Lehrstuhl für Prozeßtechnik
RHEINISCH-WESTFÄLISCHE
TECHNISCHE HOCHSCHULE AACHEN
Prof. Dr.-Ing. W. Marquardt

Lehr- und Forschungsgebiet
Theoretische Informatik
RHEINISCH-WESTFÄLISCHE
TECHNISCHE HOCHSCHULE AACHEN
Prof. Dr.-Ing. F. Baader

Diplomarbeit

Entwicklung einer Spezialisierungshierarchie für Modellierungsschritte im objekt-orientierten Datenmodell VeDa

Claudia Krobb

Betreuer: Dipl.-Ing. Dipl.-Kfm. Bernd Lohmann
Dipl.-Inf. Ulrike Sattler

Aachen, den 14. April 1997

Thema der Arbeit

Titel

Entwicklung einer Spezialisierungshierarchie für Modellierungsschritte im objekt-orientierten Datenmodell VEDA

Kurzbeschreibung[1]

Stationäre und dynamische Prozeßsimulationen werden in der Verfahrenstechnik vor allem zur Analyse und Optimierung verfahrenstechnischer Prozesse sowie für die Lösung von Problemen der Prozeßauslegung und -führung eingesetzt. Die im Zusammenhang mit der Prozeßsimulation durchgeführte detaillierte mathematische Modellierung ist aufgrund der Komplexität und Vielfalt verfahrenstechnischer Apparate und Phänomene mit hohem Aufwand verbunden. Nur durch eine deutliche Verringerung des Modellierungsaufwands können modellgestützte Techniken auf breiter Basis effizient eingesetzt werden. Die rechnergestützte Modellierung gewinnt deshalb zunehmend an Bedeutung.

Neben der systemtechnischen Strukturierung von mathematischen Prozeßmodellen werden neuerdings auch Methoden zur Beschreibung und Koordinierung von Modellierungsschritten, die zu diesen Modellen führen, untersucht. Als Voraussetzung für die Unterstützung des Modellierungsablaufs im Sinne eines feingranularen Workflow-Managements für die Modellentwicklung müssen diese Modellierungsschritte formal beschrieben werden, wozu das am Lehrstuhl für Prozeßtechnik entwickelte objekt-orientierte Datenmodell VEDA eingesetzt wird.

Z.Zt. ist es nicht möglich, anwendungsspezifische Klassen, die konkrete verfahrenstechnische Modellierungsschritte repräsentieren, weiter zu spezialisieren. Dies liegt an fehlenden Kriterien, anhand derer eine solche Spezialisierung vorgenommen werden könnte, was auf eine unzureichende Formalisierung dieser Klassen zurückzuführen ist. Insbesondere können einzelne Modellierungsschritte andere Objekte mittels Methoden verändern, und es ist auf Grund der Mächtigkeit dieser Methoden nicht möglich zu sagen, wann eine Methode spezieller als eine andere ist. Weiterhin stehen keine Kriterien zur Verfügung, anhand derer entschieden werden kann, ob ein bestimmter Modellierungsschritt in einer Abfolge von Modellierungsschritten gegen einen anderen Schritt ausgetauscht werden kann, ohne daß die nachfolgenden Schritte ebenfalls geändert werden müssen. Die Ausführbarkeitsbedingungen könnten nach dem Austausch nicht mehr erfüllt sein.

Im Verlauf dieser Arbeit soll nun zunächst die Syntax für die Klassendefinitionen von VEDA geeignet eingeschränkt bzw. modifiziert werden, um für diese Klassendefinitionen dann eine formale Semantik angeben zu können. Hier werden besonders die Möglichkeiten, Methoden angeben zu können, eingeschränkt werden müssen, um eine Formalisierung der Semantik zu ermöglichen. Durch diese Formalisierung wird zum einen den Definitionen von Modellierungsschrittklassen eine eindeutige Bedeutung zugewiesen, zum anderen wird es dadurch möglich, die Spezialisierungsrelation zwischen Klassen von Modellierungsschritten anzugeben. Die Angabe dieser Spezialisierungsrelation stellt das zentrale Ziel dieser Arbeit dar, während Modifikation der Syntax und

[1] Diese Kurzbeschreibung ist dem Antrag auf Genehmigung einer externen Diplomarbeit an den Diplomprüfungsausschuß entnommen

Angabe einer formalen Semantik notwendige Schritte zum Erreichen dieses Ziels sind. Schließlich sollen durch die Definition einiger exemplarischer Klassen von Modellierungsschritten die getroffenen Entscheidungen motiviert werden.

Inhaltsverzeichnis

1 Einleitung und Motivation

Die vorliegende Arbeit beschäftigt sich mit der "Entwicklung einer Spezialisierungshierarchie für Modellierungsschritte im verfahrenstechnischen Datenmodell VeDa". In diesem einleitenden Abschnitt wird diese Aufgabenstellung motiviert und die Vorgehensweise bei der Lösung dieser Aufgabe skizziert.

Dazu ist es zunächst notwendig, die Begriffe "Prozeßmodellierung" und "Ablaufmodellierung" einzugrenzen. Anschließend wird in Abschnitt 1.3 die Definition einer Spezialisierungsrelation für Ablaufschritte motiviert.

Auf das verfahrenstechnische Datenmodell VeDa wird in diesem einleitenden Abschnitt nicht eingegangen, seine Beschreibung wird in Abschnitt 4 nachgeholt.

1.1 Ziele der Prozeßmodellierung

Unter einem (verfahrenstechnischen) *Prozeß* wird im folgenden eine zweckgerichtete Verknüpfung von physikalischen, chemischen, biologischen und informationstechnischen Vorgängen verstanden, die Stoffe nach Art, Eigenschaften und Zusammensetzung gezielt verändern [Mar96a]. Als *Modellierung* wird nach van Gigch [vG91] eine zielorientierte Vereinfachung der Realität durch Abstraktion bezeichnet. Diese beiden Begriffsbestimmungen, welche keineswegs die einzigen möglichen Definitionen sind, helfen bei der Veranschaulichung dessen, was im folgenden als *Prozeßmodellierung* bezeichnet wird.

Ziel der *Prozeßmodellierung* ist es, einen Prozeß in einer Weise darzustellen, die ihn der Analyse und Simulation zugänglich macht und dadurch die Verbesserung bestehender und die Erstellung neuer chemischer Prozesse unterstützt.

Die Erstellung mathematischer Modelle für verfahrenstechnische Prozesse ist aufgrund der vielfältigen verfahrenstechnischen Apparate und physikalischen Phänomene, sowie der zunehmenden Anforderungen an den Detaillierungsgrad der Modelle im allgemeinen sehr aufwendig [LM96]. Der Vorgang der Modellbildung erfordert demzufolge ein gutes Verständnis des betrachteten Prozesses seitens des Modellierers, da die von ihm vorgenommenen Vereinfachungen sich gegebenenfalls im späteren Simulationsverhalten des Modells wiederspiegeln.

Zur Rationalisierung der Modellierung ist es erforderlich, den Modellierungsvorgang zu vereinfachen. Dazu werden Werkzeuge benötigt, die den Modellierer bei der Erstellung neuer und der Modifikation bestehender Modelle unterstützen, indem sie ihm mögliche Vorgehensweisen vorschlagen, Routineaufgaben abnehmen, bei der Auswahl bereits vorhandener Modellbausteine, deren Wiederverwendung zweckmäßig erscheint, behilflich sind und die Dokumentation des Prozeßmodells unterstützen [LM96].

1.2 Ziele der Ablaufmodellierung

Ein wesentliches Ziel der Ablaufmodellierung besteht in dem hier betrachteten Kontext darin, Wissen über sinnvolle Vorgehensweisen bei der Erstellung von Prozeßmodellen durch eine formale Beschreibung festzuhalten und einem Modellierer bei Bedarf zur Verfügung zu stellen. Dadurch wird es möglich, einem Modellierer Hilfestellung bei der Erzeugung neuer oder Veränderung bestehender Prozeßmodelle zu geben, zum Beispiel

durch Vorschläge für den nächsten auszuführenden Schritt. Die Empfehlung von Vorgehensweisen für die Modellerstellung erhöht die Qualität des erzeugten Modells, da Fehlerquellen beseitigt werden und die Dokumentation des Modells und der getroffenen Entscheidungen als fester Bestandteil in den Ablauf integriert werden kann. Es ist jedoch zu beachten, daß es sich bei der Modellierung immer um einen kreativen Prozeß handelt, weshalb ein Modellierer jederzeit die Möglichkeit haben sollte, von vorgegebenen Pfaden abzuweichen und neue Wege zu beschreiten.

Häufig ist nur ein Teil eines Modellierungsablaufs hinreichend gut verstanden, um ihn formal darstellen und eventuell sogar automatisieren zu können [JM96]. Pohl bezeichnet im Bereich des Requirements-Engineerings solche Teile des Ablaufs als *gut verstandene Prozeßfragmente* [Poh94]; wir werden im folgenden auch abkürzend den Begriff *Prozeßfragment* verwenden. Das mangelnde Wissen über den Gesamtablauf hat zur Folge, daß in der Regel nur Teile des Ablaufs formalisiert und somit dem Modellierer zugänglich gemacht werden können, während der Modellierer in allen anderen Bereichen weiterhin auf sich selbst gestellt ist.

Betrachtet man – in Analogie zu [Poh94] – den Zustand eines Prozeßmodells als einen Punkt in einem dreidimensionalen Raum, welcher durch die Achsen Spezifikation (Granularität der Betrachtung), Repräsentation (Grad der Formalisierung) und Konsens (Grad der Einigung des Modellierungsteams) aufgespannt wird, so bildet der Modellierungsablauf eine Trajektorie in diesem Raum (vergleiche Abbildung 1).

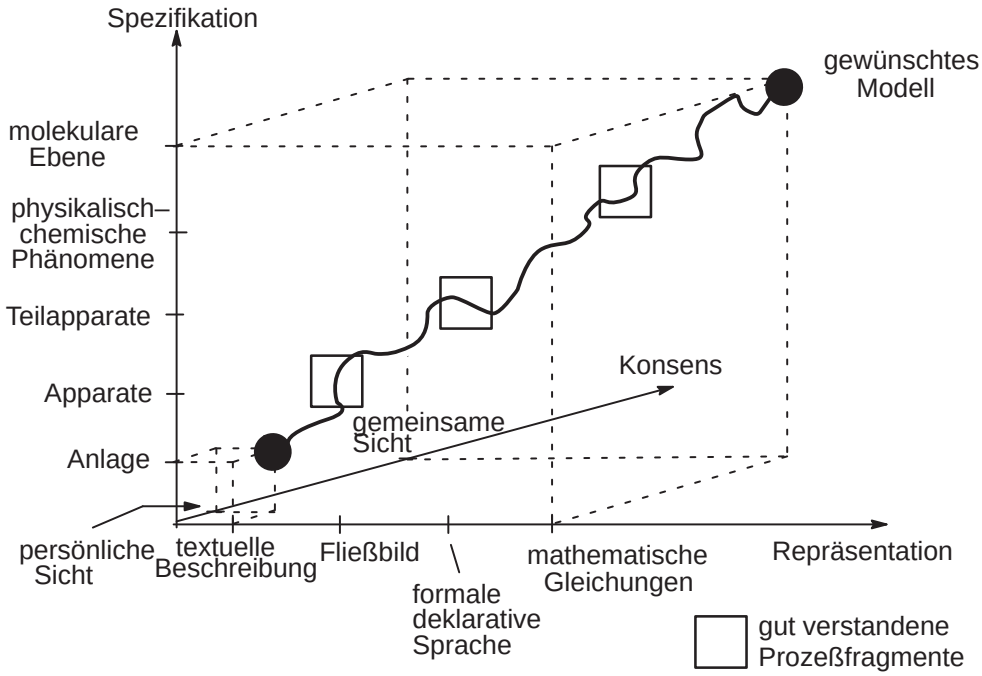

Abbildung 1: Der Modellierungsablauf für chemische Prozesse in drei Dimensionen nach Pohl [Poh94]

Auf dieser Trajektorie lassen sich dann verschiedene Prozeßfragmente identifizieren, welche zum Zweck der Formalisierung wiederum in einzelne Teilschritte zerlegt werden können.

1.3 Gründe für die Spezialisierung von Ablaufschritten

Möchte man einen Modellierungsablauf formal beschreiben, so erfordert dies sowohl Konzepte für die Angabe einzelner Modellierungsschritte als auch für die Verknüpfung dieser Schritte untereinander.

Ein Modellierungsablauf ist eine Folge von Teilschritten, welche selbst wiederum aus verschiedenen Schritten bestehen können. Die Ausführungsreihenfolge der Teilschritte kann für den Erfolg der Modellierung relevant sein, es können Wiederholungen einzelner Schrittsequenzen oder die Auswahl zwischen verschiedenen Schritten erforderlich sein. Die wohl wichtigste Anforderung an eine solche Schrittfolge ist, daß sie immer durchführbar ist, das heißt, daß garantiert wird, daß die Schritte in jeder der vorgegebenen Reihenfolgen auf jeden Fall anwendbar sind.

Das Ziel dieser Arbeit ist die Definition einer Spezialisierungsrelation für die verschiedenen Konzepte zur Beschreibung von Ablaufschritten. Diese Relation wird es ermöglichen, bereits definierte Ablaufschritte, welche in einem neuen Zusammenhang sinnvoll erscheinen, zu finden und wiederzuverwenden. Diese Wiederauffindbarkeit und die damit verbundene Wiederverwendbarkeit führen dazu, daß bei der Beschreibung von Entwicklungsabläufen nicht jedesmal wieder von vorne begonnen werden muß.

Weiterhin ist auch die Austauschbarkeit von Ablaufschritten in vorgegebenen Abfolgen von Schritten zu erwähnen. Die in dieser Arbeit zu entwickelnde Spezialisierungsrelation soll Kriterien liefern, anhand derer entschieden werden kann, ob ein Schritt innerhalb einer bereits definierten Folge von Ablaufschritten durch einen anderen Schritt ersetzt werden kann. Dabei ist insbesondere auf die Erfüllung von Vor- und Nachbedingungen einzelner Schritte zu achten, da die Einsetzung eines neuen Schritts ansonsten die Ausführbarkeit der gesamten Schrittfolge verhindern kann. Diese Kriterien dienen außerdem der Überprüfung, ob ein zusammengesetzter Ablaufschritt überhaupt noch ausführbar ist, wenn die vorgegebenen Bedingungen für die Reihenfolge, in der die einzelnen Teilschritte durchgeführt werden sollen, durch den Modellierer verletzt werden.

Die Definition von Abläufen ist insbesondere deshalb aufwendig, weil empirische Erfahrungen über die Vorgehensweise der Modellierer und Kenntnisse zur formalen Definition von Abläufen nötig sind. Weiterhin unterliegen Abläufe der Evolution, das heißt sie ändern sich und werden ständig weiterentwickelt. Viele dieser Vorgänge sind noch nicht ausreichend verstanden, um eine Formalisierung zu ermöglichen. Diese Arbeit versucht, durch die Untersuchung der Spezialisierung von Ablaufschritten einen Beitrag zur Rationalisierung der Modellbildung zu leisten.

1.4 Aufbau der Ausarbeitung

In den folgenden Abschnitten wird zunächst eine kurze Einführung in die verfahrenstechnische Prozeßmodellierung, die Ablaufmodellierung und das verfahrenstechnische Datenmodell VEDA gegeben. Dabei liegt der Focus auf der Darstellung der Grundlagen, welche für das Verständnis der vorliegenden Arbeit relevant sind. Für Details wird jeweils auf die angegebene Fachliteratur verwiesen.

Abschnitt 5 befaßt sich mit der Darstellung von Abläufen im Datenmodell VEDA, wobei auf die dazu benötigten Klassen und ihre Wechselwirkungen untereinander eingegangen wird.

Abschnitt 6 führt einen Formalismus zur Angabe von Bedingungen, welche die mögli-

chen Ausführungsreihenfolgen der Teilschritte eines Ablaufs einschränken, ein. In Verbindung mit den in Abschnitt 7 vorgestellten Vor- und Nachbedingungen einzelner Schritte dienen diese Bedingungen der Überprüfung der grundsätzlichen Durchführbarkeit eines Ablaufs. Die Vor- und Nachbedingungen bilden die Grundlage für die in Abschnitt 8 definierte Spezialisierungsrelation für Ablaufschritte in VeDa.

Abschnitt 9 faßt die erzielten Ergebnisse noch einmal kritisch zusammen und liefert Hinweise auf zukünftige Aufgaben.

Im Anhang werden die in dieser Arbeit eingeführten Formalismen für die Angabe von Abfolgebedingungen und Vor- und Nachbedingungen sowie die Arbeitsweise des in Abschnitt 7.6 entwickelten Entscheidungsalgorithmus für die Erfüllbarkeit von Vor- und Nachbedingungen an einem etwas umfangreicheren Beispiel veranschaulicht.

2 Überblick über die verfahrenstechnische Prozeßmodellierung

Dieser Abschnitt gibt einen groben Überblick über die Grundlagen der verfahrenstechnischen Prozeßmodellierung, ohne in irgendeiner Weise Anspruch auf Vollständigkeit zu erheben. Er soll lediglich einen Eindruck vom Aufbau eines Prozeßmodells und seiner Entstehung vermitteln.

Ziel der Modellierung verfahrenstechnischer Prozesse ist es, anhand eines Modells Erkenntnisse über Prozesse zu gewinnen, welche sonst nur mit einem erheblichen materiellen Aufwand, zum Beispiel durch den Bau und Betrieb einer entsprechenden Anlage, zu erlangen sind.

Die Erstellung eines Modells zu einem gegebenen Prozeß ist kein eindeutiger Vorgang, sondern hängt stark von der Erfahrung und den Vorlieben des jeweiligen Modellierers ab [Mar96b]. Trotzdem können allgemeine Richtlinien für die Erstellung von Prozeßmodellen angegeben werden.

Um die Verständlichkeit eines Modells zu erhöhen und damit die Wartbarkeit und Wiederverwendbarkeit zu verbessern, ist es sinnvoll, das Modell zu strukturieren und zu dokumentieren.

Im folgenden werden wir das von Marquardt in [Mar96b] vorgeschlagene Strukturierungs-Schema für Prozeßmodelle vorstellen. Die Spezialisierung des Prozeßmodells erfolgt dabei in zwei Richtungen, der strukturellen Beschreibung und der Verhaltensbeschreibung.

Strukturelle Prozeßbeschreibung

Eine chemische Anlage kann aufgrund ihrer materiellen Struktur in einzelne Teilanlagen unterteilt werden, welche selbst wiederum aus verschiedenen Einheiten bestehen. Bei der strukturellen Beschreibung einer Anlage entlang ihrer strukturellen Koordinate können zwei verschiedene Kategorien von Modellierungsobjekten unterschieden werden, welche jeweils zur Darstellung von abgrenzbaren materiellen Teilen der Anlage benötigt werden.

Komponenten speichern extensive Zustandsgrößen wie Masse, Impuls oder Energie, *Verknüpfungen* dagegen verbinden verschiedene Komponenten miteinander und legen die Ströme der Zustandsgrößen zwischen diesen Komponenten fest.

Beispiele für Komponenten sind ein Reaktor oder ein Kühlmantel, während Verknüpfungen zum Beispiel Rohrleitungen, Signalleitungen oder Grenzen zwischen Phasen verschiedenen Aggregationszustands sein können.

Komponenten und Verknüpfungen können jeweils dahingehend eingeteilt werden, ob sie weiter in verschiedene Bestandteile unterteilt werden oder nicht. Im ersten Fall spricht man von *zusammengesetzten* Komponenten beziehungsweise Verknüpfungen, im zweiten von *elementaren*. Bis zu welchem Detaillierungsgrad bei der Modellierung eines bestimmten Prozesses die Bestandteile des Modells weiter verfeinert werden, hängt jeweils vom Verwendungszweck des Modells und dem Kenntnisstand des Modellierers in bezug auf den Prozeß ab.

Verhaltensbeschreibung

Das Verhalten elementarer Modellbausteine wird durch zwei unterschiedliche Kon-

zepte, die *Prozeßgrößen* und die *mathematischen Gleichungen*, festgelegt. Die Werte der Prozeßgrößen beschreiben das Verhalten des Bausteins, während die Gleichungen diese Werte einschränken und die Prozeßgrößen zueinander in Beziehung setzen. Die Gleichungen repräsentieren theoretisch begründete (halb-)empirische physikalisch-chemische Gesetzmäßigkeiten oder experimentell ermittelte Zusammenhänge zur Festlegung des Ein-/Ausgabeverhaltens [Mar96a].

Spezialisierung

Sowohl die strukturellen Bausteine als auch die Prozeßgrößen und Modellgleichungen können in Spezialisierungshierarchien angeordnet werden, die auf den Gemeinsamkeiten und Unterschieden der einzelnen Konzepte basieren. Abbildung 2 zeigt in Anlehnung an [Mar96a] einen Ausschnitt aus einer möglichen Spezialisierungshierarchie für strukturelle Modellbausteine.

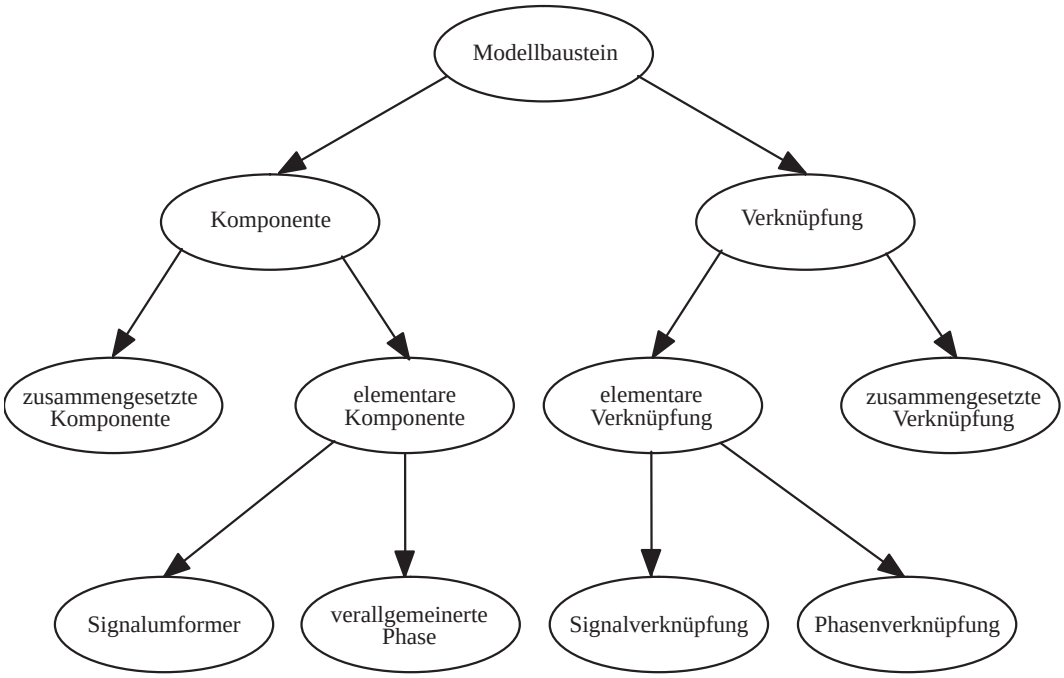

Abbildung 2: Ausschnitt aus einer Spezialisierungshierarchie für strukturelle Modellbausteine

Taxonomien wie die in Abbildung 2 dargestellte basieren auf der Einteilung von Modellbausteinen, Prozeßgrößen und Modellgleichung in verschiedene Klassen, in welchen Bausteine mit gemeinsamen Merkmalen zusammengefaßt werden.

3 Grundlagen der Überlegungen zur Ablaufmodellierung

Die Ablaufmodellierung ist ein wichtiges Hilfsmittel zur Unterstützung der Erstellung von Modellen, in dem hier betrachteten Aufgabenbereich der Erstellung mathematischer Modelle für verfahrenstechnische Prozesse.

Durch die Führung des Modellierers während der Modellerstellung wird die Qualität der Modelle im Hinblick auf Korrektheit, Konsistenz, Robustheit und einen angemessenen Detaillierungsgrad verbessert [LM96]. Weiterhin wird die zur Erstellung eines Prozeßmodells benötigte Zeit dadurch verkürzt, daß dem Modellierer Routineaufgaben abgenommen werden und die Wiederverwendbarkeit vorhandener Modelle durch die Unterstützung der Suche nach bereits existierenden Modellen und eine verbesserte Dokumentation vorhandener Bausteine erhöht wird [LM96].

3.1 Der Modellierungsablauf als kreativer Prozeß

Wie bereits in Abschnitt 1.2 dargestellt kann die Erstellung eines Prozeßmodells als Trajektorie in einem dreidimensionalen Raum angesehen werden, der durch die Achsen Spezifikation, Repräsentation und Übereinstimmung aufgespannt wird (vergleiche Abbildung 1). Von diesem Vorgang sind immer nur Teile, die *well-understood process chunks* oder *Prozeßfragmente*, gut genug verstanden, um sie formalisieren zu können.

Im folgenden werden die Person oder die Personen, welche den Modellierungsablauf formalisieren (modellieren), als *Metamodellierer* bezeichnet, während die Person oder Personen, welche den so modellierten Ablauf durchführen, um ein verfahrenstechnisches Prozeßmodell zu erstellen, *Modellierer* genannt werden.

Die Formalisierung eines Ablaufs erfolgt zweckmäßigerweise durch die Zerlegung des Vorgangs in einzelne Schritte. Diese Schritte zeichnen sich jeweils durch das Ziel, das durch ihre Durchführung erreicht werden soll, die Bedingungen, welche vor und nach der Durchführung erfüllt sein müssen beziehungsweise sollen und Angaben über die zur Durchführung des Schritte auszuführenden Aktionen aus. Diese Aktionen können sowohl vollständig automatisierte Vorgänge sein als auch Vorgänge beschreiben, welche vom Modellierer weitgehend selbständig und ohne Unterstützung durch ein Modellierungswerkzeug durchgeführt werden müssen, beispielsweise das Treffen von Entscheidungen oder die Modellierung neuer Komponenten, für die noch kein Modellierungsablauf definiert wurde.

Modellierungsschritte können zusammengesetzt sein, das heißt ein Schritt kann wiederum aus mehreren Teilschritten bestehen. In diesem Fall sind Angaben über die möglichen Abfolgen der Teilschritte erforderlich. Explizit angegebene Abfolgebedingungen sollten den Modellierer unterstützen, aber nicht einschränken. Er soll jederzeit die Möglichkeit haben, von der durch den Metamodellierer vorgeschlagenen Reihenfolge der Teilschritte abzuweichen und neue Vorgehensweisen zu entwickeln, wenn ihm dies sinnvoll erscheint.

Der Vorgang der Modellentwicklung ist im allgemeinen ein kreativer Prozeß, in welchen der Modellierer eigene Ideen einbringt und so das Ergebnis beeinflußt. Aufgrund dieser Eigenschaft ist es weder möglich noch sinnvoll, die vollständige Automatisierung eines Modellierungsvorgangs zu erreichen. Die Unterstützung des Modellierers bei gleichzeitiger Erhaltung weitgehender Freiheiten und die Automatisierung von Routineaufgaben sind allerdings wichtige Ziele der Ablaufmodellierung und leisten einen wichtigen Bei-

trag zur Rationalisierung der Prozeßmodellierung.

3.2 Die verschiedenen Arten von Ablaufschritten

Wie bereits erwähnt setzt sich ein Modellierungsablauf in der Regel aus verschiedenen Teilschritten zusammen. Diese können grob in drei verschiedene Kategorien aufgeteilt werden, nämlich elementare und zusammengesetzte Ablaufschritte sowie Auswahlschritte. Jede dieser Kategorien zeichnet sich durch bestimmte Merkmale aus, welche in diesem Abschnitt vorgestellt werden.

Elementare Ablaufschritte

Ein *elementarer Ablaufschritt* zeichnet sich dadurch aus, daß er sich nicht weiter in Teilschritte unterteilen läßt. Seine Durchführung kann in der Ausführung einer bestimmten Methode oder auch einer nicht weiter formalisierten und durch den Modellierer selbständig auszuführenden Tätigkeit bestehen.

Zusammengesetzte Ablaufschritte

Im Gegensatz zu den elementaren Ablaufschritten stehen die *zusammengesetzten Ablaufschritte*, welche sich dadurch auszeichnen, daß sie sich in einzelne Teilschritte (mindestens zwei) aufteilen lassen. Bezüglich der Reihenfolge, in welcher die einzelnen Teilschritte ausgeführt werden, können dabei bestimmte Einschränkungen bestehen. Diese ergeben sich zum einen aus den *empfohlenen Abfolgebedingungen*, welche dem zusammengesetzten Ablaufschritt selbst anhaften, zum anderen aber auch automatisch aus Bedingungen, welche den einzelnen Teilschritten zugeordnet sind. Letztere ergeben sich daraus, daß für einen Schritt bestimmte Vorbedingungen gelten, die erfüllt sein müssen, bevor die Ausführung des Schritts möglich ist.

Auswahlschritte

Auswahlschritte dienen der Selektion zwischen verschiedenen Ablaufschritten, welche in einer gegebenen Situation als nächstes durchgeführt werden könnten. Die wesentliche Aufgabe eines Auswahlschritts ist somit die Unterstützung der Entscheidung zwischen den verschiedenen Alternativen, wobei die Entscheidung auch die Frage betreffen kann, ob ein Schritt oder eine Schrittfolge wiederholt werden soll oder nicht. In diesem Fall kann eine Auswahlmöglichkeit sein, keinen weiteren Schritt durchzuführen.

In einigen Fällen kann die Auswahl des nächsten durchzuführenden Schritts automatisiert werden. Dies ist der Fall, wenn in einer konkreten Situation lediglich die Vorbedingungen eines einzigen möglichen Folgeschritts erfüllt sind.

3.3 Überblick über verschiedene Ansätze zur Repräsentation von zeitlichen Zusammenhängen und Abläufen

Dieser Abschnitt gibt einen kurzen Überblick über verschiedene Formalismen zur Repräsentation von zeitlichen Zusammenhängen zwischen verschiedenen Situationen. Dabei wird zunächst auf das Gebiet des Planning eingegangen und dieses der oben vorgestellten Ablaufmodellierung gegenübergestellt.

3.3.1 "Planning"

Planning ist ein Teilgebiet der Künstlichen Intelligenz, dessen Ziel es ist, Systeme zu erstellen, welche zu einem gegebenen Problem einen Plan erzeugen, dessen Durchführung zu einer Lösung des Problems führt [THD90]. Zum Verständnis dieser Definition ist eine Erläuterung der in ihr verwendeten Begriffe notwendig. Diese erfolgt in Anlehnung an [THD90].

Ein *Problem* besteht aus zwei Komponenten: der Beschreibung des Zustands der betrachteten Welt *vor* der Durchführung des zu erzeugenden Plans (*initial state*) und der Beschreibung des gewünschten Zustands der Welt *nach* der Durchführung (*goal state*).

Ein *Operator* beschreibt eine einzelne Aktion dadurch, daß er angibt, welche Bedingungen erfüllt sein müssen, damit der Schritt angewendet werden kann, und in welcher Weise die Durchführung der Aktion den Zustand der Welt verändert.

Ein *Plan* ist eine "organisierte" Menge von Operatoren, deren mögliche Ausführungsreihenfolgen durch bestimmte Konstruktoren eingeschränkt sind. Mögliche Konstruktoren sind Sequenz, Auswahl, Iteration und Rekursion, aber auch Konstruktoren für Nichtdeterminismus und Parallelität können bei der Planerzeugung verwendet werden [Geo90].

Als eine *Lösung* eines Problems bezeichnen wir einen Plan, dessen erster Operator im *initial state* eines Problems anwendbar ist und der die Welt durch seine Ausführung in den gewünschten *goal state* überführt.

Eine gewisse Ähnlichkeit zu Plänen weisen *Drehbücher* (Scripts) auf. Drehbücher dienen der Darstellung von Wissen über Ereignisse in Form von klischeehaften Abläufen, das heißt sie stellen dar, welche Ereignisse in bestimmten Situationen "üblicherweise" auftreten.

Charakterisiert wird ein Drehbuch durch seine Eingangsbedingungen und Ergebnisse (Nachbedingungen), die in ihm vorkommenden Objekte und Agenten, die Angabe eines allgemeineren Drehbuches, dem es zugeordnet werden kann und die Angabe der tatsächlich auftretenden Ereignissequenz.

Im Gegensatz zu einem Plan dient ein Drehbuch nicht der Erreichung eines bestimmten Zielzustands, sondern der Darstellung von allgemeinem Wissen über bestimmte Abläufe. Nach einem Drehbuch wird also nicht in einem Zustandsraum gesucht, sondern es entsteht durch Abstraktion aus einer Menge von Beobachtungen und ist somit eine Verallgemeinerung verschiedener Spezialfälle.

3.3.2 Bezug zwischen Ablaufmodellierung und Planning

Bei der Planung wird ein Problem durch die Angabe der Anfangssituation und des gewünschten Zielzustands sowie der zur Verfügung stehenden Operatoren definiert. Diese Angaben können in verschiedenen Repräsentationsformen erfolgen.

Der Planungsprozeß besteht darin, eine Abfolge von Operatoren zu suchen, welche die Welt vom Startzustand in den Zielzustand überführt. Für diese Suche existieren verschiedene Strategien, welche von der Art der Problemstellung und der gewählten Repräsentationsform abhängen. Große Probleme werden – soweit möglich – in Teilprobleme unterteilt, die getrennt bearbeitet werden können. Die bei dieser Bearbeitung erhaltenen Teillösungen werden anschließend zu einer Gesamtlösung des ursprünglichen Problems kombiniert. Es ist allerdings zu beachten, daß viele Probleme sich nicht in

vollständig voneinander unabhängige Teile zerlegen lassen. Häufig ist es aber möglich, *fast* unabhängige Teilprobleme zu identifizieren, das heißt Probleme, die untereinander nur in geringer Wechselwirkung stehen [Ric88].

In einem *vorhersagbaren* Universum, das heißt in einer Welt, die nur durch den planausführenden Agenten und nur entsprechend der vorgegebenen Operatoren und ihrer angegebenen Auswirkungen verändert werden kann, führt die Ausführung eines Plans, der zur Lösung eines bestimmten Problems erstellt wurde, immer zu Ziel.

In einem *nicht-vorhersagbaren* Universum kann dagegen nicht garantiert werden, daß die Durchführung eines Plans immer das gewünschte Ergebnis liefert, es sei denn, der Plan deckt alle möglichen Eventualitäten ab. Dies wird in den seltensten Fällen möglich und sinnvoll sein. In einer nicht-vorhersagbaren Welt werden daher in der Regel Pläne generiert werden, die *wahrscheinlich* zum Erfolg führen [Ric88]. In diesem Fall müssen während der Planausführung beim Erkennen einer Abweichung vom vorgesehenen Ergebnis geeignete Gegenmaßnahmen ergriffen werden. Diese können zum Beispiel darin bestehen, den Planungsprozeß mit dem dann aktuellen Zustand als Startzustand erneut anzustoßen oder einen Weg zu suchen, den gerade abgearbeiteten Teilplan so zu ergänzen, daß er doch noch das gewünschte Ergebnis liefert, und anschließend mit dem restlichen Plan fortzufahren.

Bei der Ablaufmodellierung werden einzelne Ablaufschritte, welche den Operatoren des Plannings vergleichbar sind, vom Metamodellierer zu Folgen zusammengesetzt, welche der Erreichung bestimmter Ziele dienen. Die Hauptunterschiede zum Planning liegen darin, daß diese Folge nicht automatisch anhand eines bestimmten Verfahrens generiert wird, sondern vom Metamodellierer auf der Basis seiner Erfahrungen und Intuition erstellt wird, und daß das verfolgte Ziel bisher nicht formal definiert werden kann, sondern bestenfalls in Form einer allgemeinen textuellen Beschreibung vorliegt.

Betrachtet man zum Beispiel die Modellierung eines Reaktors als Ablauf, so ist es nicht möglich, das Ziel in Form einer vollständigen Beschreibung des Zustands der Welt nach der Modellierung anzugeben, da zum Zeitpunkt der Definition des Ablaufs nicht klar ist, welche Modellbausteine zur Modellierung des Reaktors benötigt werden und wie diese untereinander zu verbinden sind. Ließe sich eine solche Beschreibung angeben, wäre der Ablauf als solcher überflüssig, da das durch ihn zu erzeugende Modell in diesem Fall schon vorläge.

Trotzdem ist es möglich, den grundsätzlichen Ablauf des Modellierungsvorgangs festzuhalten, denn dem Metamodellierer ist bekannt, daß verschiedene Komponenten und Verknüpfungen definiert, charakterisiert und miteinander verbunden werden müssen, um ein Reaktormodell zu erstellen. Dieses Wissen wird er in die Ablauffolge einbringen.

Es wird dementsprechend kein starrer Ablauf zur Lösung eines ganz bestimmten Problems erzeugt, sondern eine Schrittfolge, die zur Lösung vieler gleichartiger Probleme herangezogen werden kann. Dies hat zur Folge, daß der Modellierer während der Durchführung eines solchen Ablaufs ein mehr oder weniger großes Maß an Freiheit besitzt und in Abhängigkeit von der aktuellen Situation Entscheidungen treffen muß.

Ein Ablaufmodell verbindet somit die Allgemeinheit eines Drehbuches mit der Zielorientierung eines Plans.

3.3.3 Überblick über verschiedene Formalismen zur Darstellung von Wissen über Abläufe und Zustandsänderungen

Zur Beschreibung der Welt sowie der Aktionen, welche im Rahmen der Ausführung eines Plans durchgeführt werden müssen, stehen verschiedene Formalismen zur Verfügung, von denen hier einige exemplarisch skizziert werden.

STRIPS

In der *STRIPS*-Repräsentation wird der aktuelle Zustand der Welt durch eine Menge atomarer Propositionen beschrieben [All90]. Aktionen werden durch Operatoren repräsentiert, welche aus einer Vorbedingung, einer *add list* und einer *delete list* bestehen [Geo90]. Die Vorbedingungen sind prädikatenlogische Formeln, durch die festgelegt ist, welche Bedingungen in der Welt erfüllt sein müssen, damit die Aktion durchgeführt werden kann. Die *add list* gibt an, welche Bedingungen nach der Ausführung der Aktion zusätzlich gelten, während die *delete list* spezifiziert, welche Bedingungen nach der Ausführung der Aktion nicht mehr gelten und dementsprechend aus der Beschreibung des aktuellen Zustands der Welt entfernt werden müssen. Aktionen sind also Operatoren, welche den Zustand der Welt durch syntaktische Transformationen verändern [All90]. Daraus ergibt sich, daß nicht mehrere Aktionen gleichzeitig ausgeführt werden können, während der Durchführung einer Aktion keine weiteren Ereignisse auftreten können und die Welt sich ausschließlich aufgrund der geplanten Aktionen ändert [All90].

Die Annahme, daß

1. alle Bedingungen, welche in der *add list* eines Operators vorkommen, nach der Durchführung der zugehörigen Aktion erfüllt sind und
2. alle Bedingungen, welche in der Welt in ihrem Zustand vor der Durchführung der Aktion erfüllt sind und nicht in der *delete list* des Operators vorkommen, auch nach der Ausführung der Aktion noch gelten,

wird als *STRIPS assumption* bezeichnet [Geo90].

Ein auf STRIPS aufbauender Formalismus wird von Bäckström in [Bä92] vorgestellt. Der SAS^+-Formalismus dient der Darstellung von Plänen.

Der Zustand eines Systems wird in diesem Formalismus durch die Werte einer festen Menge von Zustandsvariablen anstelle von logischen Ausdrücken beschrieben. Aktionen besitzen neben Vor- und Nachbedingungen zusätzlich Invarianten, welche jeweils angeben, die Werte welcher der Zustandsvariablen durch die Aktion verändert werden beziehungsweise konstant bleiben.

Situation Calculus

Eine Situation im *Situation Calculus* ist ein vollständiger "Schnappschuß" der Welt zu einem bestimmten Zeitpunkt [All90]. Da es nicht möglich ist, den Zustand der Welt durch eine Auflistung der geltenden Bedingungen vollständig zu beschreiben, werden im Situation Calculus Sprachelemente zur Verfügung gestellt, welche die Angabe von Wissen über einen Teil einer gegebenen Situation ermöglichen.

Ein *fluent* ist eine auf den Situationen (also den möglichen Zuständen der Welt) definierte Funktion, welche im wesentlichen einer bestimmten Eigenschaft der Weltzustände zugeordnet ist [Geo90]. Durch solche *fluents* können zum Beispiel alle Bestandteile der Welt, welche eine bestimmte Farbe haben, bestimmt werden, oder es kann festgestellt

werden, ob eine bestimmte Bedingung in einer gegebenen Situation erfüllt ist.

Ein wichtiger *fluent* im Situation Calculus ist *result(p,a,s)*, welcher als Ergebnis diejenige Situation liefert, die dadurch entsteht, daß Agent p in Situation s die Aktion a ausführt [All90].

Neben *fluents* können im Situation Calculus die üblichen logischen Quantoren und Konnektoren verwendet werden [Geo90], so daß wir zum Beispiel die Aktion *Toggle*, welche das Umschalten eines Lichtschalters zwischen den Zuständen *ein* und *aus* repräsentiert, folgendermaßen beschreiben können:

$$\forall(p,s)\ NearSwitch(p,s)\ \wedge\ LightOn(s)\ \Rightarrow\ LightOff(result(p,Toggle,s))$$

$$\forall(p,s)\ NearSwitch(p,s)\ \wedge\ LightOff(s)\ \Rightarrow\ LightOn(result(p,Toggle,s))$$

Ein Problem des Situation Calculus liegt darin, daß eine große Anzahl von Axiomen benötigt wird, um zu beschreiben, welche Eigenschaften der Welt *nicht* durch eine bestimmte Aktion verändert werden. In einer Bausteinwelt müsste beispielsweise für jede Aktion, welche einen Baustein bewegt, explizit formuliert werden, daß die jeweiligen Farben der Bausteine erhalten bleiben, damit auch nach der Bewegung von Bausteinen noch Rückschlüsse über die Farbe der Bausteine möglich sind. Diese Axiome werden als *Frame Axiome* bezeichnet, der Zwang, sie alle explizit angeben zu müssen, als *Frame Problem* [Geo90].

Temporale Modelle

Im Situation Calculus ist es nicht möglich, Schlüsse über die Auswirkung von gleichzeitig oder überlappend auftretenden Aktionen oder externen Ereignissen zu ziehen. Dies begründet sich in der *result*-Funktion, welche zu einer Aktion und einer Situation einen eindeutigen Folgezustand angibt, der nicht durch gleichzeitig auftretende andere Aktionen verändert werden kann [All90].

Temporale Modelle basieren auf der Idee, Zusicherungen nicht mit bestimmten Weltzuständen zu verbinden, sondern mit den Zeitintervallen, in denen sie gelten [All90]. Dadurch wird es möglich, gleichzeitig auftretende oder überlappende Aktionen sowie externe Ereignisse (Ereignisse, die nicht aus geplanten Aktionen resultieren) zu repräsentieren.

Artale und Franconi [AF97, AF95] zeigen, wie eine KL-ONE basierte Sprache ($\mathcal{ALCF}$) um temporale Aspekte erweitert werden kann. In Anlehnung an Allen [All83] werden Aktionen durch diejenigen Bedingungen, welche während ihrer Durchführung wahr sind, beschrieben. Dadurch wird der konkrete Bezug zu Zuständen, wie er zum Beispiel in STRIPS besteht, aufgegeben. Insbesondere entfällt damit auch die explizite Angabe von Vor- und Nachbedingungen für Aktionen und Pläne.

Im Gegensatz zum Situation Calculus liegt das Problem bei den temporalen Modellen nicht mehr darin, zu bestimmen, was in einer bestimmten gegebenen Situation gilt, sondern darin, von einem gültigen Fakt zu schließen, wie lange er in Zukunft noch gültig sein wird. Dieses Problem scheint nicht leichter zu lösen zu sein als das ursprüngliche *Frame Problem* [All90].

Erweiterung einer KL-ONE basierten Sprache

Devanbu [Dev91b] und Devanbu und Litman [DL91] beschreiben ein von ihnen entwickeltes Software-Informations-System, welches zunächst das Auffinden einzelner C-

Funktionen aus einer Wissensbasis ermöglichte und später um die Konzepte Aktion, Plan und Szenario erweitert wurde [DL91]. Ausgangspunkt war dabei eine KL-ONE[BS85] basierte Sprache zur terminologischen Wissensrepräsentation (CLASSIC), welche um Möglichkeiten zur Repräsentation von Plänen ergänzt wurde. Pläne sind dabei durch eine reguläre Sprache festgelegte Abfolgen von Aktionen, ergänzt um Informationen über Start- und Zielzustände. Szenarien sind Instanzen von Plänen. In dem so entstandenen Wissensrepräsentationssystem CLASP sind verschiedene Subsumptionsprobleme für Pläne entscheidbar [DL91], es fehlen jedoch einige wünschenswerte Ausdrucksmöglichkeiten, insbesondere die Angabe von Vor- und Nachbedingungen für Pläne und Zusicherungen in Form von Invarianten.

Bei der Einordnung einer Aktion in eine Hierarchie wird in diesem System eine Aktion als allgemeiner als eine andere angesehen, wenn sowohl ihre Vor- als auch ihre Nachbedingungen allgemeiner sind [Dev91a]. Durch diese Annahme wird bewußt auf die Austauschbarkeit einer allgemeineren Aktion durch eine speziellere in einem gegebenen Plan verzichtet.

Modallogiken

Große und Khalil [GK95] betrachten die State Event Logic, die auf der Modallogik K basiert und in der Zustände und Ereignisse als gleichwertig angesehen werden. Das Universum besteht somit aus einer Menge von Paaren aus Zustand und Ereignis. Ein wesentliches Ziel der State Event Logic besteht darin, Aussagen über die Auswirkungen des parallelen Auftretens verschiedener Ereignisse zu treffen. Entsprechende Spracherweiterungen werden sowohl für die Propositionale Logik als auch für die Logik erster Stufe definiert.

Baader und Laux [BL94] zeigen, wie Modaloperatoren in terminologische Wissensrepräsentationssprachen integriert werden können. Als Beispiel dient die Erweiterung $\mathcal{ALC}_M$ der Sprache $\mathcal{ALC}$. Während die Objektdimension einen besonderen Status erhält, kann jede der anderen Dimensionen verschiedene Modalitäten besitzen, welche jeweils mit den Operatoren $\Box$ und $\diamond$ verwendet werden können. Eine Interpretation in $\mathcal{ALC}_M$ besteht aus einer Menge von Welten, einer binären Relation (*accesssibility relation*) zwischen diesen Welten für jeden Modaloperator und einer Beschreibung der einzelnen Welten in $\mathcal{ALC}$. Es wird eine *increasing domain restriction* vorausgesetzt, was bedeutet, daß in jeder Welt *w'*, welche von einer Welt *w* über eine *accessibility relation* erreichbar ist, alle Elemente, die in *w* existieren, auch in *w'* existieren.

4 Überblick über das verfahrenstechnische Datenmodell VeDa

Die folgenden Ausführungen basieren im wesentlichen auf [MBG+97, Kapitel 2], dem auch die in diesem Kapitel vorkommenden Abbildungen entnommen sind. Da die Sprachdefinition noch nicht abgeschlossen ist und zum Zeitpunkt der Erstellung dieser Arbeit nicht alle vorgesehenen Eigenschaften der Sprache in [MBG+97] dokumentiert sind, basieren einige der in dieser Arbeit getroffenen Aussagen über VeDa auf persönlichen Gesprächen [Bau97].

