIV erscheint träge und die Durchführung von Simplifikationsschritten zäh. Insgesamt benötigt der letzte Modularisierungsbeweis, der alle Abstraktionen zusammenfasst mehrere Stunden, wobei normale Modularisierungsbeweise ähnlicher Bauart eher im Bereich weniger Minuten zur Verifikation benötigen.

Der Netto-Beweisaufwand für diese Aussage liegt bei circa zwei-einhalb Arbeitstagen. Zum Beweis sind insgesamt mehr als 2500 Interaktionen notwendig, die Zahl der Beweisschritte liegt jenseits von 6000. Bei diesem Aufwand ist allerdings zu beachten, dass ein sehr großer Anteil an dem Beweisaufwand für den Nachweis der Standardabstraktionen der passiven Pläne verwendet wird. Dieser Aufwand ist unabhängig von dieser Aussage nicht verloren, da diese Abstraktionsaussagen auch in anderen Beweisen wiederverwendet werden können.

Besondere Beachtung finden sollte das medizinische Ergebnis dieser Verifikation. Zum Einen ist es hier möglich, die Verifikation über einem fehlerhaften Modell aufgrund der Einbeziehung medizinischen Hintergrundwissens durchzuführen, zum Anderen zeigt der Beweis dieser Aussage, dass auch ein positives Beweisergebnis stets interpretiert werden muss. Im Verlauf des Beweises der Aussage ist es nämlich nicht notwendig, die zusätzliche Vorbedingung der originalen Aussage zu verwenden, nämlich das Vorliegen von Kontraindikationen der brusterhaltenden Therapie. Da die Korrektheit der Aussage unabhängig von diesem Fakt ist, stellt sich die Frage nach der Korrektheit des Modells.

Nachfragen bei Medizinern ergeben, dass die Ursache für dieses Problem in der besonderen Interpretation der brusterhaltenden Therapie dieser Leitlinie liegt. Die brusterhaltende Therapie wird hier gleichgesetzt mit einer „möglichst“ brusterhaltenden Therapie. Das heißt, während der Operationen werden bei Vorliegen von Kontraindikationen einfach so lange weitere Teile der Brust und der Lymphknoten der Achseln entfernt, bis am Ende alles vom Tumor befallene Gewebe entfernt wurde. Im Extremfall gleicht dieses Ergebnis dem einer Mastektomie. Somit werden die Kontraindikationen gegen die brusterhaltende Therapie implizit modelliert.

23 Aussage 38

23.1 Inhalt der Aussage und Formalisierung

Aussage 38 verlangt, dass die brusterhaltende Therapie stets von einer Strahlentherapie begleitet wird, um die Heilungschancen zu erhöhen. Das Startereignis wird mit dem Start der brusterhaltenden Therapie zusammengelegt (Plan `bct`), das Endereignis mit der erfolgreichen Beendigung derselben. Innerhalb des Intervalls soll beobachtet werden, dass Strahlentherapie gestartet wird, die durch den Plan `dcis-radiotherapy` (abgekürzt `dcis`) repräsentiert wird. Die relevante Planhierarchie für den Beweis dieser Aussage ist in Abbildung 23.1 dargestellt.

Es handelt sich bei Aussage 38 um eine „observe during period“ Aussage, bei der also innerhalb des Intervalls ein bestimmtes Verhalten beobachtet werden muss. Prinzipiell ist der Aufbau des GDL Konstruktes ähnlich zu dem der „avoid during period“ Aussagen. Das heißt, temporallogische Formeln bestimmen Zustandsübergänge der `StartEventTrigger` und `EndEventTrigger` Variablen, welche einen Zeitkorridor definieren. Innerhalb dieses Zeitkorridors wird eine `Observed` Variable auf den Wert **`true`** gesetzt, wenn ein bestimmtes Verhalten beobachtet wird. Der wesentliche Unterschied zu den „avoid during period“ Aussagen ist, dass hier nachgewiesen werden muss, dass bei Eintreten des Endereignisses die Variable `Observed` mit dem Wahrheitswert **`true`** belegt ist.

23.2 Beweis der Aussage

Für den Beweis der Aussage sind ausschließlich Standardabstraktionen notwendig. Die Ursache dafür liegt darin, dass der Plan `dcis` ein notwendiger Unterplan für `bct` ist, das heißt, ohne eine Beendigung von `dcis` kann `bct` nicht beendet werden.

Der Plan `bct` wird also ausgeführt, bis seine Unterpläne gestartet werden. Diese werden dann abstrahiert, so dass das Verhalten aller Unterpläne (und damit das von `dcis`) beliebig wird. Mit diesem beliebigen Verhalten sind nur noch zwei unterschiedliche Fälle zu berücksichtigen. Zum Einen der, in dem `dcis` in den Zustand `completed` wechselt, zum Anderen der, in dem dies nicht geschieht. Im ersten Fall ist die Aussage sofort erfüllt, da das gewünschte Verhalten beobachtet werden kann. Im zweiten Fall wird das gesuchte Verhalten zwar nicht beobachtet, jedoch kann dann auch der Plan `bct` nicht seine Bearbeitung abschließen.

Damit kann das Endereignis unabhängig vom Verhalten der Unterpläne nur dann eintreten, wenn auch die Aussage wahr ist.

23.3 Besonderheiten der Aussage

Der Beweis dieser Aussage nutzt Spezifika von Asbru aus um vom Verhalten der Unterpläne fast gänzlich zu abstrahieren. Diese Aussage ist ein Beispiel dafür, wie weit man bei der Abstraktion von Unterplänen zum Teil gehen kann, um eine hohe Effizienz beim Beweisen zu erreichen.

Insgesamt benötigt der Beweis dieser Aussage etwa einen halben Arbeitstag, wobei dabei Teile der für Aussage 35 definierten Abstraktionsaussagen wiederverwendet werden können. Insgesamt sind für den Beweis 725 Interaktionen und 1849 Beweisschritte notwendig, wobei

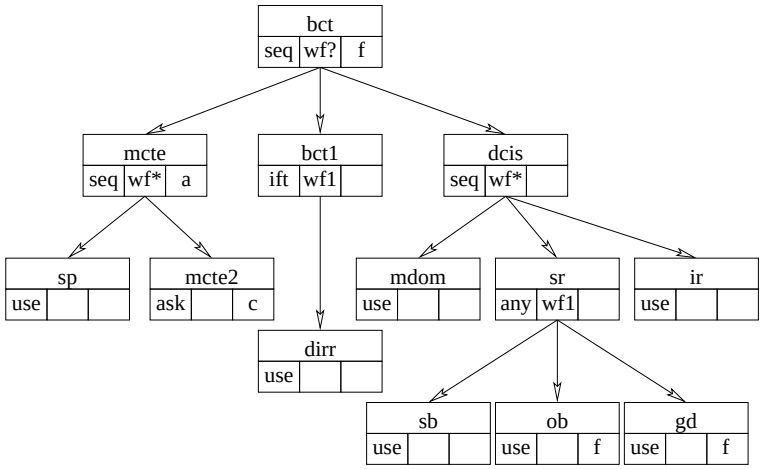


Abbildung 23.1: Relevante Planhierarchie für Aussage 38

davon 644 Interaktionen und 1629 Beweisschritte von Hilfsaussagen anderer Beweise wiederverwendet werden können.

24 Aussage 1

Aussage 1 ist ein Beispiel dafür, warum die Struktur eines Beweises manchmal für die Bewertung des Verifikationsergebnisses relevant sein kann. Im vorliegenden Fall zeigt sich, dass die Aussage falsch an das Modell angepasst wurde. Der Fehler wird offenkundig, da der Beweisbaum lediglich aus einem einzigen temporallogischen Schritt besteht.

24.1 Inhalt der Aussage und Formalisierung

Die informelle Form der Aussage 1 lautet: „No breast-conserving surgery or mastectomy in metastatic disease“. Diese Aussage fällt klar in die Kategorie der „avoid during period“ aussagen, wobei chirurgische Behandlungen unterlassen werden sollen, sobald Metastasen festgestellt werden können. Die Aussage wird laut [61] so formalisiert, dass während der Abarbeitung des ersten Kapitels der Fallstudie kein chirurgischer Eingriff festgestellt werden darf, wenn die Diagnose auf metastasierenden Brustkrebs lautet oder Metastasen in den Lymphknoten nachgewiesen werden können.

Diese Aussage ist in dieser Form nicht beweisbar, da jeder Patient, der entsprechend dem ersten Kapitel der Leitlinie behandelt wird, zwangsläufig operiert wird und Patienten mit Metastasen in den Achselknoten möglicherweise auf Basis des ersten Kapitels der Leitlinie behandelt werden. Eine Rückfrage an die Mediziner zu diesem Sachverhalt ergibt, dass die Anpassung der Aussage an die Leitlinie einen Fehler enthält.

Die originale Aussage verlangt die Unterlassung von Chirurgie bei metastasierendem Brustkrebs. Die Aussage ist an dieser Stelle nicht genau genug formuliert, da tatsächlich nur bei sogenannten fernen Metastasen („distant metastasis“) die Chirurgie unterlassen werden soll. Metastasen in den Lymphknoten der Achseln hingegen werden bei Brustkrebs als spezielle Ausprägung von lokal begrenztem Krebs angesehen und fallen somit nicht unter die Ausschluss Kriterien für die chirurgische Behandlung.

Entsprechend wird mittels dieser Aussage nachgewiesen, dass kein Patient mit der Diagnose „metastasierender Brustkrebs“ durch Kapitel 1 der Fallstudie chirurgisch behandelt wird. Im Folgenden werden die Komponenten der Aussagen stückweise beschrieben.

Die Vorbedingung der Aussage ist, dass die betrachteten Patienten die Diagnose „metastasierender Brustkrebs“ gestellt bekommen haben. Dies wird formalisiert durch die temporallogische Formel

```
□  PDH[AC]['parameter-diagnosis'] .val = metastasised-BC
```

Es wird also verlangt, dass die Diagnose zumindest für die Dauer der Behandlung nicht geändert wird. Das Startereignis für diese Aussage wird festgelegt auf den Übergang des Planes `chapter1` von einem beliebigen Zustand in den Zustand `activated`. Dies wird über die nachstehende temporallogische Formel spezifiziert:

```
□ ( (  AS[superGen('chap1')] ≠ activated
    ∧ AS'[superGen('chap1')] = activated)
  ⊃ StartEventTrigger''
  ; (StartEventTrigger' ↔ StartEventTrigger''))
```

Das Endereignis des Beobachtungsintervalls ist die Terminierung des `chapter1`-Planes. Dies wird durch nachstehende temporallogische Formel formalisiert:

```
□ ( (  StartEventTrigger ∧ ¬ isTerminated(AS, superGen('chap1'))
     ∧ isTerminated(AS', superGen('chap1')))
   ⊃ EndEventTrigger''
   ; (EndEventTrigger' ↔ EndEventTrigger''))
```

Die Aussage verlangt, dass während des angegebenen Intervalls nicht beobachtet werden kann, dass ein Plan zur chirurgischen Tumorentfernung gestartet wird. Um dies festzustellen, wird eine boolesche Variable `Observed` eingeführt. Diese soll initial auf den Wert **false** gesetzt sein und genau dann auf **true** umgesetzt werden, wenn ein entsprechender Plan gestartet wird. Dies wird durch die nachfolgende temporallogische Formel spezifiziert:

```
□ ( (  StartEventTrigger
     ∧ (  AS'[nameGen('bct', nameGen('treat', superGen('chap1')))] = activated
        ∨ AS'[nameGen('mast', nameGen('treat', superGen('chap1')))] = activated))
   ⊃ Observed''
   ; (Observed' ↔ Observed''))
```

Zuletzt muss noch der aktuelle Zustand beschrieben werden. Im Fall dieser Aussage ist es ausreichend, zu verlangen, dass alle drei relevanten Variablen, `StartEventTrigger`, `EndEventTrigger` und `Observed` initial nicht wahr sind. Die Systembeschreibung besteht aus der `inactive#` Prozedur, die parametrisiert als Plan `superGen('chap1')` gestartet wird. Die zu zeigende Aussage ist dann, dass in dem Schritt, in dem das Endereignis eintritt, die Variable `Observed` noch immer auf den Wert **false** gesetzt wird.

24.2 Beweis der Aussage

Der Beweis der Aussage erfolgt durch einfache, symbolische Ausführung ohne Anwendung der Beweisdekomposition. Nach Ausführen des ersten temporallogischen Schrittes bleiben alle drei Beobachtungsvariablen in ihren Werten unverändert, das heißt, weder Start- noch Endereignis sind eingetreten und es konnte auch keine Aussagenverletzung festgestellt werden. Der Plan `superGen('chap1')` ändert seinen Zustand durch diesen Schritt auf den Wert `considered`. Im nächsten Schritt wird die **filter**-Bedingung des Plans geprüft. Diese testet, ob der behandelte Patient an DCIS oder operablem Brustkrebs leidet. Da beides nicht der Fall ist und die Bedingung mit einer **now_ata**-Zeitannotation versehen ist, bleibt der Plan bei diesem und allen weiteren Schritten im Zustand `considered`, um weiter auf die Erfüllung der **filter**-Bedingung zu warten. Nach einem weiteren Schritt ist es möglich, Induktion anzuwenden und den Beweis zu schließen.

24.3 Besonderheiten der Aussage

Die Aussage wird zwar nominell durch die Leitlinie erfüllt, jedoch legt die Trivialität des Beweises nahe, dass die Anpassung dieser Aussage an die Leitlinie noch immer fehlerhaft ist. Die Behandlung der Patienten, die durch die Vorbedingung beschrieben werden, wird nicht von Kapitel 1 der Leitlinie beschrieben. Da Kapitel 1 der Leitlinie in diesem Fall überhaupt keine Behandlung vorsieht, wird logischerweise auch keine falsche Behandlung gestartet.

Gleichzeitig zeigt die Verifikation der Aussage noch eine zweite Anomalie, nämlich die, dass die Auswertung der Diagnose durch die **filter**-Bedingung des `superGen('chap1')` Plans unendlich oft wiederholt wird, bis ein positives Ergebnis eintritt. Da ohne eine Behandlung die Diagnose nie geändert werden kann, ist ein Warten auf eine Änderung der Bedingung hier sinnlos.

Die Aussage ist ein gutes Beispiel für die Notwendigkeit bzw. die Nützlichkeit interaktiver Beweismethoden. Die Anwendung automatischer Modellprüfung zum Testen dieser Aussage

hätte zum Ergebnis gehabt, dass die Aussage erfüllt ist, jedoch ohne auf die kleine Beweisgröße hinzuweisen. Ein manuelles Aufbauen des Zustandsraumes dagegen vermittelt ein Gefühl für bestimmte Komplexitäten und die Anwendung von Beweisdekomposition zwingt zum Nachdenken über relevantes und irrelevantes Verhalten.

25 Bewertung des Ansatzes

In diesem Kapitel wird eine Bewertung des vorgestellten Ansatzes hinsichtlich seiner Anwendbarkeit vorgenommen. Dabei wird der nötige Aufwand für die Verifikation, die Qualität der Leitlinie und die Ergebnisse der Verifikation in Verhältnis zueinander gesetzt.

25.1 Aufwand der Verifikation

In dieser Arbeit wird die Korrektheit der Brustkrebs-Fallstudie durch die Verifikation von insgesamt elf Aussagen untersucht. Von diesen elf Aussagen wurden hier vier ausgewählt und detailliert erläutert. Die übrigen bewiesenen Aussagen sind in Anhang D aufgeführt. Für die Verifikation der elf Aussagen wurden insgesamt etwa vier Personenmonate benötigt. Die Beweise bestehen aus insgesamt 20269 Beweisschritten bei 6234 Benutzerinteraktionen. Der Automatisierungsgrad der Beweise liegt zwischen 61% und 98%.

Dieser Aufwand muss ins Verhältnis zum Aufwand für die Modellierung von Leitlinien gesetzt werden. So wurden für die Modellierung der Brustkrebs Leitlinie etwa zwei Personenjahre aufgewendet. Die Verifikation nimmt somit nur etwa ein Sechstel der Zeit der Modellierung der Leitlinie ein. Der Beweisaufwand lässt sich zumindest teilweise gut parallelisieren. Die Beweise der unterschiedlichen Kapitel der Leitlinie sind unabhängig voneinander. Teilweise können selbst innerhalb eines Kapitels viele der Beweise parallel bearbeitet werden. An der vorliegenden Fallstudie hätten problemlos drei bis vier Personen gleichzeitig arbeiten können ohne sich zu behindern.

Diese Arbeit zeigt somit, dass die Verifikation medizinischer Leitlinien nicht nur möglich, sondern der Aufwand für die eigentliche Verifikation auch überschaubar ist. Für die Verifikation einer Fallstudie mit ausreichend Personal müssen etwa ein bis zwei Monate angesetzt werden.

25.2 Qualität der Leitlinie

Es zeigt sich, dass die einzelnen Abschnitte der Brustkrebs-Leitlinie höchst unterschiedliche Qualität in Bezug auf die Verifikation aufweisen. Ursache dafür ist eine Entscheidung, die bei der Modellierung der Leitlinie getroffen wurde. Leitlinien enthalten neben den wissenschaftlich gesicherten Aussagen auch Abschnitte, die nicht durch evidenzbasierte Forschungen abgedeckt sind. Solche Fälle ergeben sich immer dann, wenn beispielsweise randomisierte Studien aufgrund ethischer Bedenken nicht durchgeführt werden können, etwa bei der Behandlung von Kindern. Die Leitlinie sollte solche Bereiche klar voneinander abgrenzen, so dass ersichtlich wird, welche Teile wissenschaftlich belastbar sind und welche nicht. In der vorliegenden Leitlinie gibt es diesbezüglich Probleme.

In Fällen, in denen Fakten nicht als wissenschaftlich fundiert gekennzeichnet wurden, wurden diese nicht in das formale Modell der Leitlinie übernommen. Damit einher geht allerdings ein teilweiser Verlust der Struktur im formalen Modell der Leitlinie. Dies zeigt sich recht deutlich bei der formalen Untersuchung von Kapitel 1 der Fallstudie. Die fehlende Struktur überdeckt dort die Fehler der informellen Leitlinie. Anomalien, die durch die Verifikation aufgedeckt werden können, lassen sich dadurch nur selten bis in die informelle Leitlinie zurückverfolgen. Die aufgefundenen Anomalien weisen häufig nur auf Fehler des Modellierungsprozesses

hin.

Im Gegensatz dazu steht beispielsweise das Modell von Kapitel 2. Dort wird ein Entscheidungsbaum in der Leitlinie formalisiert. Alle Entscheidungen dieses Entscheidungsbaumes sind mit wissenschaftlich gesicherten Daten unterlegt und dies wird auch in der Leitlinie kenntlich gemacht. Somit kann der Baum vollständig in das formale Modell übertragen werden. Dadurch können Anomalien aufgefunden werden, die sich bis in die informelle Leitlinie zurückverfolgen lassen. Beispiele hierfür sind die Behandlung Schwangerer, die an einer Stelle nicht ausreichend berücksichtigt wird, sowie die Behandlung alter Patienten. In letzterem Fall liegen der Leitlinie und der zu zeigenden Aussage unterschiedliche Definitionen des Begriffs „alt“ zugrunde. Im Grenzbereich schreiben Aussage und Leitlinie daher unterschiedliche Behandlungen vor, was zu einer Verletzung der Aussage führt.

Die Qualität des formalen Modells der Leitlinie ließe sich verbessern, indem die Formalisierung bereits früher im Entwicklungszyklus der informellen Leitlinie Beachtung findet. Ohne Veränderungen des Leitlinien-Modellierungsprozesses wird die Verifikation in erster Linie Fehler auffinden, die durch die Formalisierung der Leitlinie entstanden sind. Diese Fehlersuche ist sehr nützlich und hilfreich, wenn beispielsweise das formale Modell der Leitlinie als Grundlage eines Entscheidungsunterstützungssystems verwendet werden soll. Für die alleinige Sicherung der Qualität der ursprünglichen Leitlinie dagegen scheint der Aufwand relativ zum Ergebnis recht hoch, da Verifikationsmaßnahmen größtenteils zur Aufdeckung von Problemen der Modellierung führen.

25.3 Anomalien in der Leitlinie

Insgesamt konnten durch die formale Behandlung der Leitlinie 26 Anomalien aufgedeckt werden. Zunächst wurden strukturelle Aussagen untersucht, die in Kapitel 20.2 beschrieben werden. Zur Verifikation dieser Aussagen wurde die automatische Modellprüfung angewendet. Dadurch konnten insgesamt zwölf Anomalien aufgedeckt werden. So konnte beispielsweise gezeigt werden, dass ein großer Teil des ersten Kapitels niemals ausgeführt werden kann. Nach Korrektur dieser Anomalien wurden 17 medizinische Aussagen untersucht, davon neun mit dem Gegenbeispielmechanismus und acht mit der interaktiven Verifikation. Nach Rücksprache mit Medizinern konnten in drei bereits als falsch nachgewiesenen Aussagen Fehler in der Aussagenformalisierung gefunden werden. Nach Korrektur dieser Fehler verlief die Verifikation erfolgreich.

Die interaktive Verifikation erlaubt nicht nur den Nachweis der Korrektheit von Aussagen, sie erzwingt auch die Auseinandersetzung mit der Struktur der Beweise. Dadurch ist es möglich, Fehler zu erkennen, selbst wenn eine Aussage bewiesen werden kann. Dies ist beispielsweise bei Aussage 1 der Fall. Dort kann die Aussage zwar gezeigt werden, die Struktur des Beweises deutet jedoch auf eine Anomalie hin. Bei genauer Untersuchung stellte sich heraus, dass die Aussage falsch formuliert wurde.

25.4 Zusammenfassung

Es können auch in einer Leitlinie hoher Qualität zahlreiche Anomalien gefunden werden. Der Großteil der gefundenen Anomalien lässt sich allerdings nicht bis in die informelle Leitlinie zurückverfolgen. Stattdessen zeigt sich, dass diese Anomalien während der Formalisierung eingeführt werden. Gleichzeitig fällt auch der Großteil des Aufwands, der für die Untersuchung einer Leitlinie anfällt, bei der Formalisierung an. Für die formale Qualitätssicherung einer Leitlinie werden mit den vorgestellten Werkzeugen und Vorgehensmustern ungefähr zweieinhalb Personenjahre benötigt. Betrachtet man die Ergebnisse der Verifikation so zeigt sich, dass die Erstellung eines Asbru-Modells einer Leitlinie fehleranfälliger ist als das Erstellen

einer Leitlinie selbst. Dies liegt zum Teil daran, dass der Leitlinienerstellungsprozess besser verstanden und erprobt ist als die Leitlinienformalisierung. Bevor diese Form der Qualitätssicherung im Alltag eingesetzt werden kann, muss zunächst die Technik zur Erlangung des formalen Modells verbessert werden.

Die Bedeutung eines korrekten Asbru Modells muss allerdings auch über die Qualitätssicherung der informellen Leitlinie hinaus gesehen werden. Neben der eigentlichen Qualitätssicherung bedeutet das Vorliegen eines korrekten Asbru Modells auch die Möglichkeit, die Leitlinie in elektronischer Form zum Beispiel in einem Entscheidungsunterstützungssystems verwenden zu können.

Interessant ist für die Zukunft eine Untersuchung unterschiedlicher Leitlinien, die von unterschiedlichen Institutionen erzeugt wurden. Möglicherweise könnten mit einer solchen Untersuchung Vorgehensmodelle identifiziert werden, die eine Formalisierung des informellen Dokuments erleichtern. Übergeordnetes Ziel sollte jedoch die Erstellung des formalen Modells ohne Umweg über eine informelle Repräsentation sein. Erst mit einem solchen Vorgehen lässt sich ausschließen, dass Fehler des Referenzmodells durch Formalisierungsfehler überdeckt werden.

Teil VI

Verwandte Arbeiten und Diskussion

26 Verwandte Arbeiten

Nach der Vorstellung des Ansatzes zur Verifikation medizinischer Leitlinien wird in diesem Kapitel Bezug genommen auf ähnliche Forschungsarbeiten anderer Forschergruppen.

26.1 Ansätze zur Qualitätssicherung in der Medizin

Es existieren verschiedene Ansätze, um Qualitätssicherung im Rahmen medizinischer Leitlinien zu betreiben. Derzeit geht die Initiative für solche Ansätze hauptsächlich von Medizinern aus, die an der Schnittstelle zwischen Medizin und Informatik arbeiten. Daher ist eine so umfassende Methodik mit dem Einsatz formaler Methoden wie der hier vorgestellte Ansatz derzeit noch nicht zu finden. Stattdessen liegen die Schwerpunkte der vergleichbaren Arbeiten eher im Bereich der praktischen Umsetzung von Entscheidungsunterstützungssystemen und dem Umgang mit den damit verbundenen Komplikationen.

Jeder der hier vorgestellten Ansätze setzt auf eine eigene Sprache zur formalen Erfassung medizinischer Leitlinien auf. Diese Sprachen werden in [54] anhand von einer Fallstudie beschrieben und vorgestellt. Das Ergebnis des Vergleichs ist, dass jede Sprache unterschiedliche Schwerpunkte setzt und damit für verschiedene Anwendungsszenarien unterschiedlich gut geeignet ist. So liegt der Schwerpunkt der in dieser Arbeit verwendeten Sprache Asbru im Umgang mit zeitlichen Bedingungen.

26.1.1 PROforma

PRO*forma* [28, 27] ist eine medizinische Plansprache. Die vier grundlegenden Konzepte von PRO*forma* sind Pläne, Entscheidungen, Nachfragen und Handlungen. Im Prinzip ist diese Aufteilung nicht unähnlich der von Asbru. Dabei entsprechen die Nachfragen den `ask` Plänen, Handlungen den `user` Plänen, Entscheidungen den `ifThenElse` Plänen und PRO*forma*-Pläne den übrigen Plantypen in Asbru.

Für PRO*forma* wird eine operationale Semantik für die Ausführung von Leitlinien in [68] definiert. Bei dieser Definition wird die Semantik von Bedingungen auf die Semantik von prädikatenlogischen Formeln zurückgeführt, das heißt, es wird keine Einschränkung an die Ausdrucksmächtigkeit der Bedingungen formuliert und auch kein PRO*forma* spezifisches Datenmodell spezifiziert.

Die formale Semantik wird in einer *Arezzo* genannten Entwicklungsumgebung für PRO*forma* implementiert. *Arezzo* ist eine Umgebung zur graphischen Eingabe und zur Ausführung von medizinischen Leitlinien. Damit ist die Simulation von Abläufen von Leitlinien möglich.

Gegenwärtig wird keine formale Verifikation zur Qualitätssicherung medizinischer Eigenschaften verwendet. Der Begriff der Verifikation wird im Bezug auf PRO*forma* verwendet, um den Nachweis syntaktischer Korrektheit des PRO*forma* Modells zu beschrieben.

26.1.2 EON

EON [71, 50] ist eine Modellierungssprache für medizinische Leitlinien. Der Fokus dieser Sprache liegt auf der Einsetzbarkeit für Entscheidungsunterstützungssysteme. Leitlinien, die

in EON modelliert sind, können in der Protégé 2000 Umgebung modelliert und dann ausgeführt werden.

Besonderen Wert legt EON dabei auf die praktische Verwendbarkeit der formalisierten Leitlinien. Deswegen sind die Möglichkeiten der Eingabe medizinischer Daten, also dem Äquivalent der Asbru-Krankenakte, in das System sehr ausgefeilt. Außerdem ist die Protégé 2000 Umgebung so benutzerfreundlich gestaltet, dass die Modellierung von Leitlinien von medizinischen Domänenexperten selbst durchgeführt werden kann. Wichtig bei der Gestaltung von EON ist auch die Möglichkeit, Patientenverhalten und Ontologien wiederverwendbar zu formulieren, das heißt, EON unterstützt den Aufbau einer generischen Datenbank von medizinischem Wissen bzw. Patientenmodellen.

Es gibt keine explizite formale Definition der Semantik von EON. Stattdessen wird die Semantik implizit dadurch die Implementierung der Protégé 2000 Umgebung gegeben.

26.1.3 GUIDE

GUIDE [56, 52] ist eine weitere Sprache zur Formalisierung von medizinischen Leitlinien. Das Ausführungsmodell von GUIDE setzt auf dem Modell der Petri-Netze auf, wobei diese erweitert werden, um die Verarbeitung von Patientendaten zu vereinfachen.

Besonderes Augenmerk richtet GUIDE auf die Formalisierung des medizinischen Wissens sowie einer Sichtentrennung des vorhandenen Wissens. Ziel dieser Sichtentrennung ist es, Rollen definieren zu können und für Daten innerhalb des Systems durch Annotationen bekannt zu geben, welche Rollen auf die Daten zugreifen dürfen.

Beispielsweise ist es damit möglich, Daten so zu annotieren, dass diese nur für Ärzte oder für Ärzte sowie das Pflegepersonal einsehbar sind. Damit soll erreicht werden, dass auch Patienten aus dem System autonom Daten auslesen können und dabei sichergestellt wird, dass kritische Daten nur über den Umweg des Arztes bekannt gegeben werden.

Für GUIDE existiert ein umfassendes Ausführungs- und Simulationswerkzeug, jedoch keine Ansätze zur formalen Verifikation.

26.1.4 GLARE

GLARE wird in [6, 69] vorgestellt. GLARE ist eine Sprache zur Modellierung von Leitlinien, die speziell auf die Verwendung der Leitlinie als Ausgangsmodell für Entscheidungsunterstützungssysteme abzielt. Die Sprache GLARE ist einfacher strukturiert als beispielsweise Asbru und verfolgt das Ziel, dass auch medizinische Domänenexperten in der Lage sind, ihr Wissen autonom in GLARE zu formalisieren.

GLARE kennt zur Formalisierung von Leitlinien die Konzepte der atomaren Handlungen und der komponierten Handlungen. Atomare Handlungen können nicht weiter zerlegt werden und gliedern sich in Arbeitshandlungen, Anfragehandlungen, Entscheidungen und Schlussfolgerungen. Arbeitshandlungen beschreiben die Durchführung einer medizinischen Interaktion und entsprechen damit den `user`-Plänen von Asbru. Erweitert werden die Arbeitshandlungen um Annotationen, die die Handlungen genauer beschreiben, etwa, welche Art von Ressourcen zur Ausführung der Handlungen benötigt werden, welche Kosten durch die Handlung entstehen oder ähnliches. Nachfragehandlungen beschreiben Anfragen des Systems an die Umgebung. Damit sind alle Datenanfragen an das medizinische Personal, den Patienten und auch die dem System zugrundeliegenden Datenbanken gemeint. Damit entsprechen Anfragehandlungen den `ask` Plänen in Asbru, erweitern diese jedoch, da beispielsweise in Asbru Anfragen an die Datenbank, also die Krankenakte, nicht explizit modelliert werden, da die Krankenakte in Asbru als Teil des Systems gesehen wird. Entscheidungen stellen Vergleiche zwischen voreingestellten Werten und dem Zustand des Patienten dar und Schlussfolgerungen sind die Ergebnisse solcher Vergleiche.

Komponierte Handlungen funktionieren ähnlich zu den Plänen von Asbru, die Unterpläne steuern. Eine komponierte Handlung kann eine Reihe atomarer Handlungen sequentiell oder parallel ausführen. Außerdem kann auf Basis von Entscheidungen und Schlussfolgerungen eine Auswahl atomarer Handlungen ausgeführt werden. Dazu können Wiederholungen in der Ausführung durch komplexe Pläne durchgeführt werden.

Die Handlungen werden in einen graphischen Editor eingegeben, der Abhängigkeiten zwischen den Handlungen darstellen kann. GLARE Modelle können verwendet werden um Entscheidungsunterstützungssysteme zu betreiben. Außerdem ist es möglich, idealisierte und reale Patientendaten durch Ausführung des Leitlinienmodells zu untersuchen und dabei zu evaluieren, ob ein Patient entsprechend den Vorgaben einer Leitlinie behandelt wird oder ob die Leitlinie spezielle Patienten korrekt behandeln würde.

Herausragend ist der GLARE Ansatz gegenüber den anderen Leitlinien-Modellierungssprachen in Bezug auf die Möglichkeit der Anwendung formaler Methoden. Dies wird beispielsweise in [31, 30, 70] vorgestellt. Aufsetzend auf den Ergebnissen der Verifikation von Asbru wird die Qualitätssicherung einzelner Aspekte von GLARE mit dem Modell-Prüfer SPIN [35, 72] durchgeführt.

Die Leitlinien werden dabei als Agenten modelliert. Gleichzeitig werden die anderen beteiligten Komponenten der Behandlung, also Ärzte, die temporale Datenbank sowie der Patient selbst, jeweils als weitere Agenten modelliert.

Die einzelnen Agenten werden im Folgenden in Programme der Sprache Promela übersetzt, die die Eingabesprache des Modellprüfers SPIN darstellt. SPIN ist in der Lage, Aussagen, die in der Temporallogik LTL formuliert sind, über asynchrone Systeme zu zeigen.

Laut den Publikationen der Forschergruppe können alle Aussagen verifiziert werden, die als LTL-Formeln ausdrückbar sind. Insbesonders werden dabei Aussageklassen als beweisbar genannt, die im Protocure-Ansatz als Ergebnis-Aussagen bezeichnet werden. Jedoch werden in den Publikationen weder konkrete Beispiele für diese Aussagenklasse genannt, noch Details über die verwendeten Abstraktionstechniken aufgeführt. In den Publikationen werden nur zwei konkrete Aussagen aufgeführt, die mit diesem Ansatz verifiziert worden sind. Bei diesen Aussagen handelt es sich um Prozessaussagen.

Das Datenmodell über dem dieser Ansatz arbeitet ist in gewisser Weise komplexer als das Datenmodell von Asbru. Daten in GLARE werden mit diversen Zeitstempeln versehen, die festhalten, wann ein bestimmtes Datum erfasst wurde und wie lange es gültig sein soll. Dazu kommt, dass das Ausführungsmodell von GLARE ähnlich komplex sein dürfte, wie das von Asbru, da es prinzipiell ähnlich komplexe Ausführungskonstrukte unterstützt. In diesem Zusammenhang wäre es hilfreich, Details über das Ausführungsmodell von GLARE und über die verwendeten Abstraktionstechniken auszuwerten, da die Erfahrung mit der Modellprüfung von Asbru zeigt, dass eine korrekte und effiziente Abstraktion eines Leitlinienmodells sehr schwierig automatisch zu generieren ist. Diese Informationen sind jedoch in den Publikationen nicht enthalten. Deswegen fällt eine abschließende Bewertung des Ansatzes und ein Vergleich zu dem vorliegenden Ansatz schwer.

[70] selbst grenzt sich von dem hier vorgestellten Ansatz zur automatischen Modellprüfung dadurch ab, dass der dort verwendete Modellprüfer SPIN Aussagen in LTL prüft, während hier der Modellprüfer SMV verwendet wird um Aussagen in ACTL zu beweisen.

26.1.5 GLIF

Auch GLIF [53, 75] ist eine Modellierungssprache, die das Ziel verfolgt, möglichst anwendungsorientiert zu arbeiten. GLIF gliedert Leitlinien auf in drei Teile: medizinische Entscheidungskomponenten, Handlungkomponenten und solche Entscheidungskomponenten, die nicht durch medizinisches Wissen, sondern durch ärztliche Entscheidungen gesteuert werden.

GLIF unterscheidet bei den Komponenten von Leitlinien zwischen solchen, die allgemein

die Behandlung beschreiben und solchen, die spezifisch für die Lokalisierung der Leitlinie in einem Krankenhaus verwendet werden. Ziel ist es, dass Leitlinien nur einmalig formalisiert werden und dann in der Folge in verschiedenen Institutionen eingesetzt werden können. Die lokalen Anpassungen, die sich durch unterschiedliche Ausstattungen oder Randbedingungen bei den einzelnen Institutionen ergeben können, werden ebenfalls in GLIF modelliert aber so annotiert, dass die institutionsspezifischen Komponenten stets als solche identifiziert werden können.

Für die Modellierungssprache GLIF existiert eine Ausführungskomponente. Diese wird GLEE genannt. Eine formale Semantik von GLIF wird nicht definiert, stattdessen wird eine implizite Semantik durch die Ausführung in GLEE angegeben.

Obwohl keine formalen Beweistechniken im Bezug auf GLIF eingesetzt werden, wird GLEE verwendet, um zu bewerten, in wie weit die Behandlung eines Patienten nach einer Leitlinie erfolgt ist. Dazu werden medizinische Daten des Patienten in GLEE eingegeben und die tatsächlich angeordneten Behandlungen mit den von der Leitlinie vorgeschlagenen Behandlungen abgeglichen. Bei Unterschieden kann dann evaluiert werden, ob ein Arzt hier fehlerhaft entgegen die Leitlinie gearbeitet hat oder ob möglicherweise die Leitlinie inkorrekt ist.

26.1.6 Forschergruppe um Peter Lucas

Die Forschergruppe um Peter Lucas von der Radboud Universität in Nijmegen hat sich im Rahmen eines unbenannten Ansatzes ebenfalls mit der Korrektheit medizinischer Leitlinien beschäftigt. Dieser ist in [41, 40] dokumentiert.

Der Ansatz fokussiert auf die Erstellung eines formalen Modells zur Beschreibung des Patientenverhaltens. Spezielle Sprachen zur Formalisierung der Leitlinien verwendet der Ansatz nicht. Stattdessen werden die Leitlinien mittels Temporallogik formalisiert. Damit soll der Grundstein gelegt werden, um Ergebnisaussagen zu beweisen, wie beispielsweise die Minimalinvasivität eines Eingriffs. Im Rahmen der Kooperation des Protocure-Projekts wurden Teile dieses Ansatzes übernommen und gemeinsam mit der Forschergruppe weiter entwickelt. Dies ist in Abschnitt 19 beschrieben.

26.1.7 Zusammenfassung

Es existieren eine Menge Ansätze, die Formalisierung medizinischer Leitlinien voranzutreiben. Die Meisten der Ansätze legen den Schwerpunkt auf spezielle Aspekte der Leitlinien, wie etwa die Behandlung des Datenmodells oder die Einführung von Rollenkonzepten und der Sichtentrennung bezüglich der gesammelten Daten. Praktisch alle Entwickler von Sprachen zur Formalisierung medizinischer Leitlinien stellen Möglichkeiten bereit, Leitlinienabläufe anhand von echter oder idealisierter Patientendaten zu testen.

Im Rahmen des Forschungsprojekts Protocure II war eine der Erkenntnisse, dass die Erstellung einer formalen Leitlinie ohne die Hilfe eines technischen Werkzeugs zur Ausführung und zum Test praktisch unmöglich ist. Nur mit solchen Werkzeugen ist es möglich, schon bei der Erstellung von Leitlinien die Qualität des Modells zu testen und sicherzustellen, dass das Modell in sich konsistent ist.

Den zweiten Schritt, nämlich die vollständige Verifikation von Leitlinien gehen die meisten der Gruppen bisher noch nicht. Die Ursache dürfte darin zu suchen sein, dass die Expertise dieser Gruppen in der Regel eher im Bereich der Medizin bzw. der Sprachmodellierung liegt und weniger im Bereich der formalen Methoden. Eine Ausnahme dazu macht die Gruppe um Paolo Terenziani. Diese Gruppe hat die Ergebnisse des Forschungsprojekts Protocure aufgegriffen und darauf aufsetzend eine eigene Verifikationstechnik entwickelt, die Leitlinien als agentenbasierte Systeme auffasst. Aufgrund der fehlenden Dokumentation des Ansatzes kann die Komplexität und Tragfähigkeit von außen jedoch nur unzureichend bewertet werden.

26.2 Alternative Verifikationstechniken

Die Semantik des reaktiven Asbru-Systems mit seiner Trennung zwischen den Systemschritten und den Umgebungsschritten, in denen sich der Zustand des Patienten ändern kann, sowie die Signale, die das medizinische Personal an das System sendet, kann in ITL+ ideal abgebildet werden. Die einzelnen Komponenten der Umgebung, beispielsweise das Fortschreiten der Uhrzeit, die Änderungen des Patientenmodells oder die Handlungen des Arztes, lassen sich in ITL+ jeweils als unabhängige Umgebungskomponenten spezifizieren. Damit reduziert sich die Gefahr, dass Fehler dadurch entstehen, dass mehrere Komponenten der Umgebung in einer einzelnen Systemkomponente zusammengefasst werden müssen.

