Ein offenes Deduktionssystem für die Entwicklung von Datenbankanwendungen

Diplomarbeit

von

Roland Stuckardt
6000 Frankfurt am Main

eingereicht bei

Prof. Dr. Klaus Waldschmidt
Professur für technische Informatik
Johann Wolfgang Goethe–Universität
Frankfurt am Main

Prof. Dr. Joachim W. Schmidt
Professur für Datenbanken und Informationssysteme
Universität Hamburg

Frankfurt am Main, den 12. Februar 1991

Inhaltsverzeichnis

Kapitel 1

Einleitung und Motivation

Die Disziplin *Software-Engineering* ist ein Zweig der Informatik, der sich vor allem auf die Entwicklung immer leistungsfähigerer Rechner und anspruchsvollerer Anwendungen begründet. In der Zeit vor 1965 war Programmieren die Aufgabe weniger Spezialisten mit eher technischer Ausrichtung. Die damaligen Computersysteme wurden in einer durch die Maschinennähe der existenten Programmiersprachen geprägten *operationalen* Sichtweise programmiert; die Anwendungen waren vornehmlich *numerischer* Gestalt. Mit der Verfügbarkeit immer leistungsstärkerer Hardware wurden neue Konzepte wie Multiprogrammierung, Mehrbenutzersysteme, interaktive Ein- und Ausgabe und Parallelität realisierbar, für die sich diese "*von Neumann-Sichtweise*" der Programmierung als unzureichend erwies. Die in den damaligen Programmiersprachen implementierten komplexen Softwaresysteme waren, mangels geeigneter Spezifikations-, Verifikations- und Dokumentationstechniken, unzuverlässig und unüberschaubar; folglich führten Fehlersuche in und Erweiterung von bestehenden Systemen zu großen Problemen. Aufgrund des immer breiteren Anwendungsspektrums der Datenverarbeitung wurden andererseits immer höhere Anforderungen an die Zuverlässigkeit von Computersystemen gestellt, man denke nur an Anwendungen im Bereich von Luft- und Raumfahrt, Nuklear- und Rüstungstechnologie[1]. Diese immer größer werdende Kluft zwischen Leistungsfähigkeit von Hardware und Software prägte nach 1965 den Begriff der *Software-Krise*.
Man erkannte rasch, daß die bestehenden Probleme vor allem durch die Gestalt der verwendeten *Beschreibungsmittel* (Programmiersprachen, "Low-Level"-Spezifikationsformalismen wie Flußdiagramme etc.) begründet werden. Wünschenswert wurden Sprachen, die mehr von der unterliegenden Von-Neumann-Rechnerarchitektur abstrahieren. So identifizierte beispielsweise Dijkstra die Probleme, die aus der Verwendung von Sprungbefehlen resultieren ([Dijk68], "*Go-to Statement considered harmful*"). Das Paradigma der *strukturierten Programmierung* wurde kreiert und resultierte in der Entwicklung von Programmiersprachen wie Pascal.
Eine weitere Schwierigkeit wurde in der bisher fehlenden (oder unzureichenden) *formalen Semantikdefinition von Programmiersprachen* lokalisiert. Als Basis für die Verifizierbarkeit sollte jedem Programm eine *formale* Semantik zugeordnet werden. R. Floyd sprach in diesem Zusam-

[1] Die Nennung der zuletzt genannten Anwendungsgebiete bedeutet nicht, daß der Autor den Einsatz der Informatik für derartige Aufgaben als ethisch unbedenklich einschätzt. Es ist jedoch eine historische Realität, daß sich die rasante Weiterentwicklung der Digitaltechnik insbesondere auf die Nachfrage aus den genannten, *politisch propagierten* Anwendungsgebieten begründet. Software-Engineering hat jedoch auch komplexe Systeme aus dem zivilen Bereich, wie in der Medizintechnik, Verwaltung etc. zum Gegenstand, deren Legitimität außer Frage steht.

menhang von dem Problem des "*Assigning Meanings to Programs*" ([Floy67]). Aufgrund eines auf diese Weise erhaltenen mathematischen Modells der Semantik von programmiersprachlichen Konstrukten sollte die Erstellung von Programmen auf einer formalen Basis geschehen. Auch auf dieser Stufe geschah die *Verifikation* jedoch i.a. durch *Austesten* der Programme *nach* ihrer Erstellung, was immer noch als unzureichend zu bezeichnen ist, da durch die betrachteten Testdaten jeweils nur Spezialfälle, nicht jedoch die universelle Korrektheit verifiziert wird.
Es war daher ein weiterer Schritt erforderlich. Die Programmierung sollte auf der Grundlage einer Spezifikation mit ebenfalls möglichst *formaler* Semantikdefinition geschehen, wobei die Korrektheit des erstellten Programms relativ zur gegebenen Spezifikation nachgewiesen wird. [Gogu81] identifiziert in diesem Zusammenhang als notwendige Voraussetzung der formalen Verifizierbarkeit die Existenz von Beschreibungssprachen für die zwei Aufgaben

- *Systemspezifikation*
- *Formulierung der zu erbringenden Verifikationsbedingungen.*

Auf der Programmiersprachenseite läßt sich auf den ersten Blick die *logische Programmierung*, vertreten u.a. durch die Sprache PROLOG, als extremer "Verfechter" des formalen Vorgehens ausmachen: der Programmierer *spezifiziert* die Problemdomäne in (i.a. eingeschränkter) Prädikatenlogik erster Ordnung und überläßt die (unmittelbare) "Ausführung", entsprechend dem Paradigma der logischen Programmierung "*Algorithm = Logic + Control*" ([Kowa79]), dem Sprachinterpreter, der die Ablaufkontrolle im Idealfall *ohne Benutzereingriff* zu realisieren hat. Da logische Programmiersprachen die ansonsten nur indirekt als semantisches *Modellierungskonzept* von Spezifikations- und Programmiersprachen verwendete Prädikatenlogik zum unmittelbaren Gegenstand haben, verfließt die Unterscheidung von Spezifikation und Implementation: im Extremfall der "reinen" logischen Programmierung reduziert sich gemäß [Kowa84] die Differenzierung zwischen Spezifikation und Implementation auf deren unterschiedliche *Effizienz*. Für reale logische Programmiersprachen wie PROLOG ist diese These jedoch nicht haltbar: um genügend mächtige Spezifikationen umsetzen zu können, sind die alleine auf der *restriktiven Hornklausellogik* basierenden deklarativen Konzepte von PROLOG unzureichend. Der Programmierer macht sich i.a. die aus den operationalen Konzepten wie CUT etc. resultierende *operationale Semantik* von PROLOG zunutze. Da die Semantik des logischen Kerns von PROLOG von der operationalen Semantik differiert (siehe z.B. [Beck88]), sind auch hier bestimmte Entwicklungsschritte notwendig, wie sie beispielsweise in [Devi90] identifiziert werden.
Ein allgemeiner Ansatz beinhaltet die Verwendung *unterschiedlicher*, nicht notwendigerweise formallogischer Beschreibungssprachen auf den verschiedenen Ebenen der Softwareentwicklung. Dabei verfügt im Idealfall *jede* der verwendeten Sprachen über eine *formale* Semantikdefinition. In Anspielung auf [Floy67] bezeichnet beispielsweise Abrial ein vollständig formales Vorgehen mit "*Assigning Programs to Meanings*" ([Abri84]).
Ein derartiges *formales* Vorgehen ist die Grundlage für die *Toolunterstützung* der Programmentwicklung, sei es zur *automatischen, regelbasierten Implementation* oder zur *Verifikation benutzererstellter Programme* relativ zur gegebenen Spezifikation.
Aufgrund der im letzten Jahrzehnt gewachsenen zentralen Bedeutung von *Datenbankanwendungen* in Wirtschaft und Verwaltung ist die *Konsistenthaltung umfangreicher Datenbestände* ein wichtiger Aspekt. Da diese Aufgabe von existenten Datenbank*systemen* nur unzureichend unterstützt wird, kommt der Verläßlichkeit der verwendeten Datenbanksoftware eine wichtige Rolle zu, sodaß sich der Einsatz von Methoden des Software Engineering im Datenbankbereich

gerade auch aus *ökonomischer* Sicht anbietet. Diese Diplomarbeit hat eine *Softwareentwicklungsumgebung mit Toolunterstützung im Bereich der Datenbankanwendungsentwicklung*, die sog. DAIDA-Entwicklungsumgebung, zum Gegenstand, die im Rahmen des EG-Projekts 892, ESPRIT-DAIDA, realisiert wurde[2]. Der über hochsprachliche, deklarative Sprachkonstrukte verfügenden *Implementationssprache* DBPL ([SEMa88]) liegt ein erweitertes relationales Datenbankmodell zugrunde.
Der Schwerpunkt der Arbeit liegt in der *Beschreibung und Anwendung eines zur Verifikation der einzelnen Implementationsschritte eingesetzten Beweisassistenten*, das von J.R. Abrial konzipierte *B-Tool* ([Abri86]). Der Name "B-Tool" wird im Rahmen der Diskussion einer formallogischen Axiomatisierung einer Umgebung für die Programmentwicklung in [Abri84] motiviert: das **B**-Tool unterstützt ein derartiges, im Sinne der französischen Mathematikergruppe **B**ourbaki *vollständig formales* Vorgehen.
Im Rest dieses Kapitels sollen nun zunächst die elementaren Begriffe des Software Engineering abgegrenzt und das Korrektheitsproblem der Programmentwicklung näher umrissen werden. Die folgenden Abschnitte beschreiben kurz die Möglichkeiten der Toolunterstützung sowie die datenbankspezifischen Aspekte der formal verifizierten Programmentwicklung. Im abschließenden Abschnitt wird ein Überblick über den weiteren Inhalt dieser Arbeit gegeben.

1.1 Begriffe und Methoden des Software-Engineering

Gemäß [IEEE83] ist der Begriff *software engineering* definiert als

> *the systematic approach to the development, operation, maintenance, and retirement of software*

Die Disziplin Software-Enginnering umfaßt danach nicht nur die reinen Entwicklungsaufgaben, sondern hat darüberhinaus noch den Einsatz sowie die *Wartung* von Softwaresystemen zum Gegenstand. Sie ist von ihrer Natur her bestimmt durch ein *systematisches*, d.h. insbesondere nachvollziehbares und reproduzierbares Vorgehen.
Im folgenden soll konkretisiert werden, was im Kontext dieser Arbeit unter *Softwareentwicklung* zu verstehen ist. Nach den obigen Ausführungen soll hierbei ein Programm ausgehend von einer formalen Spezifikation entwickelt werden. Die folgenden Ausführungen beziehen sich auf die weitergehende Zielsetzung, daß die Korrektheit des Programms *zu verifizieren* ist. Hierzu sollte betont werden, daß bestimmte Aspekte außer Acht gelassen werden:

> *Die Verifikation des Programms geschieht relativ zur vorgegebenen Spezifikation, jedoch kann nicht garantiert werden, daß die in der Spezifikation formulierten Anforderungen die Wünsche des Auftraggebers korrekt abbilden. Eine* formale *Validierung mit den Anforderungen des Auftraggebers ist hingegen nicht möglich.*

Qualitätsmerkmale wie Modularität, Portabilität, Effizienz etc. werden *nicht* berücksichtigt. [LiBe79] unterscheiden in diesem Zusammenhang zwischen *functional specifications* und *performance specifications*: es ist das "funktionale" Systemverhalten, das üblicherweise (und auch hier)

[2] Das Akronym DAIDA steht für *Development of Advanced Interactive Data-intensive Applications.*

Gegenstand der Programmentwicklung ist.
Die einzelnen Phasen der Lebensdauer des Produkts "Software" werden in Form sog. *Software Life Cycles* zusammengefaßt: ausgehend von einer *Anforderungsanalyse* und einer daraus gewonnenen *formalen Spezifikation* wird das System *entworfen* und *implementiert*; nach einer *Validierung* mit den informellen Anforderungen und einer eventuellen Revidierung des Entwicklungs- und Implementationsprozesses wird das resultierende System anhand der *formalen* Anforderungen *verifiziert* und während der Benutzung *gewartet*. Für die initiale Phase der Anforderungsanalyse und Formalisierung ist auch die Bezeichnung *requirements engineering* gebräuchlich.
Wegen der großen Kluft zwischen Beschreibungen des *requirements engineering* und der Implementation wird es allgemein als zweckmäßig angesehen, die Entwicklung der Implementation aus der Spezifikation in *mehreren* Schritten zu vollziehen. Die in einem einzelnen Schritt getroffenen Festlegungen von Implementationsdetails werden *Entwurfsentscheidungen* genannt.
[Part90] nennt als Beispiel die Programmkonstruktion durch *schrittweises Verfeinern* nach N. Wirth ([Wirt71]), führt jedoch als deren Nachteil auf, daß die einzelnen Entwurfsentscheidungen *problemabhängig* getroffen werden und daß keinerlei formale Einschränkungen bestehen. Als alternatives Vorgehen wird das sog. *transformational programming* genannt, dessen grundlegende Merkmale die *konstruktiv garantierte Korrektheit des Programms* und die Verwendung schematischer, für eine *Klasse* von Problemen gültiger *semantikerhaltender Transformationsregeln* sind.
In diesem Zusammenhang wird, anhand eines Zitates aus [BuFe78], der Unterschied zwischen schrittweiser Verfeinerung und "transformational programming" anschaulich beschrieben:

> *... stepwise refinement was compared with creating a sculpture out of a piece of rock, whereas transformational programming corresponds rather to modeling a sculpture using clay or plasticine.*

Unter diesem Gesichtspunkt ähnelt der DAIDA-Ansatz dem "transformational programming"[3]. Jedoch bestehen Unterschiede im Hinblick auf die Art der Verifikation: in DAIDA-Programmentwicklungen existieren keine standardisierten, garantiert semantikerhaltenden *Entwicklungsschritte*, obwohl für eine breite Klasse von Anwendungsentwicklungen in einer standardisierten Weise verfahren werden kann. Dagegen werden sich die durchzuführenden *Verifikationsnachweise* als *teilweise standardisierbar* erweisen.

1.2 Unterstützung durch Software-Werkzeuge

Die soeben durchgeführten Betrachtungen erlauben es, die Stellen zu identifizieren, an denen eine formale Toolunterstützung denkbar ist. Ein möglicher Ansatz besteht darin, semantisch gerechtfertigte Entwicklungsschritte durch das Softwaretool *anwenden* zu lassen. Ist ein Entwicklungsschritt ausgeführt, so ist dessen Korrektheit bereits konstruktiv garantiert. Dieser Ansatz wird in der in [Part90] beschriebenen Transformationsmethodologie des CIP-Projekts der technischen Universität München (CIP= *Computer-aided-Intuition-guided Program Development*) verfolgt, bei dem die "intellektuelle" Arbeit der Regelauswahl vom Benutzer, die formalen Nachweise

[3] zumindest, wenn man die Transformationen zwischen den unterschiedlichen Beschreibungssprachen der DAIDA-Umgebung außer Betracht läßt und einzig den Teil der Entwicklungsumgebung, innerhalb dessen die einzelnen Schritte hin zur Implementation durchgeführt werden, berücksichtigt (vgl. u.)

der Korrektheit der intuitiv getroffenen Entscheidungen jedoch *vor deren Ausführung* vom Tool erbracht werden.
In DAIDA hingegen wird ein *indirekter* Weg beschritten: die Entwurfsentscheidungen werden vom Benutzer getroffen; deren *Verifikation* kann jedoch zu einem *beliebigen* Zeitpunkt erfolgen. Das B-Tool generiert sog. *Beweisverpflichtungen*, deren vollständige Erbringung die Korrektheit der erstellten "verfeinerten" Systembeschreibung implizieren. Diese Vorgehen ist vergleichbar mit dem in [Good84] beschrittenen Weg: anhand der getroffenen *Entwurfsentscheidungen* werden sog. *verification conditions* generiert, deren Nachweis durch einen eigens hierfür realisierten interaktiven "*proof checker*" geschieht[4]. Während in DAIDA allerdings sowohl die Generierung als auch der Nachweis der Beweisverpflichtungen mit demselben Tool geschieht, verwendet der in [Good84] vorgestellte "Gypsy"-Ansatz *unterschiedliche Werkzeuge* für Generierung, Vereinfachung und Nachweis der Verifikationsbedingungen.
Neben der Rolle des B-Tools in Bezug auf die Verarbeitung der Beweisverpflichtungen fällt diesem noch eine weitere Aufgabe als *Abbildungsassistent* zwischen den unterschiedlichen Beschreibungssprachen der DAIDA-Umgebung zu, was allerdings in dieser Arbeit nur peripher behandelt werden soll (vgl. [Wetz90]).
Der *Nachweis der zu erbringenden DAIDA-Beweisverpflichtungen* stellt hingegen ein *zentrales Thema dieser Diplomarbeit dar*. Es werden geeignete B-Tool-Regelsammlungen erstellt, die deren Beweis erlauben. Die Rolle des B-Tools als *Deduktionssystem* steht daher im Mittelpunkt und wird ausführlich behandelt.

1.3 Datenbankspezifische Aspekte der Softwarentwicklung

Die *Beschreibung* einer Datenbankanwendung beinhaltet die Modellierung *statischer* Eigenschaften (Datenstrukturen), *dynamischer* Eigenschaften (insbes. der die Datenbankextensionen manipulierenden *Transaktionen*) sowie die Festlegung von *Integritätsbedingungen*, die von den Extensionen der Datenbank zu jedem Zeitpunkt zu erfüllen sind. Die Problemstellung des Einsatzes von Methoden des Software Engineering bei der Datenbankprogrammentwicklung unterscheidet sich wesentlich von der Aufgabe bei der Entwicklung herkömmlicher Software, wie in [Wetz90] ausführlich diskutiert wird. Die wichtigsten Thesen lauten:

- Die von Datenbankprogrammen manipulierten Datenstrukturen sind i.a. wesentlich komplexer, da sie die Gegebenheiten der realen Welt widerspiegeln.
- Bei der Erstellung herkömmlicher Programme steht dessen *Algorithmik* im Vordergrund, während die Entwicklung von Datenbanksoftware insbesondere nichttriviale Fragen der *Datenmodellierung* beinhaltet.
- Die verwendeten *Spezifikationssprachen* müssen in der Lage sein, die für die zu entwickelnde Software relevanten statischen Umweltgegebenheiten adäquat abzubilden (*semantisch reiche Datenmodelle*, Ansätze aus der Wissensrepresentation). Hierfür bedarf es neben einer geeigneten Beschreibungssprache zusätzlich einer *Entwurfsmethodik*.

[4] Die Bezeichnung *proof checker* wird für Beweiser verwendet, die Deduktionen lediglich *benutzergesteuert*, d.h. *ausschließlich nichtautomatisch*, durchführen. Für die entsprechende Funktion des B-Tools wird hingegen der Begriff *Beweisassistent* geprägt.

- Die *Algorithmik der Transaktionen* hingegen ist i.a. *von relativ elementarer Gestalt* (wobei das der Datenbankprogrammiersprache zugrundeliegende *Datenbankmodell* eine wichtige Rolle spielt, wie sich zeigen wird).

Die Aufgaben des *Requirements Engineering* haben demnach einen anderen Schwerpunkt als bei der Entwicklung "konventioneller" Programme.
Der *Implementierungsprozeß* der Datenbankprogrammentwicklung muß vor allem das Problem lösen, die relativ komplexen *Datenstrukturen* im Datenbankmodell der Zielsprache zu realisieren. Für diese Aufgabe nennt [Wetz90] die Möglichkeiten

- Implementierung semantischer Datenmodelle,
- Erweiterung eines bestehenden Datenbankmodells um mächtige Konzepte, die die Umsetzung der komplexen Beschreibungen erleichtert,
- *Abbildungsmethoden.*

In der DAIDA-Umgebung wird einerseits eine Zielsprache mit mächtigem Datenbankmodell verwendet und andererseits gemäß einer Abbildungsmethodik verfahren, innerhalb der die Systembeschreibung, ausgehend von einer Darstellung in einer Wissensrepräsentationssprache, in mehreren Schritten unter Verwendung von Zwischensprachen in die Implementationssprache DBPL übersetzt wird.
Ein wesentlicher Aspekt der Konsistenthaltung von Datenbanken besteht in der *Verwendung "sicherer" Transaktionen*, deren Durchführung, ausgehend von einem konsistenten Datenbankzustand, niemals einen inkonsistenten Zustand herbeiführt. (Ein Datenbankzustand wird als *konsistent* bezeichnet, wenn alle Integritätsbedingungen erfüllt sind.) Da i.a. nur ein kleiner Teil der Integritätsbedingungen in der Notation des verwendeten Datenbankmodells "modellinhärent" erfaßt werden kann, stellt sich das Problem, wie man die Sicherheit der Transaktionen relativ zu den übrigen Integritätsbedingungen sicherstellen kann. Hierzu gibt es u.a. Ansätze, die jeder Transaktion eine (eingeschränkte) Menge von Bedingungen zuordnen, deren Gültigkeit zur Laufzeit zu überprüfen ist und deren Nichterfüllung zu einem "Rücksetzen" der Transaktion führt. Es ist jedoch i.a. problematisch, die von einer Transaktion "bedrohten" Integritätsbedingungen zu ermitteln und *hinreichende* Tests zu ergänzen.
Die DAIDA-Umgebung behandelt insbesondere dieses Problem. DAIDA-Spezifikationen von Datenbankanwendungen beschreiben in ihrem dynamischen Teil eine Menge von *Transaktionen*. Zu jeder spezifizierten Transaktion ist in Form der sog. *Konsistenz-Beweisverpflichtung* formal zu verifizieren, daß die Konsistenz der Datenbank, ausgehend von einem konsistenten Zustand, erhalten wird. Der Nachweis von *Verfeinerungsbeweisverpflichtungen* für die konkreten Beschreibungen der Transaktionen stellt diese Eigenschaft letztlich auch für die erstellte Implementation sicher.

1.4 Zielsetzung der Diplomarbeit

Die Arbeit [Wetz90] befaßt sich hauptsächlich mit allgemeinen Aspekten formaler Methoden bei der Datenbankprogrammentwicklung. Die DAIDA-Entwicklungsumgebung wird dort global

diskutiert. Das B-Tool wird als *Abbildungsassistent* für regelbasierte Transformationen zwischen unterschiedlichen Beschreibungssprachen eingesetzt sowie als Werkzeug zur *Generierung* von Beweisverpflichtungen beschrieben.
In dieser Diplomarbeit soll der Einsatz des B-Tools als *Theorembeweiser* zum *Erbringen* der Beweisverpflichtungen behandelt werden. Zu diesem Zweck erscheint es als sinnvoll, zunächst eine *formale, systematische Sichtweise des B-Tools* zu erarbeiten, die eine Trennung der dem B-Tool zugrundeliegenden Logik von weiterreichenden, operationalen bzw. programmiersprachlichen Konzepten vornimmt. Eine derartige Beschreibung findet sich bisher nirgends. Diese *Darstellung des B-Tools als Deduktionssystem* ist der *Gegenstand des ersten Teils der Diplomarbeit.* Ausgehend von einer allgemeinen *Schichtenarchitektur für Deduktionssysteme* werden in Kapitel 2 die Besonderheiten der Architektur des B-Tools besprochen und der Begriff des *"offenen" Deduktionssystems* eingeführt. Kapitel 3 stellt, ausgehend von der Arbeit eines Projektpartners, eine für offene Deduktionssysteme geeignete *Metalogik* vor: es wird eine in der Originalarbeit nicht gegebene *Semantikdefinition* erarbeitet und ein wichtiges Resultat für das Schließen auf Metaebene bewiesen; ein weiterer Abschnitt zeigt Beziehungen zur Logikprogrammierung auf. Kapitel 4 schließlich diskutiert die Umsetzung der metalogischen Konzepte im B-Tool und beschreibt dessen Kontrollkonzepte; eine formale Darstellung des B-Tools wird erarbeitet.
Im *zweiten Teil der Arbeit* werden die *theoretischen Grundlagen der formal verifizierten Softwareentwicklung* zunächst DAIDA-unabhängig erarbeitet (Kapitel 5) und, darauf aufbauend, ein Überblick über die in der DAIDA-Umgebung verwendeten Beschreibungssprachen gegeben (Kapitel 6). Der Schwerpunkt liegt hierbei auf der Semantikbeschreibung der für die im dritten Teil der Arbeit verwendete, gegenüber der Darstellung in [Wetz90] erweiterten Verfeinerungssprache der *abstrakten Maschinen*. Deren Semantik wird *in Gestalt* der *Beweisverpflichtungen* gegeben. Der *dritte Teil der Arbeit* (Kapitel 7) behandelt schließlich das *Erbringen der Beweisverpflichtungen* von DAIDA-Programmentwicklungen. Eine Beispielentwicklung eines DBPL-Datenbankprogramms wird aufgestellt. Aus der Verarbeitung der einzelnen Beweisverpflichtungen resultieren *Teil*beweisverpflichtungen, die, etwas euphemistisch, als *Lemmas* bezeichnet werden[5]. Für die einzelnen, "typischen" Entwurfsentscheidungen resultieren bestimmte *generische Klassen von Lemmas*, für deren Nachweis geeignete Regelsammlungen und Beweisstrategien formuliert werden. Im Rahmen dieser Diskussion wird auch die Spezifikations- und Zielsprachenabhängigkeit der erstellten Beschreibungen besprochen.

[5] Die Bezeichnung "Lemma" suggeriert, daß die zu erbringenden Teilnachweise *trivial* sind, was jedoch i.a. *nicht* der Fall ist: die *Auswertung* der Beweisverpflichtungen ist hauptsächlich trivialer *syntaktischer Natur*, die resultierenden Lemmas hingegen stellen die *eigentlich zu erbringenden Beweisverpflichtungen* in prädikatenlogischer Notation dar.

Kapitel 2

Grundlagen von Deduktionssystemen

Eines der Teilziele dieser Arbeit besteht darin, die theoretischen Grundlagen des Beweisassistenten B-Tool zu erarbeiten und systematisch darzustellen. In diesem Abschnitt sollen nun elementare, den folgenden Ausführungen zugrundeliegende Begriffe des automatischen Beweisens eingeführt und kurz diskutiert werden. Auf eine *detaillierte* Behandlung von Aussagen- und Prädikatenlogik wird verzichtet (vgl. z.B. [ChLe73], [Ende72]).
Läßt man programmiersprachliche, "operationale" Komponenten wie Ein- und Ausgabefunktionen, Listen- und Zeichenkettenverarbeitung usw. außer Betracht, so kann man das B-Tool als *Deduktionssystem* beschreiben. Als Grundlage einer systematischen Beschreibung des B-Tools wird daher im folgenden zunächst ein einfaches Schichtenmodell für Deduktionssysteme besprochen. Aufbauend hierauf wird beschrieben, in welcher Form sich das B-Tool von anderen Deduktionssystemen unterscheidet. Hierbei soll für das B-Tool der Begriff des *offenen Deduktionssystems* eingeführt werden.

2.1 Ein Schichtenmodell für Deduktionssysteme

Nach [EiOh87] lassen sich Deduktionssysteme mit einer vierschichtigen Architektur beschreiben:

1. Die *symbolische Logik* legt die Syntax und Semantik der Formeln, über die das Deduktionssystem Beweise führen soll, fest.

2. Der *Logik-Kalkül* legt die Menge der syntaktisch zulässigen Ableitungsschritte fest, die das Deduktionssystem auf den Formeln der Logik durchführen darf.

3. Die *Repräsentation* der Formeln und Beweiskonfigurationen bestimmt die Darstellung der Zustände, die während einer Kette von Ableitungsschritten eingenommen werden.

4. Die *Steuerung* trifft aus der Menge der in einem bestimmten Zustand anwendbaren, vom Kalkül angebotenen Inferenzschritte eine möglichst gute Wahl.

Für die Prädikatenlogik erster Ordnung (kurz PL1) beispielsweise sind der Resolutionskalkül und der Gentzenkalkül bekannte Ansätze. Als effiziente Repräsentationsform für (auf *Klauseln* arbeitende) Resolutionsbeweiser werden sogenannte *Klauselgraphen* verwendet, für den Gentzenkalkül ist das *Tableauverfahren* zu nennen (vgl. [EiOh87]).

2.1.1 Die unterschiedlichen Ebenen des Schließens

Anhand des Schichtenmodells für Deduktionssysteme soll eine Differenzierung zwischen unterschiedlichen Ebenen des Schließens vorgenommen werden:

- **statisches Schließen:** die in der zugrundeliegenden Logik sowie im Logik-Kalkül gegebenen Konzepte des *semantischen Folgerns* und *syntaktischen Ableitens*.
- **dynamisches Schließen:** die operationale Realisierung der statischen Konzepte im Kontext einer bestimmten Repräsentation und einer Steuerung.

2.1.1.1 Statische Ebenen des Schließens

Die in der mathematischen Logik übliche Schreibweise für das *semantische Folgern* einer Formel G aus einer Menge von Annahmen Γ ist "$\Gamma \models G$". Nach Definition gilt "$\Gamma \models G$" genau dann, wenn alle Modelle von Γ ebenfalls Modelle der Zielformel G sind[1] . In der gewöhnlichen Definition der Prädikatenlogik entspricht dem semantischen Folgern auf der Objektebene die logische Implikation "$\Rightarrow$" Das *Deduktionstheorem* ([EiOh87], [ChLe73]) stellt die Verbindung zwischen "$\models$" und "$\Rightarrow$" her:

Theorem 2.1 (Deduktionstheorem) *Falls die Formeln in* $\{G\} \cup \Gamma$ *keine freien Variablen enthalten, so gilt*[2]

$$\Gamma \models G \quad \overset{\text{Meta}}{\Longleftrightarrow} \quad \models \Gamma \Rightarrow G \tag{2.1}$$

d.h. das Implikationskonnektiv "entspricht" unter der genannten Voraussetzung dem semantischen Folgern.
Logik-Kalküle legen Mengen zulässiger *syntaktischer* atomarer Ableitungsschritte auf den Formeln der betrachteten Logik fest. Sie werden in Abhängigkeit der gegebenen symbolischen Logik

[1] Ein *Modell* einer Formel ist eine *Interpretation* (d.h. eine spezifische Festlegung der semantischen Deutung von Konstanten-, Funktionen- und Prädikatensymbolen der unterliegenden Sprache), unter der die Formel den Wahrheitswert TRUE annimmt.

[2] Die Bezeichnung $\overset{\text{Meta}}{\Longleftrightarrow}$ wird verwendet, um den Unterschied zu den logischen Operatoren der *Objekt*sprache (hier das Implikationskonnektiv "$\Rightarrow$" der Prädikatenlogik) herauszustellen. Das Deduktionstheorem ist ein Beispiel für eine *metalogische Aussage*.

definiert und enthalten zusätzlich zu den syntaktischen Schlußregeln eine Menge von Formeln, sogenannte *Axiome*, die als tautologisch oder aber als widersprüchlich vorgegeben sind (*positive* bzw. *negative* Kalküle). Eine Formel G heißt *syntaktisch ableitbar* aus einer Formel H g.d.w. G durch wiederholte Anwendung der Schlußregeln und Axiome des Kalküls aus H hervorgeht (Schreibweise "$H \vdash G$"). Ein Kalkül heißt *korrekt*, falls unter wiederholter Anwendung der elementaren syntaktischen Ableitungsschritte auf die Axiome ausschließlich in der Logik semantisch folgerbare Formeln herleitbar sind. Ein Kalkül heißt *vollständig*, falls *alle* semantisch folgerbaren Formeln auch auf Kalkülebene (syntaktisch) ableitbar sind.
Das Deduktionstheorem stellt nun die Grundlage für die Mechanisierbarkeit von Beweisen dar. Ist ein Kalkül für PL1 korrekt und vollständig, so gilt unter der obengenannten Voraussetzung

$$\Gamma \models G \quad \overset{\text{Meta}}{\Longleftrightarrow} \quad \vdash \Gamma \Rightarrow G \tag{2.2}$$

Um "$\Gamma \models G$" zu zeigen, genügt es somit, die Formel "$\Gamma \Rightarrow G$" auf Kalkülebene abzuleiten, vorausgesetzt, der Kalkül ist korrekt.

2.1.1.2 Operationale Aspekte von Logikkalkülen: dynamisches Schließen

Im Kontext des automatischen Beweisens sind die Konzepte des *statischen* Schließens durch einen die operationalen Charakteristika eines gegebenen Beweisers berücksichtigenden Terminus zu ersetzen. Hierfür soll im folgenden die Bezeichnung *Ableitbarkeit* verwendet werden. Die in den Beweisern zulässigen atomaren Ableitungsschritte stellen i.a. eine Einschränkung gegenüber den im zugrundeliegenden Kalkül möglichen Schritten dar.
In Deduktionssystemen wird, meist aus Effizienzgründen, eine die Anwendung der syntaktischen Ableitungsschritte des Kalküls und damit den zu bearbeitenden Suchraum einschränkende *Steuerung* vorgegeben, wodurch z.T. die Vollständigkeit verlorengeht. Hier lassen sich fest vorgegebene Kontrollalgorithmen und benutzerdefinierbare Kontrollkonstrukte unterscheiden. Z.B. verwendet der in Interpretern der Programmiersprache PROLOG realisierte SLD-Resolutionskalkül[3] eine Steuerung, unter der die Vollständigkeit verloren geht (vgl. [Lloy84]): der durch eine das zu resolvierende Literal der Zielklausel festlegenden *Berechnungsregel* bestimmte *SLD-Baum* wird gemäß einer *Suchstrategie* durchsucht, durch die *Unvollständigkeit* resultiert (vgl. [Beck88]).

2.1.2 Ungenauigkeiten der Schichtenarchitektur

Die obenbeschriebene Schichtung von Deduktionssystemen darf nicht als strikt aufgefaßt werden. Am Beispiel der Interdependenzen der Repräsentationsschicht läßt sich dies verdeutlichen:

- **Interdependenzen mit der Kalkülebene:** Die Repäsentation soll eine effiziente Beweiszustandsdarstellung ermöglichen und aufzeigen, welche Kalkülregelinstanzen wo anwendbar sind. Um die Effizienz zu erhöhen, werden hier oft *logische Vereinfachungen* und eine *Entfernung als nutzlos angesehener Formeln*, wie beipielsweise im Resolutionskalkül die

[3] Beschränkt man sich auf den *logischen Kern* von PROLOG, so lassen sich Anfragen als *Existenzaussagen* auffassen, die durch den PROLOG-Interpreter durch syntaktisches Ableiten in einem speziellen Logikkalkül *konstruktiv*, d.h. unter Angabe von *erfüllenden Variablenbelegungen* bewiesen werden sollen ([Buer87]).

Entfernung von Klauseln mit nicht resolvierbarem Literal (*Isolationsregel*), vorgenommen. Dies entspricht genaugenommen einer Modifikation des zugrundeliegenden Logik-Kalküls, da auf Kalkülebene noch mögliche Ableitungen unmöglich werden.

- **Interdependenzen mit der Steuerungsebene:** eine Entscheidung zu einer bestimmten Repräsentation aus Effizienzgründen kann auch Auswirkungen auf die Wahl des Steueralgorithmus haben. Der in der logischen Programmierung eingesetzte SLD-Resolutionsbeweiser *wählt* (gemäß der Berechnungsregel) das zu resolvierende Literal anhand einer auf den Literalen der Zielklausel definierten totalen Ordnung ("von links nach rechts"); die neu hinzukommenden Teilziele sind bezüglich dieser Ordnung kleiner als die ursprünglichen Literale. Auf der Repräsentationsebene entspricht dieser Steuerung die effiziente Verwaltung eines *Zielkellers*: das auf dem Keller zuoberst liegende Teilziel wird, *ohne Betrachtung der übrigen Teilziele*, Zielliteral des nächsten Resolutionsschritts[4].

Hierbei wird deutlich: die Wahl einer bestimmten Repräsentationsform hat nicht nur Auswirkungen auf die Effizienz des Deduktionssystems; beispielsweise kann eine unvorsichtige Verwendung von *Eliminationsregeln* dazu führen, daß die *Vollständigkeit* des zugrundeliegenden Kalküls *verlorengeht*. Somit kann allgemein ein recht großer Unterschied zwischen logischen Formeln und Repräsentation der Formeln einerseits und zwischen Kalkülschlußregeln und den in der Repräsentation zulässigen Zustandsübergängen andererseits bestehen. In [EiOh87] wird für dieses Abbild auf Repräsentationsebene der Begriff des *logischen Zustandsübergangssystems* eingeführt. Zustände dieses Systems entsprechen den *Repräsentationen* der Zustände der Deduktionen, beispielsweise *Klauselgraphen*. Auf den Systemzuständen ist eine Übergangsrelation gemäß der von der Repräsentation angebotenen Ableitungsschritte definiert. Darüberhinaus sind noch Klassen ausgezeichneter Anfangs- und Endzustände mit bestimmter logischer Semantik (erfüllbar, unerfüllbar etc.) definiert. In einem weiteren Schritt lassen sich die Begriffe *Korrektheit* und *Vollständigkeit* auf logische Zustandsübergangssysteme übertragen.
Die interne Repräsentation der Systemzustände des B-Tools soll nicht gesondert besprochen werden. Sie dürfte der PROLOG-Repräsentation ähneln, wie die Diskussion der folgenden Kapitel verdeutlichen wird. In Kapitel 4 soll eine formale Darstellung des B-Tools als *logisches Zustandsübergangssystem* erarbeitet werden, in deren Rahmen die unter Berücksichtigung sowohl der Kalkülschlußregeln als auch der Steuerungsmechanismen möglichen Ableitungen mittels einer *Übergangsrelation* beschrieben werden.

2.2 Anwendung von Deduktionssystemen

Um ein Deduktionssystem in einem bestimmten Diskursbereich einzusetzen, bedarf es neben den vordefinierten Kalkülaxiomen und Kalkülschlußregeln weiterer Axiome. Der Benutzer des Deduktionssystems axiomatisiert den Anwendungsbereich, in dem geschlußfolgert werden soll, durch die Angabe von Formeln in der vom Beweiser unterstützten Logik. Ein Deduktionssystem mit geeigneter zugrundeliegender Logik kann so beispielsweise zum Beweis von Sätzen der Arithmetik oder der Gruppentheorie eingesetzt werden. In ersterem Fall sollte die zugrundeliegende

[4] Die Unvollständigkeit von PROLOG resultiert nicht hieraus, sondern aus dem Tiefensuche-Rahmenalgorithmus (vgl. u.).

Logik das Führen von *Induktionsbeweisen* ermöglichen, was bei der Verwendung von Prädikatenlogik zu theoretischen Problemen führt, da das Induktionsaxiom erst in der als unvollständig bekannten Prädikatenlogik zweiter Ordnung formulierbar ist. Im zweiten Fall sollten auf dem Deduktionssystem, gemäß der Gestalt der Axiome der Gruppentheorie, Gleichheitsbeweise führbar sein. Deduktionsverfahren, bei denen spezielle, die beiden genannten Anwendungsbereiche unterstützende Konzepte vorgegeben werden, sind z.B. ([BlBu87]):

- der die *konstruktive Definition* von Funktionen durch Algorithmen auf induktiv definierten Datenstrukturen sowie induktive Beweise hierüber ermöglichende Ansatz von Boyer und Moore ("*A Computational Logic*", [BoMo79]).
- das *Knuth-Bendix-Vefahren* für Gleichheitsbeweise.

Den "gewöhnlichen" Deduktionssystemen ist gemein, daß deren zugrundeliegende symbolische Logik und der eingesetzte Logik-Kalkül fest vorgegeben ist und nicht modifiziert werden kann. Es bietet sich die in Abbildung 2.1 dargestellte Situation.

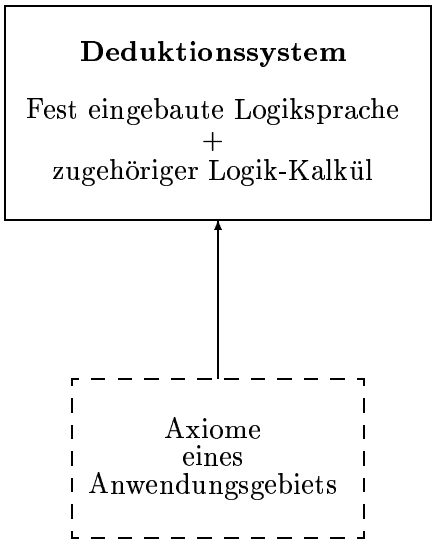


Abbildung 2.1: Anwendung "gewöhnlicher" Deduktionssysteme

2.3 Die offene Architektur des B-Tools

Der Architektur des Beweisassistenten B-Tool liegt der Gedanken zugrunde, dem *Anwender* die Wahl einer für die intendierte Anwendung geeigneten Logik sowie des Logik-Kalküls zu überlassen. Hieraus resultiert eine hohe Flexibilität. Die Architektur des B-Tools ist in Abbildung 2.2

skizziert[5] .

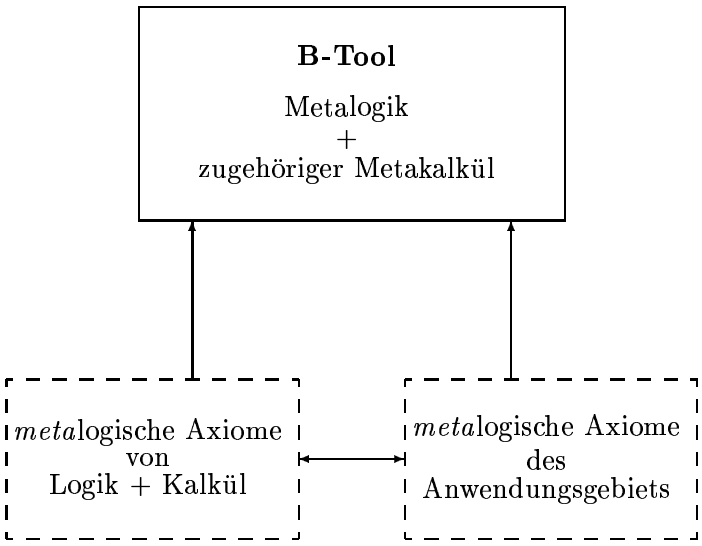


Abbildung 2.2: Anwendung des B-Tools

Im Unterschied zur Situation in gewöhnlichen Deduktionssystemen unterliegt die Logiksprache und der Logik-Kalkül ebenso der Benutzerwahl wie die Vorgabe der anwendungsspezifischen Axiome. Der Benutzer hat damit (innerhalb bestimmter Grenzen) die Möglichkeit, eine für sein Anwendungsgebiet als geeignet angesehene Logik und einen zugehörigen Kalkül im Rahmen der metasprachlichen Konzepte des B-Tools zu axiomatisieren. Man unterscheidet hierbei zwischen invarianten *metalogischen* Vordefinitionen und den auf *Objekt*ebene der Metalogik *beschreibbaren* Kalkülen.
Der Benutzer ist bei der Definition von Objektlogik, Objektkalkül und Anwendungsbereich *nicht* an starre, durch das Deduktionssystem vorgegebene syntaktische Defaults gebunden, sondern kann z.B. (mit bestimmten Einschränkungen) selbst Operatoren bestimmter Gestalt durch Angabe von Stelligkeit, Priorität und (im Falle von *binären* Operatoren) Richtung der Assoziativität[6] vorgeben.
Die soeben identifizierten Merkmale rechtfertigen die Einführung einer Bezeichnung, die Deduktionssysteme mit variabler Architektur wie der des B-Tools qualifiziert: das B-Tool wird im folgenden auch als *offenes Deduktionssystem* bezeichnet.

[5] Der horizontale Doppelpfeil soll herausstellen, daß die Axiomatisierung von Objektlogik und Kalkül sowie des Anwendungsbereichs auf derselben konzeptuellen Ebene stattfindet.

[6] Rechts- oder Linksassoziativität, d.h. das Vorgehen, gemäß dem fehlende Klammern bei iterierter Operatoranwendung zu ergänzen sind

2.4 Weiteres Vorgehen

In den folgenden zwei Kapiteln soll der Aufbau des B-Tools gemäß des Schichtenmodells für Deduktionssysteme systematisiert und diskutiert werden: Als Basis hierfür wird zunächst eine Metalogik eingeführt, in deren Rahmen die logischen Grundlagen der Ebenen 1 und 2 der Schichtenarchitektur für *offene* Deduktionssysteme modellierbar sind (Kap. 3). Die Metalogik hat hierbei insbesondere die Objektlogik sowie den Objektkalkül, in dem das offene Deduktionssystem arbeiten soll, als *Anwendungsgebiet*. Kapitel 4 diskutiert die *Realisierung* der metalogischen Konzepte im offenen Deduktionssystem *B-Tool* und identifiziert die unterschiedlichen *Steuerungsmechanismen*, womit die Komponenten der *vierten* Ebene der Schichtenarchitektur für das B-Tool beschrieben werden. Die operationale Semantik des B-Tools als Zustandsübergangssystem soll systematisiert werden. Außerdem werden Betrachtungen zur Mächtigkeit des B-Tools angestellt. Darüberhinaus soll der Aufbau des *Softwaresystems* B-Tool kurz skizziert werden.

Kapitel 3

Die Metalogik BL

In den vorangegangenen Ausführungen wurde die offene B-Tool-Architektur skizziert und vom Aufbau weniger variabler Deduktionssysteme abgegrenzt. Das in Abschnitt 2.1 diskutierte Schichtenmodell für Deduktionssysteme beschreibt jedoch auch die Architektur des B-Tools; lediglich Gestalt und Mächtigkeit der vordefinierten Logik und des vordefinierten Kalküls unterscheidet das B-Tool von anderen Deduktionssystemen.
Es soll nun zunächst, ausgehend von einem motivierenden Beispiel und orientiert an der Arbeit [Gard88] die Metalogik BL sowie ein auf BL definierter Metakalkül beschrieben werden. Im Rahmen der in Kapitel 2 eingeführten Schichtenarchitektur entspricht dies einer für *offene* Deduktionssysteme adäquaten Festlegung der beiden obersten Schichten durch eine spezielle Logik und einen darauf definierten Kalkül.
Anhand von Beispielen wird gezeigt, daß sich bekannte Logik-Kalküle der Prädikatenlogik und spezielle Anwendungsgebiete wie Gleichheitstheorien und Arithmetik im Rahmen der Metasprache axiomatisieren lassen. Weitere Abschnitte befassen sich mit dem Zusammenhang zwischen unterschiedlichen Sichtweisen metalogischer Axiome: analog zur auch im Bereich der Logikprogrammierung verwendeten Terminologie läßt sich zwischen modelltheoretischer und beweistheoretischer Sicht differenzieren. In zwei weiteren Abschnitten werden die Unterschiede der im B-Tool vordefinierten Logik gegenüber der idealisierenden Sprache BL besprochen und eine die Abbildung 2.2 präzisierende Sicht des B-Tools erarbeitet.

3.1 Motivation der metasprachlichen Konstrukte

Abbildung 3.1 zeigt die Gestalt typischer Objektkalkülschlußregeln und Axiome, wie sie in der Literatur (z.B. [EiOh87], [Rich78]) üblicherweise formuliert werden.
Die Formel auf der linken Seite beschreibt die für prädikatenlogische Kalküle typische *Schlußregel Modus Ponens*; die andere Formel beschreibt ein *Axiom*, das sog. Gesetz des ausgeschlossenen Dritten. Die Notationsform besagt, daß die unter dem Strich stehende Formel aus den darüberstehenden im Rahmen des Kalküls syntaktisch abgeleitet werden kann. Die Großbuchstaben P und Q sind Platzhalter, die durch *beliebige wohlgeformte Formeln* der dem Kalkül zugrundeliegenden Logik ersetzt werden können. *Axiome* unterscheiden sich in dieser Schreibweise von

$$\frac{P \quad P \Rightarrow Q}{Q} \qquad \frac{}{P \vee \neg P}$$

Abbildung 3.1: Beispiel: Objektkalkülschlußregel und Axiom

Schlußregeln dadurch, daß oberhalb der Linie keine Formeln stehen.
Das kleine Beispiel erlaubt es, die strukturellen Anforderungen an die Logik eines offenen Deduktionssystems zu identifizieren. Die Metasprache sollte die notwendigen Konzepte anbieten, Axiome und Schlußregeln in ihrem Rahmen *formal* abzubilden. Zu diesem Zweck muß die Möglichkeit geschaffen werden, die in der Beschreibung von Logikkalkülen üblicherweise informell formulierten Nebenbedingungen formal zu erfassen: in obigem Beispiel besteht so die Einschränkung, daß die Platzhalter P und Q nur durch *syntaktisch wohlgeformte* Formeln ersetzt werden dürfen. Diese und andere Nebenbedingungen wie "ist Variable", "ist Term" und "ist nicht frei in" sollten in der Metasprache explizit darstellbar sein.

3.2 Sequenzenlogik und Sequenzenkalkül

In diesem Abschnitt sollen eine für offene Deduktionssysteme adäquate Logik sowie ein darauf definierter Kalkül, entsprechend den beiden oberen Ebenen der Schichtenarchitektur für Deduktionssysteme, beschrieben werden. Diese *Meta*logik wird derart gewählt, daß sie die Objektlogik sowie den Objektkalkül, in dem das offene Deduktionssystem arbeiten soll, als *Anwendungsgebiet* hat. Es wird sich allerdings zeigen, daß sich Metalogik und Metakalkül derart geschickt wählen lassen, daß die Unterscheidung zwischen metalogischem und objektlogischem Schließen irrelevant ist (Abschnitt 3.4).
Ein möglicher Kandidat für eine Metasprache ist die Prädikatenlogik. Das soeben diskutierte Beispiel verdeutlicht jedoch, daß zur formalen Abbildung von Kalkülregeln, Axiomen und Axiomenschemata die üblichen prädikatenlogischen Konnektive wie Implikation, Negation, Disjunktion usw. in der *Meta*sprache nicht benötigt werden. Es genügt *ein* logisches Konnektiv, welches zur formalen Darstellung der syntaktischen Ableitungsbeziehung im Objektkalkül dient: den als horizontale Linie geschriebenen binären *Sequenzoperator*[1]. Seine Operanden oberhalb und unterhalb der Linie werden *Antezedenten* bzw. *Konsequent* genannt. Die im Folgenden beschriebene Metalogik BL besitzt ausschließlich Formeln von Sequenzengestalt, so daß man sie als *Sequenzenlogik* bezeichnet.
Es muß betont werden, daß die Beschränkung der Metasprache auf Sequenzen *keine* Einschränkung der formalisierbaren Objektsprachen darstellt. Hier sind alle üblichen logischen Konnektive formulierbar. Der Zusammenhang zwischen Objektsprache und Metasprache wird in einem gesonderten Abschnitt diskutiert. Die *ursprüngliche Version der Sequenzenlogik* geht auf Gerhard Gentzen zurück, der in [Gent34] einige heute nach ihm benannten Kalküle für unterschiedliche Varianten der Prädikatenlogik vorstellte. Die Sequenzenlogik stellt, zusammen mit

[1] Genaugenommen handelt es sich um eine unendliche *Familie* von Sequenzkonnektiven mit unterschiedlicher Stelligkeit. Die Beschränkung euf einen *binären* Sequenzenoperator ist möglich, wenn man einen logischen Konjunktionsoperator ergänzt.

den darauf definierten Schlußregeln des Sequenzenkalküls, eine organisatorische Abwandlung dieser Kalküle dar. Ihre Idee ist es, die während eines Beweises bestehenden Abhängigkeiten der Form "$G_1 \vee \ldots \vee G_m$ läßt sich aus $F_1 \wedge \ldots \wedge F_n$ syntaktisch ableiten" explizit in Form einer Sequenz darzustellen. Die syntaktischen Schlußregeln des Sequenzenkalküls lassen sich durch die Schlußregeln des Gentzenkalküls rechtfertigen und erlauben eine Verlagerung eines Beweises vom Gentzenkalkül auf die Sequenzebene. Für eine genaue Motivation und Beschreibung der herkömmlichen Sequenzenlogik sowie aller Schlußregeln des Sequenzenkalküls wird auf die Literatur verwiesen (z.B. [Rich78]).

3.2.1 Ein Beispiel für die Verwendung der Metasprache

Bevor nun eine genaue Beschreibung von Syntax und Semantik der Metalogik gegeben wird, soll deren Verwendung anhand eines Beispiels verdeutlicht werden. Abbildung 3.2 zeigt die *formale* Abbildung von Regeln eines Objektkalküls.
Der Erste der beiden Sequenzen beschreibt die *formale* Axiomatisierung der in Abbildung 3.1

$$\frac{\begin{array}{c} frm\,P \;\; frm\,Q \\ \vdash P \\ \vdash P \Rightarrow Q \end{array}}{\vdash Q} \qquad\qquad \frac{\begin{array}{c} var\,X \;\; trm\,E \;\; frm\,P \\ X \setminus P \end{array}}{[X := E]P == P}$$

Abbildung 3.2: Beispiel für die formale Kalkülabbildung auf Sequenzen

informell dargestellten Schlußregel *Modus Ponens* im Rahmen der Metasprache BL. Das Hilfssymbol "$\vdash$" steht für *syntaktische Ableitbarkeit* im Objektkalkül. Die Buchstaben P, Q, X und E stellen *Variable der Metalogik* dar und stehen für *beliebige* Zeichenketten über einem festen (endlichen) Alphabet Σ, welches zur Definition der üblichen Objektlogiken geeignet gewählt ist[2]. $frm\,P$ formalisiert die Nebenbedingung, daß P eine wohlgeformte prädikatenlogische Formel sein muß. Damit derartige Nebenbedingungen sinnvoll sind, bedarf es einer geeigneten *Axiomatisierung* durch Angabe zusätzlicher Sequenzen, deren Gestalt sich an Syntax und Semantik der unterliegenden Objektlogik richtet. Durch eine an der Objektsprache orientierten Axiomatisierung von Hilfssymbolen frm, var usw. können dann Einschränkungen formuliert werden.
Die zweite angegebene Sequenz ist ein Beispiel für die Axiomatisierung eines Hilfssymbols, in diesem Fall für die *Substitution* $[\_ := \_]\_$. Hier wird festgehalten, daß eine Substitution eines Terms für eine Variable keine Auswirkungen auf eine Formel hat, sofern die Zielvariable nicht frei in der Zielformel vorkommt (formalisiert durch Hilfssymbol "$\setminus$"). Das Hilfssymbol "$==$" ist das Gleichheitsprädikatensymbol der Metasprache BL.
Neben der Formalisierung der eigentlichen Schlußregeln und Axiome des Objektkalküls bedarf es also noch zusätzlicher Sequenzen zur Axiomatisierung der *Hilfssymbole* im Rahmen der Metalogik.

[2] Aus konzeptuellen Gründen erscheint es zweckmäßig, Σ als objektlogik*unabhängig* anzusehen und die Objektsprache ausschließlich durch Axiomatisierung zu erfassen. Bei der Semantikdefinition der Metasprache wird die Rolle der Menge Σ^* als *metalogische Domäne* deutlich werden.

3.2.2 Definition von Metalogik und Metakalkül

Das den Interpretationen über der Metasprache implizit zugrundeliegende Universum ist die Menge aller Zeichenketten über dem Alphabet Σ (vgl.u. , Abschnitt 3.2.2.2). Durch die Axiomatisierung von Hilfssymbolen wie "$\vdash$", *var*, *trm*, frm und "\" wird die Gültigkeit der Sequenzen auf Zeichenketten beschränkt, die von einer für die entsprechende syntaktische Klasse zulässigen Gestalt sind. Solchen Symbolen kommt somit die Rolle von *Prädikatsymbolen der Metasprache* zu; einziges *logisches Konnektiv* ist der *Sequenzoperator* (vgl. Syntaxdefinition unten).
Die Argumente der Metaprädikate, d.h. die *Meta-Terme*, sind Metavariable, Metakonstanten oder Metafunktionen einer bestimmten Stelligkeit mit entprechenden Untertermen. Sie werden in Abhängigkeit der vom Benutzer gewählten Objektlogik sowie des Anwendungsbereichs der Objektlogik eingeführt, was noch deutlich werden wird.
Die formale Beschreibung der Metalogik BL geschieht durch Definition der Syntax und Semantik von Sequenzen und durch Angabe der Schlußregeln des Sequenzen*kalküls*.

3.2.2.1 Syntax der Sequenzenlogik

Die Syntax von Sequenzen ist gemäß Abbildung 3.3 definiert. Dabei steht P_n für ein beliebiges

$$SEQUENT := \frac{FORMEL \ldots FORMEL}{FORMEL}$$

$$FORMEL := P_n(TERM_1 \ldots TERM_n)$$

$$TERM := f_m(TERM_1 \ldots TERM_m)$$

$$\mid VARIABLE$$

$$\mid CONSTANT$$

Abbildung 3.3: Syntax der Sequenzenlogik

n-äres BL-Prädikatensymbol und f_m für ein beliebiges BL-Funktionensymbol. Diese Metasymbolklassen werden in Abhängigkeit der Objektlogik definiert, dürfen jedoch nicht mit Objektlogiksymbolen verwechselt werden. Übliche BL-Prädikatensymbole sind die oben erwähnten ein- bzw. zweistelligen Symbole "$\vdash$", *var*, *trm*, frm, "\" und "==".
Im Unterschied zu der weithin üblichen Definition der Sequenzensyntax haben die hier betrachteten Sequenzen nur *eine* Konsequenzformel. Dies geschieht im Hinblick auf die im B-Tool formulierbaren "*Regeln*", die einen atomaren Regelkopf besitzen[3] und als Sequenzen mit der hier definierten Syntax aufgefaßt werden.

[3]im Rahmen der logischen Programmierung entspricht dies der Verwendung von *Hornklauseln*, die den Einsatz eines effizienten Abarbeitungsalgorithmus ermöglichen

3.2.2.2 Semantik der Sequenzenlogik

Die Definition eines syntaktischen Ableitungsoperators durch einen Logik-Kalkül geschieht gewöhnlich im Hinblick auf die formale Semantikdefinition der Diskurslogik. Für die hier betrachtete Sequenzenlogik sollte ein semantischer Folgerungsoperator "$\models_{meta}$" definiert sein, der durch die Schlußregeln des noch zu definierenden Sequenzenkalküls respektiert wird. Eine solche Definition wird jedoch in der den bisherigen Betrachtungen zugrundeliegenden Arbeit [Gard88] nicht gegeben. Der im nachfolgenden Abschnitt beschriebene Sequenzenkalkül kann somit nicht formal gerechtfertigt werden, da weder Korrektheits- noch Vollständigkeitsbetrachtungen durchführbar sind. Es wird jedoch zweierlei deutlich werden:

- gemäß seiner Rolle hat der Sequenzenkalkül zunächst die Aufgabe, das Schließen im axiomatisierten objektlogischen Kalkül "simulierbar" zu machen.
- eine formale Semantikdefinition für die Sequenzenlogik kann leicht nachgeholt werden (vgl. u. , Abschnitt 3.2.2.4).

Orientiert an [Rich78] wird nun eine Semantikdefinition für die *ursprüngliche Definition der Sequenzenlogik nach Gentzen* vorgestellt und der im Fall der Sequenzenlogik BL vorliegenden Situation gegenübergestellt. Die Sequenzenlogik stellt, zusammen mit den darauf definierten Schlußregeln eines Sequenzenkalküls, eine organisatorische Abwandlung bestimmter prädikatenlogischer Kalküle nach Gentzen (vgl. [Gent34]) dar. Ihre Idee ist es, die während eines Beweises bestehenden Abhängigkeiten der Form "$G_1 \vee \ldots \vee G_m$ läßt sich aus $F_1 \wedge \ldots \wedge F_n$ syntaktisch ableiten" explizit in Form einer *Sequenz* darzustellen. Die syntaktischen Schlußregeln des Sequenzenkalküls lassen sich durch die Schlußregeln des Gentzenkalküls rechtfertigen und erlauben eine Verlagerung eines Beweises vom Gentzenkalkül auf die Sequenzebene (vgl. [Rich78]). Im Gegensatz zur Metalogik BL enthalten die Antezedenten und Konsequenten jeweils prädikatenlogische Formeln, jedoch keine metalogischen Prädikate. Dort orientiert sich die Semantikdefinition für Sequenzen dann an der engen Verwandtschaft des Sequenzenoperators und der Implikation "$\Rightarrow$" (vgl. o. , *Theorem* 2.1) und es wird (leicht abgekürzt) erklärt:

Definition 3.1 *Eine Sequenz*

$$S \;=\; \frac{F_1, \ldots, F_n}{G_1, \ldots, G_m}$$

heißt allgemeingültig *(geschrieben "$\models S$") g.d.w. die Formel $F_1 \wedge \ldots \wedge F_n \;\Rightarrow\; G_1 \vee \ldots \vee G_m$ in der Prädikatenlogik allgemeingültig ist.*

Diese Definition ist nur möglich, da die Antezedenten und Konsequenten selbst ausschließlich *prädikatenlogische Formeln* enthalten. Die Semantikdefinition der Sequenzen ergibt sich damit direkt über die entsprechenden prädikatenlogischen Begriffe; Interpretations- und Modellbegriff werden für die Sequenzenlogik nicht gesondert eingeführt.
In der Metalogik BL hingegen werden die objektlogischen Formeln als *Terme*[4] dargestellt. Eine

[4] und damit als *Objekte* des Diskursbereichs

Semantikdefinition der Metasprache basiert auf der Betrachtung von *Interpretationen*[5] der metalogischen Konstanten-, Funktionen- und Prädikatensymbole. Allgemeingültigkeit einer Sequenz heißt dann Unabhängigkeit der Gültigkeit von der gewählten Interpretation. Da die objektlogischen Konnektive als Metafunktionensymbole erfaßt werden, ist deren metalogische Semantik abhängig von der gewählten Interpretation, obwohl diese auf der Objektebene eine *feste* Bedeutung besitzen. Allgemeingültigkeit einer Sequenz heißt dann Unabhängigkeit der Gültigkeit von der gewählten Interpretation.
Die Menge der zu betrachtenden Interpretationen ist jedoch einzuschränken: der innere Aufbau der objektlogischen Formeln ist für die Metasemantik irrelevant, da lediglich das syntaktische Folgern innerhalb eines im Rahmen der Metalogik axiomatisierten objektlogischen Kalküls nachvollzogen werden soll. Korrektheit bzw. Vollständigkeit des auf der Metaebene durch Anwendung der Regeln des Metakalküls "simulierten" objektlogischen syntaktischen Folgerns lassen sich auf Korrektheit bzw. Vollständigkeit des axiomatisierten objektlogischen Kalküls zurückführen. Dies ist die formale Begründung dafür, daß es genügt, objektlogische Formeln als ("semantisch strukturlose") Zeichenketten aufzufassen . In [Gard88] wird demgemäß eine *semantische Funktion* $\| \_ \|$ definiert, die den Zusammenhang zwischen metasprachlichen Termen und objektlogischen Formeln und Termen, dargestellt als Zeichenketten über dem Alphabet Σ, herstellt: für die objektlogischen *Konnektive* werden *metalogische Funktionen* derselben Stelligkeit eingeführt. Damit wird beispielsweise der Meta*term* $P \Rightarrow_{meta} Q$ als eine entsprechende *Zeichenkette* aus Σ^* *gedeutet*[6]:

$$\| P \Rightarrow_{meta} Q \| = \| P \| \Rightarrow_{obj} \| Q \| \ \in \ \Sigma^*$$

Dazu wird über die semantische Funktion auch der Zusammenhang zwischen Metakonstantensymbolen und Objektkonstanten- und Variablensymbolen hergestellt: Für jedes Objektkonstanten- bzw. Variablensymbol $s_{obj} \in \Sigma^*$ existiert ein Meta-Konstantensymbol s_{meta}, sodaß

$$\| s_{meta} \| = \ s_{obj}$$

Somit hat die Manipulation von BL-Sequenzen eine Entsprechung auf Objektlogikebene, die durch die semantische Funktion $\| \_ \|$ hergestellt wird.
Also sind bei Allgemeingültigkeitsbetrachtungen nur Interpretationen $I = (D, \Phi, \Psi)$ relevant, die diese Beziehung berücksichtigen, d.h. folgende Eigenschaften haben:

- Die Metadomäne D ist gleich der oben definierten Zeichenkettenmenge Σ^*.
- Die Abbildung Φ legt die Bedeutung der Funktionen und Konstanten[7], orientiert an der semantischen Funktion $\| \_ \|$, über Σ^* fest: für die anhand der *objektlogischen Konnektive* eingeführten Metafunktionssymbole f_{meta} gilt $\Phi(f_{meta}) \ = \ f_{intern}$, wobei

[5] im formallogischen Sinn ein Tripel $I = (D, \Phi, \Psi)$, durch das eine Menge D von Objekten eines Diskursbereichs, eine die Semantik der Funktionen und Konstanten über D festlegende Abbildung Φ und eine die Semantik der Prädikatensymbole festlegende Abbilldung Ψ festgelegt wird, vgl. z.B. [ChLe73]

[6] Der Begriff "interpretiert" wird bewußt nicht gewählt, da die semantische Funktion nicht mit der im Folgenden diskutierten formallogischen Interpretation der Metasprache übereinstimmt.

[7] auffaßbar als Funktionen der Stelligkeit 0

$f_{intern} : (\Sigma^*)^k \longrightarrow \Sigma^*$ eine Funktion gleicher Stelligkeit ist, deren *Anwendung* auf die (interpretierten) Argumente genau die durch Anwendung der semantischen Funktion entstehende Zeichenkette ergibt. Die Funktion f_{intern} übernimmt dabei die Rolle der bei Beschreibung der semantischen Funktion nicht explizit angegebenen semantischen Deutung des Funktionen*symbols* f_{meta} über der Domäne Σ^*.
Dieselbe Festlegung gilt für die den *objektlogischen Funktionssymbolen* entsprechenden Metafunktionssymbolen sowie für das Substitutionsfunktionssymbol $[\_ := \_]\_$. Die eigentliche *Semantik* der Substitutionsfunktion (Ersetzen aller freien Vorkommen einer Variablen durch den angegebenen Term) hingegen wird nicht durch die semantische Funktion erfaßt. Sie ist abhängig von der individuellen Objektlogik definiert und wird daher erst im Rahmen der Objektlogik-Axiomatisierung durch Gleichheitsbeziehungen festgelegt.

- Die Abbildung Ψ ordnet jedem n-stelligen Metaprädikat P eine Abbildung $P : (\Sigma^*)^n \longrightarrow \{TRUE, FALSE\}$ zu, wodurch die Menge der erfüllten Grundinstanzen von P über Σ^* festgelegt werden. Für Ψ werden keine Einschränkungen formuliert. Die Festlegung der Semantik der Metaprädikate geschieht, objektlogikabhängig, durch Formulierung einer entsprechenden *Axiomenmenge* Λ, die die Objektlogik und den Objektkalkül sowie ggf. zusätzlich einen spezifischen Anwendungsbereich beschreibt. Allgemeingültigkeitsbetrachtungen von Sequenzen beschränken sich auf diejenigen Festlegungen von Ψ, die die als Axiome vorgegebenen Sequenzen (entsprechend der noch vorzunehmenden Semantikdefinition, vgl. Abschnitt 3.2.2.4) erfüllen.

Für eine derartige Vorab-Festlegung der Domäne sowie der Interpretation der Funktionen wird in [Lloy84] die Bezeichnung *Vorinterpretation*[8] eingeführt. Dies entspricht einer Einschränkung der Allgemeingültigkeits- und Folgerbarkeitsbetrachtungen auf einen speziellen, intendierten Anwendungsbereich. In diesem Fall sollen Formeln *beliebiger* Objektlogiken als *Zeichenketten* dargestellt und manipuliert werden. Die Vorinterpretation nimmt daher die Vorbelegungen gemäß dieser vorgegebenen Semantik vor. Die übrigen, von der Wahl der speziellen Objektlogik abhängigen Einschränkungen werden dann durch *Axiomatisierung* erfaßt.
Diese Situation ist *nicht* vergleichbar mit der in PL1 unter bestimmten Voraussetzungen möglichen Beschränkung auf *Herbrandinterpretationen*[9]. In letzterem Fall gibt es *strukturelle Gründe*, die eine derartige Einschränkung auf ein ausgezeichnetes Universum[10] sowie eine Vorinterpretation der Konstanten- und Funktionensymbole rechtfertigen: die Unerfüllbarkeit[11] unter allen Herbrandinterpretationen ist bereits hinreichend für die Unerfüllbarkeit unter *beliebigen* Interpretationen. Eine solche Rechtfertigung gibt es jedoch für die in BL vorgenommene Vorinterpretation nicht.
In [Beck88] wird das Problem der einschränkenden Funktion der vom Benutzer *intendierten Interpretation*[12] in Bezug auf die Semantikdefinition von *Logikprogrammen* diskutiert. In der üblichen Semantikdefinition werden die Konsequenzen eines Logikprogrammes anhand seiner Herbrandmodelle definiert, was im Vergleich zur eingeschränkten Interpretationswahl zu einem eventuell zu restriktiven Allgemeingültigkeitsbegriff führen kann. Dies verdeutlicht nochmals den Unterschied zwischen Vorinterpretationen und Herbrandinterpretationen.

[8] englisch *pre-interpretation*

[9] vgl. z.B. [ChLe73]

[10] Herbranduniversum

[11] Die Beschränkung auf Unerfüllbarkeitsbetrachtungen bedeutet keine Einschränkung, da gilt: P allgemeingültig $\Leftrightarrow \neg P$ unerfüllbar; vgl. auch [EiOh87].

[12] Dieser Begriff wird dort nur informal umschrieben und umfaßt *beliebige* Einschränkungen bei der Wahl der Interpretationskomponenten D, Φ und Ψ.

3.2.2.3 Der Sequenzenkalkül

Die in [Rich78] definierten Axiome und Schlußregeln eines Sequenzenkalküls sind spezifisch für die Prädikatenlogik gewählt. Die Schlußregeln lassen sich gemäß ihrer Funktion in zwei Gruppen unterteilen:

1. *Strukturregeln*, die *nicht* auf den inneren Aufbau der prädikatenlogischen Formeln Bezug nehmen und nur eine Umorganisation der in gegebenen Sequenzen vorliegenden Information vornimmt.
2. *logische Regeln*, die auch den inneren Aufbau der Formeln berücksichtigen und damit den beweistheoretischen Gehalt der logischen Konnektive repräsentieren (*Einführungs- und Eliminationsregeln*).