VeDa ist ein objekt-orientiertes Datenmodell zur Repräsentation von Wissen über ein Anwendungsgebiet, nämlich die Modellierung verfahrenstechnischer Prozesse. Dabei wird zwischen terminologischem und assertorischem Wissen unterschieden.

Terminologisches Wissen umfaßt die zur Beschreibung abstrakter Konzepte notwendige Information, das heißt es betrifft die Repräsentation der Terminologie in einem Anwendungsgebiet und abstrahiert von einzelnen Ausprägungen bestimmter Objekte. In VeDa wird dieses Wissen in den Metaklassen und Klassen abgelegt.

Assertorisches Wissen betrifft Wissen über eine bestimmte Situation in der durch das terminologische Wissen beschriebenen Welt, das heißt über konkrete Objekte. Dieses Wissen wird in VeDa durch Instanzen dargestellt.

Die Metaklassen, Klassen und Instanzen bilden eine dreistufige Hierarchie der Wissensrepräsentation, deren oberste Ebene die Metaklassen bilden, gefolgt von den Klassen und Instanzen. Auf diese drei Stufen wird in den folgenden Abschnitten weiter eingegangen. Einen groben Überblick über die Hierarchie bietet Abbildung 3.

4.1 Metaklassen

Metaklassen sind abstrakte Konstrukte, zu denen der Anwender keinen direkten Kontakt hat. Sie werden in VeDa nicht deklarativ angegeben und können vom Anwender nicht erweitert oder verändert werden, sondern fließen in ihrer Semantik nur implizit in die Sprachdefinition ein. Durch die Metaklassen werden die grundlegenden Eigenschaften von Klassen festgelegt, welche anhand ihrer Meta- und Superklassen in die Taxonomie der Datentypen eingeordnet werden (vergleiche Abbildung 3).

Durch die Metaklasse `frame-datatype` wird der allgemeine Aufbau einer Klasse festgelegt, ihre Subklassen dienen der Festlegung von Besonderheiten für bestimmte Klassen, zum Beispiel erlaubt die Metaklasse `has-schedule` die Angabe von Abfolgebedingungen in zusammengesetzten Ablaufschritten (vergleiche Abschnitte 5.1 und 6).

In VeDa stehen zur Zeit die folgenden Metaklassen zur Verfügung:

- `concept-frame, elementary-concept-frame, composite-concept-frame`
  Klassen zur Beschreibung elementarer und zusammengesetzter Modelle. Elementare Modelle dürfen keine "Teil-von"-Beziehungen enthalten, während zusammengesetzte Modelle solche Beziehungen enthalten können und sollen. Klassen, die lediglich Ordnungsfunktion haben und deshalb keine eigenen Slots besitzen lassen sich nicht immer in die Kategorien elementar und zusammengesetzt einordnen. Diesen Klassen wird die Metaklasse `concept-frame` zugeordnet, was allerdings zur Folge hat, daß diese Klassen immer auch `abstract` sein müssen, das heißt

Abbildung 3: Taxonomie der Datentypen (unvollständig)

nicht direkt instantiiert werden dürfen.

- `has-schedule, has-condition`
  Zur Modellierung von Ablaufschritten werden zusätzliche Angaben benötigt, welche in den allgemeinen Modellierungsklassen nicht erforderlich sind. Dies sind Abfolgebedingungen in zusammengesetzten Ablaufschritten und Vor- und Nachbedingungen in verschiedenen Klassen zur Darstellung von Ablaufschritten. Die dazu erforderlichen Slotgruppen werden durch die Metaklassen `has-schedule` beziehungsweise `has-condition` zur Verfügung gestellt.

- `abstract`
  Wie oben bereits erwähnt verhindert die Zugehörigkeit einer Klasse zur Metaklasse `abstract` die direkte Instantiierung der Klasse. Instanzen einer Klasse mit Metaklasse `abstract` sind also immer Instanzen einer ihrer nicht-abstrakten Subklassen.

- `disjunct`
  Die Zuordnung einer Klasse zur Metaklasse `disjunct` hat zur Folge, daß alle Subklassen der betroffenen Klassen in Bezug auf die jeweilige Menge ihrer Instanzen disjunkt sind.

4.2 Klassen

Klassen dienen der Repräsentation von terminologischem Wissen, welches durch die Angabe von Attributen, Restriktionen und Methoden, die alle zu dieser Klasse gehörigen Objekte auszeichnen, dargestellt wird. Jede Klasse wird durch die Angabe eines *Frames* definiert, welcher den in Tabelle 1 angegebenen allgemeinen Aufbau besitzt.

Die ersten fünf Zeilen eines Frames bezeichnen wir als seinen Kopf. Sie enthalten allgemeine Angaben zur definierten Klasse. In der mit `class` bezeichneten Zeile wird der Klasse ein eindeutiger Name zugewiesen, die folgenden beiden Zeilen dienen der Dokumentation. In der Zeile `metaclasses` werden sämtliche Metaklassen aufgelistet, denen die Klasse zuzuordnen ist. Durch diese Angaben werden bestimmte Eigenschaften der Klasse festgelegt, zum Beispiel bewirkt die Zuordnung zu `has-condition`, daß zusätzliche Slotgruppen zur Angabe von Vor- und Nachbedingungen zur Verfügung stehen (vergleiche Abschnitt 5.1). Die Zeile `superclasses` dient der Einordnung der Klasse in die Klassentaxonomie. Eine Klasse erbt sämtliche Slots ihrer Superklassen einschließlich deren Namen und Facetten. Die Facetten können unter Einhaltung bestimmter Regeln gegebenenfalls noch verändert werden. Zur Vererbung vergleiche Abschnitt 4.4.

Auf den Kopf des Frames folgt der Rumpf, in dem die Eigenschaften der Instanzen dieser Klassen angegeben werden. Diese Angaben werden in Gruppen eingeteilt, je nachdem, ob sie Attribute, Methoden, Restriktionen oder Bedingungen beschreiben. Jede dieser Kategorien wird im folgenden noch genauer erläutert.

Die Angabe der Eigenschaften erfolgt in sogenannten *Slots*, welche jeweils durch einen innerhalb der Klasse eindeutigen Namen identifiziert werden. Die Definition eines Slots setzt sich aus einzelnen *Facetten* zusammen. Welche Facetten zur Definition eines Slots notwendig oder erlaubt sind richtet sich nach der Art der durch den Slot beschriebenen Eigenschaft (Attribut, Methode, etc.) und gegebenenfalls auch nach den Metaklassen der Klasse (zum Beispiel dürfen Slots von Klassen, welche der Metaklasse `elementary-concept-frame` zugeordnet sind, keine `:comp`-Facetten enthalten).

class:	<name>
documentation:	<text>
edited-by:	{<actor> x <time>}
metaclasses:	{<metaclass>}
superclasses:	{<name>}
shared-intrinsic-attributes:	
<**attribute-name**>	:dom <valuetype *or* objecttype> :val <value *or* object> :do_X_Y :init :readonly :groupleader :to_group :doc
...	
shared-relational-attributes:	
<**attribute-name**>	:dom <objecttype> :val <objectname> :do_X_Y :init :inv :comp :exc :dep :readonly :groupleader :to_group :doc
...	
individual-intrinsic-attributes:	
<**attribute-name**>	:dom <valuetype *or* objecttype> :req :init :do_X_Y :readonly :groupleader :to_group :doc
...	
individual-relational-attributes:	
<**attribute-name**>	:dom <objecttype> :req :init :do_X_Y :exc :dep :inv :comp :readonly :groupleader :to_group :doc
...	
methods:	
<**method-name**>	:interface :intern <M_predicate> :extern :ref :doc
...	
laws:	
<**law-name**>	<predicate> :refinable :doc
...	

Tabelle 1: Allgemeiner Aufbau eines Frames

Attribute

Attribute können zum einen Eigenschaften des beschriebenen Konzepts selbst, zum anderen aber auch seine Beziehungen zu anderen Konstrukten beschreiben. Wir unterscheiden daher zwischen *intrinsischen* und *relationalen* Attributen.

Eine weitere Unterscheidung ergibt sich daraus, daß Attributwerte entweder für alle Instanzen einer Klasse (und deren Subklassen) gleich sind oder das Attribut für alle einzelnen der Klasse zugeordneten Objekte individuell verschiedene Werte annehmen kann. Im ersten Fall bezeichnen wir das entsprechende Attribut als *shared*, im zweiten als *individual*. Aus diesen Überlegungen heraus ergibt sich in VEDA eine Unterteilung der Attribute in vier verschiedene Kategorien, was sich im Rumpf der Frame-Definition in den vier Slotgruppen `shared-intrinsic-attributes`, `shared-relational-attributes`, `individual-intrinsic-attributes` und `individual-relational-attributes` wiederspiegelt.

Zur Spezifikation eines Attributs stehen die in den Tabellen 2 und 3 angegebenen Facetten zur Verfügung. Für eine detailliertere Erläuterung der Bedeutung der einzelnen Facetten wird auf [MBG+97, Kapitel 2] verwiesen.

Alle VEDA-Klassen sind so zu definieren, daß es jeweils eine endliche Menge von Objekten gibt, die alle Anforderungen der Klasse erfüllen. Die in Beispiel 4.1 dargestellten Klassendefinitionen sind in dieser Kombination dementsprechend unzulässig, da die Kombination der Facetten `:req t`, `:exc t` und `:comp t` bei Instantiierung der Klasse `class47` zu einer unendlichen Kette von erforderlichen Objekten führt, weil auf jede Instanz von `class18` nur genau ein Verweis durch die Slots `part3` bzw. `part55` erfolgen darf und diese Slots jeweils nicht undefiniert sein dürfen.

Beispiel 4.1 (Unzulässige Klassendefinition)

class: `class47`
metaclasses: `composite-concept-frame`
superclasses: `class12`

individual-relational-attributes:
part3 `:dom class18 :req t :exc t :comp t`

class: `class18`
metaclasses: `composite-concept-frame`
superclasses: `class7`

individual-relational-attributes:
part55 `:dom class18 :req t :exc t :comp t`

Restriktionen

Durch Restriktionen können die Werte von Attributen eines oder mehrerer Objekte eingeschränkt werden. Man unterscheidet dementsprechend zwischen lokalen Restriktionen und Interobjekt-Restriktionen.

key	arg	Kommentar
:comp	Default: f	Teil-von-Relation (part–of) kann nur bei ...-relational-attribute- Slots verwendet werden
	f	keine Teil-von- sondern beliebige Relation
	t	Slot enthält eine Teil-von-Beziehung
:dep	Default: f	Abhängigkeit der Existenz eines referenzierten von der Existenz des referenzierenden Konstruktes
	t	referenziertes Konstrukt wird gelöscht, wenn referenzierendes Konstrukt gelöscht wird
	f	referenziertes Konstrukt wird nicht gelöscht, wenn referenzierendes Konstrukt gelöscht wird
:do_X_Y	method_name	Definition von Triggern (Daemonen) X = before, after oder instead Y = read oder write
:doc	<text>	Dokumentation des Attributs, der Methode etc.
:dom		erlaubter Wertebereich der Wert NIL wird gesondert behandelt, er darf lediglich auftreten, wenn :req f gilt
	<concept>	eine Konzeptklasse
	<value>	primitive Daten
:exc	Default: f	Exklusivität eines referenzierten part–of–Konstrukts
	f	referenziertes Konstrukt darf über keinen anderen part–of Link referenziert werden
	t	Konstrukt kann von beliebig vielen anderen Konstrukten als part–of referenziert werden
:groupleader	Default: f	virtuelles Meta-Attribut?
	t	Ja; Attribut kann von :to_group Facetten in Subklassen referenziert werden
	f	Nein: Attribut kann nicht von :to_group Facetten referenziert werden
:init		Defaultwert bei Instaniiierung
	<'concept>	Instanz einer Konzeptklasse
	this.attribute-name	Wert eines anderen Attributs des Frames
	this.method-name	Methode zur Berechnung des Defaultwertes
	<'value>	ein primitives Datum

Tabelle 2: Definition der Schlüssel (Facets) in den Slots der Frames zur Beschreibung von Eigenschaften, Restriktionen und zugewiesenen Methoden.

key	arg	Kommentar
`:inv`	`attribute-name`	der Attributname, dessen Wert mit dem Objekt übereinstimmen soll, in dem `:inv` auftritt (im Fall von Mengen und Listen muß das Objekt in der Menge/Liste enthalten sein; `:inv` muß immer symmetrisch auftreten und darf nicht verwendet werden, um zusätzliche Einschränkungen des Wertebereichs eines Attributs vorzunehmen
`:readonly`	`Default: f`	ein Änderungsschutz
	`f`	Attributwert kann beliebig modifiziert werden
	`t`	Attributwert kann nur gelesen werden; falls er geändert werden soll, muß das Objekt gelöscht und mit dem neuen Attributwert neu erzeugt werden
`:req`	`Default: f`	Erforderlichkeit einer Werteangabe in der Instanz
	`f`	Werteangabe nicht erforderlich
	`t`	Werteangabe erforderlich ($\neq$ NIL), damit muß entweder `:val oder :init angegeben sein`
`:to_group`	`attribute_name`	Zuordung zu der durch ein 'virtuelles' Meta-Attribut definierten Attributgruppe
`:val`	s. `:init`	der Wert eines Slots; für `shared- ... -attribute` in der Klassendefinition oder für `individual-...-attribute` bei der Instantiierung

Tabelle 3: Definition der Schlüssel (Facets) in den Slots der Frames zur Beschreibung von Eigenschaften, Restriktionen und zugewiesenen Methoden (Fortsetzung).

Eine Restriktion besteht aus einem Namen, welcher innerhalb der Klasse eindeutig sein muß, und einem Prädikat, welches die Restriktion beschreibt. Das Prädikat besteht aus zwei Komponenten, dem Q(antification)-Satz, welcher die Objekte spezifiziert, für welche die Bedingungen des A(ssertion)-Satzes gelten müssen. Die verwendete Sprache entspricht in ihrer Ausdrucksstärke der Logik erster Stufe.

Methoden

Methoden repräsentieren numerische oder symbolische Funktionen, das heißt prozedurale Elemente. Sie können zum einen in einer dem Formalismus zur Angabe von Restriktionen ähnlichen Notation direkt in der Klassendefinition angegeben werden, oder aber extern, zum Beispiel als C-Programm, definiert werden. In beiden Fällen muß die Signatur, das heißt die Schnittstelle der Methode, angegeben werden.

Vor- und Nachbedingungen

Vor- und Nachbedingungen werden ausschließlich für die Definition von Ablaufschritten benötigt. Die Vorbedingungen fassen alle Bedingungen zusammen, welche erfüllt sein

müssen, bevor ein Ablaufschritt durchgeführt werden kann, die Nachbedingungen geben diejenigen Bedingungen an, welche nach der Durchführung des zugehörigen Schritts auf jeden Fall erfüllt sind, stellen also eine Art Garantie dar.

Anhand der Vor- und Nachbedingungen kann überprüft werden, ob ein gegebener Schritt in einer bestimmten Situation durchführbar ist oder ob die Durchführbarkeit eine Menge von Schritten in einer gegebenen Reihenfolge für jede mögliche Situation garantiert werden kann.

Auf die Vor- und Nachbedingungen wird in Abschnitt 7 ausführlich eingegangen.

Abfolgebedingungen

Abfolgebedingungen treten in VEDA nur in zusammengesetzten Ablaufschritten auf. Sie schlagen eine Reihenfolge vor, in der die Teilschritte eines solchen zusammengesetzten Schritts durchgeführt werden können.

Es ist zu beachten, daß die Menge der möglichen Ausführungsreihenfolgen der Teilschritte eines zusammengesetzten Ablaufschritts bereits durch die jeweiligen Vor- und Nachbedingungen der einzelnen Teilschritte eingeschränkt sein kann.

Die Abfolgebedingungen werden in Abschnitt 6 detailliert beschrieben.

Attributgruppierung und Gruppierung von Bedingungen

Auf der Grundlage von [Bau96] wurden in VEDA für Attribute und Bedingungen die Facetten `:groupleader` und `:to_group` eingeführt. Diese ermöglichen die Zusammenfassung von Attributen oder Bedingungen zu Gruppen. Dazu werden die betreffenden Attribute oder Bedingungen mittels der Angabe der `:to_group`-Facette einem durch `:groupleader t` gekennzeichneten Slot einer Oberklasse zugeordnet.

Bei der Verwendung der Gruppierungs-Facetten sind folgende Bedingungen zu beachten:

- Es können jeweils nur Attribute und Bedingungen innerhalb der gleichen Slotgruppe (z.B. `individual-relational-attributes`, `scheduling-conditions`, `conditions`) einander mittels `:to_group` und `:groupleader` zugeordnet werden.
- Im Fall der Attributgruppierung muß der Wertebereich der mittels `to_group` zu einem Slot der Oberklasse zugewiesenen Slots jeweils mit dem Wertebereich des Oberklassen-Slots kompatibel sein. Dazu muß er entweder ein Subtyp/eine Subklasse des Wertebereichs der Oberklasse sein oder – bei einem mengen-/listenwertigen Wertebereich des Oberklassen-Slots – Subtyp/Subklasse eines einzelnen Elements der Menge/Liste sein.
- Der Name des Slots der Oberklasse, zu dem Attribute oder Bedingungen mittels `to_group` zugeordnet wurden, kann zum Zugriff auf die ihm zugeordneten Attribute/Bedingungen verwendet werden.
- Attribute mit der Facette `:groupleader t` dürfen selbst keine Instanzen haben, das heißt sie können nicht die Facetten `:val` oder `:init` enthalten.

4.3 Instanzen

Das assertorische Wissen über eine Welt, das heißt das Wissen über (für die jeweilige Betrachtung relevante) Eigenschaften konkreter Objekte, wird durch die Beschreibung

konkreter Instanzen, deren Beziehungen untereinander und deren Eigenschaften dargestellt.

Der Aufbau einer Instanz wird durch die jeweilige Klasse vorgeschrieben, welcher die Instanz zugeordnet ist. Diese Klasse legt fest, welche Slots zur Beschreibung der Instanz verwendet werden können und welche Einschränkungen bezüglich der jeweiligen Werte der einzelnen Slots bestehen. Diese Einschränkungen erfolgen zum einen implizit durch die Angabe von zulässigen Wertebereichen (`:dom`-Facette), zum anderen explizit durch die Angabe von Restriktionen (`laws`). Bei dem Erzeugen einer Instanz ist zu beachten, daß alle mit der Facette `:req t` versehenen Slots mit gültigen Werten belegt werden müssen.

Instanzen sind in VEDA immer genau einer Klasse zugeordnet, das heißt wenn ein Objekt benötigt wird, welches gleichzeitig Instanz der Klassen `C` und `D` ist, muß zunächst eine Klasse `E` erzeugt werden, welche sowohl Subklasse von `C` als auch Subklasse von `D` ist. Diese kann dann entsprechend instantiiert werden.

4.4 Klassentaxonomie und Vererbung

Die Beziehungen von Klassen untereinander werden durch den Slot `superclasses` spezifiziert. Dadurch lassen sich die Klassen eindeutig in eine gerichtete, azyklische Taxonomie einordnen. Ein Teil der Taxonomie, wie sie bisher in VEDA besteht, ist in Abbildung 3 dargestellt.

Die Klassentaxonomie beschreibt eine Hierarchie der Konzepte im Sinne einer Spezialisierung, wobei allgemeinere Konzepte über spezielleren angeordnet sind. Eine Subklasse erhält alle Slots ihrer Superklassen (einschließlich der angegebenen Facets) und kann noch um eigene Slots erweitert werden.

Die Facetten geerbter Attribute dürfen modifiziert werden, sofern es sich bei der Modifikation um eine Spezialisierung handelt. Dabei gelten die folgenden Regeln, welche [MBG+97] entnommen sind:

- Für die Mengen- und Listenkardinalität gilt mit der Schreibweise `a` $\prec$ `b` für "`a` ist spezieller als `b`":

`n`	$\prec$	`m1..m2`	falls $m_1 \leq n \leq m_2$
`m1..m2`	$\prec$	`>=n`	falls $m_1 \geq n$
`m1..m2`	$\prec$	`<=n`	falls $m_2 \leq n$
`m1..m2`	$\prec$	`n1..n2`	falls $m_2 \leq n_2$ und $m_1 \geq n_1$

  Jede Beschränkung der Kardinalität einer Menge oder Liste ist eine Spezialisierung von "ohne Beschränkung".

- Bei `:comp`, `:req`, `:dep`, `:excl` und `:readonly` ist `t` spezieller als `f`.

- `:do_X_Y` dürfen beliebig hinzukommen oder wegfallen, die bei ihnen angegebene Methodennamen beliebig geändert werden.

- Jeder Text in `:doc` ist eine Spezialisierung eines anderen Textes.

- Der in `:dom` angegebene Wertebereich darf entsprechend der Taxonomie der Datentypen spezialisiert werden. Diese Regel gilt auch bei zusammengesetzten Datentypen und für deren Elemente.
- Jeder angegebene Initialisierungswert ist allgemeiner als ein nicht genannter. Jeder Initialisierungswert ist eine Spezialisierung eines anderen, solange dessen Typ mit dem Wertebereich kompatibel ist.
- Der bei `:inv` angegebene Attributname kann hinsichtlich des Zugriffspfades bei komplexen Datentypen in einer Unterklasse präzisiert werden.
- `:groupleader` und `:to_group` können im Rahmen einer Verfeinerung *nicht* geändert werden.

Auch geerbte Restriktionen (`laws`) dürfen spezialisiert werden, dadurch, daß die zur Formulierung der Restriktionen verwendete Sprache der Prädikatenlogik erster Stufe entspricht ist es allerdings nicht möglich, automatisch zu überprüfen, ob eine Restriktion korrekt spezialisiert[2] wurde. Die Möglichkeit der Modifikation von geerbten Restriktionen sollte daher mit Vorsicht verwendet werden.

Die Spezialisierung von Methoden wird zur Zeit nur anhand der Signatur überprüft, wobei zusätzlich eingeführte Parameter mit Default-Werten belegt werden müssen und die bereits bestehenden Parameter gleich bleiben oder spezieller werden müssen. Auch der Ergebnistyp der Methode muß gleichbleiben oder spezieller werden.

Durch die für VEDA angegebenen Spezialisierungsregeln ist keine Subtyp-Relation im Sinne von America [Ame90] (vgl. Abschnitt 8) definiert, das heißt es ist nicht *immer*, sichergestellt, daß ein spezielleres Konstrukt anstelle eines allgemeineren verwendet werden kann, obwohl dies meist der Fall ist.

In den folgenden Abschnitten wird für Ablaufschritte in VEDA eine Spezialisierungsrelation entwickelt, in welcher diese Bedingung der Einsetzbarkeit erfüllt ist. Diese wird jedoch nicht in den Vererbungsregeln für die betroffenen Klassen verankert, sondern davon unabhängig definiert.

[2] Eine speziellere Restriktion impliziert die weniger spezielle.

5 Ablaufmodellierung in VeDa

Dieses Kapitel gibt einen Überblick über die in VeDa für die Darstellung von Abläufen zur Verfügung stehenden Basisklassen. Es werden zunächst die zur Ablaufmodellierung benötigten Klassen vorgestellt und anschließend wird in Abschnitt 5.8 auf die Beziehungen der verschiedenen Klassen untereinander, ihre Wechselwirkungen und Besonderheiten eingegangen.

5.1 Die Metaklassen `has-condition` und `has-schedule`

Wie bereits in Abschnitt 4.1 dargestellt werden Metaklassen in VeDa nicht deklarativ angegeben, sondern fließen implizit in die Sprachdefinition mit ein (vergleiche auch [MBG+97]).

Die Metaklassen `has-condition` und `has-schedule` sind Subklassen der Klasse `frame-datatype`, welche den allgemeinen Aufbau von Klassen mit Hilfe von Frames [Hay79] definiert [MBG+97]. In `has-condition` und `has-schedule` werden die zusätzlichen Besonderheiten der Klassen zur Beschreibung von Modellierungsschritten spezifiziert. Dazu gehören zum einen die Vor- und Nachbedingungen, welche von den Klassen `STEP`, `PROCESS-FRAGMENT` und `INPUT-` und `OUTPUT-SITUATION` benötigt und in der Metaklasse `has-condition` definiert werden, und zum anderen die Abfolgebedingungen, welche von der Klasse `COMPOSITE-STEP` benötigt und in der Metaklasse `has-schedule` definiert werden.

Dementsprechend wird von `has-condition` eine Slotgruppe `conditions` eingeführt, unter welcher Attribute der Form

```
<condition-name>:    <ppc> :groupleader :to_group :refinable :doc
```

zusammengefaßt werden. `<ppc>` steht dabei für eine entsprechend der Syntaxdefinition in Abschnitt 7.5.2 definierte Bedingung.

In der Metaklasse `has-schedule` wird die Slotgruppe `scheduling-conditions` eingeführt, unter welcher Attribute der Form

```
<condition-name>:    <sc> :refinable :doc
```

zusammengefaßt werden. Dabei steht `<sc>` für eine entsprechend der in Abschnitt 6.2.1 angegebenen Syntax definierte Scheduling-Condition.

5.2 Basisklassen zur Einordnung in die Taxonomie

Die Klasse `DEVELOPMENT-CONCEPT` ist die gemeinsame Superklasse aller Konzepte zur Beschreibung von Entwicklungsprozessen in VeDa. Dazu gehören sowohl Beschreibungen von Abläufen als auch die Unterstützung und Dokumentation von Entscheidungen, welche im Rahmen eines Entwicklungsprozesses getroffen werden. `DEVELOPMENT-CONCEPT` dient lediglich der Strukturierung, es werden keine neuen Attribute, Methoden oder Restriktionen eingeführt. Abbildung 5 gibt einen Überblick über die Taxonomie der Klasse `DEVELOPMENT-CONCEPT` und ihrer Subklassen.

class:	`DEVELOPMENT-CONCEPT`
documentation:	`the common ancestor of all concepts needed for representing development processes`

metaclasses: concept-frame, abstract
superclasses: ROOT

ACTIVITY-CONCEPT ist ebenfalls eine Strukturierungsklasse, welche alle Konzepte zur Beschreibung von Abläufen zusammenfaßt. Einen Überblick über die Beziehungen der Subklassen von ACTIVITY-CONCEPT untereinander gibt Abbildung 4. Sie werden in Abschnitt 5.8 noch näher erläutert.

class: ACTIVITY-CONCEPT
documentation: the common ancestor of all concepts needed for a formal process-model
metaclasses: concept-frame, abstract
superclasses: DEVELOPMENT-CONCEPT

5.3 Die Klasse PROCESS-FRAGMENT und ihre Subklassen

Die Klasse PROCESS-FRAGMENT ist die gemeinsame Oberklasse aller CONTEXTs und PROCESS-CHUNKs. Diese bilden jeweils den "Kern" eines Ablaufschrittes und verweisen auf mögliche Implementierungen, Ziele und Eingangssituationen.

Der Slot status beschreibt den aktuellen Ausführungsstand des Ablaufschrittes, das heißt er gibt an, ob der Schritt gerade ausgeführt wird (enacting), ob er ausgeführt werden kann (enactable) oder nicht (not-enactable), ob nach seiner Beendigung die Nachbedingungen bereits überprüft wurden (terminated-succesfully) oder nicht (verify), ob seine Ausführung vorerst unterbrochen wurde, um später wieder aufgenommen zu werden (suspended, not-enactable-suspended) oder ob seine Ausführung endgültig abgebrochen wurde (aborted).

Über den Slot implementation wird auf die aktive Implementierung (STEP) verwiesen. Diese enthält Verweise auf weitere mögliche Implementierungen. Welche der möglichen Implementierungen aktiv ist, wird von den STEPs selbst verwaltet (vergleiche Abschnitt 5.4).

Die im Slot postconditions angegebenen Informationen werden lediglich für die Definition von Abläufen, nicht aber für deren Durchführung benötigt (vergleiche Abschnitt 7.3.1).

In der durch den Slot input-situation referenzierten INPUT-SITUATION werden das bearbeitete Produkt und die zur Durchführung des Ablaufschrittes erforderlichen Vorbedingungen verwaltet. Der Slot GOAL verweist auf das mit der Durchführung des Schritts verfolgte Ziel.

class: PROCESS-FRAGMENT
documentation: an entity representing fragments of modeling-processes; this can be process-chunks or single steps
metaclasses: concept-frame, has-condition, abstract
superclasses: ACTIVITY-CONCEPT

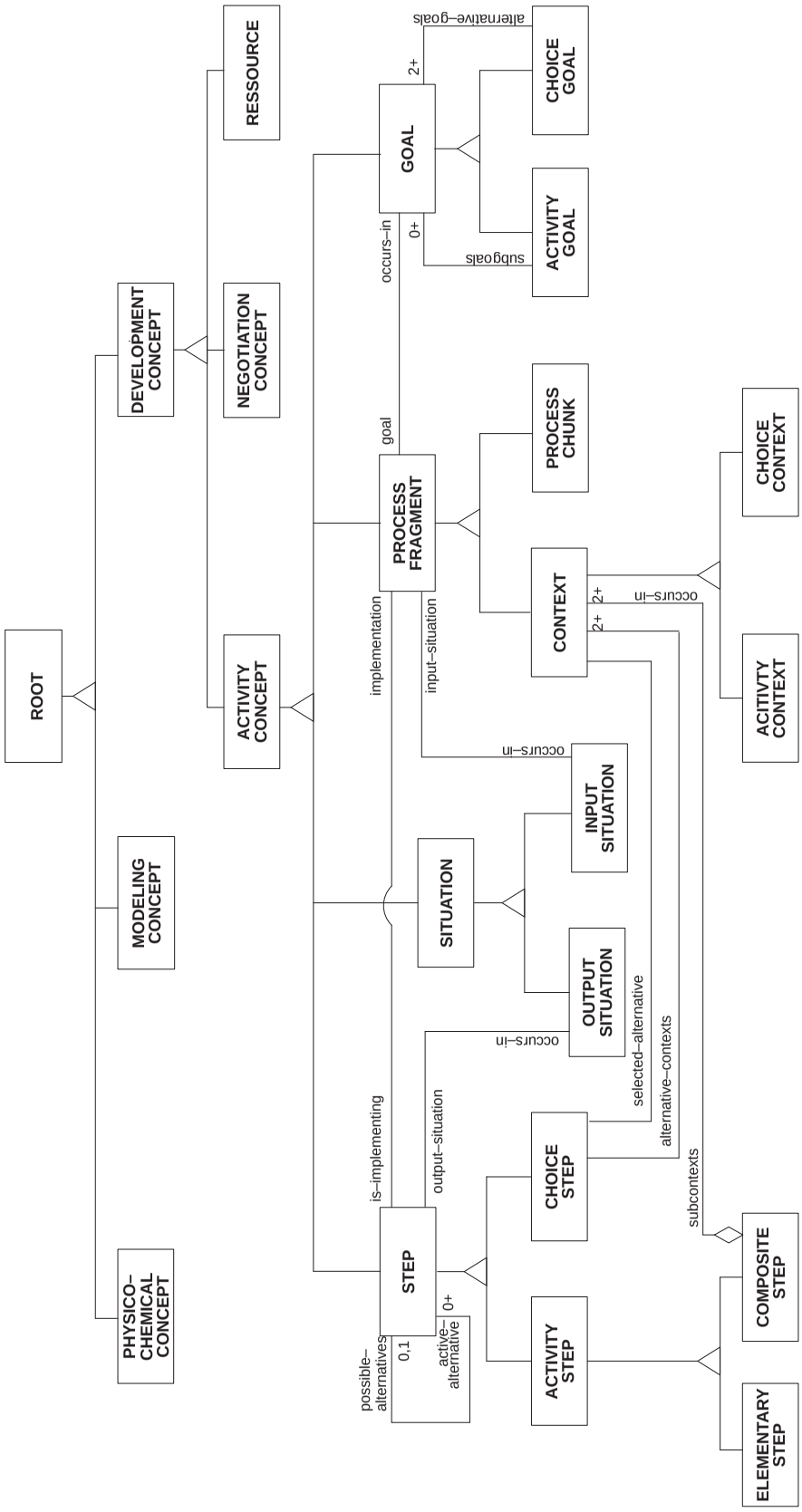

Abbildung 4: Übersicht über die Beziehungen der Klassen zur Darstellung von Ablaufschritten in VeDa (OMT-Notation)

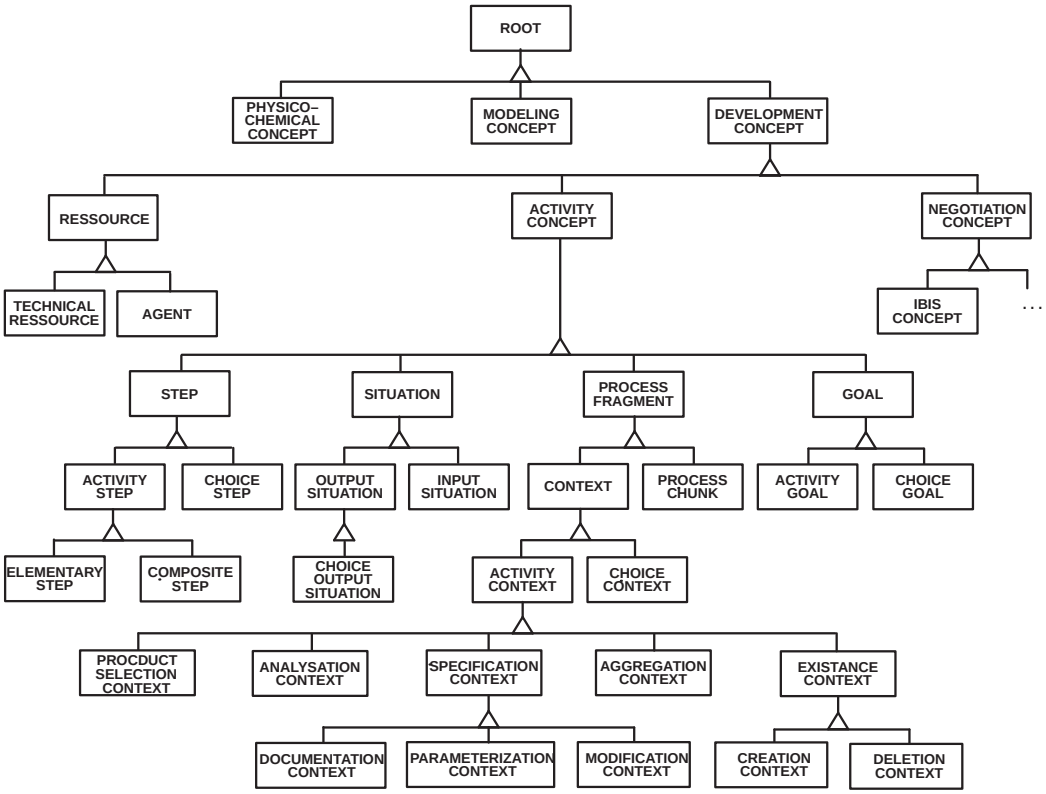

Abbildung 5: Übersicht über die Taxonomie der Klassen zur Modellierung von Ablaufschritten in VeDa

individual-intrinsic-attributes:

status: `:dom #1{'enactable, 'not-enactable, 'enacting, 'verify, 'terminated-successfully, 'suspended, 'not-enactable-suspended, 'aborted } :req t :init 'not-enactable :doc this attribute specifies whether the preconditions, scheduling-conditions and possibly other laws are satisfied (enactable/not-enactable);`

individual-relational-attributes:

implementation: `:dom STEP :inv is-implementing :doc the step which is the active implementation of this process-fragment`

input-situation: `:dom INPUT-SITUATION :inv occurs-in :req t :doc the input-situation includes the preconditions to be satisfied before the execution of the context and a set of objects satisfying these conditions`

goal: `:dom GOAL :inv occurs-in :req t :doc the goal to be achieved by this process-fragment`

conditions:

postconditions: `:groupleader t`
`:doc the postconditions which are supposed to be satisfied after the execution of the step`

Die Klasse `PROCESS-CHUNK` ist die gemeinsame Oberklasse aller Beschreibungen von größeren Teilstücken eines Modellierungsablaufs. Sie kann niemals als Bestandteil eines zusammengesetzten Ablaufschritts auftreten, sondern ist die 'größtmögliche' Einheit.

class: `PROCESS-CHUNK`
documentation: `an entity representing a well understood process chunk`
metaclasses: `concept-frame, has-condition, abstract`
superclasses: `PROCESS-FRAGMENT`
individual-relational-attributes:
implementation: `:dom STEP`
`:doc the step containing the active implementation for this chunk`
goal : `:dom ACTIVITY-GOAL`
`:doc the goal to be achieved by executing this process-chunk`

Ein `CONTEXT` ähnelt einem `PROCESS-CHUNK` insofern, als auch er ein Teilstück eines Modellierungsablaufs repräsentiert. Der Unterschied liegt hierbei in der Granularität der Betrachtung – während ein `PROCESS-CHUNK` die 'größtmögliche Einheit' der Zerlegung eines Ablaufs darstellt, beschreibt ein `CONTEXT` die Teilschritte, aus denen sich ein `PROCESS-CHUNK` oder ein `COMPOSITE-STEP` aufbauen läßt. Er ist immer Bestandteil eines zusammengesetzten Ablaufschritts oder eines Prozeßfragments.

class: `CONTEXT`
documentation: `an entity representing an activity to be executed during development processes`
metaclasses: `concept-frame, has-condition, abstract`
superclasses: `PROCESS-FRAGMENT`
individual-relational-attributes:
occurs-in: `:dom COMPOSITE-STEP :inv subcontexts`
`:req t`
`:doc the composite step this context occurs in`

Ein `ACTIVITY-CONTEXT` beschreibt elementare oder zusammengesetzte Ablaufschritte und wird somit immer durch `ELEMENTARY-` oder `COMPOSITE-STEP`s implementiert.

class: `ACTIVITY-CONTEXT`
documentation: `an entity representing either an elementary or a composite activity`

metaclasses: concept-frame, has-condition, abstract
superclasses: CONTEXT
individual-relational-attributes:
implementation: :dom ACTIVITY-STEP
:doc the active implementation for this context
goal: :dom ACTIVITY-GOAL

Anhand ihrer Aufgabe können ACTIVITY-CONTEXTs in fünf verschiedene Kategorien aufgeteilt werden. Es werden Schritte zur Produktauswahl, zur Spezifikation, Aggregation, Analyse vorhandener Bausteine und zum Erzeugen und Löschen neuer Bausteine unterschieden.

Die Spezifikationsschritte werden weiter unterteilt in Parametrisierung (z. B. Einschränkung von Wertebereichen, Wertzuweisungen), Modifikation (z. B. Umwandlung von Gleichungen) und Dokumentation. Die Schritte zum Erzeugen und Löschen von Bausteinen werden unter der Oberklasse EXISTANCE-CONTEXT zusammengefaßt.

CHOICE-CONTEXTs repräsentieren Auswahlschritte, welche die Auswahl zwischen verschieden alternativen Folgeschritten ermöglichen. Die möglichen Alternativen und die getroffene Auswahl werden von der Implementierung (CHOICE-STEP) verwaltet. Eine Auswahl zwischen verschiedenen Produkt-Typen oder ähnlichem wird *nicht* durch einen CHOICE-CONTEXT bzw. CHOICE-STEP modelliert. Für diesen Zweck stehen die Klasse PRODUCT-SELECTION-CONTEXT und ihre Subklassen zur Verfügung.

class: CHOICE-CONTEXT
documentation: an entity representing an activity choosing between at least two alternative activities
metaclasses: concept-frame, has-condition, abstract
superclasses: CONTEXT
individual-relational-attributes:
implementation: :dom CHOICE-STEP
:doc the active implementation for this context
goal: :dom CHOICE-GOAL
conditions:
selection: :is-defined (this.implementation.selected-alternative)
:or :eq(this.implementation.end-alternative,true)
:to_group postconditions

5.4 Die Klasse STEP und ihre Subklassen

Die Klasse STEP ist die gemeinsame Oberklasse für alle Implementierungen von CONTEXTen oder PROCESS-CHUNKs. Die Slots possible-alternatives und active-alternative dienen der Verwaltung der für ein bestimmtes PROCESS-FRAGMENT jeweils aktiven Implementierung. Dazu verweist jeder nicht-aktive STEP über den Slot active-alternative auf den aktiven STEP, während der aktive STEP über den Slot possible-alter-

natives auf die möglichen weiteren Implementierungen verweist. Die Veränderung der Inhalte dieser Slots zur Aktivierung einer alternativen Implementierung erfolgt durch die externe Methode make-active.

Die Attribute ressources und agents verweisen auf die technischen Ressourcen oder Personen, welche den Schritt ausführen. Jeder Schritt muß mindestens eine Ressource oder einen Agenten besitzen, welcher für seine Durchführung verantwortlich ist.

Die im Slot preconditions angegebenen Vorbedingungen dienen, wie auch die post-conditions der CONTEXTs, lediglich der Kontrolle bei der Definition neuer Abläufe, werden aber zur Durchführung eines Ablaufs nicht benötigt. Die Nachbedingungen des STEPs und das bearbeitete Produkt werden von der output-situation verwaltet.

class:	STEP
documentation:	an entity representing an implementation of a context
metaclasses:	concept-frame, has-condition, abstract
superclasses:	ACTIVITY-CONCEPT
individual-relational-attributes:	
is-implementing:	:dom PROCESS-FRAGMENT :inv implementation
possible-alternatives:	:dom #>=0{STEP} :req t :inv active-alternative :doc if this implementation is active, this slot contains references to the possible alternative implementations, otherwise it is empty
active-alternative:	:dom #<=1{STEP} :req t :inv possible-alternative :doc if this implementation is not active, this slot contains a reference to the active implementation, otherwise it is empty
output-situation:	:dom OUTPUT-SITUATION :req t :inv occurs-in :doc the output-situation contains the postconditions which are satisfied after the execution of the step
decision-issue:	:dom #>=0{ISSUE} :doc a set of issues a decision has to be made about during the execution of the step
ressources:	:dom {TECHNICAL-RESSOURCE} :inv occurs-in :doc the technical-ressources which execute the step
agents:	:dom {AGENT} :inv occurs-in :doc the user(s) who executes the step
methods:	
make-active:	:interface newactive STEP -> void :extern <'versioning.o' x 'makeactive'> :doc this method changes the active implementation of a process-fragment by changing the possible-alternative and

```
                        active-alternative slots of the steps
                        concerned and the implementation-slot
                        of the process-fragment;
                        additionally, the slot subgoals of
                        the goal associated with the context of
                        this step is changed to match the active
                        implementation
laws:
  active-or-possible:   :holds :count(this.possible-alternatives) :eq 0
                        :xor :count(this.active-alternative) :eq 0
                        :refinable t
                        :doc a step alternative may be either active
                        or possible
  agent-or-ressource:   :holds :count(this.ressources) :gt 0 :or
                        :count(this.agents) :gt 0
                        :doc there has to be at least one agent or
                        one ressource which executes the step
conditions:
  preconditions:        :groupleader t
                        :doc the preconditions of this step;
                        these conditions cannot be stronger than
                        the preconditions of the context this
                        implementation belongs to
```

Die Klasse ACTIVITY-STEP ist die gemeinsame Oberklasse von ELEMENTARY- und COMPOSITE-STEP. Diese sind immer Implementierungen von PROCESS-CHUNKs oder ACTIVITY-CONTEXTen, aber nie von CHOICE-CONTEXTen.

```
class:                  ACTIVITY-STEP
documentation:          an entity representing an implementation of
                        an activity-context
metaclasses:            concept-frame, has-condition, abstract
superclasses:           STEP
individual-relational-attributes:
  possible-alternatives: :dom #>=0{ACTIVITY-STEP}
                        :doc if this implementation is active,
                        this slot contains references to the
                        possible alternative implementations,
                        otherwise it is empty
  active-alternative:   :dom #<=1{ACTIVITY-STEP}
                        :doc if this implementation is not active,
                        this slot contains a reference to the
                        active implementation, otherwise it is empty
laws:
  no-choice-context:    :holds :not(this.context :in CHOICE-CONTEXT)
                        :doc the context an activity-step belongs to
                        is either a process-chunk or an activity-
                        context
```

Ein elementarer Ablaufschritt wird durch die Klasse `ELEMENTARY-STEP` repräsentiert. Er wird nicht weiter in Teilschritte untergliedert, sondern besteht in der Durchführung einer bestimmten Methode, welche im Slot `action` angegeben wird.

class:	`ELEMENTARY-STEP`
documentation:	`an entity representing an implementation of a context; in contrast to composite and choice steps an elementary step cannot be decomposed or refined`
metaclasses:	`concept-frame, has-condition, abstract`
superclasses:	`ACTIVITY-STEP`
individual-relational-attributes:	
is-implementing:	`:dom ACTIVITY-CONTEXT`
methods:	
action:	`:interface -> void` `:doc this is the method to be executed during the execution of this step`

Zusammengesetzte Ablaufschritte bestehen aus verschiedenen Teilschritten (`subcontexts`), welche jedoch nicht immer alle durchgeführt werden müssen. Dieser Fall tritt dann auf, wenn der `COMPOSITE-STEP` einen `CHOICE-CONTEXT` als `subcontext` enthält, da dann die Folgeschritte, zwischen denen der Auswahlschritt auswählt, Alternativen sind von denen nur eine ausgeführt wird.

Die mögliche Ausführungsreihenfolge der Teilschritte wird zum einen durch die jeweiligen Vor- und Nachbedingungen eingeschränkt, zum anderen wird durch die `scheduling-conditions` eine mögliche Reihenfolge vorgeschlagen.

class:	`COMPOSITE-STEP`
documentation:	`an entity representing an implementation of a context; this step has to be decomposed to at least two step contexts; all decomposed contexts contribute to achieving the goal of the composite step`
metaclasses:	`composite-concept-frame, has-condition, has-schedule`
superclasses:	`ACTIVITY-STEP`
individual-relational-attributes:	

subcontexts:	`:dom #>=2{CONTEXT} :groupleader t`
	`:inv occurs-in :comp t :req t`
	`:doc the contexts which occur in this composite step; in order to be able to specify the scheduling-conditions below, it must be possible to reference a subcontext by an identifier; therefore the subcontexts are referenced by attributes`

```
                        with :dom CONTEXT and facet :to_group
                        subcontexts and can be referred to by the
                        names of these attributes
  required-subcontexts: :dom {CONTEXT} :groupleader t
                        :doc the subcontexts of the composite-step
                        which have to be executed
  decision-issue:       :dom #>=0{ELIGIBLE-CONTEXT-ISSUE}
                        :doc the issue of choosing among at least
                        two alternative contexts; such a decision
                        issue can only occur during the execution
                        of a composite step if the scheduling-
                        conditions are not taken into account
laws:
  different-goals:      :all s1 :in CONTEXT :all s2 :in CONTEXT
                        :where s1 :ele-of this.subcontexts :and
                        s2 :ele-of this.subcontexts :holds
                        (s1.goal :nid s2.goal) :or (s1 :id s2)
                        :doc all subcontexts must have different
                        goals
  req-subcontexts-in-subcontexts:
                        :all context :in CONTEXT :where context
                        :ele-of this.required-subcontexts :holds
                        context :ele-of this.subcontexts
                        :doc all required subcontexts have to be
                        elements of the set subcontexts
scheduling-conditions:
  scheduling-condition:
                        :refinable t
                        :doc a suggestion in which order the
                        substeps could be executed
```

Ein `CHOICE-STEP` ist immer eine Implementierung eines `CHOICE-CONTEXT`s. Er enthält im Slot `alternative-contexts` Verweise auf die zur Wahl stehenden `CONTEXT`s und im Slot `selected-alternative` einen Verweis auf den vom Anwender ausgewählten und damit als nächstes auszuführenden `CONTEXT`. Nur wenn diese Auswahl erfolgt ist, kann der Auswahlschritt terminieren.