Im Rahmen dieser Arbeit hätten neben anderen Planbeschreibungssprachen auch andere Möglichkeiten der Modellierung und der Verifikation eingesetzt werden können, die jedoch nicht auf die ITL+ Logik zurückgreifen können. Als Beweiswerkzeuge kommen neben KIV auch STeP [14], PVS [15] und Isabelle [51] in Frage.

Ein Vergleich zwischen KIV und den Verifikationsdiagrammen von STeP findet sich in [60]. Auf Basis dieses Vergleichs besteht die Vermutung, dass die Invariantenfindung bei STeP im Allgemeinen komplexer und fehleranfälliger ist als die durch das KIV System verwendete Invariantentechnik. Dazu kommt, dass kein uniformer Weg zum Nachweis temporallogischer Aussagen – speziell geschachtelter Aussagen – bereitsteht. Stattdessen müssen die Aussagen in Abhängigkeit des führenden temporallogischen Operators mit unterschiedlichen Verifikationsdiagrammen geführt werden.

PVS unterstützt die Aussagenspezifikation mittels CTL. Dies schließt im besonderen die Verwendung der kompositionalen ITL+ Temporallogik aus, die das dekompositionale Beweisen möglich macht. Durch die Größe der Asbru-Systeme ist fraglich, ob eine Verifikation eines Asbru-Systems technisch möglich ist, wenn keine derartige Dekompositionstechnik bereit steht.

In Isabelle ist die Temporallogik TLA implementiert. Diese verlangt das Vorliegen der Systembeschreibung in einer definierten Normalform. Diese Normalform macht die Systembeschreibung unübersichtlich und schwerer lesbar als die parallelen Programme die ITL+ zur Systembeschreibung verwendet.

Neben der Modellierung von Asbru mittels paralleler Programme wäre auch eine Implementierung durch Zustandsdiagramme (State-Charts) möglich gewesen. Die Wahl fiel auf die parallelen Programme. State-Charts wurden verworfen, da diese aus technischen Gründen das dynamische Abspalten und Terminieren von Prozessen nicht in der Form unterstützen, wie dies durch Asbru benötigt wird.

Weiterhin möglich ist der Einsatz von Modell-Prüfern. Im Rahmen dieser Arbeit wird der Einsatz des Modell-Prüfers SMV vorgestellt. Statt diesem könnte auch SPIN [35, 72] oder UPPAAL [13, 29] verwendet werden.

Die Entscheidung fiel auf SMV, da Asbru mit einer synchron-parallelen Semantik definiert wird. Diese ist nur schwer in SPIN zu integrieren, das mit einer interleaving-Semantik arbeitet. UPPAAL wurde entwickelt, um speziell mit Echtzeit-Systemen und kontinuierlicher Zeit umgehen zu können. Diese Fähigkeit wird in dieser Arbeit nicht benötigt, da Asbru ein diskretes Zeitmodell hat.

27 Diskussion und Ausblick

Diese Arbeit beschreibt einen durchgängigen Ansatz zur formalen Behandlung medizinischer Leitlinien. Für diesen Ansatz wurde zunächst die Semantik von Asbru formal festgelegt. Um die korrekte, das heißt die von den Entwicklern intendierte Semantik fassen zu können, wurden verschiedene Schritte durchgeführt. Zunächst wurde von Balser et al. ein erster Entwurf der formalen Semantik in Abstimmung mit den Asbru-Entwicklern angefertigt und in [9] vorgestellt. Diese Semantik wurde im Rahmen dieser Arbeit um die Auswertung von Bedingungen ergänzt und danach generisch implementiert. Mit dieser Implementierung konnten die bis dahin unbeachteten Randfälle der Semantikdefinition aufgedeckt und mit den Entwicklern zusammen diskutiert werden. Ergebnis davon war eine prototypische Implementierung. Aus dieser wurde dann die endgültige Fassung der formalen Semantik gewonnen und in Form von SOS-Regeln festgehalten.

Diese Fassung der formalen Semantik wurde anschließend im interaktiven Verifikationssystem KIV implementiert. Die Implementierung erfolgt durch Spezifikation paralleler Programme. Dadurch ist es möglich, bereits existierende technische Beweisunterstützung, beispielsweise durch den KIV-TL-Kalkül oder durch bereits implementierte Simplifikationsstrategien, weiter zu verwenden. Die Spezifikation in KIV erfolgt unabhängig von der Definition der formalen Semantik. Damit ist gemeint, dass die Spezifikation nicht automatisch aus den SOS-Regeln oder durch eine mechanische Übersetzung generiert wurde. Durch den anschließenden Beweis, der zeigt, dass die Semantik von Asbru korrekt im KIV-System implementiert ist, werden daher nicht nur Fehler in der Implementierung sondern auch Ungenauigkeiten in der Semantik korrigiert. Die auf diese Weise validierte Semantik ist die Grundlage für eine Vielzahl von Anwendungen, beispielsweise zur Ausführung von Asbru.

Zur Qualitätssicherung von Leitlinien wird der folgende Prozess vorgeschlagen, der stufenweise mit immer gründlicheren jedoch auch aufwendigeren Techniken vorgeht. Den ersten Schritt zur Qualitätssicherung macht der Asbru-Interpreter [74]. Der Asbru-Interpreter erlaubt eine Verwendung der Asbru-Leitlinie als rudimentäres Entscheidungsunterstützungswerkzeug. Dazu wird dem Arzt die Leitlinie in elektronischer Form, als im Interpreter eingebettetes Asbru-Modell an die Hand gegeben. Der Arzt gibt die medizinischen Daten eines Patienten in den Asbru-Interpreter, der daraufhin erlaubte Behandlungsabfolgen berechnet und dem Arzt mitteilt. Somit erhält der Arzt präzise Informationen, welche Behandlung die Leitlinie im gegebenen Fall empfiehlt. Damit kann der Arzt prüfen, ob das Modell der Leitlinie seinen Erwartungen entspricht. Diese Prüfung kann auch auf der abstrakten Ebene durchgeführt werden. Dazu werden „Musterpatienten“ und deren Krankenakten erstellt und durch den Asbru-Interpreter geprüft, ob die Leitlinie die in diesem Fall erwarteten Behandlungen vorschlägt.

Ist die durch den Asbru-Interpreter bereitgestellte Möglichkeit der Qualitätssicherung erschöpft, so bietet sich als nächstes Mittel zur Qualitätssicherung die in Kapitel 18 beschriebene Modell-Prüfung an. Bei dieser kann anhand der strukturellen Eigenschaft geprüft werden, ob allgemeine Qualitätskriterien erfüllt sind. Diese Qualitätskriterien verlangen beispielsweise die Terminierung aller Pläne nach endlicher Zeit oder auch, dass für jeden Plan mindestens ein Ablauf existiert, der den Plan startet. Mit den Ergebnissen des Modell-Prüfers können Komponenten des Asbru-Modells identifiziert werden, die möglicherweise Anomalien enthalten und daher von den Erstellern des Asbru-Modells gezielt untersucht werden sollten.

Sind die Möglichkeiten ausgeschöpft, mit dieser Technik mögliche Probleme der Leitlinie aufzuspüren, so müssen leitlinienspezifische Qualitätsaussagen definiert werden. Für diese muss dann bewiesen werden, dass die Leitlinie sie erfüllt. Es gibt vielfältige Quellen für Qualitätsaussagen über Leitlinien. Beispiele hierfür sind Qualitätsindikatoren, die von den nationalen Gesundheitsbehörden definiert werden. Anhand dieser Qualitätsindikatoren soll bemessen werden, wie gut ein durchschnittlicher Patient in einer medizinischen Einrichtung betreut wird. Einige dieser Indikatoren eignen sich zur Prüfung der Qualität einer Leitlinie. Eine weitere Quelle für Aussagen sind Qualitätskriterien, die die Leitlinienentwickler selbst oder Entwickler ähnlicher Leitlinien definiert haben. Aufgrund der Heterogenität der Quellen der Aussagen wird ein komplexer Prozess zur Vereinheitlichung und Formalisierung der Aussagen verwendet, der in Abschnitt 14 vorgestellt wird.

Die formalisierten Aussagen werden dann als Beweisverpflichtung für die interaktive Verifikation verwendet. Diese Beweisverpflichtungen werden mit Hilfe der Beweistechniken der symbolischen Ausführung und der Beweisdekomposition verifiziert, die in den Abschnitten 15 bzw. 16 vorgestellt werden. Zeigt sich während des Beweisversuchs, dass die Aussage nicht korrekt ist (oder besteht die Vermutung der Inkorrektheit der Aussage bereits von Anfang an), so kann die Technik zum Nachweis von Gegenbeispielen aus Abschnitt 17 verwendet werden.

Die vorgestellten Beweistechniken wurden zur Verifikation einer realen Leitlinie verwendet und die Anwendbarkeit damit unter Beweis gestellt. Während des Protocure-Projekts wurden die Beweistechniken der interaktiven Verifikation für einen Versuch auch Projektpartnern vorgestellt, die ihren Forschungsschwerpunkt nicht im Bereich der formalen Methoden haben. Aus den Ergebnissen dieses Versuchs ergibt sich, dass die Technik der symbolischen Ausführung als intuitiv und handhabbar wahrgenommen wird. Insbesonders die Präsentation des Ergebnisses wird als sehr hilfreich zum Auffinden von Anomalien betrachtet [61]. Die Technik der Beweisdekomposition wurde in diesen Versuchen als komplex angesehen, da die kreative Findung von Dekompositionsaussagen zu schwierig ist. Seit diesem Versuch wurden Schritte unternommen, um bessere Handreichungen für die Generierung von Dekompositionsaussagen geben zu können. Ein Ziel für die Zukunft sollte sein, die Generierung von Dekompositionsaussagen stärker zu automatisieren oder zumindest zu systematisieren. Denkbar sind beispielsweise Automatismen, die den Status eines Plans beim Benutzer erfragen und die Dekompositionsaussagen dann zumindest teilweise automatisiert erstellen. Insgesamt lag der Aufwand für die interaktive Verifikation der Brustkrebs Fallstudie bei etwa vier Personenmonaten. Da während des Beweisens großer Wert auf eine Wiederverwendbarkeit der Regeln zur Steigerung der Automatisierung gelegt wurde, ist zu erwarten, dass der Beweisaufwand für eine weitere Fallstudie sich auch in diesem zeitlichen Rahmen bewegt.

Die Verbindung aller vorgestellter Techniken, also der Validierung durch Simulation, der Modellprüfung und der interaktiven Verifikation, ist nur auf Basis der gemeinsamen formalen Semantik möglich. Nur so ist sichergestellt, dass die unterschiedlichen Techniken bei Anwendung auf das gleiche Problem zu identischen Ergebnissen kommen. Mit dieser Ergebnisgleichheit ist es möglich, den Grad der Komplexität der Qualitätssicherung – und damit den Aufwand – frei zu wählen. Der Aufwand, der investiert wird, kann davon abhängig gemacht werden, wie kritisch die Leitlinie ist. In einem nächsten Schritt ist es denkbar, in einer Leitlinie Teilkomponenten zu identifizieren, die den sicherheitskritischen Teil der Leitlinie darstellen und dann nur auf diesen die Verifikationstechniken anzuwenden, wohingegen der Rest der Leitlinie nicht oder mit schwächeren Mitteln untersucht wird.

Weiterhin kann in Zukunft die Interaktion der Techniken miteinander untersucht werden. So ist es denkbar, einfache Beweisverpflichtungen, die durch einen interaktiven Beweis entstehen, durch einen automatischen Modellprüfer verifizieren zu lassen. Dazu ist es allerdings erforderlich, die Abstraktionen zwischen dem Asbru-Modell und dem Eingabemodell des Modell-Prüfers genauer zu untersuchen. Speziell die Aufsplittung der Asbru-Zustandsvariablen muss

dabei betrachtet werden.

Ein weiterer zu beachtender Punkt für die Zukunft ist die eigentliche Erstellung des Asbru-Modells. Die Semantik von Asbru ist zum Teil recht komplex. Die Brustkrebs Fallstudie hat gezeigt, dass eine genaue Einweisung der Modellierer in die formale Semantik unerlässlich ist, um qualitativ hochwertige Modelle zu generieren. Auch müssen Wege gefunden werden, wie die Qualitätssicherungstechniken tatsächlich bei Erstellung des Modells genutzt werden, also beispielsweise ein in Erstellung befindliches Modell auf die Erfüllung der strukturellen Eigenschaften hin untersucht wird. Dazu sollten die Techniken der Modell-Prüfung und des Asbru-Interpreters in die Asbru-Modellierungswerkzeuge wie etwa DELT/A [73] eingebaut werden, damit die Asbru-Modellierer diese möglichst einfach nutzen können.

Das Ziel dieser Anstrengungen ist die vollständige Begleitung der Entwicklung einer Leitlinie. Dabei sollen die Entwickler der Leitlinie nicht erst ein informelles Dokument erstellen, das dann im Anschluss formalisiert wird. Stattdessen wird die Leitlinie gleich in formaler Form erzeugt. Die Leitlinienersteller verwenden den Asbru-Interpreter um bereits bei der Erstellung grobe Fehler im Modell zu identifizieren und zu beheben. Das resultierende Modell wird dann von Spezialisten für formale Methoden mit der automatischen Modellprüfung und der interaktiven Verifikation untersucht. Erst wenn die Beteiligten mit dem Ergebnis zufrieden sind, wird aus dem formalen Asbru-Modell eine informelle, fehlerfreie Leitlinie generiert, die dann den Ärzten zur Verfügung gestellt wird.

Teil VII

Anhang

A Implementierung der statischen Asbru-Komponenten

Dieser Abschnitt beschreibt, wie die Datentypen der Syntax von Asbru sowie des Asbru-Zustands in KIV implementiert werden. Dazu wird zunächst in Abschnitt A.1 beschrieben, wie die Spezifikationen von KIV zu lesen und zu verstehen sind. Daran anschließend gibt Abschnitt A.3 einen Überblick über die Einbettung der Basisdatentypen. Anschließend daran wird in Kapitel A.4 die Einbettung der Datentypen des Asbru-Zustands beschrieben.

A.1 Spezifikationen in KIV

A.1.1 Datenspezifikationen

Datenspezifikationen in KIV beschreiben erzeugte Datentypen. Die Syntax dieser Spezifikationen soll anhand des Beispiels der Listen von Zeichenketten beschrieben werden:

```
data specification
using string

  string-list = [] with emptyP
              | . + . (.first : string; .rest : string-list);


end data specification
```

Das Schlüsselwort **data specification** leitet eine Datenspezifikation ein. Das darauf folgende, optionale Schlüsselwort **using** gibt an, welche Spezifikation(en) – und damit welche Datentypen – in diese Spezifikation eingebunden werden sollen. Im Beispiel wird die Definition der Zeichenketten `string` eingebunden, die in der Spezifikation string spezifiziert sind. Durch die Einbindung mittels **using** werden neben den Datentypen auch alle Regeln zur Simplifikation mit eingebunden und können in Beweisen verwendet werden.

Im Anschluss an die Deklaration der verwendeten Spezifikationen erfolgt die Definition des neuen Datentypen, in diesem Fall des Datentyps `string-list`. Für diesen sind zwei Konstruktoren definiert. Der eine Konstruktor ist die leere Liste [], der andere Konstruktor ist der Anfügeoperator +. Die Punkte um einen Operator geben dessen Stellung an. Die Notation . + . bedeutet, dass es sich um einen Infix Operator handelt. Analog dazu wäre „. fun“ ein Postfix-Operator und „fun .“ ein Präfix Operator. Die Standard-Stellung der Operatoren ist die Präfix-Stellung, sie muss daher nicht gesondert angegeben werden. In Klammern werden die Typisierung des Operators angegeben und gleichzeitig Selektoren für die Operanden definiert. Im Beispiel ist der erste Operand des + eine Zeichenkette, der Zweite eine Liste von Zeichenketten. Auf den ersten Operanden wird mittels des Selektors .first zugegriffen, auf den zweiten Operator mittels .rest.

Da es sich um eine Datenspezifikation handelt, werden automatisch Axiome generiert. Diese beschreiben die hier definierten Listen als frei erzeugte Datentypen. Das bedeutet, zwei Listen sind gleich, genau dann, wenn der Aufbau der Listen gleich ist, also:

- $\text{str} + \text{strx} = \text{str1} + \text{strx1} \leftrightarrow \text{str} = \text{str1} \wedge \text{strx} = \text{strx1}$

- ¬ str + strx = []
- [] = []

Zwei Listen sind also gleich, wenn das zuletzt angefügte Element sowie die Restlisten jeweils gleich sind. Weiterhin werden automatisch Axiome generiert, die die Semantik der Selektoren festlegen.

- (str + strx) .first = str
- (str + strx) .rest = strx

Diese Axiome legen fest, dass .first das zuletzt angefügte Element selektiert und .rest die entsprechende Restliste ohne das zuletzt angefügte Element.

Das Schlüsselwort **with** in Zusammenhang mit einer Konstruktordefinition erlaubt die Spezifikation eines Prädikats. Dieses prüft, ob der führende Operator eines Terms, der ein string-list Datum beschreibt, der betreffende Konstruktor ist. Im Beispiel heißt ein solches Prädikat emptyP, das auf den Konstruktor der leeren Liste prüft. Für dieses Prädikat werden automatisch Axiome für alle Konstruktoren generiert:

- ¬ emptyP(str + strx)
- emptyP([])

Durch Angabe weiterer Schlüsselworte können noch mehr Axiome automatisch generiert werden. Beispielsweise können automatisch Längenfunktionen sowie Ordnungsprädikate erzeugt werden. Dies wird in dieser Arbeit nicht verwendet und daher hier auch nicht vorgestellt.

Obligatorisch bei der Spezifikation von Datentypen ist die Angabe von Variablen. Aus Gründen der Übersichtlichkeit und Lesbarkeit werden diese in diesem Abschnitt ausgelassen.

Eine Datenspezifikation wird mit dem Schlüssenwort **end data specification** beendet.

A.1.2 Aktualisierungen

Aktualisierungen erlauben es, parametrisierte Spezifikationen zu instantiieren. Dabei wird der Parameter einer Spezifikation ersetzt durch einen konkreten Datentyp. Beispielsweise könnten Listen von Zeichenketten auch durch Instantiierung der generischen Listen erzeugt werden:

```
actualize list with string-less by morphism

list → string-list;

elem → string;

end actualize
```

Das Schlüsselwort **actualize** gibt an, dass es sich um eine Aktualisierungs-Spezifikation handelt. Nach diesem Schlüsselwort muss als nächstes der Name der Spezifikation angegeben werden, in der der Datentyp spezifiziert ist, der instantiiert werden soll, im Beispiel die generischen Listen. Das Schlüsselwort **with** trennt den Namen der generischen Spezifikation von den Namen der Spezifikation(en), in denen die Datentypen spezifiziert werden, die die generischen Parameter ersetzten sollen. Im Beispiel wird also ein Datentyp aus der `list` Spezifikation aktualisiert. Die Parameter der allgemeinen Listen werden durch Datentypen aus der `string-less` Spezifikation ersetzt. Nach dem Schlüsselwort **by morphism** erfolgt die Angabe der Aktualisierungsfunktion. Diese benennt im Beispiel die Listen `list` um zu `string-list`. Dadurch wird ein neuer Datentyp `string-list` erzeugt, dessen Semantik

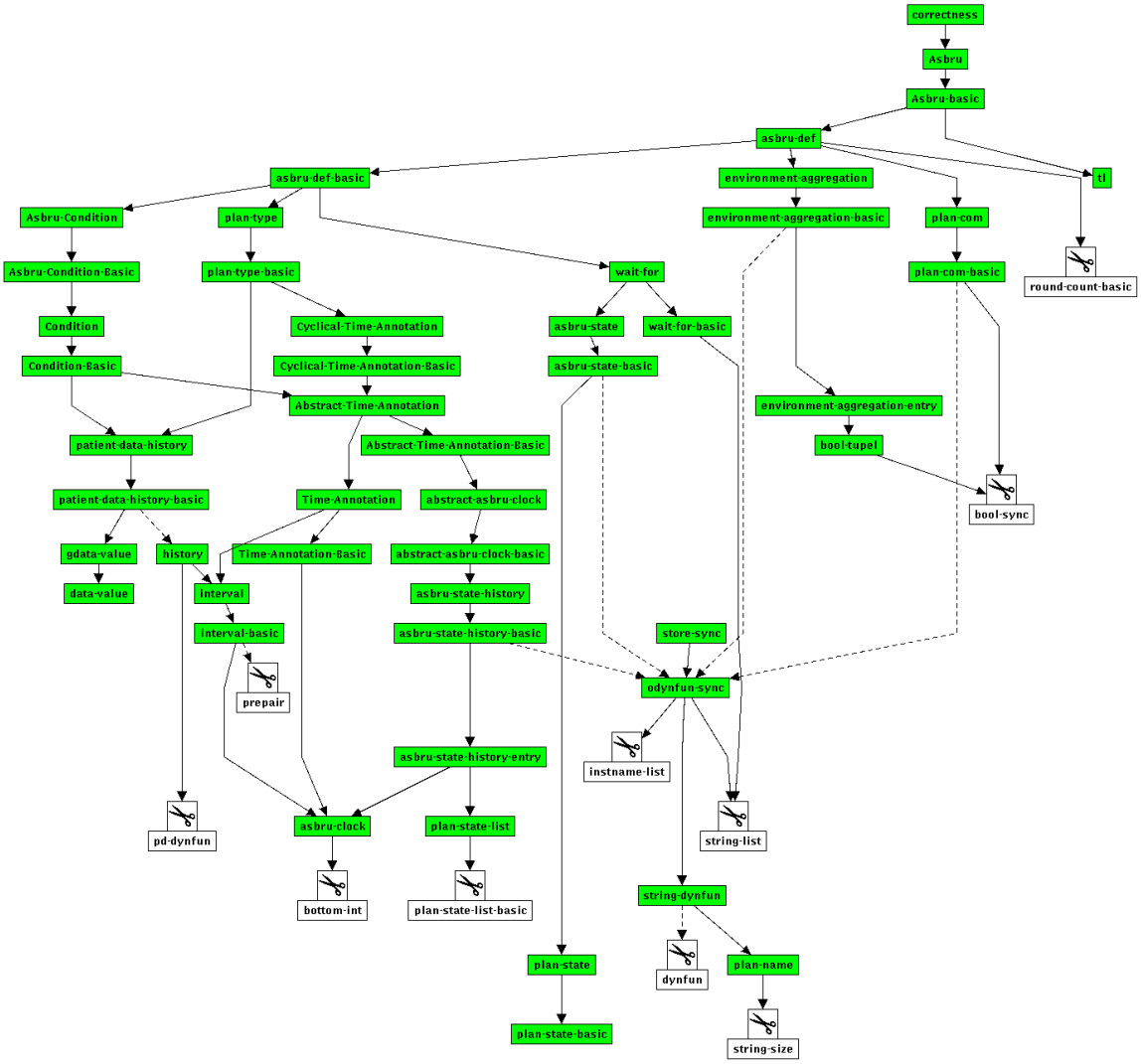


Abbildung A.1: Spezifikationsgraph von Asbru

durch die gleichen Axiome spezifiziert ist, wie die der allgemeinen Listen. Der Parametertyp der Listen, der `elem` benannt ist, wird durch die Aktualisierungsfunktion abgebildet auf den Datentypen `string`. Das Ergebnis dieser Aktualisierung ist ein Datentyp `string-list`, der Listen von Zeichenketten speichert.

Mit dem Schlüsselwort **end actualize** endet eine Aktualisierungs-Spezifikation.

A.2 Spezifikationsgraph von Asbru

Abbildung A.1 zeigt den Spezifikationsgraphen der Asbru-Implementierung in KIV. Von oben nach unten betrachtet, werden hier folgende Komponenten von Asbru spezifiziert. In der Spezifikation `Asbru` wird das dynamische Verhalten von Asbru spezifiziert, das in Abschnitt 12 beschrieben wird. Die darunterliegenden Spezifikationen `asbru-def` bzw. `asbru-def-basic` umfassen die Spezifikation des **asbru-plan** Datentyps sowie aller Prädikate und Funktionen, die zur Formulierung

der SOS-Regeln notwendig sind und die in Abschnitt C vorgestellt werden. Die Spezifikationen `Asbru-Condition`, `Asbru-Condition-Basic`, `Condition` sowie `Condition-Basic` beschreiben die Definition der Syntax der Bedingungen in Asbru sowie die Semantik der Auswertung dieser Bedingungen. In den Spezifikationen `plan-type` und `plan-type-basic` werden die Plankontrolltypen spezifiziert, die Spezifikationen `Abstract-Time-Annotation`, `Abstract-Time-Annotation-Basic`, `Time-Annotation` sowie `Time-Annotation-Basic` spezifizieren die Syntax und Semantik der Zeit-Annotationen. Dazu notwendig ist die Syntax- und Semantik-Definition der Abstrakten Referenzpunkte, die sich in den Spezifikationen `abstract-asbru-clock` und `abstract-asbru-clock-basic` findet. Die im Zusammenhang mit zyklischen Plänen verwendeten zyklischen Zeit-Annotationen werden in den Spezifikationen `Cyclical-Time-Annotation` sowie `Cyclical-Time-Annotation-Basic` definiert. Die Wartebedingungen von Asbru-Plänen werden in den Spezifikationen `wait-for` sowie `wait-for-basic` definiert. Die Planzustände die ein Asbru-Plan durchlaufen kann werden in der Spezifikation `plan-state` bzw. `plan-state-basic` definiert.

Zu den statischen Komponenten von Asbru kommen die Komponenten des Asbru-Zustands hinzu. Nahezu alle Datentypen, die den Zustand von Asbru ausmachen, sind als dynamische Funktionen spezifiziert. Diese dynamischen Funktionen sind in den Spezifikationen `dynfun`, `string-dynfun` und `odynfun-sync` spezifiziert. Der Datentyp dieser dynamischen Funktionen heißt `store` und bildet Zeichenketten ab auf einen unspezifizierten Parameter. Die verschiedenen Zustandsdatentypen aktualisieren den `store` Datentyp, so dass er nicht mehr auf generische Werte abbildet. Beispielsweise ist der Asbru-Zustand, der in den Spezifikationen `asbru-state` und `asbru-state-basic` definiert ist, eine Aktualisierung des `store` Datentyps mit Planzuständen. Die Umgebungssignalsammlung ist ebenso eine Aktualisierung des `store` Datentypen, die in den Spezifikationen `environment-aggregation` und `environment-aggregation-basic` spezifiziert werden. Die Einträge werden durch die Spezifikationen `bool-tupel` und `environment-aggregation-entry` festgelegt. Der Datentyp, der die Aktivierungssignale der Pläne speichert wird in den Spezifikationen `plan-com` und `plan-com-basic` spezifiziert und aktualisiert den `store` Datentypen durch boolesche Werte. Analog dazu enthält die Spezifikation `round-count-basic` eine Definition des **round-state** Datentyps als Aktualisierung der dynamischen Funktionen mit natürlichen Zahlen **nat**. Die Krankenakte und die dazu gehörigen Krankenakten Einträge werden in den Spezifikationen `patient-data-history` und `patient-data-history-basic` spezifiziert, die dazu gehörigen generischen Parameter in den Spezifikationen `data-value` und `gdata-value`. Krankenakten sind verschachtelte Funktionen und bilden eine Uhrzeit und einen Schlüssel auf einen Wert ab. Dies wird implementiert durch zwei verschachtelte dynamische Funktionen. Die erste bildet Schlüssel auf Werte ab, die zweite Uhrzeiten auf Funktionen, die Schlüssel auf Werte abbilden. Letztere Funktion wird in der Spezifikation `history` spezifiziert. Die Asbru-Zustandshistorie sowie deren Einträge und die Planzustandslisten werden in den Spezifikationen `asbru-state-history`, `asbru-state-history-basic`, `asbru-state-history-entry`, `plan-state-list` und `plan-state-list-basic` spezifiziert. Die Spezifikation `asbru-clock` erweitert die in der Spezifikation `bottom-int` sowie `bottom-int-basic` spezifizierten ganzen Zahlen mit dem $\perp$ Element um entsprechende Variablen und Simplifikationsregeln.

A.3 Basisdatentypen

A.3.1 Asbru-Plan Datenstruktur (asbru-plan)

Die Asbru-Plan Datenstruktur definiert die statische Sicht auf einen Plan, d.h. welche Unterpläne ein Plan hat, wie diese kontrolliert werden oder welche Bedingungen an die Ausführung des Plans geknüpft sind.

Die Datenstruktur **asbru-plan** der Semantik wird implementiert durch den Datentyp `asbru-def`. In den einzelnen Feldern dieses Typs werden jeweils die Datentypen gespeichert, die den Datentypen der Semantik entsprechen So wird der **asbru-condition**-Typ der Semantik implementiert durch den Datentyp `asbru-condition`, der Plantyp **plan-type** wird implementiert durch den Datentyp `plan-type`, die booleschen Typen **bool** der Semantik werden in der Implementierung durch die `bool` Werte ersetzt. Letztere werden in der KIV-Ebene durch die Bibliothek bereitgestellt. Die Liste der Unterpläne wird in der KIV-Ebene durch eine Liste von Zeichenketten `string-list` implementiert, da Plannamen durch Zeichenketten implementiert werden. Für die Implementierung des **wait-for** der Semantik wird das `wait-for` spezifiziert. Die Struktur von `asbru-def` entspricht damit im Wesentlichen der Struktur von **asbru-plan**, mit dem Unterschied, dass die Reihenfolge der Datenfelder geringfügig geändert ist.

```
data specification
using plan-type, Asbru-Condition, wait-for

    asbru-def
  = mk-asbru-def( . .filter        : asbru-condition;
                  . .setup         : asbru-condition;
                  . .suspend       : asbru-condition;
                  . .reactivate    : asbru-condition;
                  . .abort         : asbru-condition;
                  . .complete      : asbru-condition;
                  . .type          : plan-type;
                  . .retry-aborted : bool;
                  . .subplans      : string-list;
                  . .waitfor       : wait-for;
                  . .opt-wf        : bool;
                );

end data specification
```

Wie in der Semantik enthält ein `asbru-def` sechs Bedingungen, die über Selektoren **.filter**, **.setup**, **.suspend**, **.reactivate**, **.abort** und **.complete** selektiert werden können. Der Plantyp des Plans kann mittels des Selektors **.type** selektiert werden, das Flag, das angibt, ob Unterpläne neu gestartet werden sollen mit dem Selektor **.retry-aborted**, die Unterpläne mit dem Selektor **.subplans**, die Wartebedingung mit dem Selektor **.wait-for** und die Wartebedingung für optionale Pläne mit **.wait-for-optional**.

A.3.2 Plan-Typen Datenstruktur (plan-type)

Ein Plantyp beschreibt, wie ein Plan seine Unterpläne steuert, d.h. wann die Unterpläne gestartet und wann sie aktiviert werden sollen.

Die Plantypen **plan-type** der Semantik werden auf KIV Ebene durch die `plan-type` Datentypen implementiert. Dabei sind die einfachen Plantypen wie beispielsweise `sequential` analog zur den **plan-type** Datentypen implementiert.

In KIV sind die Konstruktoren von Datentypen parametrisierbar. Daher entfällt hier die Notwendigkeit, getrennte Spezifikationen für die `ifThenElse`-, `ask`- und `cyclical`-Kontrolltypen anzugeben. Stattdessen können die Parameter dieses Datentyps direkt angegeben werden. Dabei wird der **conditional**-Parameter der `ifThenElse`

Kontrolle der Semantik direkt durch eine Funktion implementiert, die eine Krankenakte und eine Uhrzeit auf einen Wahrheitswert abbildet. Dies entspricht der Definition des **conditional** Datentyps, wie er in Kapitel 8.1.6 angegeben wird.

```
data specification

using Cyclical-Time-Annotation, patient-data-history, variable-history

plan-type = user
          | sequential
          | parallel
          | anyorder
          | unordered
          | onabort
          | assignment
          | onabortonsuspend
          | ifthenelse(
              . .conditional: patient-data-history × int → bool) with iteP
          | ask(. .key : string) with askP
          | cyclical(. .cta: cyclical-time-annotation; . .rounds : nat) with cycP;

end data specification
```

Zyklische Plankontrollen sollen Unterpläne wiederholt ausführen. Dazu wird ein wiederkehrendes Startintervall spezifiziert. In den verschiedenen Wiederholungen dieses Startintervalls soll der Unterplan gestartet werden.

Der Typ der zyklischen Plankontrolle wird in KIV geringfügig anders spezifiziert, als dies in der Semantik geschieht. In der Semantik wird der Plantyp parametrisiert mit einer abstrakten-Zeit-Annotation sowie den drei Ganzzahl-Werten die Offset, Frequenz und die Zahl der notwendigen Wiederholungen beschreiben. In der Implementierung wird stattdessen der zyklische Typ mit einer zyklischen Zeitannotation sowie der Zahl der erforderlichen Runden parametrisiert. Die zyklische-Zeitannotation ist wie folgt spezifiziert:

```
data specification using Abstract-Time-Annotation

   cyclical-time-annotation
 = mk-cta
     ( . .ata : abstract-time-annotation;
       . .offset    : int;
       . .frequency : nat
     );

end data specification
```

Das heißt, in der KIV Spezifikation werden alle Parameter, die zur Spezifikation des Startintervalls notwendig sind in der zyklischen Zeitannotation gekapselt.

A.3.3 Abstrakte Zeit-Annotationen (abstract-time-annotation)

Abstrakte-Zeitannotationen beschreiben zeitliche Aspekte von Bedingungen. Sie werden immer dann verwendet, wenn eine Bedingung wie „Patient hat Fieber“ in zeitlichen Bezug gesetzt werden muss, die Aussage als beispielsweise lautet „der Patient Fieber über einen Zeitraum von drei Tagen“.

Abstrakte-Zeitannotationen werden in KIV analog zu den **abstract-time-annotation** Datenstrukturen der Semantik definiert.

```
data specification
using abstract-asbru-clock

    abstract-time-annotation
  =   mk-ata(. .ess : bottom-int;
```

```
            . .lss : bottom-int;
            . .efs : bottom-int;
            . .lfs : bottom-int;
            . .minDuration : int;
            . .maxDuration : bottom-int;
            . .referencePoint : abstract-asbru-clock)
    | *now_ata*;


end data specification
```

Auch die Selektoren der einzelnen Datenfelder sind gleich benannt und typisiert wie in der Semantik.

A.3.4 Plannamens-Listen (plan-list)

Plannamenslisten werden verwendet, um die Liste der Unterpläne eines Plans zu spezifizieren.

Wie bereits in Abschnitt A.1 beschrieben, werden Aktualisierungs-Spezifikationen verwendet, um parametrisierte Datentypen zu instantiieren. Im vorliegenden Fall werden in allgemeinen Listen die allgemeinen `elem` Elemente durch Zeichenketten ersetzt. Die Spezifikation der Listen kommt dabei aus der Spezifikations-Bibliothek von KIV. Diese spezifiziert die Listen mit der Standardsemantik.

```
actualize list with string-less by morphism

list → string-list;

elem → string;

end actualize
```

A.3.5 Plannamen (plan-name)

Plannamen dienen als eindeutiger Schlüssel, um auf die **asbru-plan** Plandefinition eines Planes zugreifen zu können.

Die Plannamen der Spezifikation werden in KIV durch einfache Zeichenketten implementiert. Diese Zeichenketten werden in der Basis-Bibliothek mit der Standard-Semantik spezifiziert.

A.3.6 Planinstanznamen (instance-name)

Planinstanznamen identifizieren einzelne Planinstanzen. Aus dem Planinstanznamen soll der Planname sowie der Instanzname des übergeordneten Plans auslesbar sein. Planinstanznamen werden als erzeugter Datentyp auf Basis der Zeichenketten spezifiziert.

```
enrich string-size with

   sorts
      instance-name;
   functions
      nameGen : string × instance-name → instance-name;
      superGen : string → instance-name;
      parSel : instance-name → instance-name;
      nameSel : instance-name → string;
   variables
      iname, iname0, iname1 : instance-name;
      IName, IName0, IName1 : instance-name flexible;
   induction
      instance-name generated by namegen, supergen;
```

```
   axioms
      parSel :  ⊢ parSel(nameGen(str, iname)) = iname;
      used for : s, ls;

      nameSel :  ⊢ nameSel(nameGen(str, iname)) = str;
      used for : s, ls;

      nameSel :  ⊢ nameSel(superGen(str)) = str;
      used for : s, ls;

end enrich
```

Der Datentyp der Planinstanznamen wird erzeugt durch die Konstruktoren **superGen** und **nameGen**. Der erste Konstruktor erzeugt den Instanznamen eines Top-Level Plans, der keinen übergeordneten Plan hat. Der zweite Konstruktor wird verwendet, um den Instanznamen eines normalen Plans aus dem Instanznamen seines übergeordneten Plans sowie seinem eigenen Plannamen zu erzeugen.

Die Funktion **parSel** selektiert aus einem Instanznamen den Planinstanznamen des übergeordneten Plans. Diese Funktion ist nur spezifiziert, wenn der betrachtete Plan kein Top-Level Plan ist. Die Funktion **nameSel** selektiert aus einem beliebigen Instanznamen den Plannamen der betrachteten Planinstanz.

Das Induktionsprinzip „generated by“ weist den Instanznamen als einen erzeugten Datentyp aus, so dass alle Planinstanznamen entweder durch den Konstruktor **nameGen** oder den Konstruktor **superGen** erzeugt werden.

A.3.7 Asbru-Bedingungen (asbru-condition)

Asbru-Bedingungen beschreiben Forderungen, die an bestimmte Zustandsübergänge geknüpft sind. Beispielsweise darf ein Asbru-Plan vom Zustand `considered` nur dann in den Zustand `possible` wechseln, wenn die **filter**-Bedingung (eine Asbru-Bedingung) erfüllt ist. Asbru-Bedingungen werden mit den Prädikaten **satisfied** und **satisfiable** ausgewertet. Diese Prädikate bestimmen, ob eine Bedingung erfüllt, erfüllbar oder nicht mehr erfüllbar ist. Asbru-Bedingungen erweitern Bedingungen um die Möglichkeit der Umgebung, Einfluss auf die Auswertung zu nehmen.

Asbru-Bedingungen werden in KIV analog zu den Asbru-Bedingungen der Semantik spezifiziert.

```
data specification
using Condition

    asbru-condition
  = mk-asbru-cond(. .condition : condition;
                  . .manual : bool;
                  . .override : bool);

end data specification
```

A.3.8 Bedingungen (condition)

Bedingungen in Asbru enthalten Zustandsbeschreibungen, die auf einen aktuellen Zustand zutreffen müssen, damit ein Plan bestimmte Zustandsübergangsregeln anwenden kann. Diese Beschreibungen können Zeitbezug enthalten.