Unter Bezugname auf einen als korrekt und vollständig bekannten Hilberttypkalkül für die betrachtete Prädikatenlogik (als syntaktisches Äquivalent für das semantische Folgern über "$\models$") ist dann die Korrektheit und Vollständigkeit des so definierten Sequenzenkalküls für die Sequenzenlogik nachweisbar.
Die in unserem Fall auf den Sequenzformeln der Metalogik BL definierten syntaktischen Schlußregeln des Sequenzenkalküls (vgl. Abbildung 3.4) sind, unabhängig von einer speziellen Objektlogik, vordefiniert und können somit nur *Strukturregeln* sein. Deren Wahl wird in [Gard88] nicht näher motiviert. Der Doppelstrich steht für das syntaktische Schließen auf Sequenzformeln von

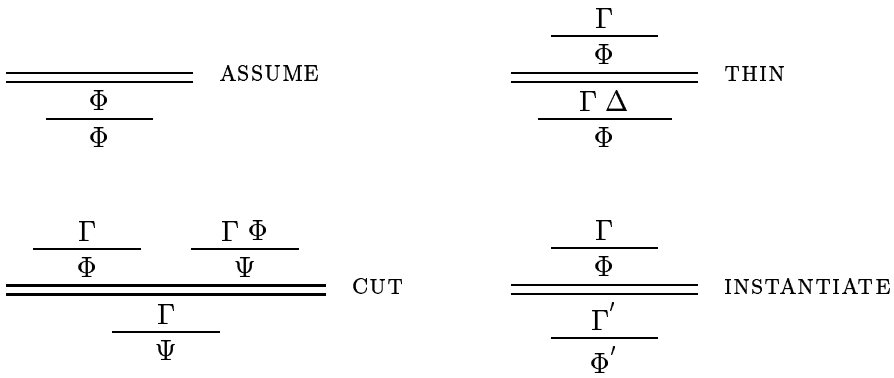


Abbildung 3.4: Schlußregeln des Sequenzenkalküls

BL: liegen die Sequenzen oberhalb des Doppelstrichs vor, so kann die (einzelne) Sequenz unterhalb des Doppelstrichs abgeleitet werden. Γ' und Φ' entstehen aus Γ bzw. Φ durch Anwendung (derselben) Substitution von (Meta)termen für (Meta)variable. Im Unterschied zur Darstellung in [Rich78] wird keine Aussage darüber gemacht, ob die Antezedens-Sequenzen als geordnet oder ungeordnet angesehen werden. Geht man von einer *geordneten Darstellung* aus, so dürfte bei der Anwendung des CUT jeweils nur über die am weitesten rechts stehende Sequenz "geschnitten" werden, was den weiter unten stehenden Beweis eines grundlegenden Resultats unmöglich macht. Ein formaler Ansatz geht daher von *Listen* von Antezedens-Sequenzen aus und definiert, wie in [Rich78], zusätzliche Schlußregeln zur Vertauschung von Sequenzen.

3.2.2.4 Definition eines semantischen Folgerungsoperators auf BL

Das in [Gard88] gewählte Vorgehen liefert ohne weiteres keine formale, sondern eine lediglich intuitive Rechtfertigung für den angegebenen Sequenzenkalkül. Um die Schlußregeln des Metakalküls *formal* zu rechtfertigen, ist ein semantischer Folgerungsoperator "$\models_{meta}$" zu definieren, bezüglich dem die Korrektheit und Vollständigkeit des hier definierten Sequenzenkalküls, d.h. von "$\vdash_{meta}$", nachzuweisen ist.
Diese Lücke kann jedoch, orientiert an Definition 3.1, geschlossen werden: man erhält eine geeignete Semantikdefinition, indem man dem Sequenzenoperator die Semantik des logischen Implikationskonnektivs verleiht und zusätzlich die logische Konjunktion in die *Meta*sprache aufnimmt:

Definition 3.2 *Die Semantik metalogischer Sequenzen ergibt sich durch die Vereinbarung*

$$\frac{F_1, \ldots, F_n}{F} \quad := \quad (F_1 \wedge \ldots \wedge F_n) \Rightarrow F$$

Diese Definition ist gerechtfertigt, da der Sequenzenoperator dazu dient, die syntaktische Folgerbarkeit im *Objekt*kalkül darzustellen. Dies kann auf Metaebene im Rahmen einer einfachen Logik geschehen, die nur die aussagenlogischen Verknüpfungen der Implikation und der Konjunktion enthält. Die Implikation "modelliert" dabei, gemäß ihrer beweistheoretischen Bedeutung, das syntaktische Folgern des Objektkalküls in der Metalogik. Wie bereits oben begründet, braucht und kann im Rahmen dieser Definition kein Bezug auf die logischen Konnektive der axiomatisierten Objektsprache genommen werden.
Die Beweise von Korrektheit und Vollständigkeit des zuvor definierten *meta*logischen Kalküls relativ zur soeben definierten Semantik sind analog zu den entsprechenden Beweisen für die Prädikatenlogik führbar. Die Aufgabe wird durch das Fehlen von Quantoren erleichtert ([Gard90])[13].

3.3 Axiomatisierung eines Prädikatenkalküls nach Hilbert

Als Beispiel soll nun eine Axiomatisierung eines prädikatenlogischen Kalküls nach Hilbert im Rahmen der Metasprache BL vorgestellt werden.

3.3.1 Axiomatisierung der Metagleichheit

Als Basis für die durchzuführende Axiomatisierung ist das bereits in einem zuvor gezeigten Beispiel verwendete Metaprädikat "==" im Rahmen der Metasprache als sog. *Metagleichheit* zu axiomatisieren. Die benötigten Sequenzen, die elementare Gleichheit definieren sowie Termersetzungen in metalogischen Funktionen und Prädikaten rechtfertigen, finden sich in Abbildung

[13] Hierbei sind allerdings die Vorbelegungen der intendierten Interpretation (vgl. o. , Abschnitt 3.2.2.2) wohl irrelevant. [Gard90] macht hierzu keine Aussage.

3.5[14]. Unter Berücksichtigung der besprochenen Vorinterpretation steht "==" für die Gleichheit

$$\frac{}{X == X} \qquad \frac{X_1 == Y_1, \ldots, X_n == Y_n}{F(X_1, \ldots, X_n) == F(Y_1, \ldots, Y_n)}$$

$$\frac{\begin{array}{c} X_1 == Y_1, \ldots, X_n == Y_n \\ P(X_1, \ldots, X_n) \end{array}}{P(Y_1, \ldots, Y_n)}$$

Abbildung 3.5: Axiomatisierung der Metagleichheit

von *Zeichenketten* über dem Objektsprachenalphabet Σ^*. Die angegebenen Sequenzen rechtfertigen die Anwendung von Termersetzungsregeln in der *Meta*sprache und dürfen nicht mit Termersetzungsregeln im Objektkalkül verwechselt werden. Das Prädikat "==" findet beispielsweise bei der Axiomatisierung der syntaktischen Auswirkungen von Substitutionen Verwendung; z.B. kann formuliert werden, daß eine Substitution für eine in einer objektlogischen Formel gebundenen Variablen keine Auswirkungen hat (vgl. Abb. 3.2).
Um Gleichheit in der *Objekt*sprache einzuführen, ergänzt man Sequenzen, durch die ein ausgezeichnetes *Objekt*sprachenprädikat "=" axiomatisiert wird. Termersetzungsschritte in der Objektsprache sind ebenfalls formalisierbar, indem man einen geeigneten objektsprachlichen Gleichheitskalkül unter Verwendung des Metaprädikats "$\vdash$" in BL axiomatisiert. Objektlogische Termersetzungsschritte sind dann ebenso wie gewöhnliche objektlogische Kalkülanwendungen im Rahmen des Metakalküls "nachvollziehbar".

3.3.2 Axiomatisierung von Objektlogik und Kalkül

Die zur Axiomatisierung von Objektlogik und Kalkül zu formulierenden Sequenzen lassen sich drei unterschiedlichen Klassen zuordnen:

1. Sequenzen, die *zur Axiomatisierung der Metaprädikatensymbole var*, *trm*, *frm* und "\" sowie der Objektebenen-Substitution anhand von Syntax und Semantik der betrachteten Objektlogik definiert werden. Insbesondere wird hier (indirekt) die Objektsprachen-Syntax erfaßt.

2. Sequenzen, die die *Axiome* bzw. *Axiomenschemata* des betrachteten Objektlogik-Kalküls abbilden. (Axiomatisierung des Metaprädikatensymbols "$\vdash$")

3. Sequenzen, die die *Schlußregeln* des betrachteten Objektlogik-Kalküls abbilden. (Axiomatisierung des Metaprädikatensymbols "$\vdash$")

[14]Symmetrie- und Transitivitätsaxiome werden wie in [Gard88] nicht angegeben, sind jedoch für ein vollständig formales Vorgehen notwendig.

Um die Darstellung auf Metaebene zu ermöglichen, müssen die benötigten Metasymbole eingeführt werden. Betrachtet man die unterschiedlichen Symbolklassen der Objektlogik, im Beispiel

- (Objektsprachen-)Variable : a, b, c, ...
- Quantoren: $\forall\_ \bullet \_$
- logische Konnektive: $\neg\_$, $\_ \Rightarrow \_$

, so werden, wie in Abschnitt 3.2.2.2 beschrieben, folgende BL-Symbole verwendet:

- (Metasprachen-)Konstantensymbole: a, b, c, ...
- (Metasprachen-)Variablen: X, Y, Z, ...
- (Metasprachen-)Funktionensymbole: $\forall\_ \bullet \_$, $\neg\_$, $\_ \Rightarrow \_$, $[\_ := \_]\_$
- (Metasprachen-)Prädikatensymbole $var\_$, $trm\_$, $frm\_$, $\_\backslash\_$, $\vdash \_ , \_ == \_$

In den folgenden drei Abschnitten werden Beispiele für die zur Axiomatisierung des Hilberttyp-Objektkalküls aufgestellten Sequenzen angegeben. Eine vollständige Auflistung der Sequenzen findet sich in [Gard88].
Die Abbildung der Objektkalkülregeln auf Metalogik-*Formeln* rechtfertigt die unten einzuführende *modell*theoretische Bezeichnung "*Theorien*" für die vom Benutzer eines (auf BL basierenden) offenen Deduktionssystems definierten "Regel"mengen, da diese im Rahmen der Metalogik als Sequenzen, d.h. als Formeln und nicht als syntaktische Schlußregeln aufgefaßt werden.

3.3.2.1 Axiomatisierung von Aspekten der Objektsprachensyntax und Semantik

Eine Teilmenge dieser Sequenzenklasse axiomatisiert die Metaprädikate *var*, *trm* und frm und erfaßt somit indirekt die Objektsprachensyntax. In [Gard88] erhält diese Klasse den Namen *WLF*, da sie die Menge der *wohlgeformten*[15] prädikatenlogischen Formeln und (in diesem Beipiel funktionslosen) Termen charakterisiert. Die entsprechenden Sequenzen sind in Abbildung 3.6 dargestellt. π steht hierbei für eine beliebige Zeichenkette über dem Objektsprachenalphabet, die als Objektlogikvariable aufgefaßt wird. Gibt es unendlich viele Variablen, so handelt es sich um eine (auf *Meta*ebene) *schematische* Axiomatisierung, da die Axiomatisierung in diesem Fall *unendlich* viele Sequenzen enthält. Dieses Problem kann hier auch nicht durch die Verwendung von Metavariablen umgangen werden. Beweisassistenten, die das Schließen in BL unterstützen, sollten auch solche Axiomatisierungen ermöglichen[16].
Darüberhinaus werden weitere Sequenzen formuliert, die *semantische* Eigenschaften wie Variablengebundenheit (d.h. das Symbol "\") und, darauf aufbauend, die Substitutionsfunktion axiomatisieren.

[15] engl. *well-formed*

[16] wie unten beschrieben wird, beispielsweise durch Vorgabe syntaktischer Defaults für bestimmte syntaktische Klassen

$$\frac{}{var\,\pi} \qquad \frac{var\,X}{trm\,X} \qquad \frac{\begin{array}{c}var\,X\\ frm\,P\end{array}}{frm\,\forall X \bullet P}$$

$$\frac{frm\,P}{frm\,\neg P} \qquad \frac{\begin{array}{c}frm\,P\\ frm\,Q\end{array}}{frm\,P \Rightarrow Q}$$

Abbildung 3.6: Axiomatisierung der Objektsprachen-Syntax

3.3.2.2 Axiomatisierung der Objektkalkülaxiome

Der betrachtete Hilberttyp-Prädikatenkalkül hat fünf (schematische) Axiome, die sich gemäß Abbildung 3.7 als Sequenzen darstellen lassen. Dabei ist zu beobachten, wie der schematische Cha-

$$\frac{frm\,P\ frm\,Q}{\vdash P \Rightarrow (Q \Rightarrow P)} \qquad \frac{frm\,P\ frm\,Q\ frm\,R}{\vdash (P \Rightarrow Q) \Rightarrow ((P \Rightarrow (Q \Rightarrow R)) \Rightarrow (P \Rightarrow R))}$$

$$\frac{frm\,P\ frm\,Q}{\vdash (\neg P \Rightarrow Q) \Rightarrow ((\neg P \Rightarrow \neg Q) \Rightarrow P)} \qquad \frac{var\,X\ trm\,T\ frm\,P}{\vdash (\forall X \bullet P) \Rightarrow [X := T]P}$$

$$\frac{\begin{array}{c}var\,X\ frm\,P\ frm\,Q\\ X \backslash P\end{array}}{\vdash (\forall X \bullet P \Rightarrow Q) \Rightarrow (P \Rightarrow \forall X \bullet Q)}$$

Abbildung 3.7: Axiomatisierung der Objektkalkül-Axiome

rakter der *Objekt*kalkülaxiome durch Verwendung von Meta*variablen* elegant abgebildet wird. Die den Objektkalkül*axiomen* entsprechenden Sequenzen zeichnen sich allgemein dadurch aus, daß das zum Formulieren der Beziehung der syntaktischen Ableitbarkeit verwendete Metaprädikatensymbol "$\vdash$" im Sequenz*konsequent*, jedoch nicht im Sequenzantezedent vorkommt. Der Sequenzantezedent ist stets von "formaler" Gestalt, d.h. er enthält nur Metaprädikate, die die Domänen der im Konsequent vorkommenden Metavariablen einschränken.

3.3.2.3 Axiomatisierung der Objektkalkülschlußregeln

Im betrachteten Objektkalkül gibt es zwei Schlußregeln: *Modus Ponens* und *Generalisierung* (vgl. Abbildung 3.8). Kalkül*schlußregeln* sind allgemein von der Gestalt "sind die Formeln P_1,

$$\frac{\begin{array}{c} frm\,P\ frm\,Q \\ \vdash P \\ \vdash P \Rightarrow Q \end{array}}{\vdash Q} \qquad \frac{\begin{array}{c} var\,X\ frm\,P \\ \vdash P \end{array}}{\vdash \forall X \bullet P}$$

Abbildung 3.8: Axiomatisierung der Objektkalkül-Schlußregeln

$\ldots$,P_n ableitbar, so ist auch Q ableitbar". Dementsprechend kommt das Metaprädikatensymbol "$\vdash$" nun sowohl im Antezedent als auch im Konsequent vor.

3.3.3 Erweiterungen

Die ausschnittsweise vorgestellte Axiomatisierung eines Hilberttyp-Prädikatenkalküls kann durch Angabe zusätzlicher Sequenzen erweitert werden:

3.3.3.1 Einführung weiterer Objektlogik-Konnektive

Einerseits enthält die oben axiomatisierte Prädikatenlogik keinen *Existenzquantor*. Um diesen einzuführen, sind, ähnlich wie oben, vier weitere Sequenzen zur Axiomatisierung der Metaprädikate (in diesem Fall frm, "\" und "==") zu formulieren (vgl. [Gard88]). In anderen Fällen lassen sich neue prädikatenlogische Konnektive als sog. *abgeleitete Symbole* unter Verwendung des Metaprädikats "==" anhand schon bekannter Konnektive definieren. Hier werden dann die entsprechenden Eigenschaften wie Wohlgeformtheit usw. von den bekannten, bereits axiomatisierten Symbolen wegen der Eigenschaften der Metagleichheit geerbt. Ein Beispiel ist die Einführung der logischen Konnektive "$\vee$", "$\wedge$" und "$\Leftrightarrow$" :

$$\frac{frm\,P\ frm\,Q}{P \vee Q == \neg P \Rightarrow Q} \qquad \frac{frm\,P\ frm\,Q}{P \wedge Q == \neg(\neg P \vee \neg Q)}$$

$$\frac{frm\,P\ frm\,Q}{P \Leftrightarrow Q == (P \Rightarrow Q) \wedge (Q \Rightarrow P)}$$

3.3.3.2 Eine schwächere Form der Gleichheit

Die in Abschnitt 3.3.1 diskutierte Voraxiomatisierung des Metaprädikats "==" kann für die hier axiomatisierte Objektsprache "Prädikatenlogik" als zu restriktiv angesehen werden: es macht Sinn, zwei prädikatenlogische Formeln (als Zeichenketten über dem Objektsprachenalphabet) auch dann als "==" anzusehen, wenn sie "bis auf Umbenennung gebundener Variablen" metagleich sind. Diese Form der Metagleichheit läßt sich, beispielsweise für den Allquantor, durch die Angabe des folgenden zusätzlichen Axioms[17] erhalten:

$$\frac{\begin{array}{c} var\,X\ var\,Y\ frm\,P \\ Y \backslash P \end{array}}{\forall X \bullet P == \forall Y \bullet [X := Y]P}$$

Diese erweiterte Form der Gleichheit wird in [Gard88] Alpha-Äquivalenz (engl. alpha-equivalence)[18] genannt und ist nach Definition *schwächer* als die bisher definierte Metagleichheit. Allerdings müssen andere Sequenzen, bei denen "==" im Antezedent steht, auf ihre Verträglichkeit mit dieser erweiterten Form der Gleichheit untersucht werden: Die bisher eingeführten Sequenzen erfüllen diese Bedingung, jedoch ist es möglich, Sequenzen zu formulieren, die nur unter der bisherigen Axiomatisierung "korrekt" sind. Sei beispielsweise "$\hookrightarrow$" ein Metaprädikat, welches, geeignet axiomatisiert, die Beziehung "erscheint gebunden in" formalisiert. Dann ist die Sequenz

$$\frac{\begin{array}{c} var\,X\ frm\,P\ frm\,Q \\ P == Q \\ X \hookrightarrow P \end{array}}{X \hookrightarrow Q}$$

mit der voraxiomatisierten Gleichheit verträglich, mit der erweiterten Gleichheit jedoch nicht. In Verbindung mit den zuvor eingeführten Axiomen erhält man hieraus eine flexiblere Version der Substitution. Bisher war es nur möglich, in Metatermen zu substituieren, in denen kein Konflikt mit dort gebundenen Variablen auftritt[19]. Das neu eingeführte Axiom gestattet die Vorab-Umbenennung gebundener Variablen und damit eine von den Namen der gebundenen Variablen unabhängige, "*sichere*" Substitution[20].

[17] engl. Bezeichnung *alpha-conversion*

[18] Die Wahl der Bezeichnung *Äquivalenz* ist wegen dem semantischen Hintergrund dieser Beziehung in der Objektlogik gerechtfertigt. In der Metasprache handelt es sich strenggenommen um *Gleichheit*.

[19] engl. *variable capture*

[20] engl. *safe substitution*

3.3.3.3 Vereinbarung von Abkürzungen

Um die Lesbarkeit von Beweisen und Formeln zu erhöhen, kann man, wiederum unter Ausnutzung der Axiome der Metagleichheit, *Namen* für Zeichenketten Ψ über dem Objektlogikalphabet, d.h. insbesondere für Objektlogikterme und Objektlogikformeln, vereinbaren:

$$\frac{}{NAME == \Psi}$$

Formal entspricht jeder vereinbarte Name einer Metakonstanten. Die Verbindung zur Objektsprache wird durch eine entsprechende Erweiterung der in Abschnitt 3.2.2.2 betrachteten semantischen Funktion $\| \_ \|$ hergestellt.

3.3.4 Axiomatisierung eines Anwendungsbereichs

In den vorangegangenen Abschnitten wurden eine Form der Prädikatenlogik sowie ein darauf definierter Kalkül nach Hilbert in der Sequenzenlogik axiomatisiert. Damit ist die Situation erreicht, wie sie in Deduktionssystemen üblicherweise *fest* vorgegeben ist. Aufbauend hierauf ergänzt der Benutzer *domänenspezifische Axiome* für den für ihn interessanten Diskursbereich. Im Gegensatz zu herkömmlichen Deduktionssystemen wird die Axiomatisierung nun nicht in einer festen Objektsprache, sondern ebenfalls in der Metasprache BL vorgenommen (vgl. o.). Aufbauend auf dem voraxiomatisierten Prädikatenkalkül wird im folgenden Abschnitt ausschnittsweise diskutiert, wie man ein einfaches arithmetisches System axiomatisiert, um darüber im Rahmen von BL schließen zu können.
Um arithmetische Aussagen in der Prädikatenlogik formulieren zu können, werden (auf Objektlogikebene) Symbole auf *Term*ebene der Metasprache sowie ein Gleichheitsprädikatensymbol eingeführt[21]:

- Konstantensymbol: 0
- Funktionensymbole: $succ\_$, $\_ + \_$, $\_ * \_$
- Prädikatensymbol: $\_ = \_$ (steht für *Objektlogik*gleichheit)

"*succ*" steht dabei für die *Nachfolgerfunktion* der Arithmetik, die einer natürlichen Zahl ihren eindeutigen Nachfolger zuordnet und dazu geeignet axiomatisiert wird (*Peano-Axiome*).
In der Metasprache werden diese Symbole wie folgt erfaßt:

- (Metasprachen-)Konstantensymbol: 0

[21] Man beachte, daß bei der Axiomatisierung des zugrundeliegenden Prädikatenkalküls weder Funktions- noch Konstantensymbole der Objektsprache eingeführt wurden, da diese im *Diskursbereich* der *Objekt*logik liegen und somit *anwendungsspezifisch* gewählt werden.

- (Metasprachen-)Funktionensymbole: $succ\_$, $\_ + \_$, $\_ * \_$, $\_ = \_$

D.h. Objektlogikfunktionensymbole und Objektlogikprädikatensymbole werden ebenfalls auf *Funktionen*symbole der Metasprache abgebildet, wobei wiederum die in Abschnitt 3.2.2.2 beschriebene semantische Beziehung besteht.
Die Axiomatisierung der neueingeführten Symbole geschieht wie oben, wobei nun der oben definierte Objektlogikkalkül zusätzliche *Axiome* erhält, mit denen einerseits die Beziehungen zwischen den Funktionen *succ*, "+" und "*" unter Verwendung der Objektlogikgleichheit "=" axiomatisiert werden, beispielsweise[22]

$$\frac{}{\vdash x * succ\, y \;=\; x * y + x}$$

und andererseits das zur vollständigen Charakterisierung der natürlichen Zahlen benötigte *Induktionsaxiom* formuliert wird:

$$\frac{var\, X\ frm\, A}{\vdash [X \;:=\; 0]A \Rightarrow ((\forall X \bullet A \;\Rightarrow\; [X \;:=\; succ(X)]A) \;\Rightarrow\; (\forall X \bullet A))}$$

Das Induktionsaxiom ist erst in Prädikatenlogik zweiter (oder höherer) Ordnung formulierbar, da es eine Aussage der Form "Für alle prädikatenlogische Formeln A gilt ... " macht[23]. Auf Metaebene hingegen werden prädikatenlogische Formeln als Zeichenketten über dem Objektsprachenalphabet dargestellt, d.h. als *Elemente* der zugrundeliegenden Domäne; somit ist das Induktionsaxiom direkt als Sequenz formulierbar.
Die Mächtigkeit von BL ist somit groß genug, um auch bestimmte Konzepte "höherer Ordnung" zu axiomatisieren, was eine Voraussetzung für viele realistische Anwendungen ist. Der bereits in der Praxis bewährte Ansatz von Boyer und Moore ([BoMo79]) basiert so u.a. auf einer "konstruktiven" Realisierung von Induktionsbeweisen, wodurch ein Weg gefunden wird, die in Logiken höherer Ordnung nicht mehr gewährleistete Vollständigkeit zu handhaben. Die weitere Diskussion wird jedoch zeigen, daß auch in BL bestimmte Einschränkungen für die axiomatisierbaren Kalküle bestehen. Eine genaue Untersuchung der "schematischen Mächtigkeit" von BL steht noch aus.

3.4 Das Schließen im Metakalkül

In der oben definierten Metasprache kann nun syntaktisches Schließen unter Anwendung der Schlußregeln des Sequenzenkalküls durchgeführt werden. Bei *Vorwärts*ableitungen beispielsweise wendet man die Regeln des Sequenzenkalküls, ausgehend von den für den aktuellen Objektkalkül

[22] die Formulierung dieser Axiome mit Hilfe von Objektvariablen entsprechenden Meta*konstanten* anstelle von Meta*variablen* und entsprechender Nebenbedingungen wird durch die Generalisierungs- und Spezialisierungsaxiome des Hilberttypkalküls gerechtfertigt. Entsprechende Aspekte werden in Abschnitt 3.7 diskutiert.

[23] d.h. , anders aufgefaßt, es wird über *Teilmengen* einer zugrundeliegenden Domäne quantifiziert

gegebenen Sequenzaxiomen Δ sowie den für den Anwendungsbereich gegebenen Axiomen Γ, wiederholt an, bis die gesuchte Zielsequenz S hergeleitet ist, d.h. in der üblichen Schreibweise

$$\Delta \cup \Gamma \vdash_{meta} S$$

Die syntaktischen Schlußregeln des Sequenzenkalküls arbeiten per definitionem auf Sequenzen. Dies führt teilweise zu recht umständlichen und unintuitiven Beweisen (vgl. Beispiel in [Gard88]). Wenn man beispielsweise eine der de Morganschen Regeln beweisen möchte, müßte man folgende *Sequenz* aus den obigen Axiomen ableiten:

$$\frac{frm\,P \;\; frm\,Q}{\neg(P \wedge Q) \;==\; (\neg P \vee \neg Q)}$$

3.4.1 Schließen auf Sequenzenlogik-Formelebene

Im folgenden wird ein Resultat hergeleitet, welches es ermöglicht, das Beweisgeschehen im Metakalkül auf die Ebene der *Formeln* der Sequenzenlogik (vgl. Syntaxdefinition in 3.2.2.1) zu verlagern und damit die *intuitive Sicht der gegebenen Sequenzen als Schlußregeln* eines benutzerdefinierten Kalküls (und nicht als *Theorie* einer Metalogik) zu rechtfertigen, wie sie beispielsweise von den meisten B-Tool-Benutzern eingenommen wird. Sei

$$S \;=\; \frac{A_1, \ldots, A_n}{P}$$

die unter den Axiomenmengen Γ und Δ zu beweisende Sequenz. Das Schließen auf (Meta)Formelebene geschieht gemäß folgenden *Verfahrens*[24]:

1. "*Annahme*" der Formeln im Antezedenten der zu beweisenden Sequenz:
   $M := \{A_1, \ldots, A_n\}$
2. Wiederhole: falls $S \in \Gamma \cup \Delta$ eine Sequenz ist, deren sämtliche Antezedenzformeln (eventuell nach Instantiierung) in M sind, so *kann*[25] deren Konsequent K zu M hinzugefügt werden: $M := M \cup \{K\}$. Wird die Konsequentformel P der abzuleitenden Sequenz erzeugt (d.h. $P \in M$), dann **stop**.

Gerechtfertigt wird dieses Vorgehen durch:

[24] Die Verwendung des Begriffs *Algorithmus* wird wegen des in Schritt 2 des Verfahrens bestehenden *Nichtdeterminismus* vermieden.

[25] Wahl durch Kontrollkomponente

Lemma 3.1 *Seien $A_1, \ldots, A_n$ die in Schritt 1 des Verfahrens getroffenen Annahmen. Dann gilt für $k \geq 0$: Ist nach der k-ten Durchführung von Schritt 2 $M = \{A_1, \ldots, A_n, T_1, \ldots, T_k\}$, so gilt für beliebiges $B \in M$: Die Sequenzen*

$$\frac{A_1, \ldots, A_n}{B} \quad , \; i = 1, \ldots, k$$

lassen sich im Rahmen des Sequenzenkalküls unter Verwendung der gegebenen Sequenzaxiome und Voraussetzungen ableiten.

Beweis: durch Induktion über k:
Induktionsanfang, $k = 0$: in diesem Fall ist $M = \{A_1, \ldots, A_n\}$. Durch Anwendung der Sequenzenkalkülregel ASSUME erhält man die Sequenzen

$$\frac{A_1}{A_1} \quad \ldots\ldots\ldots\ldots \quad \frac{A_n}{A_n}$$

Durch Erweiterung der Antezedenten durch die Schlußregel THIN erhält man schließlich die gesuchten Sequenzen

$$\frac{A_1, \ldots, A_n}{A_1} \quad \ldots\ldots \quad \frac{A_1, \ldots, A_n}{A_n}$$

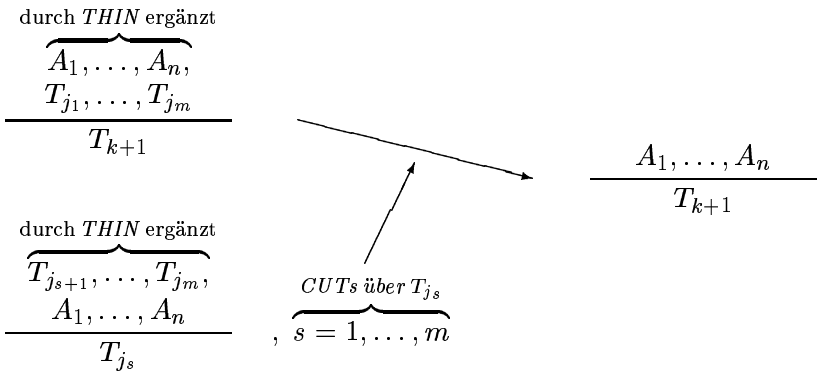
Induktionsschritt, $k \to k+1$: Sei $M = \{A_1, \ldots, A_n, T_1, \ldots, T_k\}$ die nach der k-ten Ausführung von Schritt 2 vorliegende Formelmenge. Für die Elemente von M gilt die Behauptung nach Induktionsvoraussetzung. Im Schritt k+1 wird ein Sequenzaxiom aus $\Gamma \cup \Delta$ gewählt, geeignet instantiiert und daraus eine Formel T_{k+1} hergeleitet. Dessen Antezedenten sind in M, d.h. die Sequenz ist von der Gestalt

$$\frac{\begin{array}{l} A_{i_1}, \ldots, A_{i_l}, \\ T_{j_1}, \ldots, T_{j_m} \end{array}}{T_{k+1}} \qquad \begin{array}{l} , \; i_s \in \{1, \ldots, n\}, \; 1 \leq s \leq l \\ , \; j_s \in \{1, \ldots, k\}, \; 1 \leq s \leq m \end{array}$$

und läßt sich im Sequenzenkalkül durch Anwendung der Schlußregel INSTANTIATE herleiten. Nach Induktionsvoraussetzung gilt nun für die T_{j_s} :

$$\frac{A_1, \ldots, A_n}{T_{j_s}} \quad , \; 1 \leq s \leq m$$

Durch Anwendung der Regel THIN auf die obigen Sequenzen erhält man schließlich $m+1$ Sequenzen, aus denen sich mittels m-fachem *CUT* (in der Reihenfolge $1, 2, \ldots, m$) die gesuchte Sequenz für T_{k+1} ergibt:



□

D.h. stoppt obiges Verfahren mit $P \in M$, so kann auch die gesuchte Sequenz

$$S \; = \; \frac{A_1, \ldots, A_n}{P}$$

auf der Ebene des Sequenzenkalküls abgeleitet werden. Für die oben abgebildete De-Morgan-Sequenz beispielsweise würde man die Antezedensformeln $frm\,P$ und $frm\,Q$ annehmen und versuchen, die Konsequenzformel

$$\neg(P \wedge Q) \; == \; (\neg P \vee \neg Q)$$

gemäß obigem Verfahren abzuleiten.
Der Beweis von Lemma 3.1 zeigt, daß das Verfahren des Schließens auf Formelebene als "*korrekt* relativ zu den im Metakalkül durchführbaren Deduktionen" bezeichnet werden kann. Die Umkehrung gilt ebenfalls: man kann nachweisen, daß alle Deduktionen des Sequenzenkalküls ein "Pendant" im Rahmen des obigen Verfahrens besitzen. Das entsprechende Lemma lautet:

Lemma 3.2 *Ist eine Sequenz*

$$\frac{A_1, \ldots, A_n}{B}$$

im Rahmen des Sequenzenkalküls unter Verwendung der gegebenen Sequenzaxiome ableitbar, so existiert ein $k \geq 0$, sodaß gilt: $B \in M_k$, wobei M_k die nach der k-ten Durchführung von Schritt 2 für eine spezielle Ableitung gemäß des Verfahrens vorliegende Menge M ist.

Beweis: Der Beweis geschieht per Induktion über die *Länge* n des im Metakalkül durchgeführten Beweises. Man kann o.B.d.A. davon ausgehen, daß die Sequenz erst im letzten Schritt des Beweises entstanden ist. Für $n = 0$ ist die vorliegende Sequenz ein *Axiom*, sodaß $B \in M_1$ ist, wenn

dieses Axiom beim ersten Durchlaufen von Schritt 2 "angewandt" wird. Im *Induktionsschritt* $n \to n+1$ wurde im letzten Metaableitungsschritt eine der vier Schlußregeln des Metakalküls angewandt. War die letzte angewandte Regel ein

- ASSUME: in diesem Fall ist der Sequenzkonsequent in der Menge M_0 einer *beliebigen* Ableitung gemäß des Verfahrens enthalten, d.h. die Behauptung gilt für $k = 0$.
- INSTANTIATE: auf die zuvor uninstantiierte Sequenz ist die Induktionsvoraussetzung anwendbar, d.h. es existiert ein k, sodaß deren Konsequent in einer durch Anwendung des Verfahrens erzeugten Menge B_k ist. Eine zusätzliche Instantiierung im letzten Schritt der zugehörigen Ableitung gemäß des Verfahrens ergibt die Behauptung für die instantiierte Sequenz.
- THIN: dieser Fall folgt direkt aus der Induktionsvoraussetzung für die Sequenz, deren Antezedent durch Anwendung der Regel THIN erweitert wurde. (Für das Verfahren bedeutet dies lediglich eine Erweiterung der Ausgangsmenge M_0.).
- CUT: zum Beweis dieses Falls wendet man zunächst die Induktionsvoraussetzung auf die "schneidende" Sequenz an und "verlängert" die entsprechende Ableitung mit Hilfe der nach Induktionsvoraussetzung für die "geschnittene" Sequenz existierenden Ableitung zu einer Ableitung der durch den CUT entstehenden Sequenz.

Hieraus folgt die Behauptung. □
Die beiden Beweise zeigen, daß der Metakalkül genau das "freie" Schließen im Objektkalkül modelliert, wie es ja durch das äquivalente *Verfahren* beschrieben wird.
Die Beweise sind *konstruktiv*: obwohl bestimmte Nichtdeterminismen bestehen, so lassen sich dennoch die Ableitungen gemäß einer der beiden Vorgehensweisen in eine Ableitung gemäß der anderen Vorgehensweise überführen, indem man sich an dem entsprechenden Induktionsbeweis orientiert.

3.4.2 Implikationen für Korrektheits- und Vollständigkeitsnachweise

Wegen der Gültigkeit dieser beiden Beziehungen erleichtert sich der Korrektheits- und Vollständigkeitsbeweis für den Metakalkül: es genügt, zu zeigen, daß das zuvor beschriebene Verfahren korrekt und vollständig bezüglich dem in Abschnitt 3.2.2.4 auf der Metasprache definierten semantischen Folgerungsoperator ist.
Diese Aussage läßt sich präzisieren. Das obige Verfahren entspricht einer *operationalen Formulierung eines Kalküls* für die eingeschränkte Prädikatenlogik über den Operatoren "$\wedge$" und "$\Rightarrow$", die in Abschnitt 3.2.2.4 dazu herangezogen wird, die Semantik der Metalogik BL zu definieren. Aus den Betrachtungen zuvor folgt somit: der Nachweis der Korrektheit und Vollständigkeit des *dem Verfahren zugrundeliegenden Kalküls* bezüglich dieser einfachen Sprache ist *äquivalent* zum Korrektheits- und Vollständigkeitsbeweis des Metakalküls für die Metasprache. Die Entscheidung, die Deduktionen im Rahmen des Metakalküls oder gemäß dem Verfahren durchzuführen, entspricht somit formal der *Wahl eines Kalküls*, nach dem geschlossen werden soll.

3.5 Auf BL basierende offene Deduktionssysteme

Es soll nun von einem allgemeinen Standpunkt aus diskutiert werden, wie sich das oben beschriebene Konzept der Sequenzenlogik BL und des darauf definierten Sequenzenkalküls als formale Grundlage zur Realisierung eines offenen Deduktionssystems einsetzen läßt.
Ein Deduktionssystem, welches das Schließen in der Metalogik BL unterstützt, muß zwei Grundaufgaben lösen:

1. *Verwalten* der Sequenzaxiommengen und der abgeleiteten Sequenzen.
2. *Durchführen von Ableitungen* im Sequenzenkalkül. Die Beweise geschehen, wie im letzten Abschnitt gerechtfertigt, aus der Sicht des Benutzers auf *Formel*ebene[26]. Der Benutzer sieht die Sequenzen damit als *Schlußregeln*[27] und kann automatische Deduktionen initiieren, oder läßt sich *Vorschläge* in Form von auf das aktuelle Teilziel anwendbaren Regeln machen (Funktion als Beweis*assistent*).

Anhand der in den vorangegangenen Abschnitten vorgestellten Axiomatisierung eines Hilberttyp-Prädikatenkalküls ist zu ersehen, daß viele der entstehenden Teilziele nur "formaler" Natur sind, d.h. formale Eigenschaften wie Syntax und Semantik der Objektlogik erfassen (z.B. frm, trm, var, "\" etc.). Der Anwender des Beweisassistenten hingegen möchte Beweise im Rahmen seines axiomatisierten Objekt*kalküls* führen, nicht jedoch Metadeduktionen sehen, in denen formale Nebenbedingungen bewiesen werden. Der Beweisstil auf Formelebene unterstützt diese Sicht. Um den Anwender möglichst weitgehend von der Metaebene zu isolieren, ist folgendes Vorgehen denkbar:

- Der Benutzer definiert Stelligkeit, Assoziativität und Priorität sowie die Symbolklasse für die Symbole der betrachteten Objektlogik vor, wobei vom Beweisassistenten syntaktische Defaultannahmen gemacht werden können, die abgeändert werden dürfen. Anhand dieser Vorgaben kontrolliert das System dann automatisch die Syntax der manipulierten Objektlogikformeln durch einen effizienten Parse-Algorithmus.
- Es werden wichtige, nicht unter Verwendung von Metavariablen formulierbare, jedoch bedeutsame syntaktische Klassen wie *var* vordefiniert (vgl. o.).
- Der Benutzer axiomatisiert den Objektlogikkalkül durch die Angabe von Sequenzaxiomen. Die Angabe *formaler* Sequenzantezedenten kann nun wegen der systemseitig getroffenen Defaultannahmen teilweise entfallen, beispielsweise für syntaktische Klassen wie objektlogische Variablen.
- Die Axiome der allgemein als nützlich angesehenen Metaprädikate "==" (vgl. oben) und "\" sowie für die Substitution "[_, _]_" sind (in operationaler Form) "eingebaut".
- Aus der Sicht des Benutzers finden Beweise nun weitgehend im Rahmen des Objektkalküls statt. Das Metaprädikatensymbol "$\vdash$" ist unsichtbar und steht *implizit* vor jeder Formel.

[26] Im Sinne der syntaktischen Definition der Sequenzenlogik
[27] d.h. *beweistheoretisch*

Dieses Vorgehen ermöglicht nicht nur eine vereinfachte konzeptuelle Sicht der stattfindenden Deduktionen im Rahmen eines benutzerdefinierten Kalküls, sondern ist zugleich auch *effizienter*: die bisher ebenfalls durch Ableitungen im Rahmen des Kalküls formal überprüften syntaktischen Eigenschaften von Objektlogik-Zeichenketten sowie die Behandlung von "\" und "[_, _]_" geschieht nun *algorithmisch*, was die Anzahl der zu betrachtenden Sequenzen sowie die Länge der Beweise verringert.
Die *Deduktionen* werden gemäß dem in Abschnitt 3.4.1 motivierten Verfahren durchgeführt. Aus pragmatischen Gründen bietet es sich jedoch an, *zielgerichtet*[28] vorzugehen, d.h. obiges Verfahren *rückwärts* durchzuführen. Im *automatischen* Deduktionsmodus (vgl. Kap. 4) des B-Tools wird z.B. folgendes Vorgehen gewählt[29] : man geht von der *Konsequent*formel der zu beweisenden Sequenz aus und legt diese auf einem *Zielkeller* ab. Im allgemeinen Schritt wird nun die jeweils oberste Formel P des Zielkellers "reduziert", indem man sie durch die Menge der Antezendensformeln einer gegebenen, geeignet instantiierten Sequenz ersetzt, deren Konsequentformel die betrachtete Formel P "matcht"[30]. Es wird demnach *Tiefensuche ohne Backtracking* durchgeführt[31]. Teilziele P, die im Antezedenten der abzuleitenden Sequenz stehen, werden weggelassen. Dieses Verfahren hat den Vorteil, zielgerichtet zu sein, d.h. es werden nur Sequenzen "angewendet", deren *Konsequent* auch wirklich zur Ableitung der gesuchten *Ziel*formel beitragen kann. Da jedoch kein Backtracking angewendet wird, ist es möglich, daß eine für ein vorliegendes Teilziel existierende Ableitung wegen der falschen Wahl einer anwendbaren Sequenz nicht gefunden wird. Dies unterscheidet den B-Tool-Abarbeitungsmechanismus von der in PROLOG realisierten Tiefensuche *mit* Backtracking. Das B-Tool bietet stattdessen *benutzerdefinierbare* Kontrollstrategien, die allerdings erst im Rahmen des folgenden Kapitels besprochen werden sollen. Die Beschreibung der Realisierung dieser Konzepte auf dem B-Tool wird die Ausführungen dieses Abschnitts verdeutlichen bzw. präzisieren.

3.6 Die Prälogik des B-Tools

Aufbauend auf den allgemeinen Konzepten der Metalogik BL läßt sich nun die Prälogik des offenen Deduktionssystems B-Tool beschreiben. Unter *Prälogik* sollen an dieser Stelle *zusätzliche* Vordefinitionen verstanden werden, die für die Logik des B-Tools bestehen: im Vergleich zu "vollkommen offenen" Deduktionssystemen, bei denen keine Vorabentscheidungen im Hinblick auf die zu realisierende Objektlogik getroffen werden, ist im B-Tool eine Teilmenge eines prädikatenlogischen Kalküls vordefiniert, sodaß eine Konkretisierung der in Abbildung 2.2 grob skizzierten Architektur des B-Tools notwendig ist. Es soll nun der prädikatenlogische Kalkül vorgestellt werden, gemäß dem das B-Tool teilweise voraxiomatisiert ist.
Der oben vorgestellte prädikatenlogische Kalkül nach Hilbert ist für praktische Anwendungen eine schlechte Wahl. Stattdessen bietet sich die Axiomatisierung eines speziellen *Kalküls des natürlichen Schließens*[32] an, dessen Axiome und Schlußregeln ein wesentlich intuitiveres Be-

[28] vgl. o. , im Sinne von "ausgehend vom Ziel"
[29] Diese Darstellung wird unten konkretisiert.
[30] d.h. *durch Unifikation angleichbar* ist
[31] Dies gilt nur für *automatische* Deduktionen. In Deduktionen unter (erweiterter) Benutzerkontrolle kann hingegen Backtracking initiiert werden.
[32] In [Gard88] wird keine Quelle für den "natural deduction calculus" angegeben. Gewöhnlich bezeichnet man so die von Gentzen eingeführte Kalküle, in denen *lokale Voraussetzungen* eingeführt und beseitigt werden können. Vgl. [Gent34] und [WoLo88].

weisverständnis prägen und gut mit dem zielgerichteten Beweisstil eines Beweisassistenten zusammenspielen.
Grundgedanke der hier axiomatisierten Form des natürlichen Schließens ist die *explizite* Verwaltung von *Objekt*logik-Abhängigkeiten der Form "eine Formel G ist unter den *Annahmen* ("Hypothesen") $H_1, \ldots, H_n$ ableitbar". Dafür wird anstelle des oben eingeführten Metaprädikats "$\vdash$" eine *Familie* von Metaprädikaten $\{\vdash_i | \; i \geq 0\}$ definiert mit

$$H_1, \ldots, H_i \vdash_i G \quad \overset{\text{Meta}}{\Longleftrightarrow} \quad \vdash H_1 \Rightarrow (H_2 \Rightarrow \cdots (H_i \Rightarrow G) \cdots) \tag{3.1}$$

Diese Definition erhält ihre Motivation durch die oben diskutierte *syntaktische Variante des Deduktionstheorems*: ist ein in der Sequenzenlogik axiomatisierter Objektkalkül (für die Prädikatenlogik) korrekt und vollständig, so gilt die hier definierte Beziehung für geschlossene prädikatenlogische Formeln (vgl. insbes. [GeNi87]). Semantisch entspricht diese Verallgemeinerung des Metaprädikats "$\vdash$" einer Einführung von Sequenzen auf der Objektlogikebene, wobei allerdings weiterhin metalogische Formeln formaler Gestalt existieren, die keine Sequenzenform haben. Zugunsten der oben definierten einheitlichen Sequenzensyntax ist es zweckmäßig, die Symbole "$\vdash_i$" weiterhin als Meta*prädikate* und nicht als objektlogischen Sequenzoperator (d.h. als metalogische *Funktion*) aufzufassen. Die (schematischen) Axiome und Schlußregeln des Kalküls des natürlichen Schließens sind in Abbildung 3.9 zusammengefaßt.

$$\frac{frm\,H_1, \ldots, frm\,H_n}{H_1, \ldots, H_n \vdash H_i}\ \text{HYP}$$

$$\frac{\begin{array}{c} frm\,P\ frm\,Q \\ frm\,H_1, \ldots, frm\,H_n \\ H_1, \ldots, H_n, P \vdash Q \end{array}}{H_1, \ldots, H_n \vdash (P \Rightarrow Q)}\ \text{DED}$$

$$\frac{\begin{array}{c} var\,X\ frm\,P \\ frm\,H_1, \ldots, frm\,H_n \\ X \backslash H_1, \ldots, X \backslash H_n \\ H_1, \ldots, H_n \vdash P \end{array}}{H_1, \ldots, H_n \vdash \forall X \bullet P}\ \text{GEN}$$

$$\frac{frm\,P\ frm\,Q}{\neg P \Rightarrow Q, \neg P \Rightarrow \neg Q \vdash P}\ \text{CONTRA}$$

$$\frac{\begin{array}{c} frm\,G_1, \ldots, frm\,G_n \\ frm\,H_1, \ldots, frm\,H_m\ frm\,P \\ H_1, \ldots, H_m \vdash P \\ G_1, \ldots, G_n \vdash H_1 \\ \vdots \\ G_1, \ldots, G_n \vdash H_m \end{array}}{G_1, \ldots, G_n \vdash P}\ \text{TRANS}$$

$$\frac{var\,X\ trm\,T\ frm\,P}{\forall X \bullet P \vdash [X := T]P}\ \text{SPEC}$$

$$\frac{frm\,P\ frm\,Q}{P, (P \Rightarrow Q) \vdash Q}\ \text{MP}$$

Abbildung 3.9: Axiomatisierung des natürlichen Schließens

Der Kalkül des natürlichen Schließens ermöglicht im Vergleich zum oben diskutierten Hilberttyp-Kalkül leichter lesbare und i.a. kürzere Beweise, jedoch ist er, auch unter Verwendung metalogischer Variablen, nicht direkt in der Sequenzenlogik BL axiomatisierbar. Bei der Realisierung des B-Tools wurde die allgemeine Vorteilhaftigkeit des natürlichen Schließens erkannt; die nicht unter Verwendung von Metavariablen formulierbaren "problematischen" Axiome HYP, GEN, DED sowie (in abgewandelter Form, vgl.u.) TRANS sind daher *vordefiniert*. SPEC, MP und CONTRA sind (unter Verwendung von Metavariablen) in der Sequenzenlogik axiomatisierbar und werden nicht vordefiniert. Sie müssen explizit vom Benutzer angegeben werden.
Der Nachteil dieses Vorgehens liegt darin, daß die Regeln HYP, GEN, DED und TRANS, unabhängig vom Anwendungsgebiet des Beweisassistenten, *vorgegeben* sind. Damit geht die oben beschriebene Flexibilität bei der Wahl von Objektlogik und Objektkalkül teilweise verloren. Im Rahmen der Beschreibung der genauen operationalen Realisierung der Regeln im B-Tool wird jedoch deutlich werden, daß auch eine *alternative Sicht* des B-Tools als Deduktionssystem über "purem" BL, d.h. ohne die Vordefinitionen des Kalküls des natürlichen Schließens, möglich ist. Die genaue Umsetzung der metalogischen Konzepte sowie insbesondere die Rolle der Regel TRANS im B-Tool soll im folgenden Kapitel diskutiert werden.
Durch die Voraxiomatisierung obiger Schlußregeln ist das B-Tool besonders für Anwendungen geeignet, bei denen man im Rahmen der *Prädikatenlogik* arbeiten will. Ein Teil der Flexibilität geht durch die Vorvereinbarungen zwar verloren, jedoch besteht weiterhin die Freiheit, neue Operatorsymbole in Abhängigkeit der intendierten Anwendung einzuführen ([Gard90]).
Die Architektur des offenen Deduktionssystems B-Tool läßt sich somit in der in Abbildung 3.10 dargestellten, gegenüber Abbildung 2.2 präzisierten Form skizzieren.

Am Beispiel des Kalküls des natürlichen Schließens wird ersichtlich, daß Einschränkungen bei

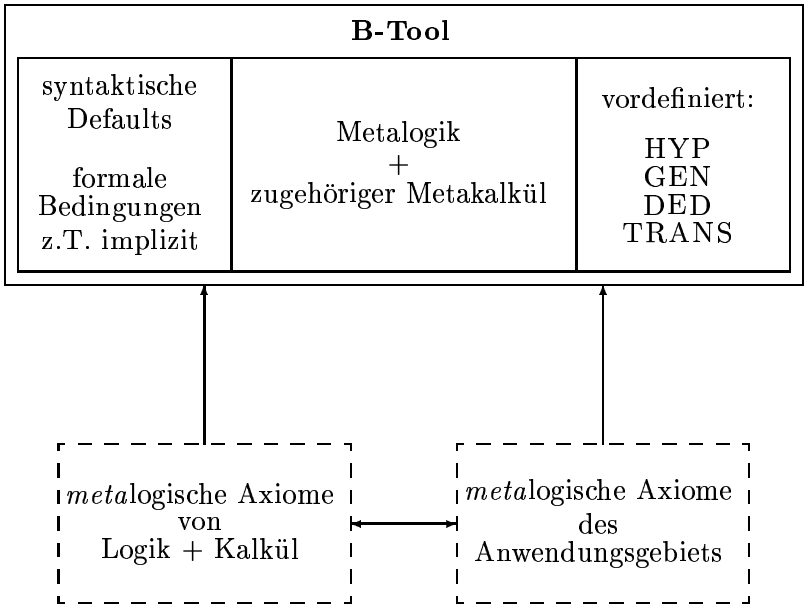


Abbildung 3.10: verfeinerte Architektur des B-Tools

der Formulierbarkeit schematischer Kalküle in der Sequenzenlogik bestehen. Die Einführung des Schließens unter Voraussetzung über die Metaprädikatenfamilie $\{\vdash_i | \ i \geq 0\}$ ist auch unter Verwendung von Metavariablen nicht durch endlich viele metalogische Axiome formulierbar[33].

3.7 Variablen-"Liften"

In den vorangegangenen Abschnitten wurde die unterschiedliche Rolle objektlogischer und metalogischer Variablen deutlich: objektlogische Variablen werden auf *Konstanten*symbole der Metalogik abgebildet, wohingegen Metavariable gemäß ihrem Diskursbereich für beliebige Zeichenketten über dem Objektlogikalphabet stehen[34].
Im Fall der Axiomatisierung prädikatenlogischer Kalküle, in denen Generalisierungs- und *Spezialisierungsregel* vorgegeben sind, besteht hier jedoch oftmals ein direkter Zusammenhang, wie das folgende Beispiel verdeutlicht:

[33][Gard88] geht nicht darauf ein, wieso man nicht eine syntaktische Klasse entsprechend *Listen* von (Hypothesen)formeln einführen und die Hypotheseninhalte durch *eine* Metavariable schematisch erfassen kann.

[34]In dem mehr an den Implementationsgesichtspunkten des B-Tools orientierten [Abri86] werden die Metavariablen *Joker* genannt, und eine Instantiierung als *syntaktische Substitution* bezeichnet.

$$\frac{}{\vdash n + 0 = n} \quad \xrightarrow{\begin{array}{c} GEN, \\ SPEC, \\ \text{Substitution} \end{array}} \quad \frac{}{\vdash m + 0 = m}$$

Die prädikatenlogischen Kalkülregeln GEN, SPEC und die Axiomatisierung der Substitution erlauben in diesem Fall eine Umbenennung der Metakonstanten n. Sequenzen dieser Gestalt wurden oben bereits zur Axiomatisierung der Arithmetik formuliert; da keine Metavariablen enthalten sind, spricht man von *nichtschematischen Sequenzen*.
Nichtschematische Sequenzen sind i.a. schwieriger zu verwenden. In obigem Beispiel liegt es nahe, zu einer äquivalenten *schematischen* Sequenz überzugehen:

$$\frac{trm\ N}{\vdash N + 0 = N}$$

Hieraus erhält man dann die gesuchte Instanz durch einmaliges Anwenden der Instantiierungsregel des Metakalküls, wobei die Überprüfung der Nebenbedingung $trm\ N$ automatisch durch den Beweisassistenten geschieht. Dieser Übergang zur schematischen Form wird *variable lifting* genannt.
Für die nichtschematische Sequenz

$$\frac{}{\vdash a \leq b \ \Leftrightarrow\ \exists x \bullet a + x = b}$$

hingegen führt der intuitive Übergang zur schematischen Version

$$\frac{var\ X\ trm\ A\ trm\ B}{\vdash A \leq B \ \Leftrightarrow\ \exists X \bullet A + X = B}$$

zu Widersprüchen[35], da es möglich ist, für A und B Terme zu substituieren, die die für X substituierte gebundene Variable enthalten ("*variable capture*"). Die korrekte "geliftete" Version enthält *zusätzliche Restriktionen*, die diesen Fall ausschließen:

$$\frac{\begin{array}{c} var\ X\ trm\ A\ trm\ B \\ X \backslash A\ X \backslash B \end{array}}{\vdash A \leq B \ \Leftrightarrow\ \exists X \bullet A + X = B}$$

In [ViGa88b] wird untersucht, welche nichtschematischen Sequenzen überhaupt in eine schematische Form überführbar sind. Zu diesem Zweck werden Beispielsequenzen unterschiedlicher Gestalt betrachtet und die beim "Liften" entstehenden Schwierigkeiten untersucht. Auf diese Weise wird schließlich eine Metaschlußregel *VAR-lift* entwickelt, die bestimmte nichtschemati-

[35] vgl. Beispiel in [Gard88]

sche Sequenzen in eine *äquivalente* schematische Form überführt. In einem Äquivalenzbeweis wird gezeigt, daß die geliftete Sequenz aus der ursprünglichen Form herleitbar ist.
Im B-Tool ist die Regel *VAR-lift eingebaut*, wobei die Defaultannahmen für die Syntax objektlogischer Variablen berücksichtigt werden. Der Benutzer kann Sequenzen nichtschematisch formulieren. Sofern möglich, interpretiert der Beweisassistent die objektlogischen Variablen als Variablen der Metalogik[36] und überwacht *intern* die Gültigkeit der zusätzlichen Nebenbedingungen der schematischen Version. Dadurch entstehen folgende Vorteile:

1. Die schematische Sequenz ist i.a. einfacher anzuwenden, wie das erste Beispiel verdeutlicht.
2. Dem Benutzer wird das fehleranfällige Formulieren der schematischen Sequenz (d.h. der zusätzlich benötigten Nebenbedingungen) abgenommen. Das System garantiert für die Korrektheit der gelifteten Version.

Demgegenüber entsteht der Nachteil, daß *VAR-lift* auch dann angewandt wird, wenn der Benutzer *nicht* im Kalkül des natürlichen Schließens arbeiten will und eine andere Objektsprache axiomatisiert, in der es keine Rechtfertigung für eine solche Regel gibt[37].

3.8 Theoretische Beziehungen zur logischen Programmierung

Die hier beschriebene, für offene Deduktionssysteme geeignete Metalogik ermöglicht (nach Abschnitt 3.4) zwei unterschiedliche Blickwinkel: die *modelltheoretische* und *beweistheoretische* Sicht. Es soll diskutiert werden, welcher Zusamenhang zu den entsprechenden Sichtweisen der *logischen Programmierung* besteht.
Die modelltheoretische Sicht betrachtet die formulierten Axiome als Sequenzenformeln der Metalogik BL, durch die Objektlogik, objektlogischer Kalkül und ein Anwendungsbereich festgelegt werden. Die Menge der Sequenzen, die sich durch (syntaktisches) Folgern im Sequenzenkalkül aus diesen Axiomen herleiten lassen, ist unter syntaktischem Folgern abgeschlossen und wird die axiomatisierte *Theorie* genannt, eine Bezeichnung, die insbesondere auch für die vom B-Tool verwalteten "Regelsammlungen" verwendet wird. Geht man davon aus, daß der Sequenzenkalkül korrekt und vollständig bezüglich der oben definierten semantischen Folgerungsbeziehung "$\models_{meta}$" ist, so entspricht dies der üblichen Definition des Theoriebegriffs[38] ; man erhält eine allgemein als *modelltheoretische Sicht* bezeichnete Beschreibung der Semantik der definierten "Regeln":

Definition 3.3 (modelltheoretische Semantik) *Sei A eine gegebene Axiomenmenge der Sequenzenlogik. Dann ist die modelltheoretische Semantik definiert als die Menge $M_{\models}$ der aus A metalogisch folgerbaren Sequenzen:*

$$M_{\models} = \{G \mid A \models_{meta} G\}$$

[36] Dies ist die Motivation für die Bezeichnung *variable lifting*.
[37] Eine formale Ungenauigkeit besteht selbst in dem Fall, in dem man im Rahmen der *vordefinierten* Schlußregeln des Kalküls des natürlichen Schließens arbeiten möchte: die Regel SPEC ist, gemäß den Ausführungen in [Gard88], *nicht* voraxiomatisiert.
[38] vgl. z.B. in [WoLo88]

Gemäß dem in Abschnitt 3.4.1 beschriebenen Verfahren des Schließens auf Formelebene kann andererseits auch eine *beweistheoretische Sicht* angenommen werden, die die vordefinierten Sequenzen entsprechend dem "beweistheoretischen Inhalt" der logischen Implikation als *Schlußregeln* auffaßt.

Definition 3.4 (beweistheoretische Semantik) *Sei S die Menge der vorgegebenen Sequenzenaxiome und "$\vdash_S$" der durch das in Abschnitt 3.4.1 angegebene Verfahren definierte Ableitungsoperator. Dann ist die beweistheoretische Semantik definiert als die Menge $M_\vdash$ der nach diesem Verfahren aus der gegebenen Formel ableitbaren* objektlogischen *Formeln*[39].

Die Sichtweisen $M_\models$ und $M_\vdash$ stimmen nach Lemma 3.1 und 3.2 überein.
Die vergleichende Diskussion modelltheoretischer und beweistheoretischer Semantikdefinitionen findet auch im Bereich der logischen Programmierung statt. In [HaSH88] beispielsweise wird die Flexibilität der beweistheoretischen Sicht von Hornklauselprogrammen besprochen: Sei

$$A_1 \wedge \ldots \wedge A_n \rightarrow B$$

eine Programmklausel eines logischen Programms[40] P. Dann unterscheidet man

- *modelltheoretische Sicht*: obenstehende Programmklausel wird als prädikatenlogische *Formel* gedeutet, wobei das Konnektiv "$\rightarrow$" als logische Implikation "$\Rightarrow$" aufgefaßt wird und die (freien) Variablen als allquantifiziert angesehen werden:

  $$\forall(A_1 \wedge \ldots \wedge A_n \Rightarrow B)$$

  Die Semantik eines logischen Programms P ergibt sich dann als die Menge aller geeignet instantiiertern[41] Ziele G, die sich aus der Menge der Programmklauseln P folgern lassen, d.h. $\{\forall(G\theta) \mid P \models \forall(G\theta)\}$. In [Lloy84] wird gezeigt, daß man sich bei Allgemeingültigkeitsbetrachtungen in diesem Fall auf die bereits oben angesprochenen *Herbrand*interpretationen von P beschränken kann. Da die Menge aller Herbrand*modelle* eines Logikprogramms P (aufgefaßt als Mengen atomarer Formeln) unter Durchschnittsbildung abgeschlossen sind, kommt man hier schließlich zu dem gittertheoretischen Begriff des *kleinsten* Herbrandmodells als die modelltheoretische Semantik von P.

- *beweistheoretische Sicht*: in der in [HaSH88] eingenommenen Sichtweise werden die Programmklauseln als (zu instantiierende) *Schlußregeln* eines einfachen Inferenzsystems aufgefaßt, d.h. obige Klausel wird (für $n \geq 1$) als *Inferenzregel*

  $$A_1 \wedge \ldots \wedge A_n \vdash B$$

[39] Genaugenommen werden Formeln der Gestalt "$\vdash F$" abgeleitet, wobei F die für den Benutzer sichtbare objektlogische Formel ist.

[40] Hierbei liegt keine bestimmte logische Programmiersprache wie beispielsweise PROLOG zugrunde. Die Autoren gehen vielmehr lediglich von dem allgemein akzeptierten Hornklausel-Paradigma ("definite Klauseln") aus und diskutieren logische Programme von einem abstrakt-formalen Gesichtspunkt aus.

[41] für die Definition des entsprechenden Begriffs der korrekten Antwortsubstitution vgl. z.B. [Beck88] und insbes. [Lloy84].

bzw. (für $n = 0$) als *Axiom* " $\vdash B$ " eines Logikkalküls angesehen. Dies entspricht genau der in [Lloy84] beschriebenen operationalen Deutung eines Logikprogramms P als monotone, stetige[42] Abbildung auf dem Gitter aller Herbrandinterpretationen des Programms, wobei deren iterierte Anwendung gegen einen Fixpunkt konvergiert.

Die Beziehung zwischen beiden Sichtweisen wird in [Lloy84] durch ein Theorem hergestellt, welches aussagt, daß der kleinste Fixpunkt des Programms gleich dem kleinsten Herbrandmodell ist.
Die verdeutlicht die *strukturelle Nähe der BL-Sequenzen zu Klauseln von logischen Programmen*. Durch Verwendung des Metaprädikats "$\vdash$" werden die im Rahmen der Objektlogik als strukturiert angesehenen Formeln in BL als *atomare* Formeln abgebildet, so daß der Vergleich zu PROLOG-Klausel*literalen* möglich wird. Die *Horn*klauselform wird durch die Einschränkung der Metasprache auf Sequenzen mit einem *einzelnen* Konsequenten impliziert.
Deduktionen werden, gemäß des oben beschriebenen Verfahrens, auf Formelebene durchgeführt. In der Betrachtung zuvor steht der dem Verfahren entsprechenden beweistheoretischen Sicht die beweistheoretische PROLOG-Sichtweise gegenüber, die die operationale Deutung eines Logikprogramms beinhaltet. Der Abarbeitungsmechanismus des B-Tools erinnert daher auch an PROLOG.

[42] im Sinne der Gittertheorie "diskret-stetig"

Kapitel 4

Beschreibung eines auf BL basierenden Beweisassistenten

Nachdem eine zur Beschreibung offener Deduktionssysteme geeignete Metalogik vorgestellt wurde, soll nun die konkrete *Realisieriung* des offenen Deduktionssystems *B-Tool* diskutiert werden, dessen Logik auf BL aufbaut.
In den folgenden Abschnitten werden zunächst, orientiert an den Grundaufgaben eines Deduktionssystems, die Verwaltung von Regelsammlungen sowie das Durchführen von Beweisen auf dem B-Tool beschrieben und die Beziehung der verwendeten Notation zur metalogischen Sicht von Kapitel 3 erarbeitet. Der anschließende Abschnitt beschreibt die Steuerungsmechanismen des B-Tools und damit dessen vierte Ebene gemäß der Schichtenarchitektur für Deduktionssysteme. Darüberhinaus sollen weitere Aspekte, u.a. der Aufbau des *Softwaresystems* B-Tool, diskutiert werden.
In der folgenden Beschreibung werden, abweichend von einer formalen, metalogischen Sicht, die vom Benutzer formulierten Sequenzaxiome "B-Tool-üblich" als *Regeln* bzw. *Fakten* bezeichnet.

4.1 Regelsammlungen

Die Verwaltung der vom Benutzer definierten Regeln und Fakten geschieht in sog. *Theorien*. Abbildung 4.1 zeigt als Beispiel für die verwendete Syntax eine Theorie `set`, in der elementare Beziehungen der Mengentheorie als Regeln formuliert sind. So besagt beispielsweise die erste Regel, daß sich der Beweis einer Mengeninklusionsbeziehung auf den Beweis einer Elementbeziehung (":" entspricht "$\in$") zur Potenzmenge derselben Menge zurückführen läßt[1]. Einzelne *Großbuchstaben* stellen hierbei sog. *syntaktische Variable* dar, die im Rahmen eines Beweises durch beliebige *syntaktisch korrekte Zeichenketten*[2] instantiiert werden können. Das Zeichen "&" im Regelantezedenten separiert hierbei die einzelnen, durch Anwendung der Regel von rechts nach links entstehenden Teilziele.

[1] In diesem Fall könnte man die Regel auch genau andersherum formulieren. Das Beispiel erhebt keinen Anspruch auf "Vollständigkeit".

[2] Zeichenketten, die die vordefinierten Operatorstelligkeiten respektieren, vgl. Abschnitt 4.3.1.

```
THEORY set IS

        (S : POW(T))  =>  (S inc T)
      ;
        (X : S) & (T : POW(S))  =>  (T \/ {X} :  POW(S))
      ;
        (S inc T)  =>  (S inc (T \/ U))

END
```

Abbildung 4.1: Theorie `set`

Auf den Regeln und Fakten einer Theorie ist eine *lineare Ordnung* definiert, die bei der Durchführung von Deduktionen bedeutsam ist und damit als ein *Konzept der Steuerung* anzusehen ist. Eine Theorie stellt somit formal eine *Liste* und keine Menge dar.

4.2 Die Beweismechanismen des B-Tools

Auf dem B-Tool werden *zwei Deduktionsmodi* angeboten, *Rückwärtsdeduktion* und *Vorwärtsdeduktion*, die nun anhand von Beispielen erläutert werden sollen. Mit Hilfe der Regeln der in Abbildung 4.1 gezeigten Theorie `set` lassen sich einige elementare Ergebnisse der Mengentheorie ableiten.
Als Grundlage für Beispielbeweise werden nun zunächst zwei weitere Theorien eingeführt. Abbildung 4.2 enthält die Theorie `prover`, mit deren Regel ein Beweis initiiert und dessen erfolgreiche Durchführung ggf. angezeigt wird. Die Abbildung 4.3 zeigt die zu beweisenden Aussagen, die in

```
THEORY prover IS

        P & bwrite("\n\nProposition is Valid\n\n")  =>  prove(P)

END
```

Abbildung 4.2: Theorie `prover`

diesem Fall ebenfalls in Form einer Theorie verwaltet werden[3]. Die *Kleinbuchstaben* stehen hierbei, im Unterschied zu den syntaktischen Variablen, für *Variable des Anwendungsbereichs*, die in diesem Kontext am besten als Diskursobjekte der Mengentheorie, d.h. als Mengen aufgefaßt

[3]Das B-Tool unterstützt auch die Eingabe *einzelner*, zu beweisender Regeln ([Stuc90]).

```
THEORY exercise IS

       prove((a : POW(b))   =>  (a inc (b \/ c)))
     ;
       prove((a : POW(b)) &  (e : b)   =>  (a \/ {e} inc b))
     ;
       prove((d inc b) &  (a : POW(d)) & (e : d) => (a \/ {e} inc (b \/ c)))

TAC
        prover;(set~;DED)~

FTAC
        fwdset~

END
```

Abbildung 4.3: Zu beweisende Aussagen

werden[4].
In der zusätzlichen Theorieklauseln `TAC` und `FTAC` der Theorie `exercise` werden *die zum Beweis der Aussagen heranzuziehenden Taktiken* formuliert, deren genaue Funktion als Steuerungsmechanismus erst in einem nachfolgenden Abschnitten erklärt wird.

4.2.1 Rückwärtsdeduktionen: zielgerichtetes Schließen

Der Beweis der ersten Aussage der Theorie `examples` wird in Abbildung 4.4 gezeigt; er ist von unten nach oben zu lesen. Dies begründet sich durch den im B-Tool realisierten *zielgerichteten Beweisstil* als Standardbeweismodus. Im ersten Schritt wird Regel 1 der Theorie `prover` rückwärts angewandt. Danach enthält der verwaltete *Zielkeller* die Formel der vierten Beweiszeile (vgl. rechte Spalte des Beweises) sowie das `bwrite`-Teilziel. Letzteres hat operationale Semantik und wird im Beweis nicht angezeigt, da es stets "beweisbar" (ausführbar) ist[5]. In den Beweiszeilen wird ebenfalls deutlich, welche Formeln jeweils für die syntaktischen Variablen substituiert werden.
Die im nächsten Schritt ausgeführte DED-Regel stellt eine der in Abschnitt 3.6 vorgestellten *eingebauten* Deduktionsregeln dar, die das Konnektiv "⇒" als logische Implikation interpretiert und eine Einführung einer lokalen *Annahme* ("Hypothese") bewirkt. Lokale Hypothesen werden in den ersten Zeilen des (Teil)beweises aufgelistet und die hierunter zu beweisenden (Teil)ziele werden durch Einrücken gekennzeichnet.

[4]Ebenso wie die Formulierung geeeigneter (zumindest korrekter) Kalküle liegt es in der Verantwortung des Benutzers, nur "sinnvolle" Ausdrücke für die syntaktischen Variablen zu substituieren. Dies ist ein Beispiel für das Abweichen von dem in Kapitel 3 verfolgten formalen Ansatz, da auf Antezedenten, die die Syntax der "Substitute" einschränken, verzichtet wird.