Dabei ist ein Spezialfall zu beachten: Es gibt Auswahlschritte, die es dem Modellierer ermöglichen, zu entscheiden, ob die Durchführung eines zusammengesetzten Schritts beendet oder fortgesetzt werden soll. In diesem Fall handelt es sich bei den Alternativen nicht nur um verschiedene `CONTEXT`s, sondern um einen oder mehrere `CONTEXT`s und einen *Beendigungsschritt*, der nicht explizit modelliert wird, da er weder Vor- und Nachbedingungen noch eine auszuführende Aktion oder Teilschritte besitzt. Ob ein solcher Beendigungsschritt gewählt werden kann, wird durch den Slot `end-alternative` angegeben.

Im Anschluß an den Auswahlschritt muß sofort der ausgewählte `CONTEXT` ausgeführt beziehungsweise – im Fall des Beendigungsschritts – die Ausführung des übergeordneten zusammengesetzten Schritts beendet werden.

class: CHOICE-STEP

documentation: an entity representing an implementation of a context; this step has to select one of at least two alternative contexts

metaclasses: concept-frame, has-condition

superclasses: STEP

individual-relational-attributes:

- **is-implementing:** :dom CHOICE-CONTEXT
- **possible-alternatives:** :dom #>=0{CHOICE-STEP} :doc if this implementation is active, this slot contains references to the possible alternative implementations, otherwise it is empty
- **active-alternative:** :dom #<=1{CHOICE-STEP} :doc if this implementation is not active, this slot contains a reference to the active implementation, otherwise it is empty
- **alternative-contexts:** :dom #>=1{CONTEXT} :req t :groupleader t :doc the contexts a choice has to be made between during the execution of this choice step
- **output-situation:** :dom CHOICE-OUTPUT-SITUATION
- **decision-issue:** :dom #>=0{ALTERNATIVE-CONTEXT-ISSUE} :doc the issue of choosing among at least two alternative step contexts
- **selected-alternative:** :dom CONTEXT :doc the context chosen to be the next one executed
- **end-alternative:** :dom boolean :readonly t :req t :doc this slot determines whether one of the alternative contexts has to be chosen or if it is possible to choose 'end', that is to choose to end the execution of the COMPOSITE-STEP this CHOICE-CONTEXT belongs to. In this case there is no selected-alternative to be executed next.

laws:

- **selected-from-possible:** this.selected-alternative :ele-of this.alternative-contexts :doc only a possible alternative context can be selected as the next one to be executed
- **same-situation:** :all context :in CONTEXT :where context :ele-of this.alternative-contexts :holds context.input-situation :id this.context.input-situation :doc this law ensures that the input-situation of the context of this choice-step is the same as the input-situations of it's alternative contexts

same-composite-step: `:all context :in CONTEXT :where context :ele-of this.alternative-contexts :holds context.occurs-in :eq this.context.occurs-in :doc this law ensures that the alternative contexts of a choice step belong to the same composite step as the context of the choice step; this law ist too restrictive and will be modified as soon as there is a formalism to express transitivity in laws (it would be sufficient if the contexts were in the same composite-step as the choice-context)`

5.5 Die Klasse SITUATION und ihre Subklassen

Die Klasse `SITUATION` ist im wesentlichen eine Strukturierungsklasse. Sie ist die gemeinsame Superklasse der Klassen `INPUT-` und `OUTPUT-SITUATION`. Diese Klassen haben einen Verweis auf die aktiven Produkte, das heißt diejenigen Produkte, welche verändert werden sollen oder wurden, gemeinsam. Zusätzlich müssen beide Klassen eine Methode besitzen, um diese aktiven Produkte zu bestimmen.

class: `SITUATION`
documentation: `the common ancestor of both input- and output-situation`
metaclasses: `concept-frame, has-condition, abstract`
superclasses: `ACTIVITY-CONCEPT`
individual-relational-attributes:
active-products: `:dom {MODELING-CONCEPT} :init this.get-active-products :doc the objects modified by the step or context`
container: `:dom {MODELING-CONCEPT} :doc this slot contains products which are needed to execute the step without being the active product; these may be products created or modified by steps which were executed before this step`
methods:
get-active-products: `:interface -> {MODELING-CONCEPT} :doc this method determines the active products of the situation; in input-situations, this will usually be achieved by user interaction, in output-situations this will be the products modified by the step`

Die Klasse `INPUT-SITUATION` ist immer mit einem `CONTEXT` verbunden. Sie dient der Beschreibung der Situation vor der Durchführung des Modellierungsschritts, das heißt

sie erfaßt die aktiven Produkte, welche verändert werden sollen, sowie die Bedingungen, welche vor der Ausführung des Schritts erfüllt sein müssen. Der Slot `products` enthält alle möglichen Produkte, welche aktiv sein könnten, im Slot `container` werden bei Bedarf Produkte abgelegt, welche von vorherigen Schritten erzeugt oder verändert wurden und zur Durchführung des zur `INPUT-SITUATION` gehörigen `CONTEXT`s benötigt werden. Diese Produkte müssen nicht unbedingt zu den im Slot `products` angegebenen gehören, das heißt es können auch Objekte sein, welche nicht den Typ des aktiven Produkts haben. Auf sie wird in der Regel nur lesend zugegriffen.

class: `INPUT-SITUATION`
documentation: `an entity describing the state of a model`
metaclasses: `concept-frame, has-condition, abstract`
superclasses: `SITUATION`
individual-relational-attributes:

occurs-in:
```
:dom {PROCESS-FRAGMENT}
:inv input-situation :req t
:doc the contexts this situation belongs
to; there may be more than one context an
input-situation might belong to, for example
a context with a composite-step-implemen-
tation and some of the subcontexts of this
composite-step
```

case:
```
:dom CASE-IMPLEMENTATION
:doc the active case which is modified by
the context
```

active-products:
```
:dom #>=1{MODELING-CONCEPT} :init
this.get-active-products(this.case,
this.products)
:doc the active products to be modified by
this context; these products are elements
of the set products
```

products:
```
:dom {MODELING-CONCEPT}
:init this.get-products
:doc the products which might become the
active-product: this is always a subset
of the products which are components
of the case
```

methods:

get-products:
```
:interface CASE case -> {MODELING-CONCEPT}
:doc a method to determine the instances
which might become the active-product: these
have to be components of the active case
```

get-active-products:
```
:interface {MODELING-CONCEPT} concepts ->
{MODELING-CONCEPT}
:doc this method determines the active
products
```

get-container:
```
:interface -> {MODELING-CONCEPT}
:doc this method determines the products
which are needed to execute this context,
```

```
                        but which are not necessarily elements of
                        the set products; usually these will be objects
                        created or modified by previously executed steps;
                        most of the time, neither the slot container nor
                        this method will be needed
laws:
  active-from-products: :all x :in MODELING-CONCEPT :where x :ele-of
                        this.active-product :holds x :ele-of
                        this.products
                        :doc this law ensures that the active
                        products are contained in the set products
conditions:
  preconditions:        :groupleader t
                        :doc the preconditions to be satisfied
                        before the execution of the context
```

In der Klasse `OUTPUT-SITUATION` werden neben den von dem zugehörigen Schritt veränderten Produkten auch die Nachbedingungen erfaßt, welche nach der Durchführung des Schritts erfüllt sein sollen.

```
class:                  OUTPUT-SITUATION
documentation:          an entity describing the state of a
                        model after the execution of a step
metaclasses:            concept-frame, has-condition, abstract
superclasses:           SITUATION
individual-relational-attributes:
  occurs-in:            :dom STEP :inv output-situation :req t
                        :doc the step this situation belongs to
conditions:
  postconditions:       :groupleader t
                        :doc the postconditions supposed to be
                        satisfied after the execution of the
                        step
```

Da alle Auswahlschritte eine Nachbedingung besitzen müssen, die aussagt, daß einer der zur Wahl stehenden Kontexte ausgewählt wurde, wird für diese eine eigene `CHOICE-OUTPUT-SITUATION` definiert.

```
class:                  CHOICE-OUTPUT-SITUATION
documentation:          an entity describing the state of a
                        model after the execution of a choice-step
metaclasses:            concept-frame, has-condition, abstract
superclasses:           OUTPUT-SITUATION
conditions:
  selection:            :is-defined(this.occurs-in.
                        selected-alternative) :or :eq(this.
                        occurs-in.end-alternative,true)
```

```
                    :to_group postconditions
                    :doc after the execution of a choice-
                    step, one of the alternative contexts
                    must be selected
```

5.6 Die Klasse GOAL und ihre Subklassen

Ein Ziel ist eine textuelle Beschreibung des vom Modellierer mit der Ausführung eines bestimmten Modellierungsschritts verfolgten Ziels. Es ist immer einem `CONTEXT` oder einem `PROCESS-CHUNK` zugeordnet.

class: `GOAL`
documentation: `describes the goal of a context`
metaclasses: `concept-frame, abstract`
superclasses: `ACTIVITY-CONCEPT`
individual-intrinsic-attributes:
  goal-description: `:dom string :req t`
  `:doc a textual description of the goal`
individual-relational-attributes:
  occurs-in: `:dom PROCESS-FRAGMENT :inv goal`
  `:doc the context or chunk this goal`
  `belongs to`

Unter der Klasse `ACTIVITY-GOAL` werden alle Ziele von elementaren oder zusammengesetzten Schritten aufgesammelt. `ACTIVITY-GOAL`s können auf Teilziele verweisen, welche zur Erreichung des Gesamtzieles notwendig sind. Diese Ziele hängen von der jeweils aktiven Implementierung des zugehörigen Schritts ab und werden bei einer Veränderung der aktiven Implementierung automatisch angepaßt. Gehört ein Ziel zu einem elementaren Ablaufschritt, so verweist es nicht auf Teilziele.

class: `ACTIVITY-GOAL`
documentation: `describes the goal of an activity-context`
metaclasses: `concept-frame, abstract`
superclasses: `GOAL`
individual-relational-attributes:
  subgoals: `:dom {GOAL} :groupleader t :do_before_read`
  `this.get-subgoals`
  `:doc the set of subgoals which have to`
  `be achieved by the active-implementation of`
  `the context`
methods:
  get-subgoals: `:interface -> {GOAL}`
  `:doc this method determines all the subgoals`
  `of the active implementation of the context;`
  `this is aquired by determining all subcontexts`
  `of the active implementation, the subcontexts`

```
                        of their implementations and so on
laws:
  no-choice-context:    :holds :not(this.occurs-in :in CHOICE-CONTEXT)
                        :doc this law ensures that an activity-goal
                        can only occur in a process-chunk or an
                        activity-context, but not in a choice-context
```

Ein CHOICE-GOAL beschreibt das Ziel eines Auswahlschritts. Es verweist auf verschiedene alternative Ziele, von welchen – in Abhängigkeit von der während der Durchführung des Schritts getroffenen Auswahl – jeweils eins erreicht werden muß.

```
class:                          CHOICE-GOAL
documentation:                  describes the goal of a choice-context
metaclasses:                    concept-frame, abstract
superclasses:                   GOAL
individual-relational-attributes:
  occurs-in:                    :dom CHOICE-CONTEXT
  alternative-goals:            :dom #>=2{GOAL} :req t :groupleader t
                                :doc the set of alternative goals of the
                                choice-context
```

5.7 Klassen zur formalen Beschreibung von Entscheidungen

Wie weiter oben bereits erwähnt wurde, müssen während der Ausführung kreativer Entwicklungsprozesse oft Entscheidungen getroffen werden, die durch ein formales Entscheidungsmodell unterstützt werden können. In VEDA werden dazu bisher lediglich Konzepte angeboten, welche auf dem sogenannten Issue-based Information Systems Modell (IBIS) basieren. Nach diesem Modell wird für die Lösung eines Problems (issue) jede zur Entscheidung anstehende Alternative (position) mit Hilfe von Argumenten (arguments) bewertet (vgl. [CB88]; [JM96]). Zu jedem Issue gibt es mindestens zwei Positionen, und zu jeder Position kann es mehrere Argumente geben. Basierend auf den Argumenten für und wider jede Position wird entschieden, welche Position das Problem am besten löst (decision). Eine Entscheidung muß nicht notwendigerweise alle Positionen betrachten; weiterhin müssen nicht sämtliche Argumente, die den einzelnen Positionen zugeordnet sind, berücksichtigt werden. In Abb. 6 sind die den einzelnen Komponenten des IBIS-Modells (issue, position, argument, decision) entsprechenden Konzepte dargestellt. Die Klasse IBIS-CONCEPT wurde als Strukturierungsklasse eingeführt [MBG+97].

Obwohl die Unterstützung und Dokumentation von Entscheidungen eine wichtige Aufgabe im Bereich der Ablaufmodellierung darstellt, wird im folgenden nicht weiter auf sie eingegangen, da sie die Durchführbarkeit eines Ablaufschritts nicht beeinflußt und deshalb für die Definition einer Spezialisierungsrelation für Ablaufschritte nicht von Bedeutung ist.

Dies gilt analog auch für die mit der Durchführung eines Schritts verfolgten Ziele, da diese in VEDA zur Zeit nur durch eine textuelle Beschreibung repräsentiert werden und sich somit einer formalen Spezialisierung entziehen.

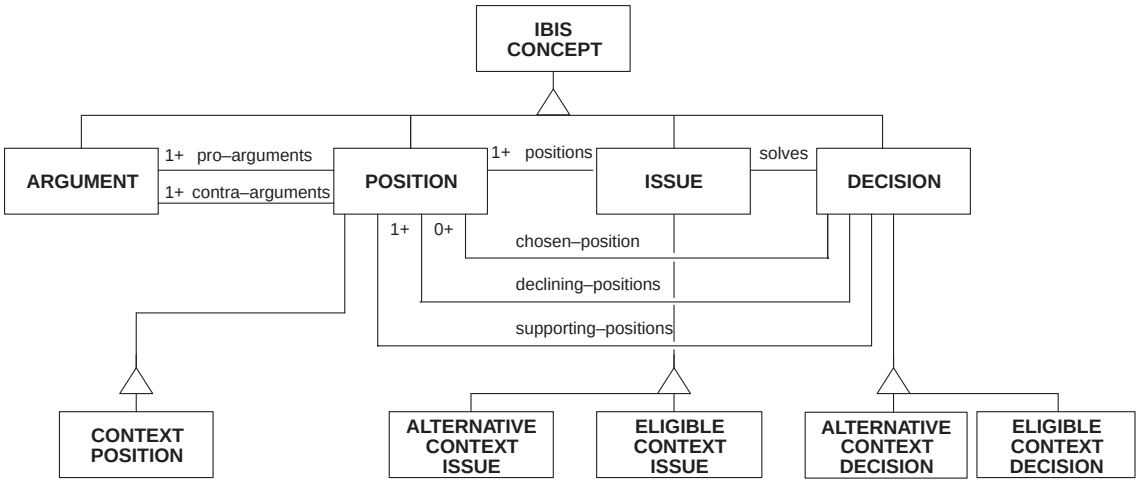

Abbildung 6: Spezialisierungs-/Generalisierungshierarchie der Konzepte zur Repräsentation von Entscheidungen.

5.8 Das Zusammenspiel der Klassen bei der Ablaufmodellierung

Ein Ablaufschritt wird in VeDa durch mindestens fünf Klassen bzw. deren Instanzen, nämlich `PROCESS-FRAGMENT`, `STEP`, `INPUT-SITUATION`, `OUTPUT-SITUATION` und `GOAL`, dargestellt. Durch die Angabe alternativer Implementierungen können zusätzliche Instanzen von `STEP` und `OUTPUT-SITUATION` hinzukommen. Zusammengesetzte Schritte enthalten zusätzlich ihre Teilschritte. Abbildung 4 gibt einen Überblick über die Beziehungen der einzelnen Klassen untereinander.

Darstellung einzelner Ablaufschritte

Das Kernstück der Darstellung eines Ablaufschritts bildet immer der `CONTEXT`. Er enthält die implementierungsunabhängigen Informationen, einschließlich Verweisen auf die `INPUT-SITUATION` und das angestrebte Ziel.

Außerdem findet sich im `CONTEXT` ein Verweis auf die aktive Implementierung. Dieser Verweis wird automatisch geändert, wenn ein anderer `STEP` aktiviert und somit zur aktiven Implementierung gemacht wird.

Darstellung von Abläufen

Neben den durch `CONTEXT`s repräsentierten Ablaufschritten gibt es noch die durch `PROCESS-CHUNK`s repräsentierten Abläufe. Diese zeichnen sich vor allem dadurch aus, daß sie nicht Bestandteil anderer Abläufe oder Ablaufschritte sein können.

Die typabhängige Auswahl des “richtigen” Schritts

Bei der Definition von Modellierungsabläufen werden häufig Schrittfolgen definiert, die für verschiedene Produkte weitgehend gleich sind, an einigen Stellen jedoch produktspezifische Funktionen beinhalten müssen.

Ein Beispiel ist ein zusammengesetzter Ablaufschritt zur genaueren Beschreibung eines Modellbausteins. Unabhängig davon, ob eine Komponente oder eine Verknüpfung erzeugt wird, kann der Ablauf jeweils das folgende Aussehen habe:

1. Typ des Modellbausteins spezialisieren

2. Modellbaustein benennen

3. Merkmalsliste des Modellbausteins ausfüllen

Je nachdem, ob der bearbeitete Baustein eine Komponente oder eine Verknüpfung ist, sollte in diesem Ablauf zum Beispiel in Schritt 3 ein passender `CONTEXT` ausgewählt werden.

In der allgemeinen Definition des Ablaufs wird anstelle eines konkreten `CONTEXT`s zur Durchführung einer Aktion eine allgemeine, nicht instantiierbare Klasse eingetragen. Die Process-Engine bestimmt während der Durchführung des Ablaufs dann aufgrund der vorliegenden Situation, welche der Spezialisierungen der allgemeinen Klasse an der entsprechenden Stelle des Ablaufs eingesetzt werden muß.

Der oben dargestellte Zusammenhang ist in Abbildung 7 skizziert.

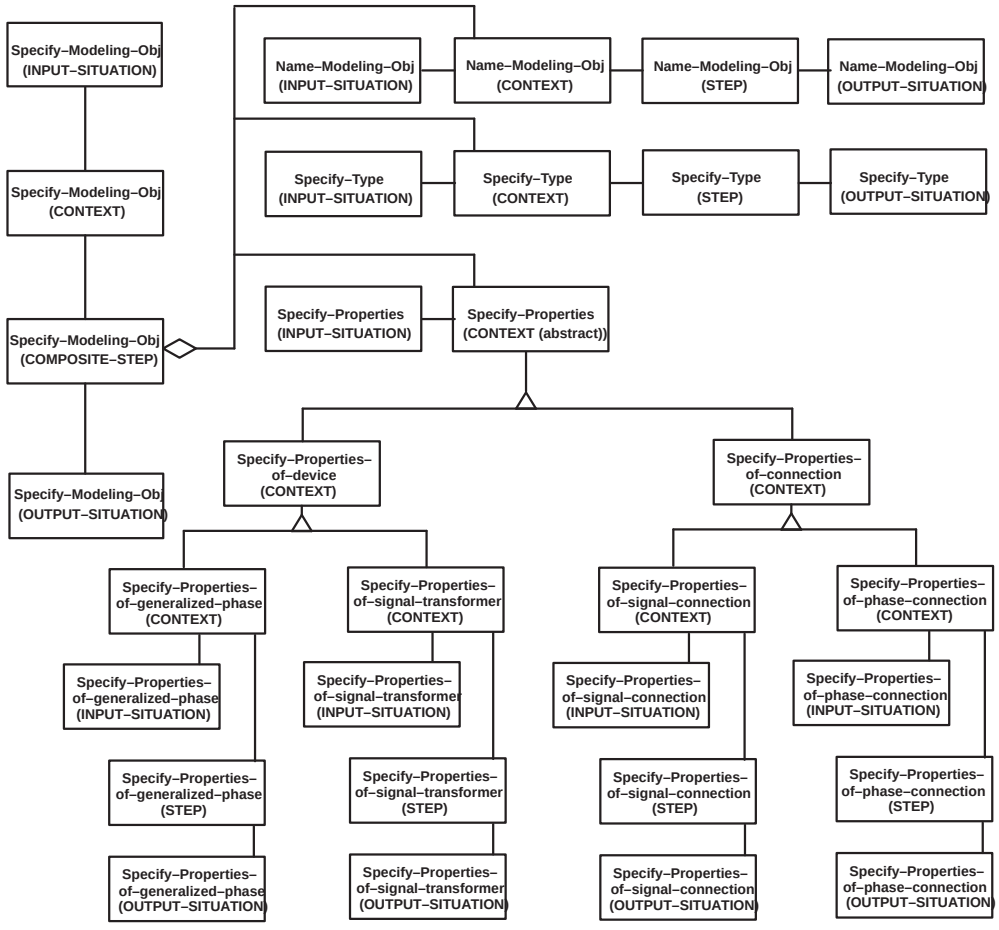

Abbildung 7: Darstellung von Modellierungsabläufen mit allgemeinen, typunabhängigen Klassen

6 Abfolgebedingungen für zusammengesetzte Ablaufschritte

Zusammengesetzte Ablaufschritte bestehen aus verschiedenen Teilschritten, von denen jeweils Teilmengen in bestimmten Abfolgen durchgeführt werden. Die Tatsache, daß eventuell nur eine Teilmenge der in einem zusammengesetzten Ablaufschritt enthaltenen Schritte durchgeführt wird, ergibt sich daraus, daß im Fall eines Auswahlschritts sämtliche Alternativschritte dem übergeordneten zusammengesetzten Schritt zugeordnet werden, aber jeweils nur einer dieser Schritte durchgeführt wird.

6.1 Die Aufgabe der Abfolgebedingungen

Mit Hilfe der Abfolgebedingungen (Scheduling-Conditions) wird innerhalb eines zusammengesetzten Ablaufschritts (genauer: in seiner Implementierung, das heißt in der Klasse `COMPOSITE-STEP`) angegeben, welche Ausführungsreihenfolgen für die Teilschritte empfohlen sind. Dazu ist es erforderlich, daß die Teilschritte über Namen referenziert werden können. Dies wird durch die Verwendung der Attributgruppierung [MBG+97, Bau96] für den Slot `subcontexts` erreicht, das heißt für jeden Teilschritt wird ein eigener Slot definiert, welcher mittels der Facette `:to_group` dem Slot `subcontexts` zugeordnet wird. Auf den betreffenden Teilschritt kann dann über den Namen des für ihn angelegten Slots Bezug genommen werden. Die `:dom`-Facette des Slots bestimmt jeweils den Typ und damit auch die Vor- und Nachbedingungen der einzelnen Teilschritte.

Die `COMPOSITE-STEP`s zu den beiden in Abbildung 8 dargestellten Abläufen enthalten dementsprechend die folgenden Attribut-Definitionen:

Definiere neuen strukturellen Modellbaustein

```
individual-relational-attributes:
  choose-obj-type:     :dom CHOOSE-OBJECT-TYPE :to_group subcontexts
  create-device:       :dom CREATE-DEVICE :to_group subcontexts
  create-connection:   :dom CREATE-CONNECTION :to_group subcontexts
  create-env-term:     :dom CREATE-ENVIRONMENTAL-TERMINAL
                       :to_group subcontexts
  char-mod-obj:        :dom CHARACTERIZE-MODELING-OBJECT
                       :to_group subcontexts
  document-mod-obj:    :dom DOCUMENT-MODELING-OBJECT
                       :to_group subcontexts
  ...
```

Spezifiziere Eigenschaften eines Signal-Source-Interface

```
individual-relational-attributes:
  name-if:             :dom NAME-INTERFACE :to_group subcontexts
  edit-if-location:    :dom EDIT-INTERFACE-LOCATION
                       :to_group subcontexts
  spec-signal:         :dom SPECIFY-SIGNAL :to_group subcontexts
  add-signal-or-exit:  :dom ADD-SIGNAL-OR-EXIT :to_group subcontexts
  ...
```

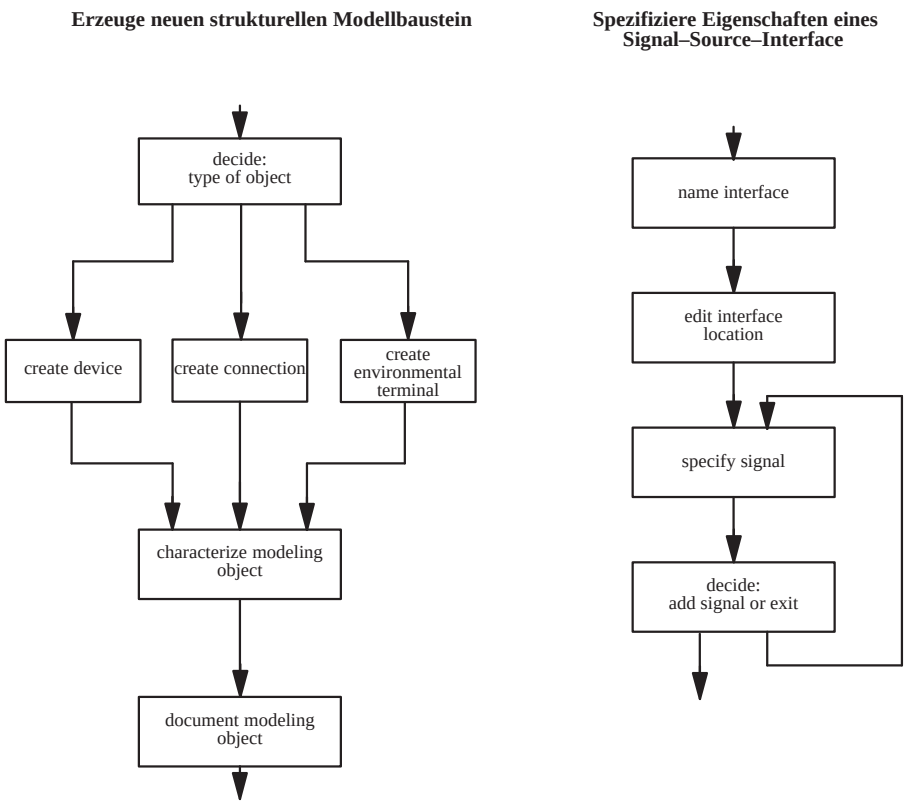

Abbildung 8: Beispiele für zwei zusammengesetzte Ablaufschritte

Wir unterscheiden zwei verschiedene Arten von Bedingungen, durch welche die Ausführungsreihenfolge der Teilschritte eingeschränkt wird:

- *Vor- und Nachbedingungen*
  Ein Schritt kann aufgrund seiner Vorbedingungen auf die Ergebnisse eines anderen Schrittes angewiesen sein.

- *empfohlene Abfolgebedingungen*
  Die Ausführungsreihenfolge wird im Sinne eines Vorschlags eingeschränkt, das heißt die empfohlenen Abfolgebedingungen dürfen vom Modellierer jederzeit verletzt werden. Dem Modellierer wird durch die empfohlenen Abfolgebedingungen eine mögliche und vollständige Ausführungsreihenfolge vorgeschlagen.

Die empfohlene Abfolge der Teilschritte für einen zusammengesetzten Ablaufschritt wird immer in dem Slot `scheduling-condition` angegeben, welcher in allen Subklassen der Klasse `COMPOSITE-STEP` vorhanden ist.

Bereits bei der Definition eines zusammengesetzten Ablaufschritts wird überprüft, ob die Abfolgebedingungen überhaupt mit den Vor- und Nachbedingungen der Teilschritte vereinbar sind (vergleiche Abschnitt 7.8). Die jeweiligen Vor- und Nachbedingungen der Teilschritte müssen die Durchführung der empfohlenen Abfolge ermöglichen. Für jeden zusammengesetzten Ablaufschritt muß eine empfohlene Abfolgebedingung existieren, die eine mögliche sinnvolle Abfolge für die Teilschritte definiert.

Während der Durchführung eines zusammengesetzten Ablaufschritts kann der Modellierer mit Hilfe der Vor- und Nachbedingungen und Abfolgebedingungen durch die Angabe von ausführbaren und empfohlenen Schritten geführt werden. Er wird bei der Modellierung unterstützt, ohne mehr als nötig eingeschränkt zu werden.

6.2 Der Formalismus zur Angabe von Abfolgebedingungen

Durch die Vor- und Nachbedingungen der Teilschritte eines zusammengesetzten Ablaufschritts wird die Menge der möglichen Ausführungsreihenfolgen der Teilschritte eingeschränkt, sie kann jedoch trotzdem noch eine unendliche Kardinalität besitzen. Durch die Abfolgebedingungen wird eine Menge konkreter Abfolgen definiert, welche eine Teilmenge der durch die Vor- und Nachbedingungen definierten Menge sein muß. Die Tatsache, daß es sich auch hier um eine Menge und nicht um eine spezielle Abfolge handelt resultiert daraus, daß in einer vorgeschlagenen Sequenz Auswahlschritte enthalten sein können, deren Folgeschritt von einer Entscheidung des Modellierers zum Zeitpunkt der Durchführung des Schritts abhängt. Dazu gehört auch die Entscheidung über eine wiederholte Durchführung eines bestimmten Schritts oder einer Schrittsequenz.

Es ist möglich, mit Hilfe des in Abschnitt 7.8 skizzierten Algorithmus die durch die Abfolgebedingungen vorgeschlagenen Abläufe auf ihre Durchführbarkeit hin zu überprüfen, sofern für Wiederholungen einzelner Schritte oder Schrittsequenzen eine Obergrenze für die Anzahl der möglichen Wiederholungen festgelegt wird[3].

Die hier definierte Sprache zur Angabe von Abfolgebedingungen umfaßt Konstrukte zur Angabe der folgenden Bedingungen:

- Die Schritte oder Schrittfolgen `A` und `B` müssen unmittelbar nacheinander durchgeführt werden (`:sequence(A,B)`)
- Die Schritte oder Schrittfolgen `A` und `B` sind alternativ zueinander, das heißt es wird entweder `A` oder `B` durchgeführt (`:alternative(A,B)`)
- Der Schritt oder die Schrittfolge `A` kann (innerhalb festgelegter Grenzen) wiederholt werden (`:loop(A)`)

Die Abfolgebedingungen geben jeweils einen Vorschlag für einen vollständigen Ablauf an. Sie definieren eine reguläre Sprache, deren Alphabet die Menge der Namen von Teilschritten ist, welche in dem betrachteten zusammengesetzten Ablaufschritt enthalten sind.

6.2.1 Definition der Syntax

Die Syntax der Abfolgebedingungen ist durch folgende BNF-Regeln (zur Definition von BNF vergleiche zum Beispiel [ASU86]) festgelegt. Startsymbol für die Ableitung ist *Start*.

[3] Dadurch wird erreicht, daß nur eine endliche Menge von Abfolgen überprüft werden muß.

Definition 6.1 (Syntax der Abfolgebedingungen)

Start ::= *seq* | *loop*.
seq ::= ':sequence(' *block* ',' *block* { ',' *block* }')'.
loop ::= (':loop' | (':loop#' *number*) | (':loop#<=' *number*) |
(':loop#>=' *number*) | (':loop#' *number* '...' *number*))
'(' *block* ')'.
alternative ::= ':alternative(' *block* ',' *block* { ',' *block* } ')'.
block ::= *seq* | *loop* | *alternative* | *string*.
number ::= *digit* {*digit-0*}.
digit ::= '1' | '2' |...| '9'.
digit-0 ::= *digit* | '0'.
string ::= *letter* { *letter* | *digit-0* | *ausym* }.
letter ::= 'a' | 'b' |...| 'z' | 'A' | 'B' |...| 'Z'.
ausym ::= '-' | '_'.

Eine zusätzliche kontextsensitiv Regel ist, daß string *immer einen Teilschritt des* COMPOSITE-STEP*s bezeichnet (das heißt der Name eines Slots ist, welcher mittels* :to_group *zu* subcontexts *gruppiert wurde).*

6.2.2 Definition der Semantik

Die Semantik der Abfolgebedingungen ist durch die folgende Definition festgelegt. Eine informelle Beschreibung wurde bereits am Anfang dieses Kapitels gegeben.

Definition 6.2 (Semantik der Abfolgebedingungen) *Eine Folge* $\phi = (x_1, \ldots, x_r)$ *von Bezeichnern von Teilschritten (Namen von Attributen, die zum Slot* subcontexts *gruppiert wurden) eines zusammengesetzten Ablaufschritts* erfüllt *eine Abfolgebedingung SC* ($\phi \models SC$) *genau dann, wenn die folgenden Bedingungen erfüllt sind. Dabei bezeichne* zahl *ein aus* zahl *abgeleitetes Literal,* x *und* y *bezeichnen jeweils Teilschritte des zusammengesetzten Ablaufschritts und* s_i, s *aus* block *ableitbare Literale.*

- $\phi \models$ x wenn $\phi =$x.
- $\phi \models$:sequence($s_1, \ldots,$ s_n) genau dann, wenn es eine Zerlegung $\phi_1 \cdot \cdots \cdot \phi_n$ von ϕ gibt, so daß $\phi_i \models s_i$ für alle $1 \leq i \leq$ n.
- $\phi \models$:alternative($s_1, \ldots, s_n$) genau dann, wenn $\phi \models s_i$ für ein i mit $1 \leq i \leq$ n.
- $\phi \models$:loop(s) wenn es eine Zerlegung $\phi_1 \cdot \cdots \cdot \phi_n$ von ϕ gibt, $n \in \mathbb{N}$ beliebig, so daß für alle $1 \leq$ j $\leq n$ gilt $\phi_j \models$ s.
- $\phi \models$:loop#n(s) wenn es eine Zerlegung $\phi_1 \cdot \cdots \cdot \phi_n$ von ϕ gibt, so daß für alle 1 $\leq$ j $\leq$ n gilt $\phi_j \models$ s.
- $\phi \models$:loop#n..m(s) wenn es eine Zerlegung $\phi_1 \cdot \cdots \cdot \phi_i$ von ϕ gibt, so daß n $\leq$ i $\leq$ m und für alle 1 $\leq$ j $\leq$ i gilt $\phi_j \models$ s.
- $\phi \models$:loop#<=n(s) wenn es eine Zerlegung $\phi_1 \cdot \cdots \cdot \phi_i$ von ϕ gibt, so daß i $\leq$ n und für alle 1 $\leq$ j $\leq$ i gilt $\phi_j \models$ s.

- $\phi \models$ `:loop#>=n(s)` wenn es eine Zerlegung $\phi_1 \cdot \cdots \cdot \phi_i$ von ϕ gibt, so daß i $\geq$ `n` und für alle $1 \leq$ j $\leq$ i gilt $\phi_j \models$ `s`.

6.3 Beispiele zur Veranschaulichung der Abfolgebedingungen

In diesem Abschnitt werden beispielhaft einige Abfolgebedingungen angeben und ihre Bedeutung aufgezeigt.

Betrachtet man das aus Abschnitt 6.1 (Abbildung 8) bekannte Beispiel des zusammengesetzten Ablaufschritts zum Erzeugen eines neuen strukturellen Modellbausteins, so ergibt sich als Abfolgebedingung

scheduling-condition:

```
:sequence( choose-obj-type,
           :alternative( create-env-term,
                         create-connection,
                         create-device),
           char-mod-obj,
           document-mod-obj)
```

Für den zusammengesetzten Ablaufschritt zum Spezifizieren der Eigenschaften eine Signal-Source-Interface ergibt sich

scheduling-condition:

```
:sequence(name-if,
          edit-if-location,
          :loop#>=1(:sequence( spec-signal,
                               add-signal-or-exit)))
```

Hier wird bei der Überprüfung der Durchführbarkeit eine obere Grenze für die Anzahl der Schleifendurchläufe angenommen, um so eine endliche Menge von möglichen Abfolgen zu erhalten. Das Ergebnis einer solchen Überprüfung ist allerdings nicht immer dahingehend zu interpretieren, daß *alle* diese Abfolgen *immer* durchführbar sind, da zum Beispiel das Löschen von Komponenten eines bestimmten Typs nur so lange wiederholt werden kann, bis keine entsprechenden Komponenten mehr existieren. Weil aber zum Zeitpunkt der Ablaufdefinition keine Informationen bezüglich der während der späteren Durchführung der Ablauffolge bestehenden Objektwelt vorliegen, können solche Probleme während der Ablaufdefinition nicht abgefangen werden. Eine ähnliche Einschränkung erhält man, wenn der Ablauf Verzweigungen enthält. Auch in diesem Fall kann nicht garantiert werden, daß zum Zeitpunkt der Durchführung alle Alternativen zur Wahl stehen, sondern nur, daß jeweils mindestens einer der alternativen Schritte ausführbar ist.

Ein weiteres Beispiel, anhand dessen die Angabe von Abfolgebedingungen demonstriert werden soll sind die in Abbildung 9 dargestellten Abläufe zur Beschreibung des Verhaltens von Modellbausteinen. Dazu werden Gleichungen angegeben, klassifiziert und Parameter spezifiziert sowie Freiheitsgrad- und Indexanalysen durchgeführt. Dieser Vorgang kann auf verschiedene Arten modelliert werden, je nachdem, welche Vorgehensweise gewählt wird. Als Abfolgebedingung ergibt sich für die linke Variante

```
:sequence( :loop#>=1(A),
           B, C
           :loop(:alternative(     :sequence(D,C),
                                   :sequence(D, :loop#>=1(A), B, C)
                                   :sequence(D, E, :loop#>=1(A), B, C)
                                   :loop( :sequence(D, E))))
           D, E)
```

und für die rechte

```
:sequence( A, B
           :loop (:alternative(    :sequence(C, D, E, A, B),
                                   :sequence(C, D, E, F)
                                   :sequence(C, D, E, F, A, B)))
           C, D, E, F)
```

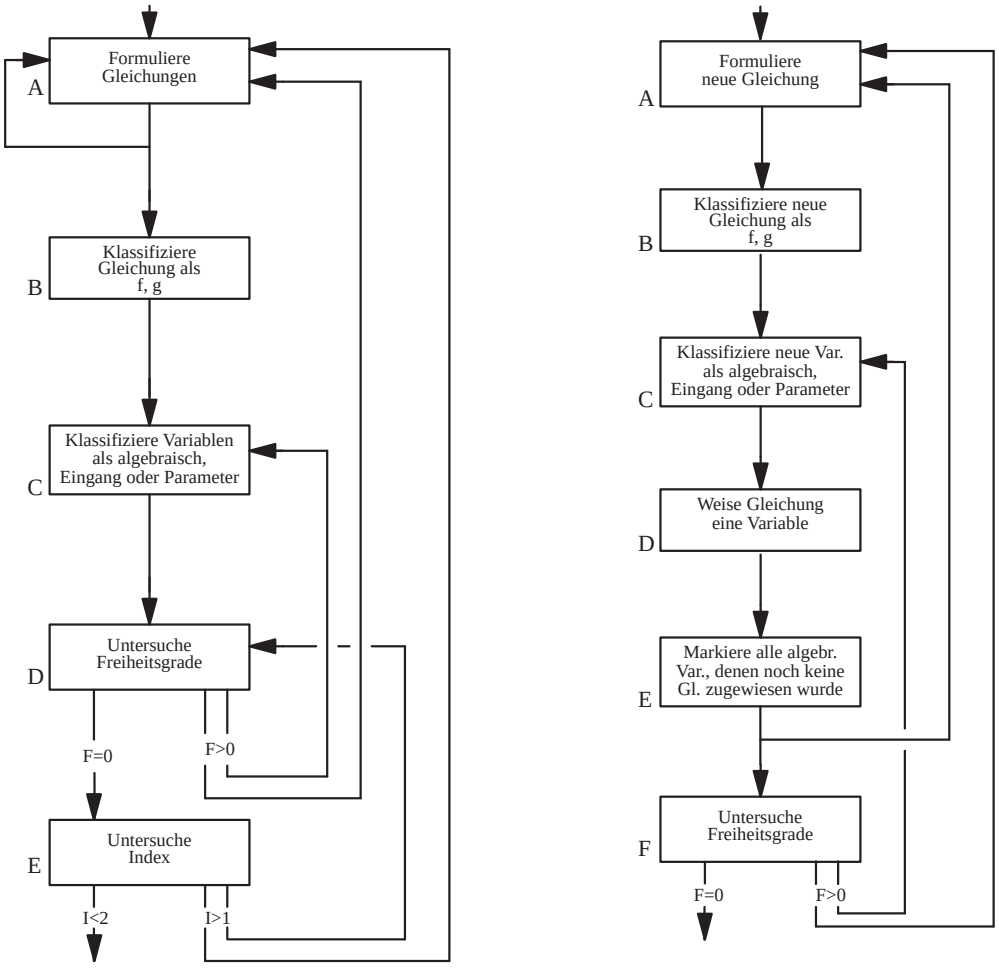

Abbildung 9: Verhaltensbeschreibung von Modellbausteinen, zwei Varianten

Als letztes Beispiel wird der Ablaufplan aus Abbildung 10, welcher [vW97] entnommen ist, betrachtet. Die zu diesem Ablauf gehörige empfohlene Abfolge ist

```
:sequence(
     A,
     :alternative( :sequence (B, C, D),
                   :sequence ( :alternative ( :sequence (B,C,E),
                                              F)
                               G)),
     J,
     :alternative (H,
                   K,
                   :sequence (K,I),
                   :sequence (K,I,H))).
```

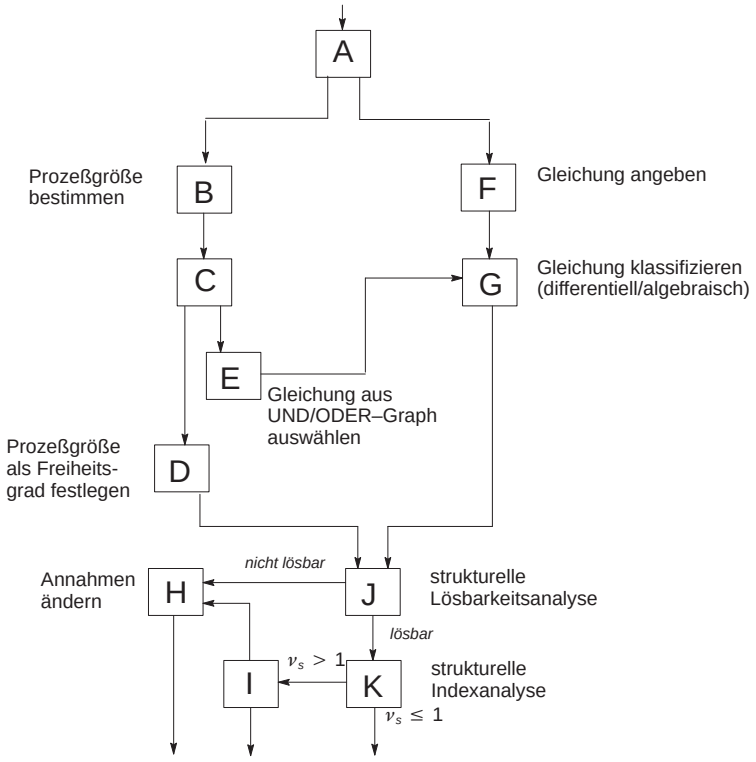

Abbildung 10: Definition einer neuen Gleichung oder Spezifizierung einer Variablen

6.4 Die Überprüfung der Abfolgebedingungen

Die Abfolgebedingungen sollen in erster Linie der Unterstützung des Modellierers während der Durchführung eines zusammengesetzten Ablaufschritts dienen. Dazu sollte jedoch schon bei der Definition eines Ablaufes sichergestellt werden, daß die vorgeschlagenen Abfolgen auch durchführbar sind.

6.4.1 Die Überprüfung der Abfolgebedingungen bei der Definition eines zusammengesetzten Ablaufschritts

Bereits während der Ablaufdefinition wird überprüft, ob die Abfolgebedingungen grundsätzlich mit den Vor- und Nachbedingungen der Teilschritte vereinbar sind (vergleiche Abschnitt 7.3.1).

Dazu müssen alle aufgrund der Abfolgebedingungen vorgeschlagenen Abfolgen der Teilschritte generiert werden und für jede dieser Abfolgen muß überprüft werden, ob sie mit den Vor- und Nachbedingungen der einzelnen Schritte vereinbar und deshalb grundsätzlich durchführbar ist. Nur wenn alle möglichen Abfolgen den Test bestehen, gelten die Abfolgebedingungen als zulässig. Bei Konflikten müssen die Bedingungen von Hand so lange verändert werden, bis sie mit den Vor- und Nachbedingungen vereinbar sind.

Eine weitere Anforderung an die Abfolgebedingungen besteht darin, daß immer eine Abfolgebedingung angegeben werden muß, welche einen vollständigen Ablauf definiert, der grundsätzlich durchführbar ist.

Damit nun die Durchführbarkeit *aller* möglichen empfohlenen Abfolgen überprüft werden kann, muß sichergestellt werden, daß es nur endlich viele solche Abfolgen gibt. Dies wird durch die Festlegung der maximalen Anzahl der Wiederholungen für Schleifen erreicht, welche nur für die Überprüfung, nicht aber für die tatsächliche Durchführung des Ablaufs relevant ist. Da im allgemeinen davon ausgegangen werden kann, daß eine Schleife, welche n-mal durchlaufen werden kann, auch $n + 1$-mal durchlaufen werden kann, ist diese Einschränkung akzeptabel.

Die Vorgehensweise bei der Überprüfung der Zulässigkeit einer empfohlenen Abfolgebedingung ist dementsprechend die folgende:

1. Generiere aus der Bedingung die (endliche) Menge aller durch die Bedingung beschriebenen Abfolgen.

2. Überprüfe für jede in der Menge enthaltene Folge mit Hilfe der in den Abschnitten 7.6 und 7.8 beschriebenen Algorithmen, ob die Folge durchführbar ist. Falls mindestens eine Abfolge nicht durchführbar ist müssen die empfohlenen Abfolgebedingungen angepaßt und die Überprüfung der Zulässigkeit wiederholt werden, anderenfalls ist die Abfolgebedingung zulässig.

6.4.2 Die Überprüfung der Abfolgebedingungen bei der Durchführung eines zusammengesetzten Ablaufschritts

Während der Ablaufdurchführung werden anhand der Vorbedingungen die jeweils als nächstes durchführbaren Schritte ermittelt. Wurden die empfohlenen Abfolgebedingungen bis zu dem betrachteten Zeitpunkt noch nicht verletzt, so kann nun aufgrund dieser Bedingungen aus der Menge der durchführbaren Schritte ein Vorschlag für den als nächstes auszuführenden Schritt ermittelt werden. Der Modellierer kann sich dann, gegebenenfalls unter Berücksichtigung des Vorschlags, für einen der möglichen Folgeschritte entscheiden.

7 Vor- und Nachbedingungen für die Durchführung von Ablaufschritten

7.1 Die Aufgaben der Vor- und Nachbedingungen

Vor- und Nachbedingungen von Ablaufschritten dienen der Beantwortung verschiedener Fragestellungen. Dazu gehören die Frage nach der Ausführbarkeit eines bestimmten Ablaufschritts in einer konkreten Situation, die Frage nach der erfolgreichen Terminierung eines Ablaufschritts, die Frage nach der prinzipiellen Durchführbarkeit eines neu definierten Ablaufs und die Frage, ob in einem bereits definierten Ablauf ein Schritt durch einen anderen ausgetauscht werden kann. Weiterhin bilden die Vor- und Nachbedingungen die Grundlage für die Definition einer Spezialisierungsrelation für Ablaufschritte.