In KIV ist neben der Abstrakten-Zeit-Annotation auch eine nicht-abstrakte Zeit-Annotation spezifiziert. Diese enthält als Uhrzeit lediglich eine reguläre Asbru-Uhr statt der abstrakten-Asbru-Uhr. Dies hängt mit der unterschiedlichen Herangehensweise bei der Auswertung von **satisfied** und **satisfiable** in KIV und Semantik zusammen. Während

in der Semantik überprüft wird, ob ein gegebenes Intervall Element einer abstrakten Zeitannotation ist, ist dies in KIV nicht möglich. Stattdessen wird in KIV zunächst die abstrakte Uhrzeit der Bedingung in eine konkrete Uhrzeit umgerechnet. Die Auswertung der Bedingung erfolgt dann auf Basis dieser nicht-abstrakten Zeit-Annotation. Diese beiden Schritte, die Auswertung der abstrakten Asbru-Uhr sowie die Berechnung der gültigen Intervalle wird auf Semantik-Seite zusammengefasst. Dies ist auf der KIV Seite aus Effizienzgründen nicht möglich. Abgesehen davon sind die Bedingungen in KIV und der Semantik identisch spezifiziert.

```
data specification
using Abstract-Time-Annotation, patient-data-history

     condition
  =    mk-cond  (. .ccf : (patient-data-history × int → bool)) with baseP
     | mk-cond  (. .ccf : (patient-data-history × int → bool);
                 . .ta  : time-annotation) with taP
     | mk-cond  (. .ccf : (patient-data-history × int → bool);
                 . .ata : abstract-time-annotation) with ataP
     | . _and . (. .fst : condition; . .snd : condition) with andP
     | . _or .  (. .fst : condition; . .snd : condition) with orP;

end data specification
```

A.3.9 Konditionale (conditional)

Ein Konditional beschreibt eine Bedingung ohne zeitlichen Bezug, also beispielsweise „der Patient hat Fieber“. In KIV gibt es keinen expliziten Datentyp, der dem Konditional der Semantik entspricht. Stattdessen wird jeweils in den Spezifikationen der Bedingungen und der Plantypen lokal ein Prädikat definiert, das eine Krankenakte und eine Uhrzeit auf einen Wahrheitswert abbilden und so das Konditional definieren.

A.3.10 Der Zahlentyp int-bottom (int-bottom)

Der Datentyp int-bottom beschreibt die Menge der ganzen Zahlen ergänzt um ein zusätzliches Element $\perp$. Dieses beschreibt unspezifizierte zeitliche Aspekte und kann in der Regel interpretiert werden als ein Platzhalter für plus- bzw. minus-Unendlich.

Implementiert wird dieser Datentyp analog zu der Definition in der Asbru-Semantik. Aus technischen Gründen muss für die ganzen Zahlen der Konstruktor num verwendet werden. Ein int-bottom, das den Wert 5 speichern soll wird also in KIV geschrieben als num(5).

```
data specification
using int-min, intnat

bottom-int =   num(. .val: int)
             | ⊥;


end data specification
```

A.3.11 Abstrakte Referenzpunkte (abstract-asbru-clock)

Abstrakte Referenzpunkte können auf statischer Ebene die Zeitpunkte von Planzustandswechseln beschreiben. Anhand eines Zustands können diese statischen Konstrukte dann zu konkreten Uhrzeiten umgerechnet werden. Daneben können abstrakte Referenzpunkte auf statischer Ebene so spezifiziert werden, dass sie stets zur jeweils aktuellen Uhrzeit ausgewertet werden. Abstrakte Referenzpunkte werden als Referenzpunkte von abstrakten Zeit-Annotationen verwendet.

Abstrakte Referenzpunkte werden in KIV identisch zu der Semantikdefinition spezifiziert.

```
data specification
using asbru-state-history

    abstract-asbru-clock
  =   clock(. .val : bottom-int)
    | enter(. .key : string; . .pstate : plan-state)
    | leave(. .key : string; . .pstate : plan-state)
    | *now*;

end data specification
```

A.3.12 Planzustände (plan-status)

Planzustände beschreiben die Stationen des Ablaufs von Asbru-Plänen. Die Übergangsregeln beschreiben, in welcher Sequenz diese Stationen angelaufen werden und welche Bedingungen für welche Übergänge erfüllt werden müssen.

Die Planzustände sind in KIV so spezifiziert, wie in der Semantik.

```
data specification

plan-state =   inactive
             | considered
             | possible
             | ready
             | activated
             | suspended
             | completed
             | rejected
             | aborted;

end data specification
```

A.3.13 Wartebedingungen (wait-for)

Wartebedingungen erlauben einem Plan, Unterpläne zu benennen, die in jedem Fall erfolgreich abgeschlossen sein müssen, bevor der Plan seine Bearbeitung abschließen kann.

Die Definition der Wartebedingungen in KIV erfolgt so wie in der Semantik vorgegeben.

```
data specification
using stringlist


   wait-for =   bwf(. .content : string)
              | . _and . (. .first  : wait-for;
                          . .second : wait-for)
              | . _xor . (. .first  : wait-for;
                          . .second : wait-for)
              | . _or . (. .first  : wait-for;
                         . .second : wait-for)
              | _not .(. .first  : wait-for)
              | wait-for-n(. .number : nat; . .watched : string-list)
              | wait-for-none;


end data specification
```

A.4 Zustandsdatentypen

Der Zustand der Asbru Semantik besteht aus sechs verschiedenen Variablen, jeweils einer vom Typ **int**, **environment-aggregation**, **hierarchy-state**, **asbru-state-history**, **bool** und **patient-data-history**. Aus Effizienzgründen können diese zum Teil nicht genau so implementiert werden, wie sie spezifiziert wurden. Daher erfolgt in Kapitel 13 ein Nachweis der Korrektheit der Implementierung.

A.4.1 Uhrzeiten (int)

Die Uhrzeit dient um Zustände ordnen zu können. Weiterhin ist die Uhrzeit wichtig, um in den Historienvariablen die korrekten Einträge zu selektieren. Die Uhrzeit in Asbru ist abstrakt gehalten und als dimensionsloser Zähler der Makro-Schritte implementiert. Damit ist eine beliebig feine Granularität möglich, indem eine entsprechende Zuordnung der Dauer von Makro-Schritten zu realer Zeit erfolgt.

Wie in der Semantik-Definition ist die Uhrzeit in KIV als Ganzzahl-Wert angegeben. Dabei stammt die Spezifikation der ganzen Zahlen aus der KIV-Bibliothek. Die ganzen Zahlen sind dort mit der Standard-Semantik implementiert.

A.4.2 Booltupel (bool-tupel)

Bool-Tupel sind Datenstrukturen, die sechs boolesche Werte speichern. Diese sechs Werte entsprechen entweder sechs Signalen zur Übersteuerung oder zur Verzögerung der sechs verschiedenen Asbru-Bedingungen der Asbru-Pläne. Zwei Booltupel Datenstrukturen speichern alle relevanten Informationen im Bezug auf die Signale, die das medizinische Personal dem Asbru-System sendet.

In KIV sind die Bool-Tupel genau so implementiert, wie sie auf Semantik Ebene spezifiziert sind.

```
data specification
using bool-sync


  bool-tupel =  mk-bt ( . .filter : bool;
                        . .setup : bool;
                        . .suspend : bool;
                        . .reactivate : bool;
                        . .abort : bool;
                        . .complete : bool);



end data specification
```

A.4.3 Booltupel-Paare (environment-aggregation-entry)

Ein Eintrag in die Umgebungssignalsammlung kapselt alle Signale, die von der Umgebung an einen Plan gesendet werden können. Dabei handelt es sich um insgesamt 13 Signale. Jeweils sechs Signale stehen für die Übersteuerung bzw. Verzögerung der Auswertung der sechs Asbru-Bedingungen eines Planes. Das letzte Signal gibt an, ob der Plan bevorzugt behandelt werden soll, wenn er Unterplan eines `anyorder` kontrollierten Plans ist.

Auch diese Datenstruktur ist genau so implementiert, wie von der Semantik vorgegeben.

```
data specification
using bool-tupel
```

```
    environment-aggregation-entry
  = mk-eae(. .manual : bool-tupel;
           . .override : bool-tupel;
           . .prefer : bool);


end data specification
```

A.4.4 Umgebungssignal-Sammlung (environment-aggregation)

Die Umgebungssignal-Sammlung speichert sämtliche Umgebungssignale aller Asbru-Pläne ab. Dies wird gemacht, um den Zustand von Asbru unabhängig von der Zahl der laufenden Pläne zu halten.

Die Sammlung der Umgebungssignale ist implementiert als eine prädikatenlogische Funktion, die Planinstanznamen abbildet auf Einträge der Umgebungssignalsammlungen. Da diese dynamischen Funktionen mehrfach benötigt werden, beispielsweise auch für die Krankenakte und die Asbru-Zustands-Historie benötigt werden, werden sie nur einmal als generischer Datentyp in KIV implementiert und dann mit den konkreten Datentypen aktualisiert.

```
actualize odynfun-sync
with environment-aggregation-entry by morphism

store → environment-aggregation;
data → environment-aggregation-entry;


end actualize
```

Der Datentyp der algebraisch spezifizierten dynamischen Funktionen in der KIV Implementierung von Asbru heißt `store`. Dieser Datentyp wird in dieser Spezifikation umbenannt nach `environment-aggregation`. Die Datenwerte, auf die ein allgemeiner `store` abbildet, heißen `data`. Diese Datentypen werden hier mit dem `environment-aggregation-entry`-Typ aktualisiert.

A.4.5 Hierarchiezustand (hierarchy-state)

Der Hierarchiezustand speichert die Laufzeitinformationen der Asbru-Pläne. Diese Laufzeit-Information beinhaltet die Zahl der bereits erfolgreich abgeschlossenen Durchläufe zyklischer Pläne, den aktuellen Planzustand sowie das Aktivierungssignal, das ein Plan benötigt, um vom Zustand `ready` in den Zustand `activated` wechseln zu können.

Der Hierarchiezustand wird in KIV in drei Komponenten zerlegt. Das heißt, anstelle von einer dynamischen Funktion die auf ein Werte-Tripel abbildet, wie in der Semantik, gibt es in KIV drei dynamische Funktionen, die jeweils einen Wert des Semantik-Tripel speichern. Eine Komponente speichert den Planzustand aller Pläne ab, die Zweite die Zahl der Durchläufe zyklischer Pläne und die dritte Komponente speichert die Aktivierungssignale aller Pläne.

Alle diese Komponenten werden als algebraisch spezifizierte dynamische Funktionen in KIV implementiert. Diese haben die Signaturen

- `string` ↦ `plan-state`
- `string` ↦ `nat`
- `string` ↦ `bool`

Der Asbru-Zustand der die Planzustände aller Pläne speichert wird als Aktualisierung der dynamischen Funktionen mit Planzuständen wie folgt definiert:

```
actualize odynfun-sync
with plan-state
by morphism

data → plan-state;
store → asbru-state;

end actualize
```

Dabei wird der allgemeine `store` Datentyp umbenannt nach `asbru-state`.

Analog dazu wird die Plan-Kommunikation als Aktualisierung der dynamischen Funktion mit booleschen Werten wie folgt definiert:

```
actualize odynfun-sync
with bool-sync
by morphism

data → bool;
store → plan-com;


end actualize
```

Zuletzt wird analog dazu der Rundenzähler als Aktualisierung der dynamischen Funktionen mit natürlichen Zahlen definiert:

```
actualize odynfun-sync
with int-basic1
by morphism

data → int;
store → round-count;


end actualize
```

A.4.6 Planzustandslisten (plan-state-list)

Die Planzustandslisten speichern alle Planzustände, die ein Asbru-Plan zwischen zwei Makro-Schritten durchläuft. Die Aufzeichnung erfolgt duplikatfolgenfrei. Das bedeutet, dass zwei aufeinanderfolgende Einträge in die Planzustandsliste stets unterschiedliche Werte haben.

In KIV werden die Planzustandslisten nicht als Aktualisierungen von generischen Listen implementiert. Stattdessen werden sie mittels einer Datenspezifikation spezifiziert. Der Grund dafür ist, dass es sich in Beweisen als nützlich für den Überblick über die Beweisverpflichtung herausgestellt hat, wenn der aktuelle Planzustand jeweils rechts an die Liste angefügt wird. Weiterhin ist aus Effizienzgründen die Simplifikationsstrategie der generischen Listen für die Planzustandslisten nur bedingt geeignet.

```
data specification
using plan-state, nat

   plan-state-list
  = [] | . ⇒ . ( . .previous : plan-state-list; . .current : plan-state) prio 8 left;


end data specification
```

A.4.7 Einträge in die Asbru-Zustands-Historie (asbru-state-history-entry)

Einträge in die Asbru-Zustands-Historie speichern für einen Plan die Planzustandslisten für alle Zeitpunkte ab. Damit ist es möglich, für einen Plan zu bestimmen, ob bestimmte Planzustandswechsel in der Vergangenheit stattgefunden haben oder nicht.

Implementiert wird diese Datenstruktur durch eine dynamische Funktion. Deren Syntax und Semantik werden allerdings nicht durch eine Aktualisierung der allgemeinen dynamischen Funktionen spezifiziert, sondern durch eine getrennte Spezifikation festgelegt. Grund dafür ist, dass die generischen dynamischen Funktionen als Abbildungen von Zeichenketten auf generische Werte definiert sind. Einträge in die Asbru-Zustands-Historie sollen allerdings Zeitpunkte, also Ganzzahl-Werte auf Daten abbilden.

```
enrich asbru-clock, plan-state-list with

sorts
  asbru-state-history-entry;

functions
  . [ . ] : asbru-state-history-entry × int → plan-state-list;

  . [ . ] : asbru-state-history-entry × int × plan-state → asbru-state-history-entry;
  . [ . ] : asbru-state-history-entry × int × plan-state-list → asbru-state-history-entry;


axioms

  select-base :
    ⊢ ashe[ac, psx][ac] = psx;
  used for : s, ls;

  select-rec :
    ⊢  ac ≠ ac0
      → ashe[ac0, psx][ac] = ashe[ac];
  used for : s, ls;

  insert-01:
    ⊢ ashe[ac, psx][ac, ps] = ashe[ac, psx + ps];
  used for : s, ls;

  insert-02:
    ⊢ ashe[ac, psx][ac + 1, ps] = ashe[ac, psx][ac + 1, [] ⇒ ps];
  used for : s, ls;

  extension :
    ⊢  ashe0 = ashe1
      ↔ (∀  ac. ashe0[ac] = ashe1[ac]);

end enrich
```

Da die dynamischen Funktionen hier neu spezifiziert werden, muss auch die Semantik der Aktualisierungs- und Selektionsfunktionen getrennt spezifiziert werden. Für Einträge in die Asbru-Zustands-Historie gibt es eine Selektionsfunktion sowie zwei Aktualisierungsfunktionen. Die Selektionsfunktion sucht nach einer „passenden“ Aktualisierung eines Eintrags. Eine Aktualisierung ist dann passend, wenn der Zeitstempel der Aktualisierung und der, der Selektion identisch sind. Eine Aktualisierung kann entweder durch die erste Aktualisierungsfunktion erfolgen, wobei die Planzustandsliste des Eintrags in die Asbru-Zustands-Historie durch die in der Funktion angegebene ersetzt wird oder aber durch die zweite Aktualisierungsfunktion. Diese ergänzt den betreffenden Eintrag nur anstatt ihn zu ersetzen. Dies wird durch die Axiome spezifiziert. Weiterhin angegeben wird ein Extensionalitätsaxiom, das angibt, wann zwei Einträge in die Asbru-Zustands-Historie gleich sind. Dies ist der Fall, wenn der Selektor zu jedem Zeitpunkt identische Werte in den Einträgen ermittelt.

A.4.8 Asbru-Zustands-Historie (asbru-state-history)

Die Asbru-Zustands-Historie speichert zentral die Einträge in die Asbru-Zustands-Historie für alle Pläne ab. Dies wird durch eine dynamische Funktion umgesetzt, die Planinstanznamen auf Einträge in die Asbru-Zustands-Historie abbildet.

In KIV wird dieser Datentyp als eine Aktualisierung der allgemeinen dynamischen Funktion durch Einträge in die Asbru-Zustands-Historie implementiert:

```
actualize odynfun-sync
with asbru-state-history-entry
by morphism

data → asbru-state-history-entry;
store → asbru-state-history;

end actualize
```

A.4.9 Krankenakten-Datentyp (data-value)

Der Datenwert-Datentyp ist eine abstrakte Repräsentation aller möglicher Datentypen, die in einer Krankenakte gespeichert werden können. Beispiele hierfür sind Zahlen, die beispielsweise benutzt werden um Blutdruck oder Temperatur zu speichern oder boolesche Werte, in denen gespeichert wird, ob bestimmte genetische Prädispositionen vorliegen. Die notwendigen Datentypen sind stark von der Fallstudie abhängig. Daher wird lediglich ein abstrakter Datentyp spezifiziert, der in Abhängigkeit der Fallstudie aktualisiert werden kann.

Entsprechend wird der Datentyp in KIV als generischer Datentyp spezifiziert:

```
specification

sorts
  data-value;

end specification
```

A.4.10 Krankenakten-Einträge und Krankenakten (patient-data und PDH)

Krankenakten-Einträge bilden alles vorhandene Wissen über einen Patienten zu einem Zustand ab. Wird bei einem Patienten in einer Fallstudie also beispielsweise die Körpertemperatur sowie Blutdruck und Puls gemessen, so enthält ein Krankenakten-Eintrag drei Felder, die mit Zeichenketten benannt werden. Diese Felder enthalten Werte vom Typ des allgemeinen Krankenakten-Datentyps bzw. seiner Aktualisierungen.

Krankenakten selber bündeln die Krankenakten-Einträge und ordnen sie zeitlich. Das heißt, in einer Krankenakte wird jedem Zeitpunkt ein Krankenakten-Eintrag zugeordnet.

Implementiert sind Krankenakten-Eintrag und Krankenakte in KIV durch die Aktualisierung einer dynamisch generierten Funktion. Der Krankenakten-Eintrag wird dabei durch Aktualisierung der allgemeinen dynamischen Funktionen, also des `store` Datentypen, spezifiziert. Aktualisiert wird diese mit dem Krankenakten-Datentyp. Für die Krankenakte selbst wird ebenfalls eine dynamische Funktion aktualisiert. Da eine Krankenakte aber nicht Zeichenketten, sondern Uhrzeiten auf Werte abbildet, muss hierfür ein neuer Typ `history` eingeführt werden.

Die `history` Datenstruktur implementiert die Krankenakte, ist also eine dynamische Funktion, die Zeitpunkte (also Ganzzahl-Werte) auf Krankenakten-Einträge abbildet.

```
actualize history
with gdata-value by morphism
```

```
history → patient-data-history;
store → patient-data;
data → data-value;


end actualize
```

Die `history` Datenstruktur spezifiziert eine dynamische Funktion, die Ganzzahlen auf dynamische Funktionen abbildet. Die Ganzzahlen repräsentieren dabei Uhrzeiten. Die dynamischen Funktionen werden durch den Datentyp `store` implementiert. Damit ist die `history` Datenstruktur wie folgt definiert:

```
enrich odynfun-sync, interval with

sorts
   history;

functions
   . [ . ] : history × int → store;
   . [ . ] : history × int × store → history;
   . [ . ] : history × interval × store → history;

predicates
  sync : history × history × history × history;

variables
  h, h0, h1, h2 : history;

axioms

  set-base :
    ⊢ h[ac, st][ac] = st;
  used for: s, ls;

  set-base-int :
    ⊢  ac ∈ ac1 × ac2
      → h[ac1 × ac2, st][ac] = st;
  used for: s, ls;

  set-rec :
    ⊢  ac ≠ ac0
      → h[ac0, st][ac] = h[ac];
  used for: s, ls;

  set-rec-int :
    ⊢  ¬ ac ∈ ac0 × ac1
      → h[ac0 × ac1, st][ac] = h[ac];
  used for: s, ls;

  extension:
    ⊢ h0 = h1 ↔ ∀  ac. h0[ac] = h1[ac];

end enrich
```

Der `history` Datentyp spezifiziert einen Standardselektor sowie eine Standardaktualisierungsfunktion. Der Selektor durchsucht die Aktualisierungen des Datentyps, bis die erste Aktualisierung gefunden wird, die den Zeitstempel trägt, nachdem durch den Selektor gesucht wird. Für die Generierung von Invarianten wird außerdem eine zweite Aktualisierungsfunktion definiert. Diese Aktualisiert die Krankenakte nicht zu einem Zeitpunkt, sondern für einen Zeitraum, der als Intervall zweier Zeitpunkte gegeben ist. Dabei soll diese Aktualisierung gelten für alle Zeitpunkte, die in diesem Intervall enthalten sind, einschließlich der Randpunkte.

Da der `history` Datentyp nicht frei erzeugt ist, muss ein Extensionalitätsaxiom angegeben werden, das spezifiziert, dass zwei `history` Typen gleich sind, wenn für jeden Zeitpunkt identische Werte durch den Selektor ermittelt werden.

A.4.11 Algebraisch definierte dynamische Funktionen (`store`)

Der `store` ist eine algebraische Definition dynamischer Funktionen.

```
generic specification

       parameter elemdata

       target

sorts store;

functions


        . [ . ] : store × elem × data → store;

        . [ . ] : store × elem → data;


variables st, st0, st1, st2 : store;


axioms

Extension : st1 = st2 ↔ ∀  a. st1[a] = st2[a]; used for : ls;

At-same  : st[a, d][a] = d; used for : s,ls;
At-other : a ≠ b → st[b, d][a] = st[a]; used for : s,ls;

end generic specification
```

Die `store` Datenstruktur spezifiziert eine Aktualisierungsfunktion sowie einen Selektor. Dabei soll der Selektor den Wert der letzten Aktualisierung zurückgeben, die mit dem entsprechenden Schlüssel `a` durchgeführt wurde, nachdem durch den Selektor gesucht wird. Da die dynamischen Funktionen nicht frei erzeugt sind, muss ein Extensionalitätsaxiom angegeben werden.

B Korrektheit der Einbettung der Prozeduren

In diesem Abschnitt werden alle bewiesenen Beweisverpflichtungen beschrieben, die zusammengenommen die Korrektheit der Implementierung der SOS Regeln durch parallele Programme in KIV nachweisen. Zusätzlich wird für jeden Planzustand sowie für jeden Plankontrolltyp ein Beweis geführt, der nachweist, dass die Vorbedingungen aller zu diesem Zustand oder Plantyp gehörigen Regeln Tautologien sind. Das bedeutet, die entsprechenden Regeln decken den kompletten Zustandsraum ab. Das bedeutet, dass beispielsweise kein Zustand existieren kann, in dem ein `sequential`-kontrollierter Plan keine Schritte mehr ausführen kann, weil eine „passende“ Übergangsregel fehlt.

B.1 `considered`-Zustand

Ein Asbru-Plan im Zustand `considered` kann seine Bearbeitung abbrechen, wenn entweder die **filter**-Bedingung nicht erfüllbar ist oder der übergeordnete Plan seine Bearbeitung eingestellt hat. Andernfalls bleibt der Plan im Zustand `considered`, wenn die **filter**-Bedingung nicht erfüllt aber erfüllbar ist und er wechselt in den Zustand `possible`, wenn die **filter**-Bedingung erfüllt ist.

B.1.1 Zurückweisung des Plans

Ein Plan wird laut SOS-Regel zurückgewiesen, wenn seine **filter**-Bedingung dies erfordert oder der übergeordnete Plan terminiert ist.

$$\frac{\begin{array}{l}\mathbf{isConsidered}(\mathbf{HS},\mathbf{iname})\\ \quad\wedge\ \ \mathbf{shouldRejectFilter}(\sigma,\mathbf{iname})\\ \quad\vee\ \mathbf{parent\_terminated}(\mathbf{HS},\mathbf{iname})\end{array}}{\mathbf{iname}\ \ \sigma,\mathbf{Tick}(\sigma[\mathbf{iname},\ \overrightarrow{\ }\ \texttt{rejected}])\ \ \varepsilon}$$

Dazu wird bewiesen, dass der Plan ordnungsgemäß zurückgewiesen wird, wenn der übergeordnete Plan terminiert oder die **filter**-Bedingung nicht erfüllbar ist:

$$\begin{array}{l}\mathbf{isConsidered}(\mathbf{AS},\texttt{iname}),\\ \quad\mathbf{shouldRejectFilter}(\texttt{iname},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AC})\\ \vee\ \mathbf{parent\_terminated}(\mathbf{AS},\texttt{iname}),\\ [\texttt{considered\#}(\texttt{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})]\\ \vdash\ \ [\mathbf{AS},\ \mathbf{Tick}]\\ \wedge\ \neg\ \mathbf{Tick}'\\ \wedge\ \mathbf{AS}'=\mathbf{AS}[\texttt{iname},\texttt{rejected}]\\ \wedge\ \circ\ \mathbf{last};\end{array}$$

B.1.2 Stillstand im Zustand considered

Wenn ein Plan im Zustand considered nicht zurückgewiesen wird aber seine **filter**-Bedingung noch erfüllbar ist, dann wartet der Plan in seinem Zustand passiv ab.

$$\frac{\begin{array}{l}\neg\ \mathbf{parent\_terminated}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \mathbf{isConsidered}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \neg\ \mathbf{shouldRejectFilter}(\sigma, \mathbf{iname}) \\ \wedge\ \neg\ \mathbf{shouldProceedFilter}(\sigma, \mathbf{iname})\end{array}}{\mathbf{iname}\ \ \sigma, \sigma[\mathbf{iname}, \xrightarrow{\ \ } \texttt{considered}]\ \ \mathbf{iname}}$$

Die Korrektheit der Einbettung wird mit nachstehender Beweisverpflichtung gezeigt:

$$\begin{array}{l}\mathbf{isConsidered}(\mathbf{AS}, \texttt{iname}), \\ \neg\ \mathbf{shouldRejectFilter}(\texttt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AC}), \\ \neg\ \mathbf{parent\_terminated}(\mathbf{AS}, \texttt{iname}), \\ \neg\ \mathbf{shouldProceedFilter}(\texttt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AC}), \\ [\texttt{considered\#(iname;}\mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})] \\ \vdash\ \ \lceil \mathbf{AS} \rceil \\ \wedge\ \mathbf{AS}' = \mathbf{AS}[\texttt{iname}, \texttt{considered}] \\ \wedge\ \circ\ [\texttt{considered\#(iname;}\mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})];\end{array}$$

B.1.3 Fortschreiten vom Zustand considered nach possible

Ist die **filter**-Bedingung eines Plans im Zustand considered erfüllt, so wird dieser Plan in den Zustand possible wechseln.

$$\frac{\begin{array}{l}\neg\ \mathbf{parent\_terminated}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \mathbf{isConsidered}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \mathbf{shouldProceedFilter}(\sigma, \mathbf{iname})\end{array}}{\mathbf{iname}\ \ \sigma, \mathbf{Tick}(\sigma[\mathbf{iname}, \xrightarrow{\ \ } \texttt{possible}])\ \ \mathbf{iname}}$$

Die Implementierung in KIV hält sich an diese Semantikregel, was durch den Nachweis der nachstehenden Beweisverpflichtung gezeigt wurde:

$$\begin{array}{l}\mathbf{isConsidered}(\mathbf{AS}, \texttt{iname}), \\ \neg\ \mathbf{parent\_terminated}(\mathbf{AS}, \texttt{iname}), \\ \mathbf{shouldProceedFilter}(\texttt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AC}), \\ [\texttt{considered\#(iname;}\mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})] \\ \vdash\ \ \lceil \mathbf{AS},\ \mathbf{Tick} \rceil \\ \wedge\ \neg\ \mathbf{Tick}' \\ \wedge\ \mathbf{AS}' = \mathbf{AS}[\texttt{iname}, \texttt{possible}] \\ \wedge\ \circ\ [\texttt{possible\#(iname;}\mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})];\end{array}$$

B.1.4 Vollständigkeit der considered Regeln

Nachstehende Beweisverpflichtung fasst die Vorbedingungen aller Beweisverpflichtungen bezüglich des considered Zustandes zusammen. Es wird gezeigt, dass diese Vorbedingungen eine Tautologie sind, das heißt, es gibt keinen Zustand, in dem ein Plan im Planzustand considered keinen Schritt mehr ausführen kann.

$$
\begin{aligned}
\vdash\ & \neg\ \mathbf{parent\_terminated}(\mathbf{AS}, \mathtt{iname}) \\
& \wedge\ \mathbf{shouldProceedFilter}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AC}), \\
& \quad \neg\ \mathbf{parent\_terminated}(\mathbf{AS}, \mathtt{iname}) \\
& \quad \wedge \neg\ \mathbf{shouldProceedFilter}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AC}) \\
& \quad \wedge \neg\ \mathbf{shouldRejectFilter}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AC}), \\
& \quad \mathbf{parent\_terminated}(\mathbf{AS}, \mathtt{iname}) \\
& \quad \vee\ \mathbf{shouldRejectFilter}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AC})
\end{aligned}
$$

B.2 possible-Zustand

Ein Plan im Zustand possible verhält sich analog zu einem Plan im Zustand considered, nur dass statt der **filter**-Bedingung die **setup**-Bedingung ausgewertet wird. Ist diese erfüllt, so wechselt der Plan in den Planzustand ready, andernfalls bleibt er im Zustand possible oder wechselt in den Zustand rejected.

B.2.1 Zurückweisung des Plans

Ein Plan im Zustand possible wird zurückgewiesen, wenn seine **setup**-Bedingung nicht mehr erfüllbar ist oder der übergeordnete Plan terminiert.

$$
\frac{\begin{array}{l}\mathbf{isPossible}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \ \mathbf{shouldRejectSetup}(\sigma, \mathbf{iname}) \\ \quad \vee\ \mathbf{parent\_terminated}(\mathbf{HS}, \mathbf{iname})\end{array}}{\mathbf{iname}\ \ \sigma, \mathbf{Tick}(\sigma[\mathbf{iname}, \xrightarrow{} \mathtt{rejected}])\ \ \varepsilon}
$$

Die erste Beweisverpflichtung zeigt, dass ein Plan im Zustand possible zurückgewiesen wird, wenn sein übergeordneter Plan terminiert oder die **setup**-Bedingung nicht mehr erfüllbar ist:

$$
\begin{aligned}
& \mathbf{isPossible}(\mathbf{AS}, \mathtt{iname}), \\
& \quad \mathbf{shouldRejectSetup}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AC}) \\
& \vee\ \mathbf{parent\_terminated}(\mathbf{AS}, \mathtt{iname}), \\
& [\mathtt{possible\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})] \\
\vdash\ & [\mathbf{AS}, \mathbf{Tick}] \\
& \wedge\ \neg\ \mathbf{Tick}' \\
& \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname}, \mathtt{rejected}] \\
& \wedge\ \circ\ \mathbf{last};
\end{aligned}
$$

B.2.2 Stillstand im Zustand `possible`

Ist im Zustand `possible` die **setup** Bedingung weder erfüllt noch nicht mehr erfüllbar, so wartet der Plan auf eine Zustandsänderung, sofern der übergeordnete Plan noch nicht terminiert ist.

$$\frac{\begin{array}{l}\neg\ \mathbf{parent\_terminated}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \mathbf{isPossible}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \neg\ \mathbf{shouldProceedSetup}(\sigma, \mathbf{iname}) \\ \wedge\ \neg\ \mathbf{shouldRejectSetup}(\sigma, \mathbf{iname})\end{array}}{\mathbf{iname}\ \ \sigma, \sigma[\mathbf{iname}, \xrightarrow{}\ \mathtt{possible}]\ \ \mathbf{iname}}$$

Dies wird durch nachstehende Beweisverpflichtung gezeigt:

$$\begin{array}{l}\mathbf{isPossible}(\mathbf{AS}, \mathtt{iname}), \\ \neg\ \mathbf{shouldRejectSetup}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AC}), \\ \neg\ \mathbf{parent\_terminated}(\mathbf{AS}, \mathtt{iname}), \\ \neg\ \mathbf{shouldProceedSetup}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AC}), \\ [\mathtt{possible\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})] \\ \vdash\ \ \lceil \mathbf{AS} \rceil \\ \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname}, \mathtt{possible}] \\ \wedge\ \circ\ [\mathtt{possible\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})];\end{array}$$

B.2.3 Fortschreiten vom Zustand `possible` nach `ready`

Ist die **setup** Bedingung im Zustand `possible` erfüllt und der übergeordnete Plan ist nicht in einem terminierten Zustand, so schreitet der Plan vorwärts in den Zustand `ready`.

$$\frac{\begin{array}{l}\neg\ \mathbf{parent\_terminated}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \mathbf{isPossible}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \mathbf{shouldProceedSetup}(\sigma, \mathbf{iname})\end{array}}{\mathbf{iname}\ \ \sigma, \mathbf{Tick}(\sigma[\mathbf{iname}, \xrightarrow{}\ \mathtt{ready}])\ \ \mathbf{iname}}$$

Dies wird gezeigt durch die nachstehende Beweisverpflichtung:

$$\begin{array}{l}\mathbf{isPossible}(\mathbf{AS}, \mathtt{iname}), \\ \neg\ \mathbf{parent\_terminated}(\mathbf{AS}, \mathtt{iname}), \\ \mathbf{shouldProceedFilter}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AC}), \\ [\mathtt{possible\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})] \\ \vdash\ \ \lceil \mathbf{AS}, \mathbf{Tick} \rceil \\ \wedge\ \neg\ \mathbf{Tick}' \\ \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname}, \mathtt{ready}] \\ \wedge\ \circ\ [\mathtt{ready\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})];\end{array}$$

B.2.4 Vollständigkeit der possible Regeln

Ein Plan im Zustand possible kann stets irgendeine Handlung durchführen. Dies wird dadurch bewiesen, dass gezeigt wird, dass stets eine der Vorbedingungen der Zustandsübergangsregeln aus dem possible Zustand wahr sein muss:

$$
\begin{aligned}
\vdash \quad & \neg\ \mathbf{parent\_terminated}(\mathbf{AS}, \mathtt{iname}) \\
& \wedge\ \mathbf{shouldProceedSetup}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AC}), \\
& \quad \neg\ \mathbf{parent\_terminated}(\mathbf{AS}, \mathtt{iname}) \\
& \wedge \neg\ \mathbf{shouldProceedSetup}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AC}) \\
& \wedge \neg\ \mathbf{shouldRejectSetup}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AC}), \\
& \quad \mathbf{parent\_terminated}(\mathbf{AS}, \mathtt{iname}) \\
& \vee\ \mathbf{shouldRejectSetup}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AC})
\end{aligned}
$$

B.3 ready-Zustand

Im Zustand ready hat ein Plan die Selection-Phase weitgehend durchlaufen und wartet auf das Aktivierungssignal seines übergeordneten Plans. Sobald er dies erhält, darf er sich in die Execution-Phase versetzen. Erhält er es nicht, so muss er abwarten. Sobald er feststellt, dass der übergeordnete Plan terminiert ist, bricht der Plan sich selbst ab.

B.3.1 Zurückweisung des Plans

Der Plan wird zurückgewiesen, wenn der übergeordnete Plan terminiert ist.

$$
\frac{\begin{array}{l}\mathbf{parent\_terminated}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \mathbf{isReady}(\mathbf{HS}, \mathbf{iname})\end{array}}{\mathbf{iname}\ \ \sigma, \mathbf{Tick}(\sigma[\mathbf{iname}, \xrightarrow{} \mathtt{rejected}])}\ \varepsilon
$$

Dies wird durch die Beweisverpflichtung gezeigt:

$$
\begin{aligned}
& \mathbf{isReady}(\mathbf{AS}, \mathtt{iname}), \\
& \mathbf{parent\_terminated}(\mathbf{AS}, \mathtt{iname}), \\
& [\mathtt{ready\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})] \\
\vdash \quad & [\mathbf{AS}, \mathbf{Tick}] \\
& \wedge \neg\ \mathbf{Tick}' \\
& \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname}, \mathtt{rejected}] \\
& \wedge \circ\ \mathbf{last};
\end{aligned}
$$

B.3.2 Stillstand im Zustand ready

Der Plan wartet im Zustand ready ab, wenn der übergeordnete Plan nicht terminiert ist und außerdem kein Aktivierungssignal für den Plan vorliegt.

$$\frac{\begin{array}{l}\neg\ \mathbf{parent\_terminated}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \mathbf{isReady}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \neg\ \mathbf{isActSig}(\mathbf{HS}, \mathbf{iname})\end{array}}{\mathbf{iname}\ \overset{\sigma, \sigma[\mathbf{iname},\ \mathtt{ready}]}{\longrightarrow}\ \mathbf{iname}}$$

Dies wird in KIV wie folgt formuliert:

$$\begin{array}{l}\quad \mathbf{isReady}(\mathbf{AS}, \mathtt{iname}), \\ \quad \neg\ \mathbf{parent\_terminated}(\mathbf{AS}, \mathtt{iname}), \\ \quad \neg\ \mathbf{isActSig}(\mathbf{PC}, \mathtt{iname}), \\ \quad [\mathtt{ready\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})] \\ \vdash\ \ \lceil \mathbf{AS} \rceil \\ \quad \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname}, \mathtt{ready}] \\ \quad \wedge\ \circ\ [\mathtt{ready\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})];\end{array}$$

B.3.3 Fortschreiten vom Zustand `ready` nach `activated`

Nach Erhalt des Aktivierungssignals darf der Plan in den Zustand `activated` fortschreiten, sofern der übergeordnete Plan nicht terminiert ist:

$$\frac{\begin{array}{l}\neg\ \mathbf{parent\_terminated}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \mathbf{isReady}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \mathbf{isActSig}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \mathbf{PL} = \mathbf{selectAllStartable}(\mathbf{HS}, \mathbf{iname})\end{array}}{\mathbf{iname}\ \overset{\sigma, \mathbf{Tick}(\sigma\ [\mathbf{iname}, \mathtt{activated}][\mathbf{PL}, \mathbf{iname}]_{nac}\ [\mathbf{PL}, \mathbf{iname}, \mathtt{inactive}][\mathbf{iname}, 0][\mathbf{iname}]_{nac})}{\longrightarrow}\ \mathbf{iname}}$$

Nachgewiesen wird dies durch den Beweis der nachstehenden Aussage:

$$\begin{array}{l}\quad \mathbf{isReady}(\mathbf{AS}, \mathtt{iname}), \\ \quad \neg\ \mathbf{parent\_terminated}(\mathbf{AS}, \mathtt{iname}), \\ \quad \mathbf{isActSig}(\mathbf{PC}, \mathtt{iname}), \\ \quad [\mathtt{ready\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})] \\ \vdash\ \ \lceil \mathbf{AS, PC, RC, Tick} \rceil \\ \quad \wedge\ \neg\ \mathbf{Tick}' \\ \quad \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname}, \mathtt{activated}][\mathtt{asbru}(\mathbf{nameSel}(\mathtt{iname}))\mathbf{.subplans}, \mathtt{iname}, \mathtt{inactive}] \\ \quad \wedge\ \mathbf{PC}' = \mathbf{PC}[\mathtt{iname}, \mathbf{false}][\mathtt{asbru}(\mathbf{nameSel}(\mathtt{iname}))\mathbf{.subplans}, \mathtt{iname}, \mathbf{false}] \\ \quad \wedge\ \mathbf{RC}' = \mathbf{RC}[\mathtt{iname}, 0] \\ \quad \wedge\ \circ\ [\mathtt{activated\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})];\end{array}$$

B.3.4 Vollständigkeit der `ready` Regeln

Auch die Regeln für Pläne im Zustand `ready` sind Vollständig in dem Sinn, dass stets eine Handlungsmöglichkeit besteht. Dies wird mittels nachstehender Aussage nachgewiesen:

$$
\begin{array}{ll}
\vdash & \neg\ \mathbf{parent\_terminated}(\mathbf{AS}, \mathtt{iname}) \\
& \wedge\ \mathbf{isActSig}(\mathbf{PC}, \mathtt{iname}), \\
& \quad \neg\ \mathbf{parent\_terminated}(\mathbf{AS}, \mathtt{iname}) \\
& \wedge \neg\ \mathbf{isActSig}(\mathbf{PC}, \mathtt{iname}), \\
& \mathbf{parent\_terminated}(\mathbf{AS}, \mathtt{iname})
\end{array}
$$

B.4 activated-Zustand

Die Zustandsübergangsregeln für Pläne im Zustand activated beschreiben vier Situationen. In einer soll der Plan abbrechen, weil seine **abort**-Bedingung erfüllt ist, seine Wartebedingung nicht mehr erfüllt werden kann oder sein übergeordneter Plan terminiert ist. Die zweite Situation liegt immer dann vor, wenn der Plan nicht abbrechen soll und sowohl die **complete**-Bedingung als auch die Wartebedingung erfüllt sind. Dann beendet der Plan die Ausführung. Die dritte Situation ist immer dann gegeben, wenn der Plan weder abbrechen noch beenden soll und die **suspend**-Bedingung erfüllt ist oder der übergeordnete Plan unterbrochen wurde. Dann soll der Plan sich unterbrechen. Ist keine dieser Situationen gegeben, so wird die Kontrolle an die Plankontrolltypenspezifischen Regeln übergeben. Dies wird in der Semantik durch den Aufruf der con Regeln gemacht, in der Implementierung durch Ausführung der body# Prozedur.