[5]Die Antezedenten einer angewandten Regel werden gemäß ihrer Anordnung *geordnet* auf dem Zielkeller abgelegt: der am weitesten links stehende Antezedent wird zuoberst auf dem Zielkeller abgelegt usw. .

```
PROOF

   1  a: POW(b)                                  HYP
   2    a inc b                                  1 set.1
   3    a inc (b\/c)                             2 set.3
   4  a: POW(b) => a inc (b\/c)                  DED
   5  prove(a: POW(b) => a inc (b\/c))           4 prover.1

END OF PROOF
```

Abbildung 4.4: Beweis der ersten Aussage

Im anschließenden Schritt wird Regel 3 der Theorie `set` angewandt und damit versucht, die Inklusionsbeziehung zur Vereinigungsmenge auf die Inklusion in der "linken" der beiden vereinigten Mengen zurückzuführen. Durch die folgende Anwendung der Regel 1 von `set` wird die `inc`-Beziehung in eine Elementbeziehung umgewandelt. Das entstehende Ziel entspricht der getroffenen Annahme und ist damit vollständig reduziert. Die Abarbeitung des nun auf dem Zielkeller obenliegenden `bwrite`-Teilziels bewirkt eine Erfolgsmeldung auf dem Bildschirm. Damit ist der Zielkeller leer und der Beweis beendet.
Auch die zweite Beispielformel läßt sich mit Hilfe der `set`-Regeln beweisen, wie Abbildung 4.5 zeigt.

```
PROOF

   1  a: POW(b)                                  HYP
   2  e: b                                       HYP
   3    a\/{e}: POW(b)                           2 1 set.2
   4    a\/{e} inc b                             3 set.1
   5  a: POW(b) & e: b => a\/{e} inc b           DED
   6  prove(a: POW(b) & e: b => a\/{e} inc b)    5 prover.1

END OF PROOF
```

Abbildung 4.5: Beweis der zweiten Aussage

4.2.2 Vorwärtsdeduktionen

Der Beweis der dritten Regel der Theorie `exercise` geschieht mit Hilfe einer zusätzlichen Theorie `fwdset` (Abbildung 4.6), deren Regeln aufgrund der angegebenen Taktiken (vgl. u.) ausschließlich in Vorwärtsrichtung angewandt werden. Inhaltlich handelt es sich erneut um elementare

Beziehungen der Mengentheorie. Merkmal fast aller Regeln in `fwdset` ist eine Schwierigkeit,

```
THEORY fwdset IS

        (S inc T) & (T inc U)  =>  (S inc U)
      ;
        (S inc T) & (T inc S)  =>  (S = T)
      ;
        (X : S - T)  =>  (X : S)
      ;
        (X : S) & (S inc T)  =>  (X : T)
      ;
        (X : POW(R)) & (R inc S)  =>  (X : POW(S))
      ;
        (X : POW(T-S))  =>  (X : POW(T))

END
```

Abbildung 4.6: Theorie `fwdset`

die bei deren Anwendung im Rückwärtsbeweismodus auftreten würde: im Regelantezedenten kommen syntaktische Variable vor, die nicht im Regelkonsequenten auftreten, d.h. für die nichts substituiert würde. Im Unterschied zur Abarbeitungsweise von PROLOG unterstützt das B-Tool teilinstantiierte Teilziele nicht. Eine Möglichkeit, derartige Regeln auch im Rückwärtsmodus anzuwenden und die uninstantiierten Variablen gleichzeitig zu binden, besteht jedoch mittels des weiter unten beschriebenen `inhyp`-Kontrollmechanismus.
Der Beweis der dritten Beispielformel der Theorie `exercise` enthält nun Beispiele von Vorwärtsanwendungen von Regeln der Theorie `fwdset` (Abbildung 4.7). Die in Beweiszeilen 4 und 5 erscheinenden Hypothesen ergaben sich hierbei per Vorwärtsdeduktion über der *Hypothese*: `a: POW(b)` wurde über die `fwdset`-Regel 5 aus den Hypothesen 2 und 1 abgeleitet; entsprechend entsteht `e: b` aus den Hypothesen 3 und 1 via Regel 4 von `fwdset`. Beide werden schließlich als Hypothesen zum Beweis des Teilziels in Zeile 6 herangezogen.
Hierbei ist zu beachten, daß die aus einer lokalen Hypothese vorwärtsabgeleiteten Formeln selbst wieder nur die *lokale* Hypothese erweitern.
Die Beschreibung des *Verfahrens zur Vorwärtsdeduktion* ist zu präzisieren. Sei $H_1, \ldots, H_n$ die Liste der bisherigen Hypothesenelemente, $H_1^*, \ldots, H_k^*$ die durch die Anwendung der Regel DED hinzukommenden "neuen" Hypothesenelemente. Dann verläuft die Vorwärtsdeduktion wie folgt:

1. Setze $L := H_1^*, \ldots, H_k^*$, die Liste der "neuen" Hypothesenelemente.
   Setze $H := H_1, \ldots, H_n, H_1^*, \ldots, H_k^*$, die Liste aller Hypothesenelemente.
   (H und L werden als *Queues* verwaltet.)

2. Solange L nicht leer ist, wiederhole:

   (a) Wähle A, das erste Element von L. Lösche A in L.

```
PROOF

  1  d inc b                                          HYP
  2  a: POW(d)                                        HYP
  3  e: d                                             HYP
  4    a: POW(b)                                      2 1 fwdset.5
  5    e: b                                           3 1 fwdset.4
  6    a\/{e}: POW(b)                                 5 4 set.2
  7    a\/{e} inc b                                   6 set.1
  8    a\/{e} inc (b\/c)                              7 set.3
  9  d inc b & a: POW(d) & e: d => a\/{e} inc (b
     \/c)                                             DED
 10  prove(d inc b & a: POW(d) & e: d => a\/{e}
     inc (b\/c))                                      9 prover.1


END OF PROOF
```

Abbildung 4.7: Beweis der dritten Aussage

(b) Führe auf A Vorwärtsdeduktionen in der durch die Vorwärtstaktik spezifizierten Reihenfolge durch: ist $A_1 \& \ldots \& A_m \rightarrow G$ eine Regel einer durch die Vorwärtstaktik spezifizierten Theorie, so versuche, A mit A_1 zu unifizieren. Ist dies möglich, so versuche in der Reihenfolge $i = 2, \ldots, m$, die Regelantezedenten A_i durch weitere Hypothesenelemente konsistent zu instantiieren. (Hierbei ist jeweils die *Gesamt*hypothese H von oben nach unten zu durchsuchen.) Können alle Antezedenten durch Hypothesenelemente gebunden werden, so füge den entsprechend instantiierten Regelkonsequenten zu H und L hinzu.

Der Iterationsschritt muß nicht unbedingt terminieren. Das B-Tool ermöglicht es daher, Höchstschrittanzahlen für die Vorwärtsdeduktion (und die Rückwärtsdeduktion) zu vereinbaren, nach denen automatisch gestoppt wird[6].
Aus Effizienzgründen werden die "neuen" Hypothesenelemente jeweils lediglich gegen den am weitesten links stehenden Regelantezedenten A_1 gematcht. U.a. deshalb ist die Anordnung der Antezedenten der Regeln von Theorien, die in Vorwärtstaktiken referenziert werden, bedeutsam. Die *Anzahl* m der Regelantezedenten einer vorwärtsanzuwendenden Regel sollte möglichst klein sein.

[6] In der aktuellen B-Tool-Version ist dies derart verwirklicht, daß Schritt 2b für ein festes A immer zuendegeführt wird und daher immer noch Endlosschleifen auftreten können.

4.3 Beziehungen zur metalogischen Sicht

Der vorangegangene Abschnitt hat die übliche, vom Benutzer eingenommene Sichtweise von Beweisen auf dem B-Tool anhand von Beispielen vorgestellt und die beiden Beweismechanismen Vorwärts- und Rückwärtsdeduktion identifiziert. Es soll nun der Zusammenhang zu der in Kapitel 3 eingenommenen metalogischen Sicht besprochen werden. Hierzu ist zu diskutieren,

- wie der Benutzer *syntaktische Vereinbarungen* entsprechend dem in Abschnitt 3.5 motivierten Vorgehen vorgeben kann,
- in welcher Weise die gemäß Abschnitt 3.6 im B-Tool *eingebauten* Schlußregeln des Kalküls des natürlichen Schließens angewandt werden und insbesondere, wie die Regel TRANS die Durchführung von Beweisschritten auf dem B-Tool modelliert,
- welche zusätzlichen Regeln, wie z.B. die *Substitution* (vgl. Abschnitt 3.5), *operational* vordefiniert sind,
- auf welcher logischen Ebene das in B-Tool-Theorien verwendete Regelkonnektiv "$\Rightarrow$" anzusiedeln ist und welche Rolle dem metalogischen Sequenzenoperator-Konnektiv sowie dem Metaprädikat "$\vdash$" zukommt.

4.3.1 Syntaktische Vereinbarungen

In der Syntaxdefinition für die vom B-Tool manipulierten, als wohlgeformt anzusehenden Zeichenketten wird festgelegt, welche Symbole als *Operanden*, welche als *Operatoren* bestimmter Stelligkeit, Priorität und Assoziativität und welche als *Klammern* aufgefaßt werden (vgl. auch [Abri86]). Die Menge der als ("semantische") objekttlogische Variablen angesehenen Symbole (Menge `var`) ist, unabhängig von den sonstigen syntaktischen Vereinbarungen, als die Menge aller Kleinbuchstaben vordefiniert. Darüberhinaus ist die Formelsprache des B-Tools *offen*, d.h. sie kann vom Benutzer in Abhängigkeit von seiner Anwendung erweitert bzw. abgeändert werden. Solcherlei Notationsadaption geschieht über die Anpassung einer Datei namens `SYMBOL`, die die Operatordefinitionen beinhaltet und beim Start des B-Tools automatisch geladen wird. (Fehlt `SYMBOL`, so wird eine Fehlermeldung gegeben.)
In Anhang A ist die Datei `SYMBOL`, ergänzt um die für die DAIDA-Anwendung vorzunehmenden Definitionen, abgebildet. Spalte 1 der Datei enthält die Symbol*syntax*, Spalte 2 vereinbart *Stelligkeit*[7], Spalte 3 enthält zusätzliche Information zur Art des Symbols und in Spalte 4 wird die *Priorität* von Operatoren festgelegt. In einer zusätzlichen Spalte erscheinen zu "öffnenden" Symbolen wie Klammern das zugehörige schließende Pendant (Spalte 2 : `opn` = "open" bzw. `clo` = "close").
Für binäre Operatoren ist *Linksassoziativität* als Default vorgegeben; wird in der 5. Spalte `right` angegeben (nur bei *binären* Operatoren), so assoziiert der Operator nach rechts.
Ein Beispiel: in

```
.          bin      nrml      10  right
```

[7] "Operatoren" der Stelligkeit `atm` (= "atomar") sind wie üblich als *Konstanten*, d.h. als *Operanden* aufzufassen

wird "." als zweistelliger Operator mit Rechtsassoziativität vereinbart. Dies hat den Hintergrund, daß "." als formales Trennzeichen in Formeln mit Quantoren verwendet wird ([Stuc90]). Aufbauend auf den Festlegungen der Datei `SYMBOL` wird nun die syntaktische Wohlgeformtheit der auf Metaebene als Zeichenketten aufgefaßten Formeln durch das B-Tool *algorithmisch* durchgeführt[8].

4.3.2 Die voraxiomatisierten Regeln des Kalküls des natürlichen Schließens

Die Diskussion der Beispielbeweise hat gezeigt, daß das B-Tool für die zu beweisenden Ziele *Hypothesen* verwaltet. Jedes Teilziel ist per Rückwärtsdeduktion auf eine Formel der eventuell durch Vorwärtsdeduktion erweiterten (lokalen) Hypothese zu reduzieren.
Hier läßt sich nun die Beziehung zu dem in Abschnitt 3.6 vorgestellten Kalkül des natürlichen Schließens herstellen. Die dort eingeführte Notation

$$H_1, \ldots, H_n \quad \vdash \quad G \tag{4.1}$$

beschreibt in der Terminologie des B-Tools, daß das Teilziel G unter der Hypothese $H_1, \ldots, H_n$ nachzuweisen ist. Hingegen "sieht" der B-Tool-Benutzer während Deduktionen i.a. nur das zu beweisende Teilziel G; er kann sich jedoch vom B-Tool die vollständige Hypothese zeigen lassen.

4.3.2.1 Die Regeln HYP, DED und GEN

Die Beispielbeweise enthalten jeweils Anwendungen der vordefinierten Regeln DED und HYP. Die Regel DED interpretiert den auch zur Separation von Regelantezedenten und Konsequent einer B-Tool-Regel verwendeten Operator "$\Rightarrow$" *als logische Implikation*. Die Anwendung einer dieser eingebauten Regeln auf dem B-Tool geschieht direkt in der durch die *Sequenzen*darstellungen dieser Regeln (Abschnitt 3.6) beschriebenen Form.

4.3.2.2 Die Regel TRANS

Es wird nun gezeigt, in welcher Weise die Regel TRANS in Form der Beweismechanismen Vorwärts- und Rückwärtsdeduktion operational realisiert wird. Sie ermöglicht die Anwendung von Theorieelementen als *Regeln* und gibt somit die formale Rechtfertigung für die bereits oben verwendete Bezeichnung.
Grundlegend ist eine Beziehung, die wegen der vordefinierten Regel DED für jede Theorie"regel" gilt:

$$\vdash (H_1 \,\&\, \ldots \,\&\, H_m \Rightarrow G) \quad \stackrel{\text{DED}}{\longrightarrow} \quad H_1 \,\&\, \ldots \,\&\, H_m \vdash G \tag{4.2}$$

[8] hierbei besteht allerdings die Ungenauigkeit, daß die syntaktische Klassen frm und trm nicht unterschieden werden.

Die modelltheoretisch als Implikations*formeln* aufgefaßten Theorieelemente können somit durch einmaliges Anwenden von DED in eine Form gebracht werden, in denen die Regelantezedenten in der lokalen Hypothese des Regelkonsequenten stehen. Wird ein Theorieelement rückwärts "angewandt", so wird dies, ausgehend von der auf diese Weise erhaltenen abgeleiteten Darstellung, durch die vordefinierte Regel TRANS gerechtfertigt: in

$$\frac{\begin{array}{c} frm\, G_1, \ldots, frm\, G_n \\ frm\, H_1, \ldots, frm\, H_m \;\; frm\, P \\ H_1, \ldots, H_m \vdash P \\ G_1, \ldots, G_n \vdash H_1 \\ \vdots \\ G_1, \ldots, G_n \vdash H_m \end{array}}{G_1, \ldots, G_n \vdash P} \;\text{TRANS}$$

entspricht das anzuwendende Theorieelement dem Antezedenten $H_1, \ldots, H_m \vdash P$. Das im Konsequent stehende Teilziel P ist unter der lokalen Hypothese $G_1, \ldots, G_n$ zu beweisen. Durch Anwendung der Regel im *Rückwärts*beweismodus entstehen hieraus m Teilziele entsprechend den Antezedenten $H_1, \ldots, H_m$ der angewandten Regel, die jeweils unter der lokalen Hypothese des Ausgangsteilziels zu beweisen sind.
Auch *Vorwärts*deduktionen werden durch die Regel TRANS formal gerechtfertigt. Neben der Regel DED wird zusätzlich das Axiomenschema HYP benötigt. Die vorwärtsanzuwendende Regel entspricht wieder dem Antezedenten $H_1, \ldots, H_m \vdash P$ der Regel TRANS. Es läßt sich nun unmittelbar durch Induktion über die Anzahl k der Vorwärtsdeduktionsschritte über einer Hypothese $G_1, \ldots, G_n$ nachweisen: für jede in der Hypothese stehende oder durch B-Tool-Vorwärtsdeduktion aus der Hypothese folgerbare Formel H gilt im Rahmen des Kalküls des natürlichen Schließens: $G_1, \ldots, G_n \vdash H$. Für die Hypothesenelemente gilt dies wegen der Axiome HYP (Induktionsanfang, $k = 0$). Im Induktionsschritt $k \longrightarrow k + 1$ ergibt sich die Behauptung unter Verwendung von TRANS unmittelbar in Form des Konsequenten des Sequenten TRANS, da im $k+1$-sten Vorwärtsschritt lediglich auf Formeln zurückgegriffen wird, für die die Induktionsvoraussetzung gilt. Die Reduktion von Teilzielen auf durch Vorwärtsdeduktion aus der Hypothese ableitbare Elemente ist daher korrekt.
Sowohl Vorwärts- als auch Rückwärtsdeduktionen auf dem B-Tool werden somit formal durch die vordefinierte Regel TRANS gerechtfertigt. Die beweistheoretische Bezeichnung "Regel" für die objektlogischen Implikationsformeln der Theorien ist also auch formal begründet.

4.3.3 Operational vordefinierte Regeln

Abschnitt 3.5 nennt Beispiele von Regeln, für die sich eine *operationale* Vordefinition anbietet. Auf dem B-Tool sind u.a. folgende Regeln vorgegeben:

- Die *Metagleichheitsregel* wendet Theorieelemente der Gestalt $S == T$ an: die durch S bezeichnete Zeichenkette wird durch T ersetzt [9]. Neben den Implikationsregeln, deren An-

[9] Um Endlosschleifen bei der Anwendung derartiger Regeln zu vermeiden, wird die Gleichung *nur von links nach rechts* angewandt.

wendung durch die Regel TRANS gerechtfertigt wird, ist somit außerdem die *Anwendung von Gleichungen* in Form von Termersetzungsschritten eingebaut.

- Die Regel ARI interpretiert bestimmte Terme als *arithmetische Ausdrücke* und führt entsprechende Auswertungen durch. Bespielsweise wird $1 + 1$ unter ARI zu 2. Formal entspricht dies einer Voraxiomatisierung objektsprachlicher Termersetzungsregeln auf metalogischer Ebene, die in operationaler Form eingebaut ist.

- Die *Substitutionsregel* SUB wird ebenfalls operational umgesetzt. Ihre Anwendung auf einen Substitutionsausdruck $[x := E]F$ ersetzt alle freien Vorkommen von x in F durch E. *Gebundene* Vorkommen von x in E werden zum Vermeiden von "variable capture" automatisch umbenannt ("Alpha-Konvertierung", vgl. o.). Kommt die zu substituierende Variable in einer Hypothesenformel *frei* vor, so ist die Substitution *nicht* ausführbar; die Hypothese wird in den Substitutionsvorgang nicht einbezogen.
  Es wird sich zeigen, daß der Regel SUB eine wichtige Rolle für den Einsatz des B-Tools im Rahmen des DAIDA-Projekts zukommt.

Darüberhinaus existieren weitere eingebaute Regeln ohne *mathematischen* Hintergrund; diese werden weiter unten kurz beschrieben.

4.3.4 Verzicht auf die vordefinierten Regeln

Eine alternative Sichtweise der formulierten Theorien im Rahmen der Metalogik BL ergibt sich, wenn man das Regelkonnektiv "$\Rightarrow$" nicht als objektlogische Implikation, sondern als metalogischen *Sequenzenoperator* auffaßt. Diese von [Gard88] nicht erwähnte Interpretationsmöglichkeit kann dazu verwendet werden, das B-Tool als *offenes Deduktionssystem ohne die Vordefinitionen des Kalküls des natürlichen Schließens* einzusetzen.
Zu diesem Zweck müssen allerdings *alternative Notationen* für die in den vordefinierten Regeln DED, GEN, SUB und ARI referenzierten objektlogischen Symbole wie '$\Rightarrow$" und "$\forall$" eingeführt werden, sofern sie in der zu axiomatisierenden Objektlogik vorkommen. Dies ist jedoch wegen der offenen Formelsprache des B-Tools unproblematisch.
Die benutzerdefinierten Theorieelemente lassen sich nun als *Sequenzen* der Metalogik BL auffassen, wobei allerdings die formalen Antezedenten fehlen können, sofern entsprechende syntaktische Vorvereinbarungen getroffen wurden. Vor jeder "nichtformalen" Antezedent- oder Konsequentformel steht *implizit*, d.h. für den Benutzer unsichtbar, das BL-Metaprädikatensymbol "$\vdash$".
Die zuvor durch die vordefinierte Objektkalkülregel TRANS modellierten Deduktionsmechanismen des B-Tools werden nun durch das in Abschnitt 3.4.1 beschriebene Verfahren des Schließens auf Sequenzenformelebene gerechtfertigt. Wegen der Übereinstimmung mit dem Schließen im Metakalkül (Lemmas 3.1 und 3.2) realisiert das B-Tool damit, von diesem Blickwinkel gesehen, Deduktionen auf *Metaebene*.
Es ist unklar, warum [Gard88] diese alternative Sichtweise des B-Tools als offenes Deduktionssystem *ohne* Voraxiomatisierung einer Objektsprache nicht erwähnt. In [Gard90] wird jedoch zugegeben, daß in den bisherigen B-Tool-Versionen keine eindeutige konzeptuelle Trennung von Objekt- und Metaebene vollzogen wird. Das "implizite" Metaprädikatensymbol "$\vdash$" ist ein Beispiel hierfür: Regeln können (aus der Sicht des Benutzers) objektlogische und metalogische Formeln nebeneinander enthalten.

4.4 Die Steuerungsmechanismen des B-Tools

Es ist zweckmäßig, zwischen folgenden Arten von Steuerungsmechanismen zu unterscheiden:

- *systemseitig vorgegebene Steuerungen* der Deduktion, in der logischen Programmierung die Berechnungsregel und die Suchstrategie auf dem zugehörigen SLD-Baum ([Beck88])
- *benutzerseitige Steuerungskonzepte* außerhalb der objektlogischen Ebene :
  - statische
  - dynamische (laufzeitabhängige)

  Konzepte.

Diese sollen im folgenden für das B-Tool identifiziert werden.
Nimmt man die in [Gard88] vorgeschlagene, in Abschnitt 4.3.2 beschriebene Sicht an, dann ist folgendes *zielgerichtete Rahmenverfahren* vorgegeben, gemäß dem die Deduktionen ablaufen:

> Zu Beginn liegt das zu beweisende Ziel auf dem Zielkeller. Im allgemeinen Schritt wird *anhand der benutzerdefinierbaren Vorgaben der Steuerungskomponente* eine auf das aktuelle, auf dem Zielkeller zuoberst liegende Teilziel (rückwärts) *anwendbare* Regel (eine vordefinierte Sequenz oder eine benutzerdefinierte Regel) ausgewählt. Ist die anzuwendende Regel
>
> - das *eingebaute* "Axiom" HYP: das aktuelle Teilziel wird nicht mehr betrachtet, da es aus der Hypothese folgt, d.h. ein Hypothesenelement ist oder aus der Hypothese via Vorwärtsdeduktion folgerbar ist.
> - die *eingebaute Regel* GEN: GEN wird, wie durch die in Abschnitt 3.6 angegebene Sequenz beschrieben, ausgeführt.
> - die *eingebaute Regel* DED: DED wird, wie durch die in Abschnitt 3.6 angegebene Sequenz beschrieben, ausgeführt. Auf der nunmehr erweiterten Hypothese werden *Vorwärtsdeduktionen* initiiert.
> - eine *benutzerdefinierte* Theorieregel: in diesem Fall wird formal die Regel DED auf das Theorieelement angewandt (Beziehung 4.2), um dieses im folgenden Schritt mittels TRANS auf das aktuelle Teilziel anzuwenden. Die Antezedenten der anzuwendenden Regeln ersetzen auf dem Zielkeller das aktuelle Teilziel. Die Regelantezedenten werden gemäß ihrer Anordnung "gepusht", wobei der am weitesten links stehende Regelantezedent auf dem Zielkeller zuoberst abgelegt wird.
>
> Es findet *kein Backtracking* statt, d.h. einmal erfolgte Regelanwendungen werden nicht mehr rückgängig gemacht[10].

[10] Diese Ausssage gilt jedoch nur für den *automatischen* Beweismodus des B-Tools, nicht jedoch für *benutzerassistierte* Deduktionen, vgl. u.

Diese Beschreibung des im B-Tool vorgegebenen Deduktionsrahmens verdeutlicht, daß systemseitig Steuerungskonzepte vordefiniert sind, die nicht durch den Benutzer beeinflußt werden können. Hierzu gehört die Entscheidung zur *zielgerichteten Suche* und die aus der Verwaltung eines *Zielkellers* resultierende geordnete Verarbeitung der Ziele (vgl. Abschnitt 2.1.1.2) sowie die nicht zulässige unabhängige Anwendung von TRANS. Außerdem werden Vorwärtsdeduktionen nur nach Anwendung der eingebauten Regel DED durchgeführt.
Es bleibt zu beschreiben, welche Hilfsmittel zur Verfügung stehen, den in diesem Rahmen stattfindenden Deduktionsprozeß *von außen* zu beeinflussen. Hierzu bestehen folgende Möglichkeiten, die in eigenen Abschnitten beschrieben werden sollen:

- (statische) Wahl der Regelanordnung innerhalb von Theorien
- (statische) Vorgabe einer *Taktik*, durch die die Auswahl einer anwendbaren Regel gesteuert werden kann
- (dynamische) Einschränkung der *Anwendbarkeit* einer Regel durch zur Laufzeit durchgeführte sog. `inhyp`-Tests
- (dynamische) Vereinbarung einer teilziellokalen Taktik durch sog. `bcall`-Aufrufe.

4.4.1 Definition von Taktiken

Für die Vereinbarung von Taktiken steht eine einfache *Taktiksprache* zur Verfügung, in deren Rahmen für die beiden Beweismodi vereinbart werden kann, welche Regeln zu welchem (relativen) Zeitpunkt während der Deduktion angewendet werden sollen.
Die Sprachprimitiva umfassen

- Referenzierungsmöglichkeiten der einzelnen vordefinierten Regeln bzw. benutzerdefinierter Theorien oder Einzelregeln einer Theorie. So schreibt

  ```
  DED
  ```

  die Anwendung der vordefinierten Regel DED vor (sofern möglich),

  ```
  usertheory.4
  ```

  verlangt die Anwendung der vierten Regel der benutzerdefinierten Theorie `usertheory` und

  ```
  usertheory
  ```

  schließlich erwirkt die Anwendung *einer* Regel aus der Theorie `usertheory`, wobei die Regeln entsprechend ihrer Anordnung *von unten nach oben* auf Anwendbarkeit getestet werden.

- *Sequenzierung*: das Zeichen ";" bewirkt die Hintereinanderschaltung von Taktiken. Beispielsweise bewirkt

  ```
  HYP;GEN;usertheory.2
  ```

zunächst den Versuch der Anwendung von HYP. Falls dies unmöglich ist, wird GEN versucht. Scheitert auch dies, wird die zweite Regel von `usertheory` auf Anwendbarkeit untersucht.
Bei nachfolgenden Deduktionsschritten wird jeweils bei der in der Sequenzierung nachfolgenden Regel(gruppe) fortgefahren. War beispielsweise GEN anwendbar, so wird im nächsten Deduktionsschritt versucht, `usertheory.2` anzuwenden. Ist dies unmöglich, wird mit HYP fortgefahren (implizite Taktikiterierung). Wurde die Taktik erfolglos durchlaufen (hier bei erneutem Erreichen von `usertheory.2`), so stoppt der Deduktionsprozeß erfolglos.

- *Iterierung*: das Zeichen "~" steht für wiederholte Anwendung einer Teiltaktik. In

  ```
  (HYP;GEN)~;usertheory.2
  ```

  wird wiederholt versucht, zunächst HYP und dann GEN anzuwenden. Nur falls *sowohl* HYP *als auch* GEN nicht anwendbar sind, wird mit `usertheory.2` fortgefahren.

Iterierung und Sequenzierung können beliebig geschachtelt werden.
Zu jeder Theorie ist es nun möglich, Taktiken sowohl für den Rückwärts- als auch für den Vorwärtsbeweismodus zu definieren. Die entsprechenden Theorieklauseln sind `TAC` und `FTAC`. Sollen Regeln aus einer der Theorien bewiesen werden, so werden die für sie vereinbarten Taktiken automatisch herangezogen[11].

4.4.2 Der `inhyp`-Test

Der `inhyp`-Kontrollmechanismus ermöglicht es, die Rückwärtsanwendbarkeit benutzerdefinierter Regeln durch *zusätzliche Bedingungen* einzuschränken: Regeln können Antezedenten der Gestalt `inhyp(Teilziel)` enthalten, wobei das Teilziel `Teilziel` Variablen enthalten kann, die durch das "Matchen" des Antezedenten instantiiert werden. Eine solche Regel ist erst dann anwendbar, wenn durch die sich ergebende Instantiierung ein Teilziel `Teilziel'` entsteht, welches (nach ggf. weiteren Instantiierungen) einem Hypothesenelement entspricht. So ist beispielsweise die Anwendung der Regel

```
(x: T) & inhyp(S inc T) => (S\/{x} inc T)
```

an die Bedingung geknüpft, daß das Teilziel `S inc T` nach der sich durch die Regelanwendung ergebenden Instantiierung in der Hypothese zu finden ist. Variablen der Antezedensformeln, die nicht im Regelkonsequenten auftreten, und objektlogische Variable (vgl. folgendes Beispiel) werden durch den `inhyp`-Test *gebunden* (vgl. Bemerkung in Abschnitt 4.2.2).
Ist

```
(set1 inc set2) =>  (set1\/{element} inc y)
```

[11] Die vorgegebenen Taktiken können aber vom Benutzer ggf. interaktiv abgeändert werden.

ein zu beweisendes Ziel, so steht nach einmaligem DED die Formel `(set1 inc set2)` (`set1` und `set2` werden als Konstanten interpretiert) in der Hypothese. Auf das noch zu beweisende Ziel `(set1\/{element} inc y)` ist obige Regel anwendbar, denn:

1. Variable `S` wird an `set1` gebunden, Variable `x` wird an `element` gebunden. `T` wird *hier* nicht instantiiert, da `y` eine Variable ist.
2. Nach Schritt 1 entsteht der `inhyp`-Test `inhyp(set1 inc T)`, der erfolgreich verläuft, da nach Instantiierung von `T` mit `set2` das Hypothesenelement `(set1 inc set2)` entsteht.
3. Die Regel ist daher anwendbar und das noch zu beweisende Teilziel lautet `element : set2` (`T` wurde durch den `inhyp`-Test instantiiert).

Auch der `inhyp`-Test kann also zu Instantiierungen führen.
Der `inhyp`-Test wird als *dynamisches* Konzept klassifiziert, da die Hypothesenformel, auf die getestet wird, *von Instantiierungen abhängen* kann.

4.4.3 Der `bcall`-Aufruf

Ein im Vergleich zum `inhyp`-Test wesentlich mächtigeres Konzept der dynamischen Beeinflussung der Steuerung stellt der sog. `bcall`-Aufruf dar. Die Abarbeitung von Teilzielen der Gestalt

```
bcall( lokale Taktik :  lokales Teilziel )
```

bewirkt eine *lokale Änderung der Taktik der gerade laufenden Deduktion*. Das lokale Teilziel wird unter Verwendung der lokalen Taktik verarbeitet. Ist das lokale Teilziel vollständig reduziert, so wird die bislang aktive Taktik (sowie ihr Abarbeitungszustand) wieder aktuell[12]. Dies entspricht einem *Unterprogrammkonzept auf der Kontrollebene*. Die entsprechende Regel heißt CAL.

4.4.4 Regelanordnung

Gemäß den Ausführungen in Abschnitt 4.4.1 ist es bedeutsam, in welcher Reihenfolge die Regeln einer (in Rückwärtstaktiken referenzierten) Theorie angeordnet werden. Der Benutzer hat die (eingeschränkte) Möglichkeit, durch eine gezielte Anordnung die Anwendung bestimmter Regeln zu forcieren. Dies soll nun konkretisiert werden.
Die (Rückwärts-)Anwendbarkeit einer von der Taktik selektierten benutzerdefinierten Regel hängt von zwei Faktoren ab: die Unifizierbarkeit des aktuellen Teilziels mit dem Kopf der Regel und die Gültigkeit aller `inhyp`-Antezedenten der Regel.

[12] Technisch kann dies dadurch realisiert werden, daß die bislang aktive Taktik auf dem Zielkeller direkt unter dem lokalen Teilziel "gerettet" und bei Sichtbarwerden reaktiviert wird.

Definition 4.1 (Anwendbarkeit einer Regel) *Eine Regel R heißt* anwendbar für ein Teilziel G *und eine Hypothese H g.d.w. der Regelkopf mit dem Teilziel G unifizierbar ist und die Hypothese H alle* `inhyp`*-Antezedenten von R erfüllt.*

Aufbauend hierauf kann eine partielle Ordnung auf den Regeln einer Theorie definiert werden:

Definition 4.2 (partielle Anwendbarkeitsordnung "$\prec$") *Eine Regel $R_1 = (A_1 \wedge \ldots \wedge A_n \Rightarrow F)$ heißt* allgemeiner als *eine Regel $R_2 = (B_1 \wedge \ldots \wedge B_m \Rightarrow G)$, geschrieben $R_2 \prec R_1$, g.d.w. R_1 in jeder Situation anwendbar ist, in der R_2 anwendbar ist, d.h. für alle Hypothesen H[13] und Teilziele T gilt: Falls T mit G unifizierbar ist und H die* `inhyp`*-Antezedenten von R_2 erfüllt, so ist T auch mit F unifizierbar und H erfüllt die* `inhyp`*-Antezedenten von R_1 (dies impliziert u.a. , daß G eine Instanz*[14] *von F ist).*

Hieraus folgt: falls $R_2 \prec R_1$ gilt und R_1 und R_1 derselben Theorie T angehören, so ist R_1 in T oberhalb von R_2 anzuordnen, da andernfalls R_2 nie angewandt wird, wenn eine Taktik die Theorie T referenziert (Die ebenfalls mögliche Referenzierung *einzelner* Regeln durch Taktiken ist eher unüblich).
Für den Benutzer bestehen Freiheitsgrade, falls zwei Regeln R_1 und R_2 einer Theorie für eine bestimmte Menge aktueller Teilziele und Hypothesen gemäß der Gestalt ihrer Regelköpfe F bzw. G und `inhyp`-Antezedenten bezüglich der Anwendbarkeit *äquivalent* sind, *ohne* daß eine der Regeln allgemeiner als die andere ist. Dies ist beispielsweise dann der Fall, wenn die Regeln keine `inhyp`-Antezedenten besitzen und keiner der Regelköpfe eine Instanz des anderen ist, jedoch eine gemeinsame Menge reduzierbarer Teilziele existiert. *Unifikationstheoretisch* heißt dies, daß die beiden Regelköpfe *unifizierbar* sind[15], jedoch keiner der Regelköpfe eine Instanz des anderen ist. Die Menge der von beiden Regeln reduzierbaren Teilziele läßt sich in diesem Fall als $\mu(F)$ schreiben, wobei μ der *allgemeinste Unifikator* der Literalmenge $\{F, G\}$ ist[16].
In diesem Fall entscheidet die vom Benutzer gewählte Regelreihenfolge innerhalb einer Theorie darüber, welche der Regeln angewandt wird. Da der Benutzer damit die Möglichkeit hat, unter Berücksichtigung des eingebauten Abarbeitungsmechanismus Vorgaben für die bevorzugte Anwendung von Regeln zu machen, kann man die Reihenfolgewahl als (wenn auch bescheidenes) indirektes Konzept der Deduktionssteuerung bezeichnen.

4.5 Konzepte mit operationaler Semantik

In den Abschnitten 4.3 und 4.4 wurden Konzepte des B-Tools im Rahmen der zugrundeliegenden Metalogik bzw. als Komponenten der Steuerung klassifiziert. Es sollen nun einige Aspekte des B-Tools diskutiert werden, die sich nicht im Rahmen dieser durch die Schichtenarchitektur für Deduktionssysteme induzierten Systematik beschreiben lassen; im einzelnen

[13] Diese Quantifizierung ist auf die abzählbar unendlich vielen syntaktisch wohlgeformten Formeln des Diskursbereichs beschränkt.

[14] Unter einer *Instanz* einer Formel G ist eine Formel G' zu verstehen, die aus G durch Substitution wohlgeformter Ausdrücke für die Variablen entsteht.

[15] Für eine Diskussion *unifikationstheoretischer Begriffe* vgl. z.B. [ChLe73].

[16] natürlich gilt $\mu(F) = \mu(G)$

- *Ein- und Ausgabefunktionen*
- *Listen- und Zeichenkettenverarbeitung*
- *dynamische Erweiterbarkeit* der Regelbasis und *Lemmagenerierung*

Die hier vorgestellten Konzepte sind u.a. essentiell für die vom B-Tool zu bewältigenden Aufgaben als Transformationstool.

4.5.1 Eingabe und Ausgabe

Teilziele spezieller Gestalt erlauben die Durchführung *interaktiver* Ein- und Ausgaben. Die Regel der Theorie `prover` des obigen Beispiels enthält ein Teilziel der Gestalt `bwrite("Text")`, dessen Abarbeitung die Ausgabe der Zeichenkette `Text` auf dem Bildschirm bewirkt. Entsprechend verlangen Teilziele der Gestalt `bread("Prompt",var)` eine Eingabe vom Terminal, mit der die Variable `var` gebunden wird.
Das Öffnen und Schreiben *sequentieller Dateien* geschieht durch die "operationalen" Teilziele `bconnect("Dateiname")` bzw. `bprint("Text")`.

4.5.2 Listen- und Zeichenkettenverarbeitung

Das B-Tool bietet Deduktionsregeln operationaler Gestalt an, mit deren Hilfe es möglich ist, vorliegende Teilziele *syntaktisch zu manipulieren*. Derartige Regeln werden in der Taktik mit `MAP` aktiviert. Die speziellen eingebauten Regeln REV bzw. QUO führen Listeninvertierung bzw. Zeichenkettenbildung ("Quoting") durch ([Stuc90]).

4.5.3 Dynamische Theorieänderungen

Teilziele spezieller Gestalt ermöglichen eine dynamische Ergänzung von B-Tool-Theorien und Modifikation der den Theorien zugeordneten Taktiken:

- `Theoriename+(Regel)`: legt `Regel` als zusätzliche Regel am Ende der Theorie `Theoriename` ab. Existiert diese Theorie noch nicht, so wird eine entsprechende Theorie neu "eröffnet".
- `Theoriename*(Regel)`: wie zuvor, jedoch wird diesmal `Regel` als *Lemma* der aktuell durchgeführten Deduktion verwaltet, welches im Anschluß daran noch zu beweisen ist. "`*`" zeigt (auch an anderer Stelle) an, daß die Regel noch zu beweisen ist.
- `Theoriename/(Taktik)`: vereinbart für die Theorie `Theoriename` die neue *Rückwärts*taktik `Taktik`.
- `Theoriename-(Taktik)`: vereinbart für die Theorie `Theoriename` die neue *Vorwärts*taktik `Taktik`.

Die Möglichkeit, während eines Beweises neue Theorieeinträge zu erzeugen, entspricht in PROLOG die *dynamische Erweiterbarkeit der Regelbasis* über das "Metaprädikat" ASSERT. Daß für derartige Konstrukte keine "saubere", deklarative Semantik definiert werden kann, ist aus Betrachtungen zur logischen Programmierung bekannt ([Beck88]).

4.6 Das B-Tool als Zustandsübergangssystem

Die Diskussion in den vorangegangenen Abschnitten hat gezeigt, in welcher Weise das B-Tool die metalogischen Konzepte von BL umsetzt und welche Steuerungskonzepte angeboten werden. Anhand dieser Darstellung kann nun, gemäß den Ausführungen in Kapitel 2, das B-Tool als *logisches Zustandsübergangssystem* in einer vereinheitlichten Weise dargestellt werden. Hierzu ist zweierlei zu leisten:

1. Definition eines *Konfigurationsbegriffs* entsprechend eines Modells der internen Zustände, die das B-Tool annimmt,
2. Definition einer *Ableitbarkeitsrelation*, orientiert an den aus den elementaren Deduktionsschritten resultierenden Konfigurationsübergängen.

4.6.1 Konfigurationen

Bei der Definition eines Konfigurationsbegriffs ist u.a. zu berücksichtigen, daß unter Verwendung des `bcall`-Konzepts *lokale* Taktiken vereinbart werden können. Ein mögliches Modell für die vom B-Tool angenommenen Zustände ergibt sich durch folgende Definition:

Definition 4.3 (B-Tool-Konfiguration) *Eine* B-Tool-Konfiguration K *ist von der Gestalt*

$$K \quad = \quad \langle (ZK_1, \tau_1), \ldots, (ZK_n, \tau_n) \rangle$$

wobei die ZK_i Zielkeller *der Gestalt*

$$((H_{1,i}; G_{1,i}), \ldots, (H_{m_i,i}; G_{m_i,i})) \quad , 1 \leq i \leq n,$$

sind, die jeweils Paare von Teilzielen und teilziellokalen Hypothesen enthalten. Zielkeller ZK_i *ist unter dem lokalen* Kontrolltripel $\tau_i = (R_i, V_i, Z)$*, bestehend aus einer Rückwärtstaktik* R_i*, einer Vorwärtstaktik* V_i*, sowie dem Abarbeitungszustand* Z *der Rückwärtstaktik, zu beweisen. Der in* K *am weitesten rechts stehende Zielkeller wird aktuell bearbeitet*[17]*.*

Es wird deutlich, daß das `bcall`-Konzept die Definition des Konfigurationsbegriffs erheblich kompliziert. Es muß jedoch betont werden, daß eine effiziente Implementation mit einem *globalen* Zielkeller auskommt[18]; der hier gewählte Ansatz erleichtert jedoch die Definition der Ableitungsrelation.

[17] K wird selbst als Keller verwaltet

[18] die globale Taktik wird auf den Zielkeller "gerettet" und nach Abarbeitung des `bcall`-Teilziels reaktiviert

4.6.2 Die Ableitungsrelation

Es sollen nun zunächst einige *elementare* Konfigurationsübergänge exemplarisch identifiziert werden. Es wird eine binäre Relation "$\xrightarrow{\text{Abl}}$", orientiert an den durchzuführenden Rückwärtsdeduktionsschrirtten, über der Menge aller B-Tool-Konfigurationen definiert. U.a. gilt:

- Ist die durch das "aktuelle" Kontrolltripel τ spezifizierte Rückwärtsregel DED, dann gilt

  $$\langle \ldots, (\ldots, (H; H_1 \& \ldots \& H_n \Rightarrow G); \tau) \rangle \quad \xrightarrow{\text{Abl}} \quad \langle \ldots, (\ldots, (H, H_1 \ldots H_n, F_1, \ldots, F_k; G); \tau') \rangle$$

  wobei das Kontrolltripel τ' aus τ durch Inkrementierung des Abarbeitungszustands entsteht und $F_1, \ldots, F_k$ durch Vorwärtsdeduktion über der erweiterten Hypothese erhalten wurde[19].

- Ist die nächste gemäß τ anzuwendende Rückwärtsregel $A_1 \& \ldots \& A_n \Rightarrow G$, dann gilt

  $$\begin{aligned} \langle \ldots, (\ldots, (H; F); \tau) \rangle \quad \xrightarrow{\text{Abl}} \quad \langle \ldots, (\ldots, (H; \theta(A_{i_1})), \\ \vdots \\ (H; \theta(A_{i_k})); \tau') \rangle \end{aligned}$$

  wobei das Kontrolltripel τ' aus τ durch Inkrementierung des Abarbeitungszustands entsteht, θ die sich durch die Unifikation von G und F sowie durch die `inhyp`-Antezedenten ergebenden Instantiierungen darstellt und $A_{i_1}, \ldots, A_{i_k}$ die nach Entfernen der `inhyp`-Antezedenten entstehende Antezedensfolge bezeichnet.

- Ist die nächste anzuwendende Regel CAL, dann gilt

  $$\langle \ldots, (\ldots, (H_1; F), (H; bcall(T : G)); \tau) \rangle \quad \xrightarrow{\text{Abl}} \quad \langle \ldots, (\ldots, (H_1; F); \tau'), ((H; G); \sigma) \rangle$$

  wobei das Kontrolltripel τ' aus τ durch Inkrementierung des Abarbeitungszustands entsteht. Durch den `bcall`-Aufruf wird ein neuer, "aktueller" Zielkeller mit lokaler Kontrolle σ geöffnet, die die in T vereinbarte Taktik im "Startzustand" enthält.

Die Fälle, in denen ein lokaler Zielkeller vollständig abgearbeitet ist, sind separat aufzuführen. Die *Ableitbarkeit* ist nun mittels der *reflexiven transitiven Hülle* "$\xrightarrow{*}$" von "$\xrightarrow{Abl}$" definierbar: ein Ziel G heißt *ableitbar relativ zur Taktik* τ g.d.w.

$$\langle((\bot, G), \tau)\rangle \quad \xrightarrow{*} \quad \langle\rangle$$

gilt, wobei das Symbol $\bot$ für die leere (Hypothesen)Liste steht.

[19] Problematisch ist hierbei, daß die Anzahl durchzuführender FWD-Schritte dynamisch wählbar ist. Zur Definition der Ableitbarkeitsrelation muß hier eine *feste* Beschränkung angenommen werden. Darüberhinaus ist der spezifizierte Elementarübergang auf die Fälle einzuschränken, in denen die Vorwärtsdeduktion *terminiert*.

4.7 Mächtigkeit des B-Tools

Aufbauend auf der nunmehr vorliegenden systematischen Darstellungs der Konzepte des B-Tools bieten sich *Mächtigkeitsbetrachtungen* unter folgenden Gesichtspunkten an:

- Analyse der *deklarativen Mächtigkeit* des B-Tools. Hier ist zu untersuchen, welche Beschreibungsmächtigkeit der *logische Kern des B-Tools* (d.h. die Metalogik BL) hat[20].
- Analyse der *operationalen Mächtigkeit* des unter Berücksichtigung auch der operationalen Konzepte zur Verfügung stehenden, in Form eines Zustandsübergangssystems beschreibbaren *Systems* B-Tool.

4.7.1 Deklarative Mächtigkeit

Gegenstand der deklarativen Darstellung auf dem B-Tool sind Objektlogiksprachen, Kalküle und Anwendungsbereiche. Zu untersuchen ist daher, welche Bedingungen von derartigen Diskursobjekten erfüllt werden müssen, um auf dem B-Tool axiomatisierbar zu sein. Offenbar gilt, daß sich genau diejenigen Diskursgegenstände auf dem B-Tool formulieren lassen, die unter Zuhilfenahme syntaktischer Variablen und unter Berücksichtigung der syntaktischen Vordefinitionen durch *endlich viele* BL-Sequenten beschreibbar sind.
Dies läuft auf eine Untersuchung der Mächtigkeit von BL im Hinblick auf die Formulierbarkeit verschiedener Objektlogikkalküle (auch von Logiken höherer Ordnung) hinaus, was hier allerdings nicht geschehen soll. Für einen Logikkalkül bedeutet die Axiomatisierbarkeit (vgl. auch die Diskussion der Beziehungen zur Logikprogrammierung in Abschnitt 3.8):

> Der Kalkül kann, entsprechend der Diskussion in Abschnitt 3.8, einem PROLOG-*ähnlichen Deduktionsmechanismus* unterzogen werden, m.a.W. es kann zu einer Darstellung im Rahmen der Hornklausellogik übergegangen werden, für die eine *kontrollierte Deduktion* effizient realisierbar ist. Die zugehörigen Objektlogik ist dabei *nicht* auf Hornklauseln beschränkt[21].

4.7.2 Operationale Mächtigkeit

Es läßt sich leicht zeigen, daß das B-Tool unter Berücksichtigung aller Konzepte *turingmächtig* ist. Dieser Nachweis könnte beispielsweise in Form der Realisierung eines *Interpreters* für eine einfache imperative Programmiersprache geschehen. Hierzu wären u.a. folgende Teilprobleme zu lösen:

- Verwalten von *Variablen* (realisierbar z.B. mittels der Listenverarbeitungskonzepte)

[20] Man nimmt sinnvollerweise die in Abschnitt 4.3.4 beschriebene Sicht des B-Tools "ohne Voraxiomatisierung" ein.

[21] Dies wird, wie oben beschrieben, durch das Metaprädikat "$\vdash$" ermöglicht, d.h. formal geschieht die Hornlogikdeduktion auf der *Meta*ebene.

- Interpretation der Elementarbefehle (mittels eingebautem "pattern matching", d.h. Unifikation, umsetzbar), und hierzu:
  - Auswertung *boolescher* Ausdrücke (mittels geeigneter Termersetzungsregeln machbar)
  - Auswertung *arithmetischer* Ausdrücke (Regel ARI)
  - Lesen und Schreiben der Variablen
- indirekte Adressierung, Ablaufkontrolle ("Zählfunktionen", Regel ARI)

Die Details eines solchen Nachweises sollen hier nicht besprochen werden.

4.8 Das B-Tool als Softwaresystem

Die aktuelle B-Tool-Version 3 läuft unter der *fensterorientierten Benutzeroberfläche* SUNVIEW auf SUN 3-Arbeitsplatzrechnern mit dem Betriebssystem UNIX.
Die unterschiedlichen Funktionen des Beweisassistenten B-Tool werden dem Benutzer im Rahmen von unterschiedlichen *Menues* angeboten:

1. Das *Hauptmenue* bietet die unterschiedlichen Funktionen zum *Laden und Abspeichern* benutzerdefinierter Theorien an.
2. Das *Theoriebearbeitungsmenue* ist vom Hauptmenue durch Selektion einer Theorie aufrufbar; es ermöglicht u.a. das interaktive *Ergänzen von Regeln* sowie die Änderung der für die Regeln der Theorie vereinbarten Taktik.
3. Das *Regelbearbeitungsmenue* wird vom Theoriebearbeitungsmenue durch die Wahl einer zu bearbeitenden Regel aufgerufen; die selektierte Regel kann z.B. *modifiziert* oder auch *gelöscht* werden.
4. Das *Beweismenue* ist von besonderem Interesse, da es die zentralen *Beweis*funktionen des B-Tools zusammenfaßt (Abb. 4.8). Es wird vom Regelbearbeitungsmenue aufgerufen, falls die selektierte Regel bewiesen werden soll. Dessen wichtigste Funktionen umfassen:
   - *Initiierung automatischer Deduktion* gemäß der für die Theorie der zu beweisenden Regel vereinbarten Taktik, wobei die Anzahl der ununterbrochen durchgeführten Rückwärts- bzw. Vorwärtsdeduktionsschritte durch entsprechende, zu vereinbarende *Beweisschrittquoten* begrenzt ist.
   - Durchführung von Beweisen im *Beweisassistentenmodus*: bei Wahl der Option `step by step` werden dem Benutzer (unter Taktikberücksichtigung) unterschiedliche Regelanwendungen vorgeschlagen. Lehnt der Benutzer einen Vorschlag ab, so erhält er die (gemäß der Taktik) "nächste" anwendbare Regel vorgeschlagen. Gibt es keine weiteren anwendbaren Regeln, so schlägt das Tool die Rücknahme der zuletzt angewandten Regel vor (*benutzergesteuertes Backtracking*).
     Durch Wahl der entsprechenden Menueoption können darüberhinaus *eingebaute* Regeln zur Anwendung vorgegeben werden. `Undo` nimmt den letzten Beweisschritt

```
CR: Loop Until Quota      1: Menu
 z: Step by Step          c: Shell
 w: Undo                  n: Show More Hyp. (on/off)
 x: DED or GEN            j: Show Big Goal (on/off)
 a: HYP                   g: Goals (on/off)
 q: LEM                   l: Change Quota
 b: Tactic Browsing       r: Theories
 v: Dump Hypotheses       *: While Proving
 s: Statistics            4: Recursive Quit
```

Abbildung 4.8: Beweismenue

zurück, `LEM` erklärt das aktuelle Teilziel zum *Lemma* und verschiebt dessen Beweis auf später.

In der Funktion des B-Tools als Beweisassistent wird demnach auch das im automatischen Beweismodus starr vorgegebene Deduktions-Rahmenverfahren (Abschnitt 4.4) teilweise der Benutzerkontrolle unterworfen.

Kapitel 5

Theoretische Grundlagen von Spezifikation und Verfeinerung

Die Softwareentwicklung wird heute allgemein als inkrementeller Prozeß angesehen, in dessen Verlauf zunächst eine Analyse der Anwendungsdomäne durchgeführt wird, um dann in mehreren Schritten ein in der gewählten Zielsprache lauffähiges Programm zu entwickeln. Um eine formale Verifizierbarkeit der durch diese *Entwicklungsphase* gewonnenen Software zu ermöglichen, bedarf es geeigneter Beschreibungssprachen mit *formaler* Syntax- und *Semantik*definition. Durch die Definition einer formalen Semantik wird idealerweise eine Beziehung zwischen Beschreibungen auf unterschiedlichen Stufen der Programm*entwicklung* hergestellt, sodaß schließlich die *Korrektheit* des entwickelten Programms *relativ zur gegebenen Spezifikation* beweisbar wird. Da der Spezifikation eine Funktion als *Kontrakt* zwischen Spezifizierer und Implementierer zukommt (vgl. [BaGo79]), wird hierdurch überprüfbar, ob der Softwareersteller die vom Anwender beauftragte Leistung erbracht hat[1]. Die zur ersten Formalisierung des Ergebnisses der Anforderungsanalyse verwendete Beschreibungssprache sollte daher andererseits auch folgende Kriterien erfüllen:

- Lesbarkeit für beide Parteien des Implementationskontrakts,
- *deklarative*, implementationsferne Beschreibung, die die logische Struktur der Anwendungsdomäne offenlegt und so ein frühzeitiges Erkennen von Inkonsistenzen ermöglicht.

Dies offenbart die Schwierigkeiten beim Entwurf geeigneter Spezifikationssprachen: es wird gute Lesbarkeit einerseits und formale Verifizierbarkeit andererseits verlangt (vgl. auch [BuGo81]). Eine Spezifikation läßt sich auch als *kognitives Modell* auffassen, welches aus der Sicht der Benutzergemeinde auf das Diskurssystem gewonnen wird und keinerlei Implementationsentscheidungen enthält. So gehen beispielsweise [BaGo79] von dieser Sichtweise aus und erarbeiten Prinzipien für als "gut" anzusehende Spezifikationen, aus denen in einem weiteren Schritt eine Reihe von

[1] Diese Aussage läßt allerdings außer Betracht, daß Spezifizierer und Anwender i.a. unterschiedliche Personen sind. Die vom Spezifizierer erstellte Beschreibung muß daher vom Anwender bestätigt sein (eine Form der *Validierung*).

Implikationen für die Gestalt als geeignet anzusehender Spezifikations*sprachen* abgeleitet werden.
Im folgenden sollen *zwei verbreitete Ansätze zur Formulierung von Systembeschreibungen* diskutiert werden:

- der *modellorientierte* Ansatz
- der *algebraische* Ansatz

Der Schwerpunkt liegt hierbei auf der Darstellung des modellorientierten Ansatzes, für den auch die Methodik der verifizierbaren Softwareentwicklung beschrieben wird. Bekannte modellorientierte Beschreibungssprachen wie Z und VDM[2] fallen in diese Klasse und werden kurz eingeordnet
Die Ausführungen dieses Abschnitts sind grundlegend für das allgemeine Verständnis der für die DAIDA-Umgebung zentralen modellbasierten Beschreibungssprache der *abstrakten Maschinen*. Die konkrete Gestalt dieser Sprache wird allerdings erst im folgenden Kapitel beschrieben, in dessen Rahmen auch die weiteren DAIDA-Sprachen vorgestellt werden und eine Diskussion der Rolle des B-Tools für die DAIDA-Umgebung erfolgt.

5.1 Modellorientierte Beschreibungen

Das auch der DAIDA-Entwurfsumgebung zugrundeliegende Prinzip der *modellorientierten Spezifikation* beinhaltet eine konzeptuelle Trennung der Beschreibung statischer und dynamischer Systemkomponenten: in der Spezifikationsphase ist ein *Modell der Daten* des zu entwickelnden Programms zu erstellen, welches dann durch Spezifikation einer Reihe von darauf "arbeitenden" Operationen ergänzt wird. In [Abri89] wird hierfür die Bezeichnung *Modellanimation* eingeführt. Die verwendete Spezifikationssprache hat für diese beiden Aufgaben die geeigneten Beschreibungskonzepte anzubieten.
Die folgende Darstellung orientiert sich an der Arbeit [Abri89], in der die grundlegenden Ideen der in der DAIDA-Umgebung verwendeten Notationsform der *abstrakten Maschinen* erarbeitet werden.

5.1.1 Datenmodellierung

Die zur Spezifikation der Systemstatik durchgeführte Datenmodellierung besteht aus folgenden Komponenten:

- Angabe der *Variablen* des spezifizierten Programms,

[2] VDM steht für *Vienna Development Method* ([Jone86]) und stellt genaugenommen mehr als nur eine *Beschreibungssprache* zur Verfügung. Die Verwendung des Begriffs "formale Methode" wird z.B. in Kap.12 von [WoLo88] kritisch beleuchtet: oft ist nur die Beschreibungssprache gemeint.

- Formulierung einer *Invarianten*, die die Eigenschaften einzelner Variablen sowie deren Beziehung untereinander festlegt,

- Zurverfügungstellen von *abstrakten Mengen*, die als gegeben vorausgesetzt werden und als *generische Parameter* in die Programmspezifikation eingehen[3]. Abstrakte Mengen werden insbes. in der Invariantenformulierung verwendet.

Zur Formulierung der Invariante stehen die *prädikatenlogischen Konnektive* sowie spezielle *mengentheoretische Prädikate* zur Verfügung:

	Syntax	Beschreibung
BASISPRÄDIKAT	"$\in$"	Mengenmitgliedschaft
LOGISCHE KONNEKTIVE	"$\Rightarrow$", "$\vee$", "$\wedge$", "$\neg$", "$\forall$", "$\exists$"	Implikation, Oder, Und, Negation, All- und Existenzquantor

Andere Beziehungen wie z.B. Mengeninklusion können hierin leicht formuliert werden.
Das *intendierte Universum* sind die gegebenen abstrakten Mengen sowie die durch bestimmte *Mengenkonstruktoren* hierüber definierbaren Mengen:

	Syntax	Beschreibung
BASISMENGEN	Mengenname	abstrakte Mengen (als generische Parameter)
MENGENKONSTRUKTOREN	$A \times B$ A, B Mengen	Kartesisches Produkt
	$\mathrm{P}(A)$ A Menge	Potenzmenge
	$\{x \mid P(x)\}$ P Prädikat	Def. per Mengen-"Komprehension"

Mit diesen Hilfsmitteln ist es möglich, bespielsweise Relationen und partielle sowie totale Funktionen als Mengen spezieller Gestalt zu definieren. Diese syntaktischen Konzepte lassen sich durch Angabe von *Axiomen* im Rahmen der Prädikatenlogik formalisieren (vgl. [Abri89a]). [Abri89] geht des weiteren davon aus, daß *induktiv definierbare Mengen* wie natürlichen Zahlen, Listen und Bäume konstruiert werden dürfen und primitiv rekursive Funktionen auf diesen Strukturen (beispielsweise Addition und Multiplikation natürlicher Zahlen) zur Verfügung stehen. Dies läßt sich durch Angabe zusätzlicher Axiome ebenfalls formal abbilden (vgl. [Abri89b])[4]; ein Fixpunktsatz von Tarski ([Tars55]) liefert die Rechtfertigung für Definitionen dieser Art. Deduktionssysteme, die zum Durchführen von Beweisen über solchen Spezifikationen eingesetzt werden, müssen daher in der Lage sein, auf Logiken *höherer Ordnung* zu arbeiten, sodaß insbesondere das *Induktionsaxiom* formulierbar ist.

[3] beispielsweise eine abstrakte Menge *MONEY*, deren genaue Gestalt (Währung, Betrag) uninteressant ist
[4] Insgesamt wird auf diese Art eine vereinfachte Form der *Zermelo-Frænkel-Mengentheorie* axiomatisiert.

5.1.2 Modellanimation

Im Rahmen der Modellanimation werden nun eine Menge von Operationen spezifiziert, die auf den in der Datenmodellierung aufgeführten Variablen arbeiten. Im Gegensatz zu einer *Realisierung* der Operation durch ein Programmsegment stehen auf dieser Ebene *deklarative* Beschreibungsmittel zur Verfügung, in denen die *gewünschten Auswirkungen* der Operationsausführung und *nicht das "Wie"* einer Implementierung erfaßt werden. Die Umsetzung einer Operationsspezifikation in ein lauffähiges Programm ist *nicht notwendig eindeutig*: im allgemeinen gibt es viele Programme, die die Spezifikation verwirklichen.
In [Abri89] wird zunächst von spezifischen Notationen für Operationsspezifikationen abstrahiert; es werden die grundlegenden Komponenten identifiziert, die die von der Operation auf den Variablen durchzuführenden Änderungen beschreiben:

- Die *Präkondition* legt fest, unter welchen Bedingungen die Ausführung der Operation als zulässig angesehen wird. Ihre Erfüllung hängt daher von den Werten der Variablen *vor* der Operationsausführung ab.
- Die *Inkondition* beschreibt das eigentliche Verhalten der Operation, d.h. sie legt Beziehungen zwischen den Variablenwerten vor *und* nach Operationsausführung fest.

Für die *Formulierung* solcherlei Operationsbeschreibungen unterscheidet [Abri89] *drei unterschiedliche Notationsformen*, die im folgenden beschrieben werden sollen:

- *prädikativer* Stil,
- *relationaler* Stil,
- *Substitutionsstil*.

5.1.2.1 Der prädikative Stil

Im prädikativen Stil werden Operationen vollständig in Form von *Prädikaten erster Ordnung* über den Programmvariablen beschrieben. Die Werte der Variablen nach Operationsausführung werden hierbei durch mit Hochkomma gekennzeichnete Variablennamen referenziert, d.h. es werden Prädikate der Form

$$In(x, x')$$
$$Pre(x)$$

formuliert, wobei x die Programmvariablen symbolisiere. Dieser Stil wird in [Hoar84] sowie ausführlich in [Hehn84b] und [HeGM86] beschrieben und liegt beispielsweise der bekannten *Vienna Development Method* ([Jone86]) zugrunde.
In [HeGM86] wird die *Operations*implementierung sogar vollständig in diesem Rahmen beschrieben. Zu diesem Zweck werden ausgezeichnete Prädikate angegeben, die das Verhalten der in gewöhnlichen Programmiersprachen vorhandenen Konstrukte beschreiben und demnach als

"berechenbar" bekannt sind. Die Operationsimplementierung besteht in einer Überführung des Spezifikationsprädikats in ein dieses Prädikat implizierendes (stärkeres) "*berechenbares*" Prädikat. Hierbei spielen Monotonieeigenschaften im Hinblick auf die schrittweise Entwicklung von Unter"programmen" eine wichtige Rolle.
[HeGM86] beschreiben allerdings lediglich *dynamische* Konzepte; Aspekte der System*statik* wie Invariantenformulierung und Variablendeklaration werden außer Betracht gelassen.

5.1.2.2 Der relationale Stil

Hier wird die Menge der möglichen Systemzustände M als kartesisches Produkt der Variablen-Wertemengen aufgefaßt. Die Präkondition wird als Teilmenge $Pre \subseteq M$, die Inkondition als Teilmenge $In \subseteq M \times M$ aufgefaßt.

5.1.2.3 Der Substitutionsstil

Diese Notation geht u.a. auf C. Hoare und E.W. Dijkstra zurück (vgl. z.B. [Dijk76]). Den Instruktionen einer (verallgemeinerten) Programmiersprache werden sog. *Prädikattransformer* zugeordnet: eine durch ein Prädikat erster Ordnung gegebene *Postkondition* P wird hierdurch auf die *schwächste Präkondition* abgebildet, unter der P erstellt wird. Die Prädikattransformer werden in Form von sog. *Substitutionen* beschrieben. Ist S eine Substitution, so steht

$$[S]P$$

für die schwächste Präkondition, unter der die Postkondition P erstellt wird[5].
In dem in DAIDA verfolgten Ansatz wird der Substitutionsstil verwendet. Abrial führt für die zur Verfügung stehenden Konstrukte die Bezeichnung "*Generalized Substitutions*" ein. Bei der Beschreibung der abstrakten Maschinen wird eine Übersicht über Syntax und Semantik dieser *Generalized Substitution Language* gegeben.

5.1.3 Initialisierung

Durch die Formulierung einer ausgezeichneten Operation, der sog. *Initialisierung*, wird die Spezifikation des Programms vollständig. Die Initialisierung wird wie eine Operation spezifiziert und zu Beginn einmalig ausgeführt, um die Zustandsvariablen zu initialisieren. Es bestehen die Einschränkungen (in diesem Abschnitt wird jeweils von einer *prädikativen* Operationsbeschreibung ausgegangen)

$$\begin{aligned} Pre_{ini}(x) &= TRUE \\ \forall x, y \bullet In_{ini}(x, x') &= In_{ini}(y, x') \end{aligned}$$

D.h. die Initialisierung ist unabhängig vom "Startzustand".

[5] Für die *Spezifikation* stehen jedoch nur eine Teilmenge der Substitutionskonstrukte zur Verfügung: Programmierkonzepte wie Sequenzierung und Iteration fehlen.

5.1.4 Modellkonsistenz

Mit der Spezifikation von Daten und zugehöriger Animation liegt nun eine mathematisch formale Modellierung des zu entwickelnden Programms vor. Die im Modell zulässigen Zustandsübergänge sollten mit den erwünschten Eigenschaften korrespondieren; eine formale Validierung ist jedoch, wie bereits zuvor begründet, unmöglich.
Andererseits sollten an die spezifizierten Operationen bestimmte Anforderungen gestellt werden, die aufgrund der formalen Semantikdefinition der Substitutionsnotation sehr wohl einer Introspektion unterliegen:

- Nach Ausführung der Initialisierung muß die Invariante erfüllt sein.
- Wird eine Operation unter der Voraussetzung der Invariante ausgeführt, so gilt die Invariante auch danach.

Gilt die erste Bedingung, so wurde gleichzeitig gezeigt, daß die formulierte Invariante I nicht widersprüchlich ist, d.h. die Invariante hat (im formallogischen Sinn) ein *Modell*. Formalisiert lesen sich die beiden Bedingungen wie folgt:

$$In_{ini}(x, x') \Rightarrow I(x') \tag{5.1}$$

$$I(x) \wedge Pre_{op}(x) \Rightarrow \forall x' \bullet (In_{op}(x, x') \Rightarrow I(x')) \tag{5.2}$$

für alle Operationen *op* ([Abri89]).
Im Rahmen der Spezifikation durch abstrakte Maschinen werden diese Bedingungen als sog. *Beweisverpflichtungen* erfaßt. Beziehungen der Gestalt 5.2 werden in DAIDA nachgewiesen, um die *Sicherheit* spezifizierter Transaktionen (vgl. Abschnitt 1.3) zu beweisen. In DAIDA wird allerdings der Substitutionsstil verwendet, sodaß Beziehung 5.2 (und alle übrigen Beweisverpflichtungen) in einer abgewandelten Form dargestellt werden (vgl. Kapitel 6).

5.1.5 Strukturierung umfangreicher Spezifikationen

Zur Spezifikation umfangreicherer Programme sind zusätzliche *Strukturierungskonzepte* wünschenswert. Abrial geht diese Zielsetzung aus der Sicht der zu erfüllenden Konsistenzbedingungen an und untersucht, in welchen Fällen sich eine Gesamtbeschreibung aus mehreren Teilbeschreibungen zusammensetzen läßt, ohne zusätzliche Konsistenzbedingungen verifizieren zu müssen. Dazu wird der Fall betrachtet, daß sich die Konsistenzbedingung einer Operation

$$I(z) \wedge Pre(z) \Rightarrow \forall v \bullet (In(z, v) \Rightarrow I(v))$$

in der Form

$$\begin{array}{l} I1(x) \wedge I2(y) \wedge Pre1(x) \wedge Pre2(y) \\ \Rightarrow \\ \forall x', y' \bullet (In1(x, x') \wedge In2(y, y') \Rightarrow I1(x') \wedge I2(y')) \end{array} \tag{5.3}$$

schreiben läßt , wobei x und y eine disjunkte Zerlegung von z sind.
Sind nun zwei animierte Modelle gegeben, die auf disjunkten Variablenmengen definiert sind und für die die Gültigkeit der individuellen Konsistenzbedingungen

$$I1(x) \wedge Pre1(x) \quad \Rightarrow \quad \forall x' \bullet (In1(x, x') \Rightarrow I1(x')) \tag{5.4}$$
$$I2(y) \wedge Pre2(y) \quad \Rightarrow \quad \forall y' \bullet (In2(y, y') \Rightarrow I2(x')) \tag{5.5}$$

einer Operation bekannt ist, so läßt sich unmittelbar zeigen[6], daß die Gültigkeit der individuellen Bedingungen 5.4 und 5.5 die Gültigkeit der Gesamtbedingung 5.3 impliziert. Unter diesen Voraussetzungen ist es somit möglich, die beiden animierten Modelle durch Vereinigung ihrer Variablenmengen, Konjunktion ihrer Invarianten und durch wahlweise

- *direkte* Übernahme der Operationen,
- Übernahme der Operationen und *Verstärkung der Präkonditionen*,
- *Kombination* der Operationen durch *Konjunktion der Prä- und Inkonditionen*.

zu einem zusammengesetzten animierten Modell zu kombinieren und die Konsistenz der betrachteten Operation aus den individuellen Konsistenzen zu folgern.
Abrial erwähnt darüberhinaus die Möglichkeit, die auf diese Weise erhaltene *Spezifikationskombination* um zusätzliche Variablen, Invariantenprädikate und Operationen zu ergänzen. Die ergänzten Invariantenprädikate können sich hierbei auf die "lokalen" Variablen der Subspezifikationen beziehen[7]. Die zusätzlich definierten Operationen haben auf die Variablen der kombinierten Modelle keinen direkten *Schreib*zugriff; sie müssen sich zu diesem Zweck der Operationen der kombinierten Spezifikationen bedienen ("*hiding principle*"). In diesem Fall sind dann lediglich die Konsistenzbedingungen der zusätzlich definierten Operationen sowie ggf. ([Abri89] läßt diesen Punkt außer Acht) die Konsistenz der aus den kombinierten Spezifikationen *übernommenen* Operationen bezüglich der *ergänzten* Invariantenprädikate nachzuweisen.
Bei der Umsetzung dieser Konzepte durch die in der DAIDA-Entwurfsumgebung zum Einsatz kommenden *abstrakten Maschinen* werden die genau zu erbringenden Beweisverpflichtungen diskutiert.
Da die neu definierten Operationen unmittelbaren *Lese*zugriff auf die Variablen der kombinierten Modelle haben, spricht Abrial von "laxem Hiding".