Es werden zwei verschiedene Grundtypen von Aussagen unterschieden, aus denen sich die Vor- und Nachbedingungen eines Ablaufschritts zusammensetzen:

- Aussagen über Eigenschaften von Objekten, zum Beispiel den Typ oder den Inhalt bestimmter Slots
- Aussagen über Veränderungen, welche durch die Durchführung eines Ablaufschritts bewirkt werden, zum Beispiel die Erzeugung neuer Objekte oder die Veränderung des Typs eines Objektes

Während der erste Typus sich mit Hilfe der in [MBG+97] definierten Sprache für Restriktionen ausdrücken lassen würden, verursacht der zweite Typus insofern Schwierigkeiten, als Aussagen dieser Art einen Bezug zwischen der Situation vor und der Situation nach der Durchführung eines Ablaufschritts herstellen.

In diesem Kapitel wird ein eigener Formalismus zur Darstellung von Vor- und Nachbedingungen entwickelt, welcher die benötigte Ausdrucksstärke besitzt und für den das Erfüllbarkeitsproblem entscheidbar ist. Die Entscheidbarkeit des Erfüllbarkeitsproblems für Vor- und Nachbedingungen ist die Voraussetzung für die Definition einer in der Praxis anwendbaren Spezialisierungsrelation für Ablaufschritte, da sie es ermöglicht zu entscheiden, ob eine Bedingung eine andere impliziert.

Nach einem allgemeinen Überblick über die Anforderungen, die an den Formalismus zur Angabe der Vor- und Nachbedingungen gestellt werden, wird ein solcher Formalismus entwickelt. Anschließend wird analysiert, welche Beziehungen zwischen verschiedenen Bedingungen herrschen, das heißt es wird untersucht, wann zwei oder mehr Bedingungen im Widerspruch zueinander stehen und gezeigt, wie man für gegebene Bedingungen feststellen kann, ob eine die andere impliziert.

Diese Überlegungen liefern die Grundlage für zwei Dinge: Zum einen ermöglichen sie es festzustellen, ob ein zusammengesetzter Ablaufschritt mit einer gegebenen Abfolge seiner Teilschritte immer durchführbar ist (vergleiche Abschnitt 6), zum anderen sind die Bedingungen die Grundlage der Spezialisierungsrelation, welche im nächsten Abschnitt definiert wird. Für diese Definition ist die Entscheidbarkeit des Erfüllbarkeitsproblems und damit die Entscheidung darüber, ob eine Bedingung eine andere impliziert, unerläßlich.

7.2 Die Zuordnung von Vor- und Nachbedingungen zu den verschiedenen Klassen

Ein Ablaufschritt wird in VeDa nicht durch ein einzelnes Objekt, sondern durch eine Kombination von Instanzen der Klassen `PROCESS-FRAGMENT, STEP, INPUT-SITUATION, OUTPUT-SITUATION` und `GOAL` beschrieben (vergleiche Abschnitt 5.8). Diese Art der Darstellung ermöglicht es unter anderem, zwischen verschiedenen alternativen Realisierungen des gleichen Schritts umzuschalten. Die Menge von Instanzen, welche zur Beschreibung eines einzelnen Ablaufschritts verwendet werden, wird im folgenden kurz als *Schritt* oder *Ablaufschritt* bezeichnet.

Die Aufteilung in verschiedene Klassen, insbesondere die Zuordnung der `INPUT-SITUATION` zu `PROCESS-FRAGMENT` und der `OUTPUT-SITUATION` zu den `STEP`s (Abbildung 11), spiegelt sich auch in der Zuordnung der Vor- und Nachbedingungen zu den einzelnen Klassen wieder. Zum einen werden Vorbedingungen eines Schritts in seiner `INPUT-SITUATION` und Nachbedingungen in der `OUTPUT-SITUATION` seiner Implementierungen angegeben (wobei die Nachbedingungen je nach Implementierung variieren können). Diese werden insbesondere bei der Durchführung eines Ablaufschritts überprüft. Zum anderen werden zusätzlich in der Klasse `PROCESS-FRAGMENT` Nachbedingungen und in der Klasse `STEP` Vorbedingungen angegeben, welche ergänzende Informationen für die Definition von Abläufen bereitstellen.

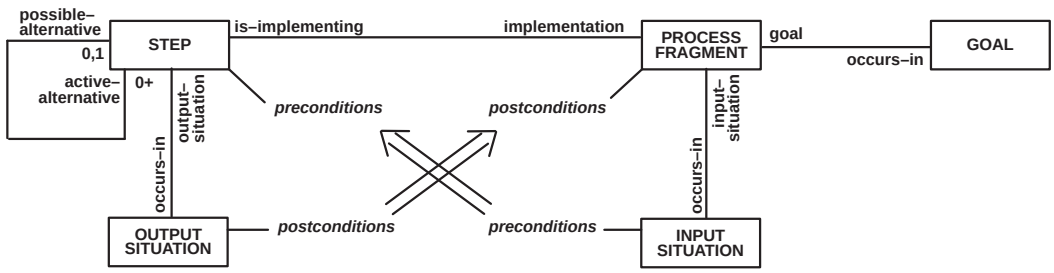

Abbildung 11: Darstellung eines Ablaufschritts in VeDa

7.3 Die Überprüfung der Vor- und Nachbedingungen

Wie im vorigen Abschnitt bereits angedeutet werden Vor- und Nachbedingungen sowohl bei der Definition von Abläufen als auch bei der Durchführung eines konkreten Schritts überprüft.

7.3.1 Die Überprüfung der Vor- und Nachbedingungen bei der Ablaufdefinition

Bei der Definition von Abläufen, das heißt bei der Angabe von Abfolgebedingungen innerhalb einer Subklasse von `COMPOSITE-STEP`, stehen nur Informationen über Klassen zur Verfügung, nicht aber über Instanzen. Dies gilt sowohl für die Ablaufschritte selbst, als auch für die Modellbausteine, auf denen sie operieren sollen.

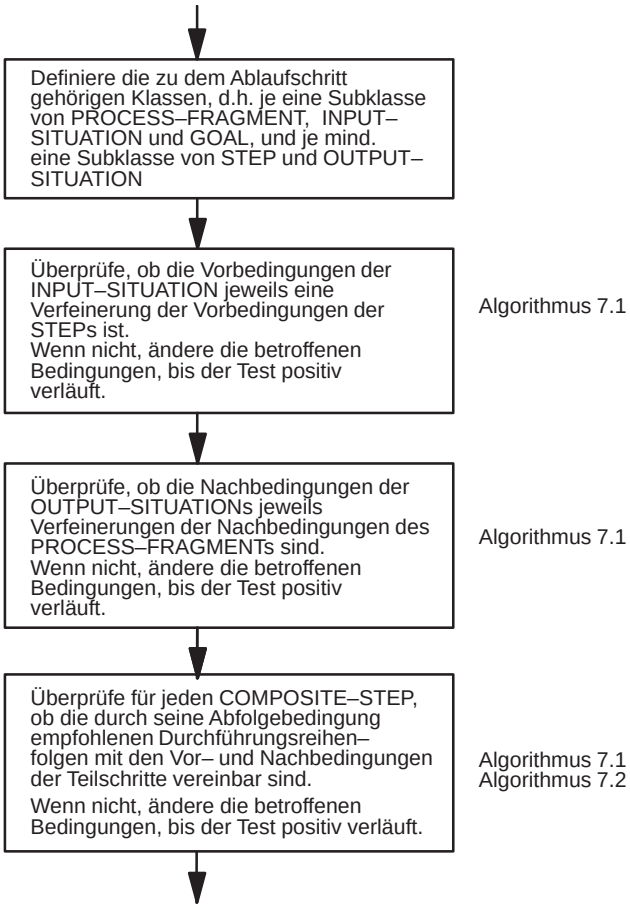

Abbildung 12: Vorgehensweise bei der Definition eines Ablaufschritts

Aus diesem Grund ist es wichtig, *allgemein* entscheiden zu können, wann eine Bedingung spezieller beziehungsweise allgemeiner ist als eine andere, ohne auf eine konkrete Objektwelt zu verweisen. Diese Eigenschaft wird durch die in Abschnitt 7.5 entwickelte Sprache zur Formulierung von Bedingungen erfüllt.

Die Vorgehensweise bei der Definition eines Ablaufschritts ist in Abbildung 12 dargestellt. Dabei können die einzelnen Teilschritte zum Teil auch in ihrer Reihenfolge vertauscht oder parallel ausgeführt werden.

Bei der Definition eines einzelnen Ablaufschritts ist darauf zu achten, daß folgende Bedingungen erfüllt sind (vergleiche Abbildung 13):

- Die in der `INPUT-SITUATION` angegebenen Vorbedingungen implizieren die Vorbedingungen der möglichen `STEP`s.
- Die in den `OUTPUT-SITUATION`s der möglichen Implementierungen (`STEP`s) angegebenen Nachbedingungen implizieren die Nachbedingungen des entsprechenden `PROCESS-FRAGMENT`s.

Diese Anforderungen an die Definition eines Ablaufschritts stellen sicher, daß zwischen den verschiedenen möglichen Implementierungen des Schritts umgeschaltet wer-

den kann, ohne daß das Verhalten des Schritts, so wie es "von außen" sichtbar ist, beeinflußt wird.

Um Bedingungen vergleichen zu können, welche in verschiedenen Klassen angegeben wurden, ist es notwendig, die in diesen Bedingungen auftretenden Pfadangaben bezüglich einer Klasse zu normieren, so daß zwei Pfade, welche den gleichen Slot beschreiben[4], auch syntaktisch gleich sind. Diese Konvention erleichtert die Angabe der in den folgenden Abschnitten vorgestellten Algorithmen, da ansonsten nicht offensichtlich ist, daß zum Beispiel `this.case` in einer `INPUT-SITUATION` den gleichen Slot bezeichnet wie `this.is-implementing.input-situation.case` in einem zugehörigen `STEP`.

Daher wird die Klasse `PROCESS-FRAGMENT` als Referenzklasse ausgewählt und alle Pfadangaben in Bedingungen anderer Klassen werden in Bezug auf diese normiert. Jedes `this.is-implementing` aus `STEP` wird dabei durch `this` ersetzt, jedes andere `this` durch `this.implementation`. Analog wird für `SITUATION`s vorgegangen. Dadurch "starten" alle Pfadangaben in sämtlichen Bedingungen in der gleichen Klasse, nämlich in `PROCESS-FRAGMENT`, und sind somit vergleichbar.

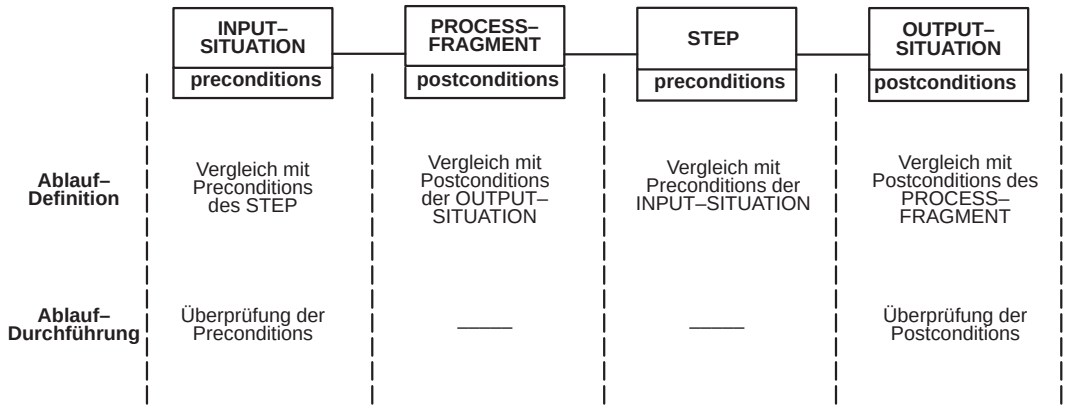

Abbildung 13: Überprüfung von Vor- und Nachbedingungen bei der Definition und Durchführung von Abläufen

Die oben angegebenen Bedingungen haben zur Folge, daß – unabhängig von der gewählten Implementierung – ein Schritt immer dann ausführbar ist, wenn die in seiner `INPUT-SITUATION` angebenen Vorbedingungen erfüllt sind, und daß nach der Durchführung eines Schritts die im `PROCESS-FRAGMENT` angegebenen Nachbedingungen erfüllt sind.

Diese Eigenschaft erlaubt es, einen Schritt A in einem gegebenen Ablauf durch einen Schritt B zu ersetzen, wenn die Vorbedingungen von A (bzw. der entsprechenden `INPUT-SITUATION`) die Vorbedingungen von B implizieren und die Nachbedingungen von B die Nachbedingungen von A implizieren.

Bei der Definition von zusammengesetzten Ablaufschritten muß beachtet werden, daß für alle möglichen Ausführungsreihenfolgen der Teilschritte (diese werden durch die Abfolgebedingungen festgelegt, vergleiche Abschnitt 6) die folgende Bedingung erfüllt ist:

Wenn *Schritt 1, Schritt 2, ... Schritt n* eine mögliche Ausführungsreihenfolge ist (*Schritt*

[4]Genauer gesagt den gleichen Verweis auf einen Slotfiller

1, ... , Schritt n sind Teilschritte des zusammengesetzten Schritts), dann gilt:

- Die Vorbedingungen von *Schritt 1* werden impliziert durch die Vorbedingungen des zusammengesetzten Schritts,
- die Vorbedingungen von *Schritt i* werden impliziert durch die Konjunktion aus den jeweils nicht von nachfolgenden Schritten verletzten Vorbedingungen des zusammengesetzten Schritts und den Nachbedingungen von *Schritt 1* bis *Schritt i-1*,
- die Nachbedingungen des zusammengesetzten Schritts werden impliziert durch die Konjunktion aus den jeweils nicht von nachfolgenden Schritten verletzten Vorbedingungen des zusammengesetzten Schritts und den Nachbedingungen von *Schritt 1* bis *Schritt n*,

Das führt dazu, daß immer, wenn der zusammengesetzte Ablaufschritt ausführbar ist, auch die Schrittfolge *Schritt 1, ... , Schritt n* ausführbar ist und nach der Ausführung der Schrittfolge die Nachbedingungen des zusammengesetzten Schritts erfüllt sind.

Die Berechnung derjenigen Bedingungen, welche nach der Durchführung einer bestimmten Schrittfolge erfüllt sind, d.h. die Bestimmung derjenigen Teilbedingungen der einzelnen Nachbedingungen, welche nicht durch einen nachgelagerten Schritt überschrieben werden, ist eine nicht-triviale Aufgabe. Auf sie wird in Abschnitt 7.8 noch genauer eingegangen.

In dem Beispiel aus Abbildung 14 sind die möglichen Ausführungsreihenfolgen *A, B, C* und *A, C, B*. Beide Reihenfolgen sind durchführbar, denn aus den Vorbedingungen des Gesamtschritts folgen die Vorbedingungen des ersten Schritts (*A*), aus den Nachbedingungen von Schritt *A* folgen sowohl die Vorbedingungen von Schritt *B* als auch die Vorbedingungen von Schritt *C*, keiner der Schritte modifiziert die Nachbedingungen eines vorigen Schritts und aus der Konjunktion der Nachbedingungen folgen die Nachbedingungen des zusammengesetzten Schritts *X*.

Hätte Schritt *B* allerdings die zusätzliche Nachbedingung `:undefined(dev.describes)`, so wäre nur die Schrittfolge *A, B, C* durchführbar.

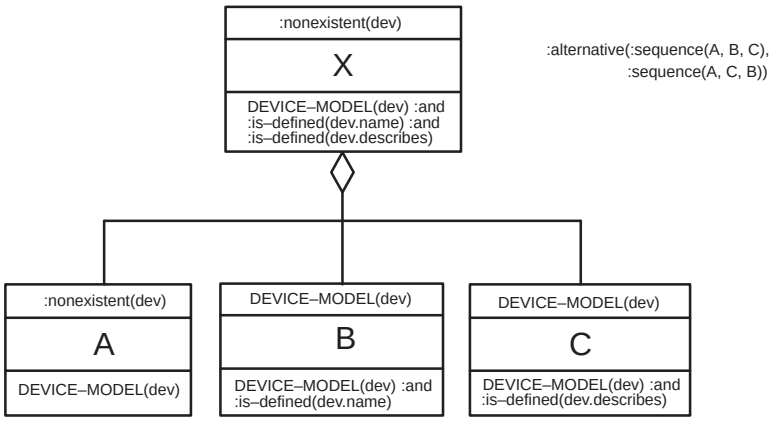

Abbildung 14: Beispiel für einen zusammengesetzten Ablaufschritt (vereinfachte Darstellung)

7.3.2 Die Überprüfung der Vor- und Nachbedingungen bei der Ablaufdurchführung

Während der Durchführung eines Ablaufs liegen konkretere Informationen vor als während der Definition, da in diesem Fall sowohl die Ablaufschritte als auch die Modelle, auf denen die Schritte operieren, als Instanzen existieren.

Während der Durchführung eines Ablaufs soll für die einzelnen Teilschritte jeweils entschieden werden, ob der Schritt in der gegebenen Situation durchführbar ist, beziehungsweise ob die Durchführung des Schritts erfolgreich beendet wurde.

Dazu werden jeweils die Bedingungen der `INPUT-` beziehungsweise `OUTPUT-SITUATION` des entsprechenden Schritts untersucht, während die den `PROCESS-FRAGMENT`s bzw. `STEP`s zugeordneten Bedingungen außer Betracht gelassen werden, da diese jeweils schwächer sind als ihr bei der Ablaufdurchführung betrachteter Gegenpart (vergleiche Abschnitt 7.3.1).

Weiterhin besteht die Möglichkeit, für einen Schritt festzustellen, ob seine Durchführung überhaupt sinnvoll ist. Dies kann in Frage gestellt werden, wenn seine Nachbedingungen bereits vor der Durchführung erfüllt sind. Die endgültige Entscheidung über die Durchführung eines Schritts sollte aber nicht ohne Bestätigung durch den Modellierer getroffen werden, da zum Beispiel die Nachbedingungen eines Schritts zur Erzeugung von Modellbausteinen nach seiner ersten Durchführung erfüllt sind, die wiederholte Durchführung des Schritts aber trotzdem sinnvoll sein kann.

7.4 Ausdrucksstärke der Vor- und Nachbedingungen

Wie in Abschnitt 7.1 bereits erwähnt sind die Grundbausteine, aus denen sich Vor- und Nachbedingungen zusammensetzen, Aussagen über Existenz und Typ von Objekten, Inhalte von bestimmten Slots von Objekten und die Veränderung von Objekten.

Um solche Aussagen treffen zu können, muß die zur Angabe von Vor- und Nachbedingungen definierte Sprache mindestens die Mittel zur Verfügung stellen, um folgende Bedingungen zu formulieren:

- ein gegebenes Objekt ist Instanz einer bestimmten Klasse (ist von einem bestimmten Typ)
- ein gegebenes Objekt ist *nicht* von einem bestimmten Typ
- ein Slot eines Objekts ist definiert/undefiniert
- ein Slot eines Objekts verweist auf ein Objekt eines bestimmten Typs
- ein Slot eines Objekts verweist auf ein bestimmtes Objekt oder enthält einen konkreten Wert
- ein oder mehrere Objekte von einem bestimmten Typ wurden erzeugt
- ein oder mehrere Objekte von einem bestimmten Typ wurden gelöscht

Die letzten beiden Aussagen bereiten insofern Schwierigkeiten, als sie eine Verbindung zwischen der Situation vor und der Situation nach der Durchführung eines Ablaufschritts herstellen.

Zur Formulierung der Nachbedingung *'Objekt x vom Typ C wurde erzeugt'* muß ein Rückgriff auf die Situation *vor* der Durchführung des Schritts und damit auf die Vorbedingungen erfolgen: Objekt x darf in dieser Eingangssituation nicht existieren!

7.5 Der Formalismus zur Angabe der Vor- und Nachbedingungen

In diesem Abschnitt wird die gewählte Sprache zur Formulierung von Vor- und Nachbedingungen motiviert, ihre Syntax und Semantik definiert und an einigen Beispielen erläutert. Anschließend wird auf die Entscheidbarkeit des Erfüllbarkeitsproblems in dieser Sprache eingegangen.

Eines der Hauptprobleme bei der Beschreibung der Auswirkungen eines Ablaufschritts in VeDa ergibt sich aus der Repräsentation eines Schritts durch Instanzen von fünf verschiedenen Klassen und der Möglichkeit, zwischen verschiedenen Implementationen (`STEP`s) zu wählen. Damit die Konsistenz der Definition eines Ablaufschritts überprüft werden kann, ist es notwendig, die einzelnen Komponenten des Schritts mit Vor- bzw. Nachbedingungen zu versehen, deren Implikationsbeziehungen untereinander entscheidbar sind. Die Konsistenz kann dann wie in Abbildung 13 dargestellt getestet werden.

Weiterhin macht es diese Art der Darstellung erforderlich, Beziehungen zwischen Bedingungen, welche in verschiedenen Klassen angegeben werden, herzustellen, zum Beispiel um die Erzeugung einer neuen Komponente durch einen Ablaufschritt formulieren zu können.

7.5.1 Motivation des gewählten Ansatzes

Die Betrachtung der in Abschnitt 3.3.3 vorgestellten Systeme zur Darstellung von Abläufen und zeitlichen Zusammenhängen zeigt, daß die dort skizzierten Formalismen nicht ohne größere Schwierigkeiten auf unser Anwendungsgebiet übertragbar sind.

Bei den STRIPS-basierten Ansätzen wie SAS$^+$ besteht das Hauptproblem bei der Übertragung auf das hier betrachtete Anwendungsgebiet darin, daß Zustände der Welt durch Propositionen beschrieben werden und dadurch nicht die von uns benötigte Ausdrucksstärke zur Verfügung steht. Es wäre zum Beispiel nicht möglich, auszudrücken, daß eine Verknüpfung `x` und eine Komponente `y` über eine Schnittstelle `z` verbunden sind.

Die temporalen Modelle wie zum Beispiel die Erweiterung von $\mathcal{ALCF}$, wie sie von Artale und Franconi [AF97] vorgestellt wird, verzichten auf die explizite Angabe von Vor- und Nachbedingungen einzelner Ereignisse bzw. Schritte zugunsten von Bedingungen, welche während des Ereignisses gelten, so daß der konkrete Bezug zu Zuständen, wie er für Ablaufschritte in VeDa erforderlich ist, nicht mehr besteht.

Bei den auf der Modallogik K aufbauenden Formalismen wie den von Baader und Laux [BL94] oder Grosse und Khalil [GK95] vorgestellten Sprachen besteht ein wesentliches Problem darin, daß Aussagen über Zustandsänderungen jeweils nur für unmittelbar aufeinanderfolgende Schritte getroffen werden können, während Bedingungen in zusammengesetzten Ablaufschritten in VeDa auch über mehrere Teilschritte hinweg ihre Gültigkeit erhalten können.

Weiterhin wäre jeweils die Übersetzung der VeDa-spezifischen Definitionen in die verwendete Basissprache wie beispielsweise $\mathcal{ALC}$ notwendig, was den Rahmen dieser Ar-

beit sprengen würde. Schließlich müssten für die Aussage, daß ein bestimmtes Objekt nicht existiert, insbesondere bei Baader und Laux [BL94] entsprechende Erweiterungen definiert werden, da aufgrund der *increasing domain restriction* Objekte nie "gelöscht" werden können.

Wie die Erläuterungen zeigen, wäre die Übertragung eines der in Abschnitt 3.3.3 beschriebenen Formalismen auf das hier betrachtete Problem der Angabe von Vor- und Nachbedingungen für Ablaufschritte mit verschiedenen Schwierigkeiten verbunden. Es wird daher im folgenden eine eigene Sprache definiert, welche auf die in dieser Arbeit bestehenden Bedürfnisse zugeschnitten ist, die erforderliche Ausdrucksstärke besitzt und für die das Erfüllbarkeitsproblem entscheidbar ist. Die Entscheidbarkeit des Erfüllbarkeitsproblems ermöglicht die Entscheidung darüber, ob eine Bedingung durch eine andere impliziert wird, und damit auch darüber, ob eine Bedingung spezieller ist als eine andere. Der Algorithmus, der testet, ob eine Bedingung spezieller als eine andere ist, ist allerdings eng verwandt mit den tableau-ähnlichen Verfahren, wie sie insbesondere in terminologischen Wissensrepräsentationssystemen verwendet werden (vergleiche z.B.[BH91b]).

Die Beziehung zwischen Vor- und Nachbedingungen, welche beim Erzeugen oder Löschen von Objekten auftritt, wird durch die Verwendung gemeinsamer Bezeichner in den entsprechenden Bedingungen gelöst. Das heißt durch die Formulierung *'Objekt x ist vom Typ C'* in der Vorbedingung und *'Objekt x ist vom Typ D'* in der Nachbedingung *eines* Schritts wird ausgedrückt, daß die Bedingungen sich auf das gleiche Objekt x beziehen, der entsprechende Schritt also eine Typumwandlung von C nach D an Objekt x durchführt und nicht etwa ein neues Objekt vom Typ D erzeugt.

Innerhalb der Klassen beziehungsweise Instanzen, welche einen Schritt definieren, gilt immer "gleicher Bezeichner, gleiches Objekt", das heißt eine bestimmte Variable bezeichnet in den Vor- und Nachbedingungen von `PROCESS-FRAGMENT`, `STEP`, `INPUT-` und `OUTPUT-SITUATION` eines Schritts immer das gleiche Objekt. Nur durch diese Vereinbarung ist es möglich, eine Verbindung zwischen den Vor- und Nachbedingungen eines Schritts herzustellen und damit Aussagen über Veränderungen des Zustands eines bestimmten Objektes treffen zu können.

Im Gegensatz dazu kann die gleiche Variable, welche in den Bedingungen zweier verschiedener Schritte verwendet wird, unterschiedliche Objekte bezeichnen.

Die Erzeugung neuer und das Löschen alter Objekte wird unter Zuhilfenahme eines Prädikats zur Darstellung des Zustands *'nicht existent'* formuliert, so daß zum Beispiel die Erzeugung eines neuen Objektes x vom Typ C durch die Vorbedingung *'Objekt x existiert nicht'* und die Nachbedingung *'Objekt x ist vom Typ C'* ausgedrückt werden kann. *'nicht existent'* kann dementsprechend als eigener 'Typ' aufgefaßt werden, welcher mit allen anderen Typen (den VeDa-Klassen) inkompatibel ist.

7.5.2 Syntax

Im folgenden seien

C = { `c` | `c` ist VeDa-Klassenname einer Klasse, die nicht Subklasse von `Primitive Datatype` ist}

V = { `v` | `v` ist Variablenbezeichner},

X = $V \dot{\cup} \{$ `this` $\}$,
P = $\{$ `p` | `p=`$\mathtt{p}_1 \ldots . \mathtt{p}_n$ für VEDA-Slotbezeichner $\mathtt{p}_1$, ... , $\mathtt{p}_n$, $n \in \mathbb{N}$ $\}$

Definition 7.1 (Syntax der Vor- und Nachbedingungen) *Für* `c` $\in C$, `x`,`y` $\in X$, `v` $\in V$, *und* `p` $\in P$, *sind folgende Konstrukte korrekte Vor- beziehungsweise Nachbedingungen:*

- `c(x)`
- `:not c(x)`
- `:nonexistent(v)`
- `:is-defined(x.p)`
- `:undefined(x.p)`
- `:all (x.p,c)`
- `:all (x.p, :not c)`
- `:eq(x,y)`
- `:eq(x.p,x)`
- `:some-eq(x.p,x)`

Wenn $\mathtt{A}_1 \ldots \mathtt{A}_n$ *korrekte Bedingungen sind, so sind auch* `(`$\mathtt{A}_1$ `:and ... :and` $\mathtt{A}_n$`)` *und* `(`$\mathtt{A}_1$ `:or ... :or` $\mathtt{A}_n$`)` *korrekte Bedingungen.*

Die Entscheidung darüber, ob eine Bedingung spezieller ist als eine andere, setzt die Existenz eines zusätzlichen Konstruktes voraus. Daher wird bereits an dieser Stelle der Begriff der *Formel* eingeführt.

Definition 7.2 (Syntax von Formeln) *Alle Bedingungen sind korrekte Formeln. Wenn* `C` *eine korrekte Formel ist, so ist auch* $\neg$`C` *eine korrekte Formel.*

Eine wichtige Regel bei der Angabe von Vor- und Nachbedingungen innerhalb eines Schritts besteht darin, daß in der Nachbedingung nur solche Variablen verwendet werden dürfen, die in der Vorbedingung des Schritts in jedem Disjunkt der Formel auftreten (bei Darstellung der Bedingung in disjunktiver Normalform) oder die in der Nachbedingung selbst explizit mit einer `:eq` oder `:some-eq` Bedingung an einen konkreten Pfad gebunden sind. Anderenfalls erhält man Nachbedingungen, in denen nicht klar ist, auf welches Objekt sie sich beziehen sollen.

Beispiel 7.1 *Sei die Vorbedingung eines Schritts* `:nonexistent(X)` *und die Nachbedingung* `C(X) :and D(Y)`. *Dann folgt aus der intuitiven Semantik der Bedingungen, daß ein Element vom Typ* `C` *von diesem Schritt neu erzeugt wird (Objekt* `X`*). Es ist aber nicht klar, ob das durch* `Y` *bezeichnete Objekt ebenfalls neu erzeugt wurde, ein bereits vor der Durchführung existierendes Objekt vom Typ* `D` *oder ein existierendes Objekt, welches vor der Durchführung des Schritts einen anderen Typ hatte, bezeichnen soll.*

7.5.3 Informelle Definition der Semantik

In diesem Abschnitt wird zunächst eine informelle Definition der Semantik des gewählten Formalismus angegeben. Dazu seien die Bezeichner wie in Abschnitt 7.5.2 gewählt. Tabelle 4 gibt die Bedeutung der atomaren Bedingungen an, die Operatoren `:and` und `:or` sind in der üblichen Weise definiert.

Ausdruck	**Bedeutung**
`c(x)`	Das durch `x` bezeichnete Objekt ist eine Instanz von `c` (oder Instanz einer Subklasse von `c`)
`:not c(x)`	Das durch `x` bezeichnete Objekt ist keine Instanz von `c` *Beachte:* `:not` entspricht nicht der logischen Negation
`:nonexistent(v)`	Variable `v` ist ein "nicht existierendes" Objekt
`:is-defined(x.p)`	der von Objekt `x` ausgehende Pfad `p` bezeichnet einen Slot, der nicht `NIL` ist
`:undefined(x.p)`	der von Objekt `x` ausgehende Pfad `p` bezeichnet einen Slot, der `NIL` ist
`:all (x.p,c)`	alle von Slot `x.p` referenzierten Objekte sind Instanzen der Klasse `c`
`:all (x.p, :not c)`	keines der von Slot `x.p` referenzierten Objekte ist Instanzen der Klasse `c`
`:eq(x,y)`	`x` und `y` sind identisch, das heißt sie bezeichnen das gleiche Objekt
`:eq(x.p,y)`	der Slot `x.p` referenziert das Objekt `y`
`:some-eq(x.p,y)`	der Slot `x.p` enthält mindestens eine Referenz auf Objekt `y`

Tabelle 4: Semantik der atomaren Bedingungen.

Bei der Vererbung von Vor- und Nachbedingungen bestehen folgende Grundregeln:

- Bedingungen werden benannt und mittels der `:to_group`-Facette den Slots `preconditions` oder `postconditions` zugeordnet. Die Vor- bzw. Nachbedingungen eines Schritts sind erfüllt, wenn sämtliche dem Slot `preconditions` seiner `INPUT-SITUATION` bzw. `postconditions` seiner aktiven `OUTPUT-SITUATION` zugeordneten Bedingungen erfüllt sind. Sämtliche Vor- und Nachbedingungen einer Superklasse werden an ihre Subklassen vererbt, können in der Subklasse aber, wenn sie die Facette `:refinable t` besitzen, durch andere Bedingungen überschrieben werden.
- Werden in verschiedenen einzelnen Vor- oder Nachbedingungen eines Schritts gleiche Bezeichner verwendet, so bezeichnen diese das gleiche Objekt.

Es wird gefordert, daß alle Vorbedingungen, welche auch nach der Durchführung des Schritts weiterhin erfüllt sind, explizit in den Nachbedingungen wiederholt werden. Dies ist nötig, da sonst nicht klar ist, welche der Vorbedingungen auch nach der Durchführung des Schritts noch gelten. Ein einfaches Beispiel soll diesen Sachverhalt verdeutlichen.

Beispiel 7.2 *Es wird die Vorbedingung* `ELEMENTARY-CONNECTION-IMPLEMENTATION(X)` *in Verbindung mit der Nachbedingung* `DIRECTED-CONNECTION-IMPLEMENTATION(X)` *betrachtet. In diesem Fall ist nicht eindeutig zu entscheiden, ob die Nachbedingung die*

Vorbedingung überschreiben soll, d.h. nach der Schrittdurchführung nur noch `DIRECTED-CONNECTION-IMPLEMENTATION(X)` *gelten soll, oder ob die Vorbedingung erhalten bleiben soll, so daß* `X` *eine* `SIGNAL-CONNECTION-IMPLEMENTATION` *ist. Einen Überblick über die im Beispiel verwendeten Klassen gibt Abbildung 15.*

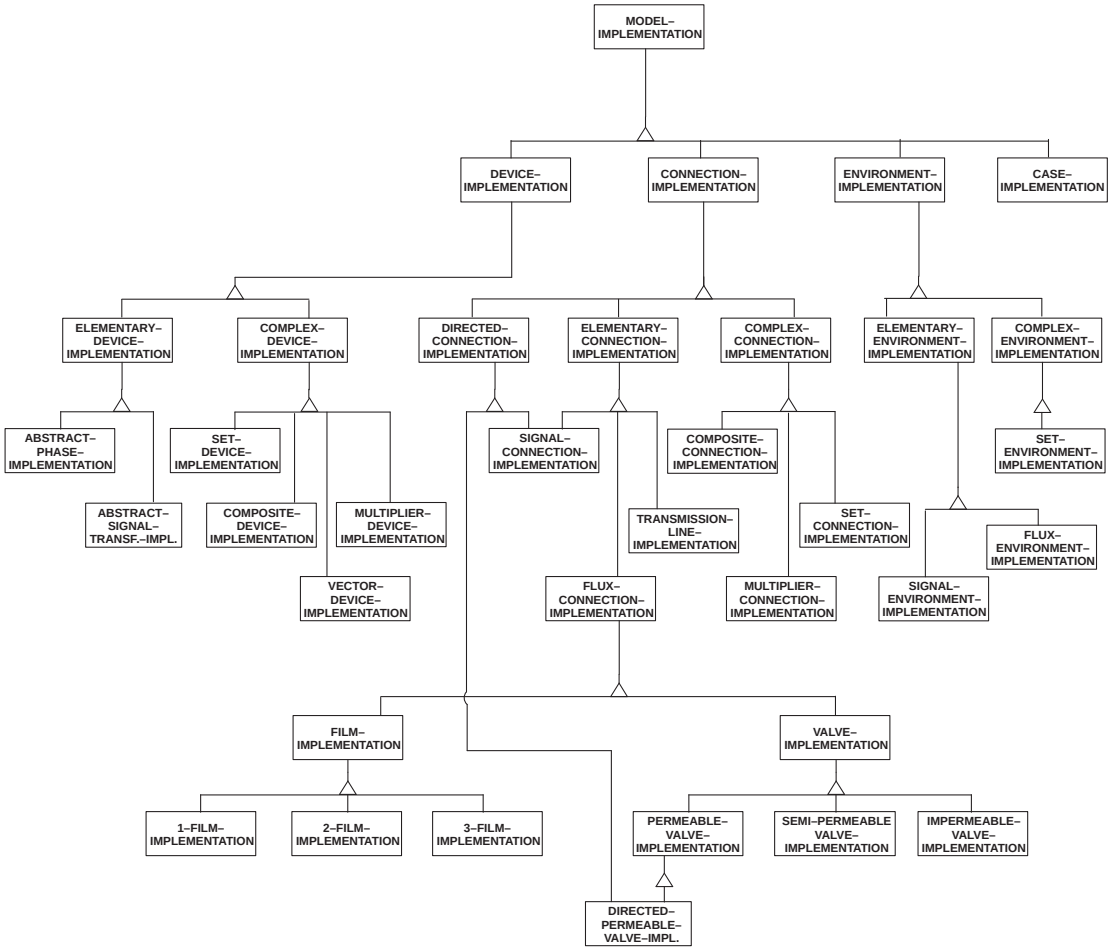

Abbildung 15: Überblick über die `MODEL-IMPLEMENTATION`-Taxonomie

Ein Problem tritt auf, wenn ein Schritt Änderungen an Objekten vornimmt, welche nicht durch seine Vor- und Nachbedingungen dokumentiert sind. Es wird davon ausgegangen, daß dies nicht geschieht. Diese Annahme läßt sich nicht durch Restriktionen sicherstellen, sondern begründet sich allein auf der Empfehlung an den Modellierer, für jeden Ablaufschritt alle möglichen Veränderungen mit Hilfe der Vor- und Nachbedingungen zu dokumentieren. Ein Verzicht auf diese Annahme würde insbesondere bei der Definition von zusammengesetzten Ablaufschritten Probleme verursachen, da Nachbedingungen eines in der Abfolge weiter vorne liegenden Schritts verletzt werden könnten, ohne daß dies aus der Spezifikation des verletzenden Schritts hervorgeht. Insofern wäre die Kontrolle eines Ablaufs in Hinblick auf seine Durchführbarkeit zum Zeitpunkt seiner Definition und somit vor der tatsächlichen Ausführung von vorneherein unmöglich. Insbesondere verlangen wir, daß im Fall von Löschungsvorgängen für alle gelöschten Objekte angegeben wird, daß sie gelöscht werden, selbst wenn sie über einen Slot mit

der Facette `:dep t` des zu löschenden Objekts referenziert werden und somit während der Schrittdurchführung automatisch verschwinden.

7.5.4 Formale Definition der Semantik

In diesem Abschnitt wird die Semantik der Sprache zur Angabe der Vor- und Nachbedingungen formal definiert.

Die in der Definition der Syntax der Vor- und Nachbedingungen verwendeten Variablen und Konzepte können auf verschiedene Weisen interpretiert werden. Zum Beispiel könnte `:nonexistent(X)` aufgefaßt werden als die Zusicherung, daß ein mit `X` bezeichnetes Objekt grün ist. Die Symbole besitzen also zunächst keine "Bedeutung". Diese wird ihnen erst dadurch zugewiesen, daß definiert wird, wie diese Symbole zu interpretieren sind. Dies geschieht durch die im folgenden angegebene Interpretation.

Sei nun $\Delta^{\mathcal{I}} = A \dot\cup \Gamma \dot\cup \{\texttt{NIL}\}$ der Interpretationsbereich, wobei für alle $a \in A$ gilt, daß a Instanz von `ROOT` ist und für alle $g \in \Gamma$ gilt, daß g nicht Instanz von `ROOT` ist. Jeder VEDA-Klasse ist eine Teilmenge von A zugeordnet, welche genau diejenigen Objekte enthält, welche Instanzen der Klasse sind.

Es seien
$$\begin{array}{lcl} \alpha & : & X \to A \cup \Gamma \\ \beta & : & P \to (2^A \cup \Delta^{\mathcal{I}} \to 2^A \cup A \cup \{\texttt{NIL}\}) \\ \gamma & : & C \to 2^A \\ \delta & : & C \times P \to \text{Facets} \end{array}$$

Die Abbildung α ordnet jedem Variablenbezeichner (einschließlich `this`) ein Element von $A \cup \Gamma$ zu. Die Elemente von A sind existierende Objekte, während die Elemente von Γ nicht existierende Objekte repräsentieren. Zu jedem Objekt $a \in A$ existieren beliebig viele[5] mit $\alpha(\texttt{v}) = a$.

β ordnet jedem Slotbezeichner `p` eine Abbildung zu, welche ein Element a aus A auf das Objekt (objektwertiger Wertebereich des Slots) oder die Menge (mengen- oder listenwertiger Wertebereich des Slots) von Objekten abbildet, welche von dem Slot `p` in a referenziert werden. Existiert in a kein entsprechender Slot oder ist der Slot undefiniert, so wird a auf `NIL` abgebildet. Teilmengen von A, Elemente aus Γ und `NIL` werden immer auf `NIL` abgebildet. Statt $\beta(\texttt{p})$ wird im folgenden kurz β_p geschrieben. Für einen zusammengesetzten Pfad $\texttt{p=p}_1\texttt{.}\ldots\texttt{.p}_n$ ist $\beta_{\texttt{p}}(a)$ definiert als $\beta_{\texttt{p}_n}(\beta_{\texttt{p}_{n-1}}(\ldots(\beta_{\texttt{p}_1}(a))\ldots))$.

Die Abbildung γ ordnet jeder VEDA-Klasse `c` diejenige Teilmenge von A zu, welche genau die Instanzen von `c` enthält. Für eine abstrakte Klasse `c` mit den nicht abstrakten Subklassen $\texttt{c}_1, \ldots, \texttt{c}_n$ gilt $\gamma(\texttt{c}) = \gamma(\texttt{c}_1) \cup \cdots \cup \gamma(\texttt{c}_n)$.

δ ordnet jedem Paar aus VEDA-Klasse und Attributbezeichner die Menge der in dem Attribut in der entsprechenden Klasse und ihren Subklassen vorhandenen Facetten zu. Enthält die Klasse kein entsprechendes Attribut, so bildet δ das Paar auf die leere Menge ab.

Eine Interpretation $\mathcal{I}$ ist gegeben durch Angabe des Interpretationsbereichs $\Delta^{\mathcal{I}}$ und der Abbildungen α, β, γ und δ. Dabei hängt δ allein von den Klassendefinitionen in VEDA und nicht von der betrachteten Objektwelt ab und muß deshalb nicht explizit

[5] Diese Voraussetzung wird von Algorithmus 7.1 ausgenutzt. Dort werden jeweils neue Namen für Objekte eingeführt, welche Teil eines längeren Pfades sind. Der Algorithmus erzeugt allerdings pro Objekt nur endlich viele Namen, so daß auch für jedes $\texttt{v} \in X$ nur endlich viele Bezeichner existieren müssen.

angegeben werden. Eine Interpretation wird daher bezeichnet durch $\mathcal{I} = (\Delta^{\mathcal{I}}, \alpha, \beta, \gamma)$.

Definition 7.3 (Semantik der Vor- und Nachbedingungen) *Seien* `c` $\in C$, `x`, `y` $\in X$ *und* `p` $\in P$. *Eine Interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \alpha, \beta, \gamma)$ erfüllt ($\models$) *die atomare Bedingung*

- `c(x)` *genau dann, wenn* $\alpha(\texttt{x}) \in \gamma(\texttt{c})$
- `:not c(x)` *genau dann, wenn* $\alpha(\texttt{x}) \in A \backslash \gamma(\texttt{c})$
- `:nonexistent(x)` *genau dann, wenn* $\alpha(\texttt{x}) \in \Gamma$
- `:is-defined(o.p)` *genau dann, wenn* $\beta_{\texttt{p}}(\alpha(\texttt{o})) \neq$`NIL`
- `:undefined(o.p)` *genau dann, wenn* $\beta_{\texttt{p}}(\alpha(\texttt{o})) =$ `NIL`
- `:all(o.p,c)` *genau dann, wenn* $\beta_{\texttt{p}}(\alpha(\texttt{o})) =$ `NIL` *oder* $\beta_{\texttt{p}}(\alpha(\texttt{o})) \subseteq \gamma(\texttt{c})$ *oder* $\beta_{\texttt{p}}(\alpha(\texttt{o})) \in \gamma(\texttt{c})$
- `:all(o.p, :not c)` *genau dann, wenn* $\beta_{\texttt{p}}(\alpha(\texttt{o})) =$`NIL` *oder* $\beta_{\texttt{p}}(\alpha(\texttt{o}) \subseteq A \backslash \gamma(\texttt{c})$ *oder* $\beta_{\texttt{p}}(\alpha(\texttt{o})) \in A \backslash \gamma(\texttt{c})$
- `:eq(x,y)` *genau dann, wenn* $\alpha(\texttt{x}) = \alpha(\texttt{y})$
- `:eq(o.p,x)` *genau dann, wenn* $\beta_{\texttt{p}}(\alpha(\texttt{o})) = \alpha(\texttt{x})$
- `:some-eq(o.p,x)` *genau dann, wenn* $\alpha(\texttt{x}) \in \beta_{\texttt{p}}(\alpha(\texttt{o}))$

Zusätzliche Randbedingungen, welche sich aus der Semantik von VEDA [MBG+97, Bau97] *ergeben, sind:*

- *Wenn* `:dom d` $\in \delta($`c,p`$)$, *dann bildet* $\beta_{\texttt{p}}$ *Elemente aus* $\gamma($`c`$)$ *immer auf* $\gamma($`d`$) \cup \{$`NIL`$\}$ *ab*
- *Wenn* `:dom #n{d}` $\in \delta($`c,p`$)$, *dann bildet* $\beta_{\texttt{p}}$ *Elemente aus* $\gamma($`c`$)$ *immer auf eine Teilmenge von* $\gamma($`d`$)$ *mit genau* `n` *Elementen oder auf* `NIL` *ab*
- *Wenn* `:dom #n..m{d}` $\in \delta($`c,p`$)$, *dann bildet* $\beta_{\texttt{p}}$ *Elemente aus* $\gamma($`c`$)$ *immer auf eine Teilmenge von* $\gamma($`d`$)$ *mit einer Kardinalität zwischen* `n` *und* `m` *oder auf* `NIL` *ab*
- *Wenn* `:dom #>={d}` $\in \delta($`c,p`$)$, *dann bildet* $\beta_{\texttt{p}}$ *Elemente aus* $\gamma($`c`$)$ *immer auf eine Teilmenge von* $\gamma($`d`$)$ *mit* `n` *oder mehr Elementen oder auf* `NIL` *ab*
- *Wenn* `:dom #<={d}` $\in \delta($`c,p`$)$, *dann bildet* $\beta_{\texttt{p}}$ *Elemente aus* $\gamma($`c`$)$ *immer auf eine Teilmenge von* $\gamma($`d`$)$ *mit* `n` *oder weniger Elementen oder auf* `NIL` *ab*
- *Wenn* `:dom /d/` $\in \delta($`c,p`$)$, *dann bildet* $\beta_{\texttt{p}}$ *Elemente aus* $\gamma($`c`$)$ *immer auf eine Teilmenge von* $\gamma($`d`$)$ *oder auf* `NIL` *ab*
- *Wenn* `:exc t` $\in \delta($`c,p`$)$ *und* `:comp t` $\in \delta($`c',p'`$)$, *dann gilt für alle* $a \in \gamma($`c`$)$, $a' \in \gamma($`c'`$)$: *wenn* $\beta_{\texttt{p}}(a) = \beta_{\texttt{p}'}(a')$, *dann gilt* $a = a'$
- *Wenn* `:req t` $\in \delta($`c,p`$)$, *dann bildet* $\beta_{\texttt{p}}$ *Elemente aus* $\gamma($`c`$)$ *immer auf eine Teilmenge von* A *oder auf Elemente aus* A *ab*

- *Wenn* `:inv p, :dom d` $\in \delta($`c,p`$)$ *und* `:inv p', :dom d'` $\in \delta($`c',p'`$)$, *dann gilt für alle* $a \in \gamma($`c`$)$, $a' \in \gamma($`c'`$)$: $\beta_{\mathtt{p}}(a) = a'$ *genau dann, wenn* $\beta_{\mathtt{p}'}(a') = a$.