B.4.1 Abbrechen des Plans

Der erste Fall beschreibt das Abbrechen des Plans. Die verschiedenen Abbruchbedingungen sind im Prädikat **shouldAbort** gekapselt.

$$
\frac{\begin{array}{l}\mathbf{shouldAbort}(\sigma, \mathbf{iname}) \\ \wedge\ \ \mathbf{isActivated}(\mathbf{HS}, \mathbf{iname}) \\ \quad \vee\ \mathbf{isSuspended}(\mathbf{HS}, \mathbf{iname})\end{array}}{\mathbf{iname}\ \ \sigma, \mathbf{Tick}(\sigma[\mathbf{iname}, \xrightarrow{}\ \mathtt{aborted}])\ _{\varepsilon}}
$$

Die Regel wird durch die Implementierung korrekt nachgebildet, wie nachfolgende Beweisverpflichtung zeigt:

$$
\begin{array}{ll}
& \mathbf{shouldAbort}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}), \\
& \quad \mathbf{isActivated}(\mathbf{AS}, \mathtt{iname}) \\
& \quad \wedge\ [\mathtt{activated\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})] \\
& \vee\ \ \mathbf{isSuspended}(\mathbf{AS}, \mathtt{iname}) \\
& \quad \wedge\ [\mathtt{suspended\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})] \\
\vdash & \ [\mathbf{AS}, \mathbf{Tick}] \\
& \wedge \neg\ \mathbf{Tick}' \\
& \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname}, \mathtt{aborted}] \\
& \wedge \circ\ \mathbf{last};
\end{array}
$$

B.4.2 Beenden des Plans

Ein Plan soll beendet werden, wenn er nicht abgebrochen werden soll und das Prädikat **shouldComplete** wahr ist, dass die Konjunktion aus Erfülltheit der Wartebedingung und der **complete**-Bedingung kapselt.

$$\frac{\begin{array}{l}\neg\ \mathbf{shouldAbort}(\sigma, \mathbf{iname}) \\ \wedge\ \mathbf{shouldComplete}(\sigma, \mathbf{iname}) \\ \wedge\ \mathbf{isActivated}(\mathbf{HS}, \mathbf{iname})\end{array}}{\mathbf{iname}\ \ \sigma, \mathbf{Tick}(\sigma[\mathbf{iname}, \xrightarrow{}\ \texttt{completed}])\ \ \varepsilon}$$

Die Korrektheit der Einbettung kann mit nachstehender Aussage gezeigt werden:

$$\begin{array}{l}\neg\ \mathbf{shouldAbort}(\texttt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}), \\ \mathbf{shouldComplete}(\texttt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}), \\ \mathbf{isActivated}(\mathbf{AS}, \texttt{iname}), \\ [\texttt{activated\#}(\texttt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})] \\ \vdash\ \ \lceil \mathbf{AS},\ \mathbf{Tick} \rceil \\ \wedge\ \neg\ \mathbf{Tick}' \\ \wedge\ \mathbf{AS}' = \mathbf{AS}[\texttt{iname}, \texttt{completed}] \\ \wedge\ \circ\ \mathbf{last};\end{array}$$

B.4.3 Unterbrechen des Plans

Der Plan soll unterbrochen werden, wenn die entsprechende Bedingung wahr ist und er weder abgebrochen noch beendet werden soll.

$$\frac{\begin{array}{l}\neg\ \mathbf{shouldAbort}(\sigma, \mathbf{iname}) \\ \wedge\ \neg\ \mathbf{shouldComplete}(\sigma, \mathbf{iname}) \\ \wedge\ \mathbf{shouldSuspend}(\sigma, \mathbf{iname}) \\ \wedge\ \mathbf{isActivated}(\mathbf{HS}, \mathbf{iname})\end{array}}{\mathbf{iname}\ \ \sigma, \mathbf{Tick}(\sigma[\mathbf{iname}, \xrightarrow{}\ \texttt{suspended}])\ \ \mathbf{iname}}$$

Die Implementierung empfindet dieses Verhalten nach, wie mit folgender Beweisverpflichtung gezeigt wurde:

$$\begin{array}{l}\neg\ \mathbf{shouldAbort}(\texttt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}), \\ \neg\ \mathbf{shouldComplete}(\texttt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}), \\ \mathbf{shouldSuspend}(\texttt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}), \\ \mathbf{isActivated}(\mathbf{AS}, \texttt{iname}), \\ [\texttt{activated\#}(\texttt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})] \\ \vdash\ \ \lceil \mathbf{AS},\ \mathbf{Tick} \rceil \\ \wedge\ \neg\ \mathbf{Tick}' \\ \wedge\ \mathbf{AS}' = \mathbf{AS}[\texttt{iname}, \texttt{suspended}] \\ \wedge\ \circ\ [\texttt{suspended\#}(\texttt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})];\end{array}$$

B.4.4 Ausführen der Plankontrolle

Liegt keiner der drei anderen Fälle vor, so sollen die niedriger priorisierten con Übergangsregeln verwendet werden.

$$\frac{\begin{array}{l}\quad\neg\ \mathbf{shouldAbort}(\sigma, \mathbf{iname})\\ \wedge\ \neg\ \mathbf{shouldSuspend}(\sigma, \mathbf{iname})\\ \wedge\ \neg\ \mathbf{shouldComplete}(\sigma, \mathbf{iname})\\ \wedge\ \mathbf{isActivated}(\mathbf{HS}, \mathbf{iname})\\ \wedge\ \mathbf{iname} \xrightarrow{\sigma,\sigma'}_{\mathtt{Con}} \mathbf{sys}\end{array}}{\mathbf{iname} \xrightarrow{\sigma,\sigma'} \mathbf{sys}}$$

Diese werden in der Implementierung durch die body# Prozedur implementiert:

$$\begin{array}{l}\quad\neg\ \mathbf{shouldAbort}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}),\\ \quad\neg\ \mathbf{shouldComplete}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}),\\ \quad\neg\ \mathbf{shouldSuspend}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}),\\ \quad\mathbf{isActivated}(\mathbf{AS}, \mathtt{iname}),\\ \quad[\mathtt{activated\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})]\\ \vdash\ [\mathtt{body\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})];\end{array}$$

B.4.5 Vollständigkeit der activated Regeln

Auch der activated Zustand ist komplett in der Hinsicht, dass stets eine der Regeln anwendbar ist. Dies wird durch die nachfolgende Beweisverpflichtung gezeigt:

$$\begin{array}{ll}\vdash & \quad\neg\ \mathbf{shouldAbort}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC})\\ & \wedge\ \neg\ \mathbf{shouldComplete}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC})\\ & \wedge\ \neg\ \mathbf{shouldSuspend}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}),\\ & \quad\neg\ \mathbf{shouldAbort}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC})\\ & \wedge\ \neg\ \mathbf{shouldComplete}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC})\\ & \wedge\ \mathbf{shouldSuspend}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}),\\ & \quad\neg\ \mathbf{shouldAbort}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC})\\ & \wedge\ \mathbf{shouldComplete}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}),\\ & \mathbf{shouldAbort}(\mathtt{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC});\end{array}$$

B.5 suspended-Zustand

Der suspended Zustand beschreibt einen Zustand, in dem der Plan keine weiteren Unterpläne startet oder aktiviert. Alle Unterpläne, die sich bereits in der Execution-Phase befinden, werden ebenfalls unterbrochen. Der suspended-Zustand wird nur verlassen, wenn der übergeordnete Plan nicht (mehr) unterbrochen ist und die **reactivate**-Bedingung erfüllt ist.

B.5.1 Abbrechen des Plans

Auch im suspended-Zustand kann ein Plan abbrechen, wenn der entsprechende Zustand vorliegt, also die Wartebedingung nicht mehr erfüllbar ist oder die **abort**-Bedingung erfüllt ist bzw. der übergeordnete Plan terminiert ist.

$$\frac{\begin{array}{l}\mathbf{shouldAbort}(\sigma, \mathbf{iname}) \\ \wedge \quad \mathbf{isActivated}(\mathbf{HS}, \mathbf{iname}) \\ \quad \vee \mathbf{isSuspended}(\mathbf{HS}, \mathbf{iname})\end{array}}{\mathbf{iname} \quad \sigma, \mathbf{Tick}(\sigma[\mathbf{iname}, \text{ aborted}]) \quad \varepsilon}$$

Auch die Implementierung empfindet diesen Zustandsübergang nach, wie durch nachstehende Beweisverpflichtung gezeigt wurde:

$$\begin{array}{l}\quad \mathbf{shouldAbort}(\text{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}), \\ \quad \mathbf{isActivated}(\mathbf{AS}, \text{iname}) \\ \wedge\ [\text{activated\#}(\text{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})], \\ \quad \mathbf{isSuspended}(\mathbf{AS}, \text{iname}) \\ \wedge\ [\text{suspended\#}(\text{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})], \\ \vdash \quad \lceil \mathbf{AS}, \ \mathbf{Tick} \rceil \\ \wedge\ \neg\ \mathbf{Tick}' \\ \wedge\ \mathbf{AS}' = \mathbf{AS}[\text{iname}, \text{aborted}] \\ \wedge\ \circ\ \mathbf{last};\end{array}$$

B.5.2 Stillstand im Zustand suspended

Soll der Plan im Zustand suspended weder abbrechen noch seine Ausführung wieder aufnehmen, so verbleibt er im Zustand suspended.

$$\frac{\begin{array}{l}\neg\ \mathbf{shouldAbort}(\sigma, \mathbf{iname}) \\ \wedge\ \neg\ \mathbf{shouldReactivate}(\sigma, \mathbf{iname}) \\ \wedge\ \mathbf{isSuspended}(\mathbf{HS}, \mathbf{iname})\end{array}}{\mathbf{iname} \quad \sigma, \sigma[\mathbf{iname}, \text{ suspended}] \quad \mathbf{iname}}$$

Analog verhält sich die Implementierung:

$$\begin{array}{l}\neg\ \mathbf{shouldAbort}(\text{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}), \\ \neg\ \mathbf{shouldReactivate}(\text{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}), \\ \mathbf{isSuspended}(\mathbf{AS}, \text{iname}), \\ [\text{suspended\#}(\text{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})] \\ \vdash \quad \lceil \mathbf{AS} \rceil \\ \wedge\ \mathbf{AS}' = \mathbf{AS}[\text{iname}, \text{suspended}] \\ \wedge\ \circ\ [\text{suspended\#}(\text{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})];\end{array}$$

B.5.3 Reaktivierung des Plans

Ein Plan im Zustand suspended wird reaktiviert, wenn er nicht abbrechen soll, seine **reactivate**-Bedingung erfüllt ist und der übergeordnete Plan nicht (mehr) unterbrochen ist.

$$\frac{\begin{array}{l}\neg\ \mathbf{shouldAbort}(\sigma, \mathbf{iname}) \\ \wedge\ \mathbf{shouldReactivate}(\sigma, \mathbf{iname}) \\ \wedge\ \mathbf{isSuspended}(\mathbf{HS}, \mathbf{iname})\end{array}}{\mathbf{iname}\quad \sigma_{,\mathbf{Tick}(\sigma[\mathbf{iname}, \xrightarrow{} \text{activated}])}\quad \mathbf{iname}}$$

Auch die Implementierung hält sich an diese Forderung und reaktivert den Plan in diesem Fall.

$$\begin{array}{l}\neg\ \mathbf{shouldAbort}(\text{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}), \\ \mathbf{shouldReactivate}(\text{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}), \\ \mathbf{isSuspended}(\mathbf{AS}, \text{iname}), \\ [\text{suspended\#}(\text{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})] \\ \vdash\ \ [\mathbf{AS}, \mathbf{Tick}] \\ \wedge\ \neg\ \mathbf{Tick}' \\ \wedge\ \mathbf{AS}' = \mathbf{AS}[\text{iname}, \text{activated}] \\ \wedge\ \circ\ [\text{activated\#}(\text{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})];\end{array}$$

B.5.4 Vollständigkeit der suspended Regeln

Die Regeln zum Umgang mit unterbrochenen Plänen sind vollständig. Jeder unterbrochene Plan kann stets durch eine der Regeln seine Ausführung fortsetzen:

$$\begin{array}{l}\vdash\ \ \neg\ \mathbf{shouldAbort}(\text{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}) \\ \quad \wedge \neg\ \mathbf{shouldReactivate}(\text{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}), \\ \quad\ \ \neg\ \mathbf{shouldAbort}(\text{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}) \\ \quad \wedge\ \mathbf{shouldReactivate}(\text{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}), \\ \quad \mathbf{shouldAbort}(\text{iname}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC});\end{array}$$

B.6 Zustandsübergänge der sequential-Plankontrolle

Die sequential Plankontrolle startet und aktiviert stets genau einen Unterplan. Dabei wird die Reihenfolge der Unterpläne in der Unterplanliste beachtet. Wird bereits ein Unterplan ausgeführt, so verhält sich die sequential-Kontrolle passiv und wartet ab.

B.6.1 Passiver Übergang

Wenn ein sequential-kontrollierter Plan einen Unterplan in der Execution-Phase oder der Selection-Phase hat oder aber kein Unterplan im startbaren Zustand vorliegt, so verhält sich der Plan passiv.

$$\frac{\begin{array}{l}\mathbf{isSequential}(\mathbf{iname})\\ \wedge\ \ \mathbf{hasExecutedSub}(\mathbf{HS},\mathbf{iname})\\ \quad\vee\ \mathbf{hasSelectedSub}(\mathbf{HS},\mathbf{iname})\\ \quad\vee\ \neg\ \mathbf{hasStartableSub}(\mathbf{HS},\mathbf{iname})\end{array}}{\mathbf{iname}\ \ {}^{\sigma,\sigma[\mathbf{iname},\ \underrightarrow{\ }\ \mathtt{activated}]}_{\mathtt{Con}}\ \ \mathbf{iname}}$$

Dies wird entsprechend von der Implementierung umgesetzt:

$$\begin{array}{l}\mathbf{isSequential}(\mathtt{iname}),\\ \quad\mathbf{hasExecutedSub}(\mathbf{AS},\mathtt{iname})\\ \quad\vee\ \mathbf{hasSelectedSub}(\mathbf{AS},\mathtt{iname})\\ \quad\vee\ \neg\ \mathbf{hasStartableSub}(\mathbf{AS},\mathtt{iname}),\\ \quad[\mathtt{body\#}(\mathtt{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})]\\ \vdash\ \ \lceil\mathbf{AS}\rceil\\ \quad\wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname},\mathtt{activated}]\\ \quad\wedge\ \circ\ [\mathtt{activated\#}(\mathtt{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})];\end{array}$$

B.6.2 Aktivierungsfall

Wenn ein `sequential`-kontrollierter Plan keine Unterpläne in `Selection`- oder `Execution`-Phase hat, so wird er, sofern ein startbarer Unterplan vorliegt, diesen starten und gleichzeitig aktivieren.

$$\frac{\begin{array}{l}\mathbf{isSequential}(\mathbf{iname})\\ \wedge\ \neg\ \mathbf{hasExecutedSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \neg\ \mathbf{hasSelectedSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \mathbf{hasStartableSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \mathbf{iname}_1 = \mathbf{nameGen}(\mathbf{selectFirstStartable}(\mathbf{HS},\mathbf{iname}),\mathbf{iname})\end{array}}{\mathbf{iname}\ \ {}^{\sigma,\mathbf{Tick}(\sigma[\mathbf{iname},\ \mathtt{activated}][\underrightarrow{\mathbf{iname}_1},\ \mathtt{considered}][\mathbf{iname}_1]_{ac})}_{\mathtt{Con}}\ \ \mathbf{iname}\|_s\ \mathbf{iname}_1}$$

Dies wird auch von der Implementierung so gehandhabt:

$$\begin{array}{l}\mathbf{isSequential}(\mathtt{iname}),\\ \neg\ \mathbf{hasExecutedSub}(\mathbf{AS},\mathtt{iname}),\\ \neg\ \mathbf{hasSelectedSub}(\mathbf{AS},\mathtt{iname}),\\ \mathbf{hasStartableSub}(\mathbf{AS},\mathtt{iname}),\\ \mathtt{sk} = \mathbf{selectFirstStartable}(\mathbf{AS},\mathtt{iname}),\\ [\mathtt{body\#}(\mathtt{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})]\\ \vdash\ \ \lceil\mathbf{AS},\ \mathbf{PC},\ \mathbf{Tick}\rceil\\ \quad\wedge\ \neg\ \mathbf{Tick}'\\ \quad\wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname},\mathtt{activated}][\mathbf{nameGen}(\mathbf{SK},\mathtt{iname}),\mathtt{considered}]\\ \quad\wedge\ \mathbf{PC}' = \mathbf{PC}[\mathbf{nameGen}(\mathbf{SK},\mathtt{iname}),\mathbf{true}]\\ \quad\wedge\ \circ\ [\ \ \mathtt{activated\#}(\mathtt{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})\\ \qquad\|_s\ \mathtt{considered\#}(\mathbf{nameGen}(\mathtt{sk},\mathtt{iname});\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})];\end{array}$$

B.6.3 Vollständigkeit

Ein sequential-kontrollierter Plan kann stets einen Zustandsübergang durchführen, das heißt, es ist stets eine der beiden oben aufgeführten Regeln anwendbar. Dies wird durch die nachstehende Aussage nachgewiesen:

$$
\begin{aligned}
\vdash \quad & \neg\ \mathbf{hasExecutedSub}(\mathbf{AS}, \mathtt{iname}) \\
& \wedge \neg\ \mathbf{hasSelectedSub}(\mathbf{AS}, \mathtt{iname}) \\
& \wedge \mathbf{hasStartableSub}(\mathbf{AS}, \mathtt{iname}), \\
& \quad \mathbf{hasExecutedSub}(\mathbf{AS}, \mathtt{iname}) \\
& \vee \mathbf{hasSelectedSub}(\mathbf{AS}, \mathtt{iname}) \\
& \vee \neg\ \mathbf{hasStartableSub}(\mathbf{AS}, \mathtt{iname}),
\end{aligned}
$$

B.7 Zustandsübergänge der anyorder-Plankontrolle

Die anyorder-kontrollierten Pläne starten stets alle Pläne, die gestartet oder neu-gestartet werden können. Jedoch wird immer nur ein Plan gleichzeitig aktiviert, so dass beliebig viele Unterpläne in der Selection-Phase sind, jedoch nie mehr als einer in der Execution Phase. Die Auswahl des zu aktivierenden Unterplans erfolgt dabei indeterministisch bzw. durch die Umgebung gesteuert.

B.7.1 Starten von Unterplänen

Liegt ein Plan vor, der gestartet oder neu-gestartet werden kann, so wird dies getan.

$$
\frac{\begin{aligned}
& \mathbf{isAnyorder}(\mathbf{iname}) \\
\wedge\ & \mathbf{hasStartableSub}(\mathbf{HS}, \mathbf{iname}) \\
\wedge\ & \mathbf{PL} = \mathbf{selectAllStartable}(\mathbf{HS}, \mathbf{iname}) \\
\wedge\ & \mathbf{planNames}(\mathbf{PL}, \mathbf{iname}, \mathbf{sys})
\end{aligned}}{\mathbf{iname}\ \ \sigma, \mathbf{Tick}(\sigma[\mathbf{iname}, \mathtt{activated}][\mathbf{PL}, \mathbf{iname}, \mathtt{considered}])\ \ \xrightarrow{}_{\mathtt{Con}}\ \ \mathbf{iname} \|_s\ \mathbf{sys}}
$$

An diese Vorgabe hält sich auch die Implementierung:

$$
\begin{aligned}
& \mathbf{isAnyorder}(\mathtt{iname}), \\
& \mathbf{hasStartableSub}(\mathbf{AS}, \mathtt{iname}), \\
& \mathtt{strx} = \mathbf{selectAllStartable}(\mathbf{AS}, \mathtt{iname}), \\
& [\mathtt{body\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})] \\
\vdash \quad & [\mathbf{AS}, \mathbf{Tick}] \\
& \wedge \neg\ \mathbf{Tick}' \\
& \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname}, \mathtt{activated}][\mathtt{strx}, \mathtt{iname}, \mathtt{considered}] \\
& \wedge \circ\ [\quad \mathtt{activated\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC}) \\
& \qquad \|_s\ \mathtt{considered\text{-}rec\#}(\mathtt{strx}, \mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})];
\end{aligned}
$$

B.7.2 Aktivierter Unterplan

Kann kein Plan (neu) gestartet werden und gibt es einen Unterplan in der Execution-Phase bzw. einen, der bereits aktiviert wurde, so verhält sich der Plan passiv.

$$\frac{\begin{array}{l}\mathbf{isAnyorder}(\mathbf{iname})\\ \land\ \neg\ \mathbf{hasStartableSub}(\mathbf{HS},\mathbf{iname})\\ \land\ \mathbf{hasExecOrSigReadySub}(\mathbf{HS},\mathbf{iname})\end{array}}{\mathbf{iname}\ \overset{\sigma,\sigma[\mathbf{iname},\ \underset{\longrightarrow}{\text{activated}}]}{}_{\texttt{Con}}\ \mathbf{iname}}$$

Dies wird auch so von der Implementierung umgesetzt:

$$\begin{array}{l}\mathbf{isAnyorder}(\text{iname}),\\ \neg\ \mathbf{hasStartableSub}(\mathbf{AS},\text{iname}),\\ \mathbf{hasExecOrSigReadySub}(\mathbf{AS},\mathbf{PC},\text{iname}),\\ [\text{body\#}(\text{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})]\\ \vdash\ \ [\mathbf{AS}]\\ \land\ \mathbf{AS}' = \mathbf{AS}[\text{iname},\text{activated}]\\ \land\ \circ\ [\text{activated\#}(\text{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})];\end{array}$$

B.7.3 Bereiter Unterplan

Ist kein Unterplan in der `Execution`-Phase und wurde auch keiner aktiviert, so wird ein Unterplan indeterministisch bzw. nach Vorgabe der Umgebung ausgewählt und aktiviert.

$$\frac{\begin{array}{l}\mathbf{isAnyorder}(\mathbf{iname})\\ \land\ \neg\ \mathbf{hasStartableSub}(\mathbf{HS},\mathbf{iname})\\ \land\ \neg\ \mathbf{hasExecOrSigReadySub}(\mathbf{HS},\mathbf{iname})\\ \land\ \mathbf{hasReadySub}(\mathbf{HS},\mathbf{iname})\\ \land\ \mathbf{iname}_1 = \mathbf{nameGen}(\mathbf{selectPrivilegedSub}(\mathbf{HS},\mathbf{iname}),\mathbf{iname})\end{array}}{\mathbf{iname}\ \overset{\sigma,\mathbf{Tick}(\sigma[\mathbf{iname},\ \underset{\longrightarrow}{\text{activated}}][\mathbf{iname}_1]_{ac})}{}_{\texttt{Con}}\ \mathbf{iname}}$$

Die Implementierung zeigt dazu analoges Verhalten:

$$\begin{array}{l}\mathbf{isAnyorder}(\text{iname}),\\ \neg\ \mathbf{hasStartableSub}(\mathbf{AS},\text{iname}),\\ \neg\ \mathbf{hasExecOrSigReadySub}(\mathbf{AS},\mathbf{PC},\text{iname}),\\ \mathbf{hasReadySub}(\mathbf{AS},\text{iname}),\\ \text{sk} = \mathbf{selectPrivilegedSub}(\mathbf{AS},\text{iname}),\\ [\text{body\#}(\text{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})]\\ \vdash\ \ [\mathbf{AS},\ \mathbf{PC},\ \mathbf{Tick}]\\ \land\ \neg\ \mathbf{Tick}'\\ \land\ \mathbf{PC}' = \mathbf{PC}[\mathbf{nameGen}(\text{sk},\text{iname}),\mathbf{true}]\\ \land\ \mathbf{AS}' = \mathbf{AS}[\text{iname},\text{activated}]\\ \land\ \circ\ [\text{activated\#}(\text{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})];\end{array}$$

B.7.4 Kein Unterplan

Existiert überhaupt kein nicht-terminierter Unterplan, so verhält sich der Plan passiv.

$$\frac{\begin{array}{l}\mathbf{isAnyorder}(\mathbf{iname})\\ \wedge\, \neg\ \mathbf{hasStartableSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\, \neg\ \mathbf{hasExecutedSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\, \neg\ \mathbf{hasReadySub}(\mathbf{HS},\mathbf{iname})\end{array}}{\mathbf{iname}\ \ \sigma,\sigma[\mathbf{iname},\ \mathtt{activated}]_{\mathtt{Con}}\ \ \mathbf{iname}}\ \longrightarrow$$

Entsprechend agiert die Implementierung:

$$\begin{array}{l}\mathbf{isAnyorder}(\mathtt{iname}),\\ \neg\ \mathbf{hasStartableSub}(\mathbf{AS},\mathtt{iname}),\\ \neg\ \mathbf{hasExecOrSigReadySub}(\mathbf{AS},\mathbf{PC},\mathtt{iname}),\\ \neg\ \mathbf{hasReadySub}(\mathbf{AS},\mathtt{iname}),\\ [\mathtt{body\#(iname;}\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC}\mathtt{)}]\\ \vdash\ \ [\mathbf{AS}]\\ \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname},\mathtt{activated}]\\ \wedge\ \circ\ [\mathtt{activated\#(iname;}\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC}\mathtt{)}];\end{array}$$

B.7.5 Vollständigkeit

Auch die Regeln zur Ausführung von anyorder Plänen sind komplett:

$$\begin{array}{l}\vdash\ \ \neg\ \mathbf{hasStartableSub}(\mathbf{AS},\mathtt{iname})\\ \ \ \wedge\, \neg\ \mathbf{hasExecutedSub}(\mathbf{AS},\mathtt{iname})\\ \ \ \wedge\, \neg\ \mathbf{hasReadySub}(\mathbf{AS},\mathtt{iname})\\ \ \ \ \ \neg\ \mathbf{hasStartableSub}(\mathbf{AS},\mathtt{iname})\\ \ \ \wedge\, \neg\ \mathbf{hasExecOrSigReadySub}(\mathbf{AS},\mathtt{iname})\\ \ \ \wedge\, \mathbf{hasReadySub}(\mathbf{AS},\mathtt{iname})\\ \ \ \ \ \neg\ \mathbf{hasStartableSub}(\mathbf{AS},\mathtt{iname})\\ \ \ \wedge\, \mathbf{hasExecOrSigReadySub}(\mathbf{AS},\mathtt{iname})\\ \ \ \mathbf{hasStartableSub}(\mathbf{AS},\mathtt{iname})\end{array}$$

B.8 Zustandsübergänge der parallel-Plankontrolle

Ein parallel-kontrollierter Plan startet seine Unterpläne alle gleichzeitig. Erst wenn alle Unterpläne entweder zurückgewiesen wurden oder im Zustand ready sind, werden alle noch laufenden Unterpläne gleichzeitig aktiviert. Nach der Aktivierung der Unterpläne verhält sich die Plankontrolle passiv.

B.8.1 Laufende Pläne

Wurden die Unterpläne bereits aktiviert bzw. sind diese in der Execution-Phase, so verhält sich der Plan passiv.

$$\frac{\begin{array}{l}\mathbf{isParallel}(\mathbf{iname})\\ \quad \wedge\ \mathbf{hasExecOrSigReadySub}(\mathbf{HS}, \mathbf{iname})\end{array}}{\mathbf{iname}\ \overset{\sigma, \sigma[\mathbf{iname},\ \mathtt{activated}]}{\longrightarrow}_{\mathtt{Con}}\ \mathbf{iname}}$$

Dies wird auch durch die Implementierung entsprechend umgesetzt:

$$\begin{array}{l}\quad \mathbf{isParallel}(\mathtt{iname}),\\ \quad \mathbf{hasExecOrSigReadySub}(\mathbf{AS}, \mathbf{PC}, \mathtt{iname}),\\ \quad [\mathtt{body\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})]\\ \vdash\ \ [\mathbf{AS}]\\ \quad \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname}, \mathtt{activated}]\\ \quad \wedge\ \circ\ [\mathtt{activated\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})];\end{array}$$

B.8.2 Aktivierung der Unterpläne

Sind die Pläne noch nicht aktiviert worden, jedoch alle Pläne im Zustand `ready` oder `rejected`, so werden die noch bereiten Pläne aktiviert.

$$\frac{\begin{array}{l}\mathbf{isParallel}(\mathbf{iname})\\ \quad \wedge\ \neg\ \mathbf{hasExecOrSigReadySub}(\mathbf{HS}, \mathbf{iname})\\ \quad \wedge\ \mathbf{allSubsReadyOrRejected}(\mathbf{HS}, \mathbf{iname})\\ \quad \wedge\ \mathbf{PL} = \mathbf{selectAllReady}(\mathbf{HS}, \mathbf{iname})\\ \quad \wedge\ \mathbf{hasSelectedSub}(\mathbf{HS}, \mathbf{iname})\end{array}}{\mathbf{iname}\ \overset{\sigma, \mathbf{Tick}(\sigma[\mathbf{iname},\ \mathtt{activated}][\mathbf{PL}, \mathbf{iname}]_{ac})}{\longrightarrow}_{\mathtt{Con}}\ \mathbf{iname}}$$

Dieses Verhalten wird auch durch die Implementierung abgebildet, wie nachstehende Beweisverpflichtung nachweist:

$$\begin{array}{l}\quad \mathbf{isParallel}(\mathtt{iname}),\\ \quad \neg\ \mathbf{hasExecOrSigReadySub}(\mathbf{AS}, \mathbf{PC}, \mathtt{iname}),\\ \quad \mathbf{allSubsReadyOrRejected}(\mathbf{AS}, \mathtt{iname}),\\ \quad \mathbf{hasSelectedSub}(\mathbf{AS}, \mathtt{iname}),\\ \quad \mathtt{strx} = \mathbf{selectAllReady}(\mathbf{AS}, \mathtt{iname}),\\ \quad [\mathtt{body\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})]\\ \vdash\ \ [\mathbf{AS},\ \mathbf{PC},\ \mathbf{Tick}]\\ \quad \wedge\ \neg\ \mathbf{Tick}'\\ \quad \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname}, \mathtt{activated}]\\ \quad \wedge\ \mathbf{PC}' = \mathbf{PC}[\mathtt{strx}, \mathtt{iname}, \mathbf{true}]\\ \quad \wedge\ \circ\ [\mathtt{activated\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})];\end{array}$$

B.8.3 Abwarten der Bereitschaft der Unterpläne

Sind die Unterpläne eines `parallel`-kontrollierten Plans bereits gestartet worden und haben einige der Unterpläne noch nicht den Zustand `ready` erreicht, so wartet der Plan passiv ab.

$$\frac{\begin{array}{l}\mathbf{isParallel}(\mathbf{iname})\\ \wedge\ \neg\ \mathbf{hasExecutedSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \neg\ \mathbf{allSubsReadyOrRejected}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \mathbf{hasSelectedSub}(\mathbf{HS},\mathbf{iname})\end{array}}{\mathbf{iname}\ \ {}^{\sigma,\sigma[\mathbf{iname},\ \mathtt{activated}]}\!\!\longrightarrow_{\mathtt{Con}}\ \ \mathbf{iname}}$$

Die Implementierung hält sich an diese Vorgabe der Spezifikation:

$$\begin{array}{l}\mathbf{isParallel}(\mathtt{iname}),\\ \neg\ \mathbf{hasExecOrSigReadySub}(\mathbf{AS},\mathbf{PC},\mathtt{iname}),\\ \neg\ \mathbf{allSubsReadyOrRejected}(\mathbf{AS},\mathtt{iname}),\\ \mathbf{hasSelectedSub}(\mathbf{AS},\mathtt{iname}),\\ [\mathtt{body\#}(\mathtt{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})]\\ \vdash\ \ \lceil\mathbf{AS}\rceil\\ \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname},\mathtt{activated}]\\ \wedge\ \circ\ [\mathtt{activated\#}(\mathtt{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})];\end{array}$$

B.8.4 Starten der Unterpläne

Wenn die Unterpläne noch nicht gestartet wurden, dann wird der `parallel` Plan alle Unterpläne gleichzeitig starten.

$$\frac{\begin{array}{l}\mathbf{isParallel}(\mathbf{iname})\\ \wedge\ \neg\ \mathbf{hasExecutedSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \neg\ \mathbf{hasSelectedSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \mathbf{hasInactiveSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \mathbf{PL} = \mathtt{asbru}(\mathbf{nameSel}(\mathbf{iname}))\mathbf{.subplans}\\ \wedge\ \mathbf{planNames}(\mathbf{PL},\mathbf{iname},\mathbf{sys})\end{array}}{\mathbf{iname}\ \ {}^{\sigma,\mathbf{Tick}(\sigma[\mathbf{iname},\ \mathtt{activated}][\mathbf{PL},\ \mathbf{iname},\ \mathtt{considered}])}\!\!\longrightarrow_{\mathtt{Con}}\ \ \mathbf{iname}\|_s\ \mathbf{sys}}$$

Die Implementierung verhält sich genauso:

$$\begin{array}{l}\mathbf{isParallel}(\mathtt{iname}),\\ \neg\ \mathbf{hasExecutedSub}(\mathbf{AS},\mathbf{PC},\mathtt{iname}),\\ \neg\ \mathbf{hasSelectedSub}(\mathbf{AS},\mathtt{iname}),\\ \mathbf{hasInactiveSub}(\mathbf{AS},\mathtt{iname}),\\ \mathtt{strx} = \mathtt{asbru}(\mathbf{nameSel}(\mathtt{iname}))\mathbf{.subplans},\\ [\mathtt{body\#}(\mathtt{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})]\\ \vdash\ \ \lceil\mathbf{AS},\ \mathbf{Tick}\rceil\\ \wedge\ \neg\ \mathbf{Tick}'\\ \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname},\mathtt{activated}][\mathtt{strx},\mathtt{iname},\mathtt{considered}]\\ \wedge\ \circ\ [\quad \mathtt{activated\#}(\mathtt{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})\\ \qquad \|_s\ \mathtt{considered\text{-}rec\#}(\mathtt{strx},\mathtt{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})];\end{array}$$

B.8.5 Kein Unterplan

Wenn überhaupt kein Unterplan mehr im Zustand `inactive` oder in `Execution`-Phase bzw. `Selection`-Phase vorhanden ist, dann verhält sich der Plan passiv.

$$\frac{\begin{array}{l}\mathbf{isParallel}(\mathbf{iname})\\ \wedge\ \neg\ \mathbf{hasExecutedSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \neg\ \mathbf{hasSelectedSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \neg\ \mathbf{hasInactiveSub}(\mathbf{HS},\mathbf{iname})\end{array}}{\mathbf{iname}\ \ {}^{\sigma,\sigma[\mathbf{iname},\ \mathtt{activated}]}_{\longrightarrow}{}_{\mathtt{Con}}\ \ \mathbf{iname}}$$

Unter diesen Bedingungen verhält sich auch die Implementierung passiv:

$$\begin{array}{l}\mathbf{isParallel}(\mathtt{iname}),\\ \neg\ \mathbf{hasExecutedSub}(\mathbf{AS},\mathbf{PC},\mathtt{iname}),\\ \neg\ \mathbf{hasSelectedSub}(\mathbf{AS},\mathtt{iname}),\\ \neg\ \mathbf{hasInactiveSub}(\mathbf{AS},\mathtt{iname}),\\ [\mathtt{body\#}(\mathtt{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})]\\ \vdash\quad [\mathbf{AS}]\\ \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname},\mathtt{activated}]\\ \wedge\ \circ\ [\mathtt{activated\#}(\mathtt{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})];\end{array}$$

B.8.6 Vollständigkeit

Die Regeln, die die `parallel`-Kontrolle beschreiben sind in den Vorbedingungen vollständig, decken also den kompletten Zustandsraum ab. Damit kann ein `parallel`-kontrollierter Plan in jedem Fall einen Schritt ausführen.

$$\begin{array}{rl}\vdash & \neg\ \mathbf{hasExecutedSub}(\mathbf{AS},\mathtt{iname})\\ & \wedge\ \neg\ \mathbf{hasSelectedSub}(\mathbf{AS},\mathtt{iname})\\ & \wedge\ \neg\ \mathbf{hasInactiveSub}(\mathbf{AS},\mathtt{iname}),\\ & \quad \neg\ \mathbf{hasExecutedSub}(\mathbf{AS},\mathtt{iname})\\ & \wedge\ \neg\ \mathbf{hasSelectedSub}(\mathbf{AS},\mathtt{iname})\\ & \wedge\ \mathbf{hasInactiveSub}(\mathbf{AS},\mathtt{iname}),\\ & \quad \neg\ \mathbf{hasExecOrSigReadySub}(\mathbf{AS},\mathtt{iname})\\ & \wedge\ \neg\ \mathbf{allSubsReadyOrRejected}(\mathbf{AS},\mathtt{iname})\\ & \wedge\ \mathbf{hasSelectedSub}(\mathbf{AS},\mathtt{iname}),\\ & \quad \neg\ \mathbf{hasExecOrSigReadySub}(\mathbf{AS},\mathtt{iname})\\ & \wedge\ \mathbf{allSubsReadyOrRejected}(\mathbf{AS},\mathtt{iname})\\ & \wedge\ \mathbf{hasSelectedSub}(\mathbf{AS},\mathtt{iname}),\\ & \mathbf{hasExecOrSigReadySub}(\mathbf{AS},\mathtt{iname})\end{array}$$

B.9 Zustandsübergänge der `unordered`-Plankontrolle

Die `unordered`-Kontrolle kennt bei der Ausführung zwei Fälle. Den Einen, in dem ein startbarer bzw. neu-startbarer Plan vorliegt, den Anderen, in dem kein Plan zu starten ist. Im ersten Fall werden alle startbaren Unterpläne gleichzeitig aktiviert und gestartet, im zweiten Fall verhält sich der Plan passiv.