5.1.6 Implementierungskonzepte: Module

Zum Entwickeln eines guten Verständnisses der Problematik der Programmentwicklung erarbeitet Abrial die prinzipiellen Unterschiede zwischen Spezifikations- und Implementierungsbeschreibungen heraus. Zu diesem Zweck orientiert er sich an den in den üblichen imperativen Programmiersprachen[8] vorhandenen *Definitions- und Instruktionskonzepten*.
In *Variablendefinitionen* ordnet man den Variablen einen bestimmten *Typ* zu. Als zulässige Typen stehen *Basistypen* sowie die daraus unter Verwendung der zulässigen *Typkonstruktoren* konstruierbaren strukturierten Typen zur Verfügung:

[6] durch zweimaliges Anwenden der Beziehung $((A \Rightarrow B) \wedge (A \Rightarrow C)) \models (A \Rightarrow (B \wedge C))$
[7] "*add some 'gluing' invariant*", [Abri89]
[8] vgl. insbes. MODULA

BASISTYPEN	INTEGER, REAL
TYPKON-STRUKTOREN	ARRAY
	Unterbereichstyp von INTEGER
	FILE

Die zur Verfügung stehenden Instruktionen zur Operationsformulierung sind

BASIS-INSTRUKTIONEN	"leere" Instruktion
	Zuweisungen (x:= E)
	Schreiben und Lesen von Dateien (write(f,x), read(f,x))
KONSTRUKTOREN	Sequenzierung (";")
	Konditional (IF b THEN S ELSE Q)
	Iteration (WHILE b DO S)

Zuweisungen haben *typkompatibel* zu erfolgen; zu diesem Zweck werden *typisierte Ausdrücke* definiert. Zur Formulierung von Bedingungen bieten Programmiersprachen darüberhinaus boolesche (aussagenlogische) Ausdrücke an:

BASISPRÄDIKAT	"=" auf typkompatiblen Variablen.
LOGISCHE KONNEKTIVE	"$\vee$", "$\wedge$", "$\neg$"

Die implementierte Einheit wird als *Modul* bezeichnet, die über einen Initialisierungsteil verfügt und mittels einer Schnittstellenvereinbarung bestimmte Operationen anbietet. Einzelne Operationen werden als *Prozeduren* oder *Funktionen* implementiert. Darüberhinaus können zusätzliche *Strukturierungsmöglichkeiten* bestehen, wie beispielsweise in MODULA-2, wo es möglich ist, ganze *Modulhierarchien* zu formulieren[9].

5.1.7 Vergleich: Modelle und Module

Aus der Diskussion der unterschiedlichen Konzepte von (animierten) Modellen und implementierenden Modulen lassen sich folgende Unterschiede ableiten:

- *animierte Modelle* verfügen zur Formulierung der statischen Systemeigenschaften über die volle *Prädikatenlogik erster Ordnung* sowie über die in diesem Rahmen axiomatisierbare

[9] In diesem Fall gilt jedoch im Unterschied zur Spezifikation das *strikte* "Hiding"-Prinzip, es sei denn, ein Modul stellt seine Variablen in der Schnittstellenvereinbarung explizit zur Verfügung (vgl. [Wirt88])

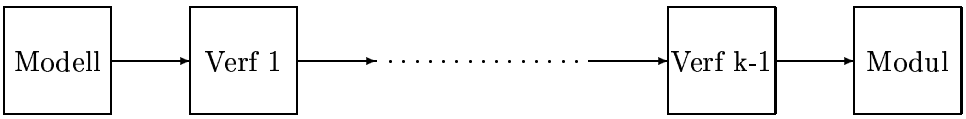
Mengentheorie und über *abstrakte Mengen*. *Dynamische Beziehungen* zwischen den Variablen sind nicht *formulierbar*, sondern müssen von der Implementation der Operationen garantiert werden.
Zur Beschreibung der System*dynamik* können die Modellanimationen *nichtdeterministisch* sein. Im Rahmen der verallgemeinerten Substitutionsnotation (s.u.) sind so beschränkter und unbeschränkter Nichtdeterminismus beschreibbar; darüberhinaus gibt es für Operationen mit Präkondition kein vergleichbares Implementierungskonzept.

- *Module* verfügen im Vergleich zu Modellen zusätzlich über *Sequenzierungs- und Iterationskonzepte* zur Operationsbeschreibung. Zur Deklaration der Variablen stehen lediglich das vorgegebene "statische" Typkonzept sowie die festvorgegebenen Basistypen zur Verfügung[10].

Gemeinsam sind beiden Konzepten das Vorhandensein von *Variablendefinitionen* (in Form von Deklarationen oder Invarianten) sowie die *Initialisierung*.

5.1.8 Vom Modell zum Modul: abstrakte Maschinen

Um einen schrittweisen Übergang vom spezifizierenden (animierten) Modell zum implementierenden Modul in einer einheitlichen Beschreibungssprache zu ermöglichen, definiert Abrial in [Abri89] das Konzept der *abstrakten Maschinen*[11]: abstrakte Maschinen enthalten alle Beschreibungsmittel animierter Modelle sowie zusätzlich die modulspezifischen Konstrukte der Sequenzierung und Iteration.
Der schrittweise Weg von der Spezifikation hin zur Implementierung kann daher vollständig in der Notation der abstrakten Maschinen geschehen. Der Implementierungsprozeß wird daher von einer Folge von abstrakten Maschinen beschrieben, an deren Anfang das Modell und an deren Ende das Modul steht. Die Zwischenstufen bilden sog. *Verfeinerungsmaschinen*.



Zwischen aufeinanderfolgenden abstrakten Maschinen dieser Folge besteht eine sog. *Verfeinerungsbeziehung*.
Abrial unterscheidet zwei unterschiedliche Arten von Verfeinerungsschritten: *Datenverfeinerung* und *Operationsverfeinerung*[12]. Diese sollen in den folgenden Abschnitten beschrieben werden.

[10] was eine Einschränkung im Vergleich zur vollen Mengentheorie bedeutet

[11] Über die genaue Gestalt der Beschreibungsform macht Abrial hier keine Aussagen. Z.B. wird nicht gesagt, in welcher Form die Operationsbeschreibungen erfolgen. Diese "abstrakte" Form des Begriffs der abstrakten Maschine wird im nächsten Kapitel durch die "konkrete" Definition der in der DAIDA-Entwurfsumgebung eingesetzten abstrakten Maschine ergänzt.

[12] engl. *Data Refinement* and *Operational Decomposition*

5.1.9 Datenverfeinerung

Im Rahmen einer Datenverfeinerung werden "neue", implementationsnähere Variablen eingeführt und deren Beziehung zu den bisherigen Variablen formuliert. Die Operationen arbeiten auf den neu eingeführten Variablen und werden demgemäß modifiziert: im einzelnen beinhaltet die verfeinerte Maschine

- eine Liste "neuer" Variablen y,
- ein Prädikat $J(x, y)$, welches deren Beziehung zu den bisherigen Variablen x beschreibt[13],
- für jede Operation S_i auf x eine Operation T_i auf y mit vergleichbarem Verhalten auf den "neuen" Variablen.

Um den Verfeinerungsschritt *formal zu rechtfertigen*, müssen, vergleichbar mit den obigen Konsistenzbedingungen, sog. *Beweisverpflichtungen* erbracht werden. Seien $Pre_S(x)$, $In_S(x)$ und $Pre_T(y)$, $In_T(y)$ die (prädikativen) Beschreibungen der Prä- und Inkonditionen der sich entsprechenden Operationen S bzw. T. Dann lautet die Beweisverpflichtung[14]

$$\begin{aligned} \forall x, y \bullet \quad & I(x) \wedge J(x,y) \wedge Pre_S(x) \\ \Rightarrow \quad & \\ & (Pre_T(y) \wedge (\forall y' \bullet (In_T(y,y') \Rightarrow \exists x' \bullet (In_S(x,x') \wedge J(x',y'))))) \end{aligned} \tag{5.6}$$

Diese Bedingung sagt aus: Wann immer

- sich die "abstrakte" und "konkrete" Beschreibung in entsprechenden Zuständen befinden ($J(x, y)$),
- die Invariante der abstrakten Beschreibung gilt ($I(x)$)[15],
- die abstrakte Operation ausführbar, d.h. Pre_S erfüllt ist,

so gilt:

- auch die Präkondition der konkreten Operation ist erfüllt,
- für alle durch die Inkondition der konkreten Operation zugelassenen Nachfolgezustände y' existiert auch ein "entsprechender" ($J(x', y')$) abstrakter Nachfolgezustand x', der die Inkondition der abstrakten Operation erfüllt ($In_S(x, x')$).

Diese Definition läßt die Möglichkeit offen, daß

[13] Dies entspricht in gewisser Hinsicht einer *Invariante der Verfeinerung*, wie die folgende Diskussion der zu erbringenden *Beweisverpflichtungen* verdeutlichen wird.

[14] In [Abri89] findet sich lediglich die Formulierung in der Substitutionsnotation. Die hier gegebene Darstellung ist eine äquivalente Formulierung im *prädikativen Stil*.

[15] Die konkrete Beschreibung "erbt" die Invariante der abstrakten Beschreibung: die Beziehung wird über $J(x, y)$ hergestellt (vgl. obige Bemerkung).

- die konkrete Operation in Fällen anwendbar ist, in der die abstrakte Operation nicht anwendbar ist (Pre_T kann erfüllt sein, wenn Pre_S nicht erfüllt ist)[16].
- die konkrete Operation "deterministischer" ist als die abstrakte Operation (Rolle des Existenzquantors)

Im Rahmen der Beschreibung der in der DAIDA-Entwurfsumgebung verwendeten abstrakten Maschinen wird eine alternative Formulierung gegeben, die sich an der Verwendung der Prädikattransformernotation orientiert. Die obige Formulierung ist allerdings intuitiver, da sie die Verwendung disjunkter "abstrakter" und "konkreter" Variablenmengen betont.

5.1.10 Operationsverfeinerung - Strukturierung von Verfeinerungen

In einer reinen Operationsverfeinerung bleiben die Variablen der Maschine unverändert. Unter Ausnutzung von *Monotoniebeziehungen* [17] werden *Teile von Operationen* in externe abstrakte Maschinen ausgelagert und dort unabhängig von der aktuellen Maschine implementiert. Die vorliegende verfeinerte Maschine greift *sowohl lesend als auch schreibend lediglich über die Operationen der externen abstrakten Maschinen* auf deren Variablen zu. Dies wird dadurch begründet, daß die externen Maschinen selbst noch einem Verfeinerungsprozeß unterworfen werden und deren Variablen noch Datenverfeinerungen unterliegen. Demnach gilt in diesem Fall das *strenge* "Hiding"-Prinzip.

5.1.11 Vergleich der Strukturierungskonzepte

Während bei der Strukturierung von Spezifikationen "Bottom-up" vorgegangen wurde (d.h. die Gesamtspezifikation wurde durch Komposition ihrer Teile gewonnen), gewinnt man die Modulstruktur der Implementation in einem "Top-down"-Prozeß, bei der die Dekompositionen rekursiv wie das Ausgangsmodell behandelt und verfeinert werden. Dies resultiert in unterschiedlich strengen "Hiding"-Konzepten.
Die sich ergebenden Strukturierungen der Spezifikation und der Implementation sind nicht zwingend vergleichbar: es handelt sich hier um prinzipiell unterschiedliche Ansätze. Die Spezifikation hat zunächst zum Ziel, ein möglichst detailliertes Abbild des zu realisierenden Systems zu erstellen. Die sich eventuell ergebende hierarchische Strukturierung weist Merkmale von *Objektorientierung* auf: Die in den Subspezifikationen beschriebenen Operationen und Invarianten arbeiten auf *disjunkten* Variablenmengen; im Rahmen der Spezifikationskombinierung können die individuellen Subspezifikationen um spezifikationsübergreifende Konzepte ergänzt werden. Bei der Top-Down-Verfeinerung hingegen soll eine wohlstrukturierte Implementation mit den Vorteilen einer guten Lesbarkeit und Adaptierbarkeit erzielt werden.

[16] Hier wird stillschweigend vorausgesetzt, daß die Anwendbarkeit einer Operation bei erfüllter Präkondition gewährleistet ist. Der Begriff der *Anwendbarkeit* wird in den Beweisverpflichtungen der in DAIDA verwendeten abstrakten Maschinen konkretisiert (sog. *Termination*).

[17] Unter *Monotonie* (des Instruktionssatzes) versteht man in diesem Fall folgendes: Ist $Op(X)$ eine Operationsbeschreibung, die eine Operation X referenziert ("aufruft") und wird X durch ein Y verfeinert ($X \sqsubseteq Y$), so gilt auch $Op(X) \sqsubseteq Op(Y)$.

Es bleibt zu untersuchen, inwieweit die Struktur der gewonnenen Spezifikation auch Aufschluß über eine mögliche Zerlegung der Implementation geben kann.

5.1.12 Z und VDM

Die Beschreibungsmethoden Z und VDM fallen in die Klasse der modellorientierten Beschreibungen: Spezifikationen sind unterteilt in eine Beschreibung des Systemzustands und eine Beschreibung von Operationen, die die Schnittstelle zur Manipulation der Zustandsvariablen definieren. In beiden Sprachen erhalten Zustandsvariable *Typvereinbarungen*[18], die sich als Bestandteil der Invariante auffassen lassen: beispielsweise können Variablen als Funktionen deklariert sein, wobei Restriktionen wie Injektivität, Surjektivität usw. ebenfalls formulierbar sind. Die Spezifikation der Systemstatik geschieht durch Vereinbarung typisierter Zustandsvariablen und durch Angabe zusätzlicher Invariantenprädikate. Zu den weiteren Gemeinsamkeiten gehören die Verwendung des *prädikativen Stils* zur Operationsbeschreibung.
Darüberhinaus gibt es jedoch schon zwischen diesen verwandten Methoden wesentliche Unterschiede:

- Z unterteilt Beschreibungen in *eigenständige* Komponenten, sog. *Schemas*, die sich unter Verwendung von *Schemakombinationsoperatoren* kombinieren lassen. Diese Operatoren lassen sich als "Fortsetzung" der logischen Konnektive "$\wedge$", "$\vee$", "$\Rightarrow$" usw. auf Schemas auffassen. Sie bewirken die entsprechende logische Verknüpfung der Invariante unter der Voraussetzung, daß die *Signaturen* (d.h. die definierten Variablen) kompatibel in dem Sinne sind, daß Variablen gleichen Namens vom gleichen "Typ" sind. Weitere Operatoren[19] erlauben das "Verstecken" von Variablen der Signatur eines Schemas bzw. eine "eingeschränkte" Komposition zweier Schemas ([Spiv89]). VDM bietet keine derartigen Strukturierungsmöglichkeiten.

- VDM unterscheidet in der Operationsspezifikation zwischen Prä- und Inkondition[20], was laut [Spiv89] die Generierung der Beweisverpflichtungen im Vergleich zu Z erleichtert. Darüberhinaus enthalten Operationsdefinitionen eine explizite Angabe der referenzierten Zustandsvariablen sowie die Art des Zugriffs (`read only` oder `read and write`), was es erlaubt, auf die Formulierung der in Z zu explizierenden "trivialen" Invariantenkonjunkte der Form $var' = var$ zu verzichten.

- In Z können "redundante" Zustandsvariablen durch Angabe einer entsprechenden Gleichheitsbeziehung zu einer anderen Zustandsvariablen eingeführt werden. Solcherlei Gleichheitsprädikate unterscheiden sich von anderen Invariantenprädikaten dadurch, daß ihr Erhalt garantiert ist und nicht durch die Operationen gewährleistet werden muß. Dies erleichtert die Operationsspezifikation; semantisch problematisch ist jedoch, daß diese *symmetrischen* Gleichheitsfestlegungen offen lassen, welches Objekt als "abhängig" anzusehen

[18] hier (im erweiterten Sinne) aufzufassen als mathematische Typvereinbarung im Sinne einer mengentheoretischen Elementbeziehung

[19] sog. "hiding operators"

[20] In VDM werden die hier als *Inkonditionen* eingeführten Prädikate zwischen den Werten der Zustandsvariablen vor und nach Operationsausführung etwas unsauber als *Post*konditionen bezeichnet.

ist[21]. In VDM gbt es derlei Definitionsmöglichkeiten nicht: Operationen müssen *alle* Invariantenprädikate respektieren[22].

Z gestattet darüberhinaus *generische* Spezifikationen, beispielsweise von abstrakten Datentypen wie Listen, "Queues" oder "Stacks" ohne Angabe der Elementmengen. Es wird unterschieden zwischen der Definition *generischer Schemas* und der Definition *generischer Konstanten* (z.B. Projektionsfunktionen auf kartesischen Produkten über beliebigen unterliegenden Mengen).

5.2 Algebraische Spezifikation

Es soll nun überblicksweise ein weiterer Ansatz zur Formulierung von Systembeschreibungen besprochen werden, der anderen bekannten Spezifikationsmethoden zugrundeliegt: die sog. *algebraischen Beschreibungen*.
Die statische Komponente einer algebraischen Spezifikation besteht aus einer sog. *Sortenvereinbarung* sowie einer *Signatur*, d.h. einer Menge von *Funktionen* und Konstanten[23] über den definierten Sorten. Die Spezifikation der Dynamik, d.h. der Semantik der definierten Funktionen, geschieht durch Angabe einer Menge von *Gleichungen*, den Axiomen der Beschreibung, die die operationalen Beziehungen der unterschiedlichen Funktionen zueinander festlegen[24]. Derartige Beschreibungen werden auch als *abstrakte Datentypen* bezeichnet.
Im Unterschied zum oben beschriebenen modellorientierten Ansatz werden Systemzustände *nicht* in Form der Variablenbelegungen, eingeschränkt durch Invariantenprädikate und Typvereinbarungen, modelliert. Die Spezifikation geschieht vielmehr durch die *Angabe der die Eigenschaften der Funktionen axiomatisierenden Gleichungen*[25]. Bei der Spezifikation von Systemen mit "persistenten Objekten" wie beispielsweise Datenbanken entsprechen die zulässigen Systemzustände den durch wiederholte "Anwendung" der Funktionen gemäß der durch die Gleichungen gegebenen Termersetzungsregeln erzeugbaren Termen (vgl. Beispiel in [Spiv88]).

5.2.1 Eine Beispielspezifikation in CLEAR

Ein abstrakter Datentyp natural wird in der algebraischen Spezifikationssprache CLEAR (vgl. [BuGo81]) wie folgt vereinbart:

```
const natural =
 theory
   sorts NAT
   opns
     zero : → NAT
```

[21] Oder man müßte symmetrische Gleichheitsbeziehungen "=" von asymmetrischen Definitionsbeziehungen ":=" konzeptuell unterscheiden, was in Z nicht vorgesehen ist.
[22] In [Spiv88] heißt es hierzu: "The style of VDM discourages redundant state components ...".
[23] als Spezialfall: eine Funktion der Stelligkeit 0
[24] Man spricht auch von der *axiomatisierten (Gleichheits-)Theorie*.
[25] In der englischen Literatur werden algebraische Spezifikationen auch als *property oriented* bezeichnet.

```
      succ : NAT → NAT
      add : NAT × NAT → NAT
    eqns
      add(zero,x) = x
      add(succ(x),y) = add(x,succ(y))
  endth
```

In diesem Fall wird nur eine Sorte NAT sowie die drei Funktionen zero, succ und add vereinbart. zero ist eine Konstante. Durch diese Signaturvereinbarungen werden die als syntaktisch wohlgeformt betrachteten *Terme* festgelegt. Die angegebenen Gleichungen legen fest, welche Terme als *gleich* anzusehen sind und erfassen auf diese Weise die Semantik der (als Nachfolger- bzw. Additionsfunktion der Arithmetik aufzufassenden) Funktionen succ und add.
Im Vergleich zu den in den modellorientierten Methoden formulierbaren strukturierten Typen durch mengentheoretische Ausdrücke steht ein sehr restriktives Instrumentarium zur Verfügung: in den statischen Funktionsvereinbarungen können lediglich *totale* Funktionen über den *strukturlosen*, in den Sortenvereinbarungen durch Namensangabe eingeführten Sorten definiert werden. Neben diesen elementaren Axiomatisierung bietet CLEAR die Möglichkeit zum Erstellen *modularer* Spezifikationen durch zusätzliche *Strukturierungskonzepte*. Unter Angabe einer `enrich`-Klausel können bereits bestehende Theorien herangezogen werden: durch

```
const multipl =
 enrich natural by
   sorts
   opns
     mult : NAT × NAT → NAT
   eqns
     mult(zero,x) = zero
     mult(succ(x),y) = add(y,mult(x,y))
 enden
```

wird die bestehende Spezifikation erweitert um eine (entsprechend axiomatisierte) Multiplikationsfunktion mult. Auch *generische Spezifikationen* sind in Form sog. *Theorieprozeduren* möglich (vgl. [BuGo81]):

```
proc generic_theory( Par1 : Ident, Par2: Triv,..)
```

vereinbart eine Theorie in Abhängigkeit von als Parameter gegebenen Theorien. Durch den Parametertyp Ident wird vereinbart, daß die für Par1 zu substituierende Theorie eine ("Objekt-")Gleichheitsbeziehung haben muß. Triv verlangt lediglich die Existenz einer Sorte `element` in der substituierten "aktuellen" Theorie. Parametertypen lassen sich als *Metasorten* auffassen.
Trotz des unterschiedlichen Ansatzes weisen algebraische Spezifikationssprachen somit, ebenso wie viele modellorientierte Sprachen, Sprachprimitiva

- zur *Modularisierung* von Spezifikationen,
- für *generische* Spezifikationen

auf.

5.2.2 Die formale Semantik von algebraischen Spezifikationen

Die Vereinbarung von Sorten, Signaturen über Sorten und Gleichheitsaxiomen ermöglicht eine Erstellung eines *syntaktischen Abbilds* des zu beschreibenden Systems. Zur Verifikation der erstellten algebraischen Spezifikation ist nachzuweisen, daß das erstellte Abbild mit der durch das zu beschreibende System induzierten *intendierten Interpretation*[26] verträglich ist. Zu diesem Zweck werden die gemäß der axiomatisierten Theorie erzeugbaren Terme interpretiert in der Form, daß

1. die definierten Sorten mit "intendierten Mengen" identifiziert werden.
2. den in der Signatur definierten Funktionen über Sorten (signaturverträglich) Funktionen über den intendierten Mengen zugeordnet werden.

Formal läßt sich dies in Form eines Paares (ϕ, ψ) von *Funktionen* ausdrücken, wobei ϕ jeder Sorte eine *Menge* zuordnet und ψ die Interpretation der spezifizierten Funktionen realisiert.
Sind die Gleichheitsaxiome erfüllt, so bezeichnet man (wie in der mathematischen Logik) die intendierte Interpretation als ein *Modell* der Spezifikation.
In der oben axiomatisierten Theorie natural ist die intendierte Interpretation der *Sorte* NAT gerade die Menge der natürlichen Zahlen, add entspricht der Addition "+", succ der Nachfolgerfunktion. Demnach entsteht die Zuordnung

$$\{zero \rightarrow 0,\ s(zero) \rightarrow 1,\ s(s(zero)) \rightarrow 2,\ \ldots$$

Um die Verträglichkeit mit den Axiomen nachzuweisen, sind die Bedingungen

$$\begin{gathered}\psi(\mathsf{add(zero,x)}) = \psi(\mathsf{x}) \\ \Rightarrow \\ 0 + \psi(\mathsf{x}) = \psi(\mathsf{x})\end{gathered}$$

und

$$\begin{gathered}\psi(\mathsf{add(succ(x),y)}) = \psi(\mathsf{add(x,succ(y))}) \\ \Rightarrow \\ 1 + \psi(\mathsf{x}) + \psi(\mathsf{y}) = \psi(\mathsf{x}) + \psi(\mathsf{y}) + 1\end{gathered}$$

[26] Dieser Begriff hat hier eine andere Bedeutung als im formallogischen Kontext von Kapitel 3.

zu verifizieren, was in diesem Fall trivial ist.
Neben derartigen "vernünftigen" Interpretationen können auch *Trivialinterpretationen* Modelleigenschaft haben; beispielsweise ergibt die Zuordnung eines festen Elements an *jeden* der erzeugbaren Terme ebenfalls ein Modell; da die Gleichungsaxiome nicht festlegen, welche Terme sich *unterscheiden* müssen[27], ist eine derartige Zuordnung stets möglich. Dieses Phänomen wird in [WoLo88] als *confusion* bezeichnet; ebenfalls problematisch ist das Vorhandensein von Mengenelementen, die keinem erzeugbaren Term zugeordnet werden (sog. *junk*). Beide Fälle werden von der Betrachtung ausgeschlossen. Dies führt zu dem Begriff der als Semantik der algebraischen Spezifikation angesehenen *initialen Modelle*, in denen diese Fälle von der Betrachtung ausgeschlossen sind.
Ein *ausgezeichnetes Modell* einer algebraischen Spezifikation ist die sog. *Termalgebra*, bei der jedem (strukturierten) Term eine ihm entsprechende (unstrukturierte) Zeichenkette zugeordnet wird. Die Gleichheitsaxiome lassen sich hier als einfache Substitutionsregeln auf Zeichenketten auffassen. In diesem Rahmen lassen sich die *initialen Modelle* präzise definieren als die *Klasse aller zur Termalgebra isomorphen Modelle* ([WoLo88]).

5.2.3 Implementierung

In obigem Beispiel der Spezifikation der Theorie natural ist die Übereinstimmung mit dem beschriebenen System unmittelbar. Verifiziert wurde in diesem Fall die Übereinstimmung mit dem "Original" (Validierung). Eine andere Frage stellt sich bei der Suche nach der *Implementierung* eines spezifizierten komplexen Systems. Hier geht es darum, eine Umsetzung der Spezifikation in eine ausführbare Beschreibung vorzunehmen, wobei das Ergebnis dieses hier als "Design" bezeichneten Vorgehens ([WoLo88]) mit der Spezifikation verträglich sein muß. Dies wird in [WoLo88] im Rahmen der im letzten Abschnitt eingeführten Begriffe eingeordnet:

> *It is the ability to find a variety of models for such theories that forms the basis of computational systems. Specification can be thought of as producing a theory which describes the artefact to be built, and design can be thought of as finding the most appropriate model of the theory.*

D.h. die Implementierung wird als ein im Hinblick auf ihre Ausführbarkeit "am meisten geeignetes" Modell der Spezifikation aufgefaßt.

[27] Sozusagen "fehlt" der Gleichheitstheorie eine *Closed World assumption*.

Kapitel 6

Das B-Tool im Rahmen der DAIDA-Entwicklungsumgebung

In den vorangegangenen Abschnitten wurden die konzeptuellen Grundlagen des B-Tools und dessen Realisierung unabhängig von einer konkreten Anwendung beschrieben. An dieser Stelle soll nun der Einsatz des B-Tools im Rahmen des DAIDA-Projekts diskutiert werden. Dazu wird zunächst ein Überblick über die im DAIDA-Projekt verwendeten Entwurfs-, Entwicklungs- und Implementationssprachen gegeben. Aufbauend auf den im vorangegangen Abschnitt beschriebenen Grundlagen modellorientierter Beschreibungen nach Abrial wird das für DAIDA zentrale Beschreibungsmittel der *abstrakten Maschinen* (vgl. [Abri90a], [AGMS88a], [AGMS88b],[AGMS88c]) ausführlich diskutiert. Ausgehend von einer aus einer konzeptuellen Beschreibung automatisch gewonnenen abstrakten Maschine wird in einer Folge von Verfeinerungsschritten eine "zielsprachennahe" abstrakte Maschine konstruiert, aus der sich schließlich automatisch das Zielsprachenprogramm gewinnen läßt[1]. Zwischen je zwei aufeinanderfolgenden abstrakten Maschinen sollte hierbei eine bestimmte Beziehung bestehen, deren Gültigkeit zu verifizieren ist. Solche Beziehungen werden in der DAIDA-Umgebung als *Beweisverpflichtungen* bezeichnet. Die Beweisverpflichtungen verkörpern somit die *Semantik des Verfeinerungsprozesses*; sie werden in einem eigenen Abschnitt diskutiert. Dies liefert den theoretischen Unterbau für den Einsatz des B-Tools zum Beweis der Korrektheit der vom Benutzer vorgegebenen Verfeinerungsschritte. Für diesen Zweck wurde von DAIDA-Projektpartnern[2] eine umfangreiche *Theoriensammlung* `devB` für das B-Tool erstellt, die die zu erfüllenden Beweisverpflichtungen automatisch erzeugt und zu beweisen versucht. Aufbau und Funktion von `devB` werden auszugsweise beschrieben.
Der eigentliche *Beweis* der Beweisverpflichtungen wird bisher von `devB` nur unvollständig durchgeführt: es entstehen Teilziele bestimmter, zielsprachenabhängiger Gestalt, die unter Verwendung der von `devB` zur Verfügung gestellten Regelsammlungen nicht bewiesen werden können und als sog. *Lemmas* erfaßt werden[3]. Dies motiviert den Inhalt des nachfolgenden Kapitels, in

[1] Die Folge der abstrakten Maschinen der Verfeinerung wird hierbei vom Benutzer aufgestellt und ist daher zu verifizieren.

[2] Unter Federführung des Entwicklers des B-Tools und der abstrakten Maschinen, J.R. Abrial (Paris), sowie insbes. der Software Engineering Section des BP Research Centre, Sunbury on Thames, GB.

[3] Betrachtet man die Zielsprache als variierenden Parameter der Entwicklungsumgebung, so ist die Struktur der erzeugten, unreduzierbaren Teilziele *anwendungsabhängig*. `devB` realisiert jedoch nur allgemeine Aspekte der

dem eine zum Beweis der Lemmas geeignete Theoriensammlung entwickelt wird. Die Gestalt der Lemmas wird von der verwendeten Spezifikationssprache TDL sowie der Zielsprache DBPL beeinflußt, sodaß eine Diskussion dieser Sprachen angesagt erscheint.

6.1 Überblick über die verwendeten Beschreibungsmittel

In den folgenden Unterabschnitten werden, orientiert an [BMSW89], die in DAIDA verwendeten Beschreibungssprachen einzeln anhand von Beispielen vorgestellt. Die Zielsprache der DAIDA-Umgebung ist die Datenbankprogrammiersprache DBPL. Für die Spezifikaton von Datenbankanwendungen wurden in den letzten Jahren Sprachen zur sog. *semantischen Datenmodellierung* entwickelt, die auf eine benutzernahe, "*konzeptuelle*" Erfassung des zu spezifizierenden Systems abzielen. Zusammen mit neuen Ansätzen zur Wissensrepräsentation und Abstraktionskonzepten wie Generalisierung, Aggregation und Klassifizierung prägten diese Ansätze den Begriff der *konzeptuellen Modellierung*. In [BrMS84] werden frühe Arbeiten in diesem Gebiet zusammengestellt und diskutiert.
Die auf den unterschiedlichen Ebenen der DAIDA-Entwicklungsumgebung verwendeten Beschreibungssprachen und ihre grundlegenden Aufgaben sind in Abbildung 6.1 zusammengefaßt.

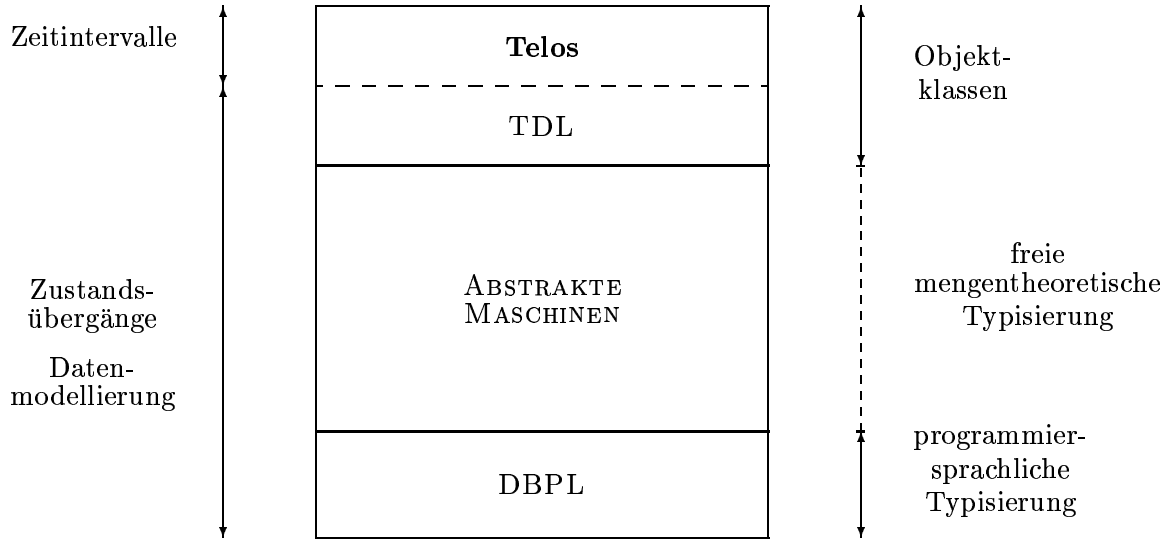


Abbildung 6.1: Beschreibungssprachen der DAIDA-Entwicklungsumgebung

Die Bezeichnungen auf der linken Seite der Abbildung beschreiben, in welcher Weise Umweltge-

Entwicklungsumgebung und überläßt den Beweis der anwendungsspezifischen Lemmas dem Benutzer.

gebenheiten in den unterschiedlichen Beschreibungssprachen konzeptualisiert werden: lediglich der Wissensrepräsentationssprache **Telos** verfügt über ein *Zeit*konzept (vgl. u.). Auf der rechten Seite der Abbildung wird herausgestellt, welche Beschreibungssprachen über Klassenkonzepte verfügen und auf welcher Ebene eine *mengentheoretische* ("freie") oder *programmiersprachliche* Typisierung stattfindet, in deren Rahmen nicht mehr zwischen Objekten und deren Attributen unterschieden wird.

6.1.1 Telos

Die Sprache **Telos** ([BKMS89]) dient der *Wissensrepräsentation* und damit der Aufbereitung der in der Anforderungsanalyse gewonnenen Information (vgl. Abschnitt 1.3). Sie weist folgende Eigenschaften auf:

- rein *deskriptive* Modellierung.
- *objektorientierte* Beschreibung von Entitäten und Aktivitäten der Anwendungsdomäne in entsprechenden Klassen, benutzerdefinierbare Klassen, Attributklassen. "*Is-A*"- und "*Instance Of*"-Beziehungen. Azyklische Klassenbeziehungen, Metaklassen.
- *zeitintervallorientierte* Beschreibung der Eigenschaften von Objekten der Domäne.
- Formulierbarkeit von Integritätsbedingungen der Anwendungsdomäne in einer speziellen prädikativen Sprache ("Assertion Language").

Da die Sprache **Telos** nicht in den Anwendungsbereich des B-Tools fällt, erfolgt keine genauere Beschreibung dieser Sprache.

6.1.2 TDL

Im nächsten Schritt wird das nunmehr formalisiert vorliegende Ergebnis der Anforderungsanalyse dazu herangezogen, eine schon etwas mehr an den Gegebenheiten der imperativen Zielsprache DBPL orientierte Beschreibung des zu entwerfenden Datenbankprogramms zu erstellen. Für diesen Zweck steht in DAIDA die Sprache TDL zur Verfügung, die sich gegenüber **Telos** in folgenden Punkten unterscheidet:

- Während **Telos** von der Art der Modellierung prinzipiell keine Unterscheidung zwischen Entitäten und Aktivitäten trifft (beide werden als *Objekte* bestimmter Klassen erfaßt), bietet TDL unterschiedliche Modellierungsmöglichkeiten für die statischen und dynamischen Systemkomponenten.
- Anstelle der "ontologischen" zeitintervallorientierten Konzeptualisierung tritt in TDL bereits die in imperativen Programmiersprachen vorherrschende *zustandsorientierte* Sicht. Zustandsübergänge erfolgen durch die Ausführung sog. *Transaktionen*.

Erhalten bleibt die Darstellung von Objekten in Klassen[4] sowie der deskriptive Charakter der Beschreibung: die Beschreibung der dynamischen Systemkomponenten ist weiterhin nichtprozedural. Da nunmehr die statischen und die dynamischen Komponenten getrennt erfasst werden, erhält man einen ersten Entwurf des zu realisierenden Softwaresystems. TDL wird daher als *Entwurfssprache* bezeichnet. Die folgenden Abschnitte gehen auf die von TDL unterstützten Beschreibungsmöglichkeiten für Systemstatik, Systemdynamik und Integritätsbedingungen näher ein.

6.1.2.1 Beschreibung der Systemstatik

Zur Beschreibung der statischen Systemkomponenten verwendet TDL den bereits aus **Telos** bekannten Ansatz unterschiedlicher Objekt*klassen*. Man unterscheidet mehrere Arten von *Datenklassendefinitionen*:

- *Basisklassen*: beispielsweise INTEGER, BOOLEAN, REAL, STRING. Klassen, deren Ausprägungen als bekannt und unveränderlich vorausgesetzt werden.
- *Aufzählungsklassen*: z.B.: agencies = {DFG,NSF,ESPRIT}
- *Aggregatklassen*: Klassen, deren Struktur als *kartesisches Produkt* weiterer Klassen definiert ist, wobei die einzelnen *Attribute* "benannt" sind[5]. Z.B.

```
AGGREGATE CLASS Money WITH
 amount : PositiveInteger;
 currency : {francs,ecu,dm,...};
END Money;
```

- *Entitätenklassen*: durch diese werden die Objekte der Anwendung erfaßt, die die möglichen Zustände des Systems festlegen. Den Ausprägungen der Entitätenklassen wird eine eigene, von der Existenz anderer Objekte unabhängige *Identität* zugebilligt[6]. Demnach spielen Entitätenklassen bei der Spezifikation von Datenbankanwendungen eine zentrale Rolle. Z.B.

```
ENTITY CLASS Companies WITH
 UNIQUE, UNCHANGING
  name : String;
 CHANGING
  engagedIn : SetOf Projects;
```

[4] [BMSW89] spechen hier von Objektorientiertheit; die Wahl dieser Bezeichnung hält einer kritischen Betrachtung nicht stand: die Operationsbeschreibungen in TDL müßten, entsprechend einem Grundmerkmal der Objektorientiertheit, *objektklassenspezifisch* erfolgen.

[5] engl. "*labeled* cartesian products"; ähnlich den Komponenten von RECORDs in PASCAL

[6] In [BMSW89] heißt es hierzu wörtlich: "Entity Classes have an associated extent: a set of objects with intrinsic, unchanging identity." Dies entspricht, wenn auch hier in anderem Zusammenhang, der in der Datenbankgemeinde üblichen Bezeichnung *"strong entity"*.

```
    budget : Money ;
END Companies;
```

Wie das Beispiel zeigt, können sowohl einzelwertige als auch mengenwertige Attribute vereinbart werden. Darüberhinaus werden Attribute als *UNCHANGING* oder *CHANGING* vereinbart, je nachdem ob eine Änderung bei einem Zustandsübergang des Systems erlaubt ist oder nicht. Zusätzlich können Attribute oder Attributgruppen als *UNIQUE* vereinbart werden, falls ihr "Wert" das zugehörige Objekt eindeutig identifiziert. Die frühzeitige Erfassung derartiger Beziehungen ist im Hinblick auf die für relationale Datenbanken bedeutsame *Schlüsseleigenschaft* interessant.

Mittels der in Abschnitt 6.1.2.3 beschriebenen prädikativen "assertion language" bestehen zusätzliche Möglichkeiten, Klassen mittels eines Prädikates *intentional* zu definieren und weitere Einschränkungen auf Attributen zu formulieren.
Damit ergibt sich der *Zustand des modellierten Systems* als die *Ausprägung* der vereinbarten Entitätenklassen in Verbindung mit den *Attributwerten*[7].

6.1.2.2 Beschreibung der Systemdynamik

In TDL werden gemäß ihrer Auswirkungen auf den Systemzustand zwei Arten von aktiven Systemkomponenten unterschieden:

- **Transaktionen** *verändern* den aktuellen Systemzustand, indem sie entweder die Extension[8] einer Entitätenklasse oder aber die Attribute von Entitäten verändern. Dies muß stets unter Wahrung der für das System vereinbarten Integritätsbedingungen geschehen.

```
TRANSACTION HireEmployee WITH
IN  name : String;
    belongs : COMPANIES;
    works : SetOf Project;
OUT e : Employees;
GOAL
    New(e) and Added(e,Employees) and (e.name´ = name)
    and (e.worksOn´ = works) and (e.belongsTo´ = belongs)
```

Unter den Attributkategorien IN und OUT werden die Eingabe- und Ausgabeparameter spezifiziert, über die mit der Transaktion kommuniziert werden kann. Getreu der nichtimperativen Gestalt von TDL-Spezifikationen wird die Menge der für die Transaktion zulässigen Zustandsübergänge in Form der im vorangegangen Kapitel beschriebenen Prä-Post-Notation spezifiziert. Unter die GIVEN-Klausel werden hierbei die Voraussetzungen

[7] Unter Berücksichtigung der Tatsache, daß Attribute selbst wieder *Entitäten*klassenausprägungen sein können, ist die Verwendung der Bezeichnung "Wert" anstelle von "Objekt" problematisch. Sie wird jedoch selbst in [BMSW89] verwendet.

[8] d.h. die Menge der Ausprägungen

formuliert, die vom "Aufrufer" der Transaktion erfüllt werden müssen. Die GOAL-Klausel formuliert die von einem die Transaktion realisierenden Zielsprachenprogrammsegment zu erbringende Leistung. Ihre Erfüllung wird daher während des Refinements statisch in Form von *Beweisverpflichtungen* verifiziert. Die speziellen Prädikate New, Added und Removed erlauben die Formulierung von *Zustandsänderungsbedingungen* in der GOAL-Klausel[9]. Darüberhinaus können noch INITIAL- und FINAL-Bedingungen formuliert werden, die sich von GIVEN bzw. GOAL in der Hinsicht unterscheiden, daß deren Gültigkeit zur *Laufzeit* überprüft werden muß[10]. Die Verletzung dieser Bedingung führt zu einer Ausnahmebehandlung während der Laufzeit.

- **Funktionen** arbeiten auf dem aktuellen Systemzustand, führen aber zu *keiner Änderung* desselben:

```
FUNCTION Plus WITH
 IN x, y : INTEGER ;
 RETURNS answer : INTEGER
 GOALS answer´ = x + y
```

Die Semantik der einzelnen Komponenten ist analog definert[11]. Zwischenergebnisse können in gesondert zu vereinbarenden Attributen der Kategorie LOCAL verwaltet werden.

Darüberhinaus bietet TDL die Möglichkeit, mittels sog. *Skripts* transaktionenübergreifende Ablaufketten zu beschreiben, die die relativen Zeitpunkte der Anwendung von Transaktionen gemäß den Gegebenheiten des modellierten Systems einschränken. Dies führt ggf. zu einer erheblichen Erhöhung der Lesbarkeit der Spezifikation und zu einer Reduktion der Bedingungsklauseln in Transaktionen. Für die Diskussion der B-Tool-spezifischen Aspekte spielen Skripts jedoch keine Rolle, so daß auf eine weitergehende Diskussion verzichtet wird (vgl. [DAID87]).

6.1.2.3 Erfassung von Integritätsbedingungen

Die in TDL zur Verfügung stehende "assertion language" erlaubt die Formulierung von Einschränkungen auf der Menge der zulässigen Zustände. Im Hinblick auf die in der Verfeinerungssprache sowie in der relationalen Zielsprache DBPL zur Verfügung stehenden Konstrukte werden Prädikate über *mengenorientierten* Ausdrücken formuliert, bei denen Klassen als *Mengen* und Attribute als *Funktionen* dargestellt werden. In der "assertion language" ist es möglich,

- abgeleitete Klassen *intentional*, d.h. durch Angabe eines einschränkenden Prädikates und einer Is-A-Beziehung zu definieren.

[9] Die in [BMSW89] verwendete Syntax erlaubt die abkürzende Formulierung der häufig auftretenden NEW-ADD-Kombinationen mittels PRODUCES in der OUT-Klausel.

[10] wie z.B. "Das Jahresbudget einer Firma darf die Summe der Gehälter aller Mitarbeiter nicht unterschreiten"

[11] Anstelle des "deklarativen" Stils kann die GOAL-Klausel auch in einer abgeleiteten Form in der RETURNS-Klausel formuliert werden:

```
RETURNS answer : INTEGER IS x + y.
```

- *Integritätsbedingungen* durch Einschränkung der für die einzelnen Attribute zulässigen Klassenobjekte zu formulieren. Dies kann auch allgemeiner durch die Zuordnung *boolescher Invariantenattribute* zur entsprechenden Klasse geschehen:

```
ENTITY CLASS Employees WITH
 UNCHANGING
  name : String;
 CHANGING
  belongsTo : Companies;
  worksOn : SetOf Projects;
 INVARIANT
  onEmpProj : True IS (this.worksOn subsetOf this.belongsTo.engagedIn)
```

Die Variable this läuft hierbei über die Menge aller Klassenausprägungen der Klasse Employees. Die formulierte Invariante muß zu jeder Zeit erfüllt sein. Spezielle Invariantenkategorien *INITIAL* und *FINAL* erlauben darüberhinaus die Formulierung von Bedingungen, die nur beim Einfügen bzw. Entfernen eines Klassenobjektes gelten müssen.

6.1.3 Abstrakte Maschinen

Die nächste Stufe auf dem Weg hin zur Implementation besteht in der Formulierung des in TDL entworfenen Systems als *abstrakte Maschine*. Die Rolle des Konzepts der abstrakten Maschinen als vereinheitlichendes Beschreibungsmittel im Rahmen des Softwareentwicklung ist aus Kapitel 5 bekannt.
In diesem Abschnitt wird nun eine von J.R.Abrial entwickelte *konkrete Notationsform* für abstrakte Maschinen vorgestellt. Durch eine schrittweise Konkretisierung[12] der in der initialen Maschine spezifizierten *statischen* und *dynamischen* Systemkomponten gelangt man zu einer abstrakten Maschine, die direkt in die Notation der jeweiligen Zielsprache transformiert werden kann. Zur Beschreibung der Systemoperationen wird die auf dem Prädikattransformerkonzept aufbauende "Generalized Substitution Language" verwendet. Die zum Nachweis der Modellkonsistenz sowie der Korrektheit der einzelnen Verfeinerungsschritte zu erbringenden Beweisverpflichtungen nehmen daher eine spezielle Gestalt an, die mit der oben gegebenen "prädikativen" Formulierung in Beziehung zu bringen ist.
Ebenfalls kurz diskutiert werden die im Rahmen von DAIDA noch nicht umgesetzten *Strukturierungskonzepte* der erweiterten Notationsform der abstrakten Maschinen, deren theoretische Grundlagen bereits besprochen wurden. Zu diesem Zweck werden die zur Rechtfertigung von Dekompositionen zu erbringenden Beweisverpflichtungen beschrieben und gezeigt, wie auf diese Weise das "laxe" bzw. "strenge" Hiding-Prinzip umgesetzt wird.
Eine Beispielverfeinerung zeigt auf, wie eine konkrete Systembeschreibung in der Notation der abstrakten Maschinen aussieht.

[12] im folgenden auch *Verfeinerung* oder Refinement genannt

6.1.3.1 Die Komponenten einer Spezifikationsmaschine

Die zur *Spezifikation* eines Systems formulierte abstrakte Maschine baut sich gemäß dem in Tabelle 6.1 gezeigten Schema auf (vgl. [Abri90a]).

KLAUSEL	BESCHREIBUNG
MACHINE	Angabe des Maschinennamens
SEES (optional)	Verweis auf sog. *Kontexte*, in denen Basismengen, Aufzählungsmengen und Konstanten sowie Eigenschaften der Basismengen vereinbart werden
USES (optional)	Möglichkeit der *Spezifikationsstrukturierung*: Verweis auf zu kombinierende abstrakte Maschinen
VARIABLES	Liste der *Zustandsvariablen* des Systems
INVARIANT	Liste von Prädikaten, die die *Typisierung* der Zustandsvariablen sowie weitere *Systeminvarianten* beschreiben
INITIALISATION	Eine in der Notation der verallgemeinerten Substitutionen formulierte Operation, die den *Startzustand* des Systems herstellt
OPERATIONS	Beschreibung der *dynamischen* Systemkomponenten als Menge von Operationen in der Notation der verallgemeinerten Substitutionen
ASSERTIONS (optional)	Weitere Prädikate, die als wahr vorgegebene Zusicherungen über das System erfassen und das Erfüllen der Beweisverpflichtungen vereinfachen
PROMOTES (optional)	Explizite *Übernahme* von Operationen einer im USE-Paragraphen referenzierten Maschine

Tabelle 6.1: Die Paragraphen einer Spezifikationsmaschine

Die Maschinenklauseln ASSERTIONS und PROMOTES sind in [Abri90a] nicht beschrieben, werden aber von der die B-Tool-Arbeitsumgebung realisierenden Theoriensammlung `devB` (vgl. u.) realisiert[13]. Für den Fall, daß Subspezifikationen über die USES-Klausel zur Verfügung gestellt wird, ist auch die VARIABLES-Klausel optional. Dasselbe gilt für die OPERATIONS-Klausel, falls Operationen über PROMOTES importiert werden.
Eine Beispielspezifikation wird in Anhang C.1.2 gezeigt.

6.1.3.2 Die statischen Komponenten von abstrakten Maschinen

In Abschnitt 5.1.1 wurden die Komponenten der Datenmodellierung im modellorientierten Ansatz nach Abrial ([Abri89]) identifiziert. Die Entsprechung im Rahmen der Notationsform der

[13] In den von `devB` zu verarbeitenden abstrakten Maschinen sind für die Schlüsselwörter der Klauseln die Abkürzungen `MCH`, `SEE`, `USE`, `VRB`, `INV`, `INI`, `OPN`, `ASN` und `PMT` zu verwenden

hier verwendeten Notationsform abstrakter Maschinen ist unmittelbar:

- *Systemvariablen* werden unter der VARIABLES-Klausel durch Auflisten ihrer Bezeichner definiert.
- Die *Systeminvariante* wird unter der INVARIANT-Klausel formuliert.

Die Gestalt der die Basismengen des Systems charakterisierenden CONTEXT-Vereinbarungen ist in Tabelle 6.2 dargestellt.

KLAUSEL	BESCHREIBUNG
CONTEXT	Angabe des Kontextnamens
SEES (optional)	Verweise auf weitere zur Verfügung stehende Kontexte
SETS (optional)	Liste *abstrakter* Basismengen
ENUMERATED SETS (optional)	Liste von durch (endliche) Elementaufzählung definierten Mengen
CONSTANTS (optional)	Vereinbarung abstrakter Systemkonstanten
RULES (optional)	Vereinbarung zusätzlicher Eigenschaften der Basismengen und Konstanten
PROPERTIES (optional)	wie RULES[14]

Tabelle 6.2: Die Paragraphen einer Kontextmaschine

Die SEES-Klausel einer abstrakten Maschine ist optional, da die Basismengen NAT, INT, BOOL $= \{TRUE, FALSE\}$, RAT und REAL als in einem für jede Maschine sichtbaren "globalen" Kontext vordefiniert angesehen werden. Alle in systemspezifischen Kontextdefinitionen vereinbarten Mengen werden implizit als *nichtleer* und *endlich* angesehen. Die durch die SEE-Verweise definierte Relation auf Kontexten ist *transitiv* und sollte *azyklisch* sein.
Der von der Beispielmaschine "gesehene" Kontext wird in Anhang C.1.1 gezeigt.
Die ASSERTIONS-Klausel ist ebenfalls zu den statischen Systemkomponenten zu rechnen und ähnelt von ihrem Inhalt der INVARIANT-Klausel. Die hierin definierten Prädikate werden jedoch bei der Generierung der *Konsistenz*-Beweisverpflichtungen abweichend behandelt:

- Die Initialisierungsoperation hat lediglich die Gültigkeit der INVARIANT-Prädikate, nicht aber der ASSERTION-Prädikate zu bewerkstelligen.
- Die weiteren Operationen haben unter Voraussetzung der Gültigkeit der Invariante *und* der ASSERTION-Prädikate den Erhalt lediglich der Invariante zu respektieren.

[14] Eine Unterscheidung zwischen RULES und PROPERTIES wird in [Abri90a] nicht getroffen. Die B-Tool-Theoriensammlung `devB` bereitet jedoch deren Inhalt im Hinblick auf deren Verwendung zum Beweis der Beweisverpflichtungen unterschiedlich auf (vgl. [Nils90a]).

Zusätzliche Beweisverpflichtungen verlangen den Nachweis, daß die ASSERTION-Prädikate aus der Invariante folgen[15]. Dies geschieht allerdings *isoliert vom Erfüllen der Konsistenzbeweisverpflichtungen.*

6.1.3.3 Die dynamischen Komponenten von Spezifikationsmaschinen

Die der in Abschnitt 5.1.2 beschriebenen Modellanimation dienenden Klauseln sind die INITIALISATION- und OPERATIONs-Paragraphen der abstrakten Maschine. Die Beschreibung der Operationen geschieht in der Sprache der verallgemeinerten Substitutionen, deren einzelne Konstrukte im folgenden beschrieben werden. Jedes Substitutionskonstrukt S erhält eine Semantik als *Prädikattransformer* (vgl. Abschnitt 5.1.2.3): einer Postkondition P wird durch S die *schwächste Präkondition* zugeordnet, unter der S P erstellt. Diese Definition erlaubt eine entsprechende abgewandelte Formulierung der entstehenden Beweisverpflichtungen.
Das Basiskonstrukt ist eine *Zuweisung* der Form

$$x := E$$

in der einer Zustandsvariablen x der Wert des Ausdrucks E zugewiesen wird[16]. Sei P ein Prädikat erster Ordnung, welches eine von der Substitution zu erstellende Postkondition formuliert. Die *schwächste Vorbedingung*, unter der die Zuweisung $x := E$ die Postkondition P realisiert, ergibt sich als das Prädikat

$$[x := E]P$$

welches sich aus P durch die *Substitution* von E für alle *freien* Vorkommen der Variablen x ergibt. Diese Definition begründet sich auf folgende Beziehung zu prädikativen Operationsspezifikationen:

$$\forall x \bullet ((x' = E) \Rightarrow P) \quad \Leftrightarrow \quad [x := E]P$$

(falls x nicht frei in E vorkommt).
Unter Anwendung elementarer prädikatenlogischer Gesetze läßt sich leicht zeigen, daß $[x := E]P$ tatsächlich die *schwächste* Vorbedingung ist, unter der P hergestellt wird ([Abri90a]).
Im folgenden werden die weiteren in *Spezifikations*maschinen zur Verfügung stehenden Substitutionskonstrukte durch die Angabe ihrer Prädikattransformersemantik definiert. Die Notation

$$[S]P$$

wird hierzu auf beliebig zusammengesetzte Substitutionen fortgesetzt.

- *leere Substitution skip*, läßt den Systemzustand unverändert. Semantik:

$$[skip]P \quad = \quad P$$

[15] Genauer: der Beweis des bezüglich der Reihenfolge der Auflistung $k+1$-ten ASSERTION-Prädikats geschieht unter der Voraussetzung der Invariante und der ASSERTION-Prädikate 1 bis k.

[16] Es werden a priori keine Einschränkungen über die Gestalt von E gemacht. E ist ein beliebiger "set theoretic term". Da jedoch x über die Invariante *typisiert* ist, ergibt sich eine entsprechende Beweisverpflichtung.

- *Parallele ("multiple") Substitution*

$$x_1, \ldots, x_n \quad := \quad E_1, \ldots, E_n$$

mit $x_i \neq x_j$ für $i \neq j$, stehend für *parallele* Ausführung der Elementarsubstitutionen $x_1 := E_1, \ldots, x_n := E_n$, erhält seine Semantik entsprechend: durch *paralleles* Substituieren der Ausdrücke E_i für die Variablen x_i in P erhält man in diesem Fall die schwächste Vorbedingung.
Abrial verallgemeinert die parallele Substitution auf nichtelementare Substitutionen $S_1, \ldots, S_n$ und führt die Schreibweise $S_1 \parallel \ldots \parallel S_n$ ein. Die Verknüpfung $\parallel$ realisiert dann gerade die in Abschnitt 5.1.5 beschriebene Operationskombination im Rahmen der Spezifikationsstrukturierung. (Allgemeine Gesetze erlauben allerdings die Rückführung der verallgemeinerten parallelen Substitution auf die elementare multiple Substitution ([Abri89])).

- Substitution mit *Vorbedingung*: Notation $Q \mid S$, wobei Q ein Prädikat (die Vorbedingung) und S eine verallgemeinerte Substitution ist. Eine Substitution dieser Form wird häufig dazu verwendet, Operationen zu formulieren, deren Anwendbarkeit nur in bestimmten Fällen zulässig ist. Semantik:

$$[Q \mid S]P \quad = \quad Q \wedge [S]P$$

d.h. falls Q nicht erfüllt ist, ist die Präkondition für beliebige Postkonditionen FALSE, d.h. die Substitution erreicht keinen Nachfolgezustand.

- Die *Auswahlsubstitution* (beschränkter Nichtdeterminismus). Notation: $S[]T$, wobei S und T verallgemeinerte Substitutionen sind. Semantik:

$$[S \,[]\, T]P \quad = \quad [S]P \wedge [T]P$$

Eine derart spezifizierte Substitution läßt insbesondere dem Implementierer die Wahl zwischen zwei Realisierungen (vgl. u.)

- Die *Substitution mit "Wächter" ("guard")*. Notation: $Q \Longrightarrow S$, wobei Q ein Prädikat ("guard") und S eine verallgemeinerte Substitution ist. Semantik:

$$[Q \Longrightarrow S]P \quad = \quad Q \Rightarrow [S]P$$

Es folgt: wann immer Q FALSE ist, stellt die Substitution *jede* Postkondition her[17].
Die konzeptuelle Trennung von Auswahl und Bedingung geht auf E.W. Dijkstra zurück, der erkannt hat, daß die aus üblichen Programmiersprachen bekannten konditionalen Konstrukte zwei getrennt erfaßbare Konzepte elementarerer Gestalt vereinheitlichen ([Dijk76]).
Die auch von Abrial eingeführte *konditionale* Substitution läßt sich definieren als

$$\begin{array}{l} [IF\ Q \\ THEN\ S \\ ELSE\ T]P \end{array} \quad = \quad [Q \Longrightarrow S \,[]\, \neg Q \Longrightarrow T]P$$

- *Unbeschränkter Nichtdeterminismus*: Verallgemeinerung der Auswahlsubstitution. Notation: $[@x.S]R$, mit der Semantik, daß die Postkondition unabhängig vom Wert von x erstellt werden soll, d.h.

$$[@x.S]P \quad = \quad \forall x \bullet [S]P$$

wobei x nicht frei in P vorkommen darf.

[17] sog. "Mirakulosität", vgl. u.

Für die unterschiedlichen Konstrukte werden zusätzliche lesbarere Schreibweisen definiert wie z.b. *PRE Q THEN S END* für die Substitution mit Vorbedingung (sog. *syntactic sugaring*). Eine Operationsdefinition wird durch die Angabe eines *Operationskopfes* komplettiert. Dieser vereinbart die *Parameter* der Operation sowie ggf. die Rückgabe eines Werts:

$$\mathsf{Operationsname}(par_1, \ldots , par_n) = S$$

$$\mathsf{x} \longleftarrow \mathsf{Operationsname}(par_1, \ldots , par_n) = S$$

wobei S eine verallgemeinerte Substitution ist.
I.a. ist S eine Substitution *mit Vorbedingung*, wobei die Vorbedingung eine mathematische Typisierung für die Parameter festlegt. Bei der Betrachtung der Gestalt der Beweisverpflichtungen werden daher lediglich Operationen dieser Gestalt betrachtet[18].
Für *Verfeinerungsmaschinen* ist die Substitutionssprache um zusätzliche "prozedurale" Konstrukte zu erweitern.

6.1.3.4 Begriffe und Eigenschaften der Substitutionsnotation

Eine als natürlich anzusehende Eigenschaft, die sich für die Substitutionsnotation nachweisen läßt, ist ihre Monotonieeigenschaft als Prädikattransformer:

> Seien X die Zustandsvariablen der Spezifikation und $P(X)$ und $Q(X)$ Prädikate über X[19]. Unter der Voraussetzung
>
> $$\forall X \bullet (P(X) \Rightarrow Q(X))$$
>
> gilt für beliebige Substitutionen S die Beziehung
>
> $$\forall X \bullet ([S]P(X) \Rightarrow [S]Q(X))$$
>
> D.h. Substitutionen sind *bezüglich der partiellen Prädikatordnung "⇒" monotone Funktionen.*

Eine weitere wichtige Eigenschaft ist ihr "distributives" Verhalten bezüglich konjunktiver Postkonditionen, die sog. *universelle Konjunktivität* ([Dijk76]):

$$[S](P \wedge Q) \quad = \quad [S]P \wedge [S]Q \tag{6.1}$$

Diese Regel wird sich als *eine wichtige Grundlage für die Vereinfachung der in der Substitutionsnotation formulierten Beweisverpflichtungen* erweisen.
Gemäß ihrer Eigenschaft als Prädikattransformer läßt sich in naheliegender Weise eine *partielle*

[18] Dies geschieht o.B.d.A., da *jede* Substitution S äquivalent ist zu *PRE TRUE THEN S END*. Abrial beweist sogar ein allgemeineres *Normalformresultat für verallgemeinerte Substitutionen* ([Abri90a], [Abri89]).
[19] d.h. in den freien Variablen X

Ordnung auf Substitutionen definieren: eine Substitution S heißt *besser* als eine Substitution T ($T \sqsubseteq S$) g.d.w. für *beliebige* Postkonditionen $P(X)$ gilt: falls T in einem Zustand $P(X)$ erstellt, so erstellt auch S in diesem Zustand $P(X)$, oder äquivalent

$$T \sqsubseteq S \quad \Rightarrow \forall X \bullet ([T]P(X) \Rightarrow [S]P(X)) \tag{6.2}$$

Dies ist Abrials *formale Definition der Verfeinerungsbeziehung für Substitutionen* ([Abri90d], [AGMS88c]). Der in Abschnitt 5.1.10 informell eingeführte Monotoniebegriff bezieht sich im Kontext der verallgemeinerten Substitutionen auf die partielle Ordnung "$\sqsubseteq$".
Bestimmte Substitutionen erstellen in manchen Zuständen *jede* Postkondition (d.h. sogar den elementaren Widerspruch $FALSE$). Man definiert den Begriff der sog. *Mirakulosität* [20]: eine Substitution S ist (aufgrund der Monotonieeigenschaft) mirakulös in genau den Fällen, in denen die Postbedingung $FALSE$ erstellt wird[21], d.h.

$$mir(S) \quad = \quad [S]FALSE$$

Solcherlei Substitutionen sind zwar nicht in Implementationen realisierbar, jedoch werden sie auf Spezifikationsebene zugelassen, denn

- *Komponenten* einer Substitution können mirakulös sein, ohne daß die Gesamtsubstitution mirakulös ist. Ein Beispiel hierfür ist der Fall der aus Auswahl- und "Guard"-Substitution kombinierten konditionalen Substitution. Es gilt

$$\begin{aligned} [Q \Longrightarrow S \,[]\, \neg Q \Longrightarrow T]FALSE \quad &= \quad (\neg Q \vee [S]FALSE) \wedge (Q \vee [T]FALSE)) \\ &= \quad ([S]FALSE \wedge Q) \vee ([T]FALSE \wedge \neg Q) \quad \vee \\ &\quad\quad ([S]FALSE \wedge [T]FALSE]) \end{aligned}$$

  d.h. sind S und T nicht mirakulös (z.B. als elementare Substitutionen), so ist auch die konditionale Substitution nicht mirakulös. Notwendige Voraussetzung ist in diesem Fall die Komplementarität der "Guard"-Prädikate, die bei der naheligenden Verallgemeinerung zu einer $CASE$-Substitution allgemeinerer Gestalt sein können.

- Die Invariante der Spezifikationsmaschine kann die Zustände, in denen Operationen mirakulös sind, ausschließen.

Als Komplement definiert man die sog. "*Realisierbarkeit*" einer Substitution (engl. *feasibility*) als[22]

$$feas(S) \quad = \quad \neg[S]FALSE$$

Auch ein *Terminationsbegriff* wird für die Substitutionssprache eingeführt: eine Substitution S terminiert, wenn sie wenigstens *eine* Postkondition erstellt. Die Monotonieeigenschaft erlaubt wiederum eine Formulierung als Prädikat erster Ordnung:

$$trm(S) \quad = \quad [S]TRUE$$

[20] engl. *miraculousity*

[21] Ohne die Monotonieeigenschaft wäre "*mir*" über ein Prädikat zweiter Ordnung (d.h. der Gestalt $\forall P \ldots$) zu definieren

[22] *Realisierbarkeit* ist *nicht* gleichzusetzen mit *Berechenbarkeit*. Realisierbarkeit einer Substitution besagt lediglich, daß der erstellte Nachfolgezustand ein *Modell* hat.

Beispielsweise gilt

$$trm(P \mid S) \quad = \quad P \wedge trm(S)$$

d.h. nur in Zuständen, in denen die Präkondition einer Substitution mit Vorbedingung *wahr* ist, wird S angewandt und, sofern S selbst terminiert, ein Endzustand erreicht. Andernfalls wird keine Aussage über den Zustand nach der Ausführung der Substitution gemacht, so daß diese nicht terminieren muß.
Strenggenommen handelt es sich bei Prädikattransformern um deklarative, nicht "ausführbare" Konstrukte; aus diesem Grund verwendet [Hehn84a] die Bezeichnung $usfl(S)$ (*Nützlichkeit*) anstelle der operationalen Terminologie $trm(S)$.

6.1.3.5 Konsistenz-Beweisverpflichtungen

Es werden zunächst Spezifikationen *ohne* USE-Verweise betrachtet. Sei C die Menge der im Kontext festgelegten Beziehungen, T die Initialisierungssubstitution und I das Konjunkt aller Invariantenprädikate der Maschine. Dann wird die in Beziehung 5.1 prädikativ formulierte Initialisierungs-Beweisverpflichtung zu

$$C \quad \Rightarrow \quad [T]I \tag{6.3}$$

D.h. zu zeigen ist, daß die Initialisierungsoperation unter Voraussetzung lediglich der Kontextdefinitionen die Systeminvariante I erstellt.
Sei $(Q \mid S)$ eine Operation der abstrakten Maschine und A das Konjunkt der Vereinbarungen der ASSERTION-Klausel. Die der Beziehung 5.2 äquivalente Beweisverpflichtung der Substitutionsnotation ist

$$A \wedge C \wedge I \wedge trm(S) \quad \Rightarrow \quad [S]I \tag{6.4}$$

D.h. zu zeigen ist, daß die Operation (Termination vorausgesetzt) die Invariante *erhält*.
Als zusätzliche Beweisverpflichtungen verlangt Abrial den Nachweis, daß eine spezifizierte Substitution realisierbar ("feasible") ist. Dies wird allerdings in der Implementation nicht umgesetzt.

6.1.3.6 Spezifikations-Dekomposition

Für den Fall, daß die Spezifikationsmaschine mittels USE-Verweisen auf lokale Spezifikationen zurückgreift, ist die Gestalt der Beweisverpflichtungen *allgemeiner*. Die Situation sei die folgende:

MCH	M1			
SEE	C			
USE	M2	$\longleftarrow$	MCH	M2
VRB	X		SEE	D
INV	I		$\vdots$	
INI	L		VRB	Y
OPN	$(Q \mid S)$		INV	J
	$\cdots$		INI	M
ASN	A		$\vdots$	
PMT	$(R \mid U)$		OPN	$(P \mid T)$
	$\cdots$			$\cdots$
			ASN	B
			$\vdots$	

Die Maschine M1 beachte ferner das laxe "Hiding"-Prinzip, d.h. sie referenziere die Zustandsvariablen der Subspezifikation lediglich *lesend* oder aber durch "Aufrufe" der von $M2$ vereinbarten Operationen.
Aufrufe lokaler Operationen in globalen Operationen können problemlos (mechanisch) durch den entsprechenden Operationsrumpf ersetzt werden; Parameterübergaben resultieren in entsprechenden Substitutionen im Rumpf der lokalen Operation, Ergebnisübergaben entsprechen einer zusätzlichen elementaren Substitution am Ende des Rumpfes der lokalen Operation. Bei der Erzeugung der Beweisverpflichtungen der Maschine $M1$ werden derartige Expansionen vorab vorgenommen.
Die Variablen von $M2$ können auch in den Invarianten- und Assertionsprädikaten von $M1$ auftreten.
Seien die lokalen Beweisverpflichtungen von $M2$ bereits erbracht[23]. Dann sind die *zusätzlich zu erbringenden Beweisverpflichtungen*

$$C \wedge D \Rightarrow [L \parallel M]I \tag{6.5}$$

$$A \wedge B \wedge C \wedge D \wedge I \wedge J \wedge trm(S) \Rightarrow [S]I \tag{6.6}$$

Da die lokale *Initialisierung* (im Unterschied zu den übrigen, lediglich lokal definierten Operationen) unabhängig von den globalen Operationen ausgeführt wird, ist diese in der Initialisierungsbeweisverpflichtung 6.5 zu berücksichtigen. Die lokalen und globalen Initialisierungen L und M arbeiten auf *disjunkten* Variablenmengen; somit entspricht die parallele Substitution einer beliebigen seriellen Ausführung von L und M[24].
Der Nachweis, daß die Operationen der globalen Maschine die Invariante erhalten, geschieht unter der zusätzlichen Voraussetzung von lokaler Invariante und lokalen ASSERTIONS-Vereinbarungen. Es ist lediglich der Erhalt der *globalen* Invariante I zu kontrollieren, da die Operation gemäß des "Hiding"-Prinzips keinen Schreibzugriff auf die lokalen Variablen Y haben und daher den lokalen Zustand unverändert läßt. Die Schnittstellenoperationen von $M2$ arbeiten lediglich auf den Variablen Y; der Erhalt der lokalen Invariante ist nach Voraussetzung bereits überprüft.
Die in der PROMOTES-Klausel angegebenen, von der lokalen abstrakten Maschine übernommenen Operationen sind wie die übrigen globalen Operationen zu behandeln, d.h. auch für diese

[23] M2 kann selbst wieder Verweise auf Subspezifikationen enthalten usw .
[24] Die *Hintereinanderausführung* von Substitutionen wird in den nächsten Abschnitten eingeführt.

ist die Beweisverpflichtung 6.6 zu erbringen, da sie (im Unterschied zu ausschließlich lokal definierten Operationen) auch auf globalem Level *unabhängig* ausführbar sind.
Die USE-Verweise der hier vorgestellten Notationsform abstrakter Maschinen realisieren somit das in Abschnitt 5.1.5 identifizierte Strukturierungskonzept für Spezifikationen. Wird auf PROMOTES-Importe verzichtet, so sind für die lokalen Operationen (mit Ausnahme der Initialisierung) keine zusätzlichen Beweisverpflichtungen mehr zu erbringen.