- *Wenn* `:inv p'`$\in \delta($`c,p`$)$, *dann gilt für alle* $a \in \gamma($`c`$)$: *wenn* $\beta_{\mathtt{p}}(a) = a'$ *oder* $a' \in \beta_{\mathtt{p}}(a)$, *dann gilt* $a = \beta_{\mathtt{p}'}(a')$ *oder* $a \in \beta_{\mathtt{p}'}(a')$.

Seien $C_1, \ldots, C_n$ *Bedingungen. Eine Interpretation* $\mathcal{I}$ *erfüllt die zusammengesetzte Bedingung*

- *(*C_1 `:and` ... `:and` C_n*) genau dann, wenn* $\mathcal{I}$ *alle* C_i, $1 \leq i \leq n$ *erfüllt.*

- *(*C_1 `:or` ... `:or` C_n*) genau dann, wenn* $\mathcal{I}$ *eines der* C_i, $1 \leq i \leq n$ *erfüllt.*

Definition 7.4 (Semantik der Formeln) *Seien* $C_1, \ldots, C_n$ *Formeln. Eine Interpretation* $\mathcal{I}$ *erfüllt die Formel*

- $\neg$ `(C)` *genau dann, wenn* $\mathcal{I}$ `C` *nicht erfüllt*

- C_1 `:and` ... `:and` C_n *genau dann, wenn* $\mathcal{I}$ *alle* C_i, $1 \leq i \leq n$ *erfüllt.*

- C_1 `:or` ... `:or` C_n *genau dann, wenn* $\mathcal{I}$ *eines der* C_i, $1 \leq i \leq n$ *erfüllt.*

- *eine Formel, die eine Bedingung ist, genau dann, wenn* $\mathcal{I}$ *die Bedingung erfüllt.*

$\mathcal{I}$ *erfüllt eine Menge von Formeln, wenn* $\mathcal{I}$ *alle Elemente der Menge erfüllt*[6].

Es ist nun möglich, den Begriff der Verfeinerung zu definieren. Die Entscheidung, ob eine Bedingung eine Verfeinerung einer anderen Bedingung ist, kann mit Hilfe von Algorithmus 7.1 getroffen werden.

Definition 7.5 (Verfeinerung) *Eine Bedingung* D *heißt eine* Verfeinerung *einer Bedingung* C, *wenn für alle Interpretationen* $\mathcal{I} = (\Delta^{\mathcal{I}}, \alpha, \beta, \gamma)$ *gilt, daß wenn* $\mathcal{I}$ D *erfüllt, so erfüllt* $\mathcal{I}$ *auch* C.

7.5.5 Beispiele

In diesem Abschnitt sollen Syntax und Semantik der Vor- und Nachbedingungen anhand von Beispielen verdeutlicht werden. Dort, wo Pfade angegeben werden müssen, ist die Klasse, in der die Bedingung steht, relevant. Die hier angegebenen Preconditions stehen jeweils in der `INPUT-SITUATION` und die Postconditions jeweils in den `OUTPUT-SITUATION`s. Weiterhin gehen die folgenden Beispiele davon aus, daß in der Klasse `CASE` ein Slot `components` existiert, in den (z. B. über eine Methode) alle Bausteine, welche Bestandteil des Cases sind, eingetragen werden.

- *es gibt einen aktiven Case*

  Precondition: `:is-defined(this.active-case)`

[6] In dieser Arbeit werden nur endliche Mengen von Formeln und damit auch nur endliche Formeln betrachtet

- *es gibt im aktiven Case ein* `COMPOSITE-DEVICE-MODEL`

  Precondition: `(:some-eq(this.active-case.components,X) :and COMPOSITE-DEVICE-MODEL(X))`

- *es gibt ein* `DEVICE-MODEL` *im aktiven Case, in welchem vor der Durchführung des Schritts der Slot* `implementation` *undefiniert und nach der Durchführung definiert ist*

  Precondition: `(DEVICE-MODEL(X) :and :some-eq (this.active-case.components, X) :and :undefined(X.implementation))`
  Postcondition: `(DEVICE-MODEL(X) :and :some-eq (this.occurs-in.active-case.components, X) :and :is-defined(X.implementation))`

- *es wird im aktiven Case ein neues* `DEVICE-MODEL` *erzeugt*

  Precondition: `:nonexistent(X)`
  Postcondition: `(DEVICE-MODEL(X) :and :some-eq (this.occurs-in.is-implementing. input-situation.active-case.components, X))`

- *für eine* `FILM-IMPLEMENTATION` *wird entschieden, ob es sich um eine* `1-, 2-` *oder* `3-FILM-IMPLEMENTATION` *handelt*

  Precondition: `FILM-IMPLEMENTATION(X)`
  Postcondition: `(1-FILM-IMPLEMENTATION(X) :or 2-FILM-IMPLEMENTATION(X) :or 3-FILM-IMPLEMENTATION(X))`

- *es werden ein* `MODEL` *und eine* `MATERIAL-ENTITY` *erzeugt, die über die Slots* `describes` *beziehungsweise* `is-modeled-as` *aufeinander verweisen*

  Precondition: `(:nonexistent(model) :and :nonexistent(mat-entity))`
  Postcondition: `(MODEL(model) :and MATERIAL-ENTITY(mat-entity) :and :eq(model.describes, mat-entity) :and :eq(mat-entity.is-modeled-as, model))`

- *Das Gleichungssystem, welches das Verhalten der Komponente* `K` *beschreibt, hat einen Freiheitsgrad größer Null*[7]

  Precondition: `:eq(K.behavior.degree-of-freedom,_R_dof) :and >0(_R_dof)`

7.6 Algorithmus zur Entscheidung der Implikation von Vor- und Nachbedingungen

Der im folgenden vorgestellte Algorithmus entscheidet die Frage, ob für gegebene Bedingungen C und D für alle Interpretationen, welche D erfüllen, auch gilt, daß diese C erfüllen.

[7] Dieses Beispiel verwendet den Vergleich eines Slotinhalts mit Konstanten. Diese sind eine Erweiterungen der Definition der Vor- und Nachbedingungen, auf sie wird in Abschnitt 7.7 eingegangen.

Da C und D aus verschiedenen Ablaufschritten stammen und deshalb gleiche Variablen enthalten können, die nicht das gleiche Objekt bezeichnen sollen, muß zunächst dafür gesorgt werden, daß solche Konflikte behoben werden.

Zur Veranschaulichung dieser Problematik kann der der einfache Ablauf aus Abbildung 16 betrachtet werden. Schritt 2 kann in diesem Ablauf beliebig oft wiederholt werden, genauer gesagt können verschiedene Instanzen des auf Klassenebene definierten Schritts hintereinander ausgeführt werden. Jede dieser Instanzen erzeugt während ihrer Durchführung ein neues `DEVICE-MODEL`. Diese neu erzeugten Objekte werden innerhalb der Vor- und Nachbedingungen jeder Instanz von Schritt 2 mit `X` bezeichnet, sind aber doch verschieden. Die Annahme "gleicher Bezeichner, gleiches Objekt" gilt nur innerhalb eines Schritts.

In einem System, welches die Durchführung von Ablaufschritten unterstützt und daher darauf angewiesen ist, zwischen den verschiedenen mit `X` bezeichneten Objekten zu unterscheiden, kann das Problem der Namensgleichheit zum Beispiel dadurch gelöst werden, daß an die Bezeichner jeweils ein Zeitstempel angehängt wird, der angibt, wann der zugehörige Schritt instantiiert wurde.

Schritt 3 hat die Existenz eines `DEVICE-MODEL` als Vorbedingung. Diese Bedingung kann nicht von dem durch Schritt 1 erzeugten Objekt erfüllt werden, sondern nur von einem der durch Schritt 2 erzeugten Objekte. Hier ist es also notwendig, die Variable `Y` aus den Bedingungen von Schritt 3 mit der Variablen `X` einer der Instanzen von Schritt 2 zu identifizieren und gleichzeitig die in den Schritten 1 und 3 mit `Y` bezeichneten Objekte zu separieren.

Es wird eine Substitution benötigt, welche Variablen in C, die auch in D vorkommen, durch Variablen ersetzen kann, die zuvor nicht in D vorkamen und Variablen, die nur in C vorkommen, durch Variablen ersetzen kann, die auch in D vorkommen.

Die Möglichkeit, solche Ersetzungen vorzunehmen, wird durch die folgende Definition geschaffen. Da die Bedingungen keine Quantoren enthalten[8] und es erlaubt ist, verschiedene Variablen mit dem gleichen Objekt zu identifizieren, wird keine simultane Substitution aller Variablen, wie sie zum Beispiel in [EFT92] für Formeln der Logik erster Stufe definiert wird, benötigt.

Definition 7.6 (Substitution) *Seien C und D Formeln entsprechend Definition 7.2, v eine in C vorkommende Variable, v' eine in C oder D vorkommende Variable und w eine beliebige Variable. Dann sei $C\frac{w}{v}$ diejenige Formel, die man erhält, wenn man in C gleichzeitig jedes Vorkommen von v durch w ersetzt.*

Für Folgen $V = (v_1, \ldots, v_n)$ und $W = (w_1, \ldots, w_n)$ von Variablen sei $C\frac{W}{V}$ definiert als $\left(\left(\ldots\left(C\frac{w_1}{v_1}\right)\ldots\right)\frac{w_n}{v_n}\right)$.

Als Substitution *wird im folgenden eine Abbildung π, welche eine Bedingung C auf $C\frac{W}{V}$ abbildet, bezeichnet. Dabei sind V und W durch π festgelegte Folgen von Variablen.*

Es kann nun definiert werden, wann eine Bedingung als spezieller als eine andere angesehen wird. Der Unterschied zur Definition der Verfeinerung (Definition 7.5) liegt darin, daß im Fall der Spezialisierung auch die Umbenennung von Variablen erlaubt ist, während bei der Verfeinerung die Implikation ohne Variablenumbenennung gelten

[8] Genau genommen sind alle Variablen, welche in Vor- und Nachbedingungen vorkommen, existentiell quantifiziert, ohne daß dies explizit in den Bedingungen angegeben wird

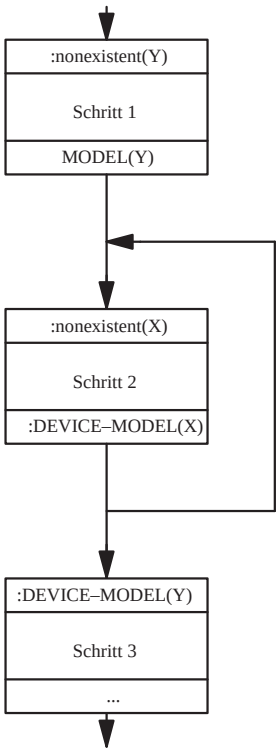

Abbildung 16: Ein einfacher Ablauf zur Veranschaulichung der Notwendigkeit der Variablensubstitution

muß. Die Verfeinerung ist dementsprechend eine restriktivere Relation als die Spezialisierung. Dies wird anhand von Beispiel 7.3 noch weiter erläutert.

Definition 7.7 (Spezialisierung) *Eine Bedingung D heißt* spezieller *als eine Bedingung C, wenn es eine Substitution π gibt, so daß für alle Interpretationen $\mathcal{I} = (\Delta^{\mathcal{I}}, \alpha, \beta, \gamma)$ gilt, daß wenn $\mathcal{I}$ D erfüllt, so erfüllt $\mathcal{I}$ auch $\pi(C)$.*

Um festzustellen, ob eine Bedingung D spezieller ist als eine Bedingung C, muß, entsprechend Definition 7.6, eine Substitution π gefunden werden, so daß jede Interpretation $\mathcal{I}$ $\pi(C)$ immer erfüllt, wenn $\mathcal{I}$ D erfüllt.

Die folgende Argumentation zeigt, daß zur Überprüfung der Spezialisierungsbeziehung nur endlich viele Substitutionen betrachtet werden müssen.

Sei V_C die Menge der in C und V_D die Menge der in D vorkommenden Variablen, dann reicht es aus, alle Substitutionen aller oder einiger Variablen aus C durch Teilmengen von $V_{neu} \cup V_D$ zu betrachten, wobei V_{neu} eine Menge von Variablen ist, welche weder in C noch in D vorkommen und für die $|V_{neu}| = |V_C \cap V_D|$ gilt. Durch die Kardinalität von V_{neu} wird sichergestellt, daß jede Variable aus C, welche auch in D auftritt, durch eine Variable, die nicht in D auftritt, ersetzt werden kann.

Beispiel 7.3 (Verfeinerung und Spezialisierung) *Ein* `CONTEXT` *enthalte die folgende Vereinbarung:*

conditions:

device-model:	`DEVICE-MODEL(X) :to_group postconditions`

elementary-impl:	`:all(X.implementation,`
	`ELEMENTARY-DEVICE-IMPLEMENTATION)`
	`:to_group postconditions :refinable t`

und in einer seiner Subklassen sei folgende Definition zu finden:

conditions:

elementary-impl:	`:all(Y.implementation,`
	`ABSTRACT-PHASE-IMPLEMENTATION)`
	`:to_group postconditions :refinable t`

Dann ist die Bedingung `elementary-impl` *der Subklasse zwar eine Spezialisierung der gleichnamigen Bedingung der Superklasse, da jede* `ABSTRACT-PHASE-IMPLEMENTATION` *auch eine* `ELEMENTARY-DEVICE-IMPLEMENTATION` *ist (vergleiche Abbildung 17), aber sie ist keine Verfeinerung, da für die Verfeinerungsbeziehung die Variable* `Y` *durch* `X` *substituiert werden müsste.*

Die ursprünglichen Nachbedingungen (nämlich die der Superklasse) müssen nach der Durchführung einer Instanz der Subklasse nicht mehr unbedingt erfüllt sein, da `X` *und* `Y` *mit verschiedenen Objekten identifiziert werden können und deshalb nicht mehr garantiert ist, daß* `X` *eine* `ELEMENTARY-DEVICE-IMPLEMENTATION` *besitzt.*

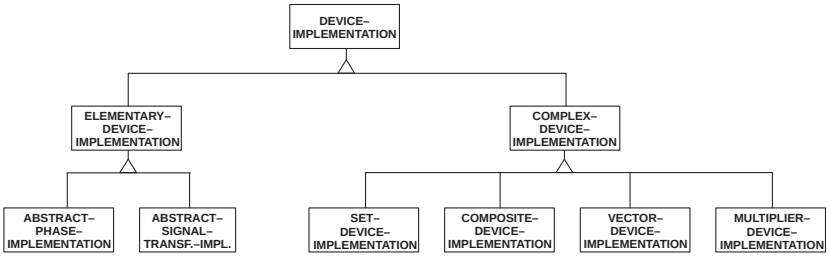

Abbildung 17: Übersicht über die Klasse `DEVICE-IMPLEMENTATION` und ihre Subklassen

Die Überprüfung der Allgemeingültigkeit einer Formel der Form $D \Rightarrow C$, das heißt die Entscheidung darüber, ob Bedingung D eine Verfeinerung von Bedingung C ist, läßt sich auf die Überprüfung der Erfüllbarkeit der Formel $\neg C \wedge D$ reduzieren. Eine solche Reduktion ist sinnvoll, da sich die Erfüllbarkeit einer Formel im allgemeinen leichter zeigen läßt als die Allgemeingültigkeit.

Lemma 7.1 *Für beliebige Bedingungen C und D gilt: $D \Rightarrow C$ für alle Interpretationen genau dann, wenn $\neg C \wedge D$ unerfüllbar ist, das heißt wenn es keine Interpretation $\mathcal{I}$ gibt, welche $\neg C \wedge D$ erfüllt.*

Begründung
$D \Rightarrow C$ gilt genau dann für alle Interpretationen, wenn die äquivalente Formel $C \vee$

$\neg D$ für alle Interpretationen gilt. Dies ist gleichbedeutend mit der Unerfüllbarkeit von $\neg(C \vee \neg D)$. Diese Formel ist nach DeMorgan äquivalent zu $\neg C \wedge D$.

Im folgenden wird ein Algorithmus entwickelt, welcher die Erfüllbarkeit einer Formel der Form $\neg C \wedge D$ entscheidet. Dazu werden alle implizit in einer Formel enthaltenen Aussagen explizit gemacht – zum Beispiel folgt aus `:eq(x.p,y)`, daß `x` ein Objekt ist und somit zumindest `ROOT(x)` gilt – bis entweder ein offensichtlicher Widerspruch auftritt oder ein widerspruchsfreies, vollständiges System von Formeln entstanden ist. Vollständig heißt, daß sämtliche implizit enthaltenen Aussagen explizit gemacht wurden. Da im Fall von Disjunktionen nur eines der Disjunkte erfüllt sein muß, damit die Formel erfüllbar ist, entstehen in solchen Fällen verschiedene mögliche Systeme von Formeln. Es reicht aus, wenn eines dieser Systeme zu einem widerspruchsfreien und vollständigen System ergänzt werden kann.

Zunächst werden wir diejenigen Mengen von VEDA-Klassen definieren, so daß kein Objekt gleichzeitig Instanz jeder Klasse aus einer Menge von unverträglichen Klassen sein kann.

Definition 7.8 (unverträglich) *Eine Menge von* VEDA*-Klassen heißt* unverträglich *genau dann, wenn sie keine gemeinsame, nicht abstrakte*[9] *Subklasse*[10] *besitzen oder verschiedene Subklassen einer Klasse sind, welche der Metaklasse* `disjunct` *zugeordnet ist.*

Definition 7.9 listet alle Formelmengen auf, welche offensichtliche Widersprüche enthalten.

Definition 7.9 (inkompatibel) *Seien* $\mathtt{c}, \mathtt{d} \in C$, $\mathtt{k} \in K$, $\mathtt{k}_R, \mathtt{l}_R \in K_R$, $\mathtt{k}_B, \mathtt{l}_B \in K_B$, $\mathtt{z} \in K \cup X$, $\mathtt{x}, \mathtt{y} \in X$ *und* $\mathtt{p}=\mathtt{p}_1.\ldots.\mathtt{p}_n$, $n \geq 1$ *eine Folge von* VEDA*-Slotbezeichnern. Folgende Formeln heißen jeweils zueinander* inkompatibel*:*

1. $\mathtt{c}_1(\mathtt{x}), \ldots, \mathtt{c}_n(\mathtt{x})$, $n \geq 2$, *wenn* $\mathtt{c}_1, \ldots, \mathtt{c}_n$ *unverträglich ist.*
2. `:nonexistent(x), c(x)`
3. $\mathtt{:undefined(x.p}_1.\ldots.\mathtt{p}_n\mathtt{)}, \mathtt{:eq(x.p}_1\mathtt{,y}_1\mathtt{)}, \mathtt{:eq(y}_1\mathtt{.p}_2.\ldots.\mathtt{p}_n\mathtt{)}, \ldots, \mathtt{:eq(y}_{n-2}\mathtt{.p}_{n-1}\mathtt{,y}_{n-1}\mathtt{)}, \mathtt{:eq(y}_{n-1}\mathtt{.p}_n\mathtt{,y}_n\mathtt{)}.$
4. $\mathtt{:undefined(x.p}_1.\ldots.\mathtt{p}_n\mathtt{)}, \mathtt{:eq(x.p}_1\mathtt{,y}_1\mathtt{)}, \mathtt{:eq(y}_1\mathtt{.p}_2.\ldots.\mathtt{p}_n\mathtt{)}, \ldots, \mathtt{:eq(y}_{n-2}\mathtt{.p}_{n-1}\mathtt{,y}_{n-1}\mathtt{)}, \mathtt{:some\text{-}eq(y}_{n-1}\mathtt{.p}_n\mathtt{,y}_n\mathtt{)}.$
5. $\mathtt{c(x)}, \mathtt{:eq(x.p}_1\mathtt{,y)}$, *wenn* $\mathtt{p}_1$ *in keiner Subklasse von* `c` *einen Slot mit einem objektwertigen Wertebereich bezeichnet.*
6. $\mathtt{c(x)}, \mathtt{:some\text{-}eq(x.p}_1\mathtt{,y)}$, *wenn* $\mathtt{p}_1$ *in keiner Subklasse von* `c` *einen Slot mit einem mengen- oder listenwertigen Wertebereich bezeichnet.*
7. $\neg C, C$, *für jede beliebige Formel* C

Definition 7.10 (clash) *Sei* S *eine Menge von Formeln.* S *enthält einen* clash *genau dann, wenn* S *inkompatible Formeln enthält.*

[9] Abstrakte Klassen können nicht instantiiert werden, vergleiche Abschnitt 4.3.

[10] Im folgenden wird der reflexive Abschluß der VEDA-Subklassenrelation als Subklassen-Begriff verwendet.

Die folgende Definition erleichtert die algorithmische Handhabung von Formeln. Anschaulich werden alle Negationszeichen, die in der Formel auftreten, soweit wie möglich nach innen gezogen, so daß nur noch atomare Bedingungen in negierter Form auftreten.

Definition 7.11 (Negationsnormalform) *Eine Formel ϕ ist in* Negationsnormalform*, wenn ϕ nur dann Teilformeln der Form $\neg C$ enthält, wenn C eine atomare Bedingung ist.*

Im folgenden wird anstelle von `:and` auch das Symbol $\wedge$ und anstelle von `:or` das Symbol $\vee$ verwendet.

Lemma 7.2 (Existenz der Negationsnormalform) *Zu einer Formel $\phi = \neg C \wedge D$, mit C und D Bedingungen, existiert immer eine äquivalente Formel in Negationsnormalform.*

Beweis

C ist eine Bedingung, daß heißt C ist aufgebaut aus atomaren Bedingungen, welche durch `:and` und `:or` verknüpft sind. Diese entsprechen den logischen Operatoren $\wedge$ beziehungsweise $\vee$.

Ist C eine atomare Bedingung, so liegt ϕ bereits in Negationsnormalform vor.

Anderenfalls ist C entweder von der Form $(C = C_1$ `:and` $\ldots$ `:and` $C_n)$ oder von der Form $C = (C_1$ `:or` $\ldots$ `:or` $C_n)$ für beliebige Bedingungen $C_i,\ 1 \leq i \leq n$.

In diesen Fällen kann unter Ausnutzung der Äquivalenzen

$$\begin{array}{lcl} \neg(C_1 \texttt{ :and } \ldots \texttt{ :and } C_n) & \Leftrightarrow & \neg C_1 \texttt{ :or } \ldots \texttt{ :or } \neg C_n \\ \neg(C_1 \texttt{ :or } \ldots \texttt{ :or } C_n) & \Leftrightarrow & \neg C_1 \texttt{ :and } \ldots \texttt{ :and } \neg C_n \end{array}$$

das Negationszeichen "nach innen gezogen" werden, bis eine Formel C' entstanden ist, die äquivalent zu $\neg C$ ist und in Negationsnormalform vorliegt. Damit ist auch $C' \wedge D$ in Negationsnormalform. □

Nun kann ein Algorithmus angegeben werden, welcher die Frage entscheidet, ob eine gegebene Formel erfüllbar ist. Wenn für eine Formel der Form $D \Rightarrow C$ gezeigt werden soll, daß sie für alle Interpretationen gilt, kann dies (entsprechend der Aussage von Lemma 7.1) dadurch geschehen, daß die Unerfüllbarkeit von $\neg C \wedge D$ gezeigt wird. Diese Aufgabe erfüllt der folgende Algorithmus. Wenn er mit dem Ergebnis "unerfüllbar" terminiert, bedeutet dies, daß Bedingung D spezieller ist als Bedingung C.

Algorithmus 7.1 (Entscheidung der Erfüllbarkeit) Dieser Algorithmus erzeugt einen Baum, in welchem jeder Knoten mit einer Menge von Formeln beschriftet ist. Der Algorithmus startet mit einem Baum, welcher nur aus der Wurzel besteht, die mit $S = \{\phi\}$ beschriftet ist, wobei ϕ diejenige Formel ist, von welcher entschieden werden soll, ob sie erfüllbar ist. ϕ sei in Negationsnormalform.

Eine Vervollständigungsregel kann nur auf ein Blatt angewendet werden, welches mit einer clash-freien Menge von Formeln beschriftet ist. Die Anwendung einer Regel $S \rightarrow S_i,\ 1 \leq i \leq n$ auf ein solches Blatt führt zur Erzeugung n neuer Nachfolger des Knotens, wovon jeder mit einer der Mengen S_i beschriftet ist.

Der Algorithmus terminiert, wenn keine der Regeln mehr auf irgendein Blatt anwendbar ist. In diesem Fall liefert er als Ergebnis "ϕ erfüllbar", wenn mindestens ein Blatt mit einer clash-freien Menge von Formeln beschriftet ist, sonst "ϕ unerfüllbar".

Liste der Vervollständigungsregeln:

Im folgenden bezeichnen C und D jeweils Formeln, `x,y,z` $\in X$, `c,d` VEDA-Klassen, $\texttt{p}_1,\ldots,\texttt{p}_n$ VEDA-Slotbezeichner und $\texttt{p=p}_1,\ldots,\texttt{p}_n \in P$.

1. Wenn $C \wedge D \in S$ und $C \notin S$ oder $D \notin S$, dann

   $$S \rightarrow S \cup \{C, D\}$$

2. Wenn $C \vee D \in S$, $C \notin S$ und $D \notin S$, dann

   $$S \rightarrow S_1 = S \cup \{C\}$$
   $$S \rightarrow S_2 = S \cup \{D\}$$

3. Wenn `:is-defined(x.p,y)` $\in S$ es gibt kein `z`$\in X$ mit `:eq(x.p,z)` $\in S$ oder `:some-eq(x.p,z)` $\in S$, dann

   $$S \rightarrow S_1 = S \cup \{\texttt{:eq(x.p,z)}\}$$
   $$S \rightarrow S_2 = S \cup \{\texttt{:some-eq(x.p,z)}\}$$

   für ein `z`$\in V$, welches bisher noch in keiner Formel aus S vorkommt.

4. Wenn $\neg$`:nonexistent(x)` $\in S$ und `ROOT(x)` $\notin S$, dann

   $$S \rightarrow S \cup \{\texttt{ROOT(x)}\}$$

5. Wenn $\neg$`c(x)` $\in S$ und weder `:not c(x)`$\in S$ noch `:nonexistent(x)` $\in S$, dann

   $$S \rightarrow S_1 = S \cup \{\texttt{:not c(x)}\}$$
   $$S \rightarrow S_2 = S \cup \{\texttt{:nonexistent(x)}\}$$

6. Wenn $\neg$`:not c(x)` $\in S$ und weder `c(x)`$\in S$ noch `:nonexistent(x)` $\in S$, dann

   $$S \rightarrow S_1 = S \cup \{\texttt{c(x)}\}$$
   $$S \rightarrow S_2 = S \cup \{\texttt{:nonexistent(x)}\}$$

7. Wenn $\neg$`:undefined(x.p)` $\in S$ und `:is-defined(x.p)` $\notin S$, dann

   $$S \rightarrow S \cup \{\texttt{:is-defined(x.p)}\}$$

8. Wenn $\neg$`:is-defined(x.p)` $\in S$ und `:undefined(x.p)` $\notin S$, dann

   $$S \rightarrow S \cup \{\texttt{:undefined(x.p)}\}$$

9. Wenn $\neg$`:all(x.p,c)` $\in S$, weder `:eq(x.p,y)`$\in S$ noch `:some-eq(x.p,y)`$\in S$ für ein `y`$\in V$ mit `:not c(y)`$\in S$, dann

   $$S \rightarrow S_1 = S \cup \{\texttt{:eq(x.p,y)},\texttt{:not c(y)}\}$$
   $$S \rightarrow S_2 = S \cup \{\texttt{:some-eq(x.p,y)},\texttt{:not c(y)}\}$$

   für ein beliebiges `y`$\in V$, welches bisher noch in keiner Formel aus S vorkommt.

10. Wenn $\neg$`:all(x.p,:not c)` $\in S$, weder `:eq(x.p,y)`$\in S$ noch `:some-eq(x.p,y)`$\in S$ für ein `y`$\in V$ mit `c(y)`$\in S$, dann

$$S \to S_1 = S \cup \{\texttt{:eq(x.p,y), c(y)}\}$$
$$S \to S_2 = S \cup \{\texttt{:some-eq(x.p,y), c(y)}\}$$

für ein beliebiges `y`$\in V$, welches bisher noch in keiner Formel aus S vorkommt.

11. Wenn $\neg$`:some-eq(x.`p_1`....`p_n`,y)` $\in S$, $n \geq 2$ und weder `:undefined(x.`p_1`....`p_n`)` noch `:eq(x.`p_1`,z)` und $\neg$`:some-eq(z.`p_2`....`p_n`,y)` $\in S$ für ein beliebiges `z`$\in X$, dann

$$S \to S_1 = S \cup \{\texttt{ :eq(x.}p_1\texttt{,z)}, \neg\texttt{:some-eq(z.}p_2\texttt{....}p_n\texttt{,y)}\}$$
$$S \to S_2 = S \cup \{\texttt{:undefined(x.}p_1\texttt{...}p_n\texttt{)}\}$$

für ein beliebiges `z`$\in V$, welches bisher noch in keiner Formel aus S vorkommt.

12. Wenn $\neg$`:eq(x.`p_1`....`p_n`,y)` $\in S$, $n \geq 2$ und weder `:undefined(x.`p_1`....`p_n`)` noch `:eq(x.`p_1`,z)` und $\neg$`:eq(z.`p_2`....`p_n`,y)` $\in S$ für ein beliebiges `z`$\in X$, dann

$$S \to S_1 = S \cup \{\texttt{ :eq(x.}p_1\texttt{,z)}, \neg\texttt{:eq(z.}p_2\texttt{....}p_n\texttt{,y)}\}$$
$$S \to S_2 = S \cup \{\texttt{:undefined(x.}p_1\texttt{...}p_n\texttt{)}\}$$

für ein beliebiges `z`$\in V$, welches bisher noch in keiner Formel aus S vorkommt.

13. Wenn `:all(x.`p_1`.....`p_n`,c)`$\in S$, `:eq(x.`p_1`,y)`$\in S$ und `:all(y.`p_2`.....`p_n`,c)`$\notin S$, dann

$$S \to S \cup \{\texttt{:all(y.}p_2\texttt{.....}p_n\texttt{,c)}\}$$

14. Wenn `:all(x.`p_1`.....`p_n`,:not c)`$\in S$, `:eq(x.`p_1`,y)`$\in S$ und `:all(y.`p_2`.....`p_n`, c)`$\notin S$, dann

$$S \to S \cup \{\texttt{:all(y.}p_2\texttt{.....}p_n\texttt{, :not c)}\}$$

15. Wenn `:all(x.p, c)`$\in S$ oder `:all(x.p, :not c)`$\in S$ und weder `:is-defined (x.p)` noch `:undefined(x.p)` liegt in S, dann

$$S \to S_1 = S \cup \{\texttt{ :is-defined(x.p)}\}$$
$$S \to S_2 = S \cup \{\texttt{ :undefined(x.p)}\}$$

16. Wenn `:eq(x.`p_1`.....`p_n`,y)` $\in S$, $n \geq 2$ und es gibt kein `z`$\in X$ mit `:eq(x.`p_1`,z)` $\in S$, dann

$$S \to S \cup \{\texttt{:eq(x.}p_1\texttt{,z), :eq(z.}p_2\texttt{.....}p_n\texttt{,y)}\}$$

für ein `z`$\in V$, welches bisher noch in keiner Formel aus S vorkommt.

17. Wenn `:some-eq(x.`p_1`.....`p_n`,y)` $\in S$, $n \geq 2$ und es gibt kein `z`$\in X$ mit `:eq(x.`p_1`,z)` $\in S$, dann

$$S \to S \cup \{\texttt{:eq(x.}p_1\texttt{,z), :some-eq(z.}p_2\texttt{.....}p_n\texttt{,y)}\}$$

für ein `z`$\in V$, welches bisher noch in keiner Formel aus S vorkommt.

18. Wenn `:eq(x,y)` $\in S$ und es gibt Formeln in S, welche `y` enthalten:

$$S \rightarrow \{\phi\frac{x}{y} \mid \phi \in S\}$$

19. Wenn `c(x)` $\in S$ und für keine nicht abstrakte Subklasse `c'` von `c` gilt `c'(x)` $\in S$, dann

$$S \rightarrow S_i = S \cup \{\ \texttt{c}_i\texttt{(x)}\ \}$$

für alle nicht abstrakten Subklassen $\texttt{c}_i$ von `c`

20. Wenn `:not c(x)` $\in S$ und für keine nicht abstrakte Klasse `c'`, die nicht Subklasse von `c` ist, gilt `c'(x)` $\in S$, dann

$$S \rightarrow S_i = S \cup \{\ \texttt{c}_i\texttt{(x)}\ \}$$

für alle nicht abstrakten Klassen $\texttt{c}_i$, die nicht Subklasse von `c` sind

21. Wenn `c'(x)` $\in S$ und es gilt nicht für alle Superklassen `c` von `c'` `c(x)` $\in S$, dann

$$S \rightarrow S \cup \{\ \texttt{c(x)} \mid \texttt{c} \text{ Superklasse von } \texttt{c'}\ \}$$

22. Wenn `c(x)`$\in S$ und `:eq(x.`$\texttt{p}_1$`,y)` $\in S$, der in der Klasse `c` durch $\texttt{p}_1$ bezeichnete Slot hat einen objektwertigen Wertebereich mit Typ `d` und `d(y)` $\notin S$, dann

$$S \rightarrow S \cup \{\ \texttt{d(y)}\}$$

23. Wenn `c(x)`$\in S$ und `:some-eq(x.`$\texttt{p}_1$`,y)` $\in S$, der in der Klasse `c` durch $\texttt{p}_1$ bezeichnete Slot hat einen mengen- oder listenwertigen Wertebereich mit Objekttyp `d` und `d(y)` $\notin S$, dann

$$S \rightarrow S \cup \{\ \texttt{d(y)}\}$$

24. Wenn `:eq(x.p,y)` $\in S$, `:eq(x.p,z)` $\in S$ und `:eq(y,z)` $\notin S$, dann

$$S \rightarrow S \cup \{\ \texttt{:eq(y,z)}\}$$

25. Wenn : θ`(x.`$\texttt{p}_1$`,y)` $\in S$, θ'`(x'.`$\texttt{p}_1$`',y)` $\in S$, mit $\theta, \theta' \in \{$`:eq, :some-eq`$\}$, `c(x)` $\in S$, `c'(x')` $\in S$, in der Klasse `c` hat der Slot $\texttt{p}_1$ die Facette `:exc t` und in der Klasse `c'` der Slot $\texttt{p}_1$`'` die Facette `:comp t` und `:eq(x,x')` $\notin S$, dann

$$S \rightarrow S \cup \{\ \texttt{:eq(x,x')}\}$$

26. Wenn `c(x)`$\in S$ und `:eq(x.`$\texttt{p}_1$`,y)` oder `:some-eq(x.`$\texttt{p}_1$`,y)` $\in S$, der in der Klasse `c` durch $\texttt{p}_1$ bezeichnete Slot hat die Facette `:inv` $\texttt{p}_1$`'` und weder `:eq(y.`$\texttt{p}_1$`',x)` $\in S$ noch `:some-eq(y.`$\texttt{p}_1$`',x)` $\in S$ dann

$$S \rightarrow S_1 = S \cup \{\ \texttt{:eq(y.}\texttt{p}_1\texttt{',x)}\}$$
$$S \rightarrow S_2 = S \cup \{\ \texttt{:some-eq(y.}\texttt{p}_1\texttt{',x)}\}$$

27. Wenn (`:eq(x.`p_1`,y)`$\in S$ oder `:some-eq(x.`p_1`,y)`$\in S$) und `:all(x.`p_1`,c)` $\in S$ und `c(y)` $\notin S$, dann

$$S \rightarrow S \cup \{\ \texttt{c(y)}\}$$

28. Wenn (`:eq(x.`p_1`,y)`$\in S$ oder `:some-eq(x.`p_1`,y)`$\in S$) und `:all(x.`p_1`, :not c)` $\in S$ und `:not c(y)` $\notin S$, dann

$$S \rightarrow S \cup \{\ \texttt{:not c(y)}\}$$

29. Wenn eine der Formeln

```
c(x),
:is-defined(x.p),
:eq(x.p,y),
:eq(y.p,x),
:some-eq(x.p,y),
:some-eq(y.p,x)
```

mit beliebigem `y` in S liegt und `ROOT(x)` $\notin S$, dann

$$S \rightarrow S \cup \{\ \texttt{ROOT(x)}\ \}$$

30. Wenn eine der Formeln
`:undefined(x.p)` $\in S$ oder
`:eq(x,y)` $\in S$ oder
`:eq(y,x)` $\in S$ oder
$\neg$ `:eq(x,y)` $\in S$ oder
$\neg$ `:eq(y,x)` $\in S$ oder
`:all(x.p,c)` $\in S$ oder
`:all(x.p,:not c)` $\in S$

mit beliebigem `y` in S liegt und weder `ROOT(x)` $\in S$ noch `:nonexistent(x)` $\in S$ dann

$$S \rightarrow S_1 = S \cup \{\ \texttt{ROOT(x)}\ \}$$
$$S \rightarrow S_2 = S \cup \{\ \texttt{:nonexistent(x)}\ \}$$

31. Wenn `:some-eq(x.`p_1`,`y_1`)`$,\ldots,$ `:some-eq(x.p,`y_n`)` $\in S$, Regel 17 ist nicht anwendbar und der durch `x.`p_1 festgelegte Slot hat eine Kardinalitätsbeschränkung von maximal $n-k$ $(k \geq 1)$, dann

$$S \rightarrow S_m = S \cup \{\ \texttt{:eq(y}_i\texttt{,y}_j\texttt{)}\}$$

für alle (i,j) mit $1 \leq i,j \leq n, i \neq j$, $1 \leq m \leq |\{(i,j) \mid i \neq j, 1 \leq i,j \leq n\}|$

32. Wenn `:some-eq(x.`p_1`,`y_1`)`$,\ldots,$ `:some-eq(x.`p_1`,`y_n`)` $\in S$, `n` ≥ 1, es gibt kein weiteres `y` so daß `:some-eq(x.`p_1`,y)`$\in S$, und der durch `x.`p_1 festgelegte Slot hat eine Kardinalitätsbeschränkung von mindestens $n+k$ $(k \geq 1)$, dann

$$S \rightarrow S \cup \{\ \texttt{:some-eq(x.}p_1\texttt{,z)}, \neg\texttt{:eq(z,}y_1\texttt{)}, \ldots, \neg\texttt{:eq(z,}y_n\texttt{)}\ \}$$

für ein beliebiges z, welches bisher in keiner Formel aus S vorkommt.

33. Wenn c(x)$\in S$, der Slot p_1 hat in c die Facette :req t, S enthält eine der Formeln :undefined(x.p_1), :all(x.p_1,d), :all(x.p_1,:not d) für ein beliebiges d$\in C$, und weder :eq(x.p_1,y) noch :some-eq(x.p_1,y)$\in S$ für ein beliebiges y$\in X$, dann

$$S \to S_1 = S \cup \{ \text{:some-eq(x.}p_1\text{,y)}\}$$
$$S \to S_2 = S \cup \{ \text{:eq(x.}p_1\text{,y)}\}$$

für ein beliebiges y, welches bisher in keiner Formel aus S vorkommt.

Ein kleines Beispiel soll die Arbeitsweise dieses Algorithmus veranschaulichen:

Beispiel 7.4 (Arbeitsweise von Algorithmus 7.1)

Um das Beispiel überschaubar zu halten gehen wir von der in Abbildung 18 dargestellten Klassenhierarchie aus und verwenden nicht die VEDA*-Taxonomie als Grundlage dieses Beispiels. Dabei ist* p *ein Slot der Klasse* c *mit* :dom ROOT *und ohne weitere Facetten, die Klassen* c, d *und* d' *sind nicht abstrakt.*

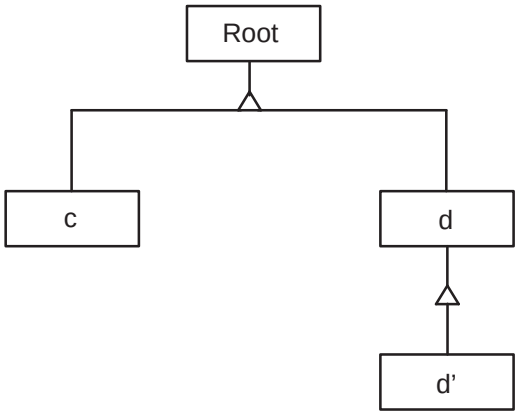

Abbildung 18: Taxonomie für Beispiel 7.4

Gegeben sind die Bedingungen D=c(x) :and :all(x.p,d') :and :eq(x.p,y) *und* C=c(x) :and :eq(x.p,y) :and d(y). *Es soll geprüft werden, ob* $D \Rightarrow C$ *gilt, d.h. ob* D *eine Verfeinerung von* C *ist.*

Dazu wird Algorithmus 7.1 auf die Formel

$$\neg(\neg D \vee C) \quad = \quad \text{c(x)} \wedge \text{:all(x.p,d')} \wedge \text{:eq(x.p,y)} \wedge (\neg\ \text{c(x)} \vee \neg\ \text{:eq(x.p,y)} \vee \neg\ \text{d(y)}).$$

angewandt.

Abbildung 19 zeigt den resultierenden Ableitungsbaum, wobei die zusammengesetzten Formeln, die durch die Ableitung nicht gelöscht werden, nicht an jedem Knoten wie-

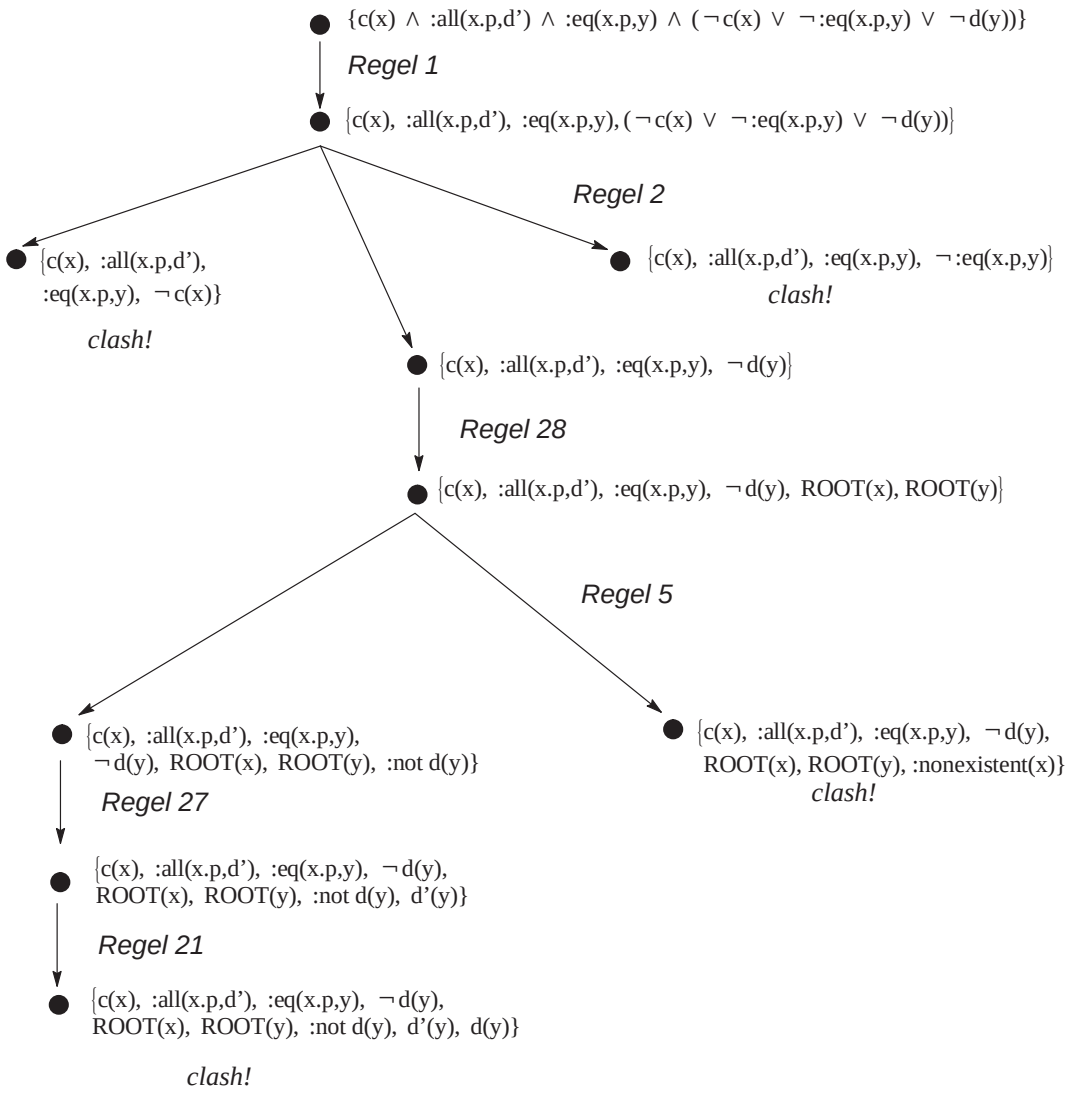

Abbildung 19: Anwendung von Algorithmus 7.1 auf Beispiel 7.4

derholt wurden. Da alle Blätter mit inkompatiblen Formeln beschriftet sind terminiert der Algorithmus mit dem Ergebnis, daß $D \Rightarrow C$ gilt.

Das folgende Lemma zeigt, daß der Algorithmus das gewünschte Ergebnis liefert.

Lemma 7.3 (Korrektheit von Algorithmus 7.1)

(i) Wenn eine Formelmenge einen clash enthält, dann ist sie unerfüllbar.

(ii) Die Vervollständigungsregeln von Algorithmus 7.1 sind lokal korrekt, d.h. daß für jede Vervollständigungsregel $S \rightarrow S_i$ gilt: Wenn eine Interpretation S erfüllt, so erfüllt sie auch eine der Mengen S_i.

(iii) Algorithmus 7.1 terminiert immer.

(iv) Eine Formelmenge, auf welche keine der Vervollständigungsregeln von Algorithmus 7.1 anwendbar ist und welche keinen clash enthält, ist immer erfüllbar.

Beweis

(i) Eine Formelmenge S enthält einen clash genau dann, wenn S inkompatible Formeln enthält. Es ist also für alle in Definition 7.9 aufgelisteten Formelmengen zu zeigen, daß die Konjunktion der Elemente der Menge unerfüllbar ist. Dazu wird immer von der Existenz einer Interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \alpha, \beta, \gamma)$ ausgegangen, welche die betreffende Konjunktion erfüllt, und diese Annahme wird zum Widerspruch geführt.