B.9.1 Starten von Unterplänen

Sind startbare Unterpläne vorhanden, so werden diese gestartet und das Aktivierungssignal dieser Pläne wird gesetzt.

$$\frac{\begin{array}{l}\mathbf{isUnordered}(\mathbf{iname})\\ \wedge\ \mathbf{PL} = \mathbf{selectAllStartable}(\mathbf{HS}, \mathbf{iname})\\ \wedge\ \mathbf{planNames}(\mathbf{PL}, \mathbf{iname}, \mathbf{sys})\\ \wedge\ \mathbf{hasStartableSub}(\mathbf{HS}, \mathbf{iname})\end{array}}{\mathbf{iname}\ \ {}^{\sigma,\mathbf{Tick}(\sigma[\mathbf{iname},\ \mathrm{activated}][\mathbf{PL},\ \mathbf{iname},\ \mathrm{considered}][\mathbf{PL},\mathbf{iname}]_{ac})}_{\mathsf{Con}}\ \ \mathbf{iname} \|_s\ \mathbf{sys}}$$

Dieses Verhalten wird von der Implementierung nachgebildet:

$$\begin{array}{l}\quad \mathbf{isUnordered}(\mathrm{iname}),\\ \quad \mathbf{hasStartableSub}(\mathbf{AS}, \mathrm{iname}),\\ \quad \mathrm{strx} = \mathbf{selectAllStartable}(\mathbf{AS}, \mathrm{iname}),\\ \quad [\mathrm{body\#}(\mathrm{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})]\\ \vdash\ \ \lceil \mathbf{AS},\ \mathbf{Tick},\ \mathbf{PC} \rceil\\ \quad \wedge\ \neg\ \mathbf{Tick}'\\ \quad \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathrm{iname}, \mathrm{activated}][\mathrm{strx}, \mathrm{iname}, \mathrm{considered}]\\ \quad \wedge\ \mathbf{PC}' = \mathbf{PC}[\mathrm{strx}, \mathrm{iname}, \mathbf{true}]\\ \quad \wedge\ \circ\ [\ \ \mathrm{activated\#}(\mathrm{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})\\ \qquad \|_s\ \mathrm{considered\text{-}rec\#}(\mathrm{strx}, \mathrm{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})];\end{array}$$

B.9.2 Kein startbarer Unterplan

Ist kein Unterplan vorhanden, der gestartet werden kann, so verhält sich der Plan passiv.

$$\frac{\begin{array}{l}\mathbf{isUnordered}(\mathbf{iname})\\ \wedge\ \neg\ \mathbf{hasStartableSub}(\mathbf{HS}, \mathbf{iname})\end{array}}{\mathbf{iname}\ \ {}^{\sigma,\sigma[\mathbf{iname},\ \mathrm{activated}]}_{\mathsf{Con}}\ \ \mathbf{iname}}$$

Dies wird entsprechend von der Implementierung umgesetzt:

$$\begin{array}{l}\quad \mathbf{isUnordered}(\mathrm{iname}),\\ \quad \neg\ \mathbf{hasStartableSub}(\mathbf{AS}, \mathrm{iname}),\\ \quad [\mathrm{body\#}(\mathrm{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})]\\ \vdash\ \ \lceil \mathbf{AS} \rceil\\ \quad \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathrm{iname}, \mathrm{activated}]\\ \quad \wedge\ \circ\ [\mathrm{activated\#}(\mathrm{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})];\end{array}$$

B.9.3 Vollständigkeit

Auch für die `unordered`-Kontrolle kann bewiesen werden, dass die Vorbedingungen universell sind und jeder `unordered` Plan stets eine Handlung ausführen kann.

$$\begin{array}{l}\vdash \mathbf{hasStartableSub}(\mathbf{AS}, \mathtt{iname}),\\ \quad \neg\ \mathbf{hasStartableSub}(\mathbf{AS}, \mathtt{iname})\end{array}$$

B.10 Zustandsübergänge der `onabort`-Plankontrolle

Der `onabort` Plantyp startet zunächst seinen ersten Unterplan. Bricht dieser die Ausführung ab, so wird der zweite Unterplan gestartet.

B.10.1 Starten des ersten Unterplans

Wurde der erste Unterplan noch nicht gestartet, so soll dies geschehen. Außerdem soll der Unterplan auch gleich aktiviert werden.

$$\frac{\begin{array}{l}\mathbf{isOnAbort}(\mathbf{iname})\\ \wedge\ \mathbf{isInactive}(\mathbf{HS}, \mathbf{iname}_1)\\ \wedge\ \mathbf{iname}_1 = \mathbf{nameGen}(\mathbf{selectFirstSub}(\mathbf{iname}), \mathbf{iname})\end{array}}{\mathbf{iname} \xrightarrow{\sigma, \mathtt{Tick}(\sigma[\mathbf{iname},\ \mathtt{activated}][\mathbf{iname}_1,\ \mathtt{considered}][\mathbf{iname}_1]_{ac})}_{\mathtt{Con}} \mathbf{iname} \|_s \mathbf{iname}_1}$$

Die Implementierung hält sich an diese Vorgabe:

$$\begin{array}{l}\quad \mathbf{isOnAbort}(\mathtt{iname}),\\ \quad \mathbf{isInactive}(\mathbf{AS}, \mathbf{nameGen}(\mathtt{sk}, \mathtt{iname})),\\ \quad \mathtt{sk} = \mathbf{selectFirstSub}(\mathtt{iname}),\\ \quad [\mathtt{body\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})]\\ \vdash\ \ [\mathbf{AS},\ \mathbf{Tick},\ \mathbf{PC}]\\ \quad \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname}, \mathtt{activated}][\mathbf{nameGen}(\mathtt{sk}, \mathtt{iname}), \mathtt{considered}]\\ \quad \wedge\ \mathbf{PC}' = \mathbf{PC}[\mathbf{nameGen}(\mathtt{sk}, \mathtt{iname}), \mathbf{true}]\\ \quad \wedge\ \neg\ \mathbf{Tick}\\ \quad \wedge\ \circ\ [\quad \mathtt{activated\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})\\ \qquad\quad \|_s\ \mathtt{considered\#}(\mathbf{nameGen}(\mathtt{sk}, \mathtt{iname}); \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})];\end{array}$$

B.10.2 Starten des zweiten Unterplans

Wird der erste Unterplan abgebrochen, so soll der zweite Unterplan gestartet bzw. neugestartet werden, sofern dies möglich ist.

$$\frac{\begin{array}{l}\mathbf{isOnAbort}(\mathbf{iname})\\ \wedge\ \mathbf{isAborted}(\mathbf{HS},\mathbf{nameGen}(\mathbf{selectFirstSub}(\mathbf{iname}),\mathbf{iname}))\\ \wedge\ \mathbf{isStartable}(\mathbf{HS},\mathbf{nameGen}(\mathbf{selectSecondSub}(\mathbf{iname}),\mathbf{iname}))\\ \wedge\ \mathbf{iname}_1 = \mathbf{nameGen}(\mathbf{selectSecondSub}(\mathbf{iname}),\mathbf{iname})\end{array}}{\mathbf{iname}\ \overset{\sigma,\mathbf{Tick}(\sigma[\mathbf{iname},\ \mathtt{activated}][\mathbf{iname}_1,\ \mathtt{considered}][\mathbf{iname}_1]_{ac})}{\longrightarrow}_{\mathtt{Con}}\ \mathbf{iname}\|_s\ \mathbf{iname}_1}$$

Die Implementierung setzt dies um, wie durch nachstehende Aussage bewiesen wurde:

$$\begin{array}{l}\quad \mathbf{isOnAbort}(\mathtt{iname}),\\ \quad \mathbf{isAborted}(\mathbf{AS},\mathbf{nameGen}(\mathbf{selectFirstSub}(\mathtt{iname}),\mathtt{iname})),\\ \quad \mathbf{isStartable}(\mathbf{nameGen}(\mathtt{sk},\mathbf{iname}),\\ \quad \mathtt{sk} = \mathbf{selectSecondSub}(\mathtt{iname}),\\ \quad [\mathtt{body\#}(\mathtt{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})]\\ \vdash\ \ \lceil\mathbf{AS},\ \mathbf{Tick},\ \mathbf{PC}\rceil\\ \quad \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname},\mathtt{activated}][\mathbf{nameGen}(\mathtt{sk},\mathtt{iname}),\mathtt{considered}]\\ \quad \wedge\ \mathbf{PC}' = \mathbf{PC}[\mathbf{nameGen}(\mathtt{sk},\mathtt{iname}),\mathbf{true}]\\ \quad \wedge\ \neg\ \mathbf{Tick}\\ \quad \wedge\ \circ\ [\ \ \mathtt{activated\#}(\mathtt{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})\\ \qquad \|_s\ \mathtt{considered\#}(\mathbf{nameGen}(\mathtt{sk},\mathtt{iname});\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})];\end{array}$$

B.10.3 Kein startbarer Unterplan

Ist der erste Unterplan nicht im Zustand `aborted` und nicht im Zustand `inactive` so kann kein Unterplan gestartet werden. Die selbe Situation ergibt sich, wenn der zweite Unterplan bereits ausgeführt wird und nicht in einem startbaren Zustand ist.

$$\frac{\begin{array}{l}\mathbf{isOnAbort}(\mathbf{iname})\\ \wedge\ \neg\ \mathbf{isInactive}(\mathbf{HS},\mathbf{nameGen}(\mathbf{selectFirstSub}(\mathbf{iname}),\mathbf{iname}))\\ \wedge\ \ \ \neg\ \mathbf{isAborted}(\mathbf{HS},\mathbf{nameGen}(\mathbf{selectFirstSub}(\mathbf{iname}),\mathbf{iname}))\\ \ \ \vee\ \neg\ \mathbf{isStartable}(\mathbf{HS},\mathbf{nameGen}(\mathbf{selectSecondSub}(\mathbf{iname}),\mathbf{iname}))\end{array}}{\mathbf{iname}\ \overset{\sigma,\sigma[\mathbf{iname},\ \mathtt{activated}]}{\longrightarrow}_{\mathtt{Con}}\ \mathbf{iname}}$$

Die Implementierung setzt diesen Teil der Spezifikation um, wie bewiesen wurde:

$$\begin{array}{l}\quad \mathbf{isOnAbort}(\mathtt{iname}),\\ \quad \neg\ \mathbf{isInactive}(\mathbf{AS},\mathbf{nameGen}(\mathbf{selectFirstSub}(\mathtt{iname}),\mathtt{iname})),\\ \qquad \neg\ \mathbf{isAborted}(\mathbf{AS},\mathbf{nameGen}(\mathbf{selectFirstSub}(\mathtt{iname}),\mathtt{iname}))\\ \quad \vee\ \neg\ \mathbf{isStartable}(\mathbf{AS},\mathbf{nameGen}(\mathbf{selectSecondSub}(\mathtt{iname}),\mathtt{iname})),\\ \quad [\mathtt{body\#}(\mathtt{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})]\\ \vdash\ \ \lceil\mathbf{AS}\rceil\\ \quad \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname},\mathtt{activated}]\\ \quad \wedge\ \circ\ [\mathtt{activated\#}(\mathtt{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})]\end{array}$$

B.10.4 Vollständigkeit

Auch die Spezifikation der `unordered`-Kontrolle ist komplett in dem Sinn, dass jeder `unordered`-kontrollierte Plan stets eine Handlung durchführen kann:

$$
\begin{aligned}
\vdash\ & \mathbf{isInactive}(\mathbf{AS}, \mathbf{nameGen}(\mathbf{selectFirstSub}(\mathtt{iname}), \mathtt{iname})),\\
& \quad \mathbf{isAborted}(\mathbf{AS}, \mathbf{nameGen}(\mathbf{selectFirstSub}(\mathtt{iname}), \mathtt{iname}))\\
& \wedge\ \mathbf{isStartable}(\mathbf{AS}, \mathbf{nameGen}(\mathbf{selectSecondSub}(\mathtt{iname}), \mathtt{iname})),\\
& \quad \neg\ \mathbf{isInactive}(\mathbf{AS}, \mathbf{nameGen}(\mathbf{selectFirstSub}(\mathtt{iname}), \mathtt{iname}))\\
& \wedge\ \quad \neg\ \mathbf{isAborted}(\mathbf{AS}, \mathbf{nameGen}(\mathbf{selectFirstSub}(\mathtt{iname}), \mathtt{iname}))\\
& \quad \vee \neg\ \mathbf{isStartable}(\mathbf{AS}, \mathbf{nameGen}(\mathbf{selectSecondSub}(\mathtt{iname}), \mathtt{iname})),
\end{aligned}
$$

B.11 Zustandsübergänge der `ifThenElse`-Plankontrolle

Der `ifThenElse` Plankontrolltyp soll bei seinem ersten Aufruf überprüfen, ob das Konditional des PLankontrolltyps wahr oder falsch ist. Im ersten Fall soll der erste Unterplan gestartet werden, im zweiten Fall soll, soweit vorhanden, der zweite Unterplan gestartet werden. Ist kein zweiter Unterplan vorhanden und das Konditional wertet zu falsch aus, dann wird sich der Plan sofort beenden.

B.11.1 Positive Auswertung des Konditionals

Wird das Konditional positiv ausgewertet, wenn gleichzeitig alle Unterpläne noch im Zustand `inactive` sind, dann soll der erste Unterplan aktiviert und gestartet werden.

$$
\frac{\begin{array}{l}\mathbf{isITE}(\mathbf{iname})\\ \wedge\ \mathbf{allSubsInactive}(\mathbf{HS}, \mathbf{iname})\\ \wedge\ \mathbf{ITEP}(\sigma, \mathbf{iname})\\ \wedge\ \mathbf{iname}_1 = \mathbf{nameGen}(\mathbf{selectFirstSub}(\mathbf{iname}), \mathbf{iname})\end{array}}{\mathbf{iname}\ \overset{\sigma, \mathbf{Tick}(\sigma[\mathbf{iname},\ \mathtt{activated}][\mathbf{iname}_1,\ \mathtt{considered}][\mathbf{iname}_1]_{ac})}{\longrightarrow}_{\mathtt{Con}}\ \mathbf{iname} \|_s\ \mathbf{iname}_1}
$$

Die Implementierung setzt diese Bedingung um:

$$
\begin{aligned}
& \mathbf{isITE}(\mathtt{iname}),\\
& \mathbf{allSubsInactive}(\mathbf{AS}, \mathtt{iname}),\\
& \mathbf{ITEP}(\mathbf{PDH}, \mathbf{AC}),\\
& \mathtt{sk} = \mathbf{selectFirstSub}(\mathtt{iname}),\\
& [\mathtt{body\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})]\\
\vdash\ & \quad [\mathbf{AS},\ \mathbf{Tick},\ \mathbf{PC}]\\
& \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname}, \mathtt{activated}][\mathbf{nameGen}(\mathtt{sk}, \mathtt{iname}), \mathtt{considered}]\\
& \wedge\ \mathbf{PC}' = \mathbf{PC}[\mathbf{nameGen}(\mathtt{sk}, \mathtt{iname}), \mathbf{true}]\\
& \wedge\ \neg\ \mathbf{Tick}\\
& \wedge\ \circ\ [\quad \mathtt{activated\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})\\
& \qquad \|_s\ \mathtt{considered\#}(\mathbf{nameGen}(\mathtt{sk}, \mathtt{iname}); \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})];
\end{aligned}
$$

B.11.2 Negative Auswertung des Konditionals mit zweitem Unterplan

Wird das Konditional zu falsch ausgewertet, so soll der zweite Unterplan gestartet werden, sofern dieser existiert:

$$\frac{\begin{array}{l}\quad \mathbf{isITE}(\mathbf{iname}) \\ \wedge\ \mathbf{allSubsInactive}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \mathbf{secondSubPresent}(\mathbf{iname}) \\ \wedge\ \neg\ \mathbf{ITEP}(\sigma, \mathbf{iname}) \\ \wedge\ \mathbf{iname}_1 = \mathbf{nameGen}(\mathbf{selectSecondSub}(\mathbf{iname}), \mathbf{iname})\end{array}}{\mathbf{iname}\ \overset{\sigma, \mathbf{Tick}(\sigma[\mathbf{iname},\ \mathtt{activated}][\mathbf{iname}_1,\ \mathtt{considered}][\mathbf{iname}_1]_{ac})}{\longrightarrow}_{\mathtt{Con}}\ \mathbf{iname} \|_s \mathbf{iname}_1}$$

Es kann nachgewiesen werden, dass die Implementierung diese Anforderung erfüllt:

$$\begin{array}{l}\quad \mathbf{isITE}(\mathtt{iname}), \\ \quad \mathbf{allSubsInactive}(\mathbf{AS}, \mathtt{iname}), \\ \quad \neg\ \mathbf{ITEP}(\mathbf{PDH}, \mathbf{AC}), \\ \quad \mathbf{secondSubPresent}(\mathtt{iname}), \\ \quad \mathtt{sk} = \mathbf{selectSecondSub}(\mathtt{iname}), \\ \quad [\mathtt{body\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})] \\ \vdash\ \ \lceil \mathbf{AS},\ \mathbf{Tick},\ \mathbf{PC} \rceil \\ \quad \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname}, \mathtt{activated}][\mathbf{nameGen}(\mathtt{sk}, \mathtt{iname}), \mathtt{considered}] \\ \quad \wedge\ \mathbf{PC}' = \mathbf{PC}[\mathbf{nameGen}(\mathtt{sk}, \mathtt{iname}), \mathbf{true}] \\ \quad \wedge\ \neg\ \mathbf{Tick} \\ \quad \wedge\ \circ\ [\ \ \mathtt{activated\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC}) \\ \qquad \|_s\ \mathtt{considered\#}(\mathbf{nameGen}(\mathtt{sk}, \mathtt{iname}); \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})];\end{array}$$

B.11.3 Negative Auswertung des Konditionals ohne zweiten Unterplan

Wird das Konditional negativ ausgewertet während gleichzeitig kein zweiter Unterplan existiert, so soll der Plan sich beenden.

$$\frac{\begin{array}{l}\quad \mathbf{isITE}(\mathbf{iname}) \\ \wedge\ \mathbf{allSubsInactive}(\mathbf{HS}, \mathbf{iname}) \\ \wedge\ \neg\ \mathbf{ITEP}(\sigma, \mathbf{iname}) \\ \wedge\ \mathbf{noSecondSubPresent}(\mathbf{iname})\end{array}}{\mathbf{iname}\ \overset{\sigma, \mathbf{Tick}(\sigma[\mathbf{iname},\ \mathtt{completed}])}{\longrightarrow}_{\mathtt{Con}}\ \varepsilon}$$

Auch die Implementierung beendet die Ausführung des Plans unter diesen Umständen:

$$
\begin{array}{l}
\quad \mathbf{isITE}(\mathtt{iname}),\\
\quad \mathbf{allSubsInactive}(\mathbf{AS}, \mathtt{iname}),\\
\quad \neg\ \mathbf{ITEP}(\mathbf{PDH}, \mathbf{AC}),\\
\quad \mathbf{noSecondSubPresent}(\mathtt{iname}),\\
\quad [\mathtt{body\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})]\\
\vdash \quad [\mathbf{AS},\ \mathbf{Tick}]\\
\quad \wedge \neg\ \mathbf{Tick}'\\
\quad \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname}, \mathtt{completed}]\\
\quad \wedge \circ\ \mathbf{last};
\end{array}
$$

B.11.4 Aktivierter Unterplan

Ist bereits ein Unterpan gestartet worden, so soll die `ifThenElse` Kontrolle passiv abwarten, solange dieser Plan noch aktiv ist:

$$
\frac{\begin{array}{l}
\quad \mathbf{isITE}(\mathbf{iname})\\
\wedge \neg\ \mathbf{allSubsInactive}(\mathbf{HS}, \mathbf{iname})\\
\wedge\ \ \mathbf{hasExecutedSub}(\mathbf{HS}, \mathbf{iname})\\
\quad \vee\ \mathbf{hasSelectedSub}(\mathbf{HS}, \mathbf{iname})\\
\quad \vee \neg\ \mathbf{hasRestartableSub}(\mathbf{HS}, \mathbf{iname})
\end{array}}{\mathbf{iname}\ \ {}^{\sigma, \sigma[\mathbf{iname},\ \mathtt{activated}]}_{\longrightarrow}\ {}_{\mathtt{Con}}\ \ \mathbf{iname}}
$$

Analog verhält sich die Implementierung:

$$
\begin{array}{l}
\quad \mathbf{isITE}(\mathtt{iname}),\\
\quad \neg\ \mathbf{allSubsInactive}(\mathbf{AS}, \mathtt{iname}),\\
\quad\ \ \mathbf{hasExecutedSub}(\mathbf{AS}, \mathtt{iname})\\
\quad \vee\ \mathbf{hasSelectedSub}(\mathbf{AS}, \mathtt{iname})\\
\quad \vee \neg\ \mathbf{hasRestartableSub}(\mathbf{AS}, \mathtt{iname}),\\
\quad [\mathtt{body\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})]\\
\vdash \quad [\mathbf{AS}]\\
\quad \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname}, \mathtt{activated}]\\
\quad \wedge \circ\ [\mathtt{activated\#}(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})]
\end{array}
$$

B.11.5 Neustart eines Unterplans

Wurde ein gestarteter Unterplan der `ifThenElse`-Kontrolle abgebrochen und ist gleichzeitig das Signal zum Neustart des Plans gesetzt, so soll der abgebrochene Plan neu gestartet und neu aktiviert werden:

$$\frac{\begin{array}{l}\quad \mathbf{isITE}(\mathbf{iname})\\ \wedge\ \neg\ \mathbf{allSubsInactive}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \neg\ \mathbf{hasExecutedSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \neg\ \mathbf{hasSelectedSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \mathbf{hasRestartableSub}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \mathbf{iname}_1 = \mathbf{nameGen}(\mathbf{selectFirstRestartable}(\mathbf{HS},\mathbf{iname}),\mathbf{iname})\end{array}}{\mathbf{iname}\ \ \sigma,\mathbf{Tick}(\sigma[\mathbf{iname},\ \mathtt{activated}\xrightarrow{}][\mathbf{iname}_1,\ \mathtt{considered}])_{\mathtt{Con}}\ \ \mathbf{iname}\|_s\ \mathbf{iname}_1}$$

Dies wird auch durch die Implementierung befolgt:

$$\begin{array}{l}\quad \mathbf{isITE}(\mathtt{iname}),\\ \quad \neg\ \mathbf{allSubsInactive}(\mathbf{AS},\mathtt{iname})\\ \quad \neg\ \mathbf{hasExecutedSub}(\mathbf{AS},\mathtt{iname})\\ \quad \neg\ \mathbf{hasSelectedSub}(\mathbf{AS},\mathtt{iname})\\ \quad \mathbf{hasRestartableSub}(\mathbf{AS},\mathtt{iname}),\\ \quad \mathtt{sk} = \mathbf{selectFirstRestartable}(\mathbf{AS},\mathtt{iname}),\\ \quad [\mathtt{body\#}(\mathtt{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})]\\ \vdash\quad [\mathbf{AS},\ \mathbf{Tick},\ \mathbf{PC}]\\ \quad \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname},\mathtt{activated}][\mathbf{nameGen}(\mathtt{sk},\mathtt{iname}),\mathtt{considered}]\\ \quad \wedge\ \mathbf{PC}' = \mathbf{PC}[\mathbf{nameGen}(\mathtt{sk},\mathtt{iname}),\mathbf{true}]\\ \quad \wedge\ \neg\ \mathbf{Tick}\\ \quad \wedge\ \circ\ [\quad \mathtt{activated\#}(\mathtt{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})\\ \qquad \|_s\ \mathtt{considered\#}(\mathbf{nameGen}(\mathtt{sk},\mathtt{iname});\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})];\end{array}$$

B.11.6 Vollständigkeit

Auch die `ifThenElse`-Kontrolle ist Vollständig - das heißt auch ein `ifThenElse` kontrollierter Plan kann stets eine der oben aufgeführten Regeln anwenden um in der Planausführung weiter zu kommen:

$$\begin{array}{l}\vdash\quad \neg\ \mathbf{allSubsInactive}(\mathbf{AS},\mathtt{iname})\\ \quad \wedge\ \neg\ \mathbf{hasExecutedSub}(\mathbf{AS},\mathtt{iname})\\ \quad \wedge\ \neg\ \mathbf{hasSelectedSub}(\mathbf{AS},\mathtt{iname})\\ \quad \wedge\ \mathbf{hasRestartableSub}(\mathbf{AS},\mathtt{iname}),\\ \qquad \neg\ \mathbf{allSubsInactive}(\mathbf{AS},\mathtt{iname})\\ \quad \wedge\quad \mathbf{hasExecutedSub}(\mathbf{AS},\mathtt{iname})\\ \qquad \vee\ \mathbf{hasSelectedSub}(\mathbf{AS},\mathtt{iname})\\ \qquad \vee\ \neg\ \mathbf{hasRestartableSub}(\mathbf{AS},\mathtt{iname}),\\ \qquad \mathbf{allSubsInactive}(\mathbf{AS},\mathtt{iname})\\ \quad \wedge\ \neg\ \mathbf{ITEP}(\mathbf{PDH},\mathbf{AC})\\ \quad \wedge\ \mathbf{noSecondSubPresent}(\mathtt{iname}),\\ \qquad \mathbf{allSubsInactive}(\mathbf{AS},\mathtt{iname})\\ \quad \wedge\ \neg\ \mathbf{ITEP}(\mathbf{PDH},\mathbf{AC})\\ \quad \wedge\ \mathbf{secondSubPresent}(\mathtt{iname}),\\ \qquad \mathbf{allSubsInactive}(\mathbf{AS},\mathtt{iname})\\ \quad \wedge\ \mathbf{ITEP}(\mathbf{PDH},\mathbf{AC})\end{array}$$

B.12 Zustandsübergänge der ask-Plankontrolle

Der `ask`-Kontrolltyp stellt lediglich einen passiven Plan dar, der ein Signal an die Umgebung sendet, bestimmte Datenwerte einzulesen. Damit wird die `ask`-Kontrolle durch eine einzige

SOS Regel beschrieben, die keine Vorbedingung hat. Entsprechend ist diese Regel automatisch vollständig.

$$\frac{\textbf{isASK(iname)}}{\textbf{iname} \xrightarrow[\sigma,\sigma[\textbf{iname},\ \mathtt{activated}]]{}_{\mathtt{Con}} \textbf{iname}}$$

Die SOS Regel wird entsprechend in der Implementierung umgesetzt, so dass die nachstehende Beweisverpflichtung bewiesen werden kann:

$$\begin{array}{ll} & \textbf{isASK}(\mathtt{iname}), \\ & [\mathtt{body\#}(\mathtt{iname}; \textbf{Tick}, \textbf{EAGG}, \textbf{PDH}, \textbf{ASH}, \textbf{AS}, \textbf{AC}, \textbf{PC}, \textbf{RC})] \\ \vdash & \quad [\textbf{AS}] \\ & \wedge\ \textbf{AS}' = \textbf{AS}[\mathtt{iname}, \mathtt{activated}] \\ & \wedge\ \circ\ [\mathtt{activated\#}(\mathtt{iname}; \textbf{Tick}, \textbf{EAGG}, \textbf{PDH}, \textbf{ASH}, \textbf{AS}, \textbf{AC}, \textbf{PC}, \textbf{RC})] \end{array}$$

B.13 Zustandsübergänge der `user`-Plankontrolle

Der `user`-Kontrolltyp stellt, wie der `ask`-Kontrolltyp ein Signal an die Umgebung dar, allerdings nicht ein Signal zum Einlesen von Daten, sondern ein Signal eine Handlung durchzuführen. Entsprechend ist die definierende SOS Regel analog zum `ask`-Kontrolltypen und wie bei diesem ist die Regel ohne Vorbedingung automatisch vollständig.

$$\frac{\textbf{isUser(iname)}}{\textbf{iname} \xrightarrow[\sigma,\sigma[\textbf{iname},\ \mathtt{activated}]]{}_{\mathtt{Con}} \textbf{iname}}$$

Diese Regel wird von der Implementierung entsprechend umgesetzt:

$$\begin{array}{ll} & \textbf{isUser}(\mathtt{iname}), \\ & [\mathtt{body\#}(\mathtt{iname}; \textbf{Tick}, \textbf{EAGG}, \textbf{PDH}, \textbf{ASH}, \textbf{AS}, \textbf{AC}, \textbf{PC}, \textbf{RC})] \\ \vdash & \quad [\textbf{AS}] \\ & \wedge\ \textbf{AS}' = \textbf{AS}[\mathtt{iname}, \mathtt{activated}] \\ & \wedge\ \circ\ [\mathtt{activated\#}(\mathtt{iname}; \textbf{Tick}, \textbf{EAGG}, \textbf{PDH}, \textbf{ASH}, \textbf{AS}, \textbf{AC}, \textbf{PC}, \textbf{RC})] \end{array}$$

B.14 Zustandsübergänge der `cyclical`-Plankontrolle

Die `cyclical`-Plankontrolle beschreibt einen Plankontrolltyp, der seinen Unterplan wiederholt ausführt, bis dieser eine vorgegebene Anzahl erfolgreicher Durchläufe erreicht hat.

B.14.1 Start des Unterplans

Ist der Unterplan im Zustand `inactive` oder ist der Plan in den Terminierungszuständen `rejected` oder `aborted`, so kann der Plan sofort neu gestartet werden, wenn das durch die zyklische-Zeitannotation vorgegebene Startintervall erreicht wurde:

$$\frac{\begin{array}{l}\mathbf{isCyclical}(\mathbf{iname})\\ \wedge\ \ \mathbf{isInactive}(\mathbf{HS},\mathbf{iname}_1)\\ \quad\vee\ \mathbf{isRejected}(\mathbf{HS},\mathbf{iname}_1)\\ \quad\vee\ \mathbf{isAborted}(\mathbf{HS},\mathbf{iname}_1)\\ \wedge\ \mathbf{isInTimeFrame}(\sigma,\mathbf{iname})\\ \wedge\ \mathbf{iname}_1 = \mathbf{nameGen}(\mathbf{selectFirstSub}(\mathbf{iname}),\mathbf{iname})\end{array}}{\mathbf{iname}\ \overset{\sigma,\mathbf{Tick}(\sigma[\mathbf{iname},\ \mathtt{activated}][\mathbf{iname}_1,\ \mathtt{considered}][\mathbf{iname}_1]_{ac})}{\longrightarrow}_{\mathtt{Con}}\ \mathbf{iname}\|_s\ \mathbf{iname}_1}$$

Die Implementierung verhält sich dazu analog:

$$\begin{array}{l}\mathbf{isCyclical}(\mathtt{iname}),\\ \quad\mathbf{isInactive}(\mathbf{AS},\mathbf{nameGen}(\mathtt{sk},\mathtt{iname}))\\ \vee\ \mathbf{isRejected}(\mathbf{AS},\mathbf{nameGen}(\mathtt{sk},\mathtt{iname}))\\ \vee\ \mathbf{isAborted}(\mathbf{AS},\mathbf{nameGen}(\mathtt{sk},\mathtt{iname})),\\ \mathbf{isInTimeFrame}(\mathbf{ASH},\mathbf{AC},\mathtt{asbru}(\mathbf{nameSel}(\mathtt{iname}))\mathbf{.type.cta}),\\ \mathtt{sk} = \mathbf{selectFirstSub}(\mathtt{iname}),\\ [\mathtt{body\#}(\mathtt{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})]\\ \vdash\ \ [\mathbf{AS},\ \mathbf{PC},\ \mathbf{Tick}]\\ \wedge\ \neg\ \mathbf{Tick}'\\ \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname},\mathtt{activated}][\mathbf{nameGen}(\mathtt{sk},\mathtt{iname}),\mathtt{considered}]\\ \wedge\ \mathbf{PC}' = \mathbf{PC}[\mathbf{nameGen}(\mathtt{sk},\mathtt{iname}),\mathbf{true}]\\ \wedge\ \circ\ [\ \ \mathtt{activated\#}(\mathtt{iname};\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})\\ \qquad\|_s\ \mathtt{considered\#}(\mathbf{nameGen}(\mathtt{sk},\mathtt{iname});\mathbf{Tick},\mathbf{EAGG},\mathbf{PDH},\mathbf{ASH},\mathbf{AS},\mathbf{AC},\mathbf{PC},\mathbf{RC})];\end{array}$$

B.14.2 Beendigung des letzten Durchlaufes

Der zyklische Kontrolltyp darf sich beenden, wenn der Unterplan im Zustand `completed` ist und damit die Zahl der notwendigen Wiederholungen erreicht wurde:

$$\frac{\begin{array}{l}\mathbf{isCyclical}(\mathbf{iname})\\ \wedge\ \mathbf{isCompleted}(\mathbf{HS},\mathbf{iname}_1)\\ \wedge\ \mathbf{iname}_1 = \mathbf{nameGen}(\mathbf{selectFirstSub}(\mathbf{iname}),\mathbf{iname})\\ \wedge\ \mathbf{hasCompletedRounds}(\mathbf{HS},\mathbf{iname})\end{array}}{\mathbf{iname}\ \overset{\sigma,\mathbf{Tick}(\sigma[\mathbf{iname},\ \mathtt{completed}])}{\longrightarrow}_{\mathtt{Con}}\ \mathbf{iname}}$$

Die Implementierung setzt diese SOS-Regel um:

$$
\begin{array}{l}
\quad \mathbf{isCyclical}(\mathtt{iname}),\\
\quad \mathbf{isCompleted}(\mathbf{AS}, \mathbf{nameGen}(\mathtt{sk}, \mathtt{iname})),\\
\quad \mathtt{sk} = \mathbf{selectFirstSub}(\mathtt{iname}),\\
\quad \mathbf{hasCompletedRounds}(\mathtt{iname}, \mathbf{RC})\\
\quad [\mathrm{body}\#(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})]\\
\vdash \quad [\mathbf{AS},\ \mathbf{Tick}]\\
\quad \wedge\ \neg\ \mathbf{Tick}'\\
\quad \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname}, \mathtt{completed}]\\
\quad \wedge\ \circ\ \mathbf{last};
\end{array}
$$

B.14.3 Neustart nach einem Durchlauf

Ist der Unterplan im Zustand `completed` so wird der Rundenzähler des Plans erhöht, wenn der Unterplan noch nicht die erforderliche Zahl der Wiederholungen erreicht hat und das Startintervall erreicht wurde:

$$
\frac{\begin{array}{l}\mathbf{isCyclical}(\mathbf{iname})\\ \wedge\ \mathbf{isCompleted}(\mathbf{HS}, \mathbf{iname}_1)\\ \wedge\ \mathbf{iname}_1 = \mathbf{nameGen}(\mathbf{selectFirstSub}(\mathbf{iname}), \mathbf{iname})\\ \wedge\ \neg\ \mathbf{hasCompletedRounds}(\mathbf{HS}, \mathbf{iname})\\ \wedge\ \mathbf{isInTimeFrame}(\sigma, \mathbf{iname})\end{array}}{\begin{array}{l}\sigma, \mathbf{Tick}(\sigma\ [\mathbf{iname}, \mathtt{activated}][\mathbf{iname}_1, \mathtt{considered}]\\ \qquad [\mathbf{iname}, \mathbf{HS}[\mathbf{iname}].\mathbf{rounds}\ + 1])\\ \mathbf{iname} \qquad\qquad \longrightarrow \qquad\qquad \mathtt{Con}\quad \mathbf{iname} \|_s\ \mathbf{iname}_1\end{array}}
$$

Nachstehende Aussage weist nach, dass dies auch durch die Implementierung umgesetzt wird:

$$
\begin{array}{l}
\quad \mathbf{isCyclical}(\mathtt{iname}),\\
\quad \mathbf{isCompleted}(\mathbf{AS}, \mathbf{nameGen}(\mathtt{sk}, \mathtt{iname})),\\
\quad \mathtt{sk} = \mathbf{selectFirstSub}(\mathtt{iname}),\\
\quad \mathbf{isInTimeFrame}(\mathbf{ASH}, \mathbf{AC}, \mathtt{asbru}(\mathbf{nameSel}(\mathtt{iname}))\mathbf{.type.cta}),\\
\quad \neg\ \mathbf{hasCompletedRounds}(\mathtt{iname}, \mathbf{RC})\\
\quad [\mathrm{body}\#(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})]\\
\vdash \quad [\mathbf{AS},\ \mathbf{PC},\ \mathbf{RC},\ \mathbf{Tick}]\\
\quad \wedge\ \neg\ \mathbf{Tick}'\\
\quad \wedge\ \mathbf{AS}' = \mathbf{AS}[\mathtt{iname}, \mathtt{activated}][\mathbf{nameGen}(\mathtt{sk}, \mathtt{iname}), \mathtt{considered}]\\
\quad \wedge\ \mathbf{PC}' = \mathbf{PC}[\mathbf{nameGen}(\mathtt{sk}, \mathtt{iname}), \mathbf{true}]\\
\quad \wedge\ \mathbf{RC}' = \mathbf{RC}[\mathtt{iname}, \mathbf{RC}[\mathtt{iname}] + 1]\\
\quad \wedge\ \circ\ [\quad \mathtt{activated}\#(\mathtt{iname}; \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})\\
\qquad \|_s\ \mathtt{considered}\#(\mathbf{nameGen}(\mathtt{sk}, \mathtt{iname}); \mathbf{Tick}, \mathbf{EAGG}, \mathbf{PDH}, \mathbf{ASH}, \mathbf{AS}, \mathbf{AC}, \mathbf{PC}, \mathbf{RC})];
\end{array}
$$

B.14.4 Passives Abwarten

Ist der Plan terminiert oder im Zustand `inactive` und ist das Startintervall derzeit nicht erreicht, so soll der Plan abwarten, sofern nicht die notwendige Zahl der Durchläufe erreicht wurde und der Plan sich beenden kann:

$$\frac{\begin{array}{l}\mathbf{isCyclical}(\mathbf{iname})\\ \wedge\ \ \mathbf{isTerminated}(\mathbf{HS},\mathbf{iname}_1)\\ \quad\vee\ \mathbf{isInactive}(\mathbf{HS},\mathbf{iname}_1)\\ \quad\rightarrow\neg\ \mathbf{isInTimeFrame}(\sigma,\mathbf{iname})\\ \wedge\ \ \mathbf{isCompleted}(\mathbf{HS},\mathbf{iname}_1)\\ \quad\rightarrow\neg\ \mathbf{hasCompletedRounds}(\mathbf{HS},\mathbf{iname})\\ \wedge\ \mathbf{iname}_1=\mathbf{nameGen}(\mathbf{selectFirstSub}(\mathbf{iname}),\mathbf{iname})\end{array}}{\mathbf{iname}\ {}^{\sigma,\sigma[\mathbf{iname},\ \overrightarrow{\ \ }\ \mathtt{activated}]}_{\mathtt{Con}}\ \ \mathbf{iname}}$$

Auch dies wird durch die Implementierung entsprechend umgesetzt:

```
isCyclical(iname),
    isTerminated(AS, nameGen(sk, iname))
  ∨ isInactive(AS, nameGen(sk, iname))
→ ¬ isInTimeFrame(ASH, AC, asbru(nameSel(iname)).type.cta),
  isCompleted(AS, nameGen(sk, iname))
→ ¬ hasCompletedRounds(iname, RC),
sk = selectFirstSub(iname),
⊢  ⌈AS⌉
∧ AS′ = AS[iname, activated]
∧ ○ [activated#(iname; Tick, EAGG, PDH, ASH, AS, AC, PC, RC)];
```

B.14.5 Vollständigkeit

Auch die den `cyclical`-Kontrolltyp definierenden Regeln sind vollständig. Jede `cyclical` Kontrolle kann also unabhängig vom Zustand stets einen Schritt ausführen.

```
⊢      isTerminated(AS, nameGen(sk, iname))
     ∨ isInactive(AS, nameGen(sk, iname))
   → ¬ isInTimeFrame(ASH, AC, asbru(nameSel(iname)).type.cta)
 ∧   isCompleted(AS, nameGen(sk, iname))
   → ¬ hasCompletedRounds(iname, RC),
   isCompleted(AS, nameGen(sk, iname))
 ∧ isInTimeFrame(ASH, AC, asbru(nameSel(iname)).type.cta)
 ∧ ¬ hasCompletedRounds(iname, RC),
   isCompleted(AS, nameGen(sk, iname))
 ∧ hasCompletedRounds(iname, RC),
     isInactive(AS, iname₁)
   ∨ isRejected(AS, iname₁)
   ∨ isAborted(AS, iname₁)
 ∧ isInTimeFrame(ASH, AC, asbru(nameSel(iname)).type.cta)
```

C Prädikate zur Formulierung der SOS-Regeln

C.1 Plankontrolltyp-Prädikate

Zur vereinfachten Entscheidung, welchen Plantyp ein Plan hat, werden nachstehende Prädikate definiert.

Definition

Die nachfolgenden Prädikate werden definiert um für einen Planinstanznamen zu entscheiden, welchen Typ er hat.

isSequential	:	**instance-name** $\mapsto$ **bool**
isAnyorder	:	**instance-name** $\mapsto$ **bool**
isParallel	:	**instance-name** $\mapsto$ **bool**
isUnordered	:	**instance-name** $\mapsto$ **bool**
isOnAbort	:	**instance-name** $\mapsto$ **bool**
isASK	:	**instance-name** $\mapsto$ **bool**
isUser	:	**instance-name** $\mapsto$ **bool**
isITE	:	**instance-name** $\mapsto$ **bool**
isCyclical	:	**instance-name** $\mapsto$ **bool**

Die Semantik dieser Prädikate ist wie folgt definiert:

isSequential(**iname**)	$\leftrightarrow$	`asbru`(**nameSel**(**iname**))**.type** = `sequential`
isAnyorder(**iname**)	$\leftrightarrow$	`asbru`(**nameSel**(**iname**))**.type** = `anyorder`
isParallel(**iname**)	$\leftrightarrow$	`asbru`(**nameSel**(**iname**))**.type** = `parallel`
isUnordered(**iname**)	$\leftrightarrow$	`asbru`(**nameSel**(**iname**))**.type** = `unordered`
isOnAbort(**iname**)	$\leftrightarrow$	`asbru`(**nameSel**(**iname**))**.type** = `onabort`
isASK(**iname**)	$\leftrightarrow$	`asbru`(**nameSel**(**iname**))**.type** = `ask`
isUser(**iname**)	$\leftrightarrow$	`asbru`(**nameSel**(**iname**))**.type** = `user`
isITE(**iname**)	$\leftrightarrow$	`asbru`(**nameSel**(**iname**))**.type** = `ifThenElse`
isCyclical(**iname**)	$\leftrightarrow$	`asbru`(**nameSel**(**iname**))**.type** = `cyclical`

Maßgebliche Aufgabe dieser Prädikate ist es, die technische Komplexität der Abfrage nach dem Plantyp zu kapseln. Damit sollen die SOS Regeln besser lesbar werden. Das Prädikat **isSequential** prüft, ob die angegebene Planinstanz vom Kontrolltyp `sequential` ist. Das Prädikat **isAnyorder** prüft auf den `anyorder` Kontrolltypen. Die restlichen Prädikate sind analog definiert.