6.1.3.7 Die Komponenten einer Verfeinerungsmaschine

In den Klauseln einer Verfeinerungsmaschine sollen die unterschiedlichen Formen der Verfeinerung formulierbar sein. Demgemäß bestehen Abweichungen zur Syntax einer Spezifikationsmaschine:

- Anstelle eines Invariantenparagraphen verfügen Verfeinerungsmaschinen über eine CHANGE-Klausel, unter der die Beziehungen zwischen den Zustandsvariablen der abstrakten und der konkreten Maschine formuliert werden. Zur Unterscheidung der Variablen der unterschiedlichen Verfeinerungsebenen wird an jeden Bezeichner die Ebenennummer als Postfix angehängt.
  Während einer operationalen Verfeinerung (Abschnitt 5.1.10) entstehen Beziehungen der Gestalt $var_i = var_{i+1}$[25].

- In der an die Stelle von USE tretende IMPORT-Klausel können die ggf. im letzten Verfeinerungsschritt vereinbarten operationalen Dekompositionen in Form von Verweisen auf noch zu implementierende Teilspezifikationen (Spezifikationsmaschinen) formuliert werden. Hierbei ist das strikte "Hiding"-Prinzip zu beachten, da nur in diesem Fall die Gültigkeit der Beweisverpflichtungen hinreichend für die Implementationskorrektheit sind.

- Die übrigen Paragraphen behalten ihre Bedeutung. Hier können im Rahmen der Konkretisierung des spezifizierten Systems *zusätzliche* Deklarationen vorgenommen werden (insbes. im Kontext und in der Assertionsklausel).
  Operationale Verfeinerung geschieht in der OPERATION-Klausel.

Zur Operationsverfeinerung steht nun eine um Implementationskonzepte erweiterte Substitutionsnotation zur Verfügung.
Anhang C.2.2 zeigt die erste Verfeinerung der obenbetrachteten Spezifikationsmaschine.

6.1.3.8 Erweiterte Substitutionsnotation

Die zur Formulierung der Spezifikation zur Verfügung stehenden Substitutionskonstrukte werden um die *prozeduralen Konzepte Sequenzierung* und *Iteration* ergänzt:

[25] CHANGE-Prädikate dieser Gestalt werden in der B-Tool-Arbeitsumgebung für diejenigen Variablen *automatisch* generiert, die nicht einer Datenverfeinerung unterliegen.

- *Hintereinanderausführung* von zwei Substitutionen S und T.
  Notation: $(S;T)$. Semantik:

$$[S;T]P \quad = \quad [S]([T]P)$$

  d.h. sie entspricht der "Hintereinanderausführung" der entsprechenden Prädikattransformer (T vor S).

- *Iteration* einer Substitution S. Notation: $WHILE \quad Q \quad DO \quad S \quad END$. Die intuitive Definition

$$\begin{aligned} WHILE\ Q\ DO\ S\ END \quad = \quad & IF\ Q \\ & THEN\ S; \\ & \qquad WHILE\ Q\ DO\ S\ END \\ & ELSE\ skip \\ & END \end{aligned}$$

  führt zu einer Definition der schwächsten Vorbedingung R für eine Postkondition P als Lösung der *Fixpunktgleichung*:

$$R \quad = \quad [\neg Q \Rightarrow P \mid Q \Longrightarrow S]R$$

  Ein Fixpunktsatz von Tarski ist die Grundlage für das Lösen derartiger Gleichungen, wobei bestimmte gittertheoretische Eigenschaften der Substitutionskonstrukte als Funktionen über (bezüglich "$\Rightarrow$" partiell geordneten) Prädikaten wichtig sind[26].

Inwieweit im Laufe des Verfeinerungsprozesses Sequenzierung und Iteration eingeführt werden, hängt wesentlich von der *Zielsprache* der Entwicklungsumgebung ab. Für Zielsprachen mit einem hohen Grad an Parallelität werden Sequenzierung und Iteration nur selten verwendet. Die noch vorzustellende relationale Datenbanksprache DBPL bietet so z.B. Selektions- und Änderungsbefehle, die auf *Mengen* von Einträgen der Datenbank arbeiten. Dementsprechend werden Iterationen wesentlich seltener vorkommen als bei der Verfeinerung in Richtung auf *navigierende* Datenbanksprachen, wie sie für Datenbanksysteme basierend auf dem Netzwerk- oder Hierarchiemodell (vgl. [Schm87]) bekannt sind. Dieser Punkt soll weiter unten anhand eines Beispiels vertiefend diskutiert werden.

6.1.3.9 Beweisverpflichtungen der Verfeinerung

Zunächst werden Verfeinerungsmaschinen *ohne* operationale Dekomposition betrachtet.
Die in Abschnitt 5.1.9 prädikativ beschriebene Beweisverpflichtung der Verfeinerung läßt sich

[26] In [Abri89] wird darauf hingewiesen, daß die übliche Semantikdefinition für die Iteration ([Dijk76], [Hehn84a]) von der hier gegebenen abweicht. Der Grund liegt in der wegen des zusätzlich zur Verfügung stehenden *unbeschränkten* Nichtdeterminismus problematischen (gittertheoretischen) Stetigkeit der Substitutionskonstrukte. Eine unintuitive Konsequenz dieser Definition ist die Existenz terminierender Schleifen, deren Iterationszahl nicht durch einen Ausdruck über den Initialwerten der Zustandsvariablen beschränkt werden kann.

mit der in Gleichung 6.2 gegebenen Definition der Verfeinerungsbeziehung "$\sqsubseteq$" zwischen *Substitutionen* in Verbindung bringen: sei R das Konjunkt der CHANGE-Prädikate einer Verfeinerungsmaschine. R induziert eine korrespondierende Substitution

$$U \quad = \quad @Z.([Y := Z]R \Longrightarrow (Y := Z))$$

wobei Y die Variablen der "konkreten" Maschine sind. U wählt demnach (ausgehend von einer Belegung der Variablen der "abstrakten" Maschine, die nicht verändert wird) nichtdeterministisch eine mögliche, die CHANGE-Bedingung herstellende Belegung der Variablen der Verfeinerung.
Die nachzuweisenden Verfeinerungsbeziehungen lauten (Namenswahl wie zuletzt)

$$R \mid (L;U) \quad \sqsubseteq \quad M \tag{6.7}$$

$$(B \wedge D \wedge R \wedge I \wedge trm(S)) \mid (S;U) \quad \sqsubseteq \quad U;T \tag{6.8}$$

d.h. es ist zu zeigen, daß die konkrete Maschine sowohl Initialisierung als auch Operationen der "abstrakten" Maschine verfeinert. Die Vergleichbarkeit der auf disjunkten Variablen arbeitenden Substitutionen wird durch die Substitution U hergestellt.
Problematisch ist die Tatsache, daß es sich hierbei um eine Formulierung der Beweisverpflichtungen als Prädikat zweiter Ordnung handelt (wegen der Def. von "$\sqsubseteq$"). Abrial weist jedoch deren Äquivalenz zu einer Darstellung im Rahmen der Prädikattransformernotation nach ([AGMS88c]):

$$D \quad \Rightarrow \quad [M]\neg([L]\neg R) \tag{6.9}$$

$$B \wedge D \wedge R \wedge I \wedge trm(S) \quad \Rightarrow \quad trm(T) \wedge [T]\neg([S]\neg(R \wedge (x = y)) \tag{6.10}$$

x und y stehen für von der abstrakten bzw. konkreten Operationen gelieferten Resultate; für Operationen ohne Ergebnisübergabe entfällt $x = y$.)
Diese Formulierung läßt sich nun anschaulich mit der Gleichung 5.6 in Verbindung bringen: Die Prädikate der Gestalt $\neg([Sub]\neg P)$ stellen die Präkonditionen dar, unter denen eine Substitution *Sub nicht* die Postkondition $\neg P$ herstellt. Dies geschieht in Zuständen, in denen

- Substitution *Sub nichtdeterministisch* sowohl P als auch $\neg P$ herstellen kann, oder
- Substitution *Sub* manchmal P herstellt, jedoch nicht immer terminiert.

Der Nachweis, daß T unter den gegebenen Vorausetzungen immer terminiert, wenn S terminiert, wird in der Beweisverpflichtung erbracht. Das Erbringen der Beweisverpflichtung stellt ferner sicher, daß die Substitution $[T;S]$ allenfalls durch eine nichtdeterministische Ausführung von S das CHANGE-Prädikat verletzen kann, daß es aber zumindest eine Möglichkeit für S gibt, das CHANGE-Prädikat herzustellen. Somit erhält man eine Formulierung der in Abschnitt 5.1.9 beschriebenen zwei Arten der Verfeinerung im Rahmen der Prädikattransformernotation, die aus der Verfeinerungsbeziehung "$\sqsubseteq$" zwischen sich entsprechenden Substitutionen hergeleitet wurde.

6.1.3.10 Theoretische Probleme des Terminationsnachweises

Die im letzten Abschnitt formulierten Beweisverpflichtungen enthalten Konjunkte der Form $trm(S)$ $(= [S]TRUE)$. Die Auswertung dieser Formeln ist für Substitutionen S, die lediglich

die in Spezifikationen vorkommenden deklarativen Substitutionskonstrukte enthalten, problemlos, jedoch können Verfeinerungen durchaus Substitutions*iterationen* enthalten. Hehner weist nach, daß das Problem "Gegeben ein Zustand (Variablenbelegung) Z, gilt $[S]TRUE$?" im allgemeinen *unentscheidbar* ist ([Hehn84a]); die Aufgabe, $[S]TRUE$ zu entscheiden, ist eine Variante des *Halteproblems*.
Statt die Termination von allgemeinen Prädikattransformern nachzuweisen, schlagen [Hehn84a] und [Dijk76] daher *Konstruktionskriterien* für Iterationen vor, deren Befolgung die Termination sicherstellt. Dieser Ansatz wird letztlich auch in der Implementierung der Entwicklungsumgebung auf dem B-Tool verfolgt: zu jeder iterierten Substitution ist eine Schleifen*variante*[27] und eine Schleifen*invariante* anzugeben, für die zu zeigen ist: Für beliebige Zustände gilt

- ein einzelner Schleifendurchlauf läßt die (Schleifen)Invariante unverändert
- Unter der Annahme der (Schleifen)Invariante gilt: *ein* Durchlauf des Schleifenrumpfs dekrementiert die Variante. Hat die Variante den Wert 0, so ist die Schleifenbedingung nicht mehr erfüllt.

Die Erzeugung der Beweisverpflichtungen wird noch dadurch kompliziert, daß Schleifen geschachtelt auftreten können.
Die Bedingungen $trm(S)$ in den Beweisverpflichtungen werden in der Entwicklungsumgebung durch die (schwächeren) *Präkonditionen* der Substitutionen S ersetzt. Abrial gibt keine Rechtfertigung für diesen Schritt und verwendet abwechselnd beide Bezeichnungen. Unter der Annahme, daß die Termination von iterierten Substitutionen konstruktiv sichergestellt wird, gibt es jedoch gute Argumente für die Korrektheit dieses Vorgehens:

- Ein Normalformresultat für verallgemeinerte Substitutionen ([Abri89]) erlaubt die Umformung jeder beliebigen, nur *Spezifikations*konstrukte verwendenden Substitution in eine äquivalente Substitution, deren Termination äquivalent zur Erfüllung ihrer Präkondition ist.
- In "realistischen" Operationen tritt eine Substitution mit Vorbedingung lediglich auf äußerer Ebene auf, wobei die Vorbedingung elementare Mengenzugehörigkeitsprädikate ("Typisierungen") für die Operationsparameter beinhaltet. Auch hier ist die Termination (unter Annahme der konstruktiven Terminationsgarantie von Iterationen) äquivalent zur Erfüllung der Substitutionsvorbedingung.

Das zuletztgenannte Argument scheint die eigentliche Motivation für die Abschwächung der Terminationsprädikate zu sein.
So nimmt nun z.B. die Beweisverpflichtung 6.10 die vereinfachte Gestalt

$$B \wedge D \wedge R \wedge I \wedge Q \quad \Rightarrow \quad P \wedge [T]\neg([S]\neg(R \wedge (x = y)) \tag{6.11}$$

an.
Auf den Nachweis der Termination der *Initialisierungs*operationen wird aufgrund ihrer meist elementaren Gestalt[28] ebenfalls verzichtet.

[27] im Folgenden stets als *Zustandsvariable* vom Typ NAT aufgefaßt, kann aber auch allgemeiner als Funktion von den Zustandsvariablen nach NAT definiert sein.
[28] i.a. parallele Elementarsubstitutionen

6.1.3.11 Implementations-Dekomposition

Die in Abschnitt 5.1.10 beschriebenen Dekompositionsmöglichkeiten zur Operationsverfeinerung werden durch Verweise in den IMPORT-Klauseln realisiert. Nach einer Folge von Verfeinerungsschritten einer abstrakten Maschine lassen sich oftmals nichttriviale Teiloperationen identifizieren, die an verschiedenen Stellen der erstellten Verfeinerung auftreten und deren Realisierung hinreichend für die Implementierbarkeit ist. Dies legt eine *Ausgliederung* der entsprechenden Operationen in eine gesondertt zu implementierende, unter der IMPORT-Klausel referenzierte Maschine nahe (vgl. Beispiel in [Abri90d]):

REF	M1			
SEE	C			
IMP	M2	$\longleftarrow$	MCH	M2
VRB	X		SEE	D
CHG	C		$\vdots$	
INI	L		VRB	Y
OPN	$(P \mid T)$		INV	J
	$\cdots$		INI	M
ASN	A		$\vdots$	
PMT	$(R \mid U)$		ASN	B
	$\cdots$		$\vdots$	

Hierbei ist zu beachten, daß die Variablen von $M2$ weiteren Datenverfeinerungen unterliegen können und demnach das oben besprochene strikte Hiding-Prinzip zu beachten ist. Weder die Invariante noch die Operationen von $M1$ dürfen direkten Bezug zu den Variablen von $M2$ nehmen.
Sei K die Initialisierung und I die Invariante der durch $M1$ verfeinerten Spezifikation. Sei ferner $(Q \mid S)$ die der Operation $(P \mid T)$ entsprechende Spezifikation. Die Beweisverpflichtungen lauten

$$C \wedge D \Rightarrow [L \parallel M]\neg([K]\neg R) \tag{6.12}$$

$$A \wedge B \wedge C \wedge D \wedge I \wedge J \wedge R \wedge Q \Rightarrow P \wedge [T]\neg([S]\neg(R \wedge (x = y)) \tag{6.13}$$

Die Aufrufe der Operationen von $M2$ lassen sich wie im Falle der Spezifikationsdekomposition expandieren.

6.1.4 DBPL

Die Zielsprache der DAIDA-Entwicklungsumgebung ist die Datenbankprogrammiersprache DBPL[29] ([SEMa88]). Obwohl nicht unmittelbarer Gegenstand der im nächsten Kapitel bearbeiteten Problematik, beeinflußt die Zielsprache DBPL doch maßgeblich die Gestalt der in der Notation der abstrakten Maschinen durchgeführten Verfeinerungsschritte: bedeutsam sind u.a.

[29] engl. Database Programming Language

- die *relationale Verwaltung* der Daten in Tupelmengen,
- die *Verwendung assoziativer Selektoren* zum Datenzugriff,
- die von Transaktionen zu erhaltenden, *nicht modellinhärenten Integritätsbedingungen* des relationalen Datenbankmodells,
- die in DBPL zur Verfügung stehenden hochsprachlichen, *deklarativen* Sprachkonzepte.

Die Eigenschaften der DAIDA-Zielsprache DBPL werden sich als ein bestimmender Faktor für die genaue Gestalt der zu erbringenden Beweisverpflichtungen erweisen. Die Diskussion im folgenden Kapitel wird dies untermauern.

6.1.4.1 Einordnung

DBPL fällt in die Klasse sog. *Datenbankprogrammiersprachen*, die die Konzepte höherer Programmiersprachen sowie datenbankspezifische Datendefinitions- und Datenmanipulationskonzepte in einem vereinheitlichten Rahmen anbieten. Datenbankprogrammiersprachen sind von den darüberhinaus bekannten, in höhere Programmiersprachen "eingebetteten" Anfragesprachen[30] abzugrenzen, da letztere

- nur Daten*manipulations*konzepte bieten, und
- das Typkonzept der entsprechenden Wirtssprache nicht durch datenbankspezifische Typen wie z.B. *Relationentypen* ergänzen.

(vgl. [BJLM87]). Datenbankprogrammiersprachen erlauben die Definition lokaler, *datenbankspezifischer* Datenstrukturen und erweitern im Idealfall die von der Programmiersprache angebotene logische (i.a. aussagenlogische) Ausdruckssprache um *mächtige*, auf den Extensionen der Datenbank definierten Konnektive.
Die Integration von Datenmanipulationskonzepten bedingt die Realisierung von Kontroll- und Fehlererholungskonzepten, insbesondere eines *Transaktionskonzeptes*: Komplexe Operationen, bei denen nur die komplette Ausführung den Erhalt der Integrität der Datenbank garantiert, können als atomar abzuwickelnde *Transaktionen* definiert werden, für deren ungeteilte Ausführung das Datenbanksystem sorgen muß. Das Transaktionskonzept ermöglicht es, die Garantie für den Erhalt der Integritätsbedingungen in die Verantwortung des Anwenders zu legen, der auf diese Weise unter der Annahme einer atomaren Abwicklung konsistenzerhaltende Operationen definieren kann (vgl. [Reut87])[31]. In DBPL werden Transaktionen als ausgezeichnete Unterprogramme definiert.

[30] Beispielsweise wird die Anfragesprache SQL in verschiedenen Wirtssprachen (PASCAL, COBOL usw.) eingebettet angeboten.

[31] Die wesentlichen Eigenschaften des Transaktionskonzepts sind im sog. ACID-Prinzip zusammengefaßt (Atomicity, Consistency, Isolation, Durability), d.h. zusätzlich sind *Synchronisationsmaßnahmen* für den konsistenten parallelen Ablauf von Transaktionen zu treffen.

6.1.4.2 Integration in MODULA 2

Der Sprache DBPL liegt das *relationale Datenbankmodell* mit seinen wertbasierten ("*assoziativen*") Selektionsmechanismen[32] zugrunde (vgl. [Schm87]). DBPL stellt eine Erweiterung der imperativen Programmiersprache MODULA 2 ([Wirt88]) dar und bietet demnach

- ein mächtiges Typsystem und strenge Typisierung,
- Strukturierung umfangreicher Programme in sog. *Module*, die ihre Objekte (Variablen, Typvereinbarungen, Prozeduren) der Umgebung über eine *Schnittstellenvereinbarung* zum Import zur Verfügung stellen,
- Vereinbarung *vielfach verwendbarer Objekte* in *Definitionsmoduln*, denen die Rolle einer *Schnittstellenvereinbarung* zukommt, sowie getrennte *Realisierung* derselben in *Implementationsmoduln*. Die Objekte können so von *verschiedenen* weiteren Moduln gleichzeitig referenziert werden. Jedes Modul sieht eine eigene *Kopie* des importierten Objekts, wobei dessen Lebensdauer mit der Lebensdauer des importierenden Moduls übereinstimmt.

Die Ergänzung relationaler Datenbankkonzepte bedingt die

- Eingliederung eines entsprechenden Datentyps RELATION in das Typsystem von MODULA 2.
- Möglichkeit der Vereinbarung sog. *persistenter* Objekte, in denen die Daten der Datenbank abgelegt werden. Diese sind in ihrer Lebensdauer *nicht* an das importierende Modul gebundenen und stehen vielen Anwendungsprogrammen gleichzeitig zur Verfügung[33].

Es besteht die Möglichkeit der uneingeschränkten Parametrisierung der Typkonstruktoren von MODULA 2, d.h. insbesondere auch durch den Typ RELATION. DBPL wird daher als *typvollständig* bezeichnet (vgl. [Matt88]). Durch die beliebige (statische) Schachtelbarkeit von Relationsdefinitionen realisiert DBPL sog. *nichtflache* Relationen, d.h. Relationen, die im Sinne der Normalformenlehre (vgl. [MaDL87], [Ullm82]) nicht in erster Normalform sind und für die erweiterte Zugriffskonzepte (in einer entsprechend erweiterten Relationenalgebra durch sog. *nest-unnest-Operatoren* umsetzbar) benötigt werden[34].
Persistente Objekte werden in durch das zusätzliche Schlüsselwort DATABASE qualifizierten Moduln als Variable (insbesondere des Typs RELATION) vereinbart. Die auf der Datenbank arbeitenden Anwendungsprogramme importieren die zuzugreifenden Datenbankvariablen. Der Zugriff auf persistente Objekte darf in DBPL nur innerhalb von Transaktionen geschehen.

[32] Im Gegensatz zu den im Netzwerk- und Hierarchiemodell verwendeten *referentiellen* Selektoren ("Zeiger")

[33] Die Bedienung parallel zugreifender Programme ist Aufgabe der *Synchronisierungskomponente* des Datenbanksystems, die die Wahrung des Isolierungsprinzips zu realisieren hat ([Reut87]).

[34] Man spricht von sog. NF^2 (**N**on **F**irst **N**ormal **F**orm) - Relationen, die wegen ihrer Mächtigkeit z.B. für Nichtstandardanwendungen von Datenbanksystemen interessant sind.

6.1.4.3 Sprachkomponenten von DBPL

Relationentypvereinbarungen geschehen unter Anwendung des Typkonstruktors RELATION OF auf einen *Elementtyp*: die Vereinbarung

VAR dbvar: RELATION OF *Elementtyp*

legt fest, daß die Variable dbvar Werte aus der *Potenzmenge* über dem Elementtyp annehmen kann.
Für den Fall, daß der Elementtyp record- oder arrayförmig ist, können (optional) eine Liste von Recordfeldbezeichnern bzw. von Konstanten aus dem Indexbereich des Arrays angegeben werden. Die so bezeichneten Elementkomponenten bilden dann einen assoziativen Identifikator der Relation, d.h. sie sind *Schlüsselattribute*: die Definition

VAR dbvar: RELATION $k_{i_1}, \ldots, k_{i_m}$ OF
RECORD
$k_1 : T_1$;
$\vdots$
$k_n : T_n$
END

vereinbart die Variable dbvar als *Menge* von Objekten des angegebenen RECORD-Typs[35]. Durch die Möglichkeit, daß einzelne RECORD-Felder selbst wieder vom Typ RELATION sein können, werden (zusammen mit der in DBPL gegebenen adäquaten Anfrage- und Manipulationssprache) die obenbesprochenen NF^2-Relationenkonzepte umgesetzt.
Die RECORD-Felder $k_{i_1}, \ldots, k_{i_m}$ werden als Schlüsselattribute definiert, d.h. die Variable dbvar kann nur solche Extensionen als Wert annehmen, die nicht mehrere Elemente mit gleichen Schlüsselattributwerten beinhalten. Die Wahl der Schlüsselattribute unterliegt bestimmten Einschränkungen: relationenwertige Attribute dürfen ebensowenig Bestandteil von Schlüsselvereinbarungen sein wie Attribute aus den *Varianten* eines RECORD's ([Matt88]).
Ist die Liste der Schlüsselattribute leer, so ist die *Gesamtheit* der RECORD-Atttribute identifizierend.
Einzelne Elemente der Relation werden durch Angabe der Schlüsselattributwerte identifiziert:

relvar$[e_1, \ldots, e_m]$

ist der Objekt*bezeichner* des entsprechenden Elements (sofern vorhanden) und kann wie eine Variable vom Elementtyp der Relation verwendet werden, wobei die Einschränkung besteht,

[35] Mathematisch entspricht eine RECORD-Vereinbarung der Bildung eines *kartesischen Produktes* über den Attributtypen.

daß die Schlüsselkomponenten nicht verändert werden dürfen.
Eine entsprechende Erweiterung der Ausdruckssprache bilden *relationenwertige Ausdrücke*, die für Zuweisungen an Variable vom Typ RELATION verwendet werden können. Jede Relationenvariable ist ein relationenwertiger Ausdruck. Weiterhin sind relationale Ausdrücke durch *extensionale* und *intensionale* Beschreibung der Relationenausprägungen bildbar:

- *extensionale Beschreibungen* definieren die Ausprägungsmenge einer Relation durch (endliche) Aufzählung der Elemente ("Extensionen").
- *intensionale Beschreibungen* (Beschreibungen durch "*Regeln*") verwenden die in DBPL gegebenen prädikativen Selektionskonzepte und Aggregatkonstrukte zum Bilden von aus anderen Relationen abgeleiteten relationalen Ausdrücken.

Sei Rtype ein Relationentyp, dann ist

$$\mathsf{Rtype}\{elem_1, \ldots, elem_n\}$$

eine *extensionale Beschreibung* einer Ausprägungsmenge vom Typ Rtype. Die in der Elementauflistung angegebenen Ausprägungen müssen typkompatibel zum Elementtyp der Relation sein[36]. Im Spezialfall $n = 0$ ergibt sich der für die *leere Relation* vom Typ Rtype stehende Ausdruck.
Intensionale Beschreibungen werden in DBPL unter Verwendung sog. *Zugriffsausdrücke* realisiert, denen als *Regeln* zur Selektion und Konstruktion von Relationen in DBPL eine zentrale Rolle zukommt. Man unterscheidet zwischen *Selektionsausdrücken* und *Konstruktionsausdrücken*.
Selektionsausdrücke haben die Gestalt

$$\mathsf{EACH}\ r\ \mathsf{IN}\ exp_{rel} : exp_{bool}$$

wobei exp_{rel} selbst ein relationenwertiger Ausdruck und exp_{bool} ein Selektionsprädikat vom Typ BOOLEAN ist. Es wird die Relation derjenigen Elemente von exp_{rel} gebildet, die das Selektionsprädikat erfüllen. *Konstruktionsausdrücke* erlauben die Bildung *allgemeinerer Aggregatkonstruktionen* über *mehreren* relationenwertigen Ausdrücken und sind von der Form

$$\begin{array}{l} exp_{aggr}\ \mathsf{OF\ EACH}\ r_1\ \mathsf{IN}\ exp_{rel1} \\ \qquad\qquad\qquad\qquad\vdots \\ \qquad\qquad\qquad\mathsf{EACH}\ r_k\ \mathsf{IN}\ exp_{relk} : exp_{bool} \end{array}$$

[36] Ist der Elementtyp recordförmig, so sind die einzelnen Elemente von der Gestalt $\{attr_1, \ldots, attr_n\}$, d.h. an anderer Stelle dienen die geschweiften Klammern zusammen mit dem Komma als Separator zur Beschreibung von *Tupeln*. Für den Fall, daß einzelne Attribute relationenwertig sind, besteht "rekursiv" die Wahl zwischen extensionaler und intensionaler Beschreibung.

Der Aggregatausdruck wird aus den in den EACH-Klauseln angegebenen relationenwertigen Bezeichnern bzw. aus *Projektionen* hierüber gebildet. Es sind genau die Elemente der "Konstruktion" enthalten, die das Prädikat exp_{bool} erfüllen. Im allgemeinsten Fall kann exp_{aggr} selbst wieder intensionale oder extensionale Relationenbesschreibungen enthalten und auf diese Weise den Wert einer abgeleiteten NF^2-Relation annehmen.
Unter Angabe eines kompatiblen Relationentyps (als "Relationenkonstruktor") lassen sich mit Hilfe von Zugriffsausdrücken relationale Ausdrücke bilden:

RType{ *Zugriffsausdruck*}

Der angegebene Zugriffsausdruck bestimmt die Ausprägung der Relation, wobei zu beachten ist, daß der Elementtyp des definierten relationalen Ausdrucks mit dem angegebenen Elementtyp übereinstimmt. Die Notationsform ist eine entsprechende Erweiterung der bereits zur Bildung extensionaler Beschreibungen verwendeten Syntax.
DBPL bietet die Möglichkeit, Selektor- und Konstruktorausdrücke zu parametrisieren und zu benennen (*Selektoren* und *Konstruktoren*). *Konstruktor*definitionen realisieren ein flexibles *Sichtenkonzept* (sog. *views*), in dessen Rahmen eine *relationenübergreifende Informationsverknüpfung* stattfinden kann. Aus diesem Grund sind die auf diese Weise *ableitbaren* Relationen lediglich für *Lese*operationen zugelassen. In *Selektor*definitionen muß exp_{rel} ein relationenwertiger Bezeichner sein. Da hier lediglich eine Auswahl unter den Elementen *einer* Relation stattfindet, sind auch Modifikationsoperationen auf den selektierten Ausdrücken zulässig. Daher können zusätzlich Einschränkungen der Verwendbarkeit der Menge der selektierten Elemente auf Lese-, Einfüge-, Lösch- oder Änderungsoperationen formuliert werden. Darüberhinaus unterstützt DBPL Selektor- und Konstruktor*typen*, die Übergabe selektor- bzw. konstruktorwertiger Variablen als *Parameter* insbesondere von Transaktionen sowie eine schrittweise Substitution aktueller Parameter von Selektoren bzw. Konstruktoren und damit die *Spezialisierung von Zugriffsausdrücken unabhängig von deren Anwendung* ([SEMa88], [Matt88]).
Für die *Zuweisung* auf relationenwertige Variable stehen spezielle Einfüge-, Lösch- und Änderungsoperatoren zur Verfügung, die Abkürzungen für die am häufigsten wiederkehrenden Konstruktorformen darstellen:

- `relvar :+ rex` : Einfügen
- `relvar :- rex` : Löschen
- `relvar :& rex` : Ändern

der durch `rex` bezeichneten Ausprägungen in `relvar` ([SEMa88]).
Eine natürliche Erweiterung des Konzepts der definiten Iteration ist die Bestimmung des Iterationsbereichs durch einen Selektor oder Selektionsausdruck: so können in DBPL FOR-Schleifen der Gestalt

FOR EACH r in exp_{rel} : exp_{bool}
DO *Statements*
END

definiert werden, die die Operationen des Schleifenrumpfs auf alle selektierten Elemente anwenden. Die Reihenfolge des Zugriffs auf die selektierten Ausprägungen kann so dynamisch bestimmt und optimiert werden.
Die *boolesche Ausdruckssprache* von MODULA wird um Existenz- und Allquantifizierung über Relationenausdrücken erweitert:

SOME r IN exp_{rel} (exp_{bool})
ALL r IN exp_{rel} (exp_{bool})

sind wahr g.d.w. die Bedingung exp_{bool} für mindestens eine bzw. für alle Ausprägungen des angegebenen relationalen Ausdrucks exp_{rel} gilt.
Die Sprache zur Datenselektion fällt in die Klasse der sog. *Prädikatenkalkülsprachen*, da Selektionen unter Angabe von *Prädikaten* (erster Ordnung) geschehen. Da mittels der allgemeinen Konstruktionsausdrücke beliebige Kombinationen von *Selektion*, *Projektion* und *Join* zur Definition abgeleiteter Relationen gebildet werden können, wird DBPL im Sinne der Definition nach E.F. Codd als *relational vollständig* bezeichnet (vgl. [BJLM87]).
Transaktionen werden unter Angabe von Parameterliste, lokaler Variablen und der Operationsfolge wie gewöhnliche Prozeduren in MODULA-2 definiert. Der Unterschied liegt in der Zulässigkeit des Zugriffs auf persistente Objekte.

6.1.4.4 Vorteile deklarativer Sprachelemente

Selektor- und Konstruktorvereinbarungen stellen mächtige *Abstraktionsmechanismen* dar, die die Lesbarkeit von Programmen erhöhen. Die Verwendung von Zugriffsausdrücken ermöglichet den *parallelen* Zugriff auf die Ausprägungen der Datenbank durch rein *deskriptive* Sprachelemente, woraus u.a. folgende Vorteile resultieren:

- Abstraktion von konkreten Zugriffspfaden, da nur angegeben wird, *welche* Daten referenziert werden sollen, nicht jedoch, in welcher *Reihenfolge* der Zugriff zu erfolgen hat. Die Konstruktion der abgeleiteten Relation kann somit zur Laufzeit unter Berücksichtigung von Effizienzgesichtspunkten (Kardinalitäten, physische Zugriffspfade etc.) optimiert werden.
- Auch komplexe Selektionen und Konstruktionen können auf deskriptive Weise beschrieben werden, so daß in vielen Fällen auf die *explizite* Verwendung von Iterationen verzichtet werden kann.

Ein die These des zweiten Arguments stützendes Beispiel für die deskriptive Mächtigkeit von Zugriffsausdrücken wird in Kapitel 7 gegeben. Es wird gezeigt, in welcher Weise sich eine komplexe (geeignet verfeinerte) Operation in Substitutionsnotation *ohne* Verwendung von Iteration in DBPL umsetzen läßt. Da gerade die Schleifensubstitution in der Notation abstrakter Maschinen im Hinblick auf deren Behandlung in Beweisverpflichtungen semantisch problematisch ist (vgl. o.), zeigt sich der Vorteil der hochsprachlichen Konzepte von DBPL im Rahmen der

DAIDA-Sprachenhierarchie. Dies wird jedoch ohne restriktive Einschränkungen durch komplexe Transformationsregeln von der Notation abstrakter Maschinen nach DBPL "erkauft"; diese Problematik wird erneut in Abschnitt 7.3.4 aufgegriffen und anhand eines Beispiels diskutiert.

6.1.5 Integritätsbedingungen des relationalen Datenbankmodells

Aus dem im relationalen Datenbankmodell gegebenen assoziativen Selektionskonzept resultieren bestimmte Integritätsbedingungen, deren Erhalt von alllen Transaktionen sichergestellt werden muß:

- *referentielle Integrität*: da im relationalen Datenbankmodell nicht in n:m-Beziehungen substituiert wird[37], können einzelne Relationen assoziative Referenzen auf andere Relationen enthalten. Die so bezeichneten Ausprägungen müssen existieren.
- *Schlüsselintegrität* : Die aktuellen Ausprägungen einer Relation müssen sich paarweise in mindestens einem Schlüsselattribut unterscheiden.

In welcher Weise sich die nicht modellinhärenten Integritätsbedingungen im Verfeinerungsprozeß niederschlagen, wird in Abschnitt 7.3.3 diskutiert.

6.2 Aufgaben des B-Tools

Wie bereits eingangs erwähnt, hat das B-Tool in der Entwicklungsumgebung die Aufgaben eines *Abbildungsassistenten* sowie eines *Theorembeweisers* zu erfüllen. Die zweitgenannte Funktion umfaßt die *Erbringung* der Beweisverpflichtungen, die mit Hilfe einer B-Tool-Regelsammlung `devB` im Rahmen der Verarbeitung abstrakter Maschinen durch das B-Tool *erzeugt* und *vereinfacht* werden.
Diese beiden Anwendungsgebiete des B-Tools sollen in den folgenden Abschnitten beschrieben werden, wobei der Schwerpunkt auf der Diskussion der Arbeitsumgebung für abstrakte Maschinen liegt, da diese die Basis für die im nachfolgenden Kapitel behandelten Problematik ist.

6.2.1 Das B-Tool als Abbildungsassistent

Das B-Tool unterstützt die formale, regelbasierte Abbildung zwischen den unterschiedlichen Beschreibungssprachen der Entwicklungsumgebung:

- Transformation einer objektorientierten TDL-Beschreibung in die Notationsform abstrakter Maschinen.

[37] vgl. [Schm87], dadurch wird redundante Datenverwaltung vermieden

- Transformation einer durch *geeignete Verfeinerungsschritte* aus der initialen Spezifikation erhaltenen abstrakten Maschine nach DBPL[38].

[Wetz90] beschreibt die semantischen Beziehungen zwischen den unterschiedlichen Darstellungsformen in Gestalt von *Transformationsregeln*, deren Anwendung vom B-Tool automatisch vollzogen werden kann[39]. Die entsprechenden B-Tool-Implementationen sind jedoch noch nicht an die oben eingeführte erweiterte Notationsform[40] für abstrakte Maschinen angepaßt. Das im nächsten Kapitel vorgestellte Beispiel beinhaltet daher lediglich die im Rahmen einer Verfeinerung erstellten abstrakten Maschinen.

6.2.2 Verarbeitung abstrakter Maschinen durch `devB`

Die Verarbeitung abstrakter Maschinen auf dem B-Tool wird durch die Theoriensammlung `devB` ([Nils90a]) implementiert. `devB` leistet im einzelnen

- *Aufbereiten* der in abstrakten Maschinen enthaltenen Information und Speichern in gesonderten Theorien.
- *Generierung* und ggf. *Beweisversuch* der Konsistenz- und Verfeinerungsbeweisverpflichtungen.

6.2.2.1 Verarbeiten der Beweisverpflichtungen: das Lemma-Problem

Zur Verifikation der Beweisverpflichtungen werden zunächst *Umformungen* gemäß den in der Substitutionssprache geltenden Beziehungen des sog. *Generalized Substitution Calculus* sowie *Vereinfachungen* gemäß der obenbeschriebenen Eigenschaft der universellen Konjunktivität (6.1) vorgenommen. Hierbei dient die im B-Tool vordefinierte Substitutionsregel SUB zur Verarbeitung der Prädikattransformernotation als Basis. (vgl. u.).
Schließlich entsteht eine nicht mehr weiter reduzierbare Menge von Teilbeweisverpflichtungen, deren Gestalt von den in der INVARIANT- bzw. CHANGE-Klausel der verarbeiteten abstrakten Maschine formulierten Prädikaten abhängig ist. Die Gestalt dieser Prädikate ist *anwendungsspezifisch*: für die Entwicklung von DBPL-Programmen aus TDL-Spezifikationen werden beispielsweise

- die in TDL vereinbarten UNIQUE-Beziehungen und Attributtypen in den Invarianten der Spezifikationsmaschinen erfaßt,
- im Laufe des Verfeinerungsprozesses *kartesische Produktnotationen* in Typvereinbarungen eingeführt, eine Entscheidung im Hinblick auf die *relationale* Verwaltung der Extensionen der Datenbank in DBPL.

[38] Das im nachfolgenden Kapitel gegebene Beispiel wird intuitiv verdeutlichen, welche Eigenschaften DBPL-realisierbare abstrakte Maschinen besitzen müssen.

[39] Dies ist *noch nicht vollständig* realisiert worden und unterstützt z.B. nur eine restriktive Teilmenge der DBPL-Zielsprachenkonstrukte.

[40] d.h. mit externen Kontexten sowie USE- und IPT-Verweisen

Derartige Teilbeweisverpflichtungen werden **Lemmas** genannt. `devB` unterstützt deren Beweis nicht unmittelbar; es bedarf einer *zusätzlichen Axiomatisierung der anwendungsspezifisch benötigten Konzepte*.
Abbildung 6.2 skizziert die Situation.

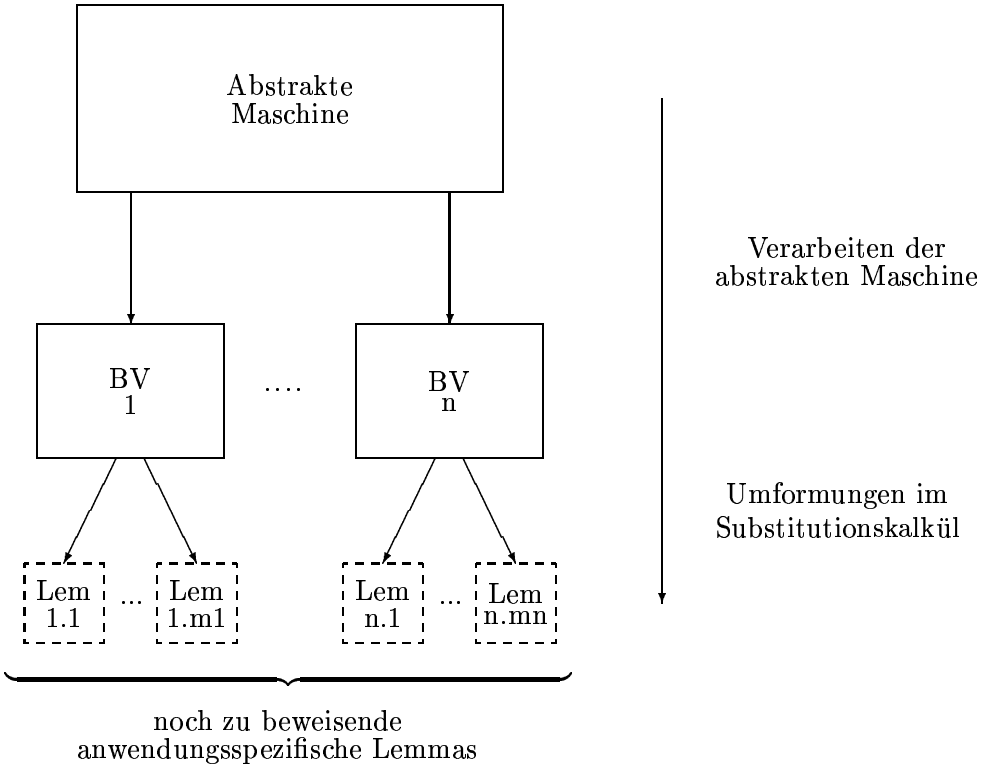


Abbildung 6.2: Verarbeitung der Beweisverpflichtungen

`devB` führt demnach ohne Weiteres lediglich einfache syntaktische Manipulationen und Vereinfachungen durch. Der eigentliche *Beweis* der Beweisverpflichtungen bedarf zusätzlicher Theorien zur Axiomatisierung der anwendungsspezifischen Symbole und findet erst auf der Ebene der Lemmas statt[41].
Gemäß der Gestalt der Lemmas wird eine *Grobklassifikation* vorgenommen: man unterscheidet zwischen

- *"Typ"-Lemmas*: Lemmas, bei denen eine Elementbeziehung der Gestalt $x : T$ (unter einer Menge von Voraussetzungen) zu beweisen ist.
  Typlemmas resultieren insbesondere aus den in den INVARIANT- bzw. CHANGE-Klauseln formulierten (mathematischen) Typvereinbarungen für die Zustandsvariablen der Maschinen.
- *"mathematische" Lemmas*: alle übrigen Lemmas, beispielsweise Inklusionsbeziehungen etc.

[41] Genaugenommen führt ein Deduktionssystem beim Beweis der Lemmas ebebnfalls nur "syntaktische Manipulationen" nach den Schlußregeln eines Logikkalküls durch.

Gegenstand des folgenden Kapitels ist die *Identifikation* der bei der Entwicklung von DBPL-Programmen typischerweise entstehenden Lemmas anhand eines Beispiels sowie die Entwicklung geeigneter Theorien für deren Beweis. In diesem Zusammenhang wird eine präzise *Lemmaklassifikation* vorgenommen, die die soeben vorgestellte Grobklassifikation verfeinert.

6.2.2.2 Beweismethoden

Eine spezielle, von `devB` verarbeitete Eingabeform sind sog. *Beweismethoden*, in denen Vereinbarungen für den Beweis der generierten Beweisverpflichtungen *und* der Lemmas getroffen werden können, insbesondere

- Angabe von Theorien und Taktiken zum Beweis der anwendungsspezifischen Lemmas.
- automatischer Beweisversuch der erzeugten Beweisverpflichtungen *und* der daraus gewonnenen Lemmas unter Verwendung eventuell angegebener Theorien und Taktiken.

Damit ist es möglich, über die Vereinbarung anwendungsspezifischer Theorien die gemäß Abbildung 6.2 noch fehlenden Teilbeweise von `devB` durchführen zu lassen.
Beweismethoden enthalten u.a. folgende Klauseln[42]:

- `AUTOP`: "Autoproof-Flag", zu vereinbaren als "`on`" oder "`off`". Falls "`on`", so wird bei den zukünftig eingelesenen abstrakten Maschinen versucht, die Beweisverpflichtungen und die dabei erzeugten Lemmas automatisch, d.h. ohne Rückfrage an den Benutzer, zu beweisen (Default: "`off`").
- `MATHTAC, TYPETAC`: hier können für die entsprechende Lemma-Art *Listen* von Paaren von Vorwärts-und Rückwärtstaktiken angegeben werden. Für jedes erzeugte Lemma werden die Taktikkombinationen in der angegebenen Reihenfolge zu Beweisversuchen herangezogen. Darüberhinaus existieren Default-Taktiken für den Beweis der Lemmas ([Nils90a]).
- `REFTAC, INVTAC` dienen zur Vereinbarung spezieller, von den in `devB` bestehenden Vordefinitionen abweichenden Taktiken zum Beweis der generierten Beweisverpflichtungen.
- `THYS`: hierunter sind *spezielle, insbes. zum Beweis der Lemmas nützliche Theorien* definierbar, die in den soeben beschriebenen Taktikdefinitionen referenziert werden können. Formuliert werden können Theorien zum Gebrauch in Vorwärtsbeweisen und Rückwärtsbeweisen (`THEORY` bzw. `FWDTHEORY`)[43].

Durch die Verarbeitung von Beweismethoden ermöglicht `devB` dem Benutzer auf komfortable Weise, die noch bestehende Beweislücke zu schließen.

[42] für eine vollständige Übersicht vgl. [Stuc90]

[43] Für die Regeln der FWD-Theorien werden, wegen der obenbeschriebenen Bedeutung der Reihenfolge ihrer Antezedenten für deren Anwendbarkeit in Vorwärtsdeduktionen, automatisch "Permutationen" ergänzt, in denen die anderen Regelantezedenten an erster Position stehen.

6.2.2.3 Arbeitsweise von devB

Anhang B zeigt einen Überblick der Theorien von devB.
Das *Einlesen der Eingaben* geschieht über die Theorie INPUT, für die eine Taktik definiert ist, die die Weiterverarbeitung regelt. In der Theorie ParseInputX werden die Eingaben gemäß ihrer Art (MCH, REF, CTX, PMD) identifiziert und deren artspezifische Weiterverarbeitung initiiert ([Stuc90]).
Die einzelnen Beweisverpflichtungen werden zunächst in abstrakter Form dargestellt, für Konsistenzbeweisverpflichtungen z.B.

```
ctx(Mch) &
inv(Mch) &
asn(Mch) &
pre(MchOp)
  =>
chkinv(inv(Mch),(pre(MchOp) | sub(MchOp))
```

Nach Anwendung der eingebauten Regel DED wird das vorliegende Ziel durch Anwendung der Theorie BuildConjectureX (im Beispiel durch die zweite Regel) in die Prädikattransformernotation übersetzt:

```
THEORY BuildConjectureX IS

      [K]P => chkinv(P,K);

      [K]P => chkinv(P,(Q | K));

      [K]P => chkinvini(P,K);

      [A]not([B]not(P)) => chkinv(P,A,B);

      [A]not([B]not(P)) => chkinv(P,A,(Q | B))
END
```

Die ersten drei Regeln erzeugen die Beweisverpflichtungen für eine *Spezifikations*maschine, wobei zwischen Operationen ohne und mit Vorbedingungen (Regeln 1 und 2) und der Initialisierungs-Berweisverpflichtung (Regel 3) unterschieden wird. Regeln 4 und 5 erzeugen die Beweisverpflichtungen der Verfeinerung.
Theorie CalculusX enthält die zum *Verarbeiten der Beweisverpflichtungen* benötigten Beziehungen des *generalized substitution Calculus*, u.a.

```
[skip]R = R;

[P | I]R = (P & [I]R);

[P==>S]R = (P => [S]R);

[S [] T]R = ([S]R & [T]R);

[@x.I]R = !x.[I]R;

[I;J]R = [I][J]R;

[I][J]R => [I;J]R;

R => [skip]R;

P & [I]R => [P | I]R;

P => [S]R => [P==>S]R;

[S]R & [T]R => [S [] T]R;
```

Durch Anwendung dieser Regeln können zusammengesetzte Substitutionen schrittweise in Elementar- bzw. Parallelsubstitutionen zerlegt werden, die durch die eingebaute Regel SUB direkt durchführbar sind. Die so entstehenden Formeln enthalten keine Konstrukte der Substitutionsnotation mehr. Formeln, die nicht in der Hypothese enthalten sind, werden als Lemmas abgespeichert. Falls die AUTOP-Flag on ist, wird zuvor ein Beweisversuch mit den in der Beweismethode vereinbarten Taktiken durchgeführt.
Die *Taktiken* für die Verarbeitung der Konsistenz- und Verfeinerungsbeweisverpflichtungen bauen sich demgemäß auf:

```
(InvariantLemmaX;DED;GetOperationX~;FLAT;GetInitialisationX; GetInvariantX~;
   BuildConjectureX;
   (((GEN;DED)~;CalculusX)~;PrepareLemma1X)~)
```

bzw.

```
(RefinementLemmaX;DED;
 (PrepareConjectureX;REV;(PrepareConjectureX;ARI)~;FLAT)~;
 GetOperationX~;GetInitialisationX~;TransCondX~;GetInvariantX;
 BuildConjectureX;FLAT~;
 (((GEN;DED)~;CalculusX)~; PrepareLemma1X)~)
```

wobei für Verfeinerungslemmas noch bestimmte Umformungen der internen Repräsentation vorgenommen werden (zweite Zeile der Taktik).
Die Verarbeitung der Beweisverpflichtungen und die Lemmagenerierung wird im folgenden Kapitel anhand eines Beispiels vertiefend diskutiert.
`devB` generiert ebenfalls die *für Schleifensubstitutionen zu erbringenden Beweisverpflichtungen* anhand der vorgegebenen Invarianten und Varianten (vgl. Abschnitt 6.1.3.10). Dieser Punkt ist für die Ausführungen des folgenden Kapitels irrelevant und soll daher nicht weiter diskutiert werden.
Die *Verarbeitung der aus den Beweisverpflichtung erzeugten Lemmas* geschieht in den Theorien `PrepareLemma1X, PrepareLemma2X` und `PrepareLemma3X`. Ist die Autoproof-Flag "`on`", so wird versucht, die Lemmas mit Hilfe der vereinbarten Taktiken zu *beweisen* (`PrepareLemma2X`), wobei zwischen Math-und Typelemmas unterschieden wird. Andernfalls erfolgt deren *Ablage* unter Beachtung der Lemmaseparation-Flag (`PrepareLemma3X`).
Die aus den Klauseln der verarbeiteten Eingaben gewonnene Information wird in den "Get*"-Theorien am Ende der Theoriensammlung abgelegt (vgl. Anhang D für das im folgenden diskutierte Beispiel) und steht in dieser Form zum Beweis der Lemmas zur Verfügung.

Kapitel 7

Behandlung des Lemma-Problems für DAIDA-Verfeinerungen

Das Problem des Beweisens der anwendungsspezifischen Lemmas soll nun speziell für die Entwicklung von DBPL-Programmen in der DAIDA-Entwicklungsumgebung behandelt werden. Es wird eine spezielle *Beweismethode* entwickelt, mit der es möglich ist, einen großen Teil der aus den Konsistenz- und Verfeinerungs-Beweisverpflichtungen resultierenden mathematischen und Typlemmas zu beweisen. Das Kapitel gliedert sich wie folgt:

1. Diskussion des theoretischen Hintergrunds der vorzunehmenden Axiomatisierung und Einordnung des zur Lösung des Lemmaproblems eingeschlagenen Weges.
2. Entwicklung einer Beispielverfeinerung, die die für Datenbankprogramme typischen Einfüge-, Lösch- und Änderungsoperationen enthält.
3. Diskussion der Entwurfs- und Zielsprachenabhängigkeit von Verfeinerungen; Implikationen der im relationalen Datenbankmodell nicht impliziten Integritätsbedingungen.
4. *Klassifikation* der Lemmas gemäß ihrer generischen Gestalt. Aufschlüsselung ihrer Herkunft (Inbeziehungsetzen zu den unterschiedlichen Beweisverpflichtungen, Invariantenprädikaten und Operationen des Beispiels).
5. Beschreibung der *Implementation*: Entwicklung spezieller Theorien und Taktiken für den Beweis der Lemmas, aufbauend auf der vorgenommenen Lemmaklassifikation.
6. Besprechung von Beispielbeweisen.
7. Ableitung allgemeiner und spezieller Richtlinien zur Theorieentwicklung. Diskussion.

7.1 Theoretischer Hintergrund

In einer frühen Arbeit ([Abri84]) stellt Abrial die Axiomatisierung einer Umgebung vor, die Konzepte aus Logik, Mengentheorie und (verallgemeinerten) Programmiersprachen beinhaltet.

Es werden Theorien angegeben für

- Aussagenlogik
- Prädikatenlogik
- Gleichheitstheorie (Axiomatisierung eines ausgezeichneten Prädikats "=" der Prädikatenlogik)
- Theorie der *Deklarationen*
- Mengentheorie (Eine im Hinblick auf ihre Verwendung zur Objekttypisierung *eingeschränkte* Form der Zermelo-Frænkel-Mengentheorie)
- Relationen, partielle und totale Funktionen
- Rekursionstheorie (Fixpunkt- und μ-Operatoren)
- Natürliche Zahlen (abgewandelte Form der Peano-Axiomatisierung)
- Theorie der *Programme* einer verallgemeinerten Programmiersprache. Dabei wird eine modellorientierte Sicht angenommen und die einzelnen Instruktionen durch Prä-Post-Prädikate im Sinne von Abschnitt 5.1.2.1 axiomatisiert.
- Theorie der *Berechnungen*

Der Entwicklung eines Programms aus einer Spezifikation entspricht in diesem formalen Rahmen die *Konstruktion eines Beweises*. Die Axiomatisierung ist unabhängig von einem bestimmten Anwendungsbereich und steht damit im Gegensatz zum hier gewählten Vorgehen, welches in diesem Abschnitt näher begründet werden soll.
[Abri84] verwendet eine Metanotation, in der auch Nebenbedingungen formaler Gestalt (freie und gebundene Variable, syntaktische Konventionen usw.) durch geeignete Axiomatisierung erfaßt werden.

7.1.1 Beziehung zur B-Tool-Logik BL

Die Metasprache verwendet die gleichen Konstrukte wie die zur theoretischen Fundierung des B-Tools formulierte Metalogik BL. In der Tat kann die in [Abri84] erfolgte Axiomatisierung einer Umgebung zur Programmentwicklung als eine *Fallstudie* aufgefaßt werden, die die prinzipiellen Anforderungen an die einem geeigneten Beweisassistenten zugrundeliegenden metalogischen Konzepte herausarbeitet. So heißt es dort

> *In being* completely formal *(Bourbaki 1970) we had in mind the possible mechanization of that part of proof writing that is* completely clerical *and thus extremely tedious and error prone.*

Die Metalogik BL bietet die benötigten Metakonzepte an. Es ist daher prinzipiell möglich, die in [Abri84] angegebene Axiomatisierung vollständig in BL zu erfassen. Da das B-Tool, wie in Abschnitt 4.3 beschrieben, auf der Metalogik BL basiert, ist zu erwarten, daß die formale Umgebung aus [Abri84] auf dem B-Tool durch Formulierung geeigneter Theorien axiomatisierbar ist. Das B-Tool realisiert somit die in [Abri84] identifizierten Anforderungen.

7.1.2 Beziehung zur DAIDA-Methodologie

In DAIDA werden Operationen als verallgemeinerte Substitutionen formuliert und die Beweisverpflichtungen unter Anwendung der Beziehungen des *Generalized Substitution Calculus* (vgl. Abschnitt 6.2.2.1) verarbeitet. Dies ermöglicht die Elimination jeglicher Substitutionsnotation durch *elementare syntaktische Umformungen*, deren Durchführung durch das B-Tool *effizient* bewerkstelligt wird. Hieraus resultieren, wie bereits zuvor beschrieben, die Elementarbeweisverpflichtungen in Gestalt der Lemmas.
In [Abri84] hingegen ist die Operationssemantik in Form von *Prä-Post-Definitionen* für die elementaren Programmkonstrukte gegeben. Beweisverpflichtungen laufen dort (im Sinne von [HeGM86], erweitert um Deklarations- und Typisierungskonzepte) auf die *logische Folgerung* der Spezifikationsprädikate aus einem oft recht komplexen *Operationsprädikat* hinaus. Der Korrektheitsbeweis kann zwar prinzipiell unter Verwendung des in [Abri84] angegebenen Axiomensystems durchgeführt werden, jedoch erscheint dies aufgrund der oft nichtelementaren Gestalt der Operationsprädikate sowie der Vielzahl "formaler" Axiome als ineffizient.
Die Substitutionsnotation erfährt damit eine weitere (relative) Rechtfertigung, da deren Eigenschaften eine syntaktische Zerlegung von Gesamtbeweisverpflichtungen in elementare Subkomponenten förmlich aufdringen.

7.1.3 Implikationen für das Lemmaproblem

Der formale, streng axiomatische Ansatz aus [Abri84] hat wegen der Ineffizienz des resultierenden Systems keine unmittelbare Relevanz. Eine andere Vorgehensweise wird sich jedoch als praktikabel erweisen: aufgrund der B-Tool-Vordefinition eines Kalküls des natürlichen Schließens sowie der syntaktischen Defaultannahmen (vgl. o., Kapitel 4) steht bereits ein unter Effizienzgesichtspunkten wesentlich *leistungsfähigerer* Rahmen zur Verfügung, als ihn [Abri84] bietet. Die Axiomatisierung der Prädikatenlogik sowie die Explizierung formaler Nebenbedingungen entfällt. Die im folgenden formulierten Theorien beinhalten lediglich solche Regeln, die für die Verifikation der Beweisverpflichtungen in DBPL-Programmentwicklungen als nützlich angesehen werden. So gibt es spezielle Axiome für kartesische Produkt-Notationen, zum Beweis des Erhalts der Schlüsselintegrität unter bestimmten Operationstypen, aber auch für elementare mengentheoretische Beziehungen.
Hierbei wird folgender *Lösungsweg* verfolgt:

- Die aufgestellten B-Tool-Theorien und die zugehörigen Taktiken werden *nicht* mit dem Anspruch auf *Vollständigkeit*, (d.h. der Beweisbarkeit *jedes* Lemmas) entwickelt.
- Die aufgestellten Regeln können *Redundanz* aufweisen. Dies steht wiederum in Kontrast zu [Abri84], wo, ausgehend von den Elementarkonstrukten "$\in$" und "{ }", alle weiteren

Konnektive *definiert* werden.
Eine derartige *Definition* abgeleiteter Symbole ist i.a. weniger effizient als die Angabe zusätzlicher, eigentlich redundanter Axiome. Letzteres scheint für diejenigen Konnektive interessant, die der Benutzer bevorzugt bei der Verarbeitung *seiner* typischen Anwendung verwendet (was im hier vorliegenden Fall DBPL-Programmentwicklungen sind).

Auf Vollständigkeit muß schon *aus theoretischen* Gründen verzichtet werden: die in [Abri84] identifizierte, durch BL im wesentlichen realisierte Metasprache erlaubt die Axiomatisierung von Prädiktenlogiken höherer Ordnung. Die in [Abri84] gegebene Axiomatisierung der natürlichen Zahlen beinhaltet so z.B. das *Induktionsaxiom*, welches eine über Prädikate über den natürlichen Zahlen allquantifizierte Aussage macht (vgl. Kap. 3)[1]. Dies ist ein Beispiel für die prinzipielle Mächtigkeit von BL; durch Angabe *schematischer* Axiome können beliebige Regeln der Prädikatenlogik zweiter Ordnung formuliert werden. Für Prädikatenlogiken *höherer* Ordnung ist bekannt, daß die noch für PL1 geltende Semientscheidbarkeit verlorengeht, d.h. es kann nicht für jede aus den Axiomen folgende Aussage ein Beweis in endlicher Zeit gefunden werden[2]. Die Unvollständigkeit resultiert daher aus den nur in Objektsprachen höherer Ordnung formulierbaren "komplexen" Axiomen des Anwendungsbereichs.
Da eine formale und vollständige Axiomatisierung mächtiger Typkonzepte, in denen *Folgen* (Arrays), Baumstrukturen usw. definiert werden können, auf den Axiomen der natürlichen Zahlen aufbaut ([Abri84]), kann in einem *beliebige* induktive Strukturen unterstützenden Ansatz auf Axiome höherer Ordnung nicht verzichtet werden[3]. Es bietet sich daher an, stattdessen anwendungsspezifische Theorien zu entwickeln, mit denen zwar einige Lemmas nicht bewiesen werden können, die aber im Vergleich zu dem in [Abri84] vorgestellten, ebenfalls unvollständigen *formalen* Ansatz wesentlich effizienter sind.
Ein weiterer Grund dafür, daß Vollständigkeit nicht immer erreichbar ist, liegt in der Verwendung bestimmter Konstrukte, deren Semantik nicht axiomatisierbar ist[4]. Hieraus können Lemmas resultieren, die nur unter Verwendung von nicht in der Entwicklungsumgebung erfaßtem "Metawissen" beweisbar sind (vgl. u.), deren Gestalt jedoch i.a. so einfach ist, daß eine visuelle Verifikation durch den Benutzer problemlos ist.
In der Arbeit [McNe89] eines DAIDA-Projektpartners wird ebenfalls das Problem des Lemmabeweisens behandelt und Theorien angegeben, die sich für ein spezielles Beispiel als geeignet erwiesen haben. Die dort vorgestellte Theoriesammlung beinhaltet insbesondere spezielle Axiome für *arithmetische Ausdrücke*, durch die es ermöglicht werden soll, Lemmas "numerischer" Gestalt zu beweisen, die für das angegebene Beispiel erzeugt werden. Darüberhinaus werden elementare mengentheoretische Konzepte axiomatisiert, u.a. Element- und Inklusionsbeziehungen, Relationen und Funktionen.
Auch der in [McNe89] gewählte Ansatz weist Ungenauigkeiten auf. So erfolgt lediglich eine teilweise Axiomatisierung der natürlichen Zahlen; ein Induktionsaxiom wird nicht angegeben. Für den arithmetischen Operator "+" ist die Assoziativität axiomatisiert, nicht jedoch für die Multiplikation "*". Für ein ausgewähltes, anhand eines dort angegebenen Beispiels erzeugten

[1] Die Bedeutung induktiver Strukturen wurde bereits an anderer Stelle motiviert.

[2] Es ist zwar bekannt, daß für jede aus den Axiomen folgende Aussage ein Beweis *endlicher* Länge existiert, jedoch ist diese Existenz für Prädikatenlogiken höherer Ordnungen *nichtkonstruktiv* (und für PL1 konstruktiv, aber nichteffektiv), vgl. [Ende72].

[3] Die "*Computational Logic*" von Boyer und Moore ([BoMo79]) zeigt, daß man auch unter *Einschränkung* der zu unterstützenden induktiven Schlußfolgerungen auf "konstruktive" Spezialfälle zu einem gangbaren Ansatz kommen kann (vgl. auch unten, Kapitel 8).

[4] Die unten beschriebene Beispielverfeinerung enthält Beispiele für derartige Konstrukte, vgl. u.

mathematischen Lemmas wird ein Beweis sowie die verwendete Taktik angegeben[5]. Jedoch wird keine Aussage darüber gemacht, inwieweit Theorien und Taktik auch zum Beweis der übrigen Lemmas verwendbar sind.
Das in [McNe89] angegebene Beispiel betrachtet überdies lediglich die für einen *Konsistenz*beweis zu erbringenden Beweisverpflichtungen. Eine Verfeinerung in Richtung einer speziellen Zielsprache wird nicht angegeben. Aufgabe der im folgenden durchgeführten Betrachtungen soll es jedoch u.a. sein, die während einer *Verfeinerung* von Spezifikationen im Hinblick auf eine DBPL-Implementation typischerweise erzeugten Lemmas zu identifizieren und geeignete, relationenspezifische Theorien zu formulieren. Eine Berücksichtigung numerischer Konzepte wie in [McNe89] findet hingegen nicht statt.

7.2 Die Beispielverfeinerung

Um einen Überblick über die Gestalt der für DBPL-Programmentwicklungen typischerweise erzeugten Lemmas zu erhalten, wird eine Beispielverfeinerung entwickelt, die die für Datenbankprogramme üblichen Einfüge-, Lösch- und Änderungsoperationen enthält. Das Beispiel stellt eine Erweiterung der in [BMSW89] angegebenen Verfeinerung dar. Insgesamt werden je zwei Einfüge-, Lösch- und Änderungsoperationen formuliert. Die die Verfeinerung konstituierende Folge von abstrakten Maschinen und die zugehörigen Kontextdefinitionen (vgl. Anhang C) sowie die im Hinblick auf die Implementation in DBPL jeweils getroffenen Entwurfsentscheidungen werden in den folgenden Abschnitten beschrieben. Anhand des Beispiels wird anschließend herausgearbeitet, in welcher Weise die Eigenschaften der Entwurfssprache TDL sowie der Zielsprache DBPL die Gestalt der Folge von abstrakten Maschinen beeinflussen und in welcher Form die im relationalen Datenbankmodell nicht inhärenten Integritätsbedingungen in der Verfeinerung berücksichtigt werden.

7.2.1 Die Spezifikation

Die in Anhang C.1.2 angegebene Spezifikationsmaschine beschreibt die Gegebenheiten einer Miniwelt, für die Beziehungen zwischen Forschungsprojekten, beteiligten wissenschaftlichen Instituten und den beschäftigten Mitarbeitern verwaltet werden sollen. Die zugehörigen abstrakten Mengen `PROJECTS`, `COMPANIES` und `EMPLOYEES` werden (neben weiteren Attributtypmengen) im zugehörigen Kontext `ExtResearchRefCtx` definiert. Die Zustandsvariablen, die die jeweils aktuellen Entitätenausprägungen enthalten, sind vom entsprechenden Potenzmengentyp, z.B.

```
companies: POW(COMPANIES);
```

Die Entitätenattribute werden durch totale Funktionen von den aktuellen Entitätenmengen in den entsprechenden Attributtyp modelliert:

...

[5] Der Beweis läßt sich jedoch nur *teilweise* anhand der angegebenen Theorien *nachvollziehen*.

```
compName: companies -> CompNames;
engagedIn: companies -> POW(projects);
budget: companies -> MONEY ;
    ...
```

Die Einschränkung des *Bild*bereichs mancher Attributfunktionen auf *aktuelle* Klassenausprägungen der Entitätenmengen

```
engagedIn: companies -> POW(projects)

consortium: projects -> POW(companies)

worksOn: employees -> POW(projects)
```

wird sich im Laufe der Verfeinerung in Maßnahmen zur Wahrung der in relationalen Datenbankbeschreibungen nicht modellinhärenten *referentiellen Integrität* niederschlagen.
Die in der Miniwelt über die Typzugehörigkeit der einzelnen Zustandsvariablen hinaus geltenden Beziehungen werden durch *zusätzliche Invariantenprädikate* erfaßt: für die Entitätenmengen `projects` und `companies` sind jeweils bereits eine *Teilmenge* der zugehörigen Klassenattribute *identifizierend*, eine Eigenschaft, die während der weiteren Verfeinerung im Hinblick auf die in der relationalen Sprache DBPL gegebenen *wertorientierten* Objektidentifikation zur Festlegung von *Schlüsselattributen* bedeutsam sein wird:

```
!x.!y.(x,y: projects =>
   (projName(x) = projName(y)  &                    (S1)
   getsGrantFrom(x) = getsGrantFrom(y)
    => x = y ))

!x.!y.(x,y: companies =>
   (compName(x) = compName(y)                       (S2)
    => x = y ))
```

Darüberhinaus sind zwei *klassenübergreifende* Integritätsprädikate **I1** und **I2** formuliert:

```
 !x.(x: employees =>
       worksOn(x) incl engagedIn(belongsTo(x)))     (I1)

  !x.(x: projects =>
       consortium(x) = {y | y:  companies & x : engagedIn(y)})     (I2)
```

I1 besagt, daß jeder Angestellte lediglich an Projekten arbeitet, an denen dessen Forschungsinstitut partizipiert (eine erweiterte, "semantische" Form der referentiellen Integrität)[6]. **I2** legt

[6]Als (realitätsnahe) Entwurfsentscheidung fließt ein, daß jeder Mitarbeiter lediglich *einem* Institut angehört.

eine naheliegende Beziehung zwischen Projekten und Forschungsinstituten fest: die Menge der dem Konsortium eines Projekts angehörigen Institute ist gleich der Menge der Institute, die in diesem Projekt engagiert sind.
Als Beispiel für eine Operationsspezifikation diene die Operation `removeProject`,

```
removeProject(proj) =
 PRE
   proj : projects &
   consortium(proj) = {}
 THEN
   projects:= projects - {proj} ||
   projName:= {proj} <<|  projName ||
   getsGrantFrom:=  {proj} <<| getsGrantFrom ||
   consortium:= {proj} <<| consortium
 END
```

die das *Löschen* eines Projektes aus der Entitätenmenge `projects` beschreibt. Für den Erhalt der Systeminvariante (der, wie besprochen, durch die Verifikation einer Beweisverpflichtung sicherzustellen ist) ist von Bedeutung, daß nicht nur die Zustandsvariable `projects`, sondern auch die Attributvariablen mit Domäne `projects` eine entsprechende Veränderung erfahren. Der mathematische Operator "`<<|`" steht für eine sog. *Domänensubtraktion*, bei der der Definitionsbereich einer Funktion um die in der als erstes Argument angegebenen Menge enthaltenen Elemente verringert wird[7]. Die in der Vorbedingung der Substitution angegebene Bedingung

```
consortium(proj) = {}
```

schränkt die Anwendbarkeit der Operation derart ein, daß lediglich solche Projekte gelöscht werden dürfen, an denen keine Institution mehr arbeitet. Dies stellt den Erhalt der Integritätsbedingung **I2** sicher.
Die Einfügeoperationen sind von spezieller Gestalt: die Wahl einer "frischen" Entität wird auf dieser Stufe über unbeschränkten Nichtdeterminismus abgebildet, ist jedoch im Laufe der Verfeinerung durch implementierbare Konzepte zu ersetzen (vgl. auch [Wetz90], [BMSW89]).