1. $\mathtt{c_1(x)}, \ldots, \mathtt{c_n(x)}$, $n \geq 2$, $\mathtt{c_1(x)}, \ldots, \mathtt{c_n(x)}$ unverträglich

   $\mathcal{I} \models \mathtt{(c_1(x)\ :and\ \ldots\ c_n(x))}$

   gdw[11] $\mathcal{I} \models \mathtt{c_1(x)}$ und ... und $\mathcal{I} \models \mathtt{c_n(x)}$

   gdw $\alpha(\mathtt{x}) \in \gamma(\mathtt{c_1})$ und ... und $\alpha(\mathtt{x}) \in \gamma(\mathtt{c_n})$

   Es ergibt sich ein offensichtlicher Widerspruch zur Voraussetzung, daß $\mathtt{c_1}, \ldots, \mathtt{c_n}$ *unverträglich sind,*

2. $\mathtt{:nonexistent(x)}, \mathtt{c(x)}$

   $\mathcal{I} \models \mathtt{(:nonexistent(x)\ :and\ c(x))}$

   gdw $\mathcal{I} \models \mathtt{:nonexistent(x)}$ und $\mathcal{I} \models \mathtt{c(x)}$

   gdw $\alpha(\mathtt{x}) \in \Gamma$ und $\alpha(\mathtt{x}) \in \gamma(\mathtt{c}) \subseteq A$

   Es ergibt sich ein offensichtlicher Widerspruch, da A *und* Γ *disjunkt sind,*

3. $\mathtt{:undefined(x.p_1.\ldots.p_n)}$,
   $\mathtt{:eq(x.p_1,y_1)}, \mathtt{:eq(y_1.p_2,y_3)}, \ldots, \mathtt{:eq(y_{n-1}.p_n,y_n)}$

   $\mathcal{I} \models \mathtt{:undefined(x.p_1.\ldots.p_n)\ :and\ :eq(x.p_1,y_1)\ :and}$
   $\mathtt{:eq(y_1.p_2,y_2)\ :and\ \ldots\ :and\ :eq(y_{n-1}.p_n,\ y_n)}$

   gdw $\mathcal{I} \models \mathtt{:undefined(x.p_1.\ldots.p_n)}$ und
   $\mathcal{I} \models \mathtt{:eq(x.p_1,y_1)}$ und
   $\mathcal{I} \models \mathtt{:eq(y_1.p_2,y_2)}$ und ... und
   $\mathcal{I} \models \mathtt{:eq(y_{n-1}.p_n,y_n)}$

   gdw $\beta_{p_n}(\beta_{p_{n-1}}(\ldots(\beta_{p_1}(\alpha(\mathtt{x})))\ldots)) = \mathtt{NIL}$ und
   $\beta_{p_1}(\alpha(\mathtt{x})) = \alpha(\mathtt{y_1})$ und $\beta_{p_2}(\alpha(\mathtt{y_1})) = \alpha(\mathtt{y_2})$
   und ... und
   $\beta_{p_n}(\alpha(\mathtt{y_{n-1}})) = \alpha(\mathtt{y_n})$

   wenn $\beta_{p_n}(\beta_{p_{n-1}}(\ldots(\beta_{p_2}(\alpha(\mathtt{y_1})))\ldots)) = \mathtt{NIL}$ und
   $\beta_{p_2}(\alpha(\mathtt{y_1})) = \alpha(\mathtt{y_2})$ und ... und $\beta_{p_n}(\alpha(\mathtt{y_{n-1}})) = \alpha(\mathtt{y_n})$

   ...

   wenn $\beta_{p_n}(\alpha(\mathtt{y_n})) = \mathtt{NIL}$ und
   $\beta_{p_n}(\alpha(\mathtt{y_{n-1}})) = \alpha(\mathtt{y_n})$

   Es ergibt sich ein offensichtlicher Widerspruch, da NIL *nicht Bestandteil des Wertebereiches der Abbildung* α *ist,*

4. $\mathtt{:undefined(x.p_1.\ldots.p_n)}$,
   $\mathtt{:eq(x.p_1,y_1)}, \mathtt{:eq(y_1.p_2,y_2)}, \ldots, \mathtt{:some\text{-}eq(y_{n-1}.p_n,y_n)}$

$\mathcal{I} \models$:undefined(x.p_1.....p_n) :and :eq(x.p_1,y_1) :and :eq(y_1.p_2,y_2)
:and ... :and :some-eq(y_{n-1}.p_n, y_n)

gdw $\mathcal{I} \models$:undefined(x.p_1.....p_n) und
$\mathcal{I} \models$:eq(x.p_1,y_1) und
$\mathcal{I} \models$:eq(y_1.p_2,y_2) und ... und
$\mathcal{I} \models$:some-eq(y_{n-1}.p_n,y_n)

gdw $\beta_{p_n}(\beta_{p_{n-1}}(\ldots(\beta_{p_1}(\alpha(\mathtt{x})))\ldots)) = \mathtt{NIL}$ und
$\beta_{p_1}(\alpha(\mathtt{x})) = \alpha(y_1)$ und $\beta_{p_2}(\alpha(y_1)) = \alpha(y_2)$ und ... und $\alpha(y_n) \in \beta_{p_n}(\alpha(y_{n-1}))$

wenn $\beta_{p_n}(\beta_{p_{n-1}}(\ldots(\beta_{p_2}(\alpha(y_1)))\ldots)) = \mathtt{NIL}$ und
$\beta_{p_2}(\alpha(y_1)) = \alpha(y_2)$ und ... und $\alpha(y_n) \in \beta_{p_n}(\alpha(y_{n-1}))$

...

wenn $\beta_{p_n}(\alpha(y_n)) = \mathtt{NIL}$ und
$\alpha(y_n) \in \beta_{p_n}(\alpha(y_{n-1}))$

Es ergibt sich ein offensichtlicher Widerspruch, da NIL *nicht Bestandteil des Wertebereiches der Abbildung α ist,*

5. c(x), :eq(x.p_1,y), wenn p_1 in keiner Subklasse von c einen Slot mit objektwertigem Wertebereich bezeichnet.

   $\mathcal{I} \models$ (c(x) :and :eq(x.p_1,y)
   gdw $\mathcal{I} \models$ c(x) und $\mathcal{I} \models$:eq(x.p_1,y)
   gdw $\alpha(\mathtt{x}) \in \gamma(\mathtt{c})$ und $\beta_{p_1}(\alpha(\mathtt{x})) = \alpha(\mathtt{y})$

   Es ergibt sich ein Widerspruch, da nach Voraussetzung δ(c,p_1) keine Facette :dom d *für ein beliebiges* d$\in C$ *enthält. Damit enthält entweder keine Subklasse von* c *einen Slot* p_1*, so daß $\beta_{p_1}(\alpha(\mathtt{x})) =$* NIL *gilt, oder eine Subklasse von* c *enthält einen Slot* p_1*, der nach der Definition von* VeDa *eine* :dom*-Facette besitzen muß, so daß β_{p_1} alle Elemente aus A auf Teilmengen von A oder auf* NIL *abbildet.*

6. c(x), :some-eq(x.p_1,y), wenn p_1 in keiner Subklasse von c einen Slot mit einem mengen- oder listenwertigem Wertebereich bezeichnet.

   $\mathcal{I} \models$ (c(x) :and :some-eq(x.p_1,y)
   gdw $\mathcal{I} \models$ c(x) und $\mathcal{I} \models$:some-eq(x.p_1,y)
   gdw $\alpha(\mathtt{x}) \in \gamma(\mathtt{c})$ und $\alpha(\mathtt{y}) \in \beta_{p_1}(\alpha(\mathtt{x}))$

   Es ergibt sich ein Widerspruch, da nach Voraussetzung δ(c,p_1) keine Facette :dom ...{d} *oder* :dom /d/ *für ein beliebiges* d$\in C$ *enthält und damit entweder keine Subklasse von* c *einen Slot* p_1 *enthält und somit $\beta_{p_1}(\alpha(\mathtt{x}) =$* NIL *oder, weil ein existierender Slot immer eine* :dom*-Facette besitzen muß, β_{p_1} alle Elemente aus A auf einzelne Elemente aus A oder auf* NIL *abbildet.*

7. $\neg C, C$
   Der Widerspruch in diesem Fall ist offensichtlich, da eine Interpretation nicht gleichzeitig Modell für eine Formel und ihre Negation sein kann.

(ii) Lokale Korrektheit bedeutet, daß für jede Vervollständigungsregel der Form $S \rightarrow S_i = S \cup S_i'$ mit $1 \leq i \leq n$, $n \in \mathbb{N}$ gilt: Ist diese Regel auf S anwendbar, dann gilt für alle Interpretationen $\mathcal{I}$

$$\mathcal{I} \models S \;\Leftrightarrow\; \textit{es gibt ein } i \textit{ mit } \mathcal{I} \models S_i.$$

Die Richtung "$\Leftarrow$" ist dabei trivial, da $S \subseteq S_i$ für alle i. Die Richtung "$\Rightarrow$" muß für jede Vervollständigungsregel einzeln gezeigt werden. Dabei ist es ausreichend, zu zeigen, daß für alle Interpretationen $\mathcal{I} \models S \;\Rightarrow\; \mathcal{I} \models S_i'$ gilt. Wir nehmen im folgenden immer an, daß S die in der Vorbedingung der entsprechenden Regel beschriebene Form hat.

1. $C \wedge D \in S$,
   $S \rightarrow S \cup \{C, D\}$

   $\mathcal{I} \models S$
   $\Rightarrow \quad \mathcal{I} \models (C \wedge D)$
   $\Leftrightarrow \quad \mathcal{I} \models C$ und $\mathcal{I} \models D$
   $\Leftrightarrow \quad \mathcal{I} \models \{C, D\}$

2. $C \vee D \in S$,
   $S \rightarrow S_1 = S \cup \{C\}$
   $S \rightarrow S_2 = S \cup \{D\}$

   $\mathcal{I} \models S$
   $\Rightarrow \quad \mathcal{I} \models (C \vee D)$
   $\Leftrightarrow \quad \mathcal{I} \models C$ oder $\mathcal{I} \models D$
   $\Leftrightarrow \quad \mathcal{I} \models \{C\}$ oder $\mathcal{I} \models \{D\}$

3. `:is-defined(x.p)` $\in S$
   $S \rightarrow S_1 = S \cup \{$`:eq(x.p,z)` $\}$
   $S \rightarrow S_2 = S \cup \{$`:some-eq(x.p,z)` $\}$

   $\mathcal{I} \models S$
   $\Rightarrow \quad \mathcal{I} \models$ `:is-defined(x.p)`
   $\Leftrightarrow \quad \beta_p(\alpha($`x`$)) \neq$ `NIL`
   $\Rightarrow \quad \beta_p(\alpha($`x`$)) \subseteq A$ oder $\beta_p(\alpha($`x`$)) \in A$
   $\Rightarrow \quad$ es existiert ein `z`[12] $\in X$ mit
   $\alpha($`z`$) \in \beta_p(\alpha($`x`$))$ oder $\alpha($`z`$) = \beta_p(\alpha($`x`$))$
   $\Rightarrow \quad \mathcal{I} \models$ `:some-eq(x.p,z)` oder $\mathcal{I} \models$ `:eq(x.p,z)`

4. $\neg$`:nonexistent(x)` $\in S$
   $S \rightarrow S \cup \{$`ROOT(x)` $\}$

   $\mathcal{I} \models S$
   $\Rightarrow \quad \mathcal{I} \models \neg$`:nonexistent(x)`
   $\Leftrightarrow \quad$ nicht $\mathcal{I} \models$`:nonexistent(x)`
   $\Leftrightarrow \quad$ nicht $\alpha($`x`$) \in \Gamma$
   $\Leftrightarrow \quad \alpha($`x`$) \in A$
   $\Leftrightarrow \quad \mathcal{I} \models$ `ROOT(x)`

5. $\neg$`c(x)` $\in S$

   $S \rightarrow S_1 = S \cup \{$`:not c(x)`$\}$
   $S \rightarrow S_2 = S \cup \{$`:nonexistent(x)`$\}$

[12] Nach Voraussetzung existieren zu jedem $a \in A$ beliebig viele `z`$\in V$, so daß $\alpha($`z`$) = a$ gilt. Damit existiert immer auch ein `z`$\in V$ mit dieser Eigenschaft, das nicht Bestandteil einer Formel aus S ist.

$\mathcal{I} \models S$
$\Rightarrow$ $\mathcal{I} \models \neg$`c(x)`
$\Leftrightarrow$ nicht $\mathcal{I} \models$`c(x)`
$\Leftrightarrow$ nicht $\alpha($`x`$) \in \gamma($`c`$)$
$\Leftrightarrow$ $\alpha($`x`$) \in \Gamma \cup A \backslash \gamma($`c`$)$
$\Rightarrow$ $\mathcal{I} \models$ `:nonexistent(x)` oder $\mathcal{I} \models$ `:not c(x)`

6. Der Beweis verläuft analog zu 5.

7. $\neg$`:undefined(x.p)`$\in S$
   $S \rightarrow S \cup \{$`:is-defined(x.p)` $\}$

   $\mathcal{I} \models S$
   $\Rightarrow$ $\mathcal{I} \models \neg$`:undefined(x.p)`
   $\Leftrightarrow$ nicht $\mathcal{I} \models$`:undefined(x.p)`
   $\Leftrightarrow$ nicht $\beta_p(\alpha($`x`$)) =$`NIL`
   $\Leftrightarrow$ $\beta_p(\alpha($`x`$)) \neq$ `NIL`
   $\Leftrightarrow$ $\mathcal{I} \models$`:is-defined(x.p)`

8. $\neg$`:is-defined(x.p)`$\in S$
   $S \rightarrow S \cup \{$`:undefined(x.p)` $\}$

   $\mathcal{I} \models S$
   $\Rightarrow$ $\mathcal{I} \models \neg$`:is-defined(x.p)`
   $\Leftrightarrow$ nicht $\mathcal{I} \models$`:is-defined(x.p)`
   $\Leftrightarrow$ nicht $\beta_p(\alpha($`x`$)) \neq$`NIL`
   $\Leftrightarrow$ $\beta_p(\alpha($`x`$)) =$ `NIL`
   $\Leftrightarrow$ $\mathcal{I} \models$`:undefined(x.p)`

9. $\neg$`:all(x.p,c)`$\in S$
   $S \rightarrow S_1 = S \cup \{$`:eq(x.p,y), :not c(x)` $\}$
   $S \rightarrow S_2 = S \cup \{$`:some-eq(x.p,y), :not c(x)`$\}$
   für ein beliebiges `y`$\in V$, welches bisher noch in keiner Formel aus S vorkommt.

   $\mathcal{I} \models S$
   $\Rightarrow$ $\mathcal{I} \models \neg$`:all(x.p,c)`
   $\Leftrightarrow$ nicht $\mathcal{I} \models$`:all(x.p,c)`
   $\Leftrightarrow$ nicht ($\beta_p(\alpha($`x`$)) =$`NIL` oder $\beta_p(\alpha($`x`$)) \subseteq \gamma($`c`$)$ oder $\beta_p(\alpha($`x`$)) \in \gamma($`c`$)$)
   $\Leftrightarrow$ $\beta_p(\alpha($`x`$)) \neq$`NIL` und $\beta_p(\alpha($`x`$)) \not\subseteq \gamma($`c`$)$ und $\beta_p(\alpha($`x`$)) \notin \gamma($`c`$)$)
   $\Rightarrow$ $\beta_p(\alpha($`x`$)) \subseteq A \backslash \gamma($`c`$)$ oder $\beta_p(\alpha($`x`$)) \in A \backslash \gamma($`c`$)$)
   $\Rightarrow$ es existiert ein `y`$\in V$ mit $\alpha($`y`$) = a \in A \backslash \gamma($`c`$)$ und
   ($a \in \beta_p(\alpha($`x`$))$ oder $a = \beta_p(\alpha($`x`$))$)
   $\Rightarrow$ $\mathcal{I} \models$`:not c(y)` und ($\mathcal{I} \models$`:eq(x.p,y)` oder $\mathcal{I} \models$`:some-eq(x.p,y)`)

10. Der Beweis verläuft analog zu 9.

11. $\neg$`:some-eq(x.p`$_1$`...p`$_n$`,y)`$\in S$, `n`≥ 2
    $S \rightarrow S_1 = S \cup \{$`:undefined(x.p`$_1$`...p`$_n$`)`$\}$
    $S \rightarrow S_2 = S \cup \{$`:eq(x.p`$_1$`,z), ` $\neg$`:some-eq(x.p`$_2$`...p`$_n$`,y)`$\}$

$\mathcal{I} \models S$
$\Rightarrow$ $\mathcal{I} \models \neg$ `:some-eq(x.p`$_1$`...p`$_n$`,y)`
$\Leftrightarrow$ nicht $\mathcal{I} \models$ `:some-eq(x.p`$_1$`...p`$_n$`,y)`
$\Leftrightarrow$ nicht $\alpha($`y`$) \in \beta_{p_n}(\ldots(\beta_{p_2}(\beta_{p_1}(\alpha($`x`$))))\ldots)$
$\Rightarrow$ $\beta_{p_n}(\ldots(\beta_{p_2}(\beta_{p_1}(\alpha($`x`$))))\ldots) =$`NIL` oder
es existiert ein `z` $\in V$ mit $\alpha($`z`$) = \beta_{p_1}(\alpha($`x`$))$ und
$\alpha($`y`$) \notin \beta_{p_n}(\ldots(\beta_{p_2}(\alpha($`z`$))))\ldots)$
$\Rightarrow$ $\mathcal{I} \models$ `:undefined(x.p`$_1$`...p`$_n$`)` oder
$\mathcal{I} \models \{$`:eq(x.p`$_1$`,z), ` $\neg$`:some-eq(z.p`$_2$`...p`$_n$`,y)`$\}$

12. Der Beweis verläuft analog zu 11.

13. `:all(x.p`$_1$`...p`$_n$`,c)`$\in S$, `:eq(x.p`$_1$`,y)`$\in S$
$S \rightarrow S \cup \{$`:all(y.p`$_2$`...p`$_n$`,c)`$\}$

$\mathcal{I} \models S$
$\Rightarrow$ $\mathcal{I} \models$`:all(x.p`$_1$`...p`$_n$`,c)` und $\mathcal{I} \models$`:eq(x.p`$_1$`,y)`
$\Leftrightarrow$ $(\beta_{p_n}(\ldots(\beta_{p_2}(\beta_{p_1}(\alpha($`x`$)))\ldots) =$`NIL` oder
$\beta_{p_n}(\ldots(\beta_{p_2}(\beta_{p_1}(\alpha($`x`$)))\ldots) \subseteq \gamma($`c`$))$ oder
$\beta_{p_n}(\ldots(\beta_{p_2}(\beta_{p_1}(\alpha($`x`$)))\ldots) \in \gamma($`c`$)))$ und
$\beta_{p_1}(\alpha($`x`$)) = \alpha($`y`$)$
$\Rightarrow$ $(\beta_{p_n}(\ldots(\beta_{p_2}(\alpha($`y`$))\ldots) =$`NIL` oder
$\beta_{p_n}(\ldots(\beta_{p_2}(\alpha($`y`$))\ldots) \subseteq \gamma($`c`$))$ oder
$\beta_{p_n}(\ldots(\beta_{p_2}(\alpha($`y`$))\ldots) \in \gamma($`c`$)))$
$\Leftrightarrow$ $\mathcal{I} \models$`:all(y.p`$_2$`...p`$_n$`,c)`

14. Der Beweis verläuft analog zu 13.

15. `:all(x.p, c)`$\in S$ oder `:all(x.p, :not c)`$\in S$

$S \rightarrow S_1 = S \cup \{$ `:is-defined(x.p)`$\}$
$S \rightarrow S_2 = S \cup \{$ `:undefined(x.p)`$\}$
Der Beweis wird exemplarisch für `:all(x.p,c)` geführt.
$\mathcal{I} \models S$
$\Rightarrow$ $\mathcal{I} \models$ `:all(x.p,c)`
$\Leftrightarrow$ $\beta_p(\alpha($`x`$)) =$`NIL` oder $\beta_p(\alpha($`x`$)) \subseteq \gamma($`c`$)$ oder
$\beta_p(\alpha($`x`$)) \in \gamma($`c`$)$
$\Rightarrow$ $\beta_p(\alpha($`x`$)) =$`NIL` oder $\beta_p(\alpha($`x`$)) \neq$`NIL`
$\Leftrightarrow$ $\mathcal{I} \models$ `:undefined(x.p)` oder $\mathcal{I} \models$ `:is-defined(x.p)`

16. `:eq(x.p`$_1$`.....p`$_n$`,y)` $\in S$, $n \geq 2$

$S \rightarrow S \cup \{$`:eq(x.p`$_1$`,z), :eq(z.p`$_2$`.....p`$_n$`,y)`$\}$
$\mathcal{I} \models S$
$\Rightarrow$ $\mathcal{I} \models$`:eq(x.p`$_1$`.....p`$_n$`,y)`
$\Leftrightarrow$ $\beta_{p_n}(\ldots(\beta_{p_2}(\beta_{p_1}(\alpha($`x`$)))\ldots) = \alpha($`y`$)$
$\Rightarrow$ $\exists a \in A$ mit $a = \beta_{p_1}(\alpha($`x`$))$ und
$\beta_{p_n}(\ldots(\beta_{p_2}(a))\ldots) = \alpha($`y`$)$
$\Rightarrow$ $\exists$`z`$\in X$ mit $\beta_{p_1}(\alpha($`x`$)) = \alpha($`z`$)$ und
$\beta_{p_n}(\ldots(\beta_{p_2}(\alpha($`z`$)))\ldots) = \alpha($`y`$)$
$\Rightarrow$ $\mathcal{I} \models$`:eq(x.p`$_1$`,z)` und $\mathcal{I} \models$`:eq(z.p`$_2$`.....p`$_n$`,y)`

17. Der Beweis verläuft analog zu 16.

18. `:eq(x,y)`$\in S$

$S \to \{\phi\frac{x}{y} \mid \phi \in S\}$

$\mathcal{I} \models S$
$\Rightarrow \quad \mathcal{I} \models$ `:eq(x,y)`
$\Leftrightarrow \quad \alpha(\mathtt{x}) = \alpha(\mathtt{y})$
$\Rightarrow \quad \mathcal{I} \models \phi \Leftrightarrow \mathcal{I} \models \phi\frac{x}{y}$

19. `c(x)`$\in S$
    $S \to S_i = S \cup \{c_i(\mathtt{x})$
    für alle nicht abstrakten Subklassen $\mathtt{c}_i$ von `c`
    $\mathcal{I} \models S$
    $\Rightarrow \quad \mathcal{I} \models$ `c(x)`
    $\Leftrightarrow \quad \alpha(\mathtt{x}) \in \gamma(\mathtt{c})$
    $\Rightarrow \quad \alpha(\mathtt{x}) \in \gamma(\mathtt{c}_1) \cup \cdots \cup \gamma(\mathtt{c}_n)$
    für alle nicht abstrakten Subklassen $\mathtt{c}_i$, $1 \leq i \leq n$, von `c`
    $\Rightarrow \quad \mathcal{I} \models \mathtt{c}_1(\mathtt{x})$ oder ... oder $\mathcal{I} \models \mathtt{c}_n(\mathtt{x})$
20. Der Beweis verläuft analog zu 19.
21. `c'(x)` $\in S$

    $S \to S \cup \{$ `c(x)` $\mid$ `c` Superklasse von `c'` $\}$
    $\mathcal{I} \models S$
    $\Rightarrow \quad \mathcal{I} \models$ `c'(x)`
    $\Leftrightarrow \quad \alpha(\mathtt{x}) \in \gamma(\mathtt{c'})$
    $\Rightarrow \quad \alpha(\mathtt{x}) \in \gamma(\mathtt{c})$
    für alle Superklassen `c` von `c'`
    $\Rightarrow \quad \mathcal{I} \models$ `c'(x)` für alle Superklassen `c` von `c'`
22. `c(x)`$\in S$, `:eq(x.`$\mathtt{p}_1$`,y)` $\in S$, der in der Klasse `c` durch $\mathtt{p}_1$ bezeichnete Slot hat einen objektwertigen Wertebereich mit Typ `d`

    $S \to S \cup \{$ `d(y)` $\}$
    $\mathcal{I} \models S$
    $\Rightarrow \quad \mathcal{I} \models$ `c(x)` und $\mathcal{I} \models$ `:eq(x.`$\mathtt{p}_1$`,y)`
    $\Leftrightarrow \quad \alpha(\mathtt{x}) \in \gamma(\mathtt{c})$ und $\beta_{p_1}(\alpha(x)) = \alpha(\mathtt{y})$
    $\Rightarrow \quad$ (wegen `:dom d`$\in \delta($`c,p`$)$)
    $\alpha(\mathtt{y}) \in \gamma(\mathtt{d})$
    $\Rightarrow \quad \mathcal{I} \models$ `d(y)`
23. Der Beweis verläuft analog zu 22.
24. `:eq(x.p,y)` $\in S$, `:eq(x.p,z)` $\in S$

    $S \to S \cup \{$ `:eq(y,z)` $\}$
    $\mathcal{I} \models S$
    $\Rightarrow \quad \mathcal{I} \models$ `:eq(x.p,y)` und $\mathcal{I} \models$ `:eq(x.p,z)`
    $\Leftrightarrow \quad \beta_p(\alpha(\mathtt{x})) = \alpha(\mathtt{y})$ und $\beta_p(\alpha(\mathtt{x})) = \alpha(\mathtt{z})$
    $\Rightarrow \quad \alpha(\mathtt{y}) = \alpha(\mathtt{z})$
    $\Leftrightarrow \quad \mathcal{I} \models$ `:eq(y,z)`
25. : θ`(x.`$\mathtt{p}_1$`,y)` $\in S$, θ'`(x'.`$\mathtt{p}_1$`',y)` $\in S$, mit $\theta, \theta' \in \{$`:eq, :some-eq`$\}$, `c(x)` $\in S$, `c'(x')` $\in S$, in der Klasse `c` hat der Slot $\mathtt{p}_1$ und die Facette `:exc t` und in der Klasse `c'` der Slot $\mathtt{p}_1$`'` die Facette `:comp t`

$S \rightarrow S \cup \{$:eq(x,x')$\}$

Der Beweis wird exemplarisch für $\theta =$:eq geführt.

$\mathcal{I} \models S$
$\Rightarrow$ $\mathcal{I} \models$:eq(x.p_1,y) und $\mathcal{I} \models$:eq(x'.p_1',y)
$\Leftrightarrow$ $\beta_{p_1}(\alpha(\mathtt{x})) = \alpha(\mathtt{y})$ und $\beta_{p_1'}(\alpha(\mathtt{x'})) = \alpha(\mathtt{y})$
$\Rightarrow$ (wegen :exc t$\in \delta($c,$p_1)$ und :comp t$\in \delta($c',$p_1')$)
$\alpha(\mathtt{x}) = \alpha(\mathtt{x'})$
$\Leftrightarrow$ $\mathcal{I} \models$:eq(x,x')

26. c(x)$\in S$ und :eq(x.p_1,y) oder :some-eq(x.p_1,y) $\in S$ und der in der Klasse c durch p_1 bezeichnete Slot hat die Facette :inv p_1'

$S \rightarrow S_1 = S \cup \{$:eq(y.$p_1$',x)$\}$
$S \rightarrow S_2 = S \cup \{$:some-eq(y.$p_1$',x)$\}$

Der Beweis wird exemplarisch für :eq(x.p_1,y)$\in S$ geführt.

$\mathcal{I} \models S$
$\Rightarrow$ $\mathcal{I} \models$ c(x) und $\mathcal{I} \models$:eq(x.p_1,y)
$\Leftrightarrow$ $\alpha(\mathtt{x}) \in \gamma(\mathtt{c})$ und $\beta_{p_1}(\alpha(\mathtt{x})) = \alpha(\mathtt{y})$
$\Rightarrow$ (wegen :inv p_1'$\in \delta($c,$p_1)$)
$\beta_{p_1'}(\alpha(\mathtt{y})) = \alpha(\mathtt{x})$
$\Leftrightarrow$ $\mathcal{I} \models$:eq(y.p_1',x)

27. (:eq(x.p_1,y)$\in S$ oder :some-eq(x.p_1,y)$\in S$) und :all(x.p_1,c) $\in S$

$S \rightarrow S \cup \{$ c(y)$\}$

Der Beweis wird exemplarisch für :some-eq(x.p_1,y) $\in S$ geführt.

$\mathcal{I} \models S$
$\Rightarrow$ $\mathcal{I} \models$:some-eq(x.p_1,y) und $\mathcal{I} \models$:all(x.p_1,c)
$\Leftrightarrow$ $\alpha(\mathtt{y}) \in \beta_{p_1}(\alpha(\mathtt{x})))$ und
$(\beta_{p_1}((\alpha(\mathtt{x}))) =$NIL oder $\beta_{p_1}((\alpha(\mathtt{x}))) \subseteq \gamma(\mathtt{c})$ oder $\beta_{p_1}((\alpha(\mathtt{x}))) \in \gamma(\mathtt{c}))$
$\Rightarrow$ $\alpha(\mathtt{y}) \in \gamma(\mathtt{c})$
$\Leftrightarrow$ $\mathcal{I} \models$ c(y)

28. Der Beweis verläuft analog zu 27.

29.

c(x) $\in S$ oder
:is-defined(x.p) $\in S$ oder
:eq(x.p,y) $\in S$ oder
:eq(y.p,x) $\in S$ oder
:some-eq(x.p,y) $\in S$ oder
:some-eq(y.p,x) $\in S$

mit beliebigem y$\in X$

$S \rightarrow S \cup \{$ ROOT(x) $\}$

Der Beweis wird exemplarisch für :is-defined(x.p) geführt.

$\mathcal{I} \models S$
$\Rightarrow$ $\mathcal{I} \models$:is-defined(x.p)
$\Leftrightarrow$ $\beta_p(\alpha(\mathtt{x})) \neq$NIL
$\Rightarrow$ $\alpha(\mathtt{x}) \in A$
$\Leftrightarrow$ $\mathcal{I} \models$ ROOT(x)

30.

:undefined(x.p) $\in S$ oder
:eq(x,y) $\in S$ oder
:eq(y,x) $\in S$ oder
$\neg$:eq(x,y) $\in S$ oder
$\neg$:eq(y,x) $\in S$ oder
:all(x.p,c) $\in S$ oder
:all(x.p,:not c) $\in S$

mit beliebigem y$\in X$

$$S \to S_1 = S \cup \{ \texttt{ROOT(x)} \}$$
$$S \to S_2 = S \cup \{ \texttt{:nonexistent(x)} \}$$

Der Beweis wird exemplarisch für :all(x.p,c) geführt.

$\mathcal{I} \models S$
$\Rightarrow$ $\mathcal{I} \models$:all(x.p,c)
$\Leftrightarrow$ $\beta_p(\alpha(\texttt{x})) =$NIL oder $\beta_p(\alpha(\texttt{x})) \subseteq \gamma(\texttt{c})$ oder $\beta_p(\alpha(\texttt{x})) \in \gamma(\texttt{c})$
$\Rightarrow$ $\alpha(\texttt{x}) \in \Gamma$ oder $\alpha(\texttt{x}) \in A$
$\Leftrightarrow$ $\mathcal{I} \models$ ROOT(x) oder $\mathcal{I} \models$:nonexistent(x)

31. c(x), :some-eq(x.p_1,y_1),..., :some-eq(x.p_1,y_n) $\in S$ und der in der Klasse c durch p_1 festgelegte Slot hat eine Kardinalitätsbeschränkung von maximal $n-k$ $(k \geq 1)$

$$S \to S_m = S \cup \{ \texttt{:eq(}y_i\texttt{,}y_j\texttt{)} \}$$

für alle (i,j) mit $1 \leq i,j \leq n, i \neq j$, $1 \leq m \leq |\{(i,j) \mid i \neq j, 1 \leq i,j \leq n\}|$

$\mathcal{I} \models S$
$\Rightarrow$ $\mathcal{I} \models$ c(x) und
$\mathcal{I} \models$:some-eq(x.p_1,y_1) und ... und $\mathcal{I} \models$:some-eq(x.p_1,y_n)
$\Leftrightarrow$ $\alpha(\texttt{x}) \in \gamma(\texttt{c})$ und $\alpha(y_1) \in \beta_{p_1}(\alpha(\texttt{x}))$ und ... und $\alpha(y_n) \in \beta_{p_1}(\alpha(\texttt{x}))$
$\Rightarrow$ (wegen der Kardinalitätsbeschränkung von p_1 in c und weil $\beta_{p_1}(\alpha(\texttt{x}) \neq$ NIL ist gilt $|\beta_{p_1}(\alpha(\texttt{x}))| \leq n-k$)
es existieren $i,j \leq n$ mit $\alpha(y_i) = \alpha(y_j)$
$\Rightarrow$ $\mathcal{I} \models$:eq(y_i,y_j)

32. :some-eq(x.p_1,y_1),..., :some-eq(x.p_1,y_n) $\in S$, n ≥ 1, es gibt kein weiteres y so daß :some-eq(x.p_1,y)$\in S$, und der durch x.p_1 festgelegte Slot hat eine Kardinalitätsbeschränkung von mindestens $n+k$ $(k \geq 1)$, dann

$$S \to S \cup \{ \texttt{:some-eq(x.}p_1\texttt{,z)}, \neg\texttt{:eq(z,}y_1\texttt{)}, \ldots, \neg\texttt{:eq(z,}y_n\texttt{)} \}$$

für ein beliebiges z, welches bisher in keiner Formel aus S vorkommt.

$\mathcal{I} \models S$
$\Rightarrow$ $\mathcal{I} \models$:some-eq(x.p_1,y_1) und ... und $\mathcal{I} \models$:some-eq(x.p_1,y_n)
$\Leftrightarrow$ $\alpha(y_1) \in \beta_{p_1}(\alpha(\texttt{x}))$ und ... und $\alpha(y_n) \in \beta_{p_1}(\alpha(\texttt{x}))$
$\Rightarrow$ (wegen der Kardinalitätsbeschränkung von p_1 in c und weil $\beta_{p_1}(\alpha(\texttt{x}) \neq$ NIL ist gilt $|\beta_{p_1}(\alpha(\texttt{x}))| \geq n+k$)
es existiert ein z$\in V$ so daß $\alpha(\texttt{z}) \in \beta_{p_1}(\alpha(\texttt{x}))$ und $\alpha(\texttt{z}) \neq \alpha(y_i)$ für alle $1 \leq i \leq n$
$\Rightarrow$ $\mathcal{I} \models$:some-eq(x.p_1,z) und $\mathcal{I} \models \neg$:eq(y_i,z) für alle $1 \leq i \leq n$

33. c(x)$\in S$, der Slot p_1 hat in c die Facette :req t, S enthält eine der Formeln :undefined(x.p_1), :all(x.p_1,d), :all(x.p_1,:not d) für beliebi-

ge $\mathtt{d} \in C$

$$S \to S_1 = S \cup \{\ \mathtt{:some\text{-}eq(x.p_1,y)}\}$$
$$S \to S_2 = S \cup \{\ \mathtt{:eq(x.p_1,y)}\}$$

für ein beliebiges $\mathtt{y}$, welches bisher in keiner Formel aus S vorkommt.
Der Beweis wird exemplarisch für $\mathtt{:all(x.p_1,d)}$ geführt.

$\mathcal{I} \models S$
$\Rightarrow$ $\mathcal{I} \models \mathtt{:all(x.p_1,d)}$ und $\mathcal{I} \models \mathtt{c(x)}$
$\Leftrightarrow$ ($\beta_{p_1}(\alpha(\mathtt{x})) = \mathtt{NIL}$ oder $\beta_{p_1}(\alpha(\mathtt{x})) \subseteq \gamma(\mathtt{d})$ oder $\beta_{p_1}(\alpha(\mathtt{x})) \in \gamma(\mathtt{d})$) und $\alpha(\mathtt{x}) \in \gamma(\mathtt{c})$
$\Rightarrow$ (wegen $\mathtt{:req\ t} \in \delta(\mathtt{c},\mathtt{p_1})$) $\beta_{p_1}(\alpha(\mathtt{x})) \subseteq \gamma(\mathtt{d})$ oder $\beta_{p_1}(\alpha(\mathtt{x})) \in \gamma(\mathtt{d})$
$\Rightarrow$ es existiert ein $\mathtt{y} \in V$ mit $\alpha(\mathtt{y}) \in \gamma(\mathtt{d})$ und ($\alpha(\mathtt{y}) \in \beta_{p_1}(\alpha(\mathtt{x}))$ oder $\alpha(\mathtt{y}) = \beta_{p_1}(\alpha(\mathtt{x}))$)
$\Leftrightarrow$ $\mathcal{I} \models \mathtt{:some\text{-}eq(x,y)}$ oder $\mathcal{I} \models \mathtt{:eq(x,y)}$

(iii) Die Formulierung der Vervollständigungsregeln stellt sicher, daß jede Regelanwendung die Menge S echt erweitert und garantiert so, daß jede Regel auf jede Formel nur höchstens einmal angewendet werden kann. Die Ausgangsmenge enthält immer genau eine Formel.

Die Regeln 1 und 2 zerlegen zusammengesetzte Formeln aus S in ihre einzelnen Komponenten. Da keine Vervollständigungsregel außer den Regeln 1 und 2 die Menge S um Formeln, welche die Symbole $\wedge$ oder $\vee$ enthalten erweitert, S zu Beginn nur eine Formel mit endlich vielen dieser Symbole enthält und die Regeln 1 und 2 immer zu einer Verkürzung der Formeln führen können diese Regeln nur endlich oft angewandt werden.

Da die Regeln 3 bis 33 nur atomare und atomare negierte Formeln in die Menge S einführen und bei einer begrenzten Anzahl von Variablen und einer beschränkten Pfadlänge nur endlich viele solcher Formeln existieren reicht es aus, zu zeigen, daß durch die Anwendung der Vervollständigungsregeln nicht unendlich viele neue Variablen eingeführt werden und die Länge der Pfade in den zu S zugefügten Formeln nicht größer ist als die Länge bereits in S vorhandener Pfade.

Die zweite Bedingung ist aus den Regeln offensichtlich, da jede Regel, welche eine Formel mit einem Pfad in S einfügt den Pfad einer bereits in S enthaltenen Formel übernimmt oder sogar verkürzt.

Zur Überprüfung der ersten Bedingung müssen alle Regeln betrachtet werden, welche neue Variablen in die Menge S einbringen, im einzelnen die Regeln 3, 9, 10, 11, 12, 16, 17, 32 und 33.

Die Anwendung der Regeln 3 und 9 bis 12 basiert auf der Existenz von Formeln in S, welche durch keine Regel außer 1 und 2 in S eingefügt werden und kann deshalb nur endlich oft angewandt werden.

Die von den Regeln 16 und 17 als Ausgangsbasis benötigten Formeln werden lediglich durch die Regeln 1 bis 3, 9 und 10 sowie durch die Regeln 16 und 17 selbst in die Menge S eingefügt. Für die Regeln 1 bis 3, 9 und 10 wurde bereits gezeigt, daß diese nur endlich oft anwendbar sind. Die Anwendung der Regeln 16 und 17 führt jeweils zu einer echten Pfadverkürzung, so daß auch diese Regeln nur endlich oft anwendbar sind.

Regel 32 führt für jeden mengen- oder listenwertigen Slot, der eine Mindestkardinalität besitzt und definiert ist, zusätzliche Elemente ein, bis die Mindestkardinalität erreicht ist. Durch die zusätzlich eingeführten Formeln der Form $\neg$`:eq(z,`y_i`)` wird verhindert, daß durch diese Regel eingeführte Variablen mit bereits existierenden Variablen identifiziert werden, so daß die Regel für einen Slot mit Mindestkardinalität m maximal $(m-1)$-mal angewandt werden kann.

Regel 33 führt für Slots mit der Facette `:req t` neue Variablen ein. Dabei wird allerdings vorausgesetzt, daß bereits Informationen über den Slot vorliegen. Die für die Anwendung von Regel 32 benötigten Formeln werden aber nur von Regeln in die Menge S eingebracht, die nur endlich oft anwendbar sind.

(iv) Es ist zu zeigen, daß für jede clash-freie und vollständige Formelmenge S ein VeDa-Modell[13] existiert. Wir definieren zunächst aus den in der Menge S enthaltenen Formeln ein "Vormodell" und zeigen anschließend, daß sich aus diesem Vormodell ein VeDa-Modell erzeugen läßt.

Sei nun τ_S die Menge der in S enthaltenen Variablenbezeichner und σ_S die Menge der in S enthaltenen Pfade (Elemente der Menge P). Wir definieren eine Interpretation $\mathcal{I}$ wie folgt:

- $\Delta^{\mathcal{I}} := \tau_S \cup \{\texttt{NIL}\}$
- $A := \{\texttt{x} \in \Delta^{\mathcal{I}} \mid \texttt{ROOT(x)} \in S\}$
- $\Gamma := \{\texttt{x} \in \Delta^{\mathcal{I}} \mid \texttt{:nonexistent(x)} \in S\}$
- $\alpha := id$[14]
- für jeden Pfad $\texttt{p} \in \sigma_S$: $\beta(\texttt{p}) := \rho$ mit

$$\rho(\texttt{x}) = \begin{cases} \{\ \texttt{y} \mid \texttt{:some-eq(x.p,y)} \in S\} & \text{falls mind. eine Formel der Form} \\ & \texttt{:some-eq(x.p,y)} \in S \\ \texttt{y} & \text{falls } \texttt{:eq(x.p,y)} \in S \\ \texttt{NIL} & \text{falls } \texttt{:undefined(x.p)} \in S \end{cases}$$

- für alle Klassen `c`: $\gamma(\texttt{c}) := \{\ \texttt{x} \in \Delta^{\mathcal{I}} \mid \texttt{c(x)} \in S\}$

Da nach Voraussetzung keine der Regeln des Algorithmus mehr auf S anwendbar ist und die Menge S keinen clash enthält ist sichergestellt, daß für jedes $\texttt{x} \in \tau_S$ genau eine der Bedingungen `ROOT(x)` oder `:nonexistent(x)` in S enthalten ist, so daß jedes `x` eindeutig einer der Mengen A und Γ zugeordnet wird.

Weiterhin gilt für jeden Pfad $\texttt{p} \in \sigma_S$, daß dieser in mindestens einer Bedingung der Form `:some-eq(x.p,y)`, `:eq(x.p,y)` oder `:undefined(x.p)` in S vorkommt. Er kann wegen der Voraussetzung, daß S keinen clash enthält, für ein gegebenes `x` nur in genau einer dieser Bedingungen in S auftreten.

Für jedes $\texttt{x} \in \tau_S$ ist durch die Anwendung der Regeln des Algorithmus außerdem sichergestellt, daß wenn $\texttt{c'(x)} \in S$ gilt, dann gilt auch $\texttt{c(x)} \in S$ für alle Superklassen `c` von `c'`.

Für die Interpretation $\mathcal{I}$ gilt, daß $\mathcal{I}$ sämtliche Formeln in S erfüllt. Weiterhin erfüllt $\mathcal{I}$ diejenigen Randbedingungen aus Definition 7.3, welche sich aus den Facetten `:dom`, `:exc`, `:comp` und `:inv` ergeben.

[13] Also eine Menge von VeDa-Objekten, welche alle in der Menge S enthaltenen Bedingungen erfüllt.
[14] id bezeichnet die Abbildung, die jedes Element auf sich selbst abbildet.

Dies wird exemplarisch für einige Formeln und eine Randbedingung gezeigt, für die übrigen Fälle verläuft der Beweis analog. Sei ϕ eine beliebige atomare Formel aus S, dann können unter anderem folgende Fälle auftreten:

- $\phi =$`c(x)`
  Es gilt nach Definition von $\mathcal{I}$ $\alpha($`x`$) =$`x` und `x`$\in \gamma($`c`$)$ und somit auch $\mathcal{I} \models \phi$.
- $\phi =$`:not c(x)`
  Wegen Regel 19 gilt, daß `c'(x)`$\in S$ für eine nicht abstrakte Klasse `c'`, die nicht Subklasse von `c` ist. Nach Definition von $\mathcal{I}$ gilt wiederum $\alpha($`x`$) =$`x` und `x`$\in \gamma($`c'`$)$ sowie `x`$\notin \gamma($`c`$)$, da `c(x)` nicht in S enthalten sein kann (S ist clash-frei). Damit gilt wegen $\gamma($`c'`$) \subseteq A$ $\mathcal{I} \models \phi$.
- $\phi =$`:all(x.p,c)`
  Wegen den Regeln 15 und 3 gilt, daß entweder `:undefined(x.p)` oder eine der Formeln `:eq(x.p,z), :some-eq(x.p,z)` für ein `z`$\in V$ in S liegt. Im zweiten Fall enthält S wegen den Regeln 13 und 27 auch die Formel `c(y)`. Nach der Definition von $\mathcal{I}$ gilt damit, daß $\beta($`p`$)$ definiert ist und `x` auf `NIL`, ein Element aus $\gamma($`c`$)$ oder eine Teilmenge von $\gamma($`c`$)$ abbildet. Damit gilt $\mathcal{I} \models \phi$.

Wir betrachten nun die Randbedingung

Wenn `:exc t` $\in \delta($`c,p`$)$ *und* `:comp t` $\in \delta($`c',p'`$)$, *dann gilt für alle* $a \in \gamma($`c`$)$, $a' \in \gamma($`c'`$)$*: wenn* $\beta_p(a) = \beta_{p'}(a') \in A$, *dann gilt* $a = a'$.

und zeigen, daß diese Bedingung in $\mathcal{I}$ erfüllt ist.

- Es seien `x,y`$\in \tau_S$ mit `x`$\in \gamma($`c`$)$ und `y`$\in \gamma($`c'`$)$. Weiterhin gelte $\beta_p($`x`$) = \beta_{p'}($`y`$) \in A$. Nach der Definition von β in $\mathcal{I}$ muß gelten, daß es ein `z` gibt mit `:eq(x.p,z)`$\in S$ und `:eq(y.p',z)`$\in S$. Damit gilt aber wegen Regel 25, daß auch `:eq(x,y)`$\in S$. Wegen Regel muß `x=y` gelten.

Die Randbedingungen, welche sich aus der Facette `:req t` ergeben, werden von $\mathcal{I}$ nicht unbedingt erfüllt, da die Regeln des Algorithmus für einen Slot $\mathtt{p}_1$ von Objekt `x`, welcher in der Startformel nicht in einer Pfadangabe auftritt, keine Objekte generiert, auf welche dieser Slot verweist, selbst wenn der Slot in der Klasse, welcher er zugeordnet ist, die Facette `:req t` besitzt. In diesem Fall gilt in der Interpretation $\mathcal{I}$, daß $\beta_{p_1}($`x`$)$ undefiniert ist, während die entsprechende Randbedingung $\beta_{p_1}($`x`$) \subseteq A$ oder $\beta_{p_1}($`x`$) \in A$ fordert.

Da aber nach der VeDa-Sprachdefinition jede VeDa-Klasse ein endliches Modell besitzen muß und aus der Menge S heraus keine weiteren Anforderungen an diejenigen Objekte gestellt werden, auf welche ein solcher Slot $\mathtt{p}_1$ verweist, kann $\mathcal{I}$ geeignet erweitert werden, indem A um die entsprechenden Instanzen erweitert wird.

□

Das folgende Theorem folgt unmittelbar aus Lemma 7.3:

Theorem 7.1 *Erfüllbarkeit und Subsumption sind für Formeln entscheidbar.*

Algorithmus 7.1 ist ein Entscheidungsverfahren für die Erfüllbarkeit von Formeln. Wie in Lemma 7.3 gezeigt terminiert er immer und liefert als Ergebnis eine Aussage darüber, ob eine gegebene Formel erfüllbar ist oder nicht.

Das Ergebnis ist korrekt, da die folgenden Eigenschaften gelten:

- Angenommen die Formel ϕ ist erfüllbar. Dann ist wegen der lokalen Korrektheit der Regeln von Algorithmus 7.1 auch eine der aus $S = \{\phi\}$ ableitbaren Mengen S', auf die keine der Regeln mehr anwendbar ist, erfüllbar. Da Formelmengen, die clashs enthalten, unerfüllbar sind, muß S' clash-frei sein.