C.2 Prädikate des Planzustand

Genauso, wie Prädikate definiert werden um die technische Komplexität des Zugriffs auf den Plantypen zu kapseln, werden Prädikate definiert, die den Planzustand einer Planinstanz auf bestimmte Zustände oder Zustandsgruppen überprüfen.

Definition

Die nachstehenden Prädikate werten über Hierarchiezustände und Planinstanznamen aus:

$$
\begin{array}{lcl}
\textbf{isInactive} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\
\textbf{isConsidered} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\
\textbf{isPossible} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\
\textbf{isReady} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\
\textbf{isActivated} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\
\textbf{isSuspended} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\
\textbf{isRejected} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\
\textbf{isAborted} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\
\textbf{isCompleted} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool}
\end{array}
$$

Die Semantik dieser Prädikate prüft, ob die angegebene Planinstanz im gegebenen Hierarchiezustand in einem bestimmten Planzustand ist. Die Semantik ist wie folgt definiert:

$$
\begin{array}{lcl}
\textbf{isInactive}(\textbf{HS}, \textbf{iname}) & \leftrightarrow & \textbf{HS}[\textbf{iname}].\textbf{state} = \texttt{inactive} \\
\textbf{isConsidered}(\textbf{HS}, \textbf{iname}) & \leftrightarrow & \textbf{HS}[\textbf{iname}].\textbf{state} = \texttt{considered} \\
\textbf{isPossible}(\textbf{HS}, \textbf{iname}) & \leftrightarrow & \textbf{HS}[\textbf{iname}].\textbf{state} = \texttt{possible} \\
\textbf{isReady}(\textbf{HS}, \textbf{iname}) & \leftrightarrow & \textbf{HS}[\textbf{iname}].\textbf{state} = \texttt{ready} \\
\textbf{isActivated}(\textbf{HS}, \textbf{iname}) & \leftrightarrow & \textbf{HS}[\textbf{iname}].\textbf{state} = \texttt{activated} \\
\textbf{isSuspended}(\textbf{HS}, \textbf{iname}) & \leftrightarrow & \textbf{HS}[\textbf{iname}].\textbf{state} = \texttt{suspended} \\
\textbf{isRejected}(\textbf{HS}, \textbf{iname}) & \leftrightarrow & \textbf{HS}[\textbf{iname}].\textbf{state} = \texttt{rejected} \\
\textbf{isAborted}(\textbf{HS}, \textbf{iname}) & \leftrightarrow & \textbf{HS}[\textbf{iname}].\textbf{state} = \texttt{aborted} \\
\textbf{isCompleted}(\textbf{HS}, \textbf{iname}) & \leftrightarrow & \textbf{HS}[\textbf{iname}].\textbf{state} = \texttt{completed}
\end{array}
$$

Aufgabe dieser Prädikate ist es, die Vorbedingung der SOS Regeln so aufzuschreiben, dass die technische Komplexität der Zugriffe gekapselt wird. Damit sollen die Vorbedingungen der SOS Regeln leichter verständlich werden. Die Semantik der Prädikate ist so definiert, dass das Prädikat **isInactive** prüft, ob der Plan im Zustand `inactive` ist. Analog dazu wird **isConsidered** definiert, das überprüft, ob die Planinstanz im Zustand `considered` ist. Die Semantik der restlichen Prädikate erfolgt analog.

Im Folgenden werden weitere Prädikate bezüglich des Planzustands von Planinstanzen definiert. Diese prüfen nicht nach, ob die angegebene Planinstanz in einem speziellen Zustand ist, sondern ob sie sich in einer bestimmten Phase der Ausführung, beispielsweise der `Selection`-Phase befindet.

Definition

Für die Überprüfung, ob ein Plan Teil einer Zustandsgruppe ist, werden nachfolgende Prädikate definiert:

$$
\begin{array}{lcl}
\textbf{isSelected} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\
\textbf{isExecuted} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\
\textbf{isTerminated} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\
\textbf{isStartable} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool}
\end{array}
$$

Deren Semantik wird durch die folgenden Gleichungen beschrieben:

$$
\begin{aligned}
\textbf{isSelected}(\textbf{HS}, \textbf{iname}) \quad &\leftrightarrow \quad \textbf{isConsidered}(\textbf{HS}, \textbf{iname}) \\
&\quad \vee \textbf{isPossible}(\textbf{HS}, \textbf{iname}) \\
&\quad \vee \textbf{isReady}(\textbf{HS}, \textbf{iname}) \\
\textbf{isExecuted}(\textbf{HS}, \textbf{iname}) \quad &\leftrightarrow \quad \textbf{isActivated}(\textbf{HS}, \textbf{iname}) \\
&\quad \vee \textbf{isSuspended}(\textbf{HS}, \textbf{iname}) \\
\textbf{isTerminated}(\textbf{HS}, \textbf{iname}) \quad &\leftrightarrow \quad \textbf{isAborted}(\textbf{HS}, \textbf{iname}) \\
&\quad \vee \textbf{isRejected}(\textbf{HS}, \textbf{iname}) \\
&\quad \vee \textbf{isCompleted}(\textbf{HS}, \textbf{iname}) \\
\textbf{isStartable}(\textbf{HS}, \textbf{iname}) \quad &\leftrightarrow \quad \textbf{isInactive}(\textbf{HS}, \textbf{iname}) \\
&\quad \vee \quad \textbf{isAborted}(\textbf{HS}, \textbf{iname}) \\
&\quad \wedge \texttt{asbru}(\textbf{nameSel}(\textbf{parSel}(\textbf{iname}))).\textbf{retry-aborted}
\end{aligned}
$$

Vier verschiedene Zustandsgruppenprädikate werden definiert. Diese heißen, der Reihe nach **isSelected**, **isExecuted**, **isTerminated** und **isStartable**. Für einen Plan wertet das Prädikat **isSelected** zu wahr aus, wenn der Plan in der `Selection`-Phase ist. Das Prädikat **isExecuted** wertet wahr aus, wenn der Plan in der `Execution`-Phase ist. Das **isTerminated** Prädikat wertet zu wahr aus, wenn der Plan in einem der Terminierungszustände ist, also `rejected`, `aborted` oder `completed`. Das letzte Prädikat **isStartable** formuliert, wann ein Plan gestartet werden kann. Dies ist der Fall, wenn der Plan entweder im Zustand `inactive` ist oder der Plan abgebrochen wurde (`aborted`) und sein übergeordneter Plan abgebrochene Unterpläne neu starten soll.

C.3 Planzustandspropagation

Asbru-Pläne propagieren ihren Zustand teilweise an ihre Unterpläne. Damit ist gemeint, dass ein Plan abbricht, wenn sein übergeordneter Plan terminiert ist. Auch wird ein Plan seinen Ablauf unterbrechen, wenn der übergeordnete Plan selbst unterbrochen wurde.

Damit Überprüfungen auf Fälle von Planzustandspropagation möglichst klar verständlich in den SOS Regeln formuliert werden können, werden die nachstehenden zwei Prädikate angegeben, die diese Prüfungen kapseln.

Definition

Es werden Prädikate zur Überprüfung des Zustands des übergeordneten Plans spezifiziert:

$$
\begin{aligned}
\textbf{parent\_terminated} &: \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\
\textbf{parent\_suspended} &: \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool}
\end{aligned}
$$

Deren Semantik ist wie folgt definiert:

$$
\begin{aligned}
\textbf{parent\_terminated}(\textbf{HS}, \textbf{iname}) \quad &\leftrightarrow \quad \neg \exists\, \textbf{SK}.\textbf{iname} = \textbf{superGen}(\textbf{SK}) \\
&\quad \wedge \textbf{isTerminated}(\textbf{HS}, \textbf{parSel}(\textbf{iname})) \\
\textbf{parent\_suspended}(\textbf{HS}, \textbf{iname}) \quad &\leftrightarrow \quad \neg \exists\, \textbf{SKiname} = \textbf{superGen}(\textbf{SK}) \\
&\quad \wedge \textbf{isSuspended}(\textbf{HS}, \textbf{parSel}(\textbf{iname}))
\end{aligned}
$$

Diese Definition besagt, dass das Prädikat **parent_terminated** zunächst überprüft, ob der Plan ein Top-Level Plan ist, der keinen übergeordneten Plan hat. Nur wenn dies der Fall

ist, wird weiter überprüft, ob der übergeordnete Plan in einem terminierten Zustand ist. In diesem Fall ist das Prädikat wahr.

Analog überprüft das Prädikat **parent_suspended** zunächst, ob der Plan ein Top-Level Plan ist und falls dies nicht gegeben ist, ob der übergeordnete Plan im Zustand `suspended` ist.

Ist der Instanzname durch den **superGen** Konstruktor gebildet, was bedeutet, dass der Plan keinen übergeordneten Plan hat, so sind diese Prädikate stets automatisch falsch.

C.4 Funktionen zur Aktualisierung des Zustands

Die SOS Regeln beschreiben den Übergang eines Paares aus Systembeschreibung und Zustand in ein Paar aus Systembeschreibung und Zustand. Dabei steht der neue Zustand in Relation zum alten Zustand. Genauer gilt, dass der neue Zustand durch Aktualisierungen aus dem alten Zustand entsteht. Es sollen in diesem Abschnitt Aktualisierungsfunktionen definiert werden, die es erlauben, die syntaktisch komplexen Aktualisierungen durch leichter verständliche Formeln auszudrücken. Typischerweise sind die Aktualisierungsfunktionen dabei hierarchisch aufgebaut, das heißt, eine Aktualisierungsfunktion des Zustands stützt sich ab auf eine, die auf einer Komponente des Zustands definiert ist.

Im Folgenden sollen diese Aktualisierungsfunktionen in zwei Gruppen spezifiziert werden. Zunächst all jene Aktualisierungen, die mit Aktivierungssignalen des Plans befasst sind, danach diejenigen, die die Planstatuskomponenten aktualisieren. Zunächst wird jedoch die Definition der Prädikate angegeben, die Berechnen, ob externe Signale an den Plan geschickt werden.

Definition

Für den Zugriff auf die Signale des Plans sind folgende Prädikate definiert:

$$\begin{array}{lcl} \textbf{planPreferred} & : & \textbf{environment-aggregation} \times \textbf{instance-name} \mapsto \textbf{bool} \\ \textbf{isActSig} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \end{array}$$

Deren Semantik wird definiert durch die nachstehenden Gleichungen:

$$\begin{array}{lcl} \textbf{planPreferred}(\textbf{EAGG}, \textbf{iname}) & \leftrightarrow & \textbf{EAGG}[\textbf{iname}].\textbf{prefer} \\ \textbf{isActSig}(\textbf{HS}, \textbf{iname}) & \leftrightarrow & \textbf{HS}[\textbf{iname}].\textbf{activate} \end{array}$$

In dieser Definition werden zwei Prädikate angegeben, die überprüfen, ob bestimmte Signale für einen Plan gesetzt sind. Das Prädikat **planPreferred** überprüft dabei, ob ein Plan bevorzugt behandelt werden soll, wenn der Plan Unterplan eines `anyorder` kontrollierten Plans ist. Ein `anyorder` kontrollierter Plan wird bei der Aktivierung seiner Unterpläne stets einen aussuchen, der sich im Zustand `ready` befindet. Unter all diesen Plänen werden die bevorzugt aktiviert, die durch das Prädikat **planPreferred** ausgezeichnet werden. Das zweite Prädikat in dieser Definition ist das Prädikat **isActSig**. Dieses überprüft, ob das Aktivierungssignal des Plans gesendet wurde, das im Semantik-Statechart 9.1 als `E` Signal bezeichnet wird. Diese Überprüfung wird im Zustand `ready` durchgeführt, um zu testen, ob der Plan in die `Execution`-Phase vorrücken darf.

Um die Aktivierungssignale zu setzten und zurückzusetzen sind einige Aktualisierungsfunktionen definiert. Die Funktionen kapseln technische Komplexität und erhöhen dadurch die Lesbarkeit der Formeln erhöhen.

Definition

Für die Aktualisierungen von Signalkomponenten des Zustands sind die nachfolgenden Funktionen und Prädikate definiert:

$$
\begin{array}{lcl}
.[.]_{ac} & : & \textbf{state} \times \textbf{instance-name} \mapsto \textbf{state} \\
.[.]_{nac} & : & \textbf{state} \times \textbf{instance-name} \mapsto \textbf{state} \\
.[.]_{ac} & : & \textbf{state} \times \textbf{plan-list} \times \textbf{instance-name} \mapsto \textbf{state} \\
.[.]_{nac} & : & \textbf{state} \times \textbf{plan-list} \times \textbf{instance-name} \mapsto \textbf{state} \\
.[.]_{ac} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{hierarchy-state} \\
.[.]_{nac} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{hierarchy-state} \\
.[.]_{ac} & : & \textbf{hierarchy-state} \times \textbf{plan-list} \times \textbf{instance-name} \mapsto \textbf{hierarchy-state} \\
.[.]_{nac} & : & \textbf{hierarchy-state} \times \textbf{plan-list} \times \textbf{instance-name} \mapsto \textbf{hierarchy-state}
\end{array}
$$

Sei σ ein Zustand mit $\sigma = (\textbf{ASH}, \textbf{HS}, \textbf{PDH}, \textbf{EAGG}, \textbf{Tick}, \textbf{AC})$ und **iname** ein Instanzname. Dann ist die Semantik dieser Funktionen definiert durch die nachstehenden Gleichungen:

$$
\begin{array}{lcl}
\sigma[\textbf{iname}]_{ac} & = & (\textbf{ASH}, \textbf{HS}[\textbf{iname}]_{ac}, \textbf{PDH}, \textbf{EAGG}, \textbf{Tick}, \textbf{AC}) \\
\sigma[\textbf{iname}]_{nac} & = & (\textbf{ASH}, \textbf{HS}[\textbf{iname}]_{nac}, \textbf{PDH}, \textbf{EAGG}, \textbf{Tick}, \textbf{AC}) \\
\sigma[\textbf{PL}, \textbf{iname}]_{ac} & = & (\textbf{ASH}, \textbf{HS}[\textbf{PL}, \textbf{iname}]_{ac}, \textbf{PDH}, \textbf{EAGG}, \textbf{Tick}, \textbf{AC}) \\
\sigma[\textbf{PL}, \textbf{iname}]_{nac} & = & (\textbf{ASH}, \textbf{HS}[\textbf{PL}, \textbf{iname}]_{nac}, \textbf{PDH}, \textbf{EAGG}, \textbf{Tick}, \textbf{AC}) \\
\textbf{HS}[\textbf{iname}]_{ac} & = & \textbf{HS}[\textbf{iname}, \textbf{HS}[\textbf{iname}][\textbf{true}]] \\
\textbf{HS}[\textbf{iname}]_{nac} & = & \textbf{HS}[\textbf{iname}, \textbf{HS}[\textbf{iname}][\textbf{false}]] \\
\textbf{HS}[[], \textbf{iname}]_{ac} & = & \textbf{HS} \\
\textbf{HS}[[], \textbf{iname}]_{nac} & = & \textbf{HS} \\
\textbf{HS}[\textbf{SK} + \textbf{PL}, \textbf{iname}]_{ac} & = & \textbf{HS}[\textbf{nameGen}(\textbf{SK}, \textbf{iname})]_{ac}[\textbf{PL}, \textbf{iname}]_{ac} \\
\textbf{HS}[\textbf{SK} + \textbf{PL}, \textbf{iname}]_{nac} & = & \textbf{HS}[\textbf{nameGen}(\textbf{SK}, \textbf{iname})]_{nac}[\textbf{PL}, \textbf{iname}]_{nac}
\end{array}
$$

Die hier definierten Funktionen haben den Zweck, die Plankommunikation zwischen einem Plan und seinem Unterplan zu beschreiben. Durch die SOS Regeln wird festgelegt, dass ein Plan seinen Unterplänen zu geeigneter Zeit das Aktivierungssignal schickt. Beim Übergang vom Zustand `ready` in den Zustand `activated` wird der Unterplan dieses Signal „konsumieren“, d.h. der Unterplan wird das Signal zurücksetzen.

Relevant für die Formulierung der SOS-Regeln sind die ersten vier Funktionen. Dabei dient die erste Funktion der Aktivierung eines einzelnen Unterplans, die zweite Funktion dient der Zurücksetzung des Aktivierungssignals bei Eintritt in die `Execution`-Phase und die dritte Funktion dient der Aktivierung mehrerer Unterpläne. Letzteres wird beispielsweise bei der `unordered`-Kontrolle benötigt, wenn alle startbaren Unterpläne gleichzeitig aktiviert werden sollen. Die vierte Funktion wird verwendet, um alle Aktivierungssignale an die Unterpläne gleichzeitig zurückzusetzen.

Die Semantik der Funktionen wird definiert durch die nächsten vier Funktionen. Diese haben eine ähnliche Typisierung, arbeiten jedoch statt auf vollständigen Zuständen lediglich auf der **HS** Komponente eines Zustands. Die Funktionen ändern dabei die **bool** Komponente des Planes ab, der beim Methodenaufruf angegeben wurde.

$$\sigma[\textbf{iname}]_{ac} = (\textbf{ASH}, \textbf{HS}[\textbf{iname}]_{ac}, \textbf{PDH}, \textbf{EAGG}, \textbf{Tick}, \textbf{AC})$$

Der Funktionsaufruf auf der Zustandskomponente wird einfach auf den Funktionsaufruf auf die **HS** Komponente des Zustands zurückgeführt. Das bedeutet, der Ergebniszustand unterscheidet sich vom Ausgangszustand nur durch die **HS** Komponente.

$$\mathbf{HS}[\mathbf{iname}]_{ac} = \mathbf{HS}[\mathbf{iname}, \mathbf{HS}[\mathbf{iname}][\mathbf{true}]]$$

Der Methodenaufruf auf der **HS** Komponente soll dabei wie folgt gelesen werden. Zunächst wird aus der **HS** Komponente der Planstatus selektiert, der zu dem Plan mit dem Instanznamen **iname** gehört.

$$\mathbf{HS}[\mathbf{iname}]$$

Dieser Planstatus soll aktualisiert werden durch die Aktualisierungsfunktion, die lediglich die **bool** Komponente des Planstatus ersetzt, in diesem Fall mit dem Wert **true**

$$\mathbf{HS}[\mathbf{iname}][\mathbf{true}]$$

Das Ergebnis dieses Methodenaufrufs ist ein Planstatus, der das Aktivierungssignal gesetzt hat, ansonsten aber gegenüber dem ursprünglichen Planstatus unverändert wird. Dieser wird nun an der Stelle **iname** in die **HS** Komponente eingefügt und ersetzt so den alten Planstatus.

$$\mathbf{HS}[\mathbf{iname}, \mathbf{HS}[\mathbf{iname}][\mathbf{true}]]$$

Ergebnis dieses Aufrufs ist ein Hierarchie-Zustand, der lediglich an der Stelle eines Aktualisierungssignals vom Ausgangs-Hierarchie-Zustand abweicht und als neue **HS** Komponente des Zustands verwendet wird.

Abgesehen von den Aktualisierungen des Aktivierungssignals werden Funktionen definiert, die den Planzustand eines Planes oder mehrerer Pläne aktualisieren. Diese werden im Folgenden spezifiziert.

Definition

Zur Aktualisierung der Planzustandskomponente im Hierarchiezustand sind folgende Funktionen definiert:

$$\begin{array}{lcl}
\mathbf{.}\,[\,\mathbf{.}\,] & : & \mathbf{state} \times \mathbf{instance\text{-}name} \times \mathbf{plan\text{-}state} \mapsto \mathbf{state} \\
\mathbf{.}\,[\,\mathbf{.}\,] & : & \mathbf{state} \times \mathbf{plan\text{-}list} \times \mathbf{instance\text{-}name} \times \mathbf{plan\text{-}state} \mapsto \mathbf{state} \\
\mathbf{.}\,[\,\mathbf{.}\,] & : & \mathbf{hierarchy\text{-}state} \times \mathbf{instance\text{-}name} \times \mathbf{plan\text{-}state} \mapsto \mathbf{hierarchy\text{-}state} \\
\mathbf{.}\,[\,\mathbf{.}\,] & : & \mathbf{hierarchy\text{-}state} \times \mathbf{plan\text{-}list} \times \mathbf{instance\text{-}name} \times \mathbf{plan\text{-}state} \mapsto \mathbf{state}
\end{array}$$

Sei σ ein Zustand mit $\sigma = (\mathbf{ASH}, \mathbf{HS}, \mathbf{PDH}, \mathbf{EAGG}, \mathbf{Tick}, \mathbf{AC})$. Dann ist deren Semantik wird definiert durch die nachstehenden Gleichungen:

$$\begin{array}{lcl}
\sigma[\mathbf{iname}, \mathbf{PS}] & = & (\mathbf{ASH}, \mathbf{HS}[\mathbf{iname}, \mathbf{PS}], \mathbf{PDH}, \mathbf{EAGG}, \mathbf{Tick}, \mathbf{AC}) \\
\mathbf{HS}[\mathbf{iname}, \mathbf{PS}] & = & \mathbf{HS}[\mathbf{iname}, \mathbf{HS}[\mathbf{iname}][\mathbf{PS}]] \\
\sigma[\mathbf{PL}, \mathbf{iname}, \mathbf{PS}] & = & (\mathbf{ASH}, \mathbf{HS}[\mathbf{PL}, \mathbf{iname}, \mathbf{PS}], \mathbf{PDH}, \mathbf{EAGG}, \mathbf{Tick}, \mathbf{AC}) \\
\mathbf{HS}[[], \mathbf{iname}, \mathbf{PS}] & = & \mathbf{HS} \\
\mathbf{HS}[\mathbf{SK} + \mathbf{PL}, \mathbf{iname}, \mathbf{PS}] & = & \mathbf{HS}[\mathbf{nameGen}(\mathbf{SK}, \mathbf{iname}), \mathbf{PS}][\mathbf{PL}, \mathbf{iname}, \mathbf{PS}]
\end{array}$$

Die Aktualisierung des Planzustands verläuft analog zur Aktualisierung des Aktivierungssignals. Das heißt, es werden Aktualisierungsfunktionen analoger Typisierung definiert. Wie auch bei der Aktualisierung der Aktivierungssignale werden Funktionen angegeben, die rekursiv Planmengen aktualisieren. Dies ist beispielsweise notwendig, wenn ein Plan beim Zustandsübergang vom Zustand `ready` in den Zustand `activated` den Planzustand aller seiner Unterpläne nach `inactive` abändert.

Abgeschlossen wird die Definition der Aktualisierungsfunktionen mit den nachfolgenden drei Aktualisierungsfunktionen. Diese Funktionen übernehmen die Aktualisierung des Rundenzählers im Planzustand. Dieser wird nur durch den jeweils zugehörigen Plan gelesen und geschrieben. Dadurch entfällt die Notwendigkeit für rekursive Aktualisierungen. Zuletzt wird noch eine Aktualisierungsfunktion für die **Tick** Variable des Zustands definiert.

Definition

Für die Aktualisierung des Rundenzählers sowie der **Tick** Komponente des Zustands sind folgende Funktionen definiert:

$$
\begin{array}{lcl}
\mathbf{.\,[\,.\,]} & : & \mathbf{state} \times \mathbf{instance\text{-}name} \times \mathbf{nat} \mapsto \mathbf{state} \\
\mathbf{.\,[\,.\,]} & : & \mathbf{hierarchy\text{-}state} \times \mathbf{instance\text{-}name} \times \mathbf{nat} \mapsto \mathbf{hierarchy\text{-}state} \\
\mathbf{Tick}\ . & : & \mathbf{state} \mapsto \mathbf{state}
\end{array}
$$

Sei σ ein Zustand mit $\sigma = (\mathbf{ASH}, \mathbf{HS}, \mathbf{PDH}, \mathbf{EAGG}, \mathbf{Tick}, \mathbf{AC})$. Dann ist deren Semantik definiert durch die nachstehenden Gleichungen:

$$
\begin{array}{lcl}
\sigma[\mathbf{iname}, \mathbf{n}] & = & (\mathbf{ASH}, \mathbf{HS}[\mathbf{iname}, \mathbf{n}], \mathbf{PDH}, \mathbf{EAGG}, \mathbf{Tick}, \mathbf{AC}) \\
 & & \\
\mathbf{HS}[\mathbf{iname}, \mathbf{n}] & = & \mathbf{HS}[\mathbf{iname}, \mathbf{HS}[\mathbf{iname}][\mathbf{n}]] \\
\mathbf{Tick}(\sigma) & = & (\mathbf{ASH}, \mathbf{HS}, \mathbf{PDH}, \mathbf{EAGG}, \mathbf{false}, \mathbf{AC})
\end{array}
$$

Die Definition der Aktualisierungsfunktionen für den Rundenzähler des Zustands ist analog zu den bereits oben beschriebenen Aktualisierungsfunktionen für Planzustand und Aktivierungssignal definiert. Bezüglich der **Tick** Komponente des Zustands gilt, dass die Umgebung jeweils vor jedem Systemschritt die **Tick** Komponente auf den Wert **true** setzt. Wenn kein Plan diese Komponente verändert, dann wird der nächste Schritt ein Makro-Schritt sein. Wenn nur ein Plan die Komponente auf den Wert **false** setzt, dann wird dies bei der Zusammenführung der Zustände durch den `sync` Operator in den globalen Endzustand übertragen. Ist am Ende einer Systemtransition die **Tick** Komponente im Zustand auf den Wert **false** gesetzt, so wird ein Makro-Schritt verhindert. Daher ist es nur notwendig, eine Funktion zu definieren, die Makro-Schritte verbietet. Der Fall, in dem ein Makro-Schritt erlaubt sein soll ist ohnehin der Normalfall.

C.5 Planzustandsprädikate für Unterpläne

Praktisch alle SOS Regeln, die Zustandsübergänge des `activated` Zustands beschrieben, überprüfen den Zustand der Unterpläne. Geprüft werden muss dabei beispielsweise, ob diese Pläne terminiert sind, aktiviert wurden oder noch im Zustand `inactive` sind. Um diese Überprüfungen möglichst verständlich und klar aufschreiben zu können, werden die nachfolgenden Prädikate angegeben.

Definition

Nachstehende Prädikate definieren, ob ein Plan Unterpläne in einem bestimmten Zustand hat:

$$
\begin{array}{lcl}
\textbf{hasInactiveSub} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\
\textbf{hasSelectedSub} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\
\textbf{hasReadySub} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\
\textbf{hasExecutedSub} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\
\textbf{hasExecOrSigReadySub} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\
\textbf{hasRestartableSub} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\
\textbf{hasStartableSub} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool}
\end{array}
$$

Deren Semantik ist wie folgt definiert:

$$
\begin{array}{ll}
& \textbf{hasInactiveSub}(\textbf{HS}, \textbf{iname}) \\
\leftrightarrow & \exists\ \textbf{SK}.\ \ \textbf{SK} \in \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.subplans} \\
& \qquad \wedge\ \textbf{isInactive}(\textbf{HS}, \textbf{nameGen}(\textbf{SK}, \textbf{iname}))
\end{array}
$$

$$
\begin{array}{ll}
& \textbf{hasSelectedSub}(\textbf{HS}, \textbf{iname}) \\
\leftrightarrow & \exists\ \textbf{SK}.\ \ \textbf{SK} \in \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.subplans} \\
& \qquad \wedge\ \textbf{isSelected}(\textbf{HS}, \textbf{nameGen}(\textbf{SK}, \textbf{iname}))
\end{array}
$$

$$
\begin{array}{ll}
& \textbf{hasReadySub}(\textbf{HS}, \textbf{iname}) \\
\leftrightarrow & \exists\ \textbf{SK}.\ \ \textbf{SK} \in \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.subplans} \\
& \qquad \wedge\ \textbf{isReady}(\textbf{HS}, \textbf{nameGen}(\textbf{SK}, \textbf{iname}))
\end{array}
$$

$$
\begin{array}{ll}
& \textbf{hasExecutedSub}(\textbf{HS}, \textbf{iname}) \\
\leftrightarrow & \exists\ \textbf{SK}.\ \ \textbf{SK} \in \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.subplans} \\
& \qquad \wedge\ \textbf{isExecuted}(\textbf{HS}, \textbf{nameGen}(\textbf{SK}, \textbf{iname}))
\end{array}
$$

$$
\begin{array}{ll}
& \textbf{hasExecOrSigReadySub}(\textbf{HS}, \textbf{iname}) \\
\leftrightarrow & \quad \textbf{hasExecutedSub}(\textbf{HS}, \textbf{iname}) \\
& \vee\ \exists\ \textbf{SK}.\ \ \textbf{SK} \in \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.subplans} \\
& \qquad \wedge\ \textbf{isReady}(\textbf{HS}, \textbf{nameGen}(\textbf{SK}, \textbf{iname})) \\
& \qquad \wedge\ \textbf{isActSig}(\textbf{HS}, \textbf{nameGen}(\textbf{SK}, \textbf{iname}))
\end{array}
$$

$$
\begin{array}{ll}
& \textbf{hasRestartableSub}(\textbf{HS}, \textbf{iname}) \\
\leftrightarrow & \exists\ \textbf{SK}.\ \ \textbf{SK} \in \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.subplans} \\
& \qquad \wedge\ \textbf{isAborted}(\textbf{HS}, \textbf{nameGen}(\textbf{SK}, \textbf{iname})) \\
& \qquad \wedge\ \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.retry-aborted}
\end{array}
$$

$$
\begin{array}{ll}
& \textbf{hasStartableSub}(\textbf{HS}, \textbf{iname}) \\
\leftrightarrow & \quad \textbf{hasRestartableSub}(\textbf{HS}, \textbf{iname}) \\
& \vee\ \textbf{hasInactiveSub}(\textbf{HS}, \textbf{iname})
\end{array}
$$

Die Prädikate werden definiert, damit der Sachverhalt formuliert werden kann, dass ein Unterplan existiert, der in einem gegebenen Zustand ist. Die Semantik ist dabei gemäß der Benennung des Prädikats definiert, das heißt, dass das Prädikat **hasInactiveSub** überprüft, ob ein Unterplan im Zustand `inactive` existiert. Besondere Bedeutung verdienen dabei die Prädikate **hasExecOrSigReadySub** und **hasRestartableSub**. Ersteres überprüft, ob ein Unterplan existiert, der entweder in der `Execution`-Phase ist oder im Zustand `ready` sein Aktivierungssignal erhalten hat. Dies bedeutet, dass der Unterplan gerade dabei ist, seinen Zustand nach `activated` zu verändern. **hasRestartableSub** testet, ob ein Unterplan existiert, der im Zustand `aborted` ist und neu-gestartet werden darf.

Analog zu diesen Prädikaten, die testen, ob ein Unterplan in einem gegebenen Zustand ist, werden die nachfolgenden Prädikate definiert. Diese bestimmen, ob alle Unterpläne eines

Plans in einem bestimmten Zustand sind.

Definition

Nachstehende Prädikate überprüfen, ob alle Unterpläne eines Plans einem bestimmten Zustand genügen:

$$
\begin{array}{ll}
\textbf{allSubsInactive} & : \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\
\textbf{allSubsSelected} & : \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\
\textbf{allSubsReady} & : \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\
\textbf{allSubsReadyOrRejected} & : \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool} \\
\textbf{allSubsTerminated} & : \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{bool}
\end{array}
$$

Deren Semantik ist wie folgt definiert:

$$
\begin{array}{ll}
 & \textbf{allSubsInactive}(\textbf{HS}, \textbf{iname}) \\
\leftrightarrow & \forall\ \textbf{SK}.\quad \textbf{SK} \in \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.subplans} \\
 & \qquad \rightarrow \textbf{isInactive}(\textbf{HS}, \textbf{nameGen}(\textbf{SK}, \textbf{iname}))
\end{array}
$$

$$
\begin{array}{ll}
 & \textbf{allSubsSelected}(\textbf{HS}, \textbf{iname}) \\
\leftrightarrow & \forall\ \textbf{SK}.\quad \textbf{SK} \in \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.subplans} \\
 & \qquad \rightarrow \textbf{isSelected}(\textbf{HS}, \textbf{nameGen}(\textbf{SK}, \textbf{iname}))
\end{array}
$$

$$
\begin{array}{ll}
 & \textbf{allSubsReady}(\textbf{HS}, \textbf{iname}) \\
\leftrightarrow & \forall\ \textbf{SK}.\quad \textbf{SK} \in \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.subplans} \\
 & \qquad \rightarrow \textbf{isReady}(\textbf{HS}, \textbf{nameGen}(\textbf{SK}, \textbf{iname}))
\end{array}
$$

$$
\begin{array}{ll}
 & \textbf{allSubsReadyOrRejected}(\textbf{HS}, \textbf{iname}) \\
\leftrightarrow & \forall\ \textbf{SK}.\quad \textbf{SK} \in \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.subplans} \\
 & \qquad \rightarrow \quad \textbf{isReady}(\textbf{HS}, \textbf{nameGen}(\textbf{SK}, \textbf{iname})) \\
 & \qquad\quad \vee\ \textbf{isRejected}(\textbf{HS}, \textbf{nameGen}(\textbf{SK}, \textbf{iname}))
\end{array}
$$

$$
\begin{array}{ll}
 & \textbf{allSubsTerminated}(\textbf{HS}, \textbf{iname}) \\
\leftrightarrow & \forall\ \textbf{SK}.\quad \textbf{SK} \in \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.subplans} \\
 & \qquad \rightarrow \textbf{isTerminated}(\textbf{HS}, \textbf{nameGen}(\textbf{SK}, \textbf{iname}))
\end{array}
$$

Auch diese Prädikate sind benannt nach den Zuständen, in denen die Unterpläne des angegebenen Plans sich befinden. **allSubsInactive** überprüft also, ob alle Unterpläne der angegebenen Planinstanz im Zustand `inactive` sind. Das Prädikat **allSubsReadyOrRejected** wird verwendet für den `parallel` Kontrolltyp. Dieser muss alle Unterpläne im Zustand `ready` synchronisieren und gemeinsam aktivieren. Pläne, die in der `Selection`-Phase zurückgewiesen werden, werden dabei ignoriert.

C.6 Planselektoren

Planselektoren dienen dazu, unter den Unterplänen einen geeigneten Plan auszuwählen. Beispielsweise soll die `sequential` Kontrolle nacheinander alle Pläne starten und aktivieren. Zu diesem Verhalten passt das Prädikat **selectFirstStartable**, das den ersten Unterplan zurückgibt, der startbar ist.

Definition

Folgende Planselektoren sind definiert, um unter den Unterplänen eines Plans einen geeigneten auszuwählen:

$$
\begin{array}{lcl}
\textbf{selectFirstStartable} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{plan-name} \\
\textbf{selectFirstRestartable} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{plan-name} \\
\textbf{selectAllStartable} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{plan-list} \\
\textbf{selectAllReady} & : & \textbf{hierarchy-state} \times \textbf{instance-name} \mapsto \textbf{plan-list} \\
\textbf{selectPrivilegedSub} & : & \textbf{hierarchy-state} \times \textbf{environment-aggregation} \\
 & & \times \textbf{instance-name} \mapsto \textbf{plan-name} \\
\textbf{selectFirstSub} & : & \textbf{plan-name} \mapsto \textbf{plan-name} \\
\textbf{selectSecondSub} & : & \textbf{plan-name} \mapsto \textbf{plan-name} \\
\textbf{secondSubPresent} & : & \textbf{plan-name} \mapsto \textbf{plan-name} \\
\textbf{noSecondSubPresent} & : & \textbf{plan-name} \mapsto \textbf{plan-name} \\
\textbf{selectFirstSub} & : & \textbf{instance-name} \mapsto \textbf{plan-name} \\
\textbf{selectSecondSub} & : & \textbf{instance-name} \mapsto \textbf{plan-name} \\
\textbf{secondSubPresent} & : & \textbf{instance-name} \mapsto \textbf{plan-name} \\
\textbf{noSecondSubPresent} & : & \textbf{instance-name} \mapsto \textbf{plan-name}
\end{array}
$$

Deren Semantik wird durch die folgenden Gleichungen beschrieben:

$$
\begin{array}{ll}
 & \textbf{selectFirstStartable}(\textbf{HS}, \textbf{iname}) \\
= & \textbf{pickFirstStartable}(\ \textbf{HS}, \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.subplans}, \\
 & \qquad \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.retry-aborted}, \textbf{iname}) \\
 & \\
 & \textbf{selectFirstRestartable}(\textbf{HS}, \textbf{iname}) \\
= & \textbf{pickFirstRestartable}(\ \textbf{HS}, \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.subplans}, \\
 & \qquad \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.retry-aborted}, \textbf{iname}) \\
 & \\
 & \textbf{selectAllStartable}(\textbf{HS}, \textbf{iname}) \\
= & \textbf{pickAllStartable}(\ \textbf{HS}, \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.subplans}, \\
 & \qquad \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.retry-aborted}, \textbf{iname}) \\
 & \\
 & \textbf{selectAllReady}(\textbf{HS}, \textbf{iname}) \\
= & \textbf{pickAllReady}(\textbf{HS}, \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.subplans}, \textbf{iname}) \\
 & \\
 & \textbf{selectPrivilegedSub}(\textbf{HS}, \textbf{EAGG}, \textbf{iname}) \\
= & \textbf{pickPrivilegedSub}(\textbf{EAGG}, \textbf{selectAllReady}(\textbf{HS}, \textbf{iname}), \textbf{iname}) \\
 & \\
 & \textbf{selectFirstSub}(\textbf{SK}) \\
= & \texttt{asbru}(\textbf{SK})\textbf{.subplans.first} \\
 & \\
 & \textbf{selectFirstSub}(\textbf{iname}) \\
= & \textbf{selectFirstSub}(\textbf{nameSel}(\textbf{iname})) \\
 & \\
 & \textbf{selectSecondSub}(\textbf{SK}) \\
= & \texttt{asbru}(\textbf{SK})\textbf{.subplans.rest.first} \\
 & \\
 & \textbf{selectSecondSub}(\textbf{iname}) \\
= & \textbf{selectSecondSub}(\textbf{nameSel}(\textbf{iname}))
\end{array}
$$

$$
\begin{array}{ll}
 & \textbf{secondSubPresent}(\textbf{SK}) \\
\leftrightarrow & \quad \texttt{asbru}(\textbf{SK}).\textbf{subplans.rest} \neq [] \\
 & \wedge\, \texttt{asbru}(\textbf{SK}).\textbf{subplans} \neq [] \\
 & \\
 & \textbf{secondSubPresent}(\textbf{iname}) \\
\leftrightarrow & \textbf{secondSubPresent}(\textbf{nameSel}(\textbf{iname})) \\
 & \\
 & \textbf{noSecondSubPresent}(\textbf{SK}) \\
\leftrightarrow & \neg\, \textbf{secondSubPresent}(\textbf{HS}, \textbf{SK}) \\
 & \\
 & \textbf{noSecondSubPresent}(\textbf{iname}) \\
\leftrightarrow & \textbf{noSecondSubPresent}(\textbf{nameSel}(\textbf{iname}))
\end{array}
$$

Das Prädikat **selectFirstStartable** wählt den ersten Plan der Liste der Unterpläne aus, der gestartet werden kann. Dabei kann es sich entweder um einen Plan im Zustand `inactive` oder um einen im Zustand `aborted`, unter der Voraussetzung, dass der übergeordnete Plan dieses Plan abgebrochene Pläne neu startet. **selectFirstRestartable** selektiert nur neu-startbare Pläne, gibt also den ersten Plan der Unterplanliste zurück, der im Zustand `aborted` ist. Das danach definierte **selectAllStartable** gibt alle Pläne der Unterplanliste zurück, die startbar sind, also im Zustand `inactive` oder neu-startbar. Analog dazu gibt **selectAllReady** alle Zustände zurück, die im Zustand `ready` sind. Das Prädikat **selectPrivilegedSub** selektiert einen bereiten Unterplan eines `anyorder` Plans. Dabei soll das Prädikat die Präferenzsignale des medizinischen Personals zur Aktivierung von Unterplänen berücksichtigen. **selectFirstSub** ist ein Prädikat, dass genau wie **selectSecondSub** technische Details der Selektion des ersten bzw. zweiten Unterplans kapselt. Die Prädikate **secondSubPresent** und **noSecondSubPresent** bestimmen, ob ein Plan einen zweiten Unterplan hat. Dies ist speziell für die `ifThenElse` Kontrolle wichtig.