7.2.2 1. Verfeinerung: Datenidentifikation

Im ersten Verfeinerungsschritt werden die durch die Invariantenprädikate S1 und S2 formulierten Eindeutigkeitsbeziehungen für die Ausprägungen `projects` und `companies` ausgenutzt, um zu der für die zu entwickelnde DBPL-Implementierung notwendigen *wertorientierten* (assoziativen) Identifikation der Datenbankextensionen zu gelangen. Dies wird erreicht, indem man die Kombination der die Ausprägungen identifizierenden Attributfunktionsvariablen mit der Identitätsfunktion über den identifizierten Klassenvariablen gleichsetzt: seien

[7] Diese *Semantik* enthält das *Symbol* "`<<|`" allerdings erst indirekt durch geeignete Axiomatisierung in den für die Lemmabeweise entwickelten Theorien.

$$\begin{aligned} f_1 &: M \longrightarrow A_1 \\ &\vdots \\ f_k &: M \longrightarrow A_k \end{aligned}$$

die Attributfunktionen, die gemeinsam die Ausprägungen der Menge M identifizieren. Dann besitzen die Ausprägungen $x \in M$ die Gestalt

$$x \;=\; (f_1(x), \ldots, f_k(x))$$

Diese Beziehung wird durch eine entsprechende *Definition* in der ASSERTION-Klausel der ersten Verfeinerungsmaschine formuliert, in dem hier betrachteten Beispiel

```
ASN

  compName := %x.(x : companies | x);

  projName := %x.(x : projects & x = (a,b) | a);
  getsGrantFrom := %x.(x : projects & x = (a,b) | b);
  projId :=  %x.(x : projects | x)
```

Da die `projects`-Ausprägungen durch mehrere Attribute identifiziert werden, wird zusätzlich ein "zusammengesetzter" Identifikator `projId` eingeführt.
In der Kontextdefinition resultieren diese Entwurfsentscheidungen in entsprechenden Vereinbarungen für die zugehörigen Basismengen:

```
COMPANIES = CompNames;
PROJECTS = ProjNames * Agencies
    ...
```

Für die Ausprägungen der Klasse `EMPLOYEES` wird ein "künstliches" identifizierendes Attribut eingeführt: der zugehörige Kontext vereinbart eine neue, abstrakte Attributmenge `EmpIds`, mit der die Menge `EMPLOYEES` identifiziert wird:

```
EMPLOYEES = EmpIds;
```

Im ASSERTION-Paragraph wird eine entsprechende neue Attributfunktion als Identität über der Ausprägungsmenge `employees` *definiert*:[8]

```
empId := %x.(x : employees | x);
```

[8]Die Notation `%x.( P | T)` ist die ASCII-Umschreibung einer Funktionsdefinition durch "Lambda-Abstraktion" ("`%`" steht für "λ").

Semantisch unterscheiden sich Definitionen von in *Invarianten*klauseln angegebenen Gleichheitsprädikaten durch ihre *Asymmetrie* sowie durch die unterschiedliche Bedeutung für Beweisverpflichtungen: existieren Definitionen für die Zustandsvariablen einer Maschine, so sind bei der Bearbeitung der jeweiligen Beweisverpflichtungen zunächst die durch die Definitionen vorgeschriebenen Ersetzungen (ggf. rekursiv) vorzunehmen, da die Operationssubstitutionen auf den in den rechten Seiten der Definitionen enthaltenen Variablen arbeiten. Andernfalls werden zum Teil inkorrekte mathematische und Typlemmas erzeugt. Durch Definitionen ausgedrückte Beziehungen sind *garantiert*. Invariantengleichungen hingegen dürfen nicht von den Operationen verletzt werden; deren Erhalt wird in den Beweisverpflichtungen verifiziert.
Die Entwurfsentscheidung der Datenidentifikation bewirkt Operationsänderungen lediglich bei den *Einfüge*operationen. Für die Operation `newCompany` wird die Wahl eines "frischen" Elements durch unbeschränkten Nichtdeterminismus überflüssig, da bereits die einschränkende Vorbedingung

```
newname : CompNames - ran(compName)
```

zusammen mit der für die Klasse `COMPANIES` definierten Identifizierung bedingt, daß ein neues Element als Parameter gegeben ist.
Für die Operation `newEmployee` kann dieser Weg wegen der Identifizierung durch ein "künstliches" Attribut nicht beschritten werden. Stattdessen wird davon ausgegangen, daß systemseitig eine Funktion zur Verfügung steht, die die benötigten "frischen" Identifikatoren für die Klasse `EMPLOYEES` generiert und in Form einer *Konstante* `newEmpId` der jeweiligen aufrufenden Operation zur Verfügung stellt[9].

7.2.3 2. Verfeinerung: Datenstrukturierung

Der zweite Verfeinerungsschritt beinhaltet die Festlegung der in relationalen Datenbankbeschreibungen üblichen *Datenstrukturen*. Wie in Abschnitt 6.1.4 besprochen, lassen sich Relationenvariable als *Abbildungen* von der Menge der *Schlüssel*attributkombinationen der aktuellen Ausprägungen auf das *kartesische Produkt* über den Attributtypmengen[10] der weiteren Attribute darstellen. Demgemäß erfolgt eine *Datenverfeinerung*, in der entsprechend typisierte[11] Zustandsvariablen eingeführt werden:

```
compClass : companies  ->  POW (projects) * MONEY;
empClass : employees  ->  EmpNames * companies * POW (projects);
projClass : projects  ->  POW (companies);
    ...
```

[9] Im Laufe der weiteren Verfeinerung könnte eine solche Funktion in einer externen, importierten abstrakten Maschine unabhängig realisiert werden.

[10] die auf dieser Ebene der Verfeinerung noch dynamischen Änderungen unterworfen sein können, da manche dieser Mengen von Zustandsvariablen abhängen

[11] Wenn im folgenden von *Typisierung* gesprochen wird, so ist darunter i.a. die Vereinbarung einer "mathematischen" Elementbeziehung zu verstehen, die nicht den Einschränkungen einer restriktiven programmiersprachlichen Typisierung unterliegt.

In zusätzlichen Definitionen werden die Beziehungen der bisherigen zu den neuen Variablen festgelegt; für die Klasse `COMPANIES` beispielsweise

```
   ...
engagedIn := %x.(x : companies & compClass(x) = (a,b) | a);
budget := %x.(x : companies & compClass(x) = (a,b) | b);
   ...
```

Die bisherigen Extensionsvariablen werden durch die Definitionsbereiche (Domänen) der eingeführten Funktionsvariablen definiert:

```
companies := dom(compClass);
employees := dom(empClass);
projects := dom(projClass)
```

Die Operationen arbeiten nun auf den *konkreten* Zustandsvariablen[12]. Daher vereinfacht sich beispielsweise der Rumpf der bedingten Substitution der oben vorgestellten Operation `removeProject` auf die einzelne Substitution

```
projClass:= {proj} <<| projClass
```

Da die Attributfunktionen der Klasse `PROJECTS` nunmehr "per definitionem" denselben Definitionsbereich wie die Funktion `projClass` besitzen, sind keine weiteren Zuweisungen notwendig. Als weitere Entwurfsentscheidung des zweiten Verfeinerungsschritts erfolgt ein Übergang von parallelen zu sequentiellen Substitutionen, was allerdings lediglich bei den beiden Einfügeoperationen zu Änderungen führt.

7.2.4 3. Verfeinerung: Datentypisierung

Die im vorangehenden Verfeinerungsschritt durchgeführte Strukturierung der zu verwaltenden Daten ist noch nicht unmittelbar in DBPL-Code umzusetzen. Die Extensionsvariablen sind bisher als *totale* Funktionen über *zeitlich varianten* Zustandsvariablen definiert. Die in DBPL vorgeschriebene *statische Typisierung* bedingt den Übergang zu einer Typisierung der Extensionsvariablen als *partielle* Funktionen über *zeitlich invarianten* Basismengen. Zu diesem Zweck wird der Kontext durch die Festlegung weiterer Basismengen erweitert. Für die unterschiedlichen Klassen werden die zur Definition des relationalen Datenbankschemas benötigten Recordtypen sowie allgemeine, statisch definierte Projektionsfunktionen für die einzelnen Recordkomponenten vereinbart. Der zugehörige *Relationen*typ expliziert die Schlüsseleigenschaft durch seine Gestalt als nunmehr partielle, statisch typisierte Funktion. Für die Klasse `COMPANIES` beispielsweise wird der Kontext um die Festlegungen

[12] hierunter sind jeweils die Variablen der aktuell betrachteten verfeinerten Maschine zu verstehen

```
CompRecType = CompNames * ProjIdRelType * MONEY;
CompName = %x.(x:CompRecType & x = (a,b,c) | a);
EngagedIn = %x.(x:CompRecType & x = (a,b,c) | b);
Budget = %x.(x:CompRecType & x = (a,b,c) | c);
CompRelType = CompNames +-> ProjIdRelType * MONEY;
```

erweitert.
Für die Datenbankextensionen werden Variable der entsprechenden Relationentypen vereinbart und (entsprechend einer Datenverfeinerung) mit den Extensionsvariablen der vorherigen abstrakten Maschine identifiziert.

```
compRel : CompRelType;
compRel = compClass;
```

Die (bisher partiell definierten) Attributfunktionen werden mit Hilfe einer Definition aus den im Kontext vereinbarten, statisch typisierten Projektionsfunktionen abgeleitet:

```
compName := compRel <| CompName;
engagedIn := compRel <| EngagedIn;
budget := compRel <| Budget ;
```

Das mathematische Symbol "<|" steht für eine Domänenrestriktion einer Funktion durch die vorstehende Menge[13].
Zusätzliche abgeleitete Variable beschreiben die in DBPL vorherrschende *tupelorientierte* Sicht der Datenbankextensionen und damit die letztlich im Datenbankschema zu vereinbarenden Relationen:

```
compRelDes := %x.(x : dom(compRel) & compRel(x) = (a,b) |(x,a,b));
empRelDes := %x.(x : dom(empRel) & empRel(x) = (a,b,c) |(x,a,b,c));
projRelDes := %x.(x : dom(projRel) & projRel(x) = a |(x,a));
```

Die als Transaktionen zu realisierenden *Operationen* werden dahingehend modifiziert, daß die in den Präkonditionen formulierten "mathematischen" Parametertypisierungen durch schwächere *statische Typprädikate* ersetzt werden, die sich direkt in DBPL umsetzen lassen. Die dynamisch zu überprüfenden Typprädikate werden jeweils in Form einer bedingten Substitution zur Laufzeit kontrolliert. Die serialisierten Elementarsubstitutionen lassen sich ebenfalls direkt in DBPL umsetzen. Die direkte Umsetzbarkeit der problematisch erscheinenden Substitution

```
...
projRel:= projRel<+(%x.(x : ventures | projRel(x) \/ {newname}));
...
```

in der Operation `newCompany` wird in Abschnitt 7.3.4 gezeigt.

[13]Die Funktion `compRel` wird an dieser Stelle als Tupelmenge, d.h. als Teilmenge des kartesischen Produkts ihrer Urbild- und Bildmengen aufgefaßt.

7.3 Entwurfs- und Zielsprachenabhängigkeit der Verfeinerung

Die *Charakteristika von Entwurfs- und Zielsprache* beeinflussen unmittelbar die Gestalt der Verfeinerungen[14] und damit das Aussehen der zu erwartenden Lemmas. Anhand des erweiterten Beispiels soll daher, als Grundlage für die Lemmaklassifikation im folgenden Abschnitt, der Einfluß der DAIDA-Sprachen TDL und DBPL herausgearbeitet werden. In einem weiteren Abschnitt wird diskutiert, in welcher Weise die im relationalen Datenbankmodell nicht modellinhärenten Integritätsbedingungen im Laufe der Verfeinerung abgebildet werden.

7.3.1 Abhängigkeit von der Entwurfssprache TDL

Die Gestalt der *Spezifikations*maschine ist abhängig von der Art ihrer Erstellung. Im obigen Beispiel weist die Initialmaschine die Merkmale einer durch automatische Transformation aus einer TDL-Systembeschreibung gewonnenen Maschine auf: für die "generellsten" Klassen der TDL-Klassenhierarchien werden abstrakte Basismengen im Kontext der Spezifikationsmaschine formuliert; Klassenausprägungen werden in entsprechenden mengenwertigen Zustandsvariablen erfaßt, und Klassenattribute werden als Funktionsvariable, die auf den Klassenausprägungen definiert sind, beschrieben. Die UNIQUE-Klauseln der Klassendefinitionen werden in Form von Invariantenprädikaten **S1** und **S2** abgebildet. Zusätzliche, klassenübergreifende Invarianten, die in TDL unter INVARIANT-Klauseln formuliert sind, resultieren in zusätzlichen Invariantenprädikaten **I1** und **I2**.
In [Wetz90] wird eine Menge *formaler Transformationsregeln* von TDL in die Notation abstrakter Maschinen sowie eine Implementation derselben auf dem B-Tool beschrieben. Für einen Teil des im letzten Abschnitt beschriebenen Beispiels[15] wird eine TDL-Spezifikation formuliert; die Spezifikationsmaschine entsteht durch automatische Transformation unter Verwendung des B-Tools. Da die entsprechende B-Tool-Regelsammlung die erweiterte Syntax der abstrakten Maschinen nicht unterstützt, wurde das erweiterte Beispiel "von Hand" (*orientiert* an den in [Wetz90] angegebenen Regeln) formuliert.
Aus der Betrachtung der Beziehungen zwischen einer TDL-Spezifikation und der "entsprechenden" *Spezifikations*maschine lassen sich Implikationen für die zum Zweck der Lemmabeweise zu erstellenden Theorien ableiten. Unabhängig von der konkret spezifizierten Datenbankanwendung existiert ein *durch die* TDL-*Transformationsregeln festgelegtes Basisrepertoire mathematischer Symbole*, welches in der Spezifikationsmaschine verwendet wird:

- in der *Invariante*: Elementbeziehung "`:`"zur Typisierung[16], Potenzmengenbildung "`POW`" zur Typisierung der Klassenausprägungsvariablen, *funktionale Typisierung* der Attributvariablen, generische Gestalt der aus den UNIQUE-Klauseln resultierenden Invariantenprädikate.
- in den *Operationsbeschreibungen* (als Auswirkung der in der Invariante festgelegten Zustandsvariablentypisierung):

[14] Unter *Verfeinerung* soll in diesem Kontext die formulierte *Folge* abstrakter Maschinen verstanden werden.
[15] dieselben Objektklassen, jedoch lediglich die Einfüge-Operation *NewEmployee*
[16] Unter *Typisierung* ist hier wiederum eine mathematische "Typfestlegung" in Form einer Elementbeziehung zu verstehen, vgl. o. .

- Löschoperationen: *Domänensubtraktion* "<<|", *Mengendifferenzbildung* "-"
- Einfügeoperationen: *funktionales Überschreiben* "<+", *Mengenvereinigung* "\/"
- Änderungsoperationen: *funktionales Überschreiben* "<+"

Da die Gestalt der Invariantenprädikate und Operationen der *Spezifikations*maschine somit hauptsächlich durch die TDL-Transformationsregeln festgelegt wird, ist zu erwarten, daß die unten formulierten Lemmabeweisregeln die Vervollständigung der *Konsistenz*beweise für eine breite Klasse von Datenbankanwendungen ermöglichen. Zusätzliche Regeln werden benötigt, falls zur Formulierung der TDL-Klasseninvarianten spezielle mathematische Operatoren verwendet werden[17].

7.3.2 Abhängigkeit von der Zielsprache DBPL

In Form von Datenverfeinerungen werden schrittweise die von DBPL unterstützten Datenstrukturen (kartesische Produkte, statische Typisierung) eingeführt. Die Gestalt der *Verfeinerungs*maschinen ist *zielsprachenabhängig*:

- In den CHANGE-Paragraphen werden relational typisierte Zustandsvariablen eingeführt und mittels Definitionen zu den bisherigen Zustandsvariablen in Beziehung gesetzt.
- Die Operationen arbeiten jeweils auf den "konkreten" Zustandsvariablen, d.h. in der zweiten und dritten Verfeinerung auf relational strukturierten Variablen.

Das Aussehen der *Verfeinerungs*beweisverpflichtungen wird somit (mit voranschreitender Verfeinerung) zunehmend durch die Charakteristika der relationalen Zielsprache beeinflußt. Es sind Regeln zu formulieren, die die Semantik der einzuführenden relationalen Notation festlegen; u.a. für

- kartesische Produkte "*"
- tupelorientierte Funktionsdarstellung: eine Funktion wird als Menge von Tupeln aus dem kartesischen Produkt aller Domänen- und Bildattributmengen aufgefaßt. Einfüge-Operationen werden in Form einer Vereinigung der Funktion mit der Menge, die nur das "neue" Tupel enthält, dargestellt[18].
- Domänenrestriktionen "<|", mittels derer dynamisch typisierte totale Funktionen mit den statisch typisierten partiellen Funktionen des letzten Verfeinerungsschrittes in Beziehung gesetzt werden

[17] beispielsweise arithmetische Operatoren wie Summe, Produkt usw.

[18] Diese Darstellung ist unpräzise, da in einer derartigen Produktdarstellung keine Trennung zwischen den identifizierenden, d.h. die Domäne bildenden Attribute und den abhängigen, d.h. die Bildmenge konstituierenden Attributen vorgenommen wird. In künftigen Untersuchungen sollte dieses Problem durch die Einführung weiterer Klammerungen beseitigt werden.

Für die den Beweis der Verfeinerungslemmas unterstützenden Theorien ist daher, ebenso wie für die Theorien zum Nachweis der Konsistenzlemmas, eine breite Verwendbarkeit, *unabhängig* von der betrachteten *Datenbank*anwendung[19], zu erwarten. Diese Aussage bezieht sich auf Standardoperationen allgemeiner Gestalt. Die Probleme, die von Operationen komplexer Gestalt[20] resultieren, werden weiter unten diskutiert.

7.3.3 Abbildung der "relationalen" Integritätsbedingungen

In Abschnitt 6.1.5 wurden die für das relationale Datenbankmodell nicht implizit gewährleisteten Integritätsbedingungen identifiziert, deren Erhalt durch die Transaktionen zu gewährleisten ist. Es soll nun diskutiert werden, in welcher Weise hierdurch die Gestalt der in der Verfeinerung aufgestellten abstrakten Maschinen beeinflußt wird.
Es wird sich zeigen, daß der Erhalt derartiger Integritätsbedingungen in Gestalt von *Beweisen entsprechender Lemmas* sichergestellt wird.

7.3.3.1 Referentielle Integrität

In den frühen Stufen der Verfeinerung werden die Klassenattribute in Form von totalen Funktionen über *zeitlich varianten Entitätenmengen* verwaltet, z.B.

```
engagedIn: companies -> POW(projects);
```

Der Erhalt der referentiellen Integrität wird durch die Respektierung dieses Typprädikats durch die Operationen sichergestellt. Zu diesem Zweck erhält jede Operation eine Präkondition, die die Ausführbarkeit auf diejenigen Parameterwerte beschränkt, für die die Invariantenprädikate, d.h. insbesondere auch die Typprädikate erhalten werden. Im Falle der Operation `newCompany` hat die Präkondition die Gestalt

```
c <-- newCompany(newname,ventures,budg) =
  PRE
    newname : CompNames - ran(compName)  &
    ventures : POW(projects)  &
    budg : MONEY
   THEN
      ...
```

d.h. nur in den Fällen, in denen die für das neue Forschungsinstitut zu vereinbarende Projektpartizipation `ventures` eine Teilmenge der *aktuellen* Extension `projects` von `PROJECTS` ist, kann

[19] Wenn oben von Anwendungs*abhängigkeit* der entstehenden Lemmas gesprochen wurde, so war darunter insbes. die Abhängigkeit von der gewählten *Ziel*sprache zun verstehen. Die Aussage im Text bezieht sich allerdings auf den Fall, in dem die Zielsprache DBPL *fixiert* ist.

[20] In ihrer Eigenschaft als *Transaktionen* werden Operationen insbes. dann komplex, wenn die im Invariantenparagraphen formulierten, zu erhaltenden Integritätsbedingungen nichttrivialer Gestalt sind.

die Operation ausgeführt werden. Dies stellt den Erhalt der durch das Typprädikat formulierten referentiellen Integrität sicher.
Im letzten Verfeinerungsschritt werden die Operationen und Datenstrukturen in eine Form gebracht, die die Übersetzung nach DBPL ermöglicht. Da die Präkonditionen der Operationen durch statisch überprüfbare Parametertypisierungen abgebildet werden sollen, werden die den Erhalt der referentiellen Integrität sicherstellenden stärkeren Bedingungen in eine dynamisch zu überprüfende Bedingung einer *konditionalen Substitution* hineinentwickelt:

```
c <-- newCompany(newname,ventures,budg) =
 PRE
   newname : CompNames &
   ventures : ProjIdRelType &
   budg : MONEY
 THEN
   IF
     newname : CompNames - ran(compName) &
     ventures : POW(projects)
   THEN
    ...
```

Durch die zugehörige Verfeinerungsbeweisverpflichtung wird sichergestellt, daß diese Operation die zugehörige Spezifikation tatsächlich realisiert.
Es bleibt festzuhalten, daß der Erhalt der im Relationenmodell nicht implizit garantierten referentiellen Integrität

- durch die nichtstatische Typisierung in der Spezifikationsmaschine *gefordert*,
- in Form von Beweisverpflichtungen für die einzelnen Operationen *überwacht*,
- im Rahmen des während der Verfeinerung zu vollziehenden Übergangs zu statischen Typisierungen durch eine Einführung konditionaler Substitutionen *sichergestellt*

wird. Ist der Erhalt der referentiellen Integrität durch die Transaktionen nicht gewährleistet, so werden Lemmas erzeugt, die *nicht beweisbar* sind.

7.3.3.2 Schlüsselintegrität

Die relationale Verwaltung der Entitäten in der Zielsprache DBPL bedingt die Festlegung assoziativer Selektoren für jede Objektklasse. Im Beispiel geschieht dies im ersten Verfeinerungsschritt, wobei die aus der UNIQUE-Vereinbarung in TDL resultierenden Invariantenprädikate **S1** und **S2** ausgenutzt werden (vgl. o.). Die Schlüsselintegrität wird gewährleistet, da die in der Spezifikationsmaschine getroffenen Typvereinbarungen eine funktionale Abhängigkeit der Attribute von der Extensionsmenge der zugehörigen Klasse festlegen und die Extensionsmengen in der

ersten Verfeinerung mit den Schlüsselattributen identifiziert werden, denn für die "kritischen" Operationstypen bedeutet dies:

- *Einfügeoperationen* wurden (mit Hilfe von unbeschränktem Nichtdeterminismus) derart *spezifiziert*, daß jeweils ein "frisches" Element zu wählen ist. Nach der Identifikation der Extensionsmengen mit den Schlüsselattributen müssen die verfeinerten Operationen derart formuliert werden, daß die Schlüsselintegrität erhalten wird. Andernfalls werden nicht erfüllbare Verfeinerungsbeweisverpflichtungen erzeugt[21].

- *Änderungsoperationen* haben jeweils einen (oder mehrere) Parameter, die die zu modifizierenden Extensionen bestimmen. Eine Modifikation der identifizierenden Attribute führt wegen der festgelegten Datenidentifikation zu einer fehlerhaften Selbstmodifikation und wiederum zu nicht beweisbaren Lemmas[22].

Durch die Entwurfsentscheidung "Datenidentifikation" des ersten Verfeinerungsschritts wird der Erhalt der Schlüsselintegrität äquivalent zu den in der Invariante der Spezifikationsmaschine formulierten funktionalen Abhängigkeiten aller Entitätenattribute von den Entitäten. Die in der Verfeinerung entwickelten Operationen müssen die Schlüsselintegrität demnach respektieren; andernfalls können nicht alle Lemmas bewiesen werden.

7.3.4 Implementierbarkeit verfeinerter Operationen

Die bisher diskutierten Kriterien für die Umsetzbarkeit von Operationen in die Zielsprache DBPL betrafen lediglich die durch das relationale Datenbankmodell auferlegten Einschränkungen für die Daten*strukturierung* sowie die stets zu erreichende *statische* Datentypisierung. Derlei Zielvorgaben dürften sich von denen anderer relationaler Zielsprachen nicht groß unterscheiden. DBPL verfügt jedoch über Konzepte, die sie gegenüber anderen relationalen Sprachen erweitert:

- Die Möglichkeit der Vereinbarung nichtflacher Relationen erlaubt die *unmittelbare* Umsetzung von Verfeinerungen, die für andere Zielsprachen weiter konkretisiert werden müßten. Im Beispiel verfügen die drei Entitätenklassen jeweils über nichtflache Attribute, die sich in DBPL direkt implementieren lassen.

- In Abschnitt 6.1.4 wurden die hochsprachlichen, deklarativen Sprachelemente von DBPL vorgestellt. In Verbindung mit den in DBPL realisierten NF^2-Strukturen resultiert dies in der direkten Realisierbarkeit komplexer Substitutionen, für die im Falle herkömmlicher relationaler Zielsprachen Iterationen eingeführt werden müssen. Die folgenden Unterabschnitte widmen sich diesem Punkt.

[21] Der Fehler ließe sich in Form eines (oder mehrerer) nicht beweisbaren Lemmas genau lokalisieren.

[22] Die ausschließlich für die Klasse `EMPLOYEES` formulierten Änderungsoperationen des Beispiels sind für diese Betrachtungen ungeeignet, da in diesem Fall ein künstliches Identifikationsattribut eingeführt wurde und solche Probleme nicht auftreten können.

7.3.4.1 Ein Beispiel für die Mächtigkeit von DBPL

Die Operation `newCompany` besitzt eine komplizierte dritte Verfeinerung (vgl. Anhang C.4.2):

```
c <-- newCompany(newname,ventures,budg) =
    PRE
      newname : CompNames &
      ventures : ProjIdRelType &
      budg : MONEY

    THEN
      IF
        newname : CompNames - ran(compName) &
        ventures : POW(projects)
      THEN
        projRel:= projRel<+(%x.(x : ventures | projRel(x) \/ {newname}));
        compRel:= compRel \/ {(newname,ventures,budg)} ;
        c:= newname
      ELSE
        c:= nilCompName
      END
    END
```

Die Schwierigkeiten resultieren aus der allgemeinen Gestalt der Operation: Für das neueinzufügende Forschungsinstitut wird bereits eine Menge von Projekten angegeben, an denen es partizipiert. Um die Integritätsbedingung **I2** zu erhalten, ist eine entsprechende Änderung der `consortium`-Attribute aller Projekte in `ventures` vorzunehmen: jedes Projekt erhält das (durch seinen Namen identifizierte) Institut als neues `consortium`-Mitglied:

```
...
projRel:= projRel<+(%x.(x : ventures | projRel(x) \/ {newname}));
..
```

Es zeigt sich, daß die Operation `newCompany` trotz dieser komplexen Zuweisung *ohne* explizite Formulierung einer Iteration über den Elementen von `ventures`, d.h. ausschließlich unter Verwendung eines Zugriffsausdrucks in DBPL umgesetzt werden kann[23]:

```
projRel:& ProjRelType{{x.ProjId, {newname,
                                  EACH y IN x.Consortium:TRUE}}
                                OF EACH x IN ventures : TRUE}
```

[23]Die Beispielspezifikation wurde in der Tat zunächst ohne Rücksicht auf eine möglichst einfache Umsetzbarkeit nach DBPL formuliert. Es zeigte sich dann bei Betrachtung der Verfeinerung, daß sich alle beschriebenen Operationen ohne Verwendung expliziter Iterationen in DBPL "deklarativ" realisieren lassen.

Der formulierte Zugriffsausdruck bildet die Menge aller Paare[24] von Projektidentifikatoren aus der Menge `ventures` und Institutsmengen, wobei letztere aus den bisherigen `consortium`-Mengen der Projekte durch Hinzufügen des neu partizipierenden Instituts entsteht. Durch Verwendung des Zugriffsausdrucks in einer DBPL-Änderungszuweisung wird somit genau die oben angegebene, mathematisch als funktionales Überschreiben formulierte Zuweisung umgesetzt.

7.3.4.2 Konsequenzen

Das Beispiel stützt die These, daß für die meisten Operationen eine *Explizierung* von Iterationen während der Verfeinerung unnötig ist. Damit wird die Generierung sowie das Erbringen der Beweisverpflichtungen vereinfacht, da die Schleifensubstitution eine komplizierte Semantik hat (vgl. o.). Dies scheint die Vorteilhaftigkeit deskriptiver, hochsprachlicher Zielsprachen zu belegen, was jedoch durch folgende Punkte relativiert wird:

- Deklarative Sprachelemente mit einem hohen Grad an Parallelität, wie sie in DBPL gegeben sind, stellen eine alternative ("implizite") Formulierungsmöglichkeit für Schleifen dar, die die direkte Umsetzung mächtiger mathematischer Formulierungen der Notation der abstrakten Maschinen erlauben. Da die Übersetzung der Verfeinerung in die Zielsprachennotation automatisch[25] geschehen soll, müssen zusätzliche Anforderungen an die Gestalt der zu implementierenden abstrakten Maschine gestellt werden, da andernfalls die Übersetzungsregeln zu komplex werden.

- In [ERMS90] wird gezeigt, daß die in DBPL zur Verfügung stehenden deklarativen Sprachkonstrukte sogar mächtig genug sind, bestimmte NP-vollständige Entscheidungsprobleme zu lösen. Eine Regelsammlung, die in DBPL deklarativ realisierbare *Einzel*substitutionen ohne restriktive Einschränkungen nach DBPL übersetzt, realisiert daher im Hinblick auf diese Mächtigkeit ein eingeschränktes *automatisches Programmieren*.

Es erscheint daher sinnvoll, sich bei der Festlegung einer geeigneten Transformationsregelmenge auf eine restriktive Teilmenge mathematischer Notationen in abstrakten Maschinen zu beschränken. Auch die von den Transformationsregeln generierten DBPL-Konstrukte sind sinnvoll einzuschränken. Die Abbildungsregeln sollten *elementarer und rein syntaktischer Gestalt* sein. Die Identifikation geigneter Regelmengen sollte ebenfalls anhand einer größeren Fallstudie geschehen, in der u.a. auch die *Implementierbarkeit* von Operationen, d.h. die Voraussetzung für deren Umsetzbarkeit in die Zielsprache DBPL untersucht wird[26]. Die zur Verfeinerung des obigen Beispiels verwendeten standardisierten Entwurfsentscheidungen sind dann um *weitere Verfeinerungsschritte* zu ergänzen, innerhalb derer u.a. ein Übergang von der oben verwendeten λ-Abstraktion zu den in DBPL verwendeten *Mengenausdrücken* stattfindet.

[24] Dies verdeutlicht die unterschiedliche, kontextabhängige Semantik der geschweiften Klammern in DBPL: zum Bilden von *Tupeln* sowie zum Bilden von *Mengen*.

[25] oder zumindest *regelbasiert*, d.h. in Form von Abbildungsschritten allgemeiner Gestalt

[26] Der hier verwendete Implementierbarkeitsbegriff unterscheidet sich von dem in der Literatur im Kontext prädikativer Spezifikationen ([HeGM86]) verwendeten: dort ist von Implementierbarkeit i.S. von *Berechenbarkeit* die Rede; die hier verwendete Bezeichnung orientiert sich hingegen an den *restriktiven Gegebenheiten einer Zielsprache*.

Die Komplexität der Übersetzungsregeln wird somit reduziert, indem die Arbeit in die zu verifizierenden Verfeinerungsschritte verlagert wird. Anstelle vieler, in ihrer Anwendung spezialisierter Übersetzungsregeln werden spezielle *Theorien* benötigt, die den Beweis der Lemmas ermöglichen, die aus den zusätzlichen Verfeinerungsschritten resultieren.

7.4 Realisierung des Definitionskonzepts

In den *Verfeinerungs*maschinen des in Abschnitt 7.2 vorgestellten Beispiels werden unter den ASSERTION-Paragraphen sog. *Definitionen* formuliert, in denen Zustandsvariable als abhängig von weiteren Zustandsvariablen vereinbart werden (vgl. die Beschreibung der Semantik von Definitionen in Abschnitt 7.2.2). Die Operationen der Verfeinerungsmaschinen arbeiten auf den in den rechten Seiten der Definitionen enthaltenen "definierenden" Variablen. Die Theoriesammlung `devB` berücksichtigt Definitionen bei der Verarbeitung der Beweisverpflichtungen *nicht* und mußte daher adaptiert werden.
Als Beispiel diene die Operation `removeEmployee`, die in der ersten Verfeinerung von der Gestalt

```
removeEmployee(empl) =
 PRE
   empl : employees
 THEN
   employees:= employees - {empl} ||
   empName:= {empl} <<| empName ||
   belongsTo:= {empl} <<| belongsTo ||
   worksOn:= {empl} <<| worksOn
 END
```

ist. Beim Übergang zur zweiten Verfeinerung erfolgt eine Datenstrukturierung für die Objektklasse `EMPLOYEES`, in deren Rahmen die Attributfuktionen `empName`, `belongsTo` und `worksOn` als *abhängig* von der neu eingeführten, relational strukturierten Variablen `empClass` definiert werden

```
empName := %x. (x : employees & empClass(x) = (a,b,c) | a);
belongsTo := %x. (x : employees & empClass(x) = (a,b,c) | b);
worksOn := %x. (x : employees & empClass(x) = (a,b,c) | c);
```

und demnach keiner expliziten Zuweisung mehr bedürfen:

```
removeEmployee(empl) =
  PRE
    empl : employees
  THEN
    empClass:= {empl} <<| empClass
  END
```

Zur Erfüllung der Verfeinerungsbeweisverpflichtung ist nun (u.a.) zu zeigen, daß die CHANGE-Beziehung

```
belongsTo_2 = belongsTo_3
```

unter der simultanen Ausführung der abstrakten und der konkreten Version der Operation invariant ist.
Würde die Definitionsbeziehung außer Acht gelassen, so wäre eine Gleichheitsbeziehung der Gestalt

```
{empl} <<| belongsTo_2 = belongsTo_3
```

nachzuweisen[27], da die konkrete Operation auf `empClass` arbeitet und `belongsTo` nur indirekt modifiziert. Werden jedoch vor "Anwendung" der Operationen (d.h. Ausführen der Substitutionen) zunächst (rekursiv) die vereinbarten Definitionen expandiert, erhält man die korrekte Form des zu erbringenden Nachweises:

```
{empl} <<| belongsTo_2 = %x. (x : dom({empl} <<| empClass_3)
                              & ({empl} <<| empClass_3)(x) = (a,b,c) | b);
```

Die *Realisierung des Definitionskonzepts* geschieht durch

- Einführung der speziellen Notation ":="; die Regelsammlung `devB` wurde derart modifiziert, daß die unter den ASSERTION-Paragraphen formulierten Definitionen in gesonderten Theorien (`GetDefinitionX` und `GetInvDefinitionX`) als *Gleichungen* abgelegt werden[28].
- Änderung der zur Verarbeitung der *Verfeinerungsbeweisverpflichtungen* verwendeten Taktiken.

Die Taktik zur Verarbeitung der Verfeinerungsbeweisverpflichtungen wird in der auch die Regelsammlungen für die Lemmabeweise enthaltenden *Beweismethode* (vgl. Anhang F) abgeändert: unter der REFTAC-Klausel wird die Taktik

```
(DED;RefinementLemmaTacticsX)
```

vereinbart. `RefinementLemmaTacticsX` ist eine in der Beweismethode formulierte Theorie, die mittels `bcall` eine vom jeweiligen Typ des Verfeinerungslemmas abhängige Taktik zur Verfügung stellt[29]:

[27] Die Gleichung entsteht aus obigem CHANGE-Prädikat durch Anwendung der in den abstrakten und konkreten Versionen der Operation formulierten Substitutionen.
[28] Die Details der Implementierung sind nicht instruktiv und wird daher nicht näher besprochen.
[29] vgl. die von `devB` vorgesehene, in Abschnitt 6.2.2.3 beschriebene Taktik

- für die Verfeinerungsbeweisverpflichtungen gewöhnlicher Operationen referenziert die Taktik (u.a.) die "Anwendung" der als Gleichungen gespeicherten Definitionen:

```
    bwrite("\n processing Refinement Proof Obligation \n ")
  &
    bcall((RefinementLemmaX;DED;(PrepareConjectureX;REV;
           (PrepareConjectureX;ARI)~;FLAT)~;GetOperationX~;
           GetInitialisationX~;TransCondX~;GetInvariantX;
           GetDefinitionX~;
           BuildConjectureX;FLAT~;(((GEN;DED)~;CalculusX)~;
           MathTypeLemmaGenerationX)~)
        : (Check_refinement: M) )
=>
    Check_refinement: M
```

  Die Generierung der Lemmas wird in der Theorie `MathTypeLemmaGenerationX` (wieder über `bcall`) initiiert. Durch die dort verwendete Taktik (die Definitionen werden, sofern möglich, resubstituiert) wird die Erzeugung einiger, trivialerweise erfüllter Typlemmas im Unterschied zu dem in [Wetz90] verfolgten Weg vermieden.

- Da bei der Formulierung der *Initialisierungs*operationen der Verfeinerungen die Existenz von Definitionen nicht berücksichtigt wurde, ist in den entsprechenden Beweisverpflichtungen auf die Anwendung der Definitionsgleichungen zu verzichten[30]. Daher wird in diesem Fall die von `devB` vorgesehene Taktik verwendet.

Die Taktik, gemäß der die *Konsistenz*beweisverpflichtungen verarbeitet werden, wurde nicht modifiziert, da die in der DAIDA-Umgebung anhand der TDL-Abbildungsregeln generierten Spezifikationsmaschinen keine Definitionen enthalten. Eine entsprechende Änderung ist jedoch analog möglich.
Als unerwünschter Nebeneffekt generiert `devB` ASSERTION-Lemmas für die als ASSERTIONs formulierten Definitionen. Die Beweismethode enthält eine Theorie `AsnDefLemma`, die den sofortigen Nachweis der aus den Definitionen erzeugten Lemmas ermöglicht (vgl. Anhang F).

7.5 Klassifikation der Lemmas

In Abschnitt 6.2.2.3 wurde beschrieben, in welcher Weise die Theoriensammlung `devB` Beweisverpflichtungen verarbeitet. Hier soll nun am Beispiel der oben vorgestellten DAIDA-Verfeinerung

- die Herkunft der Lemmas aus den unterschiedlichen Beweisverpflichtungen lokalisiert,
- die genaue Gestalt der Lemmas besprochen,
- bestimmte *generische* Klassen von Lemmas als Basis für die zu erstellenden Theorien identifiziert

[30] Dieser (bereits in [Wetz90] verfolgte) Weg stellt einen *Detaillfehler* dar: die Definitionen sollten auch die *Initialisierung* abhängiger Variablen überflüssig machen.

werden.
Die Anhänge E.1 und E.2 zeigen die mathematischen bzw. Typlemmas, aufgegliedert nach den Ebenen der Vefeinerung. Wegen der unterschiedlichen Gestalt der Beweisverpflichtungen von Spezifikation und Verfeinerung (vgl. Abschnitte 6.1.3.5 und 6.1.3.9) werden die dort entstehenden Lemmas im folgenden jeweils in getrennten Abschnitten beschrieben.

7.5.1 Beziehung der Lemmas zu den Beweisverpflichtungen

Es soll nun zunächst, aufbauend auf den Betrachtungen von Abschnitt 6.2.2.3, die genaue Herkunft der einzelnen Lemmas aus den unterschiedlichen Beweisverpflichtungen aufgezeigt werden. Hierzu werden Beispiele diskutiert. Es wird sich zeigen, daß sich die entstehenden Lemmas bis auf wenige Sonderfälle jeweils einer bestimmten Operation *und einem bestimmten* INVARIANT- *bzw.* CHANGE-*Prädikat* zuordnen lassen.
Aufbauend hierauf nimmt der nachfolgende Abschnitt eine Klassifizierung der Lemmas vor.

7.5.1.1 Lemmas der Konsistenzbeweisverpflichtungen

Die Diskussion in Abschnitt 6.2.2.3 hat gezeigt, daß die Konsistenzbeweisverpflichtungen der Operationen und der Initialisierung (Beziehungen 6.4 bzw. 6.3) durch die Regeln der Theorie `BuildConjectureX` aufgestellt werden:

```
[K]P => chkinv(P,K);

[K]P => chkinv(P,(Q | K));
```

bzw.

```
[K]P => chkinvini(P,K);
```

wobei `K` durch die die jeweilige Operation konstituierende verallgemeinerte Substitution und `P` durch das Konjunkt der Systeminvarianten der Spezifikationsmaschine gebunden wird. In der zweiten Regel des Operationskonsistenznachweises wird `Q` durch eine Operationspräkondition gebunden. Da diese (gemäß der Arbeitsweise von `devB`) automatisch als Hypothese der hieraus erzeugten Lemmas generiert wird, erscheint `Q` *nicht* auf der linken Regelseite. Auf diese Weise werden (gemäß Beziehung 6.4) außerdem die CTX-, INV- und ASN-Vereinbarungen der betrachteten Spezifikationsmaschine berücksichtigt .
Für die Operationen der *Spezifikations*maschine des Beispiels wird `K` jeweils durch *elementare* parallele Substitutionen gebunden, die nicht mehr unter Anwendung der in Theorie `CalculusX` (vgl. Abschnitt 6.2.2.3) definierten Gleichungen zu vereinfachen sind. Die Substitution wird daher unmittelbar (unter Anwendung der eingebauten B-Tool-Regel SUB) ausgeführt.
Eine Aufstellung aller Lemmas des Konsistenznachweises findet sich in den Anhängen E.1.1 und E.2.1.

Die Verarbeitung der Beweisverpflichtungen und die Lemmagenerierung soll nun anhand eines Beispiels diskutiert werden. Anhang G.1 zeigt die Verarbeitung der Konsistenzbeweisverpflichtung der Operation `removeProject` : nach Generierung der Beweisverpflichtung in Beweisschritt 135 (Regel `BuildConjectureX.2`) wird die entstehende parallele Substitution in Schritt 134 unmittelbar ausgeführt. Das hieraus entstehende Prädikat entspricht der Invariante bis auf diejenigen Konjunkte, deren Variablen durch die Operationssubstitution

```
projName_1,consortium_1,getsGrantFrom_1,projects_1
  :=
{proj}<<|projName_1,{proj}<<|consortium_1,
{proj}<<|getsGrantFrom_1,projects_1-{proj}
```

modifiziert werden. Die Anwendung der eingebauten Regel AND zerlegt das aus der Substitution resultierende Konjunkt in seine Komponentenprädikate (Schritt 133), wobei die durch die Substitution nicht veränderten Invariantenprädikate unmittelbar auf die Hypothese reduziert werden können, da diese die Invariante *vor* Operationsausführung enthält. Die auf diese Weise *nicht* nachweisbaren Konjunkte sind:

```
engagedIn_1 : companies_1 --> POW(projects_1-{proj})

projects_1-{proj}: POW(PROJECTS)

{proj}<<|projName_1: projects_1-{proj} --> ProjNames

{proj}<<|getsGrantFrom_1: projects_1-{proj} --> Agencies

{proj}<<|consortium_1: projects_1-{proj} --> POW(companies_1)

worksOn_1: employees_1 --> POW(projects_1-{proj})

!x.!y.(x,y: projects_1-{proj}
        =>
       (({proj}<<|projName_1)(x) = ({proj}<<|projName_1)(y) &
        ({proj}<<|getsGrantFrom_1)(x) = ({proj}<<|getsGrantFrom_1)(y)
          => x = y))

!x.(x: projects_1-{proj} =>
        ({proj}<<|consortium_1)(x) =
        {y | y: companies_1 & x: engagedIn_1(y)})
```

Diese werden getrennt weiterverarbeitet (Schritte 28, 42, 56, 70, 84, 98, 116, 132). Da für deren Beweis an dieser Stelle keine geeigneten Theorien und Taktiken zur Vefügung stehen, entstehen hieraus acht Lemmas (sechs Typ- und zwei mathematische Lemmas).
Es soll nun exemplarisch die genaue Gestalt von zwei der hier entstehenden Lemmas diskutiert

werden. Die übrigen Konjunkte führen zur Generierung entsprechender Lemmas.
Aus dem dritten Konjunkt resultiert das *Typ*lemma Nr. 15

```
ctx(ExtResearchCompaniesMch) &
inv(ExtResearchCompaniesMch_1) &
pre(ExtResearchCompaniesMchremoveProject_1(proj)) &
asn(ExtResearchCompaniesMch_1)
=>
{proj}<<|projName_1: projects_1-{proj} --> ProjNames
```

welches den Nachweis verlangt, daß die Typbeziehung der Variablen `projName_1` auch *nach* Ausführung der Operation `removeProject`, d.h. für die nunmehr modifizierten Variablen gilt. In diesem Fall ist der Beweis einfach; er bedarf lediglich einer Beweisregel, der die Semantik von Domänensubtraktionen "`<<|`" auf totalen Funktionen beschreibt (vgl. u.).
Aus dem siebten Konjunkt entsteht das *mathematische* Lemma Nr. 5:

```
ctx(ExtResearchCompaniesMch) &
inv(ExtResearchCompaniesMch_1) &
pre(ExtResearchCompaniesMchremoveProject_1(proj)) &
asn(ExtResearchCompaniesMch_1) &
x,y: projects_1-{proj} &
({proj}<<|projName_1)(x) = ({proj}<<|projName_1)(y) &
({proj}<<|getsGrantFrom_1)(x) = ({proj}<<|getsGrantFrom_1)(y)
=>
x = y
```

Es ist nachzuweisen, daß die Eindeutigkeitsbeziehung **S1** auch für die nach Entfernen des Elements `proj` entstehende Projektmenge `projects_1-{proj}` gilt. Ein Beweis dieses (offensichtlich erfüllten) Lemmas bedarf ebenfalls einer speziellen Beweisregel in den zu entwickelnden Theorien, die weiter unten besprochen werden sollen.
Die Verarbeitung der Konsistenznachweise der übrigen Operationen sowie deren Lemmaerzeugung verläuft analog.
Die Beweisverpflichtung der *Initialisierungs*operation führt zu vier mathematischen und acht Typlemmas, die aus der Substitution der Initialwerte der Zustandsvariablen in die zwölf Invariantenprädikate resultieren. Das mathematische Lemma Nr. 2

```
ctx(ExtResearchCompaniesMch) &
x,y: {} &
{}(x) = {}(y) & {}(x) = {}(y)
=>
  x = y
```

entsteht so z.B. durch Anwendung der Initialsubstitution auf das Invariantenprädikat **S1**. Es ist zu zeigen, daß die Eindeutigkeitsbeziehung der Klasse `Projects` nach Ausführung der Initialisierungsoperation gilt, was wegen der widersprüchlichen Voraussetzung `x,y: {}` trivialerweise der Fall ist. Typlemma Nr. 4

```
ctx(ExtResearchCompaniesMch)
=>
{}: {} --> CompNames
```

entsteht durch Substitution in die Typbeziehung

```
compName: companies --> CompNames
```

der Invariante. Nachzuweisen ist die (triviale) Beziehung, daß die leere Funktion eine totale Funktion von der leeren Menge in die Bildmenge `CompNames` ist. Der Beweis dieser Lemmas wird durch die Angabe einfacher Regeln ermöglicht (vgl. u.).

7.5.1.2 Lemmas der Verfeinerungsbeweisverpflichtungen

Die Generierung der in Abschnitt 6.1.3.9 abgeleiteten *Verfeinerungs*beweisverpflichtungen 6.9 und 6.10 geschieht durch die Regeln

```
[A]not([B]not(P)) => chkinv(P,A,B);

[A]not([B]not(P)) => chkinv(P,A,(Q | B))
```

der Theorie `BuildConjectureX`. `P` wird durch das die Beziehungen zwischen "abstrakten" und "konkreten" Variablen[31] herstellende CHANGE-Prädikat[32] gebunden; die syntaktischen Variablen `A` und `B` werden jeweils mit einem Paar (zusammengehöriger) konkreter bzw. abstrakter Initialisierungs- oder Operationssubstitutionen instantiiert. Eine durch `Q` gebundene Präkondition der *abstrakten* Operation wird ignoriert, da die Präkondition der *Spezifikation* der betrachteten Operation jeweils bereits in der Hypothese steht[33]. Gemäß den Ausführungen in Abschnitt 6.1.3.10 wird die Terminierungsvoraussetzung $trm(S)$ der Beweisverpflichtung 6.10 durch die Präkondition der zugehörigen *Spezifikations*operation[34] ersetzt.
Zu beachten ist, daß für Operationen mit Ergebnis (im Beispiel die *Einfüge*operationen) ein Konjunkt $x = y$ zu `P` gehört, welches die Resultate der abstrakten und konkreten Operation gleichsetzt.
Die Lemmagenerierung verläuft entsprechend dem Verfahren für die Konsistenzbeweisverpflichtungen, wobei allerdings vor Auswertung der Substitutionskonstrukte eventuell vereinbarte *Definitionen*, wie in Abschnitt 7.4 beschrieben, in `P` expandiert werden. Wegen der Form des durch Substitutionsanwendungen abzuleitenden Prädikats

[31] Die Bezeichnungen "*abstrakt*" und "*konkret*" beziehen sich in dem hier besprochenen Zusammenhang auf *Paare* von "Objekten" (Variablen, Operationen, Maschinen) zweier aufeinanderfolgender Verfeinerungsebenen, die in einer "Verfeinerungsbeziehung" zueinander stehen.

[32] wobei die automatisch generierten Gleichheiten $var_i = var_{i+1}$ zwischen während des Verfeinerungsschritts unverändert gebliebenen Variablen berücksichtigt werden (vgl. Kapitel 6)

[33] Die folgende Diskussion wird aufzeigen, welche Rolle der Präkondition der *konkreten* Operation zukommt.

[34] d.h. der in der Spezifikationsmaschine formulierten abstraktesten Version der betrachteten Operation

```
[A]not([B]not(P))
```

wird zunächst die abstrakte und dann die konkrete Substitution angewandt. Für den Fall, daß `B` elementar ist, gilt

```
not([B]not(P)) = not(not([B]P))
```

eine Beziehung, die durch Anwendung von Regel 45 der Theorie `CalculusX`

```
M => not(not(M))
```

ausgenutzt werden kann[35].
Aufbauend auf diesen allgemeinen Vorbetrachtungen soll nun als Beispiel zunächst die Verarbeitung der Beweisverpflichtung der ersten Verfeinerung der Operation `removeProject` betrachtet werden. Es ist das durch den Term

```
[proj: projects_2 & consortium_2(proj) = {} |
    consortium_2,projects_2
  :=
    {proj}<<|consortium_2,projects_2-{proj}]
     not([projName_1,consortium_1,
            getsGrantFrom_1,projects_1
           :=
            {proj}<<|projName_1,{proj}<<|consortium_1,
            {proj}<<|getsGrantFrom_1,projects_1-{proj}]not P
```

definierte Prädikat zu berechnen, wobei `P` für das (definitionsexpandierte) CHANGE-Prädikat der ersten Verfeinerungsmaschine steht. Die abstrakte Substitution ist direkt durchführbar: es entsteht der Term

```
[proj: projects_2 & consortium_2(proj) = {} |
    consortium_2,projects_2
  :=
    {proj}<<|consortium_2,projects_2-{proj}]not(not(Q))
```

wobei `Q` das nach Anwendung der abstrakten Substitution aus `P` entstehende Prädikat ist. Im folgenden Schritt wird (Regel `CalculusX.37`) die Definition

```
[I]R & P => [P | I]R
```

[35] Hier wird nochmals die Semantik des Terms $\neg([Sub]\neg(P)$ partiell bestätigt: er unterscheidet sich von $[Sub]P$ nur in den Fällen, in denen Sub nichtdeterministisch ist oder nicht immer terminiert.

der bedingten Substitution angewandt; man erhält die zwei Teilziele

```
 [consortium_2,projects_2
:=
 {proj}<<|consortium_2,projects_2-{proj}]not(not(Q))
```

und

```
proj: projects_2 & consortium_2(proj) = {}
```

Ersteres Teilziel wird nach Anwendung von Regel `CalculusX.45` und SUB, wie für Konsistenzbeweise bekannt, weiterverarbeitet. Es entspricht dem Teil

$$B \wedge D \wedge R \wedge I \wedge trm(S) \quad \Rightarrow \quad [T]\neg([S]\neg(R \wedge (x = y))$$

der zu erbringenden Beweisverpflichtung 6.10. Letzteres Teilziel ist gleich der Präkondition der *konkreten* Operation, welche unter den Voraussetzungen `CTX`, `CHG` und `ASN` der konkreten Maschine sowie der Präkondition der zugehörigen Operation der Spezifikationsmaschine zu beweisen ist. Da Termination mit der Gültigkeit der Operationspräkonditionen identifiziert wird, entspricht dies gerade dem zweiten Teil

$$B \wedge D \wedge R \wedge I \wedge trm(S) \quad \Rightarrow \quad trm(T)$$

der zu erbringenden Beweisverpflichtung, d.h. auch der *Terminationsnachweis* wird durch die generierte Beweisverpflichtung erbracht.
Aus der Weiterverarbeitung der beiden Teilziele resultieren für die erste Verfeinerung der Operation `removeProject` die mathematischen Lemmas Nr. 14 und 15. Beide entstehen aus dem erstgenannten Teilziel. Die in der Hypothese stehende Präkondition der Operations*spezifikation* stimmt mit der der ersten Verfeinerung überein; daher ist die Reduktion des zweiten Teilziels schon während der Beweisverpflichtungs*verarbeitung* durchführbar und es werden keine Lemmas generiert.
Das mathematische Lemma Nr. 14 entsteht aus der Gleichung

```
getsGrantFrom _1 = getsGrantFrom_2
```

welche nach Anwendung der Definition für `getsGrantFrom_2` und der Operationssubstitutionen zu

```
{proj} <<| getsGrantFrom _1 = %x.(x: projects_2-{proj} & x = a,b | b)
```

wird. Nach weiteren Gleichheitsanwendungen[36] gemäß den obenbeschriebenen Taktiken wird hieraus das Lemma

[36] die die Reduktion der durch die Substitutionsanwendungen unverändert gebliebenen CHG-Gleichungen erlaubt und dadurch die Erzeugung vieler "trivialer" Lemmas vermeidet

```
  ctx(ExtResearchCompaniesMch) &
  inv(ExtResearchCompaniesMch_2) &
  pre(ExtResearchCompaniesMchremoveProject_1(proj)) &
  asn(ExtResearchCompaniesMch_2)
=>
  {proj}<<|%x.(x: projects_2 & x = a,b | b)
 =
  %x.(x: projects_2-{proj} & x = a,b | b)
```

erzeugt.
Das mathematische Lemma Nr. 15 entsteht entsprechend für die Attributfunktion `projName`.
Das Beispiel zeigt, daß man mittels der angewendeten Taktik (vgl. Abschnitt 7.4) schon einige Teilziele trivialer Gestalt eliminieren kann. Hierunter fallen insbesondere die CHANGE-Gleichungen derjenigen Variablen, die durch die Substitution nicht tangiert werden[37].
Die Verarbeitung der Beweisverpflichtungen der *zweiten* Verfeinerung verläuft entsprechend und soll daher nicht gesondert diskutiert werden.
Aufgrund der Einführung *konditionaler* Substitutionen und der Substitutions*serialisierung* ist die Generierung der Lemmas des *dritten* Verfeinerungsschritts relativ komplex. U.a. werden die Regeln 35, 38 und 39 der Theorie `CalculusX` zur Umformung verwendet:

```
    [I][J]R   =>   [I;J]R
              ...
  P => [S]R   =>   [P ==> S]R
 [S]R & [T]R  =>   [S [] T]R
```

Die Gestalt der generierten Lemmas ist dementsprechend (vgl. u.).
Es bleibt festzuhalten:

- Für *algorithmische Verfeinerungsschritte* entstehen fast ausschließlich *mathematische Lemmas*, da in den CHANGE-Prädikaten keine Typvereinbarungen formuliert werden
- *Typlemmas* entstehen
  - im Falle von *Datenverfeinerungen* für die neu eingeführten konkreten Variablen
  - während *algorithmischer Verfeinerungen* nur dann, falls die Präkondition der konkreten Operation nicht direkt auf die Präkondition der abstrakten Operation zurückführbar ist. In der Beispielverfeinerung geschieht dies z.B. für die Operation `newCompany` (Typlemma Nr. 36). Derartige Lemmas dienen somit dem *relativen Terminationsnachweis*.

In DAIDA-Programmentwicklungen finden *Daten*verfeinerungen für die oben verwendeten standardisierten Entwurfsentscheidungen lediglich im Rahmen der relationalen Datenstrukturierung

[37] In einem in Abschnitt 8.3 vorgestellten weiterführenden Projekt werden derartige Trivialziele ebenfalls identifiziert und anschaulich als *inertia* (*"Trägheit"*) bezeichnet.

während des zweiten und dritten Verfeinerungsschritts statt. Daher werden für das Beispiel (insgesamt) 76 Typlemmas, jedoch 125 mathematische Lemmas erzeugt. Wegen der standardisierten Gestalt der im Laufe der Verfeinerung einzuführenden strukturierten Zustandsvariablen ist es dennoch möglich und sinnvoll, die entstehenden Typlemmas generisch zu klassifizieren.

7.5.2 Klassifikation der Lemmas gemäß ihrer generischen Struktur

Nachdem nun die Herkunft der Lemmas aus den Beweisverpflichtungen anhand von Beispielen erörtert wurde, soll in einem weiteren Schritt eine genaue Klassifikation vorgenommen werden, die als Basis für die zu erstellende Theoriensammlung dienen wird.
Der folgende Unterabschnitt beschreibt die Vorgehensweise. In den nachfolgenden zwei Unterabschnitten wird die Klassifikation, getrennt für Lemmas der Konsistenz- und Verfeinerungsbeweisverpflichtungen, erarbeitet. Abschließend wird kurz diskutiert, inwieweit die Klasseneinteilung als *vollständig* zu bezeichnen ist.

7.5.2.1 Vorgehensweise

Die Ausführungen in Abschnitt 7.5.1 haben die zwischen den INVARIANT- bzw. CHANGE-Prädikaten und den Lemmas bestehenden Beziehungen erarbeitet: es zeigte sich, daß die meisten der Lemmas einer *bestimmten* Operation und einem *bestimmten* Prädikat zugeschrieben werden können. Es wurde deutlich, daß aus Verfeinerungsbeweisverpflichtungen hauptsächlich *mathematische* Lemmas resultieren.
Im Rahmen der nun vorzunehmenden Lemmaklassifikation wird ein *abstrakterer Blickwinkel* eingenommen: es wird sich zeigen, daß die meisten der Lemmas in Abhängigkeit des Operations*typs* (Einfügen, Löschen, Ändern) eingeordnet werden können, sofern die Operation, aus der das Lemma resultiert, von "gewöhnlicher", *elementarer* Gestalt ist. Das oben vorgestellte Verfeinerungsbeispiel ist für die folgenden Betrachtungen eine gute Grundlage, da es von den drei Operationsgrundtypen jeweils zwei Operationen beinhaltet. Der Begriff der *elementaren Operation* soll hier nicht formal definiert werden. Von den Operationen des Beispiels fällt lediglich `newCompany` aus dem Rahmen, da zum Erhalt der Integritätsbedingung `I2` eine (relativ komplexe) Substitution auf der Attributfunktion `consortium` der Klasse `Projects` vorgenommen wird. Die übrigen Operationen sind als elementar in obigem Sinne zu bezeichnen.
Die folgenden Abschnitte erarbeiten eine derartige Klassifikation jeweils getrennt für die mathematischen und Typlemmas, entsprechend der in Abbildung 7.1 gezeigten Grobeinordnung.
Die Klasseneinteilung wird, ausgehend von den für die *Beispiel*verfeinerung erzeugten Lemmas, erarbeitet. Wegen der Anzahl der sich ergebenden Lemmaklassen wird die Klassifikation jeweils nur *exemplarisch* dargestellt. Dies geschieht i.a. durch die *Identifikation einer generischen Darstellung*, mittels der sich die Lemmas einer Klasse umschreiben lassen.
Das Beispiel dient demnach als *Ausgangspunkt* für die Klassifikation, anhand dessen sich geeignete Klassifikations*kriterien* ableiten lassen. Aufbauend hierauf werden weitere Lemmaklassen identifiziert, für die in der Beispielverfeinerung keine "Repräsentanten" auftreten. Die erarbeitete Klassifikation ist daher als *allgemein* zu bezeichnen. Derartige *Vollständigkeitsaspekte* werden in einem gesonderten Unterabschnitt diskutiert.

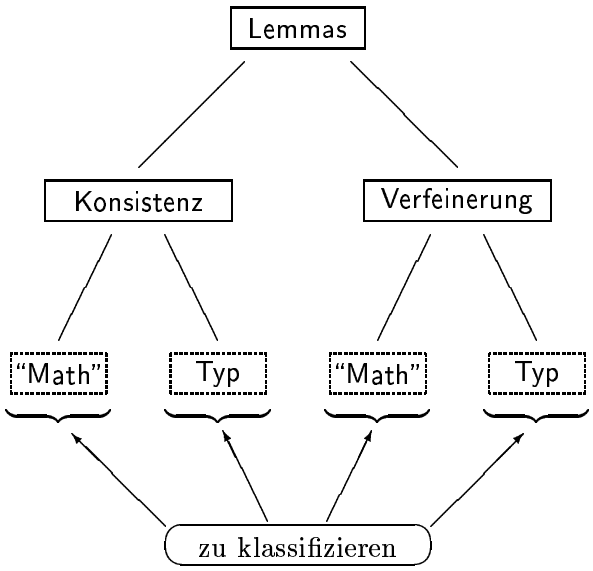


Abbildung 7.1: Grobstruktur der Lemmaklassifikation

7.5.2.2 Lemmas der Konsistenzbeweisverpflichtungen

7.5.2.2.1 Typlemmas Es werden nun zunächst die *Typlemmas* des Konsistenznachweises betrachtet. Aufgrund der standardisierten Gestalt der mittels fester Übersetzungsvorschriften aus einer TDL-Beschreibung gewonnenen Spezifikationsmaschine werden sich diese als allgemein klassifizierbar erweisen.
Als Beispiel diene das Typlemma Nr. 24 der Beispielverfeinerung, welches sich als *charakteristisch für Einfüge-Operationen* erweisen wird:

```
  ctx(ExtResearchCompaniesMch) &
  inv(ExtResearchCompaniesMch_1) &
  pre(c <-- ExtResearchCompaniesMchnewCompany_1(newname,ventures,budg)) &
  asn(ExtResearchCompaniesMch_1) &
  co: COMPANIES-companies_1
=>
  compName_1<+{co|->newname}: companies_1\/{co} --> CompNames
```

Es entsteht durch die Zuweisung eines Attributwerts der neueinzufügenden Ausprägung an eine zugehörige Attributvariable, in diesem Fall `compName_1`[38]. Da die TDL-Abbildungsregeln die Typisierung der Attributfunktionen als funktionenwertige Zustandsvariable festlegen, sind Lemmas dieser Gestalt auch für Einfügeoperationen auf anderen Entitätenmengen zu erwarten. Wie erwartet, läßt sich dies am Beispiel belegen: die Typlemmas 24, 25, 26, 30, 31, und 32 sind von dieser Form.
Auch für die Operationstypen "Löschen" und "Ändern" ist anhand der Beispiels zu beobachten,

[38]Die in der Operationsbeschreibung gebräuchliche Schreibweise `compName(co):= newname` wird als *funktionales Überschreiben* "`<+`" interpretiert.

daß die Typlemmas, unabhängig von den im speziellen betrachteten Operationen dieses Typs, von ihrer Struktur her in wenige Klassen einteilbar sind. Dies liegt allgemein daran, daß die TDL-Abbildungsregeln die Variablentypisierung der initialen abstrakten Maschine *strukturell festlegen*[39]; die auf diese Weise erhaltene *funktionale Sicht der Daten* ist für jede TDL-Klasse schematisch identisch.
Als weiteres Beispiel soll Typlemma Nr. 19 betrachtet werden; es beschreibt die für den Zustand nach Ausführung der *Lösch*operation `removeEmployee` nachzuweisende Typbeziehung der Ausprägungsvariable `employees`:

```
  ctx(ExtResearchCompaniesMch) &
  inv(ExtResearchCompaniesMch_1) &
  pre(ExtResearchCompaniesMchremoveEmployee_1(empl)) &
  asn(ExtResearchCompaniesMch_1)
 =>
  employees_1-{empl}: POW(EMPLOYEES)
```

Ein von der Struktur her identisches Lemma ergibt sich für die Löschoperation `removeProject` (Typlemma Nr. 14).
Es soll nun am Beispiel des zuvor diskutierten Typlemmas Nr. 24 konkretisiert werden, welche Ähnlichkeit i.a. zwischen "strukturell identischen" Lemmas besteht. Im betrachteten Fall lassen sich die Lemmas der Klasse, in die Typlemma Nr. 24 eingeordnet wird, *generisch darstellen* als

```
  ctx(M) &
  inv(M) &
  pre(R <-- O(P)) &
  asn(M) &
  N: W - S
=>
  F<+{N |-> B}: S\/{N} --> T
```

wobei den Großbuchstaben die intuitive Rolle von "syntaktischen Variablen" zukommt[40]. Diese Darstellung kann noch präzisiert werden, wenn man bestimmte Prädikate der Maschineninvarianten und der Operationspräkondition im Antezedenten des Lemmas expliziert:

- die Invariante enthält die Typisierung von `F` im Zustand *vor* der Operationsausführung
- die Operationspräkondition enthält eine Typvereinbarung für den einzufügenden Attributwert `B`:

Die generische Darstellung konkretisiert sich zu

```
  F : S --> T &
```

[39] i.S.v. Typisierungsschemata wie z.B. `attrvar: class_ext --> attrtype`
[40] hier zunächst *ohne* Bezug zu BL

```
  N: W - S &
  B : U &
   ...
=>
  F<+{N |-> B}: S\/{N} --> T
```

Dies gibt erste Anhaltspunkte für die beim Beweis zu lösenden Probleme: eine *Regel*, die einen Teilaspekt der Semantik des funktionalen Überschreibens korrekt erfasst, ist

```
  F : S --> T &
  N: W - S &
  B : T
=>
  F<+{N |-> B}: S\/{N} --> T
```

Für den Fall, in dem `U` in der Operationspräkondition dem Funktionsbereichstyp `T` entspricht, können die durch Anwendung der Regel entstehenden Teilziele unmittelbar auf die Hypothese zurückgeführt werden (Lemmas 25, 26, 30, 31). Andernfalls ist noch der Nachweis zu erbringen, daß sich die Beziehung `B : T` auf die Präkondition `B : U` zurückführen läßt, d.h. es ist zu zeigen, daß $\texttt{U} \subseteq \texttt{T}$ gilt. Die Regelsammlung, die weiter unten in ihrer Gesamtheit diskutiert wird, hat also außer der zuvor beschriebenen Regel weitere allgemeine und elementare Beziehungen der Mengentheorie, in diesem Fall Inklusionsbeziehungen, zu beinhalten.
Für die übrigen Fälle lassen sich die generischen Darstellungen analog formulieren.
Tabelle 7.1 faßt die auf diese Weise erhaltenen unterschiedlichen generischen Klassen von Typlemmas, gegliedert nach den unterschiedlichen Operationstypen, zusammen. In der Spalte "Herkunft" wird angegeben, auf welche Art die Typlemmas einer Klasse aus den Typprädikaten der Invariante entstehen. Zu beachten ist hierbei, daß *Änderungs*operationen lediglich Typlemmas der angegebenen Klasse erzeugen können[42], da eine Änderung nie die Ausprägungsmenge manipuliert. Derartige Aspekte werden in Abschnitt 7.5.2.4 besprochen.
Das oben diskutierte Typlemma Nr. 24 ist somit ein Repräsentant der Klasse **TE2**.

7.5.2.2.2 Mathematische Lemmas Die *mathematischen Lemmas* der Konsistenzbeweisverpflichtungen lassen sich i.a. *nicht* mittels derart einfacher generischer Schablonen einordnen. Der Grund liegt darin, daß diese Lemmas aus den Nicht-Typprädikaten der Maschineninvarianten resultieren. Diese wiederum entstehen gemäß den TDL-Transformationsvorschriften aus den TDL-Klasseninvarianten der INVARIANT-Klassenparagraphen sowie den UNIQUE-Klassenattributen. Für erstere ist prinzipiell keine einfache generische Struktur zu erwarten, da an dieser Stelle Invariantenprädikate formuliert werden, die die Domäne einer *speziellen* Datenbankanwendung[43] modellieren, wie im Beispiel die Integritätsbedingungen **I1** und **I2**.

[41] Für die Lemmas dieser Klasse existiert keine einfache generische Darstellung wie im Falle des im Text gegebenen Beispiels. Sie zerfällt genaugenommen in mehrere Unterklassen entsprechend der genauen Gestalt des Attributfunktionenbilds.

[42] sofern sie korrekt formuliert sind

[43] Wenn oben von Lemmas *anwendungsspezifischer* Gestalt gesprochen wurde, dann bezog sich dies auf die spezifische *Klasse* von DAIDA-Verfeinerungen. An dieser Stelle sind spezielle Lemmas, die für eine spezifische Anwendung innerhalb des "fixierten" DAIDA-Rahmens resultieren, gemeint (eine weitere Ebene der Spezialisierung).

Operationstyp	Klasse	Lemma-Nr.	Herkunft
INITIALISIERUNG	**TI**	1,..,12	Initialisierung beliebiger Zustandsvariablen
LÖSCHEN	**TL1**	14, 19	Entfernen aus der Ausprägungsmenge
	TL2	15, 16, 17, 20, 21, 22	Restriktion der Attributfunktionen
	TL3[41]	13, 18	Subtraktion in Attributfunktionenbild
EINFÜGEN	**TE1**	23, 29	Einfügen in die Ausprägungsmenge
	TE2	24, 25, 26, 30, 31, 32	Erweiterung der Attributfunktionen
	TE3[41]	27, 28	Einfügen in Attributfunktionenbild
ÄNDERN	**TÄ**	33, 34, 35	Überschreiben von Attributfunktionen

Tabelle 7.1: Klassifikation der Typlemmas des Konsistenznachweises

Ein Lösungsansatz für diese Art von Teilbeweisverpflichtungen bestünde darin, eine möglichst allgemeine Regelsammlung zu entwerfen, die darüberhinaus den Beweis auch solcher Lemmas ermöglicht. Für die im Beispiel beschriebene Anwendung genügt die Aufstellung allgemeiner Theorien für die "üblichen" mengentheoretischen Konzepte sowie speziellerer Axiome für λ-Abstraktionen, Funktionskomposition usw. Schränkt man die möglichen Klasseninvarianten nicht durch die Vorgabe eines eingeschränkten Satzes mathematischer Operatoren ein, so muß ggf. eine *anwendungsabhängige* Erweiterung der gegebenen Axiomatisierung vorgenommen werden, d.h. in diesem Fall kann man zwangsläufig keine Vollständigkeit der Regelsammlung erreichen.
Es bietet sich somit folgendes Vorgehen an:

- Axiomatisierung bestimmter mathematischer Basistheorien, die in den meisten Fällen benötigt werden.
- Die Vorgabe einer Theoriesammlung für *spezielle Anwendungsbereiche*, die vom Benutzer ggf. hinzugezogen werden oder sogar erweitert werden können.