- Angenommen die Formel ϕ ist unerfüllbar und es gibt eine clash-freie, aus $S = \{\phi\}$ ableitbare Formelmenge S', auf welche keine der Regeln mehr anwendbar ist. Dann ist nach Lemma 7.3 S' erfüllbar. Da aber für alle Regeln $S \rightarrow S'$ entweder $S \subseteq S'$ oder $\mathcal{I} \models S \Leftrightarrow \mathcal{I} \models S'$[15] gilt muß auch S und damit ϕ erfüllbar sein, was im Widerspruch zur Voraussetzung steht.
  Damit enthalten bei unerfüllbarem ϕ alle aus $S = \{\phi\}$ ableitbaren Formelmengen clashs.

Die Entscheidung darüber, ob eine Bedingung D eine Bedingung C erfüllt, d.h. ob $D \Rightarrow C$ gilt, kann entsprechend Lemma 7.1 auf die Entscheidung über die Erfüllbarkeit einer Formel reduziert werden.

7.7 Erweiterung der Vor- und Nachbedingungen um die Behandlung von Konstanten

In diesem Abschnitt wird kurz darauf eingegangen, wie der in den vorigen Abschnitten definierte Formalismus zur Angabe der Vor- und Nachbedingungen erweitert werden kann, um auch den Umgang mit Konstanten, insbesondere den Vergleich der Inhalte verschiedener Slots, zu ermöglichen. In den Beispielen aus Abschnitt 7.5.5 wurde diese Erweiterung bereits verwendet.

Eine Behandlung des erweiterten Formalismus durch Algorithmus 7.1 würde den Rahmen dieser Arbeit sprengen, es kann aber davon ausgegangen werden, daß eine Übertragung der in [BH91a] vorgestellten Technik auf die hier vorliegende Sprache möglich ist. Dies hat eine Erhaltung des in Abschnitt 7.6 erzielten Entscheidbarkeitsresultats auch für den erweiterten Formalismus zur Folge.

Zur Erweiterung der Syntax werden zusätzlich die Mengen K_R und K_B von Variablenbezeichnern definiert, welche untereinander und zur Menge V disjunkt sind. Die Variablen aus K_R bezeichnen reelle Zahlen, diejenigen aus K_B Boole'sche Werte.

Für `k`$\in K_R \cup K_B$, `k`$_R$, `l`$_R \in K_R$ und `k`$_B$, `l`$_B \in K_B$ werden zusätzlich folgende atomare Bedingungen definiert:

- `:eq(o.p,k)`
- θ`(k`$_R$`,l`$_R$`)` für $\theta \in$ {`>`, `<=`, `=`, `!=` }
- θ`(k`$_R$`)` für $\theta \in$ { `>r`, `<=r`, `=r`, `!=r` }, r $\in$ `R`
- `REAL(k`$_R$`)`, `:not REAL(k`$_R$`)`
- θ(k$_B$,l$_B$) für $\theta \in$ { `=B`, `!=B` }

[15] Regel 18

- =TRUE(k_B)
- =FALSE(k_B)
- BOOLEAN(k_B), :not BOOLEAN(k_B)

Die Semantik dieser zusätzlichen Bedingungen wird durch Tabelle 5 informell festgelegt.

Für die Einbindung von Konstanten entsprechend des in [BH91a] vorgestellten Verfahrens ist es erforderlich, daß für jedes der Prädikate auch das negierte Prädikat definiert ist, ein Prädikat zur Beschreibung des gesamten Wertebereiches zur Verfügung steht und das Erfüllbarkeitsproblem für Konjunktionen von Prädikaten entscheidbar ist.

Tabelle 6 stellt die Negationsbeziehungen zwischen den oben definierten Prädikaten dar, der gesamte Wertebereich wird jeweils durch das Prädikat REAL bzw. BOOLEAN bezeichnet.

Ausdruck	**Bedeutung**
:eq(o.p,k)	der Slot o.p referenziert eine Konstante mit dem durch k angegebenen Wert
θ(k,l)	die Konstanten k und l stehen in der durch θ angegebenen Relation zueinander[16]
θr(k)	k liegt in der durch θr bezeichneten Relation[17]
REAL(k), :not REAL(k)	k ist/ist nicht reellwertig
BOOLEAN(k), :not BOOLEAN(k)	k ist/ist nicht vom Typ Boolean

Tabelle 5: Informelle Definition der Semantik der Erweiterungen des Formalismus zur Angabe von Vor- und Nachbedingungen

θ	$\overline{\theta}$
>	<=
=	!=
=r	!=r
>r	<=r
REAL	:not REAL
=B	!=B
=TRUE	=FALSE
BOOLEAN	:not BOOLEAN

Tabelle 6: Prädikate auf konstanten Wertebereichen und ihre Negation.

7.8 Algorithmus zur Zusammenfassung von Vor- und Nachbedingungen in zusammengesetzten Ablaufschritten

In zusammengesetzten Ablaufschritten gibt es "globale" Vor- und Nachbedingungen, nämlich diejenigen des COMPOSITE-STEPs, und "lokale" Vor- und Nachbedingungen der einzelnen Teilschritte.

[16] >, <=, =, != bezeichnen die üblichen Vergleiche von reellen Zahlen, =B, !=B Gleichheit bzw. Ungleichheit von Boole'schen Werten.

[17] >r, <=r, =r, !=r vergleichen den Wert des angegebenen Parameters mit der reellen Zahl r, =TRUE, =FALSE mit den Boole'schen Werten TRUE bzw. FALSE.

Für jede aufgrund der Abfolgebedingungen vorgeschlagene Abfolge *Schritt-1, Schritt-2, Schritt-3, ..., Schritt-k* der Teilschritte muß gelten, daß

- die Vorbedingungen von *Schritt-1* aus den "globalen" Vorbedingungen folgen,
- die Vorbedingungen von *Schritt-i* aus den nach der Durchführung von *Schritt-(i-1)* geltenden Bedingungen folgen und
- die "globalen" Nachbedingungen aus den nach der Durchführung von *Schritt-k* geltenden Bedingungen folgen.

Eine Bedingung C folgt aus einer Bedingung D, wenn D eine Spezialisierung von C ist, d.h. eine Substitution π existiert, so daß für alle Interpretationen $\mathcal{I}$ gilt, daß wenn $\mathcal{I}$ D erfüllt, so erfüllt $\mathcal{I}$ auch $\pi(D)$.

Insbesondere bei automatischen Auswahlschritten sind die Nachbedingungen von der Form $v_1 \vee \cdots \vee v_n$ wobei jedes der Disjunkte im Prinzip den Vorbedingungen eines der alternativen Folgeschritte entspricht und die Formel $v_1 \wedge \cdots \wedge v_n$ unerfüllbar ist. Es wird deutlich, daß für Auswahlschritte die oben aufgeführten Forderungen an die Folgerungsbeziehungen zwischen den Bedingungen der einzelnen Schritte zu streng sind. Für Auswahlschritte fordern wir daher, daß wenn $v_1 \vee \cdots \vee v_m$ die nach seiner Durchführung geltende Bedingung ist für jeden möglichen Folgeschritt gilt, daß seine Vorbedingungen aus einem der v_i folgen.

Die nach der Durchführung eines Schritts geltenden Bedingungen stimmen nicht immer mit den Nachbedingungen des betreffenden Schritts überein, sondern können zusätzlich noch weitere Bedingungen enthalten, nämlich all diejenigen, welche vor der Durchführung des Schritts galten und nicht von dem Schritt verändert wurden.

Ein kleines Beispiel soll diesen Sachverhalt verdeutlichen.

Beispiel 7.5 *Nehmen wir an, daß ein zusammengesetzter Ablaufschritt die Vorbedingung*

$\Phi =$ `(DEVICE-MODEL(model_xy) :or CONNECTION-MODEL(model_xy)) :and MATERIAL-ENTITY(mat_ent)` *habe, wobei* `model_xy` *der Bezeichner eines bestimmten Objektes ist. Der erste Teilschritt des zusammengesetzten Schritts habe die Vorbedingung*

$\Phi_1 =$`DEVICE-MODEL(model_ab) :or CONNECTION-MODEL(model_ab)`

und die Nachbedingung

$\Psi_1 =$`:nonexistent(model_ab).`

Nach der Durchführung des ersten Teilschritts gilt nun

`MATERIAL-ENTITY(mat_ent) :and :nonexistent(model_ab)`

Die Variablen `model_xy` *und* `model_ab` *werden miteinander identifiziert und das zugehörige Objekt von dem Schritt gelöscht, während das durch* `mat_ent` *bezeichnete Objekt von diesem Vorgang unberührt bleibt.*

Beachte: Gleichheitsconstraints (`:eq`), welche vor der Durchführung eines Schritts erfüllt waren, können nicht durch die Nachbedingung des Schritts aufgehoben werden, da durch diese Aufhebung aus einem einzigen Objekt zwei verschiedene Objekte gemacht würden, ohne daß ein neues Objekt durch einen Schritt erzeugt wurde.

Der im folgenden vorgestellte Algorithmus berechnet zu der vor Durchführung eines Schritts gültigen Bedingung Φ und der Nachbedingung Ψ des Schritts die Bedingung $\Upsilon(\Phi, \Psi)$, welche nach der Durchführung des Schritts erfüllt ist.

Dabei wird aus Gründen der Übersichtlichkeit davon ausgegangen, daß auf Ψ bereits die oben erwähnte Substitution π angewandt wurde. Existieren mehrere mögliche Substitutionen, welche die Anforderungen aus der Definition der Spezialisierungsrelation erfüllen, so muß die Durchführbarkeit des Ablaufs für jede der Substitutionen gewährleistet sein, d.h. der im folgenden beschriebene Vorgang muß mehrfach wiederholt werden.

Für die Überprüfung der Durchführbarkeit einer Abfolge müssen also jeweils die die nach den einzelnen Schritten geltenden Bedingungen (Υ) berechnet werden und die Implikationsbeziehungen überprüft werden (vgl. Algorithmus 7.1). Diese Zusammenhänge sind in Abbildung 20 noch einmal dargestellt.

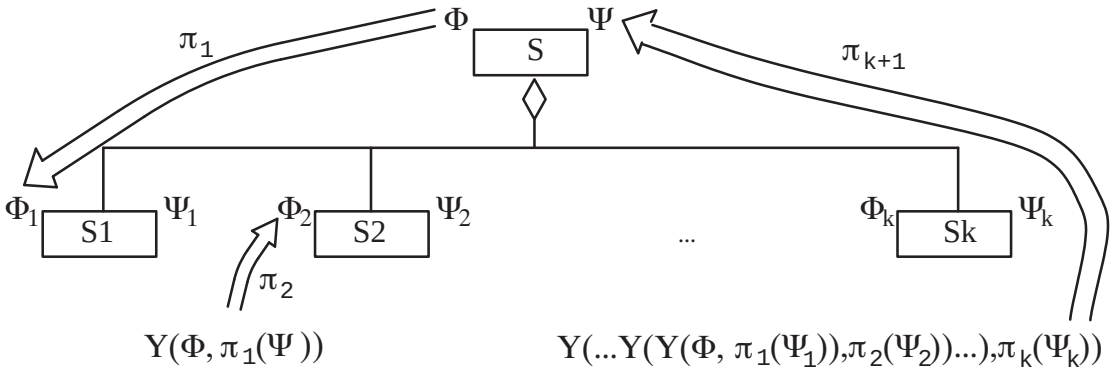

Abbildung 20: Beispiel für einen zusammengesetzten Ablaufschritt (vereinfachte Darstellung)

Seien also nun die Bedingungen Φ und Ψ jeweils in disjunktiver Normalform gegeben. Dabei bezeichnet Φ die vor der Durchführung des betrachteten Schritts geltenden Bedingungen und Ψ die Nachbedingungen des Schritts.

Es muß nun für jede Kombination aus einem Disjunkt ϕ_i von Φ und einem Disjunkt ψ_j von Ψ überprüft werden, welche der Konjunkte aus ϕ_i nicht im Widerspruch zu ψ_j stehen und daher übernommen werden können. Das Ergebnis des im folgenden skizzierten Algorithmus ist eine Bedingung $\Upsilon(\phi_i, \psi_j)$. Die gesuchte Bedingung $\Upsilon(\Phi, \Psi)$ ist dann definiert als $\bigvee_{i,j} \Upsilon(\phi_i, \psi_j)$.

Algorithmus 7.2

Eingabe: *Konjunktionen ϕ_i und ψ_j*
Ausgabe: $\Upsilon(\phi_i, \psi_j)$
Hilfsmengen: B_ϕ *:= Menge der Konjunkte aus* ϕ_i

B_ψ := *Menge der Konjunkte aus* ψ_j
$\Upsilon := \emptyset$
$A := \emptyset$

Folgende Schritte sind in der angegebenen Reihenfolge durchzuführen:

1. *Erweitere* B_ψ *um alle impliziten Folgerungen aus Elementen von* B_ψ. *(Dieser Schritt kann in Anlehnung an die Regeln aus Algorithmus 7.1 definiert werden, ist aber weniger aufwendig, da* ψ_j *keine Negationen enthält.)*

2. *Realisiere alle Gleichheiten, d.h. ersetze für jedes* `:eq(x,y)`, `x,y` $\in X$, *welches in einer der Mengen* B_ϕ *oder* B_ψ *vorkommt sämtliche Vorkommen von* `y` *in beiden Mengen durch* `x` *und lösche die dabei entstehenden Bedingungen der Form* `:eq(x,x)`. *Erweitere die Menge A jeweils um die Bedingung* `:eq(x,y)`.

3. *Wenn* B_psi *einen clash enthält wurde bei der Definition der Bedingungen ein Fehler gemacht und der Algorithmus wird an dieser Stelle mit einem Mißerfolg abgebrochen. In diesem Fall müssen die Definitionen der Bedingungen überarbeitet werden.*

4. *Lösche alle (atomaren) Bedingungen, welche sowohl in* B_ϕ *als auch in* B_ψ *enthalten sind, aus* B_ϕ.

5. *Lösche alle (atomaren) Bedingungen aus* B_ϕ, *welche mit Bedingungen aus* B_ψ *inkompatibel sind (vergleiche Definition 7.9).*

6. *Wenn eine Teilmenge von* B_ϕ *in Verbindung mit* B_ψ *einen clash enthält, streiche alle an diesem clash beteiligten Bedingungen aus* B_ϕ.

7. *Erweitere* B_ψ *um alle Bedingungen, welche noch in* B_ϕ *enthalten sind.*

8. $\Upsilon := \bigwedge_{\psi \in B_\psi} \psi \wedge \bigwedge_{a \in A} a$

Zu diesem Algorithmus ist anzumerken, daß er zum einen nicht exakt definiert wurde (vergleiche insbesondere Schritt 1), zum anderen nicht unbedingt die maximale nach der Durchführung des Ablaufschritts geltende Bedingung liefert. Dies ergibt sich aus der sehr restriktiven Behandlung der clashs in Schritt 6 ergibt. Hier besteht noch der Bedarf einer genaueren Untersuchung. Diese würde allerdings den Rahmen der Arbeit sprengen.

8 Die Spezialisierungsrelation für Ablaufschritte in VeDa

Die in den vorangehenden Kapiteln getroffenen Definitionen und Vereinbarungen ermöglichen zusammen mit dem Entscheidbarkeitsresultat für Formeln aus Abschnitt 7.6 die Definition einer Spezialisierungsrelation für Ablaufschritte in VeDa. Diese muß verschiedene Aspekte eines solchen Schritts berücksichtigen:

- Die Darstellung *eines* Schritts durch mindestens *fünf* verschiedene Objekte, die untereinander in Beziehung stehen,
- die Möglichkeit, zwischen *verschiedenen* Implementierungen für *einen* Ablaufschritt zu wählen,
- die Mächtigkeit der in einem Ablaufschritt zur Verfügung stehenden Methoden, insbesondere die Tatsache, daß diese den Zustand *fremder* Objekte verändern können.

8.1 Vererbung versus Typisierung

Es existieren verschiedene Ansätze zur Spezialisierung von Objektverhalten, vergleiche zum Beispiel [KS94, MD93, WL95]. In diesen Arbeiten wird jeweils das Verhalten eines Objekts betrachtet, dessen Zustand, der durch die Wertebelegung bestimmter Attribute festgelegt ist, durch das Auftreten von Ereignissen verändert wird. Ein Ereignis besteht dabei in der Regel in der Ausführung einer durch das Objekt selbst zur Verfügung gestellten Methode, welche frei von Seiteneffekten ist, d. h. den Zustand anderer Objekte nicht verändert.

Solche Einschränkungen der Mächtigkeit von Methoden sind in dem hier betrachteten Aufgabengebiet nicht möglich. Methoden, welche von VeDa-Objekten zur Verfügung gestellt werden, können durchaus den Zustand anderer Objekte verändern. Als Beispiel sei die Methode `make-active` der Klasse `MODEL` genannt [MBG+97], die zwischen verschiedenen möglichen Modellen einer `MATERIAL-ENTITY` "umschaltet" und dazu die Werte der Slots `possible-alternatives` und `active-alternative` aller alternativen Modelle der betreffenden `MATERIAL-ENTITY` verändert. Neben der in diesem Beispiel gezeigten Veränderung des Zustands fremder Objekte können Methoden in VeDa auch ganze Bausteine erzeugen oder löschen. Es ist daher nicht möglich, die Ergebnisse aus Arbeiten zur Spezialisierung von Objektverhalten auf Ablaufschritte in VeDa zu übertragen.

Grundsätzliche Überlegungen zur Definition von Spezialisierungsrelationen, welche von bestehenden Vererbungsbeziehungen der betreffenden Objekte unabhängig sind, liefert Snyder in [Sny87]. Seine Argumentation für die Trennung von Vererbungs- und Subtyp-Hierarchien kann anhand des folgenden Beispiels nachvollzogen werden.

Beispiel 8.1 *Grundlage dieses Beispiels ist die mathematische Analyse von Gleichungssystemen, die auch in dem im Anhang verwendeten Beispiel auftritt, die verwendeten Schritte sind allerdings leicht modifiziert. Gegeben sind drei verschiedene Ablaufschritte, einer für die* strukturelle Lösbarkeitsanalyse, *einer für die* strukturelle Indexanalyse *und einer für eine kombinierte* strukturelle Analyse von Lösbarkeit und Index.

Diese Schritte können unabhängig voneinander definiert werden, die strukturelle Analyse von Lösbarkeit und Index *kann als Erweiterung eines der beiden anderen Schritte*

definiert werden (Extension Inheritance [Mey]*) oder die Schritte für* strukturelle Lösbarkeitsanalyse *und* strukturelle Indexanalyse *können aus dem Schritt* strukturelle Analyse von Lösbarkeit und Index *durch Streichung der jeweils nicht benötigten Methoden erzeugt werden (Uneffecting Inheritance* [Mey]*).*

Betrachtet man lediglich die Schnittstellen der verschiedenen Ablaufschritte, d.h. die von ihnen zur Verfügung gestellten Dienste, so wird man die strukturelle Analyse von Lösbarkeit und Index *unabhängig von der gewählten Implementierung als eine Spezialisierung der beiden anderen Schritte ansehen, da überall dort, wo ein Schritt zur* strukturellen Lösbarkeitsanalyse *oder zur* strukturellen Indexanalyse *benötigt wird, auch der komplexere Schritt zur* strukturellen Analyse von Lösbarkeit und Index *eingesetzt werden kann.*

Verbindet man mit einem Schritt jedoch zusätzlich zu seinen Funktionen noch weitere Eigenschaften, wie zum Beispiel die Zusicherung, daß bestimmte Werte von ihm nicht verändert werden, so ergibt sich ein anderes Bild. In diesem Fall kann der Schritt zur strukturellen Analyse von Lösbarkeit und Index *nicht mehr unbedingt als Spezialisierung des Schritts zur* strukturellen Indexanalyse *angesehen werden, da letzterer die früheren Ergebnisse der strukturellen Lösbarkeit nicht verändert, während der erste Schritt dies eventuell tut.*

Aufbauend auf diesen Überlegungen wird die Spezialisierungsrelation für Ablaufschritte in VEDA unabhängig von der Vererbungshierarchie für die entsprechenden Klassen definiert. Neben einer Spezialisierungsrelation für Ablaufschritte, welche aus Instanzen verschiedener Klassen zusammengesetzt werden, werden analoge Definitionen für die Spezialisierung der einzelnen "Bausteine", also `PROCESS-FRAGMENT, STEP, INPUT-` und `OUTPUT-SITUATION`, angegeben.

Obwohl die hier definierten Relationen dazu dienen, verschiedene Klassen, welche aufgrund ihrer Vererbungsbeziehungen in der VEDA-Taxonomie angeordnet sind, in eine neue Hierarchie einzuordnen, ist es wichtig zu beachten, daß sie nicht dazu dienen, die Vererbungsbeziehungen nachträglich zu verändern. Die Eintragung neuer Klassen in den Slot `superclasses` könnte zur Folge haben, daß eine Klasse zusätzliche Attribute erbt, die sie vor dieser Veränderung nicht besaß. Eine solche Verwendung der Spezialisierungsrelation ist deshalb nicht zu empfehlen.

Bei der Definition der Spezialisierungsrelation in VEDA werden zusätzlich zur Signatur, das heißt den zur Verfügung gestellten Methoden und Attributen, noch Objekteigenschaften in Form von Vor- und Nachbedingungen in Betracht gezogen.

Grundlage der Definition sind die von America [Ame90] und Liskov und Wing [LW93] formulierten Bedingungen für die Subtyp-Relation.

Nach America [Ame90] ist σ ein Subtyp von τ ($\sigma < \tau$), wenn folgende Bedingungen gelten:

- Die Objekteigenschaften von τ sind eine Teilmenge der Objekteigenschaften von σ (Objekteigenschaften sind Eigenschaften, die das Zusammenspiel von zwei oder mehr Methoden des Objekts beschreiben, beispielsweise die FIFO-Eigenschaft eines Speichers.).
- für jede Methode m_τ in τ existiert eine entsprechende Methode m_σ in σ, so daß:
  - m_τ und m_σ haben den gleichen Namen.

- m_τ und m_σ haben die gleiche Anzahl Parameter.
- Der i-te Parameter von m_τ ist ein Subtyp des i-ten Parameters von m_σ (Kontravarianz).
- Entweder haben sowohl m_τ als auch m_σ einen Ergebnistyp oder keine der Prozeduren hat einen Ergebnistyp.
- Wenn es einen Ergebnistyp gibt, dann ist das Ergebnis von m_σ ein Subtyp des Ergebnisses von m_τ (Kovarianz).
- Die Methodeneigenschaften von m_τ sind eine Teilmenge der Methodeneigenschaften von m_σ (unter Methodeneigenschaften verstehen wir Zusicherungen an eine bestimmte Methode, z.B. daß die Methode den Wert eines bestimmen Attributs nicht verändert).

Liskov und Wing [LW93] stellen eine modifizierte Definition dieser Subtyp-Relation vor. Die wesentlichen Unterschiede sind:

- m_τ und m_σ können unterschiedliche Namen haben,
- jeder Methode werden Vor- und Nachbedingungen zugeordnet und für korrespondierende Methoden m_τ und m_σ muß gelten, daß die Vorbedingungen von m_σ schwächer sind als die von m_τ und die Nachbedingungen von m_τ schwächer sind als die von m_σ,
- zusätzliche Methoden dürfen in σ nur eingeführt werden, wenn ihre Funktion durch die Methoden aus τ beschrieben werden kann.

Ein Problem, welches bei der Definition der Subtyprelation auftritt ist, daß die Objekt- und Methodeneigenschaften nur in Form von "Markierungen" (z.B. textuellen Beschreibungen) angegeben werden, welche jedoch nicht überprüft werden können.

America betrachtet nur Objekte, deren Methoden lediglich auf dem Objekt selbst, nicht aber auf anderen Objekten operieren. Die Veränderung *fremder* Objekte ist aber für die hier betrachteten Ablaufschritte der Regelfall. Es wird daher nötig sein, obige Definition in einigen Punkten zu modifizieren.

8.2 Annahmen und Einschränkungen

Bevor nun eine Definition für eine Spezialisierungsrelation für Ablaufschritte in VeDa angegeben wird, werden in diesem Abschnitt noch einmal alle Annahmen und Einschränkungen, welche im Verlauf der Arbeit getroffen wurden, zusammengefaßt. Dazu gehören auch solche Restriktionen, welche zum Beispiel aus der Definition des Formalismus für die Vor- und Nachbedingungen oder dem generellen Aufbau einer VeDa-Klasse resultieren. Aus diesen Annahmen und Einschränkungen ergeben sich Richtlinien für die Definition von Ablaufschritten.

8.2.1 Zusammenfassung der Annahmen und Einschränkungen

Die wichtigste Annahme, welche über die VeDa-Klassentaxonomie getroffen wurde, ist, daß sie fest ist. Eine Klasse, welche einmal in die VeDa-Taxonomie eingeordnet wurde, behält ihren Platz in dieser Taxonomie bei. Diese Voraussetzung wird von dem

Algorithmus zur Entscheidung der Erfüllbarkeit einer Bedingung (vergleiche Abschnitt 7.6) ausgenutzt, da ansonsten aus der Beziehung "D Subklasse von C" nicht geschlossen werden könnte, daß jede Instanz von D immer auch Instanz von C ist.

Da eine Veränderung der VEDA-Klassentaxonomie die Implikationsbeziehungen zwischen Bedingungen beeinflussen kann, müssten in diesem Fall die Anordnung der Klassen zur Beschreibung von Modellierungsabläufen in Vererbungs- und Subtyphierarchien, die Zulässigkeit bestehender Definitionen von Ablaufschritten sowie die Durchführbarkeit bereits definierter Abläufe überprüft werden.

Eine weitere Voraussetzung für die korrekte Funktion des in Abschnitt 7.6 vorgestellten Algorithmus ist, daß jedes Objekt in VEDA genau eine ihm direkt zugeordnete Klasse besitzt, deren Instanz es ist, d. h. es gibt in VEDA keine multiple Instantiierung.

Für alle Klassen, die nicht zur Beschreibung von Ablaufschritten verwendet werden, wird eine Subklasse immer als eine Spezialisierung ihrer Superklasse angesehen, das heißt es wird davon ausgegangen, daß Instanzen einer Subklasse alle Bedingungen erfüllen, welche auch an Instanzen ihrer Superklasse gestellt werden können. Es wird also aus der Vererbung auf die Einsetzbarkeit geschlossen. Diese Annahme entspricht leider nicht immer der Realität, denn obwohl in VEDA angestrebt wird, daß die Vererbungshierarchie einer Subtyphierarchie entspricht, kann dies nicht garantiert werden, da in Subklassen zum Beispiel Restriktionen und Methoden verändert werden können [MBG+97], so daß Instanzen eine Subklasse *nicht* immer anstelle einer Instanz ihrer Superklasse eingesetzt werden kann. Es liegt somit in den Händen des Metamodellierers, darauf zu achten, daß Subklassen tatsächlich immer spezieller sind, sofern auf diese Beziehung in den Vor- und Nachbedingungen eines Ablaufschritts zurückgegriffen wird.

Weiterhin werden Restriktionen (`laws`), welche an einige Klassen gekoppelt sind und zum Beispiel Einschränkungen bezüglich der möglichen Werte einzelner Slots vornehmen, außer Betracht gelassen, da die Sprache zur Angabe dieser Regeln die Mächtigkeit der Prädikatenlogik erster Stufe besitzt [MBG+97] und die Erfüllbarkeit der Restriktionen daher im allgemeinen unentscheidbar ist (vergleiche z.B. [FGS90]).

Bei der Überprüfung der Vor- und Nachbedingungen werden gleiche Variablenbezeichner mit gleichen Objekten identifiziert, das heißt wenn eine Variable X sowohl in der Vor- als auch in der Nachbedingung eines Schritts[18] vorkommt, bezieht sie sich in beiden Fällen auf das gleiche Objekt. Dies gilt auch für geerbte Bedingungen, das heißt wenn eine Klasse eine Bedingung, welche die Variable X enthält, erbt und selbst eine zusätzliche Bedingung für X definiert, so bezeichnet X wiederum in beiden Regeln das gleiche Objekt. Diese Vereinbarung bezieht sich auf alle Klassen, die zu ein und demselben Schritt gehören, das heißt die Variable X in der `INPUT-SITUATION` bezeichnet das gleiche Objekt wie die Variable X in `PROCESS-FRAGMENT, STEP` und `OUTPUT-SITUATION` des gleichen Schritts. In Klassen, welche nichts mit dem betrachteten Schritt zu tun haben, kann X auch ein anderes Objekt bezeichnen. Durch diese Konvention wird sichergestellt, daß ein Bezug zwischen den Vor- und Nachbedingungen eines Schritts aufgebaut werden kann, verschiedene Schritte aber voneinander unabhängig definiert werden können.

Für jeden einzelnen Ablaufschritt müssen in den Nachbedingungen neben den erst nach der Durchführung des Schritts geltenden Bedingungen auch diejenigen Bedingungen aus der Menge der Vorbedingungen angegeben werden, welche "immer noch" gelten, da

[18] Ein Schritt besteht aus Instanzen verschiedener Klassen, nämlich `PROCESS-FRAGMENT, STEP, INPUT-` und `OUTPUT-SITUATION` und `GOAL`

diese nicht aus den Vorbedingungen des Schritts abgeleitet werden können (vergleiche Beispiel 7.2). In den Nachbedingungen dürfen keine Variablen verwendet werden, welche nicht auch in den Vorbedingungen auftreten. Die letzte Bedingung stellt sicher, daß sich die Nachbedingungen eines Schritts immer auf seine eigenen Vorbedingungen beziehen und damit eine eindeutige Zuordnung der Variablen möglich ist. Anderenfalls wäre zum Beispiel unklar, auf welches Objekt sich die Nachbedingung `:nonexistent(x)` bezieht, wenn die Variable `x` in den Vorbedingungen des Schritts nicht vorkommt.

Aus der Annahme heraus, daß kein Schritt Änderungen an Objekten vornimmt, welche nicht in seinen Vor- und Nachbedingungen dokumentiert sind, ergibt sich unmittelbar, daß sämtliche Bedingungen, welche vor der Durchführung eines Schritts erfüllt waren und nicht mit seinen Vorbedingungen in Relation stehen auch nach der Durchführung des Schritts noch gelten. Diese Vereinbarung entspricht im Prinzip der *STRIPS-Assumption* (vgl. Abschnitt 3.3.3). Sie ist für die Definition der Spezialisierungsrelation für Ablaufschritte in VEDA nicht relevant, spielt aber bei der Überprüfung der Durchführbarkeit einer Schrittfolge eine wesentliche Rolle.

Im Fall der Aktionen elementarer Ablaufschritte kann aufgrund der Tatsache, daß eine Aktion in der Regel eine externe Methode sein wird, nicht überprüft werden, ob eine Aktion spezieller ist als eine andere. Da aber die Auswirkungen der Methode in den Vor- und Nachbedingungen des betreffenden Schritts dokumentiert sind, ist eine solche Überprüfung auch nicht notwendig. Es wird deshalb bei der Definition der Spezialisierungsrelation für Ablaufschritte in VEDA auf die Einbeziehung der Methoden in diese Definition vollständig verzichtet.

8.2.2 Richtlinien für die Definition von Abläufen

Aus den im vorigen Abschnitt zusammengefaßten Annahmen und Einschränkungen ergeben sich folgende Richtlinien, welche bei der Definition von Abläufen beachtet werden müssen, um eine sinnvolle Spezialisierungsrelation zu gewährleisten:

- Sämtliche Forderungen, welche an eine Instanz der Klasse C gestellt werden, müssen auch von Instanzen von Subklassen von C erfüllt werden, sofern C in den Vor- oder Nachbedingungen eines Schritts verwendet wird.
- Alle Vorbedingungen, welche auch nach der Durchführung des Schritts erfüllt sind, müssen explizit in den Nachbedingungen des Schritts wiederholt werden.
- Alle von einem Schritt bewirkten Veränderungen müssen ausdrücklich in seinen Vor- und Nachbedingungen dokumentiert werden.

8.3 Definition der Spezialisierungsrelation

Ein Ablaufschritt in VEDA wird durch die zu seiner Darstellung benötigten Klassen bzw. deren Instanzen, im einzelnen `PROCESS-FRAGMENT`, `STEP`, `INPUT-SITUATION`, `OUTPUT-SITUATION` und `GOAL`, bestimmt. Wenn im folgenden von einer Spezialisierungsrelation für *Schritte* oder *Ablaufschritte* gesprochen wird bezieht sich diese Formulierung dementsprechend immer auf eine solche Menge von Klassen.

Da es eines der wesentlichen Ziele der zu definierenden Spezialisierungsrelation ist, die allgemeine Einsetzbarkeit eines spezielleren für einen allgemeineren Schritt zu garantie-

ren, müssen die in Abbildung 21 dargestellten Beziehungen zwischen den Vorbedingungen ϕ und den Nachbedingungen ψ der Schritte σ und τ gelten. Sie besagen, daß ein Schritt σ spezieller (und damit einsetzbar) ist, wenn seine Vorbedingungen aus denen des allgemeineren Schritts τ folgen und er nach seiner Durchführung all das garantiert, was auch τ garantiert.

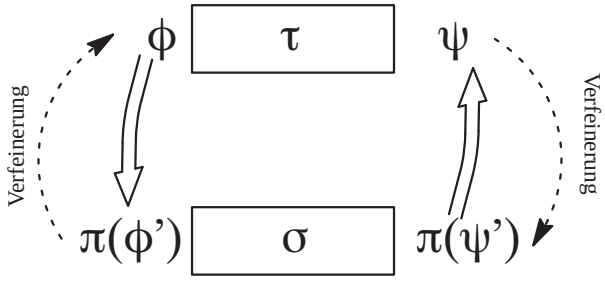

Abbildung 21: Graphische Darstellung der Spezialisierungsrelation für Ablaufschritte

Im folgenden wird der Begriff *Subklasse* entsprechend der Definition der Vererbungsbeziehung in [MBG+97] (vergleiche auch Abschnitt 4.4) verwendet. Zusätzlich zu den dort angegebenen Regeln gilt für geerbte Vor- und Nachbedingungen, daß diese in der Subklasse lediglich im Sinne einer Verfeinerung verändert werden dürfen.

Die folgenden Definitionen legen fest, wann eine Klasse als Subtyp einer anderen gilt. Für die Subklassen von `SITUATION`, `PROCESS-FRAGMENT` und `STEP` ist dabei die Motivation für die Definition der Subtyphierarchie, daß ein Subtyp anstelle des Supertyps in der Definition eines Ablaufschritts eingesetzt werden kann. Für die Definition der Subtyprelation für `INPUT-SITUATION`s ist daher beispielsweise der Wertebereich des Slots `occurs-in`, welcher das zugehörige `PROCESS-FRAGMENT` referenziert, irrelevant, da er sich aus dem Schritt ergibt, in den die `INPUT-SITUATION` "eingebaut" werden soll. Weiterhin würde die Berücksichtigung dieser Slots eine zyklische Definition der Subtyprelation zur Folge haben.

Definition 8.1 (Subtyp) *Eine Klasse σ heißt* Subtyp *einer Klasse τ, wenn für jede Methode m_τ in τ eine entsprechende Methode m_σ in σ existiert, so daß:*

- *m_τ und m_σ haben den gleichen Namen.*
- *m_τ und m_σ haben die gleiche Anzahl Parameter.*
- *Der i-te Parameter von m_τ ist ein Subtyp des i-ten Parameters von m_σ (Kontravarianz).*
- *Entweder haben sowohl m_τ als auch m_σ einen Ergebnistyp oder keine der Prozeduren hat einen Ergebnistyp.*
- *Wenn es einen Ergebnistyp gibt, dann ist das Ergebnis von m_σ ein Subtyp des Ergebnisses von m_τ (Kovarianz).*
- *Die Methodeneigenschaften von m_τ sind eine Teilmenge der Methodeneigenschaften von m_σ (unter Methodeneigenschaften verstehen wir Zusicherungen an eine*

bestimmte Methode, z.B. daß die Methode den Wert eines bestimmen Attributs nicht verändert).

und außerdem eine der folgenden Bedingungen erfüllt ist:

- *sowohl σ als auch τ Subklassen von* `INPUT-SITUATION` *sind, die Vorbedingungen von τ Verfeinerungen der Vorbedingungen von σ sind und für alle Restriktionen und Attribute außer dem Verweis auf das zugehörige* `PROCESS-FRAGMENT` *die Vererbungsbedingungen aus Abschnitt 4.4 für die Beziehung "σ Subklasse von τ" erfüllt sind,*
- *sowohl σ als auch τ Subklassen von* `OUTPUT-SITUATION` *sind, die Nachbedingungen von σ Verfeinerungen der Nachbedingungen von τ sind und für alle Restriktionen und Attribute außer dem Verweis auf den zugehörigen* `STEP` *die Vererbungsbedingungen aus Abschnitt 4.4 für die Beziehung "σ Subklasse von τ" erfüllt sind,*
- *sowohl σ als auch τ Subklassen von* `PROCESS-FRAGMENT` *sind, die Nachbedingungen von σ Verfeinerungen der Nachbedingungen von τ sind, die* `INPUT-SITUATION` *von σ ein Subtyp der* `INPUT-SITUATION` *von τ ist und für alle Restriktionen und Attribute außer den Verweisen auf* `STEP`*s die Vererbungsbedingungen aus Abschnitt 4.4 für die Beziehung "σ Subklasse von τ" erfüllt sind,*
- *sowohl σ als auch τ Subklassen von* `STEP` *sind, die Vorbedingungen von τ Verfeinerungen der Vorbedingungen von σ sind und für alle Restriktionen und Attribute außer dem Verweis auf das zugehörige* `PROCESS-FRAGMENT` *die Vererbungsbedingungen aus Abschnitt 4.4 für die Beziehung "σ Subklasse von τ" erfüllt sind oder*
- *σ und τ nicht beide Subklassen der gleichen Klasse aus der Menge* `INPUT-SITUATION, PROCESS-FRAGMENT, STEP` *sind und für alle Restriktionen und Attribute die Vererbungsbedingungen aus Abschnitt 4.4 für die Beziehung "σ Subklasse von τ" erfüllt sind.*

Für Subklassen von `OUTPUT-SITUATION` gilt dementsprechend, daß die Vererbungshierarchie nahezu identisch zur Subtyp-Hierarchie aufgebaut ist, während sich für die Subklassen von `INPUT-SITUATION` eine der Vererbungshierarchie entgegengesetzte Relation ergibt. Diese Definitionen ergeben sich aus der Forderung nach der Einsetzbarkeit eines Subtyps anstelle des Supertyps, da dabei die Vorbedingungen schwächer, die Nachbedingungen aber strenger werden dürfen.

Die rekursive Definition der Subtyprelation gestaltet sich in VeDa insofern schwierig, als eine Trennung von Subtyp- und Vererbungshierarchie im Sinne von Snyder zur Zeit nicht vorgesehen ist. Für alle Klassen, die nicht Subklasse von `ACTIVITY-CONCEPT` sind, wird daher wie oben definiert die Subtyp- und die Vererbungshierarchie gleichgesetzt. Diese Vereinfachung erspart es uns, eine Subtyphierarchie für alle VeDa-Klassen zu definieren (die aufgrund der Mächtigkeit der als Methoden verwendeten C-Programme und der Restriktionen auch gar nicht überprüft werden könnte), führt aber gleichzeitig zu Unsauberkeiten in der Spezialisierungsrelation für Ablaufschritte, da zum Beispiel Methoden aufgrund ihrer Signatur als spezieller angesehen werden, welche es bei Anwendung einer strengen Subtyprelation nicht sind.

Die Subtypbeziehung liefert ein hinreichendes Kriterium dafür, wann eine einzelne Klasse in der Definition eines Ablaufschritts durch eine andere ersetzt werden kann. Das Kriterium ist jedoch nicht notwendig, da `ELEMENTARY-STEP`s und `COMPOSITE-STEP`s nicht

zueinander in einer Subtyprelation stehen können (sie besitzen deutlich verschiedene Attribute und Methoden), es jedoch trotzdem möglich sein kann, in einem gegebenen Ablaufschritt eine dieser Klassen durch die andere zu ersetzen.

Nun kann der Begriff der Spezialisierung für ganze Ablaufschritte definiert werden.

Definition 8.2 (Spezialisierung von Ablaufschritten) *Ein Ablaufschritt σ ist* spezieller *als ein Ablaufschritt τ, wenn folgende Bedingungen erfüllt sind:*

1. *Es gibt mindestens eine Substitution π, so daß die in der* `INPUT-SITUATION` *von τ angegebenen Vorbedingungen eine Verfeinerung der in der* `INPUT-SITUATION` *von σ angegebenen Vorbedingungen unter dieser Substitution sind*[19].
2. *Für alle Substitutionen π, für die 1. erfüllt ist, gilt, daß die im* `PROCESS-FRAGMENT` *von σ angegebenen Nachbedingungen Verfeinerungen der im* `PROCESS-FRAGMENT` *von τ angegebenen Nachbedingungen unter dieser Substitution π sind.*

Bedingung 1 stellt sicher, daß der Schritt σ immer dann ausführbar ist, wenn τ ausführbar ist.

Bedingung 2 sorgt dafür, daß alle Schritte, welche nach der Durchführung von Schritt τ ausführbar sind, auch nach der Durchführung von Schritt σ ausführbar sind[20]. Werden in den Vor- und Nachbedingungen von τ gleiche Bezeichner verwendet, so müssen an den entsprechenden Stellen in den Vor- und Nachbedingungen von σ ebenfalls gleiche Bezeichner verwendet werden. Dies wird durch die Forderung der Gültigkeit für alle Substitutionen, für die Bedingung 1 gilt, erreicht.

Es ist zu beachten, daß hier die im `PROCESS-FRAGMENT` angegebenen Nachbedingungen der Schritte miteinander verglichen werden, welche jeweils das Minimum der Nachbedingungen der möglichen Implementierungen bilden. Dies ist sinnvoll, da zum Zeitpunkt der Definition von Abläufen meist nicht bekannt ist, welche der möglichen Implementierungen bei der späteren Durchführung des Ablaufs aktiv sein wird.

Angaben über die Beziehungen zwischen Attributen und Methoden der beteiligten Klassen brauchen nicht getroffen zu werden, da die Subtyprelation für Ablaufschritte Aussagen über die Einsetzbarkeit eines Schritts in eine gegebene Abfolge ermöglichen soll, die Attribute verschiedener Schritte einer Abfolge aber unabhängig voneinander sind (d.h. die Schritte verweisen nicht gegenseitig aufeinander).

Auf die Einbeziehung der `GOAL`s in die Definition der Spezialisierungsrelation wurde verzichtet, da Ziele bisher nur informell, das heißt in Form von beschreibenden Texten, angegeben werden können. Daher ist eine Aussage darüber, ob ein Ziel spezieller ist als ein anderes, im allgemeinen nicht möglich.

[19] Dies entspricht der Definition der Spezialisierungsbeziehung zwischen Bedingungen.

[20] Dies gilt bei einer "lokalen" Betrachtung. Innerhalb eines komplexeren Ablaufs könnten durch den eingesetzten Schritt die entsprechend Algorithmus 7.1 "weitergereichten" Bedingungen sich ändern

9 Zusammenfassung und Ausblick

In diesem Abschnitt werden die Ergebnisse der Arbeit noch einmal zusammengefaßt und anschließend einer kritischen Bewertung unterzogen. Dabei wird sich zeigen, daß die Spezialisierungsrelation für Ablaufschritte in VeDa, wie sie in dieser Arbeit definiert wurde, in der Praxis zu einer extrem flachen Hierarchie der Ablaufschritte führen wird. Ein Teil dieses Abschnitts wird daher der Frage gewidmet sein, wie die gewonnenen Erkenntnisse eingesetzt werden können, um trotzdem die gewünschten Ergebnisse, im wesentlichen Unterstützung der Wiederverwendung bereits definierter Ablaufschritte und Hilfe bei der Konstruktion zusammengesetzter Ablaufschritte, zu erzielen.

9.1 Zusammenfassung

Ziel der vorliegenden Arbeit ist die Definition einer Spezialisierungsrelation für Ablaufschritte im objekt-orientierten Datenmodell VeDa. Diese Aufgabenstellung wurde in Abschnitt 1 motiviert, die wesentlichen Gründe für die Definition der Relation sind:

- Die Unterstützung der Wiederverwendung von Ablaufschritten durch eine Verbesserung der Wiederauffindbarkeit und
- die Entscheidung über die Austauschbarkeit einzelner Schritte in einer bereits definierten Schrittfolge.

Die Entscheidung darüber, ob in einer gegebenen Abfolge von Schritten ein Schritt durch einen anderen ausgetauscht werden kann setzt voraus, daß die Durchführbarkeit einer Schrittfolge überprüft werden kann.

Nach einem kurzen Überblick über die Gebiete der verfahrenstechnischen Prozeßmodellierung und der Ablaufmodellierung in den Abschnitten 2 und 3 wurden in Abschnitt 4 zunächst die wichtigsten Merkmale des Datenmodells VeDa vorgestellt. Abschnitt 5 war der Beschreibung der für die Ablaufmodellierung in VeDa benötigten Klassen gewidmet.

In Abschnitt 6 wurden die Abfolgebedingungen definiert, welche es ermöglichen, eine Reihenfolge, in der Teilschritte eines zusammengesetzten Ablaufschritts ausgeführt werden können, vorzuschlagen. Es werden verschiedene Arten von Einschränkungen, welche sich auf die Durchführungsreihenfolge der Teilschritte auswirken, unterschieden:

1. Restriktionen, welche sich aus den Vor- und Nachbedingungen der einzelnen Teilschritte ergeben. Diese Einschränkungen können vom Modellierer nicht umgangen werden, sie ergeben sich aus der Definition der einzelnen Schritte.
2. Restriktionen, welche vom Metamodellierer explizit angegeben werden, aber vom Modellierer jederzeit verletzt werden dürfen. Diese Einschränkungen sind *nicht* erzwungen, sie sind Vorschlag für eine auf jeden Fall durchführbare Abfolge der Teilschritte.

In Abschnitt 7 wurde ein Formalismus zur Formulierung von Vor- und Nachbedingungen für einzelne Ablaufschritte definiert, welcher die besonderen Erfordernisse der Modellierung verfahrenstechnischer Modellierungsabläufe in VeDa berücksichtigt.

Aus der Darstellung von Ablaufschritten durch Instanzen verschiedener Klassen und den zwischen den Vor- und Nachbedingungen eines Schritts bestehenden Beziehungen ergibt sich die Notwendigkeit, einen Formalismus zu entwickeln, welcher entsprechend "maßgeschneiderte" Ausdrucksmöglichkeiten bietet. Ein wesentliches Mittel zur Erreichung dieses Ziels ist die Vereinbarung, daß innerhalb einer Klasse gleiche Bezeichner gleiche Objekte repräsentieren. Das Erzeugen und Löschen von Objekten durch einen Schritt kann durch die Verwendung des speziellen Prädikats `:nonexistent` ausgedrückt werden, welches für einen Bezeichner aussagt, daß das durch ihn bezeichnete Objekt nicht existiert. Während die Nachbedingung `DEVICE-MODEL(x)` für sich genommen nur ausdrückt, daß ein Objekt vom Typ `DEVICE-MODEL` existiert, erhält sie in Verbindung mit der Vorbedingung `:nonexistent(x)` die Bedeutung, daß während der Durchführung des zugehörigen Schritts ein solches Objekt neu erzeugt wurde.