Die Selektoren stützen die formale Semantikdefinition auf weitere Prädikate ab. Diese sollen im folgenden definiert werden.

Definition

Nachstehende Prädikate werden als Hilfsfunktionen der Unterplanselektoren verwendet.

$$
\begin{array}{lcl}
\textbf{pickFirstStartable} & : & \textbf{hierarchy-state} \times \textbf{bool} \times \textbf{plan-list} \times \textbf{instance-name} \\
 & & \mapsto \textbf{plan-name} \\
\textbf{pickFirstRestartable} & : & \textbf{hierarchy-state} \times \textbf{bool} \times \textbf{plan-list} \times \textbf{instance-name} \\
 & & \mapsto \textbf{plan-name} \\
\textbf{pickAllStartable} & : & \textbf{hierarchy-state} \times \textbf{bool} \times \textbf{plan-list} \times \textbf{instance-name} \\
 & & \mapsto \textbf{plan-name} \\
\textbf{pickAllReady} & : & \textbf{hierarchy-state} \times \textbf{plan-list} \times \textbf{instance-name} \\
 & & \mapsto \textbf{plan-name} \\
\textbf{pickPrivilegedSub} & : & \textbf{environment-aggregation} \times \textbf{plan-list} \times \textbf{instance-name} \\
 & & \mapsto \textbf{plan-name}
\end{array}
$$

Deren Semantik wird formal wie folgt definiert:

pickFirstStartable(**HS**, **bv**, **SK** + **PL**, **iname**)
= **SK**, wenn **isStartable**(**HS**, **bv**, **nameGen**(**SK**, **iname**))

pickFirstStartable(**HS**, **bv**, **SK** + **PL**, **iname**)
= **pickFirstStartable**(**HS**, **PL**, **iname**), sonst

pickFirstRestartable(**HS**, **SK** + **PL**, **iname**, **bv**)
= **SK**, wenn **isAborted**(**HS**, **iname**) und **bv**

pickFirstRestartable(**HS**, **bv**, **SK** + **PL**, **iname**)
= **pickFirstRestartable**(**HS**, **PL**, **iname**), sonst

pickAllStartable(**HS**, [], **iname**, **bv**)
= []

pickAllStartable(**HS**, **SK** + **PL**, **iname**, **bv**)
= **SK** + **pickAllStartable**(**HS**, **PL**, **iname**, **bv**),
wenn **isStartable**(**HS**, **bv**, **nameGen**(**SK**, **iname**))

pickAllStartable(**HS**, **SK** + **PL**, **iname**, **bv**)
= **pickAllStartable**(**HS**, **PL**, **iname**), sonst

pickAllReady(**HS**, [], **iname**)
= []

pickAllReady(**HS**, **SK** + **PL**, **iname**)
= **SK** + **pickAllReady**(**HS**, **PL**, **iname**), wenn **isReady**(**HS**, **nameGen**(**SK**, **iname**)

pickAllReady(**HS**, **SK** + **PL**, **iname**)
= **pickAllReady**(**HS**, **PL**, **iname**), sonst

pickPrivilegedSub(**EAGG**, **SK** + [], **iname**)
= **SK**

pickPrivilegedSub(**EAGG**, $\mathbf{SK}_1$ + **SK** + **PL**, **iname**)
= $\mathbf{SK}_1$, falls **planPreferred**(**EAGG**, **nameGen**($\mathbf{SK}_1$, **iname**))

pickPrivilegedSub(**EAGG**, $\mathbf{SK}_1$ + **SK** + **PL**, **iname**)
= **pickPrivilegedSub**(**EAGG**, **SK** + **PL**, **iname**),
falls nicht **planPreferred**(**EAGG**, **nameGen**($\mathbf{SK}_1$, **iname**))

Die Planselektoren wie etwa **selectAllReady** können am einfachsten rekursiv definiert werden. Für diese Definition werden Hilfsfunktionen verwendet. Dies ist notwendig, da die Methodensignatur von **selectAllReady** einen rekursiven Aufruf aufgrund eines fehlenden abnehmendem Parameter nicht zulässt. Die Funktion **pickAllReady** hingegen hat als Aufrufparameter nicht nur den Planinstanznamen des Planes, dessen Unterpläne untersucht werden sollen, sondern auch noch die Liste der Plannamen der Unterpläne, die noch nicht untersucht wurden. Aus diesen Plannamen lässt sich zusammen mit dem Planinstanznamen des übergeordneten Plans der Planinstanzname des Unterplans generieren und prüfen, ob dieser im geforderten Zustand ist oder nicht. Dazu sind drei Regeln notwendig. Ein Terminierungsfall, wenn keine weiteren Unterpläne mehr zu untersuchen sind sowie je ein positiver und negativer Rekursionsfall, falls der aktuelle Plan im geforderten Planzustand ist oder eben nicht.

Im Fall von **pickFirstStartable** und **pickFirstRestartable** sind keine Terminierungsfälle

spezifiziert. Ursache hierfür ist, dass der Selektor einen zu startenden bzw. neu zu startenden Plan zurückliefern soll. Existiert kein solcher Plan ist die Semantik undefiniert. Bei den SOS Regeln wurde allerdings darauf geachtet, dass diese Selektoren nur in Fällen zur Anwendung kommen, in denen vorher mit dem Prädikat **hasStartableSub** bzw. **hasRestartableSub** überprüft wurde, dass ein entsprechender Plan vorliegt. Bei der Auswahl des zu aktivierenden Unterplans der `anyorder` Kontrolle mittels des **pickPrivilegedSub** Selektors gilt die Besonderheit zu beachten, dass dieser den ersten Plan sucht, der vom Arzt ein Priorisierungssignal erhalten hat. gibt es keinen solchen Plan, so wird der letzte der Liste ausgewählt.

C.7 Erfüllbarkeit

Zum einfachen Zugriff werden als nächstes Prädikate definiert, die eine kompakte Definition der Erfüllbarkeit und Erfülltheit der sechs verschiedenen Asbru-Bedingungen eines Plans erlauben.

Definition

Zur kompakteren Definition der Erfüllbarkeit und Erfülltheit von Asbru-Bedingungen sind die Folgenden Prädikate definiert:

$$
\begin{array}{lcl}
\mathbf{satisfiable}_{\mathbf{filter}} & : & \mathbf{state} \times \mathbf{instance\text{-}name} \mapsto \mathbf{bool} \\
\mathbf{satisfiable}_{\mathbf{setup}} & : & \mathbf{state} \times \mathbf{instance\text{-}name} \mapsto \mathbf{bool} \\
\mathbf{satisfiable}_{\mathbf{suspend}} & : & \mathbf{state} \times \mathbf{instance\text{-}name} \mapsto \mathbf{bool} \\
\mathbf{satisfiable}_{\mathbf{reactivate}} & : & \mathbf{state} \times \mathbf{instance\text{-}name} \mapsto \mathbf{bool} \\
\mathbf{satisfiable}_{\mathbf{abort}} & : & \mathbf{state} \times \mathbf{instance\text{-}name} \mapsto \mathbf{bool} \\
\mathbf{satisfiable}_{\mathbf{complete}} & : & \mathbf{state} \times \mathbf{instance\text{-}name} \mapsto \mathbf{bool} \\
\mathbf{satisfied}_{\mathbf{filter}} & : & \mathbf{state} \times \mathbf{instance\text{-}name} \mapsto \mathbf{bool} \\
\mathbf{satisfied}_{\mathbf{setup}} & : & \mathbf{state} \times \mathbf{instance\text{-}name} \mapsto \mathbf{bool} \\
\mathbf{satisfied}_{\mathbf{suspend}} & : & \mathbf{state} \times \mathbf{instance\text{-}name} \mapsto \mathbf{bool} \\
\mathbf{satisfied}_{\mathbf{reactivate}} & : & \mathbf{state} \times \mathbf{instance\text{-}name} \mapsto \mathbf{bool} \\
\mathbf{satisfied}_{\mathbf{abort}} & : & \mathbf{state} \times \mathbf{instance\text{-}name} \mapsto \mathbf{bool} \\
\mathbf{satisfied}_{\mathbf{complete}} & : & \mathbf{state} \times \mathbf{instance\text{-}name} \mapsto \mathbf{bool}
\end{array}
$$

Deren Semantik ist definiert durch die nachstehenden Gleichungen:

$$
\begin{array}{lcl}
\mathbf{satisfiable}(\sigma, \mathbf{iname})_{\mathbf{filter}} & \leftrightarrow & \mathbf{satisfiable}(\texttt{asbru}(\mathbf{nameSel}(\mathbf{iname})).\mathbf{filter}, \sigma) \\
\mathbf{satisfiable}(\sigma, \mathbf{iname})_{\mathbf{setup}} & \leftrightarrow & \mathbf{satisfiable}(\texttt{asbru}(\mathbf{nameSel}(\mathbf{iname})).\mathbf{setup}, \sigma) \\
\mathbf{satisfiable}(\sigma, \mathbf{iname})_{\mathbf{suspend}} & \leftrightarrow & \mathbf{satisfiable}(\texttt{asbru}(\mathbf{nameSel}(\mathbf{iname})).\mathbf{suspend}, \sigma) \\
\mathbf{satisfiable}(\sigma, \mathbf{iname})_{\mathbf{reactivate}} & \leftrightarrow & \mathbf{satisfiable}(\texttt{asbru}(\mathbf{nameSel}(\mathbf{iname})).\mathbf{reactivate}, \sigma) \\
\mathbf{satisfiable}(\sigma, \mathbf{iname})_{\mathbf{abort}} & \leftrightarrow & \mathbf{satisfiable}(\texttt{asbru}(\mathbf{nameSel}(\mathbf{iname})).\mathbf{abort}, \sigma) \\
\mathbf{satisfiable}(\sigma, \mathbf{iname})_{\mathbf{complete}} & \leftrightarrow & \mathbf{satisfiable}(\texttt{asbru}(\mathbf{nameSel}(\mathbf{iname})).\mathbf{complete}, \sigma) \\
\mathbf{satisfied}(\sigma, \mathbf{iname})_{\mathbf{filter}} & \leftrightarrow & \mathbf{satisfied}(\ \texttt{asbru}(\mathbf{nameSel}(\mathbf{iname})).\mathbf{filter}, \sigma, \\
 & & \quad \sigma.\mathrm{EAGG}[\mathbf{iname}].\mathbf{confirm.filter}, \\
 & & \quad \sigma.\mathrm{EAGG}[\mathbf{iname}].\mathbf{override.filter}) \\
\mathbf{satisfied}(\sigma, \mathbf{iname})_{\mathbf{setup}} & \leftrightarrow & \mathbf{satisfied}(\ \texttt{asbru}(\mathbf{nameSel}(\mathbf{iname})).\mathbf{setup}, \sigma, \\
 & & \quad \sigma.\mathrm{EAGG}[\mathbf{iname}].\mathbf{confirm.setup}, \\
 & & \quad \sigma.\mathrm{EAGG}[\mathbf{iname}].\mathbf{override.setup}) \\
\mathbf{satisfied}(\sigma, \mathbf{iname})_{\mathbf{suspend}} & \leftrightarrow & \mathbf{satisfied}(\ \texttt{asbru}(\mathbf{nameSel}(\mathbf{iname})).\mathbf{suspend}, \sigma, \\
 & & \quad \sigma.\mathrm{EAGG}[\mathbf{iname}].\mathbf{confirm.suspend}, \\
 & & \quad \sigma.\mathrm{EAGG}[\mathbf{iname}].\mathbf{override.suspend})
\end{array}
$$

$$
\begin{aligned}
\textbf{satisfied}(\sigma, \textbf{iname})_{\textbf{reactivate}} \quad &\leftrightarrow \quad \textbf{satisfied}(\ \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.reactivate}, \sigma, \\
&\qquad\qquad \sigma.\text{EAGG}[\textbf{iname}]\textbf{.confirm.reactivate}, \\
&\qquad\qquad \sigma.\text{EAGG}[\textbf{iname}]\textbf{.override.reactivate}) \\
\textbf{satisfied}(\sigma, \textbf{iname})_{\textbf{abort}} \quad &\leftrightarrow \quad \textbf{satisfied}(\ \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.abort}, \sigma, \\
&\qquad\qquad \sigma.\text{EAGG}[\textbf{iname}]\textbf{.confirm.abort}, \\
&\qquad\qquad \sigma.\text{EAGG}[\textbf{iname}]\textbf{.override.abort}) \\
\textbf{satisfied}(\sigma, \textbf{iname})_{\textbf{complete}} \quad &\leftrightarrow \quad \textbf{satisfied}(\ \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.complete}, \sigma, \\
&\qquad\qquad \sigma.\text{EAGG}[\textbf{iname}]\textbf{.confirm.complete}, \\
&\qquad\qquad \sigma.\text{EAGG}[\textbf{iname}]\textbf{.override.complete})
\end{aligned}
$$

Das bedeutet, dass die Prädikate mit Hilfe des Instanznamens des Plans die Plandefinition nachschlagen und aus dieser dann die entsprechende Bedingung selektieren. Zusätzlich zur Selektion der Bedingung übernehmen diese Prädikate die Auswahl der korrekten Umgebungssignale um Eingriffe des medizinischen Personals zu beachten. Das heißt, handelt es sich das **satisfied** Prädikat mit der Ergänzung $_{\text{filter}}$, dann wird die **filter**-Bedingung des Plans zusammen mit den zugehörigen Umgebungssignalen ausgewertet. Analog verhält es sich mit den übrigen Selektoren.

Mit Hilfe dieser Prädikate werden weitere Prädikate definiert. Diese sollen die komplette Auswertung der Bedingung kapseln. Speziell werden dabei die Fälle der Terminierung und der Unterbrechung betrachtet. Bei diesen gibt es jeweils mehrere Bedingungen, die beachtet werden müssen. Beispielsweise muss für die Überprüfung des Abbruchs eines Plans sowohl die **abort**-Bedingung ausgewertet werden, als auch das **wait-for** Konstrukt sowie der Zustand des übergeordneten Plans.

Definition

Nachstehende Prädikate kapseln jeweils den Grund für einen bestimmten Planzustandswechsel.

$$
\begin{aligned}
&\textbf{shouldAbort} && : \ \sigma \times \textbf{instance-name} \mapsto \textbf{bool} \\
&\textbf{shouldComplete} && : \ \sigma \times \textbf{instance-name} \mapsto \textbf{bool} \\
&\textbf{shouldSuspend} && : \ \sigma \times \textbf{instance-name} \mapsto \textbf{bool} \\
&\textbf{shouldReactivate} && : \ \sigma \times \textbf{instance-name} \mapsto \textbf{bool} \\
&\textbf{shouldRejectFilter} && : \ \sigma \times \textbf{instance-name} \mapsto \textbf{bool} \\
&\textbf{shouldRejectSetup} && : \ \sigma \times \textbf{instance-name} \mapsto \textbf{bool} \\
&\textbf{shouldProceedFilter} && : \ \sigma \times \textbf{instance-name} \mapsto \textbf{bool} \\
&\textbf{shouldProceedSetup} && : \ \sigma \times \textbf{instance-name} \mapsto \textbf{bool}
\end{aligned}
$$

Deren Semantik ist wie folgt definiert:

$$
\begin{aligned}
\textbf{shouldAbort}(\sigma, \textbf{iname}) \quad \leftrightarrow \quad & \textbf{parent\_terminated}(\sigma.\text{HS}, \textbf{iname}) \\
& \vee \textbf{satisfied}(\sigma, \textbf{iname})_{\textbf{abort}} \\
& \vee \neg\, \textbf{satisfiable}(\ \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{wf}, \\
& \qquad\qquad \sigma.\text{HS}, \textbf{iname}, \\
& \qquad\qquad \texttt{asbru}(\textbf{nameSel}(\textbf{iname})) \\
& \qquad\qquad\qquad \textbf{.retry-aborted})
\end{aligned}
$$

$$
\begin{array}{lcl}
\textbf{shouldComplete}(\sigma, \textbf{iname}) & \leftrightarrow & \textbf{satisfied}(\sigma, \textbf{iname})_{\textbf{complete}} \\
 & & \wedge\ \textbf{satisfied}(\sigma.\mathrm{HS}, \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{wf}) \\
 & & \wedge\ \ \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\textbf{.wait-for-optional} \\
 & & \rightarrow \textbf{allSubsTerminated}(\sigma.\mathrm{HS}, \textbf{iname}) \\
\textbf{shouldSuspend}(\sigma, \textbf{iname}) & \leftrightarrow & \textbf{satisfied}(\sigma, \textbf{iname})_{\textbf{suspend}} \\
 & & \vee\ \textbf{parent\_suspended}(\sigma.\mathrm{HS}, \textbf{iname}) \\
\textbf{shouldReactivate}(\sigma, \textbf{iname}) & \leftrightarrow & \textbf{satisfied}(\sigma, \textbf{iname})_{\textbf{reactivate}} \\
 & & \wedge\ \neg\ \textbf{parent\_suspended}(\sigma.\mathrm{HS}, \textbf{iname}) \\
\textbf{shouldRejectFilter}(\sigma, \textbf{iname}) & \leftrightarrow & \neg\ \textbf{satisfiable}(\sigma, \textbf{iname})_{\textbf{filter}} \\
\textbf{shouldRejectSetup}(\sigma, \textbf{iname}) & \leftrightarrow & \neg\ \textbf{satisfiable}(\sigma, \textbf{iname})_{\textbf{setup}} \\
\textbf{shouldProceedFilter}(\sigma, \textbf{iname}) & \leftrightarrow & \textbf{satisfied}(\sigma, \textbf{iname})_{\textbf{filter}} \\
\textbf{shouldProceedSetup}(\sigma, \textbf{iname}) & \leftrightarrow & \textbf{satisfied}(\sigma, \textbf{iname})_{\textbf{setup}}
\end{array}
$$

Das Prädikat **shouldAbort** prüft, ob eine Abbruchbedingung vorliegt. Dies ist der Fall, wenn die **abort**-Bedingung eines Planes in der `Execution`-Phase wahr wird. Auch soll ein Plan abbrechen, wenn die Wartebedingung, die die notwendigen Unterpläne spezifiziert, nicht mehr erfüllbar ist und als dritten Grund für einen Abbruch gibt es den Abbruch des übergeordneten Plans, der den Unterplan ebenfalls terminieren lässt. Das im Sinne der Asbru-Semantik Inverse zu diesem Prädikat stellt das **shouldComplete** Prädikat dar. Dieses beschreibt den Zustand, in dem ein Plan seine Ausführung erfolgreich beenden darf. Dazu ist es notwendig, dass die **complete**-Bedingung des Plans erfüllt ist, auch muss die Wartebedingung erfüllt sein. Weiterhin dürfen keine Unterpläne mehr aktiv sein, wenn der Plan spezifiziert, dass er auf laufende Unterpläne wartet. Das Prädikat **shouldSuspend** kapselt die Bedingung, unter der ein Plan in der `Execution`-Phase seine Ausführung unterbrechen soll. Dazu ist zu überprüfen, ob die **suspend**-Bedingung erfüllt ist oder der übergeordnete Plan im terminierten Zustand ist. Dazu invers ist das **shouldReactivate**-Prädikat. Dieses überprüft, ob ein unterbrochener Plan wieder zurück in den `activated` Zustand versetzt werden soll. Dazu muss zum einem die **reactivate**-Bedingung erfüllt sein, zum anderen muss der übergeordnete Plan im Zustand `activated` sein.

Die Prädikate **shouldRejectFilter** und **shouldRejectSetup** sind jeweils Platzhalter für die negierte Erfüllbarkeit der jeweiligen **filter**- bzw. **setup**-Bedingung. Wie die Prädikate **shouldProceedFilter** und **shouldProceedSetup**, die Platzhalter für die Erfülltheit sind, werden diese Prädikate nur definiert, um eine einheitliche Benennung aller Erfüllungs-Bedingungen in den SOS-Regeln zu ermöglichen.

C.8 Planname Prädikate

Die nachfolgenden Prädikate definieren die Namensgebung von Asbru-Systemen. Einige Plankontrolltypen, wie etwa der `parallel` Kontrolltyp starten mehrere Unterpläne gleichzeitig. Um das Asbru-System beschreiben zu können, dass durch einen solchen parallelen Startvorgang entsteht, werden Prädikate definiert.

Definition

Zur Definition der Beziehung zwischen Plannamen und Asbru-Systemen sind nachstehende Prädikate definiert:

$\in$. : **instance-name** $\times$ **asbru-system** $\mapsto$ **bool**
planNames : **plan-list** $\times$ **instance-name** $\times$ **asbru-system** $\mapsto$ **bool**

Deren Semantik ist wie folgt definiert:

$$
\begin{array}{lcl}
\textbf{iname} \in \varepsilon & \leftrightarrow & \text{false} \\
\textbf{iname} \in \textbf{iname}_1 & \leftrightarrow & \textbf{iname} = \textbf{iname}_1 \\
\textbf{iname} \in \textbf{sys}_1 \|_s \textbf{sys}_2 & \leftrightarrow & \quad \textbf{iname} \in \textbf{sys}_1 \\
 & & \vee\ \textbf{iname} \in \textbf{sys}_2 \\
\textbf{planNames}(\textbf{PL}, \textbf{iname}, \textbf{sys}) & \leftrightarrow & \forall\ \textbf{SK}. \quad \textbf{SK} \in \textbf{PL} \\
 & & \qquad \leftrightarrow\ \textbf{nameGen}(\textbf{SK}, \textbf{iname}) \in \textbf{sys}
\end{array}
$$

Das $\in$ Prädikat beschreibt, wann ein Planname Element eines Asbru-Systems ist, d.h. wann ein Asbru-Plan dieses Namens in einer Hierarchie existiert. Dieses Prädikat wird als Hilfsprädikat für die Definition des **planNames** Prädikats verwendet. Dieses überprüft, ob die übergebene Asbru-Hierarchie genau diejenigen Pläne enthält, die über die Planliste angegeben werden. Um die Planinstanznamen dieser Pläne zu bilden, wird der Planinstanzname eines Plans mitgegeben. Die Pläne der Plannamensliste müssen Unterpläne dieser Planinstanz sein.

C.9 Prädikate spezifisch für einzelne Plantypen

Für `cyclical` und den `ifThenElse` Plantypen werden spezielle Prädikate verwendet, die die Anwendbarkeit bestimmter SOS Regeln überprüfen. Bei `cyclical` werden Prädikate angegeben, die testen, ob das Startintervall der zyklischen Bedingung in einem Zustand erreicht ist oder nicht. Weiterhin gibt es ein Prädikat, das testet, ob die notwendige Zahl an Wiederholungen des Unterplans der zyklischen Kontrolle erreicht wurde oder nicht. Zuletzt wird ein Prädikat definiert, dass die Auswertung des Konditionals des `ifThenElse` Plantyps übernimmt.

Definition

Zum Umgang mit den Plantypen `cyclical` und `ifThenElse` werden folgende Prädikate definiert:

isInTimeFrame : **state** $\times$ **instance-name** $\mapsto$ **bool**
hasCompletedRounds : **hierarchy-state** $\times$ **instance-name** $\mapsto$ **bool**
ITEP : **state** $\times$ **instance-name** $\mapsto$ **bool**

Deren Semantik wird wie folgt definiert:

$$
\begin{array}{l}
\textbf{isInTimeFrame}(\sigma, \textbf{iname}) \\
\leftrightarrow \quad \textbf{isCyclical}(\textbf{iname}) \\
\quad \wedge\ [\![\sigma.\text{AC} \in \texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\ \textbf{.type}]\!]_\sigma
\end{array}
$$

$$
\begin{array}{l}
\textbf{hasCompletedRounds}(\textbf{HS}, \textbf{iname}) \\
\leftrightarrow \quad \textbf{isCyclical}(\textbf{iname}) \\
\quad \wedge\ (\textbf{HS}[\textbf{iname}].\textbf{rounds} + 1) \leq \texttt{asbru}(\textbf{nameSel}(\textbf{iname})).\textbf{type.rounds}
\end{array}
$$

$$
\begin{array}{l}
\textbf{ITEP}(\sigma, \textbf{iname}) \\
\leftrightarrow \quad \textbf{isITE}(\textbf{iname}) \\
\quad \wedge\ [\![\texttt{asbru}(\textbf{nameSel}(\textbf{iname}))\ \textbf{.type.conditional}]\!]_\sigma
\end{array}
$$

Das heißt, das Prädikate **isInTimeFrame** kapselt die Auswertung der Element-Beziehung zwischen der Uhrzeit des Zustands und der Zeit-Annotation der zyklischen Plankontrolle. Das Prädikat **hasCompletedRounds** kapselt die Auswertung der Frage, ob ein zyklischer Plan seinen Unterplan oft genug ausgeführt hat. Das Prädikat **ITEP** wertet das Konditional einer `ifThenElse` Kontrolle aus.

C.10 Das sync-Prädikat

Die Ausführung von Asbru-Systemen verlangt, dass die Ergebnisse mehrerer Pläne zusammengeführt werden können. Jeder Plan kann in jedem Zustand seinen Planzustand abändern. Das bedeutet, dass multiple Aktualisierungen der **HS** Komponente des Zustands in einem Schritt möglich sein müssen. Um aus diesen Aktualisierungen einen Gesamtzustand zu berechnen, wird das Prädikat sync verwendet.

Das sync Prädikat wertet über vier Variablen vom jeweils gleichen Typ aus. Interpretiert werden muss es dabei mittels eines parallelen Systems und dessen Ergebniswerten. Zur Anschauung soll hier noch einmal die betreffende SOS Regel gezeigt werden:

$$\frac{\begin{array}{r}\mathbf{sys}_1 \xrightarrow{\sigma_1,\sigma_2} \mathbf{sys}_1' \\ \wedge\ \mathbf{sys}_2 \xrightarrow{\sigma_1,\sigma_3} \mathbf{sys}_2' \\ \wedge\ \mathsf{sync}(\sigma_1,\sigma_2,\sigma_3,\sigma_4)\end{array}}{\mathbf{sys}_1 \|_s\ \mathbf{sys}_2 \xrightarrow{\sigma_1,\sigma_4} \mathbf{sys}_1\text{'} \|_s\ \mathbf{sys}_2\text{'}}$$

Zwei Asbru-Systeme machen unabhängig voneinander jeweils einen Schritt. Ausgangszustand für diesen Schritt ist σ_1. Das erste Asbru-System berechnet einen Ergebniszustand σ_2, das zweite Asbru-System berechnet einen Ergebniszustand σ_3. Das sync Prädikat soll auf Basis des Ausgangszustands nun die beiden Ergebniszustände zusammenführen, so dass ein globaler Ergebniszustand σ_4 entsteht. Dabei muss speziell bei Asbru berücksichtigt werden, dass mehrere Systeme in ihren Schritten auf die gleiche Variable zugreifen, nämlich die **HS** Komponente des Zustands.

Definition

Das Prädikat sync wird verwendet, um gleichzeitige Aktualisierungen des Zustands durch parallele Prozesse konfliktfrei zusammenzuführen:

sync : **state** × **state** × **state** × **state** ↦ **bool**
sync : **asbru-state-history** × **asbru-state-history**
×**asbru-state-history** × **asbru-state-history** ↦ **bool**
sync : **hierarchy-state** × **hierarchy-state**
×**hierarchy-state** × **hierarchy-state** ↦ **bool**
sync : **patient-data-history** × **patient-data-history**
×**patient-data-history** × **patient-data-history** ↦ **bool**
sync : **environment-aggregation** × **environment-aggregation**
×**environment-aggregation** × **environment-aggregation** ↦ **bool**
sync : **bool** × **bool** × **bool** × **bool** ↦ **bool**
sync : **int** × **int** × **int** × **int** ↦ **bool**
sync : **plan-status** × **plan-status** × **plan-status** × **plan-status** ↦ **bool**

Die Semantik von sync wird durch die folgenden Gleichungen beschrieben:

$$
\begin{aligned}
& \mathsf{sync}(\sigma_1, \sigma_2, \sigma_3, \sigma_4) \\
\leftrightarrow\ & \quad \mathsf{sync}(\sigma_1.\mathrm{ASH}, \sigma_2.\mathrm{ASH}, \sigma_3.\mathrm{ASH}, \sigma_4.\mathrm{ASH}) \\
& \wedge\ \mathsf{sync}(\sigma_1.\mathrm{HS}, \sigma_2.\mathrm{HS}, \sigma_3.\mathrm{HS}, \sigma_4.\mathrm{HS}) \\
& \wedge\ \mathsf{sync}(\sigma_1.\mathrm{PDH}, \sigma_2.\mathrm{PDH}, \sigma_3.\mathrm{PDH}, \sigma_4.\mathrm{PDH}) \\
& \wedge\ \mathsf{sync}(\sigma_1.\mathrm{EAGG}, \sigma_2.\mathrm{EAGG}, \sigma_3.\mathrm{EAGG}, \sigma_4.\mathrm{EAGG}) \\
& \wedge\ \mathsf{sync}(\sigma_1.\mathrm{Tick}, \sigma_2.\mathrm{Tick}, \sigma_3.\mathrm{Tick}, \sigma_4.\mathrm{Tick}) \\
& \wedge\ \mathsf{sync}(\sigma_1.\mathrm{AC}, \sigma_2.\mathrm{AC}, \sigma_3.\mathrm{AC}, \sigma_4.\mathrm{AC})
\end{aligned}
$$

$$
\begin{aligned}
& \mathsf{sync}(\mathbf{ASH}_1, \mathbf{ASH}_2, \mathbf{ASH}_3, \mathbf{ASH}_4) \\
\leftrightarrow\ & \quad \mathbf{ASH}_1 = \mathbf{ASH}_2 \wedge \mathbf{ASH}_3 = \mathbf{ASH}_4 \\
& \vee\ \mathbf{ASH}_1 = \mathbf{ASH}_3 \wedge \mathbf{ASH}_2 = \mathbf{ASH}_4 \\
& \vee\ \mathbf{ASH}_2 = \mathbf{ASH}_3 \wedge \mathbf{ASH}_2 = \mathbf{ASH}_4
\end{aligned}
$$

$$
\begin{aligned}
& \mathsf{sync}(\mathbf{HS}_1, \mathbf{HS}_2, \mathbf{HS}_3, \mathbf{HS}_4) \\
\leftrightarrow\ & \forall\ \mathbf{iname}.\ \mathsf{sync}(\mathbf{HS}_1[\mathbf{iname}], \mathbf{HS}_2[\mathbf{iname}], \mathbf{HS}_3[\mathbf{iname}], \mathbf{HS}_4[\mathbf{iname}])
\end{aligned}
$$

$$
\begin{aligned}
& \mathsf{sync}(\mathbf{PDH}_1, \mathbf{PDH}_2, \mathbf{PDH}_3, \mathbf{PDH}_4) \\
\leftrightarrow\ & \quad \mathbf{PDH}_1 = \mathbf{PDH}_2 \wedge \mathbf{PDH}_3 = \mathbf{PDH}_4 \\
& \vee\ \mathbf{PDH}_1 = \mathbf{PDH}_3 \wedge \mathbf{PDH}_2 = \mathbf{PDH}_4 \\
& \vee\ \mathbf{PDH}_2 = \mathbf{PDH}_3 \wedge \mathbf{PDH}_2 = \mathbf{PDH}_4
\end{aligned}
$$

$$
\begin{aligned}
& \mathsf{sync}(\mathbf{EAGG}_1, \mathbf{EAGG}_2, \mathbf{EAGG}_3, \mathbf{EAGG}_4) \\
\leftrightarrow\ & \quad \mathbf{EAGG}_1 = \mathbf{EAGG}_2 \wedge \mathbf{EAGG}_3 = \mathbf{EAGG}_4 \\
& \vee\ \mathbf{EAGG}_1 = \mathbf{EAGG}_3 \wedge \mathbf{EAGG}_2 = \mathbf{EAGG}_4 \\
& \vee\ \mathbf{EAGG}_2 = \mathbf{EAGG}_3 \wedge \mathbf{EAGG}_2 = \mathbf{EAGG}_4
\end{aligned}
$$

$$
\begin{aligned}
& \mathsf{sync}(\mathbf{Tick}_1, \mathbf{Tick}_2, \mathbf{Tick}_3, \mathbf{Tick}_4) \\
\leftrightarrow\ & \quad \mathbf{Tick}_1 = \mathbf{Tick}_2 \wedge \mathbf{Tick}_3 = \mathbf{Tick}_4 \\
& \vee\ \mathbf{Tick}_1 = \mathbf{Tick}_3 \wedge \mathbf{Tick}_2 = \mathbf{Tick}_4 \\
& \vee\ \mathbf{Tick}_2 = \mathbf{Tick}_3 \wedge \mathbf{Tick}_2 = \mathbf{Tick}_4
\end{aligned}
$$

$$
\begin{aligned}
& \mathsf{sync}(\mathbf{AC}_1, \mathbf{AC}_2, \mathbf{AC}_3, \mathbf{AC}_4) \\
\leftrightarrow\ & \quad \mathbf{AC}_1 = \mathbf{AC}_2 \wedge \mathbf{AC}_3 = \mathbf{AC}_4 \\
& \vee\ \mathbf{AC}_1 = \mathbf{AC}_3 \wedge \mathbf{AC}_2 = \mathbf{AC}_4 \\
& \vee\ \mathbf{AC}_2 = \mathbf{AC}_3 \wedge \mathbf{AC}_2 = \mathbf{AC}_4
\end{aligned}
$$

$$
\begin{aligned}
& \mathsf{sync}(\theta_1, \theta_2, \theta_3, \theta_4) \\
\leftrightarrow\ & \quad\ \ \theta_1.\mathbf{activate} = \theta_2.\mathbf{activate} \wedge \theta_3.\mathbf{activate} = \theta_4.\mathbf{activate} \\
& \quad \vee\ \theta_1.\mathbf{activate} = \theta_3.\mathbf{activate} \wedge \theta_2.\mathbf{activate} = \theta_4.\mathbf{activate} \\
& \quad \vee\ \theta_2.\mathbf{activate} = \theta_3.\mathbf{activate} \wedge \theta_2.\mathbf{activate} = \theta_4.\mathbf{activate} \\
& \wedge\ \ \theta_1.\mathbf{rounds} = \theta_2.\mathbf{rounds} \wedge \theta_3.\mathbf{rounds} = \theta_4.\mathbf{rounds} \\
& \quad \vee\ \theta_1.\mathbf{rounds} = \theta_3.\mathbf{rounds} \wedge \theta_2.\mathbf{rounds} = \theta_4.\mathbf{rounds} \\
& \quad \vee\ \theta_2.\mathbf{rounds} = \theta_3.\mathbf{rounds} \wedge \theta_2.\mathbf{rounds} = \theta_4.\mathbf{rounds} \\
& \wedge\ \ \theta_1.\mathbf{state} = \theta_2.\mathbf{state} \wedge \theta_3.\mathbf{state} = \theta_4.\mathbf{state} \\
& \quad \vee\ \theta_1.\mathbf{state} = \theta_3.\mathbf{state} \wedge \theta_2.\mathbf{state} = \theta_4.\mathbf{state} \\
& \quad \vee\ \theta_2.\mathbf{state} = \theta_3.\mathbf{state} \wedge \theta_2.\mathbf{state} = \theta_4.\mathbf{state}
\end{aligned}
$$

Die verschiedenen Ergebniszustände gleichzeitiger Schritte unterschiedlicher Asbru-Systeme werden so zusammengeführt, dass die einzelnen Komponenten des Zustands getrennt voneinander betrachtet werden und getrennt voneinander zusammengeführt werden.

Eine Zusammenführung der **ASH** Komponenten des Zustands gestaltet sich recht einfach. Keine SOS-Regel verändert diese Komponente. Entsprechend muss die Ergebnisberechnung lediglich dafür sorgen, dass das Ergebnis der Zusammenführung gleich dem Ausgangswert ist, sofern keine Aktualisierungen durchgeführt werden. Um nicht pauschal Änderungen an der **ASH** Komponente zu verbieten, wird eine Definition gewählt, die aus drei Komponenten besteht. Sollte das erste Asbru-System keine Änderung durchführen, so ist der Ergebniswert der der zweiten Komponente. Analog ist der Ergebniswert der Zusammenführung der Ergebniswert der ersten Komponente, sofern der Ergebniswert der zweiten Komponente gleich dem Ausgangswert ist. Zuletzt wird noch angegeben, dass der Ergebniswert dem der ersten Komponente entspricht, sofern beide Komponenten wertgleich sind.

Informell bedeutet dies, hat nur eine Komponente eine Aktualisierung durchgeführt, so wird das Ergebnis dieser Aktualisierung als Ergebniswert genommen. Haben beide Komponenten identische Aktualisierungen durchgeführt, so ist das Ergebnis der Zusammenführung das Ergebnis einer der Komponenten. Trivialerweise führt diese Definition in jedem Fall dazu, das das Ergebnis gleich dem Ausgangswert ist, wenn keine Aktualisierungen vorgenommen wurden.

Komplexer gestaltet sich die Zusammenführung des Hierarchiezustands **HS**. Dieser kann nicht nur von einer Vielzahl verschiedener Systeme geändert werden, sogar einzelne Planstatus-Komponenten können gleichzeitig von verschiedenen Systemen geändert werden, wenn zusätzlich zu einem Planzustandswechsel das Aktivierungssignal durch den übergeordneten Plan gesetzt wird. Entsprechend muss die Definition abgestützt werden auf die Synchronisierung der Planstatuskomponenten der einzelnen Pläne.

Für die **PDH**, die **EAGG** sowie die **AC** Komponente gilt, dass diese von den Plänen nicht beschrieben werden. Daher wird die Synchronisierung genau so definiert wie bei der **ASH** Komponente. Das heißt, wenn eine Aktualisierung nur von einer Komponente des Systems durchgeführt wird, dann ist das Ergebnis der Zusammenführung das Berechnungsergebnis der ändernden Komponente. Wird keine Komponentenänderung durchgeführt, so ist das Ergebnis der Zusammenführung der Ausgangswert.

Ein Spezialfall stellt die **Tick** Komponente dar. Wie bereits beschrieben, soll die Umgebung die **Tick**-Komponente vor jedem Schritt auf den Wert **true** setzen. Während der Ausführung setzen alle Pläne, die einen Mikro-Schritt erzwingen wollen, die **Tick** Komponente auf den Wert **false**. Es tritt hier also der Fall auf, dass mehrere Pläne die gleiche Aktualisierung der Komponente vornehmen können. Dies wird durch das sync-Prädikat zugelassen. In diesem Fall wird der aktualisierte Wert das Ergebnis der Zusammenführung.