Es ist zu erwarten, daß hieraus i.a. ein Effizienzgewinn resultiert, da die benötigten Beweisregeln *selektiv* eingesetzt werden[44].
Für die aus den UNIQUE-Prädikaten resultierenden und damit DAIDA-üblichen mathematischen Lemmas kann allerdings eine einfache generische Klassifizierung vorgenommen werden. Die Einfüge-Operationen führen (im Fall einer UNIQUE-Auszeichnung von Klassenattributen der entsprechenden TDL-Klasse) zu mathematischen Lemmas der Gestalt wie z.B.

[44]Im Rahmen eines in Abschnitt 8.3 diskutierten weiterführenden Projekts wird eine derartige Trennung in anwendungsunabhängiges und anwendungsspezifisches "Wissen" ebenfalls vorgenommen (vgl. u.).

```
  ctx(ExtResearchCcompaniesMch) &
  inv(ExtResearchCompaniesMch_1) &
  pre(c <-- ExtResearchCompaniesMchnewCompany_1(newname,ventures,budg)) &
  asn(ExtResearchCompaniesMch_1) &
  co: COMPANIES-companies_1 &
  x,y: companies_1\/{co} &
  (compName_1<+{co|->newname})(x) = (compName_1<+{co|->newname})(y)
=>
  x = y
```

wobei allerdings die *Anzahl* identifizierender Attribute und damit die Anzahl von Zeilen der Gestalt

```
(F<+{E|->N})(X) = (F<+{E|->N})(Y)
```

variieren kann. Die *generische Struktur dieses Lemmatyps* ist (unter Berücksichtigung der zum Beweis nützlichen speziellen Hypothesenformeln dieses Lemmatyps)

```
( !X.!Y.(X,Y: T =>
            ( (F_1(X) = F_1(Y))  &
                      ⋮
            ( (F_k(X) = F_k(Y))  &
(X,Y : T \/ {E})  &
(F_1 : T  --> S_1)  &
      ⋮
(F_k : T  --> S_k)  &
(N_1 : S_1 - ran(F_1))  &
      ⋮
(N_k : S_k - ran(F_k))  &
(F_1<+E|->N_1)(X) = (F_1<+E|->N_1)(Y)  &
                ⋮
(F_k<+E|->N_k)(X) = (F_k<+E|->N_k)(Y)  &
      ...
  =>
   x = y
```

wobei k die Anzahl der in ihrer Gesamtheit identifizierenden Klassenattribute ist. Auch diese Darstellung wird zu einer speziellen Klasse von Beweisregeln führen.
Das in Abschnitt 7.5.1.1 vorgestellte mathematische Lemma Nr. 5 ist Repräsentant einer entsprechenden, aus *Lösch*operationen resultierenden Lemmaklasse, die sich in analoger Weise generisch beschreiben läßt. Auch für diese Klasse werden spezielle Beweisregeln abgeleitet.
Tabelle 7.2 zeigt die aus diesen Betrachtungen ableitbare generische Klassifikation der DAIDA-charakteristischen mathematischen Lemmas des Konsistenznachweises.

Operationstyp	Klasse	Lemma-Nr.	Herkunft
INITIALISIERUNG	**MIW**	1, 2, 3, 4	**I1**, **I2**, **S1** und **S2** (widerspr. Hypothese)
LÖSCHEN	**MSL**	5	Löschen auf Eindeutigkeitsbeziehung
EINFÜGEN	**MSE**	8	Einfügen auf Eindeutigkeitsbeziehung

Tabelle 7.2: Klassifikation der mathematischen Lemmas des Konsistenznachweises

7.5.2.3 Lemmas der Verfeinerungsbeweisverpflichtungen

Für die Lemmas der Verfeinerungsnachweise soll nun in einer analogen Weise verfahren werden. Hier werden zunächst die *mathematischen* Lemmas betrachtet, da sich diese, wie bereits besprochen, für die Verfeinerungsnachweise als "umfangreichere" Lemmaart erweisen.

7.5.2.3.1 Mathematische Lemmas Als Beispiel für die generische Einordnung der *mathematischen* Verfeinerungslemmas soll das Lemma Nr. 23

```
 ctx(ExtResearchCompaniesMch)  &
 inv(ExtResearchCompaniesMch_3)  &
 pre(ExtResearchCompaniesMchremoveEmployee_1(empl))  &
 asn(ExtResearchCompaniesMch_3)
=>
 {empl}<<|%x.(x: dom(empClass_3) & empClass_3(x) = a,b,c | c)
 =
 %x.(x: dom({empl}<<|empClass_3) & ({empl}<<|empClass_3)(x) = a,b,c | c)
```

diskutiert werden. Es resultiert aus der Beweisverpflichtung der zweiten Verfeinerung der Operation `removeEmployee` und der CHANGE-Gleichung

```
worksOn_2 = worksOn_3
```

unter Berücksichtigung der Definition der Variablen `worksOn` in der zweiten Verfeinerungsmaschine.
Es fällt auf, daß dieses Lemma *charakteristisch* für *Lösch*operationen des zweiten Verfeinerungsschritts ist: es entsteht für Attributvariablengleichungen

```
attrvar_2 = attrvar_3
```

von Attributen `attr`, die *nicht* den Klassenidentifikator bilden, d.h. als (im Unterschied zu *identifizierenden Attributen* wie `projName`, `getsGrantFrom` und `compName`) *abhängige Attribute*

bezeichnet werden können. Von den mathematischen Lemmas der zweiten Verfeinerung sind noch Nr. 24 und 25 (entsprechend den weiteren nichtidentifizierenden Attribute der Klasse `EMPLOYEES`) sowie Nr. 18 für die Operation `removeProject` (Attribut `consortium` der Klasse `PROJECTS`) von derselben generischen Gestalt:

```
 ctx(M)  &
 inv(M3)  &
 pre(MOp1(E))  &
 asn(M3)
=>
 {E}<<|%X.(X: dom(F) & F(X) = A1,...,An | Ai)
 =
 %X.(X: dom({E}<<|F) & ({E}<<|F)(X) = A1,...,An | Ai)
```

Lemmas dieser Gestalt folgen unmittelbar aus der Semantik von λ-Abstraktion und Domänenrestriktion. Um den Beweis dieses Lemmatyps zu ermöglichen, wird eine entsprechende Gleichheitsregel formuliert (vgl. u.). In diesem speziellen Fall ist die nichtfestgelegte Attributanzahl `n` unproblematisch, da sie für die Anwendbarkeit der Gleichheitsregel irrelevant ist.
Für *identifizierende* Klassenattribute führen die Beweisverpflichtungen der Löschoperationen der zweiten Verfeinerung zu Lemmas abweichender Gestalt. Die mathematischen Lemmas Nr. 19 und Nr. 20 sind ein Beispiel hierfür. Sie entstehen für die *zusammen* den Identifikator der Klasse `PROJECTS` bildenden Attribute `getsGrantFrom` und `projName`.
Zusätzlich ist der Fall zu unterscheiden, daß, wie für die Klasse `COMPANIES`, ein *einzelnes* identifizierendes Attribut existiert, was jedoch für die Löschoperationen des Beispiels nicht vorkommt.

Anhand der Beispielverfeinerung zeigt sich, daß eine derartige generische Klassifikation auch für die mathematischen Lemmas der übrigen Operationstypen vorgenommen werden kann: die Kriterien, gemäß denen sich eine Klassifikation als zweckmäßig erweist, sind:

- Typ der Operation (`L`, `E`, `Ä`)
- Art der CHG-Gleichung, aus der das Lemma resultiert: zwischen
  - (gemeinsam) *identifizierenden* Klassenattributvariablen (im Beispiel die Variablen `projName` und `getsGrantFrom`)
  - *abhängigen* Klassenattributvariablen
  - *Klassenidentifikatoren*, d.h. zwischen den im ersten Verfeinerungsschritt eingeführten Identitätsfunktionen auf den Ausprägungsmengen (im Beispiel die Variablen `empId` und `projId`) bzw. *alleine* identifizierenden Klassenattributen (im Beispiel die Variable `compName`)
  - Klassenausprägungsvariablen
- die *Stufe* der Verfeinerung

Die Verfeinerungsstufe ist bedeutsam, da die in der Spezifikationsmaschine formulierten, aus der TDL-Beschreibung abgeleiteten Zustandsvariablen in den einzelnen Verfeinerungsebenen unter-

schiedlich repräsentiert werden[45], was sich auch in einer entsprechenden "Manipulation" durch die Operationen niederschlägt. Die durch die standardisierte TDL-Übersetzung resultierende Sicht der Daten wirkt sich daher auch auf die Klasseneinteilung der *Verfeinerungs*lemmas aus. Die *Struktur* der mathematischen Lemmas ist wiederum unabhängig von der betrachteten Entitätenklasse.
Spezielle Sorten mathematischer Lemmas, die *nicht* gemäß obiger Kriterien klassifizierbar sind, werden für die dritte Verfeinerung generiert. Die bedeutendste Klasse resultiert aus dem Übergang von dynamisch typisierten Operationsvorbedingungen zu statisch typisierten Vorbedingungen und Einführung einer bedingten Substitution: ein Teil der Lemmas entsteht unter der zusätzlichen Voraussetzung der *nicht* erfüllten Substitutionsbedingung, wie z.B. Nr. 65:

```
 ctx(ExtResearchCompaniesMch) &
 inv(ExtResearchCompaniesMch_4) &
 pre(ExtResearchCompaniesMchremoveEmployee_1(empl)) &
 asn(ExtResearchCompaniesMch_4) &
 not(empl: dom(empRel_4))
=>
 empClass_3 = {empl}<<|empClass_3
```

Hier kann gezeigt werden, daß der Hypotheseninhalt jeweils *widersprüchlich* ist, d.h. die Lemmas sind *trivialerweise erfüllt*: es gilt stets

$$\texttt{pre(MOp}_1\texttt{(X))} \quad \Rightarrow \quad \texttt{pre(MOp}_3\texttt{(X))}$$

wobei sich die Konjunkte von $\texttt{pre(MOp}_3\texttt{(X))}$ entsprechend der statischen oder dynamischen Überprüfbarkeit in zwei Konjunktmengen *stat* und *dynam* separieren lassen. Die Negation der Substitutionsbedingung ist nach Konstruktion gerade $\neg dynam$. Durch Anwendung elementarer logischer Beziehungen folgt hieraus:

$$\begin{aligned} \texttt{pre(MOp}_1\texttt{(X))} \wedge \neg\ dynam \quad &\Rightarrow \quad \texttt{pre(MOp}_3\texttt{(X))} \wedge \neg\ dynam \\ &\Rightarrow \quad stat \wedge dynam \wedge \neg\ dynam \\ &\Rightarrow \quad FALSE \end{aligned}$$

d.h. der Hypotheseninhalt ist, wie behauptet, widersprüchlich.
Obwohl nicht strukturell identisch, sollen diese Lemmas dennoch einer gemeinsamen Klasse zugeordnet werden, da sie mittels desselben Beweisprinzips (Explizierung eines Widerspruchs in der Hypothese) nachweisbar sind[46].
Tabelle 7.3 faßt die generischen Klassen der mathematischen Verfeinerungslemmas zusammen.

[45] insbesondere durch die eingeführten "relational strukturierten" Zustandsvariablen, durch Formulierung entsprechender *Definitionen*

[46] Die unten vorgestellte Regelsammlung unterstützt den Beweis von Lemmas dieser Klasse bisher nicht.

Stufe	Operationstyp	Klasse	Lemma-Nr.	CHG-Gleichungsart
1. Ver-	LÖSCHEN	**MV1L1**	14, 15	gemeinsam ident. Attribut
feinerung		**MV1L2**	-[47]	einzeln ident. Attribut
	EINFÜGEN	**MV1E1**	-[47]	gemeinsam ident. Attribut
		MV1E2	16	einzeln ident. Attribut
	ÄNDERN	**MV1Ä1**	-[47]	gemeinsam ident. Attribut
		MV1Ä2	-[47]	einzeln ident. Attribut
2. Ver	LÖSCHEN	**MV2L1**	19, 20	gemeinsam ident. Attribut
feinerung		**MV2L2**	18, 23, 24, 25	abhängiges Attribut
		MV2L3	17, 22	Klassenidentifikator[48]
		MV2L4	22, 26	Ausprägungsmenge
	EINFÜGEN	**MV2E1**	-[47]	gemeinsam ident. Attribut
		MV2E2	32, 33, 37, 38, 39	abhängiges Attribut
		MV2E3	34, 36	Klassenidentifikator[48]
		MV2E4	35, 40	Ausprägungsmenge
	ÄNDERN	**MV2Ä1**	-[47]	gemeinsam ident. Attribut
		MV2Ä2	42, 43, 44, 47, 48, 49	abhängiges Attribut
		MV2Ä3	41, 46	Klassenidentifikator[48]
		MV2Ä4	45, 50	Ausprägungsmenge
3. Ver	LÖSCHEN	**MV3L1**	53, 54	gemeinsam ident. Attribut
feinerung		**MV3L2**	52, 62, 63, 64	abhängiges Attribut
		MV3L3	51, 61	Klassenidentifikator[48]
	EINFÜGEN	**MV3E1**	-[47]	gemeinsam ident. Attribut
		MV3E2	75, 76, 91, 92, 93	abhängiges Attribut
		MV3E3	77, 90	Klassenidentifikator[48]
	ÄNDERN	**MV3Ä1**	-[47]	gemeinsam ident. Attribut
		MV3Ä2	104, 105, 106, 116, 117, 118	abhängiges Attribut
		MV3Ä3	103, 115	Klassenidentifikator[48]
		MV3Ä4	107, 119	Ausprägungsmenge
		MV3Ä5	102, 114	strukturierte Variable
	allgemein	**MV3W**	55,..,60, 65,..,70, 78,..,89, 94,..,101, 108,..,113, 120,..,125	allgemein (Widersprüche)

Tabelle 7.3: Klassifikation der mathematischen Lemmas der Verfeinerung

Lemmas von Klassen mit leerer "Lemma-Nr"-Spalte treten im Beispiel nicht auf, können jedoch allgemein vorkommen. In Abschnitt 7.5.2.4 wird dieser Punkt diskutiert.

7.5.2.3.2 Typlemmas Mehrere charakteristische Klassen von *Typ*lemmas lassen sich für die im *zweiten* Verfeinerungsschritt durchgeführte *Datenverfeinerung* identifizieren. Für jeden der drei Operationstypen "Einfügen", "Löschen" und "Ändern" entstehen Lemmas, die aus der entsprechenden Modifikation der neu eingeführten "strukturierten" Zustandsvariablen resultieren. Als Beispiel sollen die für die *Lösch*operationen der zweiten Verfeinerung erzeugten Lemmas dienen: Lemma Nr. 45

```
  ctx(ExtResearchCompaniesMch) &
  inv(ExtResearchCompaniesMch_3) &
  pre(ExtResearchCompaniesMchremoveEmployee_1(empl)) &
  asn(ExtResearchCompaniesMch_3)
=>
  {empl}<<|empClass_3: dom({empl}<<|empClass_3)
       --> Strings*dom(compClass_3)*POW(dom(projClass_3))
```

ist (wie Lemma Nr. 44) von der hierfür charakteristischen generischen Struktur[49]

```
  F : dom(F) --> A_1 * ... * A_n  &
  E : S  &
  S = dom(F) &
  ...
=>
  {E} <<| F : dom({E} <<| F) --> A_1 * ... * A_n
```

Die nachzuweisende Beziehung folgt direkt aus der Semantik der Domänenrestriktion. Zum Beweis dieses Lemmatyps existiert eine spezielle Beweisregel (vgl. u.).
Für die Typlemmas der *Einfüge-* und *Lösch*operationen der zweiten Verfeinerung verlaufen die Betrachtungen analog.
Die Datenstrukturierung der *dritten* Verfeinerung führt ebenfalls für jeden der drei Operationstypen zu einer speziellen Klasse von Typlemmas, deren generische Gestalt, obwohl komplexer, prinzipiell in derselben Weise erarbeitet werden kann wie zuvor. Der Beweis dieser Lemmas wird von der aktuell vorliegenden Regelsammlung allerdings noch nicht unterstützt.
Aus der Entwurfsentscheidung der ersten Verfeinerung, den unbeschränkten Nichtdeterminismus mit Hilfe einer systemseitig zur Verfügung gestellten, "frische" Identifikatorkonstanten liefernden Funktion zu eliminieren, resultiert für die Operation `newEmployee` das auf der Diskursebene *unbeweisbare* Typlemma 37:

[47] Lemmas dieser Klasse kommen *wegen der Gestalt des Beispiels* nicht vor. *Allgemein* können jedoch entsprechende Lemmas auftreten, vgl. u., Abschnitt 7.5.2.4.

[48] sowie *alleinig* identifizierende Attribute

[49] Die Formel `S = dom(F)` steht genaugenommen nicht in der Hypothese, sondern fließt als *Definitionsgleichung* ein.

```
    ctx(ExtResearchCompaniesMch) &
    inv(ExtResearchCompaniesMch_2) &
    pre(e <-- ExtResearchCompaniesMchnewEmployee_1(newname,belongs,works)) &
    asn(ExtResearchCompaniesMch_2)
  =>
    newEmpId: EmpIds-employees_2
```

Es ist lediglich unter Zuhilfenahme von nicht durch Regeln erfaßbarem *Metawissen* (*über* den Verfeinerungsprozeß) zu beweisen und kann daher nur "von Hand" verifiziert werden. Auch auf den weiteren Verfeinerungsstufen werden derartige Lemmas erzeugt[50].
Tabelle 7.4 zeigt die generischen Klassen der Typlemmas der Verfeinerung. Die die relativen Terminationsnachweise verlangenden Typlemmas (vgl. o.) werden, unabhängig von der Verfeinerungsstufe, in *einer* gesonderten Klasse aufgeführt, da deren Beweis jeweils nach demselben Prinzip verläuft.

Stufe	Operationstyp	Klasse	Lemma-Nr.	Herkunft
2. Verfeinerung	INITIALISIERUNG	**TV2I**	38, 39, 40, 41	neue Variable
	LÖSCHEN	**TV2L**	44, 45	strukturierte Variable
	EINFÜGEN	**TV2E**	46, 50	strukturierte Variable
	ÄNDERN	**TV2Ä**	52, 53	strukturierte Variable
3. Verfeinerung	INITIALISIERUNG	**TV3I**	54, 55, 56[51]	neue Variable
	LÖSCHEN	**TV3L**	58, 60	strukturierte Variable
	EINFÜGEN	**TV3E**	62, 66	strukturierte Variable
	ÄNDERN	**TV3Ä**	70, 77	strukturierte Variable
allgemein	allgemein	**TVT**	36, 59, 61, 64 65, 68, 69, 71 72, 73, 75, 76	relativer Terminationsnachweis

Tabelle 7.4: Klassifikation der Typlemmas der Verfeinerung

7.5.2.4 Zur Vollständigkeit der Klassifikation

Die in den Tabellen der letzten Abschnitte dargestellten Lemmaklassen wurden, ausgehend von den für das Verfeinerungs*beispiel* generierten Lemmas, identifiziert. Die Lemmaklassifikation nimmt lediglich Bezug auf Kriterien *unabhängig von den spezifischen Detaills einer Verfeinerung* und ist daher *allgemeingültig*. Andererseits stellt sich die Frage, inwieweit die gewonnene Klassifikation *vollständig* ist, d.h. *alle* im Rahmen einer *beliebigen* DAIDA-Standardverfeinerung

[50] Diese sollen jedoch im Folgenden nicht als Lemmaklasse betrachtet werden, da sie nicht durch Regeln beweisbar sind.

[51] Das (hier nicht aufgeführte) Typlemma Nr. 57 resultiert aus einem kleinen Fehler in der Verfeinerung: die Variable `tEmpId` wurde irrtümlicherweise in der dritten Verfeinerung *nochmals* deklariert. `devB` erkennt derartige Fehler *nicht*.

entstehenden generischen Lemmaklassen beinhaltet. Dies soll nun kurz erörtert werden.
In die obigen Tabellen wurden bereits auch die Lemmaklassen aufgenommen, die zwar nicht im betrachteten Beispiel auftreten, jedoch prinzipiell vorkommen können. Dieser Fall tritt lediglich für die mathematischen Lemmas der Verfeinerung auf (Tabelle 7.3, Lemmaklassen *ohne* Eintrag in der Spalte "Lemma-Nr"). Ergänzt wurden:

- die Klassen **MV1L1**, **MV1E2**, **MV1Ä1** und **MV1Ä2** des ersten Verfeinerungsschritts: Lemmas ergeben sich lediglich für die CHANGE-Gleichungen derjenigen Zustandsvariablen, die der durchgeführten Datenidentifikation zugrundeliegen, d.h. für diejenigen Attribute, die aufgrund von TDL-UNIQUE-Vereinbarungen "einzeln" oder "gemeinsam identifizierend" sind. Die im Beispiel nicht vorkommenden Fälle wurden ergänzt, wobei besonders ins Gewicht fiel, daß für die Klasse `Employees`, die u.a. Gegenstand der beiden Änderungsoperationen ist, keine UNIQUE-Beziehung besteht.
- die Klassen **MV2E1**, **MV2Ä1**, **MV3E1** und **MV3Ä1** des zweiten bzw. dritten Verfeinerungsschritts: der Grund ist wiederum das Fehlen *identifizierender* Attribute der Klasse `Employees`.

Weitere Lemmaklassen, die auf den ersten Blick zu fehlen scheinen, können jedoch *nicht* auftreten. Hierzu gehören:

- für den Konsistenznachweis:
  - weitere Typlemmaklassen für Änderungsoperationen (Änderungsoperationen modifizieren lediglich *Attribute* von Objekten, jedoch nicht die Ausprägungsmengen)
  - eine Klasse **MSÄ** (Eine Änderungsoperation sollte lediglich die "*abhängigen*" Entitätenattribute modifizieren, andernfalls ließe sie sich als nicht elementare Operation, bestehend aus kombinierten Löschen und Einfügen, auffassen.)
- für die *mathematischen* Lemmas der Verfeinerungsbeweisverpflichtungen: einige Lemmaklassen treten nicht auf, da die zugehörigen CHANGE-Gleichungen schon während der Verarbeitung der Beweisverpflichtungen eliminierbar sind. Dies ist insbesondere dann der Fall, wenn die betroffene(n) Zustandsvariable(n) im entsprechenden Verfeinerungsschritt *keine Darstellungsänderung(en)* erfährt (erfahren), oder wenn die von dem betrachteten Operationspaar (abstrakte und konkrete Operation) durchgeführten Substitutionen auf den betrachteten Verfeinerungsebenen identische rechte Seiten haben. Auf diese Weise werden u.a. die Lemmas des dritten Verfeinerungsschritts, resultierend für Einfüge- und Löschoperationen auf den *Ausprägungsmengen* und den *strukturierten Variablen*, *vor* ihrer Entstehung eliminiert.

7.5.2.5 Zusammenfassung

Die durchgeführten Betrachtungen haben gezeigt, daß eine weitgehende Klassifikation der erzeugten Lemmas möglich ist. Die Einordnung geschieht hierbei anhand von Kriterien, die *unabhängig von der im speziellen betrachteten Verfeinerung* sind und lediglich vom *Typ* der zu-

gehörigen Operation, der *Art* der manipulierten Zustandsvariablen (Ausprägungsmenge, Attribut usw.) sowie ggf. von der Stufe der Verfeinerung abhängen. Es wurde deutlich, daß die Abbildung der TDL-Beschreibungen in die initiale abstrakte Maschine die Gestalt der Lemmaklassen maßgeblich beeinflußt, da hier die Typisierung der Systemzustandsvariablen[52] festgelegt wird. Ein zweiter bestimmender Faktor sind die standardisierten Verfeinerungsschritte, über die die *Zielsprachencharakteristika* einfließen.
Die anhand des Beispiels identifizierten Lemmaklassen sind daher *unabhängig von der betrachteten Verfeinerung* gültig. Die Einteilung wurde um Klassen, die nicht im betrachteten Beispiel auftreten, jedoch allgemein identifiziert werden können, ergänzt.
Die im folgenden beschriebenen Theorien unterstützen daher den Beweis der Lemmas *unabhängig* von der speziell betrachteten DAIDA-Verfeinerung, sofern sie sich gemäß der obigen Klassifikation einordnen lassen. Zu betonen ist allerdings, daß nur die Lemmas für Operationen mit elementarer, "standardisierbarer" Gestalt sicher bewiesen werden können. Falls Intergritätsprädikate spezieller, anwendungsabhängiger Gestalt existieren, sind *anwendungsspezifische Theorien* zu ergänzen.

7.6 Theorien und Taktiken zum Beweis der Lemmas

Anhand der Lemmaeinordnung wurde eine *Theoriesammlung* entwickelt, die den Beweis der für eine DAIDA-Verfeinerung erzeugten Lemmas ermöglichen soll. Diese soll im folgenden überblicksweise beschrieben werden.
Anhang F zeigt die hierfür formulierte *Beweismethode* (vgl. Kapitel 6), die die Theorien und Taktiken zum Beweis der mathematischen und Typlemmas enthält. Bild 7.5 gibt einen geordneten Überblick über deren Inhalt.
Neben den bereits besprochenen Komponenten zur Implementation des Definitionskonzepts enthält die Beweismethode die Theorien und Taktiken zum Beweis der Lemmas, die in den folgenden Unterabschnitten einzeln besprochen werden. Die Gruppierung der Regeln in unterschiedliche Theorien wurde derart vorgenommen, daß

- einerseits eine hinreichend feine Unterteilung vorliegt, die eine selektive Referenzierung geeigneter Regelgruppen in Abhängigkeit einer gegebenen Lemma*art* (d.h. mathematisches oder Typlemma der Konsistenz- oder Verfeinerungsbeweisverpflichtung) ermöglicht,

- andererseits keine zu feine Separierung vorliegt, um die Taktiken überschaubar zu halten.

Der nachfolgende Abschnitt bespricht die *Beweise* einiger Lemmas der betrachteten Beispielverfeinerung und verdeutlicht damit die Funktionsweise der verwendeten Theorien und Taktiken.

[52] Die zur *relationalen Repräsentation* des Systemzustands während der Verfeinerung eingeführten "strukturierten" Variablen fallen nicht unter diese Betrachtung.

BEWEISMETHODE `ExtResearchCompaniesPmd`	
TAKTIKEN	
REFTAC	für Definitionskonzept
TYPETAC	Paare von Vorwärts- und Rückwärtstaktiken zum Beweis der Lemmas
MATHTAC	Paare von Vorwärts- und Rückwärtstaktiken zum Beweis der Lemmas
THEORIEN FÜR DAS DEFINITIONSKONZEPT	

RÜCKWÄRTSTHEORIEN	`RefinementLemmaTacticsX`
	`InitMathTypeLemmaGenerationX`
	`MathTypeLemmaGenerationX`
	`AsnDefLemmasX`

THEORIEN ZUM BEWEIS DER LEMMAS	
RÜCKWÄRTSTHEORIEN	`support`, `empty`, `hypdown`, `exfalsoquodlibet`, `refinementEqualities`, `refinementsupport`
VORWÄRTSTHEORIEN	`fsupport`, `fempty`, `specialize`, `frefinesupport`

Tabelle 7.5: Schematischer Aufbau der entwickelten Beweismethode

7.6.1 Die Theorien zum Beweis der Lemmas

7.6.1.1 Die Theorie `support`

Theorie `support` enthält Regeln allgemeiner Art[53] sowie besondere, zum Beweis spezieller *Konsistenz*lemmas geeignete Regeln.
Beispiele allgemeiner Regeln sind:

- elementare mengentheoretische *Axiome*. Z.B. besagt

  ```
  S : POW(S)
  ```

  daß alle Mengen Elemente ihrer Potenzmenge sind.

- elementare mengentheoretische *Schlußregeln*:

  ```
   S : T  &  R : T
  =>
   (R \/ S) : T
  ```

  sagt aus: um zu zeigen, daß die Vereinigung zweier Mengen von einem bestimmten Typ `T` ist, genügt der Nachweis, daß jede der beiden Mengen individuell die Typbeziehung erfüllt.

Zu den speziellen Regeln zählen:

[53] d.h. Regeln, die nicht zum Beweis eines bestimmten Lemmatyps dienen, sondern allgemein angewandt werden

- Fallunterscheidungen[54]:

```
inhyp(X : T) &
((X = E)  =>  N incl G(H(E))) &
(X : T - {E}  =>  F(X) incl G(H(X)))
=>
((F<+{E|-> N})(X) incl G(H(X)))
```

  Um nachzuweisen, daß das Bild einer durch funktionales Überschreiben erhaltenen Funktion für ein beliebiges Argument die angegebene Inklusionsbeziehung erfüllt, bietet sich eine Fallunterscheidung nach dem Wert des Arguments an: es genügt, die Beziehung für die Fälle

  (i) `X = E`, d.h. das Bild von `X` ist gleich dem "überschreibenden" Wert `N`

  (ii) `X : T - {E}`

  zu beweisen, wobei `T` der Typ von `X` gemäß einer für `X` getroffenen Typvereinbarung ist[55].

- Regeln zum Beweis generisch klassifizierbarer *Typlemmas des Konsistenznachweises*: beispielsweise ermöglicht die Regel

```
inhyp(A : R - T) &
inhyp(T : POW(R)) &
B : S  &  F : T --> S
=>
F<+{A|-> B} : (T \/ {A}) --> S
```

  den Beweis von Lemmas der Lemmaklasse **TE2**.

- Regeln zum Nachweis des Erhalts von Eindeutigkeitsbeziehungen für die unterschiedlichen Operationstypen: die Regel

```
inhyp(!x.!y.(x,y: T =>
             ((F(x) = F(y)) & (G(x) = G(y)) =>  (x = y)))) &
inhyp(x,y : T - U) &
inhyp((U <<| F)(x) = (U <<| F)(y)) &
inhyp((U <<| G)(x) = (U <<| G)(y))
=>
(x = y)
```

  realisiert den unmittelbaren Beweis der für die Lösch-Operationen des Beispiels aus dem Eindeutigkeitsprädikat **S1** resultierenden mathematischen Lemmas der Klasse **MSL**: in diesem Fall befinden sich gerade die Formeln in der Hypothese, deren Vorhandensein durch die `inhyp`-Tests der Regel getestet wird. Durch die Verwendung von `inhyp`-Tests wird die Anwendbarkeit der (sehr speziellen) Regel auf die sinnvollen, unmittelbar erfolgreichen Fälle eingeschränkt.
  Ein Problem besteht darin, daß die o.g. Regel nur für diejenigen Eindeutigkeitsprädikate

[54] hier exemplarisch so formuliert, daß sich in einem speziellen Fall der Erhalt der Integritätsbedingung **I1** nachweisen läßt. Es ist zu untersuchen, in welcher Weise sich derartige Regeln auch *allgemein* formulieren lassen; dies führt zu der Frage der adäquaten Abbildung von "Objekten höherer Ordnung", vgl. auch Abschnitt 8.3.

[55] Eine solche Vereinbarung wird in DAIDA-Verfeinerungen immer existieren: Integritätsprädikate werden stets *klassenbezogen* formuliert, was in einer entsprechenden Typisierung in der Lemmahypothese resultiert.

anwendbar ist, die die Eindeutigkeitsbeziehung in Abhängigkeit von *zwei* identifizierenden Attributen beschreiben. Die *allgemeine* Realisierung dieses Regeltypus ist jedoch wesentlich komplexer. Deshalb bietet es sich in diesem Fall (auch aus Effizienzgründen) an, die Regel für "vernünftige" Anzahlen von identifizierenden Attributen auszuformulieren[56]. Dies verdeutlicht die prinzipiellen Grenzen bei der Formulierung generischer Regeln unter Verwendung von Metavariablen[57].

7.6.1.2 Die Theorie `empty`

Theorie `empty` enthält Axiome zum Beweis der aus Initialisierungsbeweisverpflichtungen resultierenden Typlemmas: so besagt beispielsweise

```
{} :  {} --> T
```

daß die "leere" Funktion[58] eine totale Abbildung von der leeren Menge in eine beliebige Bildmenge ist. Hierdurch wird der unmittelbare Beweis der für die Initialisierung der Attributfunktionsvariablen resultierenden Typlemmas der Klasse **TI** ermöglicht.

7.6.1.3 Die Theorie `fempty`

Theorie `fempty` besteht lediglich aus der *Vorwärts*regel

```
a : {}   =>   FALSE
```

Diese expliziert den aus der unerfüllbaren Voraussetzung `a : {}` resultierenden Widerspruch, wie er für mathematische Konsistenzlemmas der Initialisierungsoperationen (Klasse **MIW**) auftritt (vgl. Theorie `exfalsoquodlibet`).

7.6.1.4 Die Theorie `fsupport`

Theorie `fsupport` stellt eine *allgemeine* Regelsammlung für Vorwärtsdeduktionen dar, die zum Beweis der Konsistenz- und Verfeinerungslemmas referenziert wird. Beispiele aus der Theorie `fsupport` sind:

[56] In der Beweismethode wird die Regel, orientiert am Beispiel, lediglich für maximal zwei Schlüsselkomponenten formuliert.

[57] Bei der Diskussion der im Folgenden vorgestellten Theorien werden weitere Beispiele für anscheinend nicht generisch realisierbare Regeln gegeben, deren Umsetzung mittels bestimmter Tricks jedoch möglich ist.

[58] aufgefaßt als die Menge ihrer Urbild-Bild-Paare $\{(x, y) \mid F(x) = y\}$

- Regeln zur Ableitung von *Eigenschaften von Funktionsdomänen* (`dom`) *und Bildbereichen* (`ran`): z.B. expliziert

```
 F : T --> S
=>
 dom(F) = T
```

  eine Eigenschaft *totaler* Funktionen `F`: deren Domäne `dom(F)` ist (im Gegensatz zu partiellen Funktionen) *gleich* der Menge `T`, auf der `F` definiert ist.

- Regeln zur Explizierung von *Typbeziehungen*:

```
 E : T  &   F : T --> S
=>
 F(E) : S
```

  sagt aus, daß die Anwendung einer totalen Funktion auf ein Element ihrer Domäne stets ein Element ihres Bildbereichs liefert.

- Regeln zur Explizierung bestimmter *Gleichheiten*: die Regel

```
 (X : T - {E})  &  (E : T)  &  (F : T --> S) & (N : S)
=>
 (F<+{E |-> N})(X) =  F(X)
```

  sagt aus, daß unter der Voraussetzung `X : T - {E}` der Wert für `X` unter einer Funktion `F` von einem funktionalen Überschreiben des Bilds von `E` unabhängig ist.
  Eine andere Gleichheitsregel beschreibt ein einfaches Absorptionsgesetz für den Mengenvereinigungsoperator "`\/`":

```
 S incl T
=>
 (S \/ T) = T
```

7.6.1.5 Die Theorie `specialize`

Theorie `specialize` unterstützt den Beweis einiger mathematischer Lemmas der Konsistenzbeweisverpflichtungen: die dort formulierten Vorwärtsregeln erlauben eine *Spezialisierung* allquantifizierter Hypothesenformeln durch Substitution eines "geeigneten" Terms für die gebundene Variable: so besagt die Regel

```
 !x.(x : T => P)   &  (Y : T)
=>
 [x:= Y]P
```

beispielsweise, daß sich ein in der (an die Menge `T` gebundenen) Variablen `x` allquantifiziertes Prädikat `P` durch Substitution eines "typkompatiblen" Terms `Y` für `x` spezialisieren läßt - ein oft wiederkehrender Fall, da viele Invariantenprädikate von der Gestalt

```
!x.(x : T => P)
```

sind[59].

7.6.1.6 Die Theorie `hypdown`

Theorie `hypdown` enthält die einzelne Regel

```
 inhyp([x:= E]P)  &   (([x:= E]P) => Q)
=>
 Q
```

die es ermöglicht, einen durch Anwendung einer Regel der Theorie `specialize` in der Hypothese erzeugten Substitutionsausdruck vor das zu beweisende Teilziel zu ziehen (eine semantisch zulässige Umkehrung der eingebauten Schlußregel DED). Dieser technische Trick ist notwendig, da die Substitutionsregel SUB nicht auf Formeln der Hypothese anwendbar ist, dies jedoch notwendig werden kann. Nach Substitutionsdurchführung und erneutem DED steht die aus der Substitution resultierende Formel wieder in der Hypothese.
Der `inhyp`-Test schränkt die Anwendbarkeit der (ansonsten universell anwendbaren) Regel auf die Fälle ein, in denen die Hypothese ein Substitutionsausdruck enthält.

7.6.1.7 Die Theorie `exfalsoquodlibet`

Theorie `exfalsoquodlibet`[60] erlaubt die unmittelbare Reduktion eines Teilziels, sofern dessen Hypothese den elementaren Widerspruch *FALSE* enthält (vgl. o.):

```
inhyp(FALSE)   =>   P
```

7.6.1.8 Die Theorie `refinementEqualities`

Theorie `refinementEqualities` faßt *charakteristische*, zum Beweis der *mathematischen* Lemmas der Verfeinerungsbeweisverpflichtungen nützliche *Gleichheitsbeziehungen* zusammen[61]. Die

[59] In der Beispielverfeinerung sind die Invariantenprädikate `I1` und `I2` von solcher Gestalt. Eine allgemeinere Version der formulierten Beweismethode sollte auch Spezialisierungsregeln für Prädikate enthalten, bei denen über mehrere Variablen quantifiziert wird.

[60] lat.: *aus dem Falschen, wie es beliebt*, in der Logik übliche Bezeichnung für die semantische Eigenschaft von Implikationsformeln (und, über das Deduktionstheorem, des semantisches Folgerns) für einen "falschen" Antezedenten den Wahrheitswert TRUE anzunehmen.

[61] Die Theorie enthält allerdings auch allgemeine Gleichungen, sodaß sie auch für den Beweis der *Typ*lemmas der Verfeinerung referenziert wird.

Gleichungen ermöglichen es, allgemeine, aus den im Beispiel getroffenen "typischen" Entwurfsentscheidungen für die unterschiedlichen Operationstypen resultierenden Lemmas zu verifizieren. So existieren beispielsweise Regeln, die es erlauben, die aus den unterschiedlichen Darstellungen der Objekte in der ersten und zweiten Verfeinerungsmaschine resultierenden mathematischen Lemmas für die Einfüge-Operationen unmittelbar zu beweisen: Gleichung

```
(%x.(x : dom(F \/ {(A,B,C)}) & (F \/ {(A,B,C)})(x) = (G,H) | G))
     =
((%x.(x : dom(F) & F(x) = (G,H) | G))<+{A |-> B})
```

besagt, daß die nach Neueinfügung eines *Tupels* durch Mengenvereinigung[62] in der zweiten Verfeinerungsmaschine entstehende Projektionsfunktion der ersten Bildkomponente äquivalent zu einer durch funktionales Überschreiben entstehenden Projektionsfunktion ist. Letztere Darstellung entspricht gerade der Repräsentationsform in der ersten Verfeinerungsmaschine (unter Berücksichtigung von CHANGE- und Definitionsbeziehungen). Durch die angegebene Gleichung werden Lemmas der Klasse **MV2E2** beweisbar.
Ähnliche Regeln sind für weitere Klassen mathematischer Verfeinerungslemmas formuliert (vgl. Anhang F).
Der soeben diskutierte Regel*typ* verdeutlicht ein aus der Verwendung von Tupelnotationen resultierendes Problem: die beschriebene Beziehung ist für *beliebige* Tupelkomponentenanzahl und *beliebigen* Bildkomponentenindex zu formulieren. Dies ist zwar prinzipiell unter Verwendung der vom B-Tool angebotenen mächtigen Konzepte auf "operationalem" Weg lösbar[63], jedoch geht durch die Verwendung prozeduraler Konzepte wie `bcall` der deklarative Charakter der Regelsammlung und deren leichte Durchschaubarkeit (ähnlich wie in PROLOG-Programmen bei Verwendung des CUTs) verloren. Es wurde daher folgender Weg beschritten:

- In Fällen, in denen eine einfache, deklarative Lösung nicht möglich erscheint, wurden die entsprechenden Regeln für vernünftige Tupellängen ($n = 4, n = 5$) ausformuliert[64]. In obigem Beispiel resultiert dieses Vorgehen in $\Theta(n^2)$ Regeln.
  Für die Lemmaklasse `MV2Ä2` wird durch geschicktes Vorgehen eine Reduktion der Regelanzahl auf $\Theta(n)$ erreicht (vgl. Anhang F). Auch die im folgenden Abschnitt vorgestellte Theorie `refinementsupport` enthält Beispiele für mögliche Tricks (Lemmaklasse **TV2Ä**).

- In bestimmten Fällen kann eine Ausformulierung der Regel für eine beschränkte Tupellänge durch die *Verwendung applikativer Konzepte* umgangen werden: die Gleichung

  ```
  (dom(F \/ {(A,B)})) = (dom(F) \/ {first(A)})
  ```

  beschreibt, in welcher Weise sich die Domäne einer Funktion ändert, wenn ein Tupel (durch Mengenvereinigung) hinzugefügt wird. Hierbei wird die *Linksassoziativität* des Separators "`,`" ausgenutzt: durch die syntaktische Variable `A` wird eine *Liste* von Tupelkomponenten

[62] aufgrund der Datenstrukturierung

[63] Auf dem B-Tool könnte man (ohne großen Aufwand) einen Interpreter für eine einfache Programmiersprache implementieren, d.h. die vom B-Tool interpretierte Sprache ist *turingmächtig*, vgl. Kap. 4.

[64] Für realistische Anwendungen sollte $n = 10$ genügen. Zu bemerken ist außerdem, daß auch in der Regelsammlung `devB` z.T. derartige Einschränkungen bestehen: wegen der speziellen Natur der Verarbeitung von Operationsparameter darf deren Anzahl nicht größer als 26 sein.

gebunden, deren erste Komponente das die Funktionsdomäne erweiternde Element ist[65]. Diese wird durch die ebenfalls axiomatisierten Gleichheitsbeziehungen für `first` ermittelt:

```
first((A)) = A
      ;
first((A,B)) = first((A))
```

7.6.1.9 Die Theorie `refinementsupport`

Theorie `refinementsupport` faßt spezielle Regeln zum Beweis der *Typ*lemmas der *Verfeinerungs*beweisverpflichtungen zusammen.
Als Beispiel soll eine spezielle Regel diskutiert werden, die den Beweis der aus den Einfügeoperationen der zweiten Verfeinerung typischerweise resultierenden Typlemmas der Klasse **TV2E** ermöglicht:

```
 inhyp(F : W --> A * B) &
 (W = dom(F)) &
 (A * B) = (R * S) &
 (tail(G,H,I) : (A * B))
=>
 (F \/ {(G,H,I)}) : dom(F \/ {(G,H,I)}) --> R * S
```

Um zu beweisen, daß die nach Einfügen eines "neuen" Tupels entstehende Funktion die angegebene Typbeziehung[66] erfüllt, genügt es, zu zeigen: In der Hypothese befindet sich die aus der Invariante stammende Typisierung der Funktion, und es gilt:

- die Domänen sind identisch,
- die Bildbereiche der Funktionen sind gleich[67],
- das neueingefügte Tupel verletzt nicht die Typdefinition des Bildbereichs der Funktion.

Die Regel funktioniert *generisch für beliebige Tupelkomponentenanzahl* (≥ 3), wobei erneut die Linksassoziativität binärer Konnektive ausgenutzt wird. Zum Beweis der entstehenden Teilziele bedarf es zusätzlicher Regeln:

[65] Dies bedarf der nicht immer haltbaren Voraussetzung, daß die Funktionsdomäne nicht gleich einem kartesischen Produkt mehrerer Mengen ist. Eine weitere Ungenauigkeit entsteht durch die Verwendung runder Klammern als Tupelkonstruktor, da diese vom B-Tool als redundant angesehen werden. Diese Probleme ließen sich durch Verwendung einer anderen Klammerart (z.B."`[ ]`") als Tupelkonstruktor unmittelbar lösen.

[66] der *Konsequent* der Regel

[67] Dieser Umweg muß beschritten werden, da `R * S` aus `A * B` wegen der modifizierten Taktik zur Verarbeitung der Verfeinerungsbeweisverpflichtungen unter Anwendung von Definitionen hervorgegangen sein kann.

- der Nachweis der Bildbereichsgleichheit bedarf einer zusätzlichen Regel, die das Gleichheitsprädikat "=" auf kartesische Produkte "*" fortsetzt[68]:

```
 (A = L)  &  (B = M)
=>
 (A * B) = (L * M)
```

- eine ähnliche Regel setzt das Enthaltenseinsprädikat ":" auf kartesische Produkte fort:

```
 (A : S)  &  (B : T)
=>
 (A,B) : (S * T)
```

- das Hilfskonstrukt `tail` ist derart zu axiomatisieren, daß es bei "Anwendung" auf ein Tupel dessen "Bildkomponenten" liefert (wobei wiederum die o.g. Voraussetzung über die Atomarität von Funktionsdomänen einfließt):

```
tail(R,S) = S
      ;
tail(R,S,T) = (tail(R,S),T)
```

7.6.1.10 Die Theorie `frefinesupport`

Die Theorie `frefinesupport` ergänzt die von `fsupport` zur Verfügung gestellte Regelsammlung um Vorwärtsregeln allgemeiner Gestalt, die sich speziell zum Beweis der *Verfeinerungs*beweisverpflichtungen als nützlich erweisen. Auf eine weitergehende Beschreibung soll daher verzichtet werden.

7.6.1.11 Die Rolle der Metavariablen

Die Verwendung von Metavariablen wirft ein bereits in Kapitel 3 diskutiertes Problem auf: in vielen Fällen sollten Metavariable nur durch Ausdrücke bestimmter syntaktischer Kategorien instantiiert werden. Für die Regel

```
S : POW(S)
```

beispielsweise sollte `S` nur durch *mengenförmige* Ausdrücke instantiiert werden. Ein formal korrekter Ansatz bedingt die Axiomatisierung von Prädikaten für die unterschiedlichen syntaktischen Kategorien, wie sie bereits in Kap. 3 beschrieben wurde sowie die Ergänzung formaler, die Anwendbarkeit der Regel auf die zulässigen Fälle einschränkenden Antezedenten.

[68] Obwohl prinzipiell nicht notwendig, erleichtert diese Regel den Nachweis der Gleichheit, falls Definitionen angewandt wurden.

Bei "vernünftig" formulierten Spezifikationen und Verfeinerungen[69] führt diese Ungenauigkeit allerdings i.a. nicht zu Fehlern: in diesem Fall treten beim Beweis der Lemmas nur Teilziele auf, für die die Anwendbarkeit einer Theorieregel gleichzeitig die Gültigkeit der zugehörigen formalen Nebenbedingung impliziert.

7.6.2 Die zum Beweis der Lemmas verwendeten Taktiken

Die entwickelte Beweismethode enthält, jeweils getrennt nach mathematischen und Typlemmas, *Paare von Vorwärts- und Rückwärtstaktiken* sowohl zum Beweis der Konsistenzlemmas als auch der Verfeinerungslemmas[70].
Die angegebenen Taktiken sind derart gewählt, daß sie den Nachweis von Lemmas im *automatischen* Beweismodus des B-Tools unterstützen. In bestimmten Fällen kann sich jedoch eine *Benutzerinteraktion* als notwendig erweisen, wie die im nachfolgenden Abschnitt erfolgende Besprechung von Beispielbeweisen verdeutlicht; das B-Tool wird hierbei im *Beweisassistentenmodus* verwendet.
Folgendes Taktikpaar erwies sich zum Beweis sowohl der *mathematischen* als auch der *Typ*lemmas des *Konsistenz*nachweises als gute Heuristik:

```
(DED;empty~;hypdown;SUB;(DED;exfalsoquodlibet;support;HYP;hypdown;SUB)~)
,
(fempty~;GetPredicateX~;fsupport~;specialize~)~
```

Die obere der Taktiken ist die *Rückwärtstaktik*. Sie legt fest, daß nach einem initialen DED zunächst getestet werden soll, ob sich das Ziel mittels einer der Trivialtypbeziehungen der Theorie `empty` reduzieren läßt[71]. Im Anschluß daran wird wiederholt versucht, weitere Vereinfachungen, insbesondere mit den Regeln der für mathematische *und* Typlemmas entworfenen Theorie `support`), durchzuführen. Kombinationen von `hypdown`, SUB und DED erlauben die Auswertung von eventuell in der Hypothese erscheinenden Substitutionsausdrücken.
Die *Vorwärtstaktik* referenziert die obenbeschriebenen Vorwärtstheorien der Beweismethode sowie die von `devB` während der Verarbeitung der abstrakten Maschinen angelegte Theorie `GetPredicateX`, die die Daten der Invarianten und Operationspräkonditionen sowie weitere Implikationsbeziehungen enthält (vgl. Anhang D). Durch wiederholte Vorwärtsanwendung der "Regeln" dieser Theorie kann so der Inhalt der zunächst abstrakt beschriebenen Lemmahypothese konkretisiert werden.
Für die *Verfeinerungs*lemmas unterscheiden sich die verwendeten Taktiken. Das Taktikpaar zum Nachweis der *mathematischen* Verfeinerungslemmas ist von relativ einfacher Gestalt:

```
(DED;(refinementEqualities~;GetInvDefinitionX~;HYP~)~)
,
(fempty~;GetPredicateX~;fsupport~;GetDefinitionX~)~
```

[69] unter "vernünftig" ist hier zu verstehen, daß die Invariantenprädikate und die Ausdrücke in den Operationssubstitutionen jeweils selbst die syntaktischen Konventionen respektieren

[70] Da `devB` nicht die Angabe getrennter Taktiken für Konsistenz- und Verfeinerungslemmas unterstützt, sind die Verfeinerungstaktiken in der in Anhang F gezeigte Beweismethode in Kommentarklammern gesetzt.

[71] Für *mathematische* Lemmas sind diese Regeln nicht anwendbar.

Nach initialem DED spielt die Anwendung geeigneter Gleichungen der speziell für die mathematischen Verfeinerungslemmas entwickelten Theorie `refinementEqualities` die zentrale Rolle. Auf die Hypothese werden wiederum die allgemeinen Vorwärtsregeln der Theorie `fsupport` angewandt.
Das nach dem initialen DED vorliegende Ziel ist i.a. unter Berücksichtigung von *Definitionen* entstanden. Um dessen Rückführung auf die Hypothese zu ermöglichen, werden die Gleichungen der Theorie `GetDefinitionX` in Vorwärtsdeduktionen auf den Hypotheseninhalt angewandt. Die zu beweisenden Teilziele werden, sofern möglich, in inverser Richtung (d.h. gemäß Theorie `GetInvDefinitionX`) umgeformt.
Für den Beweis der Verfeinerungs*typ*lemmas wird das Taktikpaar

```
(DED;empty~;SUB;refinementsupport~;
  (exfalsoquodlibet;support;GetInvDefinitionX~;HYP~;SUB)~;
  (GetDefinitionX~;refinementsupport~)~;
  refinementEqualities~)~
,
(fempty~;GetPredicateX~;fsupport~;FwdSemanticsX~;specialize~)~
```

verwendet. In diesem Fall ist die Referenzierung der Spezialtheorie `refinementsupport` zentral. Die Gestalt der *Rückwärts*taktik entspringt einer Heuristik zur Berücksichtigung der Definitionsgleichungen bei der Reduktion der Teilziele auf die Hypothese.
Durch die *Vorwärts*taktik werden die *Kontextprädikate* in der Hypothese expliziert (Theorie `FwdSemanticsX`, vgl. Anhang D), da diese zum Beweis der Verfeinerungstyplemmas bedeutsam sein können.

7.7 Diskussion von Beispielbeweisen

Anhang G.2 zeigt Beweise einer Auswahl mathematischer und Typlemmas, die unter Verwendung der zuvor besprochenen Theorien und Taktiken durchgeführt wurden. Ein großer Teil der für das Beispiel generierten Lemmas (Anhang E) kann bewiesen werden.
Für einige Beweise ist noch Benutzerinteraktion notwendig[72]. Problematisch sind vor allem die Beweise der nicht in eine der generischen Klassen fallenden Lemmas:

- die aus den anwendungsspezifischen Integritätsbedingungen resultierenden Lemmas des Konsistenznachweises.
- die aus den den Erhalt komplexer Integritätsbedingungen sicherstellenden Substitutionen resultierenden Lemmas der Konsistenz- und Verfeinerungsbeweisverpflichtungen.

Darüberhinaus bedarf der Nachweis einiger "klassifizierbarer" Lemmas insbesondere der zweiten und dritten Verfeinerungsstufe der Benutzerassistenz, falls die vielen bestehenden Gleichheitsbeziehungen (Definitionen, CHANGE-Prädikate in der Hypothese) einen automatischen Beweis

[72] d.h. Verwendung des B-Tools im Beweisassistenten-Modus. Das B-Tool schlägt gemäß der Taktik die Anwendung von Regeln vor, die der Benutzer zu bestätigen hat (vgl. Kapitel 4).

unmöglich machen. Der Beweis von Typlemma Nr. 53 (vgl. Anhang G.2.1.10) ist ein Beispiel hierfür. Hier kann oftmals ein vom B-Tool nicht vollständig durchgeführter Beweis *komplettiert* werden, indem der Benutzer die "anzuwendenden" Gleichheitsbeziehungen auswählt.
Auch zeigt sich hierbei, daß das "Beweisverhalten" der Beweismethode selbst zwischen Lemmas ein und derselben Lemmaklasse differieren kann: der Beweis des in dieselbe Klasse fallenden Typlemmas Nr. 52 (vgl. Anhang G.2.1.9) wird *ohne* Benutzerinteraktion gefunden.
Anhang G.2 zeigt Beweise sowohl von generisch klassifizierbaren Lemmas als auch von Lemmas spezieller Gestalt. Zu jedem Beweis wird

- die *generische Klasse* des bewiesenen Lemmas (gemäß Abschnitt 7.5), sofern eine Einordnung vorliegt
- die verwendete, hinreichende *FWD-Quote* (Anzahl der nach jeder HYP-Anwendung gemäß der Vorwärtstaktik durchgeführten Vorwärtsregelanwendungen), sofern von Belang
- ggf. benötigte Benutzerassistenz

angegeben.
Für einige Lemmaklassen der dritten Verfeinerung enthalten die Theorien `refinementsupport` und `refinementEqualities` in der aktuell vorliegenden Beweismethode noch keine geeigneten Regeln; diese ließen sich jedoch, analog zu den Regeln für die Lemmaklassen der zweiten Verfeinerung, entwickeln und ergänzen.
Es sollen nun exemplarisch einige der in Anhang G.2 abgebildeten Beweise näher erläutert werden.

7.7.1 Beweise von Lemmas der Konsistenzbeweisverpflichtungen

Der Beweis des für den Konsistenzbeweis der Initialisierungsoperation nachzuweisenden mathematischen Lemmas Nr. 2

```
 ctx(ExtResearchCompaniesMch) &
 x,y: {} &
 {}(x) = {}(y)
=>
 x = y
```

wird in Anhang G.2.2.1 gezeigt. Der Beweis ist von unten nach oben zu lesen. Im ersten Schritt wird die vordefinierte Regel DED angewendet, durch die der Antezedent des implikationsförmigen Lemmas in die Hypothese gelangt (es wird nur der Teil der Hypothese gezeigt, der zum Beweis beiträgt):

```
1  x,y: {}                                          HYP
```

Durch Vorwärtsdeduktion gemäß der für mathematische Lemmas des Konsistenznachweises vereinbarten Vorwärtstaktik wird hieraus durch Anwendung der Vorwärtsregel `fempty.1`

```
a : {} => FALSE
```

die Formel `FALSE` abgeleitet. Dies besagt (vgl. o.), daß die aktuelle Hypothese widersprüchlich ist. Das unter Voraussetzung der nunmehr erweiterten Hypothese noch zu beweisende Teilziel `x = y` (Beweiszeile 3) wird im folgenden Rückwärtsschritt mittels der durch die Rückwärtstaktik spezifizierten Regel `exfalsoquodlibet.1` reduziert, wobei auf die zum Beweis beitragende Hypothesenformel `FALSE` (Zeile 2) verwiesen wird:

```
    ...
2    FALSE                                    1 fempty.1
3    x = y                                    2 exfalsoquodlibet.1
    ...
```

Anhang G.2.2.4 zeigt den Beweis des mathematischen Lemmas Nr. 11, innerhalb dessen die *Fallunterscheidungsregel* `support.7` zum Einsatz kommt. In Beweiszeile 27

```
27    (worksOn_1<+{ee|->works})(x) incl
      engagedIn_1((belongsTo_1<+{ee|->belongs})
      (x))                                      4 20 26 support.7
```

teilt sich der Beweis gemäß der Hypothesenformel

```
4  x: employees_1\/{ee}                        HYP
```

in zwei Unterfälle auf, die getrennt bewiesen werden:

```
20    x: employees_1 => (worksOn_1<+{ee|->works
      })(x) incl engagedIn_1((belongsTo_1<+{ee
      |->belongs})(x))
```

und

```
26    x: {ee} => (worksOn_1<+{ee|->works})(x)
      incl engagedIn_1((belongsTo_1<+{ee|->
      belongs})(x))
```

Zu beachten ist, daß für die beiden Teilbeweise *lokale Hypothesen* (gekennzeichnet durch Einrücken) verwaltet werden.

7.7.2 Beweise von Lemmas der Verfeinerungsbeweisverpflichtungen

Der Beweis des mathematischen Lemmas Nr. 34 (vgl. Anhang G.2.2.12) zeigt die Verwendung applikativer Regeln der Theorie `refinementEqualities`. Nach Anwendung der für Lemmas der Klasse **MV2E3** formulierten Regel `refinementEqualities.9` (Zeile 4) entsteht das Teilziel

```
3    %x.(x: dom(compClass_3)\/{newname} | x) =
     %x.(x: dom(compClass_3)\/{first(newname,
     ventures)} | x)                          2 refinementEqualities.31
```

welches unter Verwendung der applikativen Regeln für `first` in das als elementare Gleichheit *erfüllte* Ziel

```
1    %x.(x: dom(compClass_3)\/{newname} | x) =
     %x.(x: dom(compClass_3)\/{newname} | x)   EQL
```

umgeformt wird.
Anhang G.2.1.8 gibt ein Beispiel für die Anwendung der zum Beweis der *Typ*lemmas der Verfeinerung aufgestellten Regeln der Theorie `refinementsupport`. Die in Schritt 32 angewandte, bereits in Abschnitt 7.6.1.9 ausführlich diskutierte Regel `refinementsupport.7` ermöglicht den Beweis von Lemmas der Klasse **TV2E** (vgl. o.), wie in diesem Fall von Typlemma 50.
Die Verarbeitung des Teilziels

```
31   tail(newEmpId,newname,belongs,works):
     EmpNames*companies_3*POW(projects_3)
```

welches den Beweis der *Typkorrektheit* der neueingefügten abhängigen Attribute verlangt, verdeutlicht die Funktion der für das Hilfskonstrukt `tail` formulierten Gleichheitsaxiome (Beweiszeilen 28 bis 25). Das Teilziel

```
20   EmpNames*companies_3*POW(projects_3) =
     Strings*dom(compClass_3)*POW(dom(
     projClass_3))
```

verlangt den Nachweis der *Gleichheit* der Bildattributmengen. Es kann in diesem Fall unter Anwendung von Definitionsgleichungen und Hypothesen *direkt* reduziert werden (Zeile 17), obwohl Theorie `refinementsupport` die spezielle Regel

```
(A = L)  &  (B = M)     =>      (A * B) = (L * M)
```

zum Aufbrechen der Gleichheit kartesischer Produkte in *elementare* Mengengleichheiten anbietet, was für ähnliche Beweise u.U. notwendig sein kann.

7.8 Zur Technik der Theorieentwicklung

Bei der Entwicklung der zuvor diskutierten Beweismethoden wurden einige Erfahrungen gemacht, die in diesem Abschnitt in Form von Richtlinien zur Entwicklung von Regelsammlungen und zugehörigen Taktiken vorgestellt werden sollen. Hierbei erfolgt eine Differenzierung zwischen *allgemeinen* und DAIDA-*spezifischen* Vorgehensweisen.

7.8.1 Allgemeine Richtlinien

Zu den allgemein zweckmässigen Verfahrensweisen gehören:

- Bei der Formulierung von *Rückwärts*theorien sind allgemeinere (Implikations)Regeln oberhalb von spezielleren (Implikations)Regeln anzuordnen, da die Theorie von unten nach oben nach anwendbaren Regeln durchsucht wird. Dieser Punkt wurde bereits in Abschnitt 4.4.4 formal diskutiert.
- Ein weitgehender Verzicht auf operationale Lösungsansätze (vgl. o.) erhöht die Lesbarkeit der Regelsammlung ("*Deklarativität*" der entwickelnden Theorien).
- Vorwärtstheorien müssen sorgfältig entwickelt werden, um Endlosschleifen in Vorwärtsdeduktionen zu vermeiden (vgl. Abschnitt 4.2.2).
- Viele Regeln lassen sich sowohl in Rückwärtstheorien als auch in Vorwärtstheorien formulieren. Welcher Weg im Einzelfall einzuschlagen ist, ist sorgfältig abzuwägen:
  - umfangreiche Vorwärtstheorien führen zu großen Hypothesen. Vorwärtsableitungen werden mit wachsender Hypothese ineffizient. Die meisten abgeleiteten Hypothesenformeln werden nicht benötigt, um das aktuelle Teilziel zu beweisen. Die Durchsuchung der Hypothese dauert sehr lange.
  - wird eine Regel in einer *Rückwärtstheorie* formuliert, so ist sie lediglich dann anwendbar, wenn deren Regelkopf das aktuelle Teilziel matcht. Die so eingeschränkte Anwendbarkeit verkleinert den Suchraum (das Prinzip des *zielgerichteten Schließens*), führt jedoch u.U. in Sackgassen; da kein Backtracking stattfindet, wird so ein eventuell existierender alternativer Beweisweg nicht gefunden[73].

  Es sollte daher sorgfältig abgewogen werden, welche Regeln in Vorwärtsrichtung formuliert werden. Wegen des verwendeten Vorwärtsdeduktionsalgorithmus (Abschnitt 4.2.2) sollten Vorwärtsregeln wenige Antezedenten besitzen. Für Regeln mit Antezedensvariablen, die nicht im Konsequent vorkommen, kann eine Vorwärtsverwendung notwendig sein; es besteht jedoch u.U. die Möglichkeit, die betroffenen Antezedenten mit `inhyp`-Test zu versehen und die Regel rückwärts anzuwenden (vgl. folgenden Punkt).
- Die Definition 4.2 verdeutlicht, daß Regeln mit allgemeinem Konsequenten durch `inhyp`-Antezedenten in der Anwendbarkeit eingeschränkt werden können. Hiervon wird in der obigen Anwendung u.a. bei der Formulierung der `support`-Regeln zum Beweis der aus

[73] Durch Vorwärtsdeduktionen werden im Ggs. hierzu viele Deduktionen *parallel* vollzogen.

den Eindeutigkeitsbeziehungen **S1** und **S2** resultierenden Konsistenzlemmas Gebrauch gemacht.

7.8.2 DAIDA-spezifische Richtlinien

Für die oben diskutierte Problemstellung erweist es sich als zweckmäßig,

- Theorien und Taktiken wegen der aus der Gestalt der Beweisverpflichtungen resultierenden unterschiedlichen Form der Lemmas *vertikal* getrennt für Konsistenz- und Verfeinerungslemmas zu entwerfen. Regeln für mathematische und Typlemmas sind meist paarweise inkompatibel und müssen daher nicht in unterschiedliche Theorien separiert werden.
- wegen der beim Beweis der Lemmas entstehenden umfangreichen Hypothesen Vorwärts-Schlußregeln möglichst selektiv und in Abhängigkeit des vorliegenden Lemmatyps anzuwenden. Die Vorwärtsregeln sollten demnach in möglichst viele kleinere Vorwärtstheorien separiert und durch individuelle Taktiken möglichst gezielt angewandt werden[74].

7.9 Diskussion

Die in diesem Abschnitt besprochene Anwendung des B-Tools verdeutlicht dessen *unterschiedliche* Funktionen

- zur Verarbeitung der abstrakten Maschinen, Generierung und Verarbeitung der Beweisverpflichtungen[75] sowie als Abbildungsassistent der DAIDA-Sprachenhierarchie.
- als Deduktionssystem zum Beweis der DAIDA-Lemmas

Die vom B-Tool angebotenen Konzepte unterstützen beide Aufgaben und bieten spezielle Manipulationsmöglichkeiten für die Substitutionsnotation an. Zur Realisierung der Lemmabeweisumgebung stehen prinzipiell auch die mächtigen prozeduralen Konzepte zur Verfügung, mit Hilfe derer die Verarbeitung der abstrakten Maschinen umgesetzt wird.
Es stellt sich die Frage, ob es sinnvoll ist, die unterschiedlichen Funktionen getrennt zu realisieren und dabei ein möglichst effizient arbeitendes "herkömmliches" Deduktionssystem einzusetzen, welches auf vollständig formaler Ebene[76] arbeitet. Hier käme beispielsweise ein Deduktionssystem auf Basis der *Theorieresolution*, wie in [BlBu87] beschrieben, in Frage. Aufgrund der bereits zuvor besprochenen Bedeutung der Induktion für programmiersprachliche Konzepte ist allerdings eine Einschränkung der Betrachtungen auf Prädikatenlogik erster Ordnung problematisch. Eine Möglichkeit besteht darin, lediglich *bestimmte Instanzen* des Induktionsaxioms zu

[74] ein Weg, welcher im obigen Beispiel noch nicht eingeschlagen wurde
[75] bis hin zur Lemmaerzeugung
[76] Eine Schwäche der realisierten Regelsammlung, die uneingeschränkte Instantiierbarkeit der Metavariablen, wurde oben diskutiert.

berücksichtigen, für die ein induktives Schließen mittels *konstruktiver* Verfahren, d.h. *Algorithmen*, durchgeführt werden kann ([Hutt87])[77].
Ein alternativer Ansatz bestünde in der Verwendung *algebraischer Methoden*, für die bereits umfangreiche Erfahrungen aus dem Gebiet der verifizierten Programmentwicklung vorliegen: [Part90] gibt eine umfangreiche Sammlung algebraischer Theorien für unterschiedliche mathematische Konzepte an, die sich als formale Umgebung für die Programmentwicklung bewährt haben. Da in algebraischen Ansätzen Operatoren *sortenbezogen* vereinbart werden, tritt das in obigem Vorgehen bestehende Problem der uneingeschränkten Instantiierbarkeit der Metavariablen nicht auf.
Die Tatsache, daß das B-Tool keine der o.g. Aufgaben *direkt* löst, sondern lediglich zu deren *Realisierung* herangezogen wird, führt zu einer alternativen Sichtweise: das B-Tool kann als *Interpreter einer speziellen, regelorientierten Sprache* angesehen werden, bei der der Kontrollfluß mittels der Taktiksprache und weiterer Kontrollmechanismen (siehe Kap. 4) beeinflußt werden kann. Die Objekte, die durch die Regelvariablen gebunden werden, sind nicht unstrukturiert, sondern bezüglich einer vordefinierten Syntax wohlgeformte Ausdrücke. In dieser Rolle kann die Implementierung der o.g. Probleme auf dem B-Tool mit einer herkömmlichen programmiersprachlichen Lösung verglichen werden. Tabelle 7.6 gibt eine Übersicht dieses Vergleichs.

	AM-und BV-Verarbeitung, Abbildungsassistent	Deduktionssystem
B-Tool-Regel- und Taktiksprache	Syntaktisch strukturierte Eingaben vereinfachen die Verarbeitung. Substitutionskonzepte sind vordefiniert.	Automatisches Beweisen kann im Rahmen von Regel- und Taktiksprache unmittelbar realisiert werden.
herkömmliche (nichtlogische) Programmiersprache[78]	Scanner und Parser für die unterschiedlichen Beschreibungssprachen und das Substitutionskonzept müssen ausprogrammiert werden.	Formelrepräsentation, Deduktionsmechanismen und Steuerung müssen ausprogrammiert werden.

Tabelle 7.6: Vergleich der B-Tool-Sprache mit herkömmlichen Programmiersprachen

Die Betrachtung der zum Beweis der Lemmas verwendeten Beweismethode (vgl. Anhang F) zeigt ein wichtiges Problem auf: einige der Beweisregeln, insbesondere der Theorien `refinementEqualities` und `refinementsupport`, sind von relativ komplexer Gestalt. Es stellt sich die Frage, ob solche Regeln die vom "Theoriedesigner" intendierten Eigenschaften haben. Dies ist umsomehr problematisch, da

- die "Korrektheit" einer Regel i.a. erst anhand ihres Kontexts, d.h. anhand ihrer relativen Position und der weiteren Regeln ihrer Theorie unter Berücksichtigung der Suchreihenfolge des B-Tools zu beurteilen ist,

[77] Ein solcher Ansatz wird auch in dem bereits erwähnten Boyer-Moore-Ansatz ([BoMo79]) verwendet.
[78] d.h. der dritten Generation

- die Eigenschaften einer Regel im Falle der Verwendung "applikativer Tricks" (vgl. o.) schwer zu überschauen sind.

Eine alternative Vorgehensweise bestünde darin, auf die Verwendung von Regeln zu verzichten, deren Semantik "kontextabhängig" ist oder die sich applikative Tricks zunutze machen: ausgehend von einer elementaren, als konsistent *vorausgesetzten* Theoriensammlung[79] werden nur solche Axiome hinzugenommen, für die gezeigt werden kann, daß sie aus den bisherigen Regeln folgen. Auf diese Weise erhält man ein Axiomensystem, daß konsistent ist, falls das Ausgangsaxiomensystem konsistent ist. Der Vorteil dieses Ansatzes liegt darin, daß die vorgegebene, elementare Axiomenbasis wesentlich zuverlässiger "visuell" zu verifizieren ist als die oben formulierten komplexen Theorien.
Ein im Abschlußkapitel 8 besprochenes weiterführendes Projekt verfolgt einen derartigen Weg.