Die wichtigste Anforderung an den Formalismus zur Darstellung von Vor- und Nachbedingungen ist die Entscheidbarkeit der Erfüllbarkeit. Diese wurde in Abschnitt 7.6 gezeigt und ermöglicht die Entscheidung darüber, ob eine Bedingung spezieller ist als eine andere, d.h. die andere impliziert. Dabei werden zwei Arten von Implikation unterschieden: Bei der Verfeinerung dürfen Variablen nicht umbenannt (substituiert) werden, während die Spezialisierung derart definiert ist, daß eine Bedingung D spezieller ist als eine Bedingung C, wenn es mindestens eine Substitution π gibt, so daß D eine Verfeinerung von $\pi(C)$ ist.

Damit war es möglich, die Spezialisierungsrelation in Abschnitt 8 so zu definieren, daß die Einsetzung eines spezielleren Schritts anstelle eines allgemeineren in einen gegebenen Ablauf möglich ist[21]. Abbildung 21 zeigt die Beziehungen zwischen den Vor- und Nachbedingungen eines Schritts und seiner Spezialisierung.

Bei der Definition von Ablaufschritten muß darauf geachtet werden, daß die Vorbedingungen der `INPUT-SITUATION` Verfeinerungen der Vorbedingungen der `STEP`s und die Nachbedingungen der `OUTPUT-SITUATION`s Verfeinerungen der Nachbedingungen des `PROCESS-FRAGMENT`s sind. In diesem Fall dürfen Variablen nicht substituiert werden, da innerhalb eines Ablaufschritts gilt, daß gleiche Bezeichner das gleiche Objekt repräsentieren.

Bei der Definition zusammengesetzter Ablaufschritte kann aufgrund der Vor- und Nachbedingungen der Teilschritte festgestellt werden, ob eine bestimmte Ausführungsreihenfolge der Teilschritte überhaupt durchführbar ist.

Die Vor- und Nachbedingungen für Ablaufschritte und die Entscheidbarkeit der Erfüllbarkeit kommen also in drei verschiedenen Bereichen zur Anwendung:

- Bei der Überprüfung, ob ein Ablaufschritt korrekt definiert wurde,
- bei der Entscheidung darüber, ob ein Ablaufschritt spezieller ist als ein anderer und
- bei der Überprüfung, ob ein zusammengesetzter Ablaufschritt bei einer vorgeschlagenen Reihenfolge der Teilschritte durchführbar ist.

Für die Definition der Spezialisierungsrelation wurden verschiedene Annahmen getroffen, die in der Realität nicht immer erfüllt oder überprüfbar sind. Die wesentlichsten sind

[21] Auf Probleme bei der Einsetzung wird im folgenden Abschnitt eingegangen.

- Die VeDa-Klassentaxonomie ist fest und entspricht für alle Klassen, welche nicht Subklassen von `ACTIVITY-CONCEPT` sind, einer Subtyphierarchie.
- In VeDa findet keine multiple Instantiierung statt.
- Alle von einem Ablaufschritt an Objekten vorgenommenen Änderungen sind in seinen Vor- und Nachbedingungen dokumentiert.

9.2 Bewertung der Ergebnisse

Wie die Betrachtung einiger Beispielschritte gezeigt hat, kommt es bei einander ähnlichen Ablaufschritten in der Praxis selten vor, daß diese sich im Sinne der in dieser Arbeit definierten Spezialisierungsrelation zueinander in Beziehung setzen lassen. Meist wird einer der Schritte sowohl restriktivere Vorbedingungen als auch restriktivere Nachbedingungen besitzen. Dadurch ergibt sich eine sehr flache Spezialisierungshierarchie für vollständige Ablaufschritte. Dies hat zur Folge, daß insbesondere das Ziel der Unterstützung bei der Suche nach vorhandenen Schritten durch die Spezialisierungsrelation nicht in der erwarteten Form erreicht wird.

Allerdings ergeben sich besser strukturierte Vererbungshierarchien für die einzelnen Komponenten der Schritte, also für die Subklassen von `SITUATION, PROCESS-FRAGMENT` und `STEP`, da in den `SITUATION`s jeweils nur eine Vor- oder eine Nachbedingung enthalten ist und die Vererbungsbedingungen für `PROCESS-FRAGMENT`s und `STEP`s zur Folge haben, daß sowohl die Vor- als auch die Nachbedingungen einer Subklasse restriktiver werden als die der Superklasse. Diese Vererbungshierarchien können für die Suche nach einzelnen Komponenten von Ablaufschritten verwendet werden.

Die Frage nach der Einsetzbarkeit eines Schritts in eine gegebenen Schrittfolge wird sich im allgemeinen Fall nicht sinnvoll anhand der Spezialisierungshierarchie für Ablaufschritte beantworten lassen, da die Vor- und Nachbedingungen eines Teilschritts in einer Abfolge nicht isoliert betrachtet werden können, sondern im Zusammenhang mit den Vor- und Nachbedingungen der übrigen Teilschritte gesehen werden müssen.

Es wird in vielen Fällen so sein, daß ein Schritt in einer gegebenen Abfolge durch einen anderen ersetzt werden kann, obwohl er nicht zu diesem in einer Spezialisierungsbeziehung steht. Dies ergibt sich daraus, daß in einer Abfolge jeder einzelne Schritt durch seine Durchführung nur einen Teil der geltenden Bedingungen verändert und es somit möglich ist, daß deutlich mehr Bedingungen erfüllt sind als durch den ursprünglich in der Folge vorhandenen Schritt in seinen Vorbedingungen angegeben wurden. Betrachtet man bei der Einsetzung eines neuen Schritts also den gesamten Schritt im Zusammenhang, kann dieser Schritt evtl. trotz restriktiverer Vorbedingungen in die Folge eingesetzt werden.

Weiterhin kann ein Schritt, der in eine durchführbare Folge von Ablaufschritten anstelle eines weniger speziellen Schritts eingesetzt wird, dazu führen, daß die Folge anschliessend nicht mehr durchführbar ist. Dies kann dann auftreten, wenn die zusätzlichen Nachbedingungen des eingesetzten Schritts Nachbedingungen von Schritten, welche in der Abfolge früher durchgeführt werden, überschreiben, die von dem ursprünglichen Schritt nicht beeinflußt wurden. Die Entscheidung über die Einsetzbarkeit wird dementsprechend immer mit Hilfe des in Abschnitt 7.8 vorgestellten Algorithmus getroffen werden müssen.

Ein ähnlicher Effekt wie der oben für vollständige Ablaufschritte beschriebene tritt

auch bei der Spezialisierung einzelner Klassen zur Beschreibung von Ablaufschritten auf. Die Spezialisierungsrelation stellt zwar sicher, daß eine speziellere Klasse anstelle einer allgemeineren in einen bereits definierten Ablaufschritt "eingebaut" werden kann, sie liefert aber nur eine hinreichende und keine notwendige Bedingung für solche Ersetzungen. Zum Beispiel können `ELEMENTARY-STEP`s und `COMPOSITE-STEP`s aufgrund ihrer unterschiedlichen Attribute und Methoden grundsätzlich nicht zueinander in einer Spezialisierungsrelation stehen, obwohl es durchaus sein kann, daß beide in einem gegebenen Ablaufschritt eingesetzt werden können.

9.3 Ausblick

Da die Definition des Datenmodells VeDa noch nicht endgültig abgeschlossen ist können sich in Zukunft noch Änderungen ergeben, die auch Einfluß auf die hier vorgestellten Formalismen und Algorithmen haben. Diese können zu zusätzlichen Annahmen und Einschränkungen, aber auch zu Änderungen an den Algorithmen selbst führen. Auch eine Erweiterung des Formalismus zur Angabe von Vor- und Nachbedingungen um zusätzliche Konstrukte wäre denkbar.

Zur Menge der wünschenswerten Erweiterungen gehört unter anderem die Möglichkeit, Vor- und Nachbedingungen eines Schritts, welche jeweils mehrere durch Disjunktionen verknüpfte Bedingungen enthalten, einander zuzuordnen. Damit wäre es beispielsweise möglich, die Bedingung *"es wurde eine Gleichung zugefügt oder eine Variable gelöscht"* zu formulieren.

Die Vorbedingung des betreffenden Schritts würde dann zum Beispiel lauten

```
(1| :nonexistent(eq)) :or (2| :variable(var)),
```

die Nachbedingung wäre von der Form

```
(1| :equation(eq)) :or (2| :nonexistent(var)).
```

Durch die Numerierung können die Disjunkte der Vor- und Nachbedingungen einander zugeordnet werden, so daß, falls der Schritt durchgeführt wird, weil Vorbedingung 1 erfüllt ist, nach der Durchführung des Schritts auf jeden Fall auch Nachbedingung 1 gilt (analog für Bedingung 2). Diese Erweiterung des Formalismus zur Angabe der Vor- und Nachbedingungen hätte insbesondere Auswirkungen auf die Überprüfung der Durchführbarkeit eines zusammengesetzten Ablaufschritts.

Die in der vorliegenden Arbeit entwickelten Algorithmen, welche zur Entscheidung über die Einordnung der einzelnen Klassen und der gesamten Schritte in Vererbungs- und Subtyphierarchien benötigt werden, wurden keiner Komplexitätsbetrachtung unterzogen, sind aber in der vorliegenden Form für eine Implementierung voraussichtlich zu aufwendig. Hier wird in Zukunft nach Optimierungsmöglichkeiten gesucht werden müssen. Algorithmus 7.2 wurde außerdem nur skizziert, so daß vor einer Implementierung eine detaillierte Ausarbeitung erforderlich ist, welche insbesondere die oben vorgeschlagene Erweiterung des Formalismus berücksichtigen sollte.

Obwohl die Liste der offenen Punkte und Verbesserungsmöglichkeiten lang ist (und die Menge wünschenswerter Eigenschaften der vorgestellten Formalismen wird mit der Zeit sicherlich noch wachsen) wurde durch die vorliegende Arbeit ein kleiner, aber wichtiger Schritt auf dem Weg zu einer flexiblen Unterstützung verfahrenstechnischer Modellierungsabläufe gemacht.

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[AF95] Alessandro Artale und Enrico Franconi. Hierarchical Plans in a Description Logic of Time and Action. In A. Borgida, M. Lenzerini, D. Nardi und B. Nebel, Hrsg., *Proceedings of the International Workshop on Description Logics*, Rome, June 1995.

[AF97] Alessandro Artale und Enrico Franconi. A Temporal Description Logic for Reasoning about Actions and Plans. Submitted to Journal of Artificial Intelligence Research, 1997.

[All83] James F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[All90] James F. Allen. Formal models of Planning. In James Allen, James Hendler und Austin Tate, Hrsg., *Readings in Planning*, Series in Representation and Reasoning, Seiten 50–54. Morgan Kaufmann, San Mateo, 1990.

[Ame90] Pierre America. Designing an Object-Oriented Programming Language with Behavioural Subtyping. In J. W. de Bakker, W. P. de Roever und G. Rozenberg, Hrsg., *Foundations of Object-Oriented Languages*, Lecture Notes in Computer Science 489, Seiten 60–90. Springer-Verlag, Wien, New York, 1990.

[ASU86] A. V. Aho, R. Sethi und J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison–Wesley, Reading, Mass., 1986.

[Bä92] Christer Bäckström. Equivalence and Tractability Results for SAS^+ Planning. In Bernhard Nebel, Charles Rich und William Swartout, Hrsg., *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR '92)*, San Mateo, California, 1992. Morgan Kaufmann.

[Bau96] Markus Baumeister. Attribute grouping: Emulating metamodels without instantiation. In *Third International Conference on Object-Oriented Information Systems (OOIS)*, London, December 1996. Southbank University.

[Bau97] Markus Baumeister. Persönliche Gespräche. 1996/97.

[BH91a] Franz Baader und Philipp Hanschke. A Scheme for Integrationg concrete Domains into Concept Languages. Research Report RR-91-10, DFKI, Kaiserslautern, 1991.

[BH91b] Franz Baader und Bernhard Hollunder. A Terminological Knowledge Representation System with Complete Inference Algorithm. In *Proceedings of the Workshop on Processing Declarative Knowledge, PDK-91*, Lecture Notes in Artificial Intelligence 567, Seiten 67–86, Wien, New York, 1991. Springer-Verlag.

[BL94] Franz Baader und Armin Laux. Terminological Logics with Modal Operators. Research Report RR-94-33, DFKI, Kaiserslautern, September 1994.

[BS85] R. J. Brachmann und J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[CB88] J. Conklin und M.L. Bergman. gIBIS - a hypertext tool for exploratory policy discussion. *ACM Trans. Office Information Systems*, Seiten 303 – 331, 1988.

[Dev91a] Premkumar Devanbu. Discussion Group Report: Action Modeling. In Christof Peltason, Kai von Luck und Carsten Kindermann, Hrsg., *KIT - Report 95: Terminological Logic Users Workshop*. Technische Universität Berlin, Dezember 1991.

[Dev91b] Premkumar Devanbu. Terminological Logics for Software Information Systems. In Christof Peltason, Kai von Luck und Carsten Kindermann, Hrsg., *KIT - Report 95: Terminological Logic Users Workshop*. Technische Universität Berlin, Dezember 1991.

[DL91] Premkumar T. Devanbu und Diane J. Litman. Plan-Based Terminological Reasoning. In James Allen, Richard Fikes und Erik Sandewall, Hrsg., *Principles of Knowledge Representation and Reasoning: Proceedings of the Second International Conference (KR91)*, Representation and Reasoning, San Mateo, California, 1991. Morgan Kaufmann.

[EFT92] H.-D. Ebbinghaus, J. Flum und W. Thomas. *Einführung in die mathematische Logik*. BI Wissenschaftsverlag, Mannheim, 1992.

[FGS90] Gerhard Friedrich, Georg Gottlob und Markus Stumptner. Wissensrepräsentation. In Georg Gottlob, Thomas Frühwirth und Werner Horn, Hrsg., *Expertensysteme*, Angewandte Informatik, Kapitel 2, Seiten 21–60. Springer-Verlag, Wien, New York, 1990.

[Geo90] Michael P. Georgeff. Planning. In James Allen, James Hendler und Austin Tate, Hrsg., *Readings in Planning*, Series in Representation and Reasoning, Seiten 5–25. Morgan Kaufmann, San Mateo, 1990.

[GK95] Gerd Große und Hesham Khalil. State Event Logic. *MEDLAR Special Issue of the Bulletin of the Interest Group in Pure and Applied Logics*, 1995.

[Hay79] Patrick J. Hayes. The Logic of Frames. In D. Metzing, Hrsg., *Frame Conceptions and Text Understanding*. Walter de Gruyter and Co., Berlin, 1979.

[JM96] Matthias Jarke und Wolfgang Marquardt. Design and evaluation of computer-aided process modeling tools. *Intelligent Systems in Process Engineering*, 92:97–109, 1996.

[KS94] Gerti Kappel und Michael Schrefl. Inheritance of Object Behaviour – Consistent Extension of Object Life Cycles. In J. Eder und L. A. Kalinichenko, Hrsg., *Proceedings of the Second International East/West Database Workshop, Klagenfurt*, Berlin, 1994. Springer-Verlag.

[LM96] Bernd Lohmann und Wolfgang Marquardt. On the systematization of the process of model development. *Computers and Chemical Engineering*, 20:213–218, 1996. Suppl.

[LW93] Barbara Liskov und Jeannette M. Wing. A New Definition of the Subtype Relation. In Oscar Nierstrasz, Hrsg., *ECOOP '93– Object-Oriented Programming, Proceedings of the 7th European Conference, Kaiserslautern, Germany*,

Lecture Notes in Computer Science 707, Seiten 118–141, Wien, New York, July 1993. Springer-Verlag.

[Mar96a] Wolfgang Marquardt. Modellbildung und Simulation verfahrenstechnischer Prozesse. Vorlesungsskript, Lehrstuhl für Prozeßtechnik, RWTH Aachen, 1996.

[Mar96b] Wolfgang Marquardt. Trends in Computer-Aided Process Modeling. *Computers and Chemical Engineering*, 20(6/7):591–609, 1996.

[MBG+97] Wolfgang Marquardt, Markus Baumeister, Willi Geffers, Bernd Lohmann, Ralf Bogusch, Ulrike Sattler und Daniel Souza. VeDa: Objekt-Orientierte Datenmodellierung für die Repräsentation von verfahrenstechnischen Prozeßmodellen. Interner Bericht, Lehrstuhl für Prozeßtechnik, RWTH Aachen, 1997.

[MD93] John D. McGregor und Douglas M. Dyer. A Note on Inheritance and State Machines. *ACM SIGSOFT, Software Engineering Notes*, 18(4):61–69, Oktober 1993.

[Mey] Bertrand Meyer. Object Oriented Software Construction. 2nd Edition, in Vorbereitung.

[Poh94] Klaus Pohl. *A Process Centered Requirements Engineering Environment.* Dissertation, RWTH-Aachen, 1994.

[Ric88] Elaine Rich. *KI – Einführung und Anwendungen.* McGraw-Hill, Hamburg, 1988.

[Sny87] Alan Snyder. Inheritance and the Development of Encapsulated Software Components. In B. Shriver und P. Wegner, Hrsg., *Research directions in Object-Oriented Programming*, Seiten 165–188. The MIT Press, 1987.

[THD90] Austin Tate, James Hendler und Mark Drummond. A Review of AI Planning Techniques. In James Allen, James Hendler und Austin Tate, Hrsg., *Readings in Planning*, Series in Representation and Reasoning, Seiten 26–49. Morgan Kaufmann, San Mateo, 1990.

[vG91] J. P. v. Gigch. *System Design Modeling and Metamodeling.* Plenum Press, New York, 1991.

[vW97] Lars von Wedel. Definition von Modellierungsschritten für die Struktur- und Verhaltensbeschreibung verfahrenstechnischer Prozesse. Studienarbeit, Lehrstuhl für Prozeßtechnik, RWTH Aachen, 1997.

[WL95] George M. Wyner und Jintae Lee. Applying Specialization to Process Models. Technical Report CCSWP 187, Center for Coordination Science, MIT, Massechusetts, September 1995.

A Veranschaulichung der Vor- und Nachbedingungen an einem Ablauf zur Verhaltensbeschreibung

In diesem Abschnitt werden für ein Prozeßfragment, welches die Beschreibung des Verhaltens einer Komponente unterstützt, für die einzelnen Teilschritte die Vor- und Nachbedingungen formuliert. Dadurch soll der Formalismus zur Angabe dieser Bedingungen veranschaulicht werden.

A.1 Vor- und Nachbedingungen für einzelne Schritte

Bei der Verhaltensbeschreibung für eine Komponente wird ein neues Gleichungssystem erzeugt oder ein bestehendes verändert, so daß dieses System mathematisch lösbar ist. Notwendige Bedingungen für die Lösbarkeit eines solchen Systems aus Differentialgleichungen und algebraischen Gleichungen sind ein positives Ergebnis der strukturellen Lösbarkeitsanalyse und Anzahl der Freiheitsgrade gleich Null. Eine hinreichende Bedingung für die Lösbarkeit eines differential-algebraischen Gleichungssystems sind positive Ergebnisse der mathematischen Lösbarkeits- und Indexanalysen.

Als Beispiel für diesen Ablauf verwenden wir das in den Abbildungen 22 und 23 dargestellte Prozeßfragment. Im folgenden sind die darin verwendeten Teilschritte mit ihren jeweiligen Vor- und Nachbedingungen aufgelistet. Wie bereits in früheren Beispielen werden die Vorbedingungen für die `INPUT-SITUATION`, die Nachbedingungen für das `PROCESS-FRAGMENT` eines Schrittes angegeben.

0. **Gesamtes Prozeßfragment**

   Vorbedingung

   Es existiert ein aktiver Case, welcher mindestens ein aktives `DEVICE-MODEL` *enthält, welches entweder noch keine Verhaltensbeschreibung besitzt oder in dessen Gleichungssystem zur Verhaltensbeschreibung die Anzahl der Freiheitsgrade ungleich Null ist oder welches bei null Freiheitsgraden nicht lösbar ist.*

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
   (:undefined(dev.behavior) :or
      (:eq(dev.behavior.degree-of-freedom,_R-dof) :and
         (!=0(_R-dof) :or
            (=0(_R-dof) :and
            :eq(dev.behavior.math-solvability,_B-ms) :and
            =FALSE(_B-ms)))))
```

   Nachbedingung:

   Für die Komponente aus der Vorbedingung existiert ein Gleichungssystem mit mindestens einer Gleichung, welches lösbar ist und keine Freiheitsgrade besitzt.

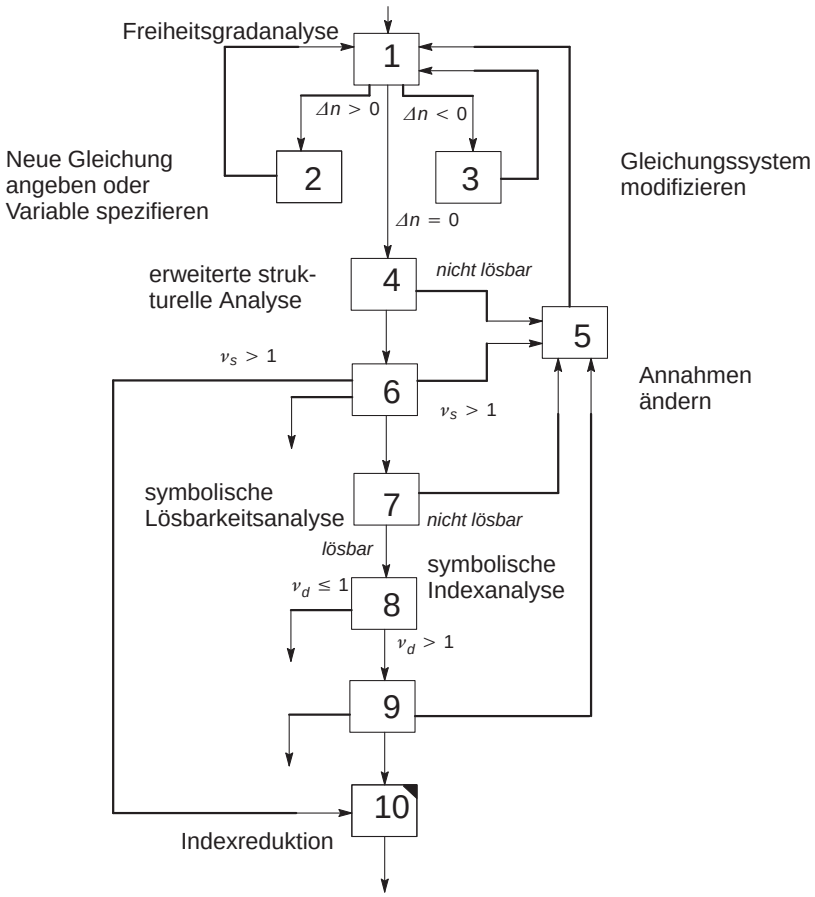

Abbildung 22: Ablauf zur Detaillierung und Analyse der Verhaltensbeschreibung einer Komponente

```
:eq(this.case,c)                                :and
:some-eq(c.components, dev)                     :and
DEVICE-MODEL(dev)                               :and
:is-defined(dev.behavior)                       :and
:eq(dev.behavior.count-equations, _R-ce)        :and
>0(_R-ce)                                       :and
:eq(dev.behavior.math-solvability,_B-solve)     :and
=TRUE(_B-solve)                                 :and
:eq(dev.behavior.degree-of-freedom,_R-dof)      :and
=0(_R-dof)
```

1. **Freiheitsgradanalyse**

   Vorbedingung

   Es existiert ein aktiver Case, welcher mindestens ein aktives `DEVICE-MODEL` *enthält, welches eine Verhaltensbeschreibung besitzt.*

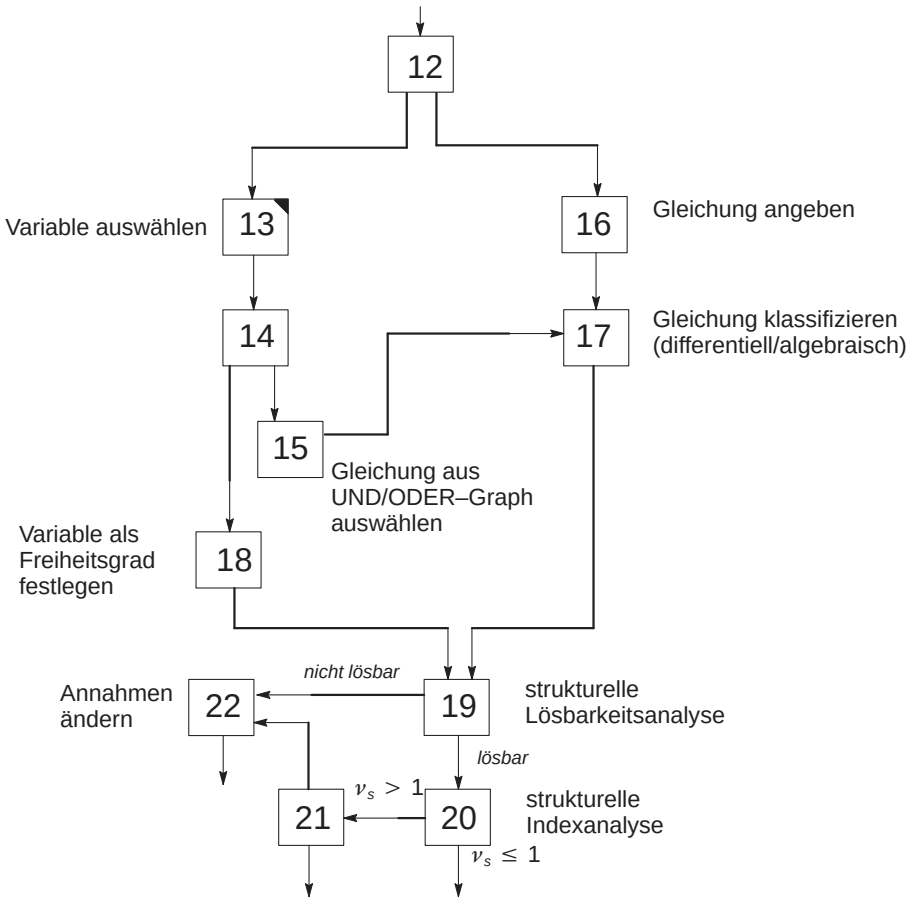

Abbildung 23: Ablauf zur Angabe einer neuen Gleichung oder Spezifizierung einer Variablen

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
(:is-defined(dev.behavior)
```

Nachbedingung:

Für die Verhaltensbeschreibung wurde der Freiheitsgrad ermittelt.

```
:eq(this.case,c)                                :and
:some-eq(c.components, dev)                     :and
DEVICE-MODEL(dev)                               :and
:is-defined(dev.behavior.degree-of-freedom)
```

2. **Neue Gleichung angeben oder Variable spezifizieren**

   Vorbedingung

Es existiert ein aktiver Case, welcher mindestens ein aktives `DEVICE-MODEL` *enthält, welches eine Verhaltensbeschreibung mit einem Freiheitsgrad größer null besitzt.*

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.degree-of-freedom,_R-dof) :and
>0(_R-dof) :and
:nonexistent(eq) :and
   (:nonexistent(var) :or
      (:not INPUT(var) :and
      :not PARAMETER(var) :and
      :not CONSTRAINED(var) :and
      VARIABLE(var) :and
      :some-eq(dev.behavior.variables,var)))
```

Nachbedingung:

Das Gleichungssytem enthält mindestens eine Variable mehr vom Typ Input, Parameter oder Constrained oder eine neue Gleichung.

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
   (INPUT(var) :or
   PARAMETER(var) :or
   CONSTRAINED(var)) :and
   (  :some-eq(dev.behavior.variables,var) :and
      :undefined(dev.behavior.math-solvability) :and
      :undefined(dev.behavior.math-index) :and
      :undefined(dev.behavior.degree-of-freedom) :and
      :undefined(dev.behavior.struct-solvability) :and
      :undefined(dev.behavior.struct-index) :or
:  some-eq(dev.behavior.equations,eq))
```

3. **Gleichungssystem modifizieren**

   Vorbedingung

   Das Verhalten des aktiven Modellbausteins wird durch ein Gleichungssystem mit ≤ 0 *Freiheitsgraden beschrieben.*

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.degree-of-freedom,_R-dof) :and
<0(_R-dof):and
:undefined(dev.behavior.math-solvability) :and
```

```
:undefined(dev.behavior.math-index) :and
:undefined(dev.behavior.degree-of-freedom) :and
:undefined(dev.behavior.struct-solvability) :and
:undefined(dev.behavior.struct-index)
```

Nachbedingung:

Der Typ einer Variablen wurde auf "computed" gesetzt oder es wurde eine Gleichung zugefügt oder eine Variable entfernt. Diese Bedingung läßt sich nicht in Form einer Nachbedingung ausdrücken[22]

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:undefined(dev.behavior.math-solvability) :and
:undefined(dev.behavior.math-index) :and
:undefined(dev.behavior.degree-of-freedom) :and
:undefined(dev.behavior.struct-solvability) :and
:undefined(dev.behavior.struct-index)
```

4. **Erweiterte strukturelle Analyse**

   Vorbedingung

   Es existiert ein aktiver Case, welcher mindestens ein aktives `DEVICE-MODEL` *enthält, dessen Verhaltensbeschreibung mindestens eine Gleichung enthält und keine Freiheitsgrade besitzt.*

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.count-eq,_R-eq) :and
>0(_R-eq) :and
:eq(dev.behavior.degree-of-freedom,_R-dof) :and
=0(_R-dof)
```

   Nachbedingung:

   Für die Verhaltensbeschreibung wurden strukturelle Lösbarkeit und struktureller Index ermittelt.

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.count-eq,_R-eq) :and
>0(_R-eq) :and
```

[22] Auf dieses Problem wird in Abschnitt 9.3 eingegangen.

```
:eq(dev.behavior.degree-of-freedom,_R-dof) :and
=0(_R-dof) :and
:is-defined(dev.behavior.struct-solvability) :and
:is-defined(dev.behavior.struct-index)
```

5. **Annahmen ändern**

   Vorbedingung

   Das Verhalten des aktiven Modellbausteins wird durch ein Gleichungssystem beschrieben, das nicht strukturell lösbar ist oder einen strukturellen Index größer eins hat.

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
   ((:eq(dev.behavior.struct-solvability,_B-ss) :and
      =FALSE(_B-ss)) :or
   (:eq(dev.behavior.struct-index,_R-si) :and
      >1(_R-si)) )
```

   Nachbedingung:

   Der Typ einer Variablen wurde verändert oder es wurde eine Gleichung zugefügt oder gelöscht. Diese Bedingung läßt sich nicht als Nachbedingung formulieren.

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:undefined(dev.behavior.math-solvability) :and
:undefined(dev.behavior.math-index) :and
:undefined(dev.behavior.degree-of-freedom) :and
:undefined(dev.behavior.struct-solvability) :and
:undefined(dev.behavior.struct-index)
```

6. **Auswahl: bei struktureller Lösbarkeit: Annahmen ändern, symbolische Lösbarkeitsanalyse oder Ende**

   Vorbedingung

   Es existiert ein aktiver Case, welcher mindestens ein aktives `DEVICE-MODEL` *enthält, welches eine Verhaltensbeschreibung besitzt das strukturell lösbar ist und dessen struktureller Index definiert ist.*

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.struct-solvability,_B-ss) :and
=TRUE(_R-ss) :and
:is-defined(dev.behavior.struct-index)
```

Nachbedingung:

Auswahlschritte erhalten immer die vor ihrer Durchführung erfüllten Bedingungen.

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.struct-solvability,_B-ss) :and
=TRUE(_R-ss) :and
:is-defined(dev.behavior.struct-index)
```

7. **Symbolische Lösbarkeitsanalyse**

Vorbedingung

Es existiert ein aktiver Case, welcher mindestens ein aktives `DEVICE-MODEL` *enthält, welches eine Verhaltensbeschreibung mit mindestens einer Gleichung und ohne Freiheitsgrade besitzt.*

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.degree-of-freedom,_R-dof) :and
=0(_R-dof) :and
:eq(dev.behavior.count-eq,_R-eq) :and
>0(_R-eq)
```

Nachbedingung:

Für die Verhaltensbeschreibung wurde die mathematische Lösbarkeit ermittelt.

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.degree-of-freedom,_R-dof) :and
=0(_R-dof) :and
:eq(dev.behavior.count-eq,_R-eq) :and
>0(_R-eq) :and
:is-defined(dev.behavior.math-solvability)
```

8. **Symbolische Indexanalyse**

Vorbedingung

Es existiert ein aktiver Case, welcher mindestens ein aktives `DEVICE-MODEL` *enthält, welches eine Verhaltensbeschreibung mit mindestens einer Gleichung und ohne Freiheitsgrade besitzt.*

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.degree-of-freedom,_R-dof) :and
=0(_R-dof) :and
:eq(dev.behavior.count-eq,_R-eq) :and
>0(_R-eq) :and
```

Nachbedingung:

Für die Verhaltensbeschreibung wurde der mathematische Index ermittelt.

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.degree-of-freedom,_R-dof) :and
=0(_R-dof) :and
:eq(dev.behavior.count-eq,_R-eq) :and
>0(_R-eq) :and
:is-defined(dev.behavior.math-index)
```

9. **Auswahl: bei mathematischem Index >1: Indexreduktion?**

   Vorbedingung

   Es existiert ein aktiver Case, welcher mindestens ein aktives `DEVICE-MODEL` *enthält, welches eine Verhaltensbeschreibung besitzt deren mathematischer Index größer 1 ist.*

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.math-index,_R-mi) :and
>1(_R-mi)
```

   Nachbedingung:

   Auswahlschritte erhalten immer die vor ihrer Durchführung erfüllten Bedingungen.

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.math-index,_R-mi) :and
>1(_R-mi)
```

10. **Indexreduktion**

Vorbedingung

Es existiert ein aktiver Case, welcher mindestens ein aktives `DEVICE-MODEL` *enthält, welches eine Verhaltensbeschreibung ohne Freiheitsgrade besitzt und in dem der strukturelle oder der mathematische Index größer 1 ist.*

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.degree-of-freedom,_R-dof) :and
=0(_R-dof) :and
   (  (:eq(dev.behavior.struct-index,_R-si) :and
      >1(_R-si)) :or
      (:eq(dev.behavior.math-index,_R-mi) :and
      >1(_R-mi)))
```

Nachbedingung:

Der mathematische Index wurde auf einen Wert ≤ 1 *reduziert.*

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.degree-of-freedom,_R-dof) :and
=0(_R-dof) :and
:eq(dev.behavior.struct-index,_R-si) :and
<2(_R-si) :and
:eq(dev.behavior.math-index,_R-mi) :and
<2(_R-mi)
```

12. **Auswahl: Neue Gleichung angeben oder Variable spezifizieren?**

Vorbedingung

Es existiert ein aktiver Case, welcher mindestens ein aktives `DEVICE-MODEL` *enthält, welches eine Verhaltensbeschreibung mit einem Freiheitsgrad größer null besitzt.*

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.degree-of-freedom,_R-dof) :and
>0(_R-dof)
```

Nachbedingung:

Auswahlschritte erhalten immer die vor ihrer Durchführung erfüllten Bedingungen.

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.degree-of-freedom,_R-dof) :and
>0(_R-dof)
```

13. **Variable auswählen**

    Vorbedingung

    Es existiert ein aktiver Case, welcher mindestens ein aktives `DEVICE-MODEL` *enthält, dessen Verhaltensbeschreibung mindestens eine Variable enthält und* > 0 *Freiheitsgrade besitzt.*

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.count-variables,_R-cv) :and
>0(_R-cv) :and
:eq(dev.behavior.degree-of-freedom, _R-dof) :and
>0(_R-dof)
```

    Nachbedingung:

    Eine Variable aus der Verhaltensbeschreibung wurde ausgewählt in den betreffenden Slot eingetragen.

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:some-eq(dev.behavior.variables,sv) :and
:eq(dev.behavior.selected-variable,sv)
```

14. **Auswahl: Gleichung auswählen oder Variable als externe Größe festlegen**

    Vorbedingung

    Eine Variable aus der Verhaltensbeschreibung wurde ausgewählt und in den betreffenden Slot eingetragen.

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:some-eq(dev.behavior.variables,sv) :and
:eq(dev.behavior.selected-variable,sv)
```

Nachbedingung:

Auswahlschritte erhalten immer die vor ihrer Durchführung erfüllten Bedingungen.

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:some-eq(dev.behavior.variables,sv) :and
:eq(dev.behavior.selected-variable,sv)
```

15. **Gleichung aus UND/ODER-Graph auswählen**

Vorbedingung

Es existiert ein aktiver Case, welcher mindestens ein aktives `DEVICE-MODEL` *enthält, in dessen Verhaltensbeschreibung eine Variable selektiert ist und die Anzahl der Freiheitsgrade* > 0 *ist.*

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:some-eq(dev.behavior.variables,sv) :and
:eq(dev.behavior.selected-variable,sv) :and
:nonexistent(eq) :and
:eq(dev.behavior.degree-of-freedom,_R-dof) :and
>0(_R-dof)
```

Nachbedingung:

Eine neue Gleichung wurde erzeugt und in den Slot active-equation der Variablen eingetragen.

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
EQUATION(eq) :and
:some-eq(dev.behavior.variables,sv) :and
:eq(dev.behavior.selected-variable,sv) :and
:some-eq(dev.behavior.selected-variable.active-equation,eq) :and
:undefined(dev.behavior.math-solvability) :and
:undefined(dev.behavior.math-index) :and
:undefined(dev.behavior.degree-of-freedom) :and
:undefined(dev.behavior.struct-solvability) :and
:undefined(dev.behavior.struct-index)
```

16. **Gleichung angeben**

Vorbedingung

Es existiert ein aktiver Case, welcher mindestens ein aktives `DEVICE-MODEL` *enthält, welches eine Verhaltensbeschreibung mit* > 0 *Freiheitsgraden besitzt.*

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:is-defined(dev.behavior) :and
:eq(dev.behavior.degree-of-freedom, _R-dof) :and
>0(_R-dof) :and
:nonexistent(eq)
```

Nachbedingung:

Es wurde eine neue Gleichung erzeugt und der Verhaltensbeschreibung zugefügt.

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
EQUATION(eq) :and
:some-eq(dev.behavior.equations,eq) :and
:undefined(dev.behavior.math-solvability) :and
:undefined(dev.behavior.math-index) :and
:undefined(dev.behavior.degree-of-freedom) :and
:undefined(dev.behavior.struct-solvability) :and
:undefined(dev.behavior.struct-index)
```

17. **Gleichung klassifizieren**

Vorbedingung

Es existiert ein aktiver Case, welcher mindestens ein aktives `DEVICE-MODEL` *enthält, welches eine Verhaltensbeschreibung mit mindestens einer Gleichung besitzt (optional könnte für diesen Schritt zusätzlich gefordert werden, daß der Typ der zu klassifizierenden Gleichung bisher undefiniert ist).*

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:some-eq(dev.behavior.equations,eq)
```

Nachbedingung:

es wurde der Typ einer Gleichung festgesetzt

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:some-eq(dev.behavior.equations,eq) :and
:is-defined(eq.type)
```

18. **Variable als externe Größe festlegen**

    Vorbedingung

    Eine Variable aus der Verhaltensbeschreibung wurde ausgewählt; die Variable ist nicht vom Typ Input, Parameter oder Konstante.

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:some-eq(dev.behavior.variables,sv) :and
:eq(dev.behavior.selected-variable,sv) :and
:not INPUT(sv) :and
:not PARAMETER(sv) :and
:not CONSTANT(sv)
```

    Nachbedingung:

    Der Typ der Variablen wurde auf einen der Werte Input, Parameter, Konstante geändert.

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:some-eq(dev.behavior.variables,sv) :and
:eq(dev.behavior.selected-variable,sv) :and
   (PARAMETER(sv) :or
   INPUT(sv) :or
   CONSTANT(sv)) :and
:undefined(dev.behavior.math-solvability) :and
:undefined(dev.behavior.math-index) :and
:undefined(dev.behavior.degree-of-freedom) :and
:undefined(dev.behavior.struct-solvability) :and
:undefined(dev.behavior.struct-index)
```

19. **Strukturelle Lösbarkeitsanalyse**

    Vorbedingung

    Es existiert ein aktiver Case, welcher mindestens ein aktives `DEVICE-MODEL` *enthält, welches eine Verhaltensbeschreibung mit mindestens einer Gleichung besitzt.*

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev)
```

Nachbedingung:

Für die Verhaltensbeschreibung wurde die strukturelle Lösbarkeit ermittelt.

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:is-defined(dev.behavior.struct-solvability)
```

20. **Strukturelle Indexanalyse**

Vorbedingung

Es existiert ein aktiver Case, welcher mindestens ein aktives `DEVICE-MODEL` *enthält, welches eine Verhaltensbeschreibung mit mindestens einer Gleichung besitzt, dessen Gleichungssystem strukturell lösbar ist.*

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.struct-solvability,_B-ss) :and
=TRUE(_B-ss)
```

Nachbedingung:

Für die Verhaltensbeschreibung wurde der strukturelle Index ermittelt.

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.struct-solvability,_B-ss) :and
=TRUE(_B-ss) :and
:is-defined(dev.behavior.struct-index)
```

21. **Auswahl: bei Strukturellem Index >1 Annahmen ändern?**

Vorbedingung

Es existiert ein aktiver Case, welcher mindestens ein aktives `DEVICE-MODEL` *enthält, welches eine Verhaltensbeschreibung besitzt, deren struktureller Index größer 1 ist.*

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.struct-index,_R-si) :and
>1(_R-si)
```

Nachbedingung:

Auswahlschritte erhalten immer die vor ihrer Durchführung erfüllten Bedingungen.

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.struct-index,_R-si) :and
>1(_R-si)
```

22. **Annahmen ändern**

Vorbedingung

Das Verhalten des aktiven Modellbausteins wird durch ein Gleichungssystem beschrieben, das nicht strukturell lösbar ist.

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.struct-solvability,_B-ss) :and
=FALSE(_B-ss)
```

Nachbedingung:

Der Typ einer Variablen wurde verändert oder es wurde eine Gleichung zugefügt oder gelöscht. Diese Bedingung läßt sich nicht als Nachbedingung formulieren.

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:undefined(dev.behavior.math-solvability) :and
:undefined(dev.behavior.math-index) :and
:undefined(dev.behavior.degree-of-freedom) :and
:undefined(dev.behavior.struct-solvability) :and
:undefined(dev.behavior.struct-index)
```

A.2 Spezialisierung einzelner Schritte

Für den Schritt **Strukturelle Indexanalyse** sind neben den hier angegebenen Bedingungen auch alternative Forderungen bzw. Zusicherungen denkbar. Diese hängen im wesentlichen von der Mächtigkeit des verwendeten Verfahrens ab. Eine erste Variante setzt zusätzlich voraus, daß das untersuchte Gleichungssystem keine Freiheitsgrade besitzt, eine zweite Variante verzichtet auf die Forderung nach der strukturellen Lösbarkeit des Gleichungssystems. Auch die Forderung, daß die Anzahl der Freiheitsgrade bekannt sein muß, kann fallengelassen werden. Weitere Alternativen ergeben sich aus der Zusammenfassung der strukturellen Lösbarkeits- und Indexanalyse in dem Schritt **Erweiterte strukturelle Analyse**, welcher wiederum sowohl mit als auch ohne Forderungen an die Anzahl der Freiheitsgrade formuliert werden kann.

Es ergeben sich dementsprechend verschiedene Varianten, die mit ihren Vor- und Nachbedingungen im folgenden noch einmal aufgelistet werden. Alle verlangen die Existenz eines Gleichungssystems zur Verhaltensbeschreibung, erhalten ihre Vorbedingungen und füllen ggf. vor der Durchführung des Schrittes nicht definierte Slots aus.

A **Strukturelle Indexanalyse**

Vorbedingung: keine weiteren Forderungen außer der Existenz der Verhaltensbeschreibung

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:is-defined(dev.behavior)
```

Nachbedingung: keine weiteren Garantien

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:is-defined(dev.behavior) :and
:is-defined(dev.behavior.struct-index)
```

B **Strukturelle Indexanalyse**

Vorbedingung: Anzahl der Freiheitsgrade bekannt

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:is-defined(dev.behavior.degree-of-freedom)
```

Nachbedingung: keine weiteren Garantien

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:is-defined(dev.behavior.degree-of-freedom)
```

C **Strukturelle Indexanalyse**

Vorbedingung: keine Freiheitsgrade

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.degree-of-freedom,_R-dof) :and
=0(_R-dof)
```

Nachbedingung: keine weiteren Garantien

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.degree-of-freedom,_R-dof) :and
=0(_R-dof)
```

D **Strukturelle Indexanalyse**

Vorbedingung: strukturelle Lösbarkeit ist gegeben

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.struct-solvability,_B-ss) :and
=TRUE(_B-ss) :and
:is-defined(dev.behavior.degree-of-freedom)
```

Nachbedingung: der strukturelle Index ist definiert

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.struct-solvability,_B-ss) :and
=TRUE(_B-ss) :and
:is-defined(dev.behavior.struct-index) :and
:is-defined(dev.behavior.degree-of-freedom)
```

E **Erweiterte Strukturelle Analyse**

Vorbedingung: keine weiteren außer der Existenz der Verhaltensbeschreibung

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev)
```

Nachbedingung: die strukturelle Lösbarkeit ist definiert

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:is-defined(dev.behavior.struct-solvability)
```

F **Erweiterte Strukturelle Analyse**

Vorbedingung: Anzahl der Freiheitsgrade bekannt

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:is-defined(dev.behavior.degree-of-freedom)
```

Nachbedingung: die strukturelle Lösbarkeit ist definiert

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:is-defined(dev.behavior.degree-of-freedom) :and
:is-defined(dev.behavior.struct-solvability)
```

G **Erweiterte Strukturelle Analyse**

Vorbedingung: Anzahl der Freiheitsgrade null

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.degree-of-freedom,_R-dof) :and
=0(_R-dof)
```

Nachbedingung: die strukturelle Lösbarkeit ist definiert

```
:eq(this.case,c) :and
:some-eq(c.components, dev) :and
DEVICE-MODEL(dev) :and
:eq(dev.behavior.degree-of-freedom,_R-dof) :and
=0(_R-dof) :and
:is-defined(dev.behavior.struct-solvability)
```

Für die Schritte A bis G kann nun ermittelt werden, in welcher Spezialisierungsbeziehung sie zueinander stehen. Es ergibt sich aufgrund der Implikationsbeziehungen zwischen den Vor- und Nachbedingungen der einzelnen Schritte die in Abbildung 24 dargestellte Hierarchie.

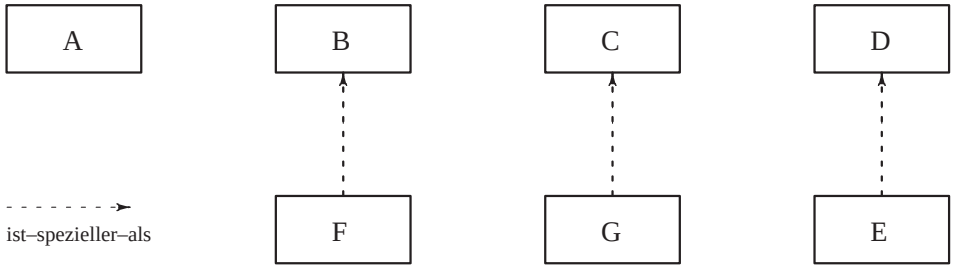

Abbildung 24: Spezialisierungshierarchie für die Ablaufschritte zur strukturellen Indexanalyse