Die Zusammenführung der Planstatus-Komponenten wird durch die Zusammenführung der einzelnen Werte des Planstatus-Tripel definiert. Das heißt, unabhängig von den anderen Werten des Tripels wird geprüft, ob jeder der drei Werte nur von einem Asbru-System geändert wurde bzw. ob beide Asbru-Systeme die gleiche Änderung durchgeführt haben. In diesen Fällen wird jeweils der aktualisierte Wert als Ergebnis der Zusammenführung genommen. Wird keine Aktualisierung durchgeführt, so ist der Ergebniswert wieder genau der Ausgangswert der Zusammenführung.

D Bewiesene Aussagen

In diesem Abschnitt werden alle Aussagen vorgestellt, die bezüglich der Brustkrebs-Leitlinie bewiesen wurden. Die technischen Details einiger ausgewählter Aussagen beschreibt Teil V dieser Arbeit.

Zusätzlich zu den Aussagen der Brustkrebs-Leitlinie wird eine Aussage einer Leitlinie zur Behandlung von Gelbfieber bei Neugeborenen in Abschnitt D.12 aufgeführt. Das Asbru-Modell der Brustkrebs-Leitlinie enthält keine Zeit-Annotationen. Daher kann innerhalb dieses Modells der Beweis von Aussagen, deren Korrektheit von der Auswertung von Zeit-Annotationen abhängen, nicht gezeigt werden. Um auch dieses Vorgehen vorzustellen, wird eine Aussage über dem Asbru-Modell der Gelbfieber-Fallstudie gezeigt.

D.1 Aussage 1

Aussage 1 verlangt, dass Patienten, die an metastasierendem Krebs leiden, nicht chirurgisch behandelt werden sollen. Die Anpassung der Aussage an das Leitlinien-Modell laut [61] geschieht an das erste Kapitel der Fallstudie.

Die Vorbedingung der Aussage ist so, wie sie von [61] vorgestellt wird nicht korrekt. Dort wird davon ausgegangen, dass metastasierender Brustkrebs immer dann vorliegt, wenn die Diagnose auf „metastasised-BC“ lautet oder Metastasen in den Lymphknoten der Achseln nachgewiesen werden können. Nach Konsultation mit Medizinern stellt sich heraus, dass der zweite Teil dieser Anpassung falsch ist, da der Begriff des „metastasierendem Krebs“ nur ferne Metastasen beschribt, also beispielsweise Metastasen in Lunge, Leber oder Knochenmark. Metastasen in den Achselknoten werden als spezielle Ausprägung von lokal begrenztem Brustkrebs angesehen.

Mit der Änderung der Aussage, gelingt der Nachweis. Dabei wird das Startereignis auf die Aktivierung des `chapter1`-Plans gelegt, das Endereignis auf die Beendigung dieses Plans. Zu zeigen ist, dass keine chirurgische Behandlung in diesem Intervall stattfindet.

Eine zusätzliche Annahme im Sinne des Hintergrundwissens ist, dass die Diagnose der Krebsart sich nicht ohne Behandlung ändert.

Mit der Diagnose „metastasised-BC“ wird die Anwendung des `chapter1`-Plans verhindert. Stattdessen verbleibt dieser aufgrund seiner **filter**-Bedingung unendlich lang in seiner `Selection`-Phase, ohne dass Startereignis oder Endereignis eintreten. Damit kann die Aussage nachgewiesen werden.

Trotz erfolgreicher Verifikation zeigt der Nachweis dieser Aussage eine Anomalie des Asbru-Modells. Es ist unnötig, bei Einstieg in das Kapitel die Diagnose mehrfach zu überprüfen. Der Plan sollte stattdessen nach der ersten Überprüfung seine Ausführung fortsetzen oder zurückgewiesen werden. Weiterhin wird offensichtlich, dass die Aussage noch immer nicht richtig angepasst sein kann. Tatsächlich sollte die Aussage nicht über dem Modell von Kapitel 1 nachgewiesen werden, in dem der metastasierende Brustkrebs nicht behandelt wird. Stattdessen käme Kapitel 5 für die Anpassung der Aussage in Betracht. Dort existiert jedoch kein Plan, der chirurgische Eingriffe repräsentiert. Damit ist die Aussage auch bei richtiger Anpassung an das Modell trivial wahr. Technische Details dieser Aussage werden in Abschnitt 24 beschrieben.

D.2 Aussage 5

Aussage 5 verlangt, dass Frauen, die vor der Menopause stehen, mittels Chemotherapie behandelt werden, sofern der Krebs Östrogen-Rezeptor positiv ist und die Lymphknoten der Achseln von Metastasen befallen sind.

Zunächst schlägt der Beweis der Aussage fehl. Grund dafür ist der Spezialfall der Behandlung schwangerer Frauen. Diese werden, sofern dies möglich ist, nicht mittels Chemotherapie behandelt, um das Kind nicht zu schädigen. Es kann daher nur eine eingeschränkte Aussage gezeigt werden. Diese sagt aus, dass die ursprüngliche Aussage zumindest für nicht-schwangere Frauen eingehalten wird.

Die Aussage wird bezüglich des Modells des zweiten Kapitels der Fallstudie gezeigt. Die Aussage wird dahingehend formalisiert, dass für Frauen, die die Vorbedingung erfüllen, stets einer der Chemotherapie-Pläne `TAC6` oder `FEC5` gestartet wird.

Aufgrund der speziellen Struktur des zweiten Kapitels der Fallstudie kann darauf verzichtet werden, die Beweistechnik der Beweis-Dekomposition einzusetzen. Stattdessen kann die Aussage direkt mit symbolischer Ausführung bewiesen werden. Dies ist bedingt dadurch, dass das zweite Kapitel der Fallstudie einen in Asbru kodifizierten Entscheidungsbaum darstellt. Dieser hat nicht die aus dem ersten Kapitel der Fallstudie bekannten Quellen des Indeterminismus durch `anyorder` kontrollierte Pläne. Stattdessen werden innerhalb des Kapitels nur deterministisch Entscheidnungen auf Basis des angegebenen Zustands des Patienten getroffen. Dieser Zustand wird mit Hilfe von Annahmen an die Patientenakte gespeichert. Im Fall von Aussage 5 wird etwa der Fakt, dass die Lymphknoten der Achseln von Krebs befallen sind mittels der Zustandsbeschreibung

```
PDH[AC] .BC-nodes = positive,
```

gespeichert. Weiterhin wird angenommen, dass sich weder die Krankenakte noch die Uhrzeit während eines Mikro-Schritts ändern. Somit bleiben die initialen Annahmen über den Zustand des Patienten bis zum ersten Makro-Schritt erhalten. Der Entscheidungsbaum selbst wird innerhalb von Mikro-Schritten durchlaufen. Dadurch geht die Information über den Patientenzustand erst nach Abarbeiten des Entscheidungsbaums mit Einsetzen der Behandlungen verloren.

Die Aussage wird als eine „observe-during-period“ Aussage formuliert, wobei der Beobachtungszeitraum mit dem Start des ersten Plans `atoBC` beginnt und erst mit der Beendingung des Plans endet. Während dieses Intervalls muss beobachtet werden, dass Chemotherapie gestartet wird.

Die Aussage kann durch symbolische Ausführung nahezu vollautomatisch bewiesen werden Lediglich bei Anwendung der Induktionen sind händische Generalisierungen notwendig. Insgesamt sind für den Beweis weniger als zehn Interaktionen nötig.

D.3 Aussage 6

Aussage 6 verlangt, dass Frauen, deren Lymphknoten nicht mit Krebs befallen sind und deren Haupttumor als risikoarm eingestuft wird, keine systematische Behandlung erfahren.

Diese Aussage wird, wie auch die Aussage 5 über dem zweiten Kapitel der Fallstudie gezeigt, da dieses die systematische Behandlung von lokal begrenztem Krebs beschreibt. Mittels Annahmen an die Krankenakte wird der Zustand der Patientin modelliert. Das Modell der Leitlinie bietet dabei keine direkte Möglichkeit, die Risikoarmut des Krebses zu formalisieren. Die Mediziner gehen davon aus, dass diese am besten dadurch auszudrücken ist, dass der Tumor „klein“ ist und der Differentierungsgrad den Wert „BRI“ hat.

Für Patienten mit dieser Vorbedingung soll gelten, dass vom Start bis zum Ende der Behandlung durch das zweite Kapitel keine Chemotherapie oder Endokrine Therapie ausgeführt wird. Zu beachten ist dabei, dass durch die vielen auseinander- und zusammenlaufenden Plan/Unterplan Beziehungen im zweiten Kapitel sehr viele Planinstanznamen zu überwachen sind.

Die Formalisierung der Aussage verlangt, dass eine boolesche Variable (`Observed`) immer dann auf den Wert **true** gesetzt wird, wenn eine beliebige Planinstanz gestartet wird, die die Behandlung mit Chemotherapie oder Endokriner Therapie symbolisiert. Dabei existieren allein für den Plan `FEC5`, der die Behandlung mittels Chemotherapie darstellt, sieben verschiedene Aufrufpfade und damit sieben verschiedene Planinstanzen. Insgesamt müssen 17 Planinstanznamen überwacht werden.

Die so formulierte Aussage ist nicht beweisbar. Im Fall der Behandlung schwangerer Patienten kann auch bei der gegebenen Vorbedingung die Chemotherapie gestartet werden. Schwangerer Frauen, die sich im zweiten oder dritten Drittel der Schwangerschaft befinden und bei denen eine Behandlung nicht verzögerbar ist, werden nach der Leitlinie stets mit Chemotherapie behandelt. Hierbei handelt es sich um den einzigen Fall, bei dem, die Aussage verletzt wird.

Daher wird eine zusätzliche Annahme zu den Vorbedingungen zugefügt. Diese sagt aus, dass die Behandlung der Patientin, sofern sie schwanger ist, verzögert werden darf. Mit dieser zusätzlichen Annahme ist die Aussage beweisbar. Im Anschluss an die Verifikation wird den Medizinern diese zusätzliche Annahme präsentiert, damit sie entweder der Formalisierung der Aussage zustimmen, oder die Leitlinie verbessern können.

Die Aussage kann vollständig automatisch, also ohne eine einzige Interaktion gezeigt werden.

D.4 Aussage 7

Aussage 7 ist eng verwandt mit Aussage 6 und verlangt, dass keine systematische Therapie angewendet wird, wenn eine Patientin älter als 65 Jahre ist und an Östrogenrezeptor-negativem Brustkrebs leidet.

Prinzipiell ist das Gerüst zur Verifikation der Aussage von Aussage 6 zu übernehmen, wobei lediglich die Patientengruppe ausgetauscht werden muss. Um die Aussage beweisen zu können, ist es notwendig, gewisses Hintergrundwissen zu formalisieren. Beispielsweise ist es wichtig zu wissen, dass 65-jährige Frauen Postmenopausal sind und außerdem nicht schwanger sein können. Bei der Anpassung an die Fallstudie ergibt sich noch eine zweite Komplikation. In der niederländischen Leitlinie werden drei Altersbereiche für Frauen unterschieden, diese sind „young“, „middle-aged“ und „old“. Nach der Leitlinie fallen in die letzte Kategorie alle Frauen, die älter als 70 Jahre sind. Damit sind einige Patienten aus der von der Aussage vorgegebenen Patientengruppe nach der Definition der Leitlinie „middle-aged“. Diese Patienten werden mit Chemotherapie behandelt.

Entsprechend muss zum Nachweis der Korrektheit hier davon ausgegangen werden, dass die Patienten alle das symbolische Alter „old“ erreicht haben. In der Diskussion mit den Medizinern ergibt sich, dass diese Annahme gerechtfertigt ist. Die beiden unterschiedlichen Altersgrenzen mit 65 Jahren einerseits und 70 Jahren andererseits ergeben sich daraus, dass die Aussage aus einer amerikanischen Leitlinie stammt. Innerhalb dieser Leitlinie wird eine andere Kosten/Nutzen Abwägung zwischen realen Kosten und den Nebenwirkungen der Chemotherapie einerseits und der zu erwartenden Steigerung der Lebenserwartung andererseits getroffen.

Wie bei Aussage 6 kann beim Beweis von Aussage 7 eine vollständige Automatisierung des Beweises erreicht werden.

D.5 Aussage 13

Aussage 13 verlangt, dass eine Patientin vor Durchführung einer Brustentfernung über die Möglichkeit zur Rekonstruktion der Brust informiert werden muss. Gezeigt werden soll diese Aussage über dem Modell des ersten Kapitels der Fallstudie. Zum Beweis ist die Technik der Beweis-Dekomposition notwendig.

Aufgrund der fehlenden Struktur der Behandlungsabfolgen im Modell des ersten Kapitels der Leitlinie kann diese Aussage nicht ohne weiteres gezeigt werden. Es ist dabei notwendig, Hintergrundwissen in die Formulierung der Aussage mit einzubauen. Die formalisierte Aussage wird um eine Umgebungsannahme erweitert, die aussagt, dass sich der Zustand des Plans zur Patienten-Information nach dem Starten des chirurgischen Eingriffs nicht mehr ändert. Dies entspricht informell der Forderung, dass eine Patientin, wenn sie überhaupt aufgeklärt wird, vor Beginn der OP aufgeklärt werden muss, da eine nachträgliche Information nutzlos ist.

Darüber hinaus müssen weitere Annahmen getroffen werden. Das Modell des ersten Kapitels verlangt eine Information des Patienten über die Brustrekonstruktion nur dann, wenn der Patient keine brusterhaltende Therapie wünscht. Im Rahmen von Aussage 35 wird bewiesen, dass nur eine Patientin, die die Brustentfernung wünscht, diese auch erhalten wird. Um diese Aussage beweisen zu können sind zusätzliche Annahmen notwendig, da das Modell des ersten Kapitels Anomalien enthält. Speziell geht es dabei um den Referenzpunkt einer Zeitannotation, der den Planzustandswechsel eines nicht existierenden Plans referenziert. Bezüglich dieses Planzustandswechsels müssen die gleichen Annahmen getroffen werden, wie beim Beweis der Aussage 35. Dieser wird in Abschnitt 22 beschrieben.

Zu beachten ist dabei weiterhin, dass die Mediziner im Rahmen der Zusatzannahmen des Beweises der Aussage 35 angegeben haben, dass die brusterhaltende Therapie während der Behandlung ohne Eingriffsmöglichkeiten des Patienten zu einer Mastektomie, also einer kompletten Brustentfernung werden kann. Dies wird aus dem Modell nicht ersichtlich. Kommt es zu dem Fall, dass eine als brusterhaltende Therapie gestartete Operation als Brustentfernung endet, so wird die Aussage verletzt. Durch die gewählte Anpassung der Aussage an das Modell wird die Aussage 13 jedoch in diesem Fall dennoch als erfüllt angesehen.

Beim Beweis ergeben sich nun im wesentlichen drei Fälle. Als erstes der, in dem keine Mastektomie durchgeführt wird und die Aussage somit erfüllt ist. Als zweites gibt es den Fall, in dem bei der Patientin eine komplette Brustentfernung durchgeführt wird und die Patientin im Vorfeld korrekt informiert wurde. Als drittes gibt es den Problemfall, in dem die Patientin operiert wird, ohne dass eine vorherige Information stattgefunden hat. In diesem Fall kann die Behandlung aber niemals abgeschlossen werden. Zum Abschluss der Behandlung ist der Abschluss der Operation und der Patienten-Information notwendig. Die zusätzliche Umgebungsannahme verhindert jedoch, dass der Plan zur Patienten-Information in diesem Zustand abschließen kann. Da in diesem Fall nie das Endereignis eintritt, wird die Aussage nicht verletzt.

D.6 Aussage 14

Aussage 14 verlangt, dass alle Patientinnen, die an DCIS leiden nach der chirurgischen Entfernung des Gewebes mit Strahlentherapie behandelt werden. Es gibt zur chirurgischen Behandlung von DCIS nur zwei Möglichkeiten, zum Einen die brusterhaltende Therapie, zum Anderen die Brustentfernung. Die Aussage muss für beide Fälle bewiesen werden.

Zu diesem Zweck wird die Aussage in zwei Teilaussagen zerlegt und getrennt bewiesen. Teil I beweist, dass im Falle der brusterhaltenden Therapie immer Strahlentherapie auf den chirurgischen Eingriff folgt, Teil II der Aussage zeigt dies analog für die Brustentfernung.

Die beiden Aussagen können gezeigt werden, da der erfolgreiche Abschluss der Strahlentherapie sowohl Bedingung für den Abschluss der brusterhaltenden Therapie ist, wie auch für den Abschluss der Brustentfernung. Dies wird durch die Wartebedingungen der chirurgischen Pläne sichergestellt.

Zum Beweis des ersten Teils der Aussage wird die Beweis-Dekomposition eingesetzt, während für den Beweis der zweiten Aussage die symbolische Ausführung alleine eingesetzt werden kann.

Der Unterschied bei der Anwendung der unterschiedlichen Beweistechniken ergibt sich daraus, dass die relevante Planhierarchie im ersten Teil der Aussage signifikant größer ist als im zweiten Teil. Einmal sind insgesamt 14 Pläne in drei Hierarchieebenen beteiligt, im anderen Fall lediglich fünf Pläne in zwei Hierarchieebenen. Die Entscheidung für bzw. gegen die Beweis-Dekompositionstechnik basiert auf einer Abschätzung, dass im Fall des Beweises der zweiten Teils von Aussage 14 der Aufwand für das finden der korrekten Invarianten höher liegt als die direkte symbolische Ausführung.

D.7 Aussage 19, Teil II

Aussage 19, Teil II verlangt, dass Patientinnen, bei denen die Achselknoten auf Metastasen untersucht werden danach nicht mittels Strahlentherapie behandelt werden. Dabei gilt, dass die Strahlentherapie und die Untersuchung der Achselknoten auf Metastasen zwei sich wechselseitig ausschließende Behandlungen sind. Das bedeutet, dass die Leitlinie nach Beendigung der einen Behandlung die andere Behandlung nicht mehr starten wird.

Die Aussage wird als eine avoid-during-period Aussage formuliert und bewiesen, bei der der Start des Strahlentherapie Plans nicht mehr beobachtet werden darf, nachdem die Untersuchung der Achselknoten abgeschlossen wurde. Das Startereignis fällt zusammen mit dem Beenden des Plans zur Untersuchung der Achsel. Das Endereignis fällt zusammen mit dem Ende der Behandlung. Nicht beobachtet werden darf der Start der Strahlentherapie nach dem Eintritt des Startereignis.

Um den Beweis durchzuführen wird die Technik der Beweisdekomposition angewendet. Dabei werden die Pläne der Untersuchung der Achselknoten und der Strahlentherapie dahingehend abstrahiert, dass diese Pläne nur nach Senden des Aktivierungssignal in den Zustand `activated` übergehen können und in diesem niemals abbrechen. Das heißt, die Pläne bleiben so lange im Zustand `activated`, bis sie in den Zustand `completed` wechseln wodurch dann ausgeschlossen wird, dass der jeweils andere Plan noch gestartet wird.

Mit dieser Abstraktion gelingt der Beweis der Aussage.

D.8 Aussage 20

Aussage 20 verlangt, dass nach der Entfernung vom Krebs befallener Lymphknoten der Achsel die Achsel nicht bestrahlt werden soll. Dies kann aufgrund der Wartebedingung und des Kontrolltyps des Plans zur Behandlung der Achselknoten garantiert werden. Der Beweis der Aussage wird detailliert in Abschnitt 21 beschrieben.

D.9 Aussage 35

Aussage 35 verlangt, dass der Wunsch des Patienten nach brusterhaltender Therapie bzw. kompletter Brustentfernung respektiert wird, soweit dies medizinisch möglich ist.

Um diese Aussage zu beweisen sind komplexe Zusatzannahmen notwendig, da im ersten Kapitel der Fallstudie Planzustandsübergänge von Plänen referenziert werden, die dort nicht

existieren. Mit entsprechenden Annahmen ist es dennoch möglich, die Aussage zu beweisen. Der Beweis dieser Aussage wird detailliert in Abschnitt 22 beschrieben.

D.10 Aussage 37

Aussage 37 verlangt, dass Endokrine Therapie nur nach einer Chemotherapie gestartet wird. Wörtlich lautet die Aussage „Endocrine therapy must be started only after chemotherapy is completed“. In Zusammenarbeit mit Medizinern wird diese Aussage so interpretiert, dass es durchaus zulässig ist, Endokrine Behandlungen durchzuführen, wenn keine Chemotherapie gestartet wurde. Die Aussage verlangt nur, dass die Reihenfolge eingehalten wird, wenn beide Behandlungen angewendet werden.

Die Aussage wird im zweiten Kapitel der Fallstudie gezeigt. Wie auch bei den anderen Beweisen in diesem Kapitel reicht symbolische Ausführung alleine aus, um die Aussage zeigen zu können. Die schiere Größe des Beweise bringt allerdings technische Probleme mit sich. Um die Zeit, die zum Beweis nötig ist zu verkleinern, wird der Beweisbaum in verschiedene Teile zerlegt, die jeweils unabhängig voneinander auf verschiedenen Computern gleichzeitig berechnet werden können.

Wie auch bei den anderen Aussagen, die über dem Modell des zweiten Kapitels der Leitlinie gezeigt wurden, gilt auch hier, dass der Automatisierungsgrad extrem hoch ist. Lediglich Abstraktionen vor der Anwendung von temporallogischer Induktion müssen von Hand durchgeführt werden. Speziell bei Aussage 37 muss zusätzlich noch händisch entschieden werden, an welchen Stellen Unteraussagen gebildet werden müssen, die dann in getrennten Beweisen gezeigt werden sollen.

Die Berechnungen der Beweise wurden über Nacht auf vier unterschiedlichen Prozessoren durchgeführt. Die Rechenzeit betrug insgesamt weniger als 4*10 Stunden.

D.11 Aussage 38

Aussage 38 verlangt, dass alle Patientinnen, die mit brusterhaltender Therapie behandelt werden, zusätzlich mit Strahlentherapie behandelt werden sollen. Der Beweis der Aussage kann erfolgreich durchgeführt werden. Er wird detailliert in Abschnitt 23 beschrieben.

D.12 Jaundice-Indikator

Da die Brustkrebs-Fallstudie keine komplexen Zeit-Annotationen enthält, wird zum Test des Umgangs des Kalküls mit Zeit-Annotationen eine Aussage aus der Jaundice-Fallstudie bewiesen. Dabei handelt es sich um den Indikator, der formuliert wurde für einen Plans zur Behandlung eines Säuglings mit Licht-Therapie. Dieser Indikator verlangt, dass der Behandlungsplan abbrechen soll, wenn er nicht anschlägt, sich also der Zustand des Säuglings nach einer Karenzzeit von vier Stunden nach Beginn der Licht-Therapie nicht bessert. Nach Ablauf dieser vier Stunden muss für die nächsten zwei Stunden eine konstante Besserung des Patientenzustands nachgewiesen, oder der Plan abgebrochen werden.

Der Beweis dieser Aussage wird mittels symbolischer Ausführung durchgeführt. Der Beweis gliedert sich in mehrere Teile, zunächst den Aktivierungsteil, in dem der Plan zur Behandlung und dessen Unterplan gestartet und aktiviert werden. Sobald der Plan und sein Unterplan im Zustand `activated` sind beginnt der zweite Teil des Beweises, in dem auf das Eintreten des Startereignisses gewartet wird, das vier Stunden nach dem Planstart liegt. Sobald das Startereignis vorliegt wird im nächsten Teil des Beweises auf das Eintreten des Endereignisses gewartet, welches sechs Stunden nach Planstart auftritt. In dieser Phase muss in jedem Schritt

nachgewiesen werden, dass der Zustand des Patienten sich im letzten Schritt gebessert hat oder aber der Behandlungsplan seine Bearbeitung abbricht.

Im Letzten Teil, der mit dem Eintritt des Endereignisses beginnt, muss nichts mehr gezeigt werden, stattdessen wird in diesem Teil nur noch auf die Terminierung des Behandlungsplans gewartet.

Der Beweis dieser Aussage wird detailliert in [62] vorgestellt.

Literaturverzeichnis

[1] Implementing clinical practise guidelines. Effective Health Care Bulletin, 1994. York: University of York.

[2] Monatsschrift Kinderheilkunde, 2004. http://www.dakj.de/index.php?id=92.

[3] G. Berry A. Benveniste. The synchronous approach to reactive and real-time systems. In *Another Look at Real-time programming*, Proceeding of IEEE, 1991.

[4] M. Abadi and L. Lamport. Conjoining Specifications. *ACM Transactions on Programming Languages and Systems*, pages 507–534, 1995.

[5] L. Aceto, W. Fokkink, and C. Verhoef. Structural operational semantics. In *Handbook of Process Algebra*, pages 197–292. Elsevier, 2001.

[6] L. Anselma, P. Terenziani, S. Montani, and A. Bottrighi. Towards a comprehensive treatment of repetitions, periodicity and temporal constraints in clinical guidelines. *Artificial Intelligence in Medicine*, 38(2):171–195, 2006.

[7] A.Seyfang, S.Miksch, and et al. Deliverable D2.2a: Specification of formats of Intermediate, Asbru and KIV representations. Project Deliverable D2.2a, University of Augsburg, September 2005. URL: `http://www.protocure.org/deliverables.html/`.

[8] M. Balser. *Verifying Concurrent System with Symbolic Execution – Temporal Reasoning is Symbolic Execution with a Little Induction.* PhD thesis, University of Augsburg, Augsburg, Germany, 2005.

[9] M. Balser, C. Duelli, and W. Reif. Formal semantics of Asbru – an overview. In *Proceedings of IDPT 2002*. Society for Design and Process Science, 2002.

[10] S. Bäumler, A. Dunets, et al. Deliverable D4.3: Model checking of Asbru. Project Deliverable D4.3, University of Augsburg, July 2006. URL: `http://www.protocure.org/deliverables.html/`.

[11] S. Bäumler, J. Schmitt, and W. Reif. Dealing with synchronous parallelism - the asbru case study, 2008? not yet submitted, conference unknown.

[12] M. Ben-Ari. *Mathematical logic for computer science.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[13] J. Bengtsson, K. Guldstrand Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL in 1995. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 431–434, 1996.

[14] N. Bjørner, A. Browne, M. Colón, B. Finkbeiner, Z. Manna, H. Sipma, and T. Uribe. Verifying temporal properties of reactive systems: A STeP tutorial. *Formal Methods in System Design*, 16(3):227–270, June 2000.

[15] A. Cau and et al. Using PVS for Interval Temporal Logic Proofs - Part 1: The Syntactic and Semantic Encoding.

[16] A. Cau, B. Moszkowski, and H. Zedan. *ITL – Interval Temporal Logic*. Software Technology Research Laboratory, SERCentre, De Montfort University, The Gateway, Leicester LE1 9BH, UK, 2002. www.cms.dmu.ac.uk/~cau/itlhomepage.

[17] Antonio Cau and Pierre Collette. Parallel composition of assumption-commitment specifications: A unifying approach for shared variable and distributed message passing concurrency. *Acta Inf.*, 33(2):153–176, 1996.

[18] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, number 131 in LNCS. Springer, 1981.

[19] P. Clayton and G. Hripcsak. Decision support in healthcare. *International Journal of Bio-Medical Computing*, 39:59–66, 4 1995.

[20] E. Coiera. *Guide to health informatics*. Hodder Arnold, 2nd ed. edition, 2003.

[21] L. de Moura, H. Rueß, and M. Sorea. Lazy Theorem Proving for Bounded Model Checking over Infinite Domains. In *Proceedings of the 18th International Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Computer Science*, pages 438–455. Springer-Verlag, July 2002.

[22] G. Duftschmid and S. Miksch. Knowledge-based verification of clinical guidelines by detection of anomalies. *OEGAI Journal*, 18(2):37–39, 1999.

[23] G. Duftschmid, S. Miksch, Y. Shahar, and P. Johnson. Multi-Level Verification of Clinical Protocols, 1998.

[24] A. Dunets. Automatische Verifikation medizinischer Guidelines. Diplomarbeit, Universität Augsburg, 2005. (in German).

[25] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" revisited: on branching versus linear time temporal logic. *ACM*, 33(1):151–178, 1986.

[26] Evidence-Based Medicine Working Group. Evidence-based medicine. *Journal of the American Medical Association*, 268:2420 – 2425, 1992. http://jama.ama-assn.org/cgi/reprint/268/17/2420.

[27] J. Fox, N. Johns, C. Lyons, A. Rahmanzadeh, R. Thomson, and P. Wilson. PROforma: a general technology for clinical decision support systems. *Computer Methods and Programs in Biomedicine*, 54:59–67, 1997.

[28] J. Fox, V. Patkar, and R. Thomson. Decision support for health care: the proforma evidence base. *Inform Prim Care*, 14(1):49–54, 2006.

[29] M. B. Gaid, B. Bérard, and O. De Smet. Verification of an evaporator system with Uppaal. *Journal Européen des Systémes Automatisés*, 39(9-10):1079–1098, 2005.

[30] L. Giordano, A. Martelli, P. Terenziani, A. Bottrighi, and S. Montani. A temporal approach to the specification and verification of Interaction Protocols. In F. Corradini, F. De Paoli, E. Merelli, and A. Omicini, editors, *WOA*, pages 171–176. Pitagora Editrice Bologna, 2005.

[31] L. Giordano, P. Terenziani, A. Bottrighi, S. Montani, and L. Donzella. Model Checking for Clinical Guidelines: an Agent-based Approach. In *AMIA Annu Symp Proc.*, pages 289–293, 2006.

[32] P. Groot, A. Hommersom, P. Lucas, M. Balser, and J. Schmitt. Experiences in quality checking medical guidelines using formal methods. In *Verification and Validation of Software Systems*, pages 164–178, 2007.

[33] P. Habermehl and T. Vojnar. Regular Model Checking Using Inference of Regular Languages. In *Proceedings of 6th International Workshop on Verification of Infinite-State Systems – INFINITY 2004*, pages 61–71, 2004.

[34] A. Hoffmann. Fehleranalyse medizinischer behandlungsprotokolle durch berechnung temporallogischer gegenbeispiele. Diplomarbeit, Universität Augsburg, 2008. (in German).

[35] G. J. Holzmann. The Model Checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[36] A. Hommersom, P. Groot, P. Lucas, M. Balser, and J. Schmitt. Combining task execution and background knowledge for the verification of medical guidelines. In M. Bramer, F. Coenen, and A. Tuson, editors, *Proceedings of AI-2006, the 26th SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence*, number XXIII in Research and Development in Intelligent Systems, pages 3–16. Springer, 2006.

[37] A. Hommersom, P. Groot, P. Lucas, M. Balser, and J. Schmitt. Verification of medical guidelines using background knowledge in task networks. *IEEE Transactions on Knowledge and Data Engineering*, 19(6):832–846, June 2007.

[38] C. Jones. Specification and design of (parallel) programs. In *IFIP*, 1983.

[39] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[40] P. Lucas. Quality Checking of Medical Guidelines through Logical Abduction. In *Proc. of AI-2003, volume XX*, pages 309–321. Springer, 2003.

[41] P. J. F. Lucas. Symbolic Diagnosis and its Formalisation. *The Knowledge Engineering Review*, 12(2):109–146, 1997.

[42] Z. Manna and A. Pnuelli. Temporal verification diagrams. In M. Hagiya an J. Mitchell, editor, *International Symposium an Theoretical Aspects of Computer Software*, volume 789, pages 726–765. Springer Verlag, 1994.

[43] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1990.

[44] S. Miksch and K. Hammermüller. Asbru, a Plan-Representing Language Modelling Time-oriented, Skeletal Plans in Sport. In A. Baca, editor, *Proceedings of the 2nd International Symposium Computer Science in Sport*, page 1999, 1999. http://www.ifs.tuwien.ac.at/ silvia/pub/publications/mik_css99.pdf.

[45] S. Miksch, Y. Shahar, and P. Johnson. Asbru: A Task-Specific, Intention-Based, and Time-Oriented Language for Representing Skeletal Plans. In *UK, Open University*, pages 1–25, 1997.

[46] S. Miksch, Y. Shahar, and P. Johnson. Medizinische Leitlinien und Protokolle: das Asgaard/Asbru Projekt. *KI-Journal*, pages 34–37, 1997. Themenheft Medizin.

[47] Jayadev Misra and K. Mani Chandi. Proofs of networks of processes. *IEEE Transactions of Software Engineering*, 1981.

[48] M.Marcos, S.Miksch, and et al. Deliverable D2.2bcd: Models of selected guideline in intermediate, Asbru and KIV representations. Project Deliverable D2.2bcd, University of Augsburg, November 2005. URL: `http://www.protocure.org/deliverables.html/`.

[49] B. Moszkowski. A temporal logic for multilevel reasoning about hardware. *IEEE Computer*, 18(2):10–19, 1985.

[50] M. Musen. Domain ontologies in software engineering: Use of Protégé with the EON architecture, 1998.

[51] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[52] S. Panzarasa, S. Quaglini, G. Micieli, S. Marcheselli, M. Pessina, C. Pernice, A. Cavallini, and M. Stefanelli. Improving compliance to guidelines through workflow technology:implementation and results in a Stroke Unit. In K. Kuhn et al., editor, *MEDINFO 2007*, pages 834 – 839. IOS Press, 2007.

[53] M. Peleg and et al. Glif3: The Evolution of a Guideline Representation Format. In *Proc AMIA Symp*, pages 645–649, 2000.

[54] M. Peleg, S. Tu, J. Bury, P. Ciccarese, J. Fox, R. Greenes, R. Hall, P Johnson, N. Jones, A. Kumar, S. Miksch, S. Quaglini, A. Seyfang, E. Shortliffe, and M. Stefanelli. Comparing computer-interpretable guideline models: A case-study approach. *JAMIA*, 10(1), 2003.

[55] C. Polo-Conde, M. Marcos, A. Seyfang, J. Wittenberg, S. Miksch, and K. Rosenbrand. Assessment of mhb: an intermediate language for the representation of medical guidelines. In *Proc. of the 10th Conference of the Spanish Association for Artificial Intelligence (CAEPIA-05)*, pages I–19 – I–28, 2005.

[56] S. Quaglini, P. Ciccarese, G. Micieli, and A. Cavallini. Non-Compliance with Guidelines: Motivations and Consequences in a case study. In Tu SW K. Kaiser, S. Miksch, editor, *Proceedings of the Symposium on Computerised Guidelines and Protocols (CPG 2004)*, pages 75–87. IOS Press, 2004.

[57] W. Reif. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, LNCS 1009. Springer-Verlag, Berlin, 1995.

[58] W. Reif. Formale Methoden für sicherheitskritischeSoftware – Der KIV-Ansatz. *Informatik – Forschung und Entwicklung*, 14(3), 1999.

[59] E. J. Rutgers, J.W. Nortier, M. K. Tuut, G. van Tienhoven, H. Struikmans, M. Bontenbal, M. F. von Meyenfeldt, G. Vreugdenhil, T. Benraadt, B. Garssen, and J. L. Peterse. Dutch Institute for Healthcare Improvement Guideline: Treatment of Breast Cancer. *Ned Tijdschr Geneeskd*, 45, November 2002.

[60] J. Schmitt. Verifikation nebenläufiger Systeme. Diplomarbeit, Universität Augsburg, 2004. (in German).

[61] J. Schmitt, M. Balser, and et al. Deliverable D4.2c: Formal verification of selected guideline properties. Project Deliverable D4.2c, University of Augsburg, July 2006. URL: `http://www.protocure.org/deliverables.html/`.

[62] J. Schmitt, A. Hoffmann, M. Balser, W. Reif, and M. Marcos. Interactive verification of medical guidelines. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Formal Methods 2006, Proceedings*, volume 4085 of *LNCS*, pages 32–47. Springer, 2006.

[63] J. Schmitt, W. Reif, A. Seyfang, and S. Miksch. Temporal dimension of medical guidelines: The semantics of Asbru time annotations. In *Proceedings of Workshop on AI techniques in healthcare: evidence-based guidelines and protocols (ECAI'06)*, 2006.

[64] A. Seyfang, R. Kosara, and S. Miksch. Asbru 7.3 reference manual. Technical report, Vienna University of Technology, 2002.

[65] A. Seyfang, S. Miksch, and M. Marcos. Combining diagnosis and treatment using Asbru. *International Journal of Medical Informatics*, 68(1-3):49–57, 2002.

[66] A. Seyfang, S. Miksch, C. Polo-Conde, J. Wittenberg, M. Marcos, and K. Rosenbrand. MHB - A Many-Headed Bridge Between Informal and Formal Guideline Representations. In *AIME*, pages 146–150, 2005.

[67] R. Stegers. From natural language to formal proof goal: Structured goal formalisation applied to medical guidelines. Master's thesis, Vrije Universiteit, Amsterdam, 2006.

[68] D. R. Sutton and J. Fox. The Syntax and Semantics of the PROforma Guideline Modeling Language. *J Am Med Inform Assoc*, 10(5):433–443, Sep–Oct 2003.

[69] P. Terenziani, C. Carlini, and S. Montani. Towards a comprehensive treatment of temporal constraints in clinical guidelines. In *Proc TIME 2002*, pages 20–27, 2002.

[70] P. Terenziani, L. Giordano, A. Bottrighi, S. Montani, and L. Donzella. Spin model checking for the verification of clinical guidelines. In *Proceedings of Workshop on AI techniques in healthcare: evidence-based guidelines and protocols (ECAI'06)*, 2006.

[71] S. W. Tu and M. A. Musen. Modeling Data and Knowledge in the EON Guideline Architecture, 2001.

[72] W. Visser and H. Barringer. CTL* model checking for SPIN. In *In Software Tools for Technology Transfer, LNCS*, pages 200–0, 1999.

[73] P. Votruba, S. Miksch, A. Seyfang, and R. Kosara. Tracing the formalization steps of textual guidelines. *Stud Health Technol Inform*, 2004. http://www.ifs.tuwien.ac.at/ silvia/pub/publications/vot_cgp2004.pdf.

[74] P. Votruba, A. Seyfang, M. Paesold, and S. Miksch. Environment-driven skeletal plan execution for the medical domain. In *Proceedings of Workshop on AI techniques in healthcare: evidence-based guidelines and protocols (ECAI'06)*, 2006.

[75] D. Wang, M. Peleg, S. W. Tu, A. A. Boxwala, O. Ogunyemi, Q. Zeng, R. A. Greenes, V. L. Patel, and E. H. Shortliffe. Design and implementation of the GLIF3 guideline execution engine. *J. of Biomedical Informatics*, 37(5):305–318, 2004.

Lebenslauf

Jonathan Schmitt wurde am 17. Mai 1980 in Augsburg, Deutschland geboren. Er erreichte 1999 das Abitur und leistete danach seinen Wehrdienst ab. Sein Interesse für Technologie im Allgemeinen und Computer im Speziellen brachte ihn zu seinem Entschluss, Informatik zu studieren.

Er nahm sein Studium im Oktober 2000 an der Universität Augsburg auf. Er entdeckte früh sein Interesse an den theoretischen Aspekten der Informatik und vertiefte diese nicht nur in zahlreichen Vorlesungen, sondern auch durch hilfswissenschaftliche Tätigkeiten am Lehrstuhl für Softwaretechnik und Programmiersprachen. Neben der theoretischen Informatik spezialisierte er sich auch auf die Themengebiete Softwaretechnik und Multimedia. Im April 2004 schloss er sein Studium ab. In seiner Diplomarbeit untersuchte und verglich er verschiedene Beweistechniken für temporallogische Aussagen.

Ebenfalls im April 2004 begann er an am Lehrstuhl für Softwaretechnik und Programmiersprachen als wissenschaftlicher Mitarbeiter zu promovieren. Die Forschungsarbeiten, die zu seiner Promotion führten, fanden im Rahmen des EU-geförderten Projektes Protocure statt. Dieses Projekt untersuchte die Verbesserung von Qualität im Gesundheitswesen durch die Übertragung von Methoden der Softwaretechnik in die medizinische Domäne. Das Projekt wurde abschließend von der EU mit der Bestnote „good to excellent project“ bewertet.

Gegenwärtig betreut und leitet Jonathan Schmitt Vorlesungen und Seminare im Studiengang „Software Engineering“ des Elitenetzwerk Bayern als wissenschaftlicher Mitarbeiter.