Zu untersuchen bleibt, inwieweit eine *vollständig formale* Umgebung für die Lemmabeweise effizient realisierbar ist. Dies könnte z.B. in Form einer Implementierung eines Resolutionsbeweisers auf dem B-Tool sowie, wie zuvor besprochen, durch eine Voraxiomatisierung einer "vernünftigen" Teilmenge der Mengentheorie, ergänzt um konstruktive Ansätze für Induktionsbeweise, geschehen[80]. Die oben durchgeführten Betrachtungen zeigen allerdings, daß die Entwurfs- und Zielsprachencharakteristika sowie die *regelbasierte* Transformation in die Notation der abstrakten Maschinen dazu führt, daß viele der zu beweisenden Lemmas in eine der oben identifizierten generischen Klassen fallen, für die es standardisierte Beweisregeln gibt. Dies ist ein starkes Argument für den hier verfolgten Ansatz.

[79] Es ist aus theoretischen Gründen nicht möglich, die Konsistenz von Axiomensystemen (außer in wenig mächtigen Logiken) allgemein zu entscheiden ([Ende72]).

[80] Das im folgenden Kapitel besprochene Projekt geht beispielsweise von einer (sehr restriktiven) *Theorie der endlichen Mengen* aus.

Kapitel 8

Zusammenfassung und Ausblick

Die in der Arbeit durchgeführten Betrachtungen hatten die Toolunterstützung bei der formal verifizierten Datenbankprogrammierung zum Gegenstand. Ausgehend von einer allgemeinen Darstellung der Gegenstände und Methoden des Software-Engineering wurde, aufbauend auf den Thesen und Ergebnissen der Arbeit [Wetz90], der Einsatz des Software-Werkzeugs *B-Tool* im Rahmen der DAIDA-Entwicklungsumgebung diskutiert. Während [Wetz90] eine allgemeine Diskussion von formalen Methoden zum Gegenstand hatte und das B-Tool vor allem in der Rolle eines *Abbildungsassistenten* zwischen den Sprachen TDL und "Abstrakte Maschinen" bzw. "Abstrakte Maschinen" und DBPL sowie zur *Generierung* der Beweisverpflichtungen beschrieb, lag der Schwerpunkt der vorliegenden Arbeit auf der Diskussion des B-Tools als *Theorembeweiser* zum *Nachweis* der Beweisverpflichtungen.
Im folgenden Abschnitt erfolgt eine kurze Rekapitulation der Ergebnisse der Diplomarbeit. Im Anschluß daran sollen einige noch offene Probleme sowie interessante Erweiterungen der DAIDA-Entwurfsumgebung diskutiert werden, die sich anhand der in der Diplomarbeit angestellten Betrachtungen identifizieren ließen. Ein abschließender Ausblick vergleicht den in der Arbeit beschrittenen Weg zum automatisierten Nachweis der Transaktionssicherheit mit dem in einem weiterführenden Projekt verfolgten Ansatz.

8.1 Zusammenfassung der Ergebnisse

Im *ersten Teil der Arbeit* wurde eine systematische Darstellung des B-Tools erarbeitet. Ausgehend von einer Schichtenarchitektur für Deduktionssysteme wurden die Besonderheiten der B-Tool-Architektur besprochen:

- die (relativ) freie Wahl einer Objektlogik und eines zugehörigen Kalküls
- die Möglichkeit der Definition einer *Formelsprache* (aufbauend auf bestimmten syntaktischen Defaults) durch den Benutzer, wobei die syntaktische Wohlgeformtheit objektlogischer Formeln *algorithmisch* überprüft wird.

Der Begriff des *offenen Deduktionssystems* wurde eingeführt. Die ursprünglich allein für das B-Tool konzipierte Metalogik ([Gard88]) erwies sich in ihrer allgemeinen Form als geeignet für die formale Grundlage beliebiger offener Deduktionssysteme; die B-Tool-Prälogik ergibt sich gemäß [Gard88] erst durch *zusätzliche Annahmen*. Im einzelnen wurden folgende Ergebnisse erarbeitet:

- Für die Metalogik wurde ein *semantischer Folgerungsoperator* definiert, womit eine in [Gard88] vorhandene formale Lücke geschlossen wurde.
- Es wurde bewiesen, daß eine alternative Sicht der Deduktionen im Metakalkül auf *Formelebene* des Metakalküls möglich ist. Metalogische Sequenzen verhalten sich wie *Schlußregeln*. Daraus folgt, daß der Metakalkül die "Simulation" des objektlogischen Schließens ermöglicht. Korrektheits- und Vollständigkeitseigenschaft des axiomatisierten Kalküls werden daher auf Metaebene reproduziert.
- Der Zusammenhang zu PROLOG wurde diskutiert: BL-Sequenzen können modelltheoretisch und beweistheoretisch gesehen werden. In BL axiomatisierbare Objektkalküle erlauben eine PROLOG-ähnliche Abarbeitung.
- Es wurde gezeigt, in welcher Weise das B-Tool sowie dessen Deduktionsmechanismen auf einer *Voraxiomatisierung eines Kalküls des natürlichen Schließens* in BL basieren. Eine alternative, nicht in [Gard88] erwähnte Sicht als "reines" offenes Deduktionssystem wurde diesem Blickwinkel gegenübergestellt.
- Die Kontrollmechanismen des B-Tools wurden, gemäß der Schichtenarchitektur für Deduktionssysteme, identifiziert und getrennt von weiteren, operationalen Komponenten dargestellt. Dies ermöglichte eine *formale Darstellung des B-Tools als Zustandsübergangssystem*.

Der *zweite Teil der Diplomarbeit* befaßte sich mit der theoretischen Basis der DAIDA-Entwicklungsumgebung. Neben einer Darstellung modellorientierter und algebraischer Methoden des Software-Enginnering lag der Schwerpunkt auf der Erarbeitung der *um zusätzliche Strukturierungskonzepte erweiterten Notation der abstrakten Maschinen*. Die hierfür resultierenden *Beweisverpflichtungen* wurden beschrieben und der theoretischen Darstellung gegenübergestellt. Darüberhinaus erfolgte eine Diskussion der für die folgenden Betrachtungen maßgeblichen Zielsprache DBPL sowie eine knappe Beschreibung der die Verarbeitung der abstrakten Maschinen realisierenden B-Tool-Regelsammlung `devB`.
Im *dritten Teil der Arbeit* wurde schließlich die konkrete Anwendung des B-Tools zum *Beweis* der Beweisverpflichtungen realisert. Dies umfaßte:

- Erarbeitung einer *größeren Beispielverfeinerung*, die Beispiele für die in Datenbankanwendungen typischen elementaren Einfüge-, Lösch- und Änderungstransaktionen enthält.
- Anhand der Beispielverfeinerung ließen sich bestimmte *Standard-Entwurfsentscheidungen* im Hinblick auf Spezifikations- und Zielsprache der DAIDA-Umgebung identifizieren. Es zeigte sich, daß die hochsprachlichen, deklarativen Sprachkonstrukte von DBPL die Gestalt der zu erbringenden Beweisverpflichtungen vereinfacht, da in vielen Fällen auf die Formulierung von Iterationen verzichtet werden kann. Dies bedingt allerdings entweder komplexe Transformationsregeln oder aber zusätzliche Verfeinerungsschritte.
- Ein *Definitionskonzept* wurde implementiert.

- Die aus den Konsistenz- und Verfeinerungsbeweisverpflichtungen der DAIDA-Entwicklungsumgebung resultierenden *Lemmas* wurden, ausgehend von einer *Beispielverfeinerung, gemäß ihrer generischen Gestalt klassifiziert*. Dies erwies sich, einerseits aufgrund der durch die Abbildungsregeln von der Entwurfssprache TDL in die Notation der abstrakten Maschinen relativ fixierten Gestalt der initialen abstrakten Maschine, andererseits aufgrund der durch die Gegebenheiten der relationalen Zielsprache festgelegten Standard-Entwurfsentscheidungen während der Verfeinerung, für eine große Anzahl der Lemmas als möglich. Es zeigte sich, daß die *Typ*lemmas des Konsistenznachweises, jedoch die *mathematischen* Lemmas des Verfeinerungsnachweises die wichtigsten Klassen generischer Lemmas bildeten. Die erarbeitete Klasseneinteilung ist unabhängig von der betrachteten DAIDA-Verfeinerung: die Klassifikation wurde derart erweitert, daß sie auch Lemmaklassen beinhaltet, die nicht im Rahmen des Beispiels vorkommen.
- Aufbauend auf der generischen Lemmaklassifizierung wurden in Form einer *Beweismethode* eine Menge von B-Tool-Theorien sowie geeignete Beweistaktiken formuliert, die den Nachweis von vielen der entstehenden Lemmas ermöglichen. Da die Regeln lediglich auf die *generische* Gestalt der Lemmas Bezug nehmen, sind diese *unabhängig von einer konkreten Datenbankanwendung* verwendbar, sofern die Transaktionen von allgemeiner Gestalt sind.

Die Erfahrungen, die beim Erstellen der Regelsammlung zum Beweis der Lemmas gemacht wurden, sind in einem gesonderten Abschnitt dokumentiert und diskutiert worden.

8.2 Interessante Erweiterungen

Neben den im vorigen Abschnitt zusammengefaßten Ergebnissen können anhand der Ausführungen der Arbeit einige offene Probleme sowie interessante Erweiterungen identifiziert werden:

- Es ist zu untersuchen, inwieweit das B-Tool auch zur Realisierung eines *vollständig formalen Ansatzes* zum Beweis der Lemmas verwendet werden kann; dies beinhaltet u.a. die bereits oben identifitzierte Problemstellung der Identifikation einer geeigneten Basistheorie und der Ergänzung von Regeln, die in diesem Rahmen *beweisbar* sind und die Lemmabeweise unterstützen.
- Die bisher formulierten Beweisregeln unterstützen lediglich den Beweis von Lemmas, die für *Elementaroperationen* resultieren. Es ist zu untersuchen, inwieweit sich dieser Ansatz auf nichtelementare Operationen verallgemeinern läßt. Dies könnte in Form einer erweiterten Fallstudie geschehen, die zunächst weitere charakteristische, jedoch im obigen Sinne nicht als elementar zu bezeichnende Einfüge-, Lösch- und Änderungsoperationenschemas identifiziert und die Lemmaklassifikation entsprechend erweitert.
- Es ist zu untersuchen, inwieweit die in Kapitel 6 identifizierten, zusätzlichen Strukturierungskonzepte der erweiterten abstrakten Maschinen auch in der DAIDA-Umgebung ausgenutzt werden können. Bei der Beschränkung der dynamischen Systemkomponenten auf *Transaktionen* erscheint dies wegen deren üblicherweise relativ elementaren Gestalt (vgl. Abschnitt 1.3) allerdings ad hoc als wenig sinnvoll.

- Die Funktionen des B-Tools als *Abbildungsassistent* sind für die nunmehr erweiterte Syntax der abstrakten Maschinen zu realisieren. Die in [Wetz90] exemplarisch angegebenen Regeln für die automatische Transformation der finalen abstrakten Maschinen nach DBPL sind zu vervollständigen und auf dem B-Tool zu implementieren. Gemäß der Diskussion in Abschnitt 7.3.4 beinhaltet diese Aufgabe u.a. die *Identifikation einer "Normalform" einer direkt nach DBPL transformierbaren abstrakten Maschine*, deren Elementarzuweisungen mittels elementarer syntaktischer Regeln in DBPL-Mengenausdrücke übersetzt werden können.

- Es ist zu untersuchen, in welcher Weise zusätzliche Software-Tools bestimmte Aufgaben der Entwicklungsumgebung übernehmen bzw. vereinfachen können. Interessant wäre beispielsweise der Einsatz *sprachsensitiver Editoren* für die unterschiedlichen Beschreibungssprachen der Umgebung.
  Für DBPL ist ein derartiges Werkzeug bereits realisiert ([Nieb89]). Für die Erstellung umfangreicherer *abstrakter Maschinen*, insbesondere mit den in Kapitel 6 vorgestellten Strukturierungskonzepten, sollte sich ein solches Werkzeug ebenfalls als nützlich erweisen. Dies gilt auch für die Formulierung von Beweismethoden, insbesondere dann, falls komplexe, schlecht überschaubare Regelsammlungen (wie im Fall der in dieser Arbeit erstellten Beweismethode) formuliert werden.

Beispielsweise basiert der im *NuPrl-Projekt* ([Cons86]) verfolgte Ansatz auf der Verwendung *sprachsensitiver, regelgesteuerter Editoren* für unterschiedliche Aufgaben einer formalen Beweisumgebung[1]. Wie beim B-Tool werden sowohl automatische als auch benutzergesteuerte Deduktionen unterstützt. Eine wichtige Komponente stellt hierbei ein sog. *refinement editor* dar, mit Hilfe dessen eine *interaktive, zielgerichtete Beweisentwicklung* in dem vom NuPrl-System gesteckten formalen Rahmen möglich ist[2]. Im "*refinement editor*" können, ausgehend von einem zu beweisenden Ziel als Wurzel, interaktiv partielle Beweisbäume erstellt und gemäß *zulässiger* Deduktionsregeln erweitert werden. Der Editor ist bezüglich der syntaktischen Struktur zulässiger *Beweisbäume* sprachsensitiv. Die Syntax der eingegebenen Formeln wird, sofern diese nicht automatisch (durch Regelanwendung) hervorgehen, durch integrierte *Parser* kontrolliert.
Der Benutzer kann im *refinement editor* durch entsprechende Edierbefehle das aktuelle Teilziel frei wählen, während das B-Tool selbst im Beweisassistentenmodus enge Grenzen gemäß der vorgegebenen Taktik setzt. Eine entsprechende Erweiterung wäre auch für das B-Tool interessant: die *strukturierte Aufbereitung von Beweisen* in Form von Beweisbäumen und das Verwalten in einer *Beweisbibiothek* könnte die Grundlage für eine *Wiederverwertbarkeit von Teilbeweisen* sein.

[1] Die NuPrl-Umgebung basiert auf dem *funktionalen, konstruktiven Ansatz nach Boyer und Moore* ([BoMo79]) und vereinheitlicht Komponenten zur Auswertung funktionaler Ausdrücke, Editoren für eine funktionale *Objektsprache* und für die algorithmische Metasprache ML (zur Beschreibung von Beweisstrategien), Bibliotheksfunktionen sowie die durch einen Beweiseditor realisierte Beweisumgebung. D.h. NuPrl ist genaugenommen mehr als nur ein Beweisasistent.

[2] Die Bezeichnung "refinement" bezieht sich demnach in diesem Fall auf die "Verfeinerung" unvollständiger *Beweise*.

8.3 Ausblick: das ADABTPL-Projekt

Das ADABTPL-Projekt[3] wurde an der Universität von Massachusetts, Amherst, initiiert. Die Zielsetzungen überschneiden sich mit denen des (mittlerweile abgeschlossenen) ESPRIT-Projekts DAIDA: es soll die Technik der Entwicklung datenintensiver Anwendungen verbessert werden. Gemäß [StSh90] umfaßt dies u.a.

- die Entwicklung einer formalen Spezifikationstechnik für Datenbankanwendungen,
- die formale Verifikation der "Konsistenz" von Spezifikationen[4] unter Verwendung von Deduktionssystemen,
- Eine Technik zum Transformieren von Spezifikationen in effiziente Implementationen.

Es soll nun ein kurzer Überblick über die Gegenstände des ADABTPL-Projekts gegeben werden: zunächst wird besprochen, welcher Weg zum Beweis der Spezifikationskonsistenz eingeschlagen wird und welche Gemeinsamkeiten bzw. Unterschiede zum im Rahmen dieser Arbeit verfolgten Ansatz bestehen. Im anschließenden Abschnitt wird ein weiterer Gegenstand des ADABTPL-Projekts kurz beschrieben: die Verwendung der "typreflektiven" Sprache TRPL[5], die es insbesondere gestattet, eine *statische Typkontrolle*[6] für generische Datenbankoperationen der Relationenalgebra zu realisieren. Abschließend werden einige spezifische Zielsetzungen des ADABTPL-Projekts identifiziert und diskutiert, wie eine Nutzung der in DAIDA gemachten Erfahrungen im Rahmen einer möglichen Kooperation aussehen könnte.

8.3.1 Konsistenznachweise im ADABTPL-Ansatz

Im Unterschied zum DAIDA-Ansatz umfaßt die formale Verifikation im Rahmen des ADABTPL-Projekts bisher lediglich den Nachweis der "Sicherheit" von Transaktionen[7]. Diese Zielsetzung entspricht gerade dem *Konsistenz*nachweis im Rahmen von DAIDA: es ist zu zeigen, daß jede spezifizierte Transaktion, ausgeführt in einem die spezifizierten Integritätsbedingungen erfüllenden Datenbankzustand, die Gültigkeit der Integritätsbedingungen *erhält*. *Implementations*fragen (und damit die in Rahmen dieser Diplomarbeit untersuchten Fragestellungen des Software-Engineering) sollen Gegenstand weiterer Untersuchungen des laufenden Projekts sein.
Die wesentlichen Aspekte der formalen Verifikation der Transaktionssicherheit im ADABTPL-Projekt faßt die Arbeit [ShSt89] zusammen. *Spezifikationen* werden in der Sprache ADABTPL beschrieben und beinhalten eine prozedurale Darstellung von Transaktionen, eine Beschreibung des Datenbankschemas sowie der Integritätsbedingungen des Datenbanksystems. Die verwendeten Systemkomponenten (Werkzeuge) sind

[3] Das Akronym ADABTPL steht für *Abstract DAtaBase Type Programming Language*.
[4] Es wird sich zeigen, daß unter *Konsistenz* hier genau dasselbe wie in DAIDA zu verstehen ist.
[5] TRPL steht für *Type Reflective Programming Language*
[6] d.h. Typkontrolle zur Compilezeit
[7] engl. *transaction safety*

- ein *Theorembeweiser*, basierend auf einer Ergänzung des konstruktiven, applikativen Boyer-Moore-Ansatzes [BoMo79] um spezielle Axiome für endliche Mengen (dies ist insbes. wegen der nicht in Logik erster Ordnung abbildbaren Konzepte höherer Ordnung bedeutsam, die sich, sofern (individuell) *konstruktiv* axiomatisierbar, im Rahmen der Boyer-Moore-Logik beschreiben lassen)

- ein *Schema-Übersetzer*, der ADABTPL-Spezifikationen verarbeitet und anwendungsspezifisches Wissen extrahiert (vergleichbar mit der Verarbeitung von DAIDA-Spezifikationen durch `devB`),

- ein *Konsistenzbeweiser*, der, aufbauend auf einer sowohl allgemeines als auch anwendungsspezifisches Wissen enthaltenden *Wissensbank*, die Korrektheitsbeweise mit Hilfe des Boyer-Moore-Beweisers durchführt

- ein *Testgenerator*, der die unbeweisbaren Teilziele der Konsistenznachweise dazu verwendet, *sicherheitsgarantierende* Laufzeittests in den Transaktionen zu ergänzen[8].

Um die Konsistenzbeweise für eine möglichst breite Klasse von Transaktionen zu unterstützen, wird eine anwendungsunabhängige, *generische* Theorie entwickelt, die sich auf die Identifikation einer Grundmenge von typischen Integritätsprädikaten und Datenbankmodifikationsoperationen[9] begründet: zulässig sind

- *Integritätsbedingungen*, formuliert mit Hilfe von Relationen und Operatoren über den *natürlichen Zahlen*, Element- und Enthaltenseinsbeziehungen, All- und Existenzquantifizierung, Disjunktheit, Redundanzbeziehungen, Aggregate (Summen- und Zählfunktionen) über Relationenattributen

- fünf Arten von elementaren *"Updates"* (Einfügen und Löschen einzelner Tupel, multiple Einfügungen, prädikativ gesteuerte Änderungen, prädikativ gesteuertes, selektives Ändern oder Löschen) sowie die *beliebige Kombination* dieser Elementaroperationen durch schachtelbare Konditionale.

Mit Hilfe dieses nichttrivialen Repertoires sollte der größte Teil der üblichen Datenbankanwendungen beschreibbar sein.
Hier fällt ein wesentlicher Unterschied zum Vorgehen im Rahmen dieser Diplomarbeit auf: während im ADABTPL-Projekt eine *generelle* Basis für die Spezifikation von Datenbankanwendungen identifiziert wurde, zeigt Kapitel 7 dieser Arbeit lediglich eine *Fallstudie*, deren Resultat (die formulierten Theorien) aufgrund der restriktiven Einschränkung des Operations- und Integritätsprädikatenrepertoires weniger generell ist als das des ADABTPL-Ansatzes. Das allgemeine Vorgehen des ADABTPL-Projekts macht sich zunutze, daß die Spezifikationssprache ADABTPL relativ nahe an einer Notation einer implementierbaren Datenbanksprache ist, sodaß eine "vernünftige" Grundmenge von Updateoperationen identifiziert werden kann.
Der nächste Schritt besteht in der *Formulierung von Theorien*, die das zur Verifikation benötigte formale Schließen, aufbauend auf einem Boyer-Moore-Beweiser, unterstützen. Der Aufbau einer derartigen formalen Basis umfaßt die stufenweise Erstellung folgender Komponenten:

[8] Im DAIDA-Ansatz *fehlt* eine derartige "werkzeuggestützte Rückkopplung".
[9] d.h. Einfügen, Löschen, Ändern

1. Axiomatisierung einer Theorie *endlicher Mengen*: dies geschieht (in algebraischem Stil) durch die Angabe von Axiomen für die Grundfunktionen `emptyset`, `insert`, `choose`, und `rest` über den Sorten `fsets`, `elements` und `boolean`[10].
   Darüberhinaus sind die Sorten `boolean` und `natural` sowie die zugehörigen Operatoren voraxiomatisiert.

2. Die allgemeine Theoriebasis kann nun durch den Beweis elementarer Beziehungen erweitert werden, was künftige Beweise abkürzt.

3. Aufbauend auf dieser Grundlage können nun weitere Funktionen wie `MEMBER`, `UNION` und `DELETE` oder generische Quantifizierungen $\forall(S,P)$, $\exists(S,P)$[11] definiert werden, wie sie im Kontext von Datenbankspezifikationen vorkommen. Bedeutsam ist hierbei, daß jede Definition "konstruktiv" ist, d.h. sofern rekursiv, auf eine Rekursionsfundierung rückführbar ist[12].

4. Über dieser *kombinierten* Theorie können nun weitere allgemeine Resultate bewiesen und zur Theoriebasis ergänzt werden.

Die im letzten Schritt bewiesenen und zur Wissensbasis hinzugefügten Beziehungen werden in der ADABTPL-Terminologie als *Lemmas* bezeichnet[13]. Aufgabe eines "*Knowledge Engineers*" ist es, die zur Wissensbasis hinzuzufügenden Lemmas sinnvoll auszuwählen, da dies als ausschlaggebend für die Effizienz der Verifikationssnachweise angesehen wird.
Nach [ShSt89] ist die auf diese Weise entstehende Wissensbank jedoch noch nicht geeignet, die Konsistenz einer individuellen Anwendungsspezifikation nachzuweisen, da die Lemmas von zu allgemeiner Gestalt sind. Andererseits sollte der die Spezifikation erstellende Datenbankentwickler von der Pflicht befreit sein, anwendungsspezifische Theorien aufzustellen. Dies wird in [ShSt89] als Motivation für die im ADABTPL-Projekt vorgenommene Erweiterung des Boyer-Moore-Ansatzes um spezielle *Metakonzepte* wie *Metafunktionen* und *Metalemmas* genannt. Als Beispiel wird die *generische Formulierung einer UPDATE-Funktion* angegeben. Die zuvor identifizierten Modifikations-Grundtypen lassen sich zum großen Teil als *Instanzen* dieser Funktion ansehen. Da das generische UPDATE Funktionen und Prädikate als Parameter hat, handelt es sich um eine *Funktion höherer Ordnung*.
Die Bedeutung von Metafunktionen liegt in der Art, in der zugehörige Lemmas verwendbar sind. Die erweiterte Boyer-Moore-Umgebung des ADABTPL-Ansatzes unterstützt den Beweis und die Verwaltung derartiger *Metalemmas*. Diese sind von der Gestalt[14]

```
(implies MH (implies H (equal L R)))
```

`MH` ist die sog. *Metahypothese*, der *Konsequent* des Metalemmas gibt eine unter der Bedingung `H` anwendbare Termersetzung an. Der *Beweis* von Metalemmas geschieht unter der zusätzlichen,

[10] Aus *konstruktivistischen* Gründen muß jeder endlichen Menge eine Ordnung zugrundeliegen, was durch die Axiomatisierung weiterer (nur intern sichtbarer) Funktionen `before` und `smaller` formalisiert wird.

[11] als Prädikate höherer Ordnung, stehend für die All- bzw. Existenzquantifizierung eines als Parameter gegebenen Prädikats P über einer als Parameter gegebenen Menge S

[12] aus diesem Grund sind die generischen Quantifizierungen problemlos, da sie wegen der auf endlichen Mengen definierten Ordnung konstruktiv definierbar sind

[13] Diese Bezeichnung ist hier zutreffend, da es sich um *bewiesene* Elementarresultate handelt.

[14] formuliert in dem vom Boyer-Moore-Beweiser unterstützten applikativen Stil

"lokalen" Annahme der Metahypothese. Wird nun eine Funktion als *Instanz* einer generischen Funktion definiert, so "erbt" diese die entsprechenden *Instanzen* der rechten Seiten der zugehörigen Metalemmas, sofern die Metahypothese unter den Instantiierungen bewiesen werden kann. Außer einer derartigen *Anwendung zur Definitionszeit* existiert die Möglichkeit der Anwendung von Metalemmas während laufender Beweise ("*on demand*") sowie zur *Schema-Verarbeitungszeit* ([ShSt89]).
Auf diese Weise können *anwendungsspezifische* Lemmas automatisch (in Form von Instanzen von Metalemmas) zur anwendungsunabhängigen Wissensbasis hinzugefügt werden, ohne daß der Systementwickler involviert wird.
Das Vorgehen zum *Aufstellen und Beweisen der Konsistenzbeweisverpflichtungen* weist (von der Art der durchzuführenden Elementarschritte) Parallelen zur Arbeitsweise der DAIDA-Regelsammlung `devB` auf: die Konsistenznachweise der einzelnen Transaktionen lauten

```
(implies (P db) (P (T input db)))
```

wobei `P` durch das Konjunkt aller Integritätsbedingungen und `T` durch den Rumpf der Transaktionsspezifikation expandiert wird.
Die Einzelschritte sind

1. *Formulierung* der Beweisverpflichtungen (wie zuvor beschrieben)
2. Expandierung der im Datenbankschema formulierten Typprädikate (als Grundlage für den weiteren Beweis)
3. Elimination von nicht durch die Transaktion tangierten Integritätsbedingungen (engl. *elimination of inertia*); in DAIDA entspricht dies der Reduktion von unter der Operationssubstitution nicht veränderten Integritätsbedingungen durch die Regeln AND und HYP (vgl. Abschnitt 7.5.1.1)
4. Identifikation der auf diese Weise *nicht reduzierbaren* Teilziele (DAIDA-Terminologie: *Lemmas*) und getrennte Verarbeitung derselben durch folgende Schritte:
   (a) Metalemma-Anwendung
   (b) Standardlemma-Anwendung und Funktionsauswertung

Die *unbewiesenen Teilziele* werden dem Testgenerator zugeführt.

8.3.2 Typreflektion und statische Typkontrolle

In einer folgenden Projektphase wurden die Zielsetzungen für die Sprache ADABTPL erweitert: in dem zuvor diskutierten Anwendungsbereich diente ADABTPL zur Formulierung von Systembeschreibungen, die zwischen einer Systemmodellierung in einer *Entwurfssprache* und der *Implementation* des Systems angesiedelt sind. Im nunmehr erweiterten ADABTPL-Rahmen soll die Sprache nicht nur zur aufbereiteten Formulierung von Spezifikationen, sondern auch als *Implementationssprache* mit *strenger Typisierung* Verwendung finden ([StSh90]).

Das zugrundeliegende *Typsystem* muß in der Lage sein, auch die Typkontrolle für *generische Operationen*, wie beispielsweise für den *Natural Join* der Relationenalgebra, durchzuführen. Das Problem liegt darin, daß der Ergebnistyp derartiger Operationen von den Typen der Eingabe abhängt: für die Ermittlung des Typs der Ergebnisrelation einer Anwendung der Operation *Natural Join* müssen anhand der Eingaberelationen die Menge der sich überschneidenden sowie die nur in einer der Eingaberelationen vorkommenden Komponenten*namen* identifiziert werden, bevor der Ausgabetyp ermittelt werden kann.
Gegenstand der Untersuchungen des ADABTPL-Projekts ist u.a., Typkontrollen für derartige Operationen *statisch*, d.h. zur *Übersetzungs*zeit, durchzuführen ([SFSS90]). Der Vorteil gegenüber einer dynamischen Typkontrolle liegt darin, daß kein Laufzeitzugriff auf die Schemainformation erfolgen muß und dementsprechend die Abarbeitungsgeschwindigkeit erhöht wird. Als Basis hierfür wurde die Sprache TRPL entwickelt, die es erlaubt, die Berechnungen, die zur *statischen* Typkontrolle derartiger Operationen durchzuführen sind, zu beschreiben. Die Sprache ermöglicht die statische *Reflektion* auf die Typinformation, die in einem bestimmtem Abarbeitungszustand einer Übersetzung vorliegt. Für den generischen natürlichen Verbund kann so auf die (bereits vorliegenden) Typen der Eingaberelationen zugegriffen werden, um den Ergebnistyp zu "synthetisieren".
Die Arbeit [SFSS90] zeigt, ausgehend von einer Identifikation der auftretenden Probleme, in welcher Weise die statische Typkontrolle für den natürlichen Verbund sowie für weitere, als problematisch anzusehende Operationen wie "Nesting" in der NF^2-Relationenalgebra, im Rahmen von TRPL realisiert werden kann.
Statische Typkontrolle wird auch in anderen Datenbanksprachen wie beispielsweise DBPL realisiert. Der Vorteil des im ADABTPL-Projekt verfolgten Wegs liegt darin, daß es mit Hilfe sog. *TRPL-Makros* möglich ist, die (statische) Typkontrolle für ein breites Spektrum generischer Operationen zu realisieren; man ist nicht mehr an ein fest vorgegebenes Repertoire derartiger Operationen gebunden. Dieser Ansatz wird daher auch als Möglichkeit angesehen, den Grad des *Polymorphismus* in Datenbanksprachen zu erhöhen ([SFSS90], "*Exceeding the Limits of Polymorphism in Database Programming Languages*").
Die Sprache TRPL erlaubt es, die Typkontrolle der Sprache ADABTPL auf der Basis eines Typsystems niedrigerer Komplexität, ergänzt um Schemazugriff zur Übersetzungszeit, zu realisieren. Auf diese Weise kann die Sprache ADABTPL *implementiert* werden, indem TRPL-Makros geschrieben werden, die eine "Übersetzung" von ADABTPL-Spezifikationen in Sprachen wie DBPL realisieren. Dies ist Gegenstand weiterer Untersuchungen ([StSh90]).

8.3.3 Einbringen von Erfahrungen aus dem DAIDA-Projekt

Die Problemstellungen des ADABTPL-Projekts

- formale Verifikation der *Konsistenz* abstrakter Spezifikationen von Datenbankanwendungen
- (verifizierte) Entwicklung der Implementation, ausgehend von einer Spezifikation in ADABTPL
- Axiomatisierung geeigneter Theorien, die die formale Verifikation ermöglichen

überschneiden sich mit entsprechenden Zielsetzungen von DAIDA, wobei allerdings ein wesentlicher Unterschied besteht:

> *Im ADABTPL-Projekt soll die Sprache ADABTPL als* integrierte Spezifikations- und Implementationssprache *eingesetzt werden. DAIDA hingegen basiert auf einem* mehrschichtigen *Ansatz, der zwischen Modellierungs-, Verfeinerungs- und Implementationssprache unterscheidet.*

Gegenstand weiterer Untersuchungen im Rahmen des ADABTPL-Projekts könnte, wie bereits erwähnt, die Erörterung der Frage sein, in welcher Weise sich Datenbankprogrammiersprachen wie DBPL als *Zielsprache* zur Implementierung von ADABTPL-Spezifikationen in die ADABTPL-Umgebung integrieren lassen, wobei die Übersetzungsvorschriften in der Sprache TRPL beschrieben werden. Eine entsprechende Erweiterung des ADABTPL-Ansatzes ist auch auf der Entwurfsseite in Form des Einsatzes von Wissensrepräsentationssprachen wie Telos denkbar. Zu überdenken ist hierbei allerdings, inwieweit die zur formalen Verifikation verwendete Theoriebasis zu erweitern ist, daß auch die Korrektheit der einzelnen Übersetzungsschritte nachgewiesen werden kann.
Im Rahmen einer *Kooperation* mit den Initiatoren des ADABTPL-Projekts könnten derartige Problemstellungen, insbesondere aber die Integration von DBPL in den ADABTPL-Rahmen, untersucht werden.

Verweis auf den Anhang

Zur vorliegenden Diplomarbeit gehört ein *Anhang*, dessen Inhalt an dieser Stelle überblicksweise beschrieben werden soll.

- **Anhang A** zeigt die Datei `SYMBOL`, mit Hilfe derer dem B-Tool *syntaktische Vereinbarungen* vorgegeben werden. Die abgebildete Datei enthält die Vorgaben, die für den Einsatz des B-Tools in der DAIDA-Umgebung zu treffen sind.
- **Anhang B** gibt einen Überblick über die Theoriensammlung `devB`, die die Verarbeitung abstrakter Maschinen auf dem B-Tool realisiert. Zu jeder Theorie wird die Anzahl der enthaltenen Regeln angegeben.
- **Anhang C** zeigt die Folge abstrakter Maschinen der *DAIDA-Beispielverfeinerung*, die als Ausgangspunkt für die Lemmaklassifikation dient.
- **Anhang D** gibt eine Auflistung der "Theorien", in denen die während der Verarbeitung der abstrakten Maschinen des Beispiels extrahierte Information abgelegt wird. Hiermit ist es möglich, die durchgeführten Beweise (vgl. Anhang G) nachzuvollziehen, da die dort verwendeten Taktiken diese Theorien referenzieren.
- **Anhang E** zeigt die für das Verfeinerungsbeispiel erzeugten *mathematischen* und *Typlemmas*, aufgegliedert nach den unterschiedlichen Verfeinerungsebenen.
- **Anhang F** zeigt die *Beweismethode*, die die Theorien und Taktiken zum Beweis der Lemmas enthält und das Definitionskonzept realisiert.
- **Anhang G** zeigt *Beispielbeweise* von mathematischen und Typlemmas, die mit Hilfe der vorgestellten Beweismethode durchgeführt wurden, sowie die Verarbeitung einer Konsistenzbeweisverpflichtung.

Literaturverzeichnis

[Abri84] J.R. Abrial. *Programming as a Mathematical Exercise*. In: [HoSh86].

[Abri86] J.R. Abrial. *An Informal Introduction to B*. Technical Report, 26 Rue des Plantes, Paris 75014, November 1986.

[Abri88] J.R. Abrial. *A Set Theoretic Model for the Generalized Substitution Notation*. Technical Report, 26 Rue des Plantes, Paris 75014, August 1988.

[Abri89] J.R. Abrial. *A Formal Approach to Large Software Construction*. In: J.L.A. van der Snepscheut (Ed.). Mathematics of Program Construction. Lecture Notes in Computer Science 375, Springer-Verlag 1989.

[Abri89a] J.R. Abrial. *Introduction to Set Notations (Part 1: Basic Concepts)*. Buch-Preprint, 26 Rue des Plantes, Paris 75014, Mai 1989.

[Abri89b] J.R. Abrial. *Introduction to Set Notations (Part 2)*. Buch-Preprint, 26 Rue des Plantes, Paris 75014, 1989.

[Abri90a] J.R. Abrial. *Abstract Machines (Part 1: Basic Concepts)*. Buch-Preprint (Umfaßt [AGMS88a]), 26 Rue des Plantes, Paris 75014, August 1989.

[Abri90b] J.R. Abrial. *Abstract Machines (Part 2: Programming Concepts)*. Buch-Preprint (Umfaßt [AGMS88b]), 26 Rue des Plantes, Paris 75014, 1989.

[Abri90c] J.R. Abrial. *Abstract Machines (Part 3: Refinement. Theory)*. Buch-Preprint (vgl. [AGMS88c]), 26 Rue des Plantes, Paris 75014, Januar 1990.

[Abri90d] J.R. Abrial. *Abstract Machines (Part 3: Refinement. Practice)*. Buch-Preprint (vgl. [AGMS88c]), 26 Rue des Plantes, Paris 75014, Februar 1990.

[AGMS88a] J.R. Abrial, P. Gardiner, C.C. Morgan, J.M. Spivey. *Abstract Machines (Part 1: Basic Concepts)*. Technical Report, 26 Rue des Plantes, Paris 75014, Juni 1988.

[AGMS88b] J.R. Abrial, P. Gardiner, C.C. Morgan, J.M. Spivey. *Abstract Machines (Part 2: Programming Concepts)*. Technical Report, 26 Rue des Plantes, Paris 75014, Juni 1988.

[AGMS88c] J.R. Abrial, P. Gardiner, C.C. Morgan, J.M. Spivey. *Abstract Machines (Part 3: Refinement)*. Technical Report, 26 Rue des Plantes, Paris 75014, Juni 1988.

[BaGo79] R. Balzer, N.Goldman. *Principles of Good Software Specification and Their Implications for Specification Languages*. Proceedings of IEEE Conference on Specifications of Reliable Software, pp 58-67. Auch in [GeGe86].

[Beck88] Clemens Beckstein. *Zur Logik der Logik-Programmierung. Ein konstruktiver Ansatz*. Informatik-Fachbericht 199, Springer-Verlag 1988.

[BJLM87] A. Blaser, M. Jarke, H. Lehmann, G. Müller. *Datenbanksprachen und Datenbankbenutzung*. Kapitel 6 von [LoSc87].

[BlBu87] K.H. Bläsius, H.-J. Bürckert (Hrsg.). *Deduktionssysteme*. Oldenbourg-Verlag, 1987.

[BKMS89] A. Borgida, M. Koubarakis, J. Mylopoulos, M. Stanley. *Telos. A Knowledge Representation Language for requirements Modeling.*. Technical Report KRR-TR-89-4, Dept. of Computer Science, University of Toronto, Februar 1989.

[BMSW89] A. Borgida, J. Mylopoulos, J.W. Schmidt, I. Wetzel. *Support for Data-Intensive Applications: Conceptual Design and Software Development*. Preliminary Report. EG-Projekt 892, ESPRIT-DAIDA. (auch erschienen in: R. Hull, R. Morrison, D. Stemple (eds.). Proceedings of the Second International Workshop on Database Programming Languages, Salishan, Oregon, S.258-280. Morgan Kaufman, 1989)

[BoMo79] R.S. Boyer, J.S. Moore. *A Computational Logic*. Academic Press, 1979.

[BrMS84] M.L. Brodie, J. Mylopoulos, J.W. Schmidt (Eds). *On Conceptual Modelling. Perspectives from Artificial Intelligence, Databases and Programming Languages*. Springer-Verlag, 1984.

[Buer87] H.-J. Bürckert. *Deduktion als Berechnung*. Kapitel 4 von [BlBu87].

[BuFe78] R.M. Burstall, M.S. Feather.*Program development by transformation: an overview*. In: M. Amirchahy, D. Neel (eds.): Les fondements de la programmation. Proc. Toulouse CREST Course on Programming., IRIA-SEFI, Le Chesnay, France, 1978.

[BuGo81] R.M. Burstall, J.A. Goguen. *An Informal Introduction to Specifications Using CLEAR*. In: The Correctness Problem in Computer Science. S. Boyer, J.S. Moore (Eds), Academic Press, pp. 170-189. Auch in [GeGe86].

[ChLe73] C.-L. Chang, R.C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[ClMe87] W.F. Clocksin, C.S. Mellish. *Programming in PROLOG*. 3rd Edition. Springer-Verlag 1987.

[Cons86] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall 1986.

[DAID87] *Final Version on TDL design*. ESPRIT PROJECT 892, DAIDA, Deliverable DES1.2, 1987.

[Devi90] Y. Deville. *Logic Programming - Systematic Programming Development*. Addison Wesley 1990.

[Dijk68] E.W. Dijkstra. *Goto Statement considered harmful.* CACM 11, 1968, S. 147-148, 538, 541.

[Dijk76] E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[ERMS90] J. Eder, A. Rudloff, F. Matthes, J.W. Schmidt. *Data Construction with Set Expressions in DBPL.* Professur DBIS, Fachbereich Informatik, Universität Hamburg, Dezember 1990.

[EiOh87] N. Eisinger, H.J. Ohlbach. *Grundlagen und Beispiele* (von Deduktionssystemen). Kapitel 2 von [BlBu87].

[Ende72] H.B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[Ende77] H.B. Enderton. *Elements of Set Theory.* Academic Press, 1977.

[Floy67] R.W. Floyd. *Assigning Meanings to Programs.* In: Proc. Symp. Appl. Math. (ed. J.T. Schwartz), vol. 19, pp. 19-31, Providence, Rhode Island: American Mathematical Society.

[Gard88] Trevor Vickers, Paul Gardiner J.R. Abrial, C.C. Morgan, J.M. Spivey. *The logic of 'B'.* Programming Research Group Oxford, Draft September 1988.

[Gard90] Paul Gardiner. *Persönliche Kommunikation.* Programming Research Group Oxford, Juli 1990.

[GeGe86] N. Gehani, A.D. McGettrick. *Software Specification Techniques.* Addison Wesley, 1986.

[GeNi87] M.R. Genesereth, N.J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann Publishers, 1987.

[Gent34] G. Gentzen. *Untersuchungen über das logische Schließen.* Mathematische Zeitschrift 39 (1934), S. 176-210, 405-431.

[Gogu81] J.A. Goguen. *More Thoughts on Specification and Verification.* ACM SIGSOFT, **6** (3), pp. 38-41. Auch in [GeGe86].

[Good84] D.I. Good. *Mechanical Proofs about Computer Programs.* In [HoSh86].

[Grie81] D. Gries. *The Science of Programming.* Springer-Verlag 1981.

[HaSH88] L. Hallnäs, P. Schroeder-Heister. *A Proof Theoretic Approach to Logic Programming.* Zentrum Philosophie und Wissenschaftstheorie, Universität Konstanz, 1988.

[Haye87] I. Hayes. *Specification Case Studies.* Prentice Hall, 1987.

[Hehn84a] E.C.R. Hehner. *The Logic of Programming.* Prentice Hall, 1984.

[Hehn84b] E.C.R. Hehner. *Predicative Programming Part 1, Part 2.* Communications of the ACM February 1984, Vol 27, Number 2, pp. 134-151.

[HeGM86] E.C.R. Hehner, L.E. Gupta, A.H. Malton. *Predicative Methodology.* Lecture Outline, Marktoberdorf Summer School, August 1986.

[Hoar84] C.A.R. Hoare. *Programs are Predicates*. In [HoSh86].

[HoSh86] C.A.R. Hoare, J.C. Shepherdson (Eds.). *Mathematical Logic and Programming Languages*. Prentice Hall, 1986.

[Hutt87] D. Hutter. *Vollständige Induktion*. Kapitel 5 von [BlBu87].

[IEEE83] IEEE Standard Glossary of software engineering terminology, IEEE standard 729, 1983.

[Jone86] C.B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 1986

[Kowa79] R. Kowalski. *Algorithm = Logic + Control*. CACM 22:7, S. 424-436 (1979).

[Kowa84] R. Kowalski. *The Relation between Logic Programming and Logic Specification*. In [HoSh86].

[LiBe79] B.H. Liskov, V. Berzins. *An Appraisal of Program Specifications*. In: Research Directions in Software Technology. P. Wegner (Ed.), pp. 276-301, MIT Press. Auch in [GeGe86].

[Lloy84] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag 1984. 2nd, Extended Edition 1987.

[LoSc87] P.C. Lockemann, J.W. Schmidt (Eds.). *Datenbank-Handbuch*. Springer-Verlag, 1987.

[Matt88] F. Matthes. *Typvollständigkeit in Datenbankprogrammiersprachen - DBPL-Sprachentwurf und Implementation*. Diplomarbeit, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, November 1988.

[MaDL87] H.C. Mayr, K.R. Dittrich, P.C. Lockemann. *Datenbankentwurf*. Kapitel 5 von [LoSc87].

[McNe89] Iain R. McNeil. *Validating and Verifying SML Specifications - Efficient Prototypes and Proofs*. Software Engineering Section, I.T.R.U., B.P. Research Centre, Chertsey Road, Sunbury on Thames, Middlesex, TW16 7LN, ESPRIT DAIDA Project 892 Deliverable PRO 4.3, Februar 1989.

[Miln84] R. Milner. *The Use of Machines to Assist in Rigorous Proof*. In [HoSh86].

[Nieb89] P. Niebergall. *Realisierung eines sprach- und datenbanksensitiven Programmierwerkzeugs*. Diplomarbeit. Fachbereich Informatik, Johann Wolfgang Goethe-Universität Frankfurt a. M. , August 1989.

[Nils90a] D. Nilsson. *devB User Manual*. Software Engineering Section, I.T.R.U., B.P. Research Centre, Chertsey Road, Sunbury on Thames, Middlesex, TW16 7LN, Draft Juni 1990.

[Nils90b] D. Nilsson. *B User Manual*. Software Engineering Section, I.T.R.U., B.P. Research Centre, Chertsey Road, Sunbury on Thames, Middlesex, TW16 7LN, Draft 15.6.90.

[Part90] H.A. Partsch. *Specification and Transformation of Programs. A Formal Approach to Software Development*. Springer-Verlag 1990.

[Reut87] A. Reuter. *Maßnahmen zur Wahrung von Sicherungs- und Integritätsbedingungen.* Kapitel 4 von [LoSc87].

[Rich78] M. Richter. *Logikkalküle.* Teubner-Verlag 1978.

[Schm84] D. Schmidt. *A Programming Notation for Tactical Reasoning.* In: Proceedings of Seventh International Conference on Automated Deduction. Lecture Notes in Computer Science 170, Springer-Verlag 1984, pp. 445-459.

[Schm87] J.W. Schmidt. *Datenbankmodelle.* Kapitel 1 von [LoSc87].

[SEMa88] J.W. Schmidt, H. Eckhardt, F. Matthes. *DBPL Report.* Universität Frankfurt, November 1988.

[SFSS90] D. Stemple, L. Fegaras, T. Sheard, A. Socorro. *Exceeding the Limits of Polymorphism in Database Programming Languages.* Lecture Notes in Computer Science 416, Springer-Verlag 1990, pp.269-285.

[ShSt89] T. Sheard, D. Stemple. *Automatic Verification of Database Transaction Safety.* ACM Transactions on Database Systems, Vol. 14, Nr. 3, September 1989, S.322-368.

[Spiv88] J.M. Spivey. *Understanding Z, A Specification Language and its Formal Semantics.* Cambridge University Press, 1988.

[Spiv89] J.M. Spivey. *The Z Notation. A Reference Manual.* Prentice Hall, 1989.

[StSh90] D. Stemple, T. Sheard. *Status and Plans of the ADABTPL Project.* Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003. Oktober1990.

[Stuc90] R. Stuckardt. *B-Tool-Benutzermanual.* Professur DBIS, Fachbereich Informatik, Universität Hamburg, November1990.

[Tars55] A. Taski. *A Lattice-theoretical Fixpoint Theorem and its Applications.* Pacific J. Math. **5** (1955), S.285-309.

[Ullm82] J.D. Ullman. *Principles of database systems.* 2nd Edition, Computer Science Press, 1982.

[ViGa88a] Trevor Vickers, Paul Gardiner. *A Tutorial on B: A Theorem Proving Assistant.* Programming Research Group Oxford, Draft Juli 1988.

[ViGa88b] Trevor Vickers, Paul Gardiner. *Variable Lifting.* Programming Research Group Oxford, Draft 16.November 1988.

[Wetz90] I. Wetzel. *Formale Unterstützung bei der Datenbankprogrammierung.* Diplomarbeit. Fachbereich Informatik, Johann Wolfgang Goethe-Universität Frankfurt a. M. , Januar 1990.

[Wirt71] N. Wirth. *Program Development by Stepwise Refinement.* CACM 14,1971, S. 221-227.

[Wirt88] N. Wirth. *Programming in Modula-2.* Fourth Edition, Springer-Verlag 1988.

[WoLo88] Jim Woodcock, Martin Loomes. *Software Engineering Mathematics.* Pitman 1988.

Anhang A

Die Datei SYMBOL zur Symbolvordefinition

Gezeigt wird ein Listing der Datei `SYMBOL`, mit Hilfe derer dem B-Tool *syntaktische Vereinbarungen* vorgegeben werden. Die abgebildete Datei enthält die Vorgaben, die für den Einsatz des B-Tools in der DAIDA-Umgebung notwendig sind.

```
END         clo      keyt
]           clo      nrml
}           clo      nrml

evl         bin      void     11
spe         bin      void     10  right
ghost       atm      void      0
error       atm      nrml      0
=>          bin      blba     -6
DED         atm      nrml      0
GEN         atm      nrml      0
HYP         atm      nrml      0
ARI         atm      nrml      0
bootX       atm      nrml      0
THEORY      opn      keyw      11  END
IS          bin      keyl     -9
TAC         bin      keyl     -9
bwrite      unr      nrml     11
bprint      unr      nrml     11
bread       unr      nrml     11
<=          bin      nrml      0
>=          bin      nrml      0
/=          bin      nrml      0
inhyp       unr      nrml     11
false       atm      nrml      0
```

```
<=>          bin      blba     -4
FTAC         bin      keyl     -9
bconnect     unr      nrml     11
vrb          unr      nrml     11
"/dev/tty"   atm      nrml      0
:=           bin      nrml     -2
==           bin      blba     -4
num          unr      nrml     11
bcall        unr      nrml     11
bmap         bin      blba      0
JOK1         unr      nrml     11
JOK2         bin      blba      0
bcat         unr      nrml     11
bcopy        unr      nrml     11
blank2       bin      blba      0
blank0       atm      nrml      0
bnew         unr      nrml     11
blmap        bin      blba      0
brul         unr      nrml     11
brev         unr      nrml     11
bflat        unr      nrml     11
bhalt        atm      nrml      0
REV          atm      nrml      0
FLAT         atm      nrml      0
bsmap        bin      blba      0
bslmap       bin      blba      0
MAP          atm      nrml      0
LMAP         atm      nrml      0
FEQL         atm      nrml      0
bget         unr      nrml     11

&            bin      bbat     -5
;            bin      term     -7
_            bin      nrml      11
.            bin      nrml      10  right
/            bin      nrml      1
*            bin      nrml      1
,            bin      nrml     -1
:            bin      blka     -4
|            bin      blba     -8
^            bin      nrml      0
-            bin      nrml      0
+            bin      nrml      0
=            bin      blba     -4
\            bin      nrml      0
<            bin      nrml      0
>            bin      nrml      0
```

```
[           opn      nrml     11   ]
{           opn      nrml     11   }

!           unr      nrml     11
@           unr      nrml     11
#           unr      nrml     11
%           unr      nrml     11

~           pst      nrml     11
'           pst      nrml     11
?           atm      nrml      0

ENDIF       clo      keyt
ENDPRE      clo      keyt
ENDWHILE    clo      keyt
ENDCASE     clo      keyt
ENDVAR      clo      keyt
ENDEITHER   clo      keyt
ENDFOR      clo      keyt

PROG        opn      keyl      11  ENDPROG
BEGIN       opn      keyl      11  END
IF          opn      keyw      11  ENDIF
PRE         opn      keyl      11  ENDPRE
WHILE       opn      keyw      11  ENDWHILE
CASE        opn      keyw      11  ENDCASE
VAR         opn      keyw      11  END
EITHER      opn      keyw      11  ENDEITHER
FOR         opn      keyw      11  ENDFOR
ANY         opn      keyw      11  END
LET         opn      keyw      11  END
WHERE       bin      keyl      -9
BE          bin      keyl      -9
DATA        bin      keyl      -9
SPGM        bin      keyl      -9
MAIN        bin      keyl      -9
IN          bin      keyl      -9
THEN        bin      keyl      -9
ELSE        bin      keyl      -9
WHEN        bin      keyw      -9
DO          bin      keyl      -9
OF          bin      keyl      -9
PROC        unr      blka      -6
FUNC        unr      blka      -6
TO          bin      keyw      -9
ELSIF       bin      keyw      -9
ORELSE      bin      keyw      -9
THENDO      bin      keyl      -9
```

```
OTHERS      bin      keyl      -9
||          bin      bbat      -7
TRUE        atm      nrml       0
FALSE       atm      nrml       0
and         bin      blba       1
or          bin      blba       0
dom         unr      nrml      11
cod         unr      nrml      11
ran         unr      nrml      11
crd         unr      nrml      11
not         unr      nrml      11
inc         bin      blba       0
incl        bin      blba       0
<>          atm      nrml       0
{}          atm      nrml       0
[]          bin      blba      -7
<--         bin      blba      -1
==>         bin      blba      -6
-->         bin      blba       0
+->         bin      blba       0
>->         bin      blba       0
>+>         bin      blba       0
-->>        bin      blba       0
+->>        bin      blba       0
>->>        bin      blba       0
<->         bin      blba       0
USE         bin      keyl      -9
SEE         bin      keyl      -9
VRB         bin      keyl      -9
INV         bin      keyl      -9
PMT         bin      keyl      -9
INI         bin      keyl      -9
OPN         bin      keyl      -9
ASN         bin      keyl      -9
IPT         bin      keyl      -9
CHG         bin      keyl      -9
SETS        bin      keyl      -9
ENUM        bin      keyl      -9
CST         bin      keyl      -9
RULES       bin      keyl      -9
PRP         bin      keyl      -9
AUTOP       bin      keyl      -9
LEMSEP      bin      keyl      -9
MATHTAC     bin      keyl      -9
TYPETAC     bin      keyl      -9
ASNTAC      bin      keyl      -9
REFTAC      bin      keyl      -9
INVTAC      bin      keyl      -9
```

```
THYS        bin      keyl      -9
PATH        bin      keyl      -9
FWDPRP      bin      keyl      -9
```

Anhang B

Verzeichnis der Theoriensammlung devB

Dieser Anhang gibt einen Überblick über die Theoriensammlung `devB`, die die Verarbeitung abstrakter Maschinen auf dem B-Tool realisiert. Die Zahl in Klammern hinter dem Theorienamen ist die Anzahl der Regeln der Theorie.

```
 1: bootX (9)
 2: MACHINES (0)
 3: BlmapMacrosX (65)
 4: StoreX (2)
 5: RemoveBtmpListX (13)
 6: GluePrefixX (0)
 7: GlueGeneralPrefixX (1)
 8: StoGluePrefixX (6)
 9: StoreConcAbsVarbListOverlapX (10)
10: StoreConcAbsVarbListResidueX (6)
11: StoreAbsOperationResidueX (6)
12: PromoteOperationX (3)
13: StorePromotedUsedOperationX (4)
14: PromoteUsedOperationX (15)
15: CheckIptOrUseMchStoredX (6)
16: FilterLoadMachineX (4)
17: CheckCtxStoredX (13)
18: ReorderAntecedent1X (11)
19: ReorderAntecedent2X (10)
20: UpdateEntryX (4)
21: dummypopstackX (24)
22: popstackX (7)
23: rebuiltX (17)
24: raiseX (13)
```

25: reduceX (87)
26: compileinitX (3)
27: StoreContextClausesX (12)
28: StoreInvLemmaX (18)
29: BuildAssertionLemmasX (9)
30: StoreRefLemmaX (19)
31: StoreOperationX (25)
32: CheckParamRenameX (6)
33: StoreParamRenameX (1)
34: BuildParamRenameX (16)
35: BuildOperationListsX (9)
36: StoreOperationListsX (5)
37: StorePredicateX (7)
38: StoreContextListX (5)
39: StoreInvariantX (31)
40: BuildVariableListsX (8)
41: StoreTheoriesX (15)
42: StoreVersionX (7)
43: StoreLevelX (3)
44: GenerateInvLemmasX (8)
45: genvarX (21)
46: CalculusX (51)
47: TransCondX (33)
48: BuildConjectureX (5)
49: PrepareConjectureX (17)
50: PrepareLemma1X (1)
51: PrepareLemma2X (6)
52: PrepareLemma3X (5)
53: Analyse2X (18)
54: Rebuilt2X (10)
55: Popstack2X (4)
56: UnRenameX (1)
57: PrintOpnListX (12)
58: PrintListX (4)
59: GenerateClauseX (12)
60: PrintSpecificationX (2)
61: dumplemmasX (9)
62: EnableCatAndGetX (2)
63: EnableRenameX (5)
64: BuildCompleteSpecRenameX (10)
65: BuildCompleteRefRenameX (10)
66: CheckForLibMchX (2)
67: CheckFilterX (4)
68: CheckEmptyX (1)
69: AccumInvX (9)
70: BuildImportedOrUsedRename1X (20)
71: BuildImportedOrUsedRename2X (28)
72: AccumImportedOrUsedRenameX (24)

```
 73: RebuildImportedOrUsedCtxX (9)
 74: RebuildImportedMchX (6)
 75: RebuildUsedMchX (6)
 76: RebuildImportedOrUsedMchX (11)
 77: BuildUseOpnRenameX (9)
 78: BuildVrbRenameX (29)
 79: StoreParamRenameItfX (4)
 80: StoreAbsItfOpX (17)
 81: BuildErrorX (30)
 82: StoreProofMethodX (34)
 83: BuildProofMethodX (12)
 84: BuildContextX (9)
 85: BuildInterfaceX (7)
 86: BuildRefinementMachineX (13)
 87: BuildSpecificationMachineX (15)
 88: ParseInputX (13)
 89: TempX (0)
 90: InvariantLemmaX (0)
 91: RefinementLemmaX (0)
 92: GetProofMethodX (15)
 93: GetLevelX (0)
 94: GetVersionX (0)
 95: GetEnumX (0)
 96: GetParamRenameX (26)
 97: GetRenameX (0)
 98: GetStoredX (0)
 99: GetContextListX (0)
100: GetContextClausesX (0)
101: GetVariableListX (0)
102: GetOperationListsX (0)
103: GetOperationX (0)
104: GetInitialisationX (0)
105: GetInvariantX (0)
106: GetInvariantUsedX (0)
107: GetPredicateX (0)
108: SemanticsX (0)
109: FwdSemanticsX (0)
```

Anhang C

Die abstrakten Maschinen des Verfeinerungsbeispiels

Es wird die Folge abstrakter Maschinen der *DAIDA-Beispielverfeinerung* gezeigt, die als Ausgangspunkt für die Lemmaklassifikation dient. Zu den Spezifikations- und Verfeinerungsmaschinen wird jeweils der zugehörige Kontext gezeigt. (In der zweiten Verfeinerung wird *kein* neuer Kontext vereinbart.)

C.1 Spezifikation

C.1.1 Kontextvereinbarung

```
CTX

   |* Kontext der Spezifikation: Definition der Basismengen.\\ *| ;

   ExtResearchCompaniesCtx

SETS

   Strings,
   COMPANIES,
   PROJECTS,
   EMPLOYEES,
   CompNames,
   MONEY,
   EmpNames,
   ProjNames
```

```
ENUM

   Agencies = { ESPRIT,DFG,NSF }

PRP

   CompNames = Strings ;
   EmpNames = Strings ;
   ProjNames = Strings

END
```

C.1.2 Spezifikationsmaschine

```
MCH

   |* Spezifikationsmaschine des erweiterten
      ResearchCompanies-Beispiels.\\
      Roland Stuckardt, 17.11.90\\                                    *| ;

   ExtResearchCompaniesMch

SEE

   ExtResearchCompaniesCtx

VRB

   companies, compName, engagedIn, budget,
   projects, projName, getsGrantFrom, consortium,
   employees, empName, belongsTo, worksOn

INV

  |* Klassenspezifische Integrit\"atsbedingungen und Invarianten:\\ *|;

   companies: POW(COMPANIES);
   compName: companies --> CompNames;
   engagedIn: companies --> POW(projects);
   budget: companies --> MONEY ;

   projects: POW(PROJECTS);
   projName: projects --> ProjNames;
```

```
   getsGrantFrom: projects --> Agencies;
   consortium: projects --> POW(companies) ;

   employees: POW(EMPLOYEES);
   empName: employees --> EmpNames;
   belongsTo: employees --> companies;
   worksOn: employees --> POW(projects);

   |* funktionale Abh\"angigkeiten (Eindeutigkeitsbeziehungen):\\ *| ;

   !x.!y.(x,y: projects =>
     (projName(x) = projName(y)  &
     getsGrantFrom(x) = getsGrantFrom(y)
      => x = y )) ;

   !x.!y.(x,y: companies =>
     (compName(x) = compName(y)
      => x = y )) ;


   |* Klassen\"ubergreifende Integrit\"atsbedingungen: \\ *|;

   !x.(x: employees =>
          worksOn(x) incl engagedIn(belongsTo(x)));

   !x.(x: projects =>
          consortium(x) = {y | y:  companies & x : engagedIn(y)})


INI

   companies := {} ||
   compName := {} ||
   engagedIn := {} ||
   budget := {} ||
   projects := {} ||
   projName := {} ||
   getsGrantFrom := {} ||
   consortium := {} ||
   employees := {} ||
   empName := {} ||
   belongsTo := {} ||
   worksOn := {}


OPN

      |* L\"osch-Operationen:
```

```
          \biz
           {
             \bitem Entfernen eines Projekts
             \bitem Entfernen eines Angestellten
           }
           aus der Datenbank.\\ *|
;

  removeProject(proj) =
   PRE
     proj : projects &
     consortium(proj) = {}   /* Erhalt der referentiellen Integritaet */
   THEN
     projects:= projects - {proj} ||
     projName:= {proj} <<|  projName ||
     getsGrantFrom:=  {proj} <<| getsGrantFrom ||
     consortium:= {proj} <<| consortium
   END

 ;


 removeEmployee(empl) =
   PRE
     empl : employees
   THEN
     employees:= employees - {empl} ||
     empName:= {empl} <<| empName ||
     belongsTo:= {empl} <<| belongsTo ||
     worksOn:= {empl} <<| worksOn
   END

 ;

      |* Einf\"uge-Operationen:
         \biz
          {
            \bitem Einf\"ugen eines Forschungsinstituts
            \bitem Einf\"ugen eines Angestellten
          }
          in die Datenbank.\\ *|
;

  c <-- newCompany(newname,ventures,budg) =
   PRE
     newname : CompNames - ran(compName) &   /*   Keine Verletzung der   */
     ventures : POW(projects)  &             /* Eindeutigkeitsbeziehung  */
     budg : MONEY
```

```
    THEN
     ANY co WHERE
       co : COMPANIES - companies
     THEN
       compName(co):= newname ||
       engagedIn(co):= ventures ||
       budget(co):= budg ||
       companies:= companies \/ {co} ||
       consortium:= consortium<+(%x.(x : ventures | consortium(x) \/ {co})) ||
       c:= co
    END
   END

;

   e <-- newEmployee(newname,belongs,works)  =
     PRE
       newname: EmpNames &
       belongs: companies &
       works: POW(engagedIn(belongs))
     THEN
       ANY ee WHERE
         ee: EMPLOYEES-employees
       THEN
         empName(ee):= newname ||
         worksOn(ee):= works ||
         belongsTo(ee):= belongs ||
         employees:= employees \/ {ee} ||
         e:=ee
       END
     END
;

      |* \"Anderungs-Operationen:
         \biz
          {
            \bitem \"Andern aller Attribute
            \bitem \"Andern der bearbeiteten Projekte
          }
          eines Angestellten.\\ *|
;

 updateEmployer(empl,belongs,works) =
   PRE
     empl : employees &
     belongs : companies &
     works: POW(engagedIn(belongs))
   THEN
```

```
      belongsTo(empl):= belongs ||
      worksOn(empl):= works
    END

  ;

  updateJobs(empl,works) =
    PRE
      empl : employees &
      works : POW(engagedIn(belongsTo(empl)))
    THEN
      worksOn(empl):= works
    END


 END  /* MCH */
```

C.2 Erste Verfeinerung

C.2.1 Kontextvereinbarung

```
CTX

   |* Kontext der 1. und 2. Verfeinerung: zus\"atzliche
      Datenidentifikationsbeziehungen.\\ *| ;

   ExtResearchCompaniesCtx2

SETS

   Strings,
   CompNames,
   EmpNames,
   ProjNames,
   EMPLOYEES,
   PROJECTS,
   COMPANIES,
   MONEY,
   EmpIds

ENUM
```

```
   Agencies = { ESPRIT, DFG, NSF }


PRP

   CompNames = Strings;
   EmpNames = Strings;
   ProjNames = Strings;
   COMPANIES = CompNames;
   EMPLOYEES = EmpIds;
   PROJECTS = ProjNames * Agencies

END
```

C.2.2 Erste Verfeinerungsmaschine

```
 REF

    |* 1. Verfeinerungsmaschine des erweiterten
       ResearchCcompanies-Beispiels.\\
       Roland Stuckardt, 17.11.90\\                              *| ;

    ExtResearchCompaniesMch

 SEE

    ExtResearchCompaniesCtx2

 VRB

    companies, compName, engagedIn, budget,
    projects, projName, getsGrantFrom, consortium,
    employees, empName, belongsTo, worksOn,
    empId, projId


 INI

  companies, compName, engagedIn, budget := {},{},{},{} ||
  employees, empName, belongsTo, worksOn := {},{},{},{} ||
  projects, projName, getsGrantFrom := {},{},{} ||
  consortium, empId, projId := {},{},{}
```

```
OPN

  |* 1. Verfeinerungen der L\"osch-Operationen: \\ *| ;


  removeProject(proj) =
   PRE
     proj : projects &
     consortium(proj) = {}
   THEN
     projects:= projects - {proj}  ||
     consortium:= {proj} <<| consortium
   END

 ;


 removeEmployee(empl) =
   PRE
     empl : employees
   THEN
     employees:= employees - {empl} ||
     empName:= {empl} <<| empName ||
     belongsTo:= {empl} <<| belongsTo ||
     worksOn:= {empl} <<| worksOn
   END

 ;


  |* 1. Verfeinerungen der Einf\"uge-Operationen: \\ *| ;


  c <-- newCompany(newname,ventures,budg) =
   PRE
     newname : CompNames - ran(compName) &
     ventures : POW(projects) &
     budg : MONEY
   THEN
     engagedIn(newname):= ventures ||
     budget(newname):= budg ||
     companies:= companies \/ {newname} ||
     consortium:= consortium<+(%x.(x : ventures | consortium(x) \/ {newname})) ||
     c:= newname
   END
```

```
;

  e <-- newEmployee(newname,belongs,works)  =
  PRE
    newname : EmpNames &
    belongs : companies &
    works : POW(engagedIn(belongs))
  THEN
    empName(newEmpId) := newname ||
    worksOn(newEmpId) := works ||
    belongsTo(newEmpId) := belongs ||
    employees := employees \/ {newEmpId} ||
    e := newEmpId
  END
;


 |* 1. Verfeinerungen der \"Anderungs-Operationen: \\ *| ;


updateEmployer(empl,belongs,works) =
  PRE
    empl : employees &
    belongs : companies &
    works: POW(engagedIn(belongs))
  THEN
    belongsTo(empl):= belongs ||
    worksOn(empl):= works
  END

;

updateJobs(empl,works) =
  PRE
    empl : employees &
    works : POW(engagedIn(belongsTo(empl)))
  THEN
    worksOn(empl):= works
  END


ASN

 |* Definitionen zur Datenidentifikation: \\ *| ;

compName := %x.(x : companies | x);
empId := %x.(x : employees | x);     /* "kuenstliche" Identifikation */
projName := %x.(x : projects & x = (a,b) | a);
```

```
 getsGrantFrom := %x.(x : projects & x = (a,b) | b);
 projId :=  %x.(x : projects | x)


END  /* MCH */
```

C.3 Zweite Verfeinerung

```
REF

   |* 2. Verfeinerungsmaschine der erweiterten
      ResearchCompanies-Beispiels.\\
      Roland Stuckardt, 17.11.90\\                    *| ;

   ExtResearchCompaniesMch

SEE

   ExtResearchCompaniesCtx2

VRB

   companies, compName, engagedIn, budget,
   projects, projName, getsGrantFrom, consortium,
   employees, empName, belongsTo, worksOn,
   empId, projId,
   compClass, empClass, projClass, tEmpId

CHG

   |* Datenverfeinerung: Einf\"uhrung relational strukturierter
      Variablen: \\ *| ;

   compClass : companies --> POW (projects) * MONEY;
   empClass : employees  -->
       EmpNames * companies * POW (projects);
   projClass : projects --> POW (companies);

   tEmpId: EmpIds
```

```
INI

   companies, compName, engagedIn, budget := {},{},{},{} ||
   employees, empName, belongsTo := {},{},{} ||
   worksOn, projects, projName := {},{},{} ||
   getsGrantFrom, consortium, empId := {},{},{} ||
   projId, compClass, empClass := {},{},{} ||
   projClass, tEmpId := {},{}



OPN


  |* 2. Verfeinerungen der L\"osch-Operationen: \\ *| ;


  removeProject(proj) =
   PRE
     proj : projects &
     consortium(proj) = {}
   THEN
     projClass:= {proj} <<| projClass
   END

 ;


 removeEmployee(empl) =
   PRE
     empl : employees
   THEN
     empClass:= {empl} <<| empClass
   END

 ;


  |* 2. Verfeinerungen der Einf\"uge-Operationen: \\ *| ;


  c <-- newCompany(newname,ventures,budg) =
   PRE
     newname : CompNames - ran(compName) &
     ventures : POW(projects)  &
     budg : MONEY
   THEN
     projClass:= projClass<+(%x.(x : ventures | projClass(x) \/ {newname}));
```

```
    compClass:= compClass \/ {(newname,ventures,budg)} ;
    c:= newname
  END

;

  e <-- newEmployee(newname,belongs,works)  =
PRE
  newname   : EmpNames &
  belongs   : companies &
  works   :   POW (engagedIn(belongs))
THEN
  tEmpId:= newEmpId;
  empClass   := empClass \/ {(tEmpId, newname, belongs, works)};
  e := tEmpId
END
;


 |* 2. Verfeinerungen der \"Anderungs-Operationen: \\ *| ;


updateEmployer(empl,belongs,works) =
  PRE
    empl : employees &
    belongs : companies &
    works: POW(engagedIn(belongs))
  THEN
    empClass(empl):= (empName(empl),belongs,works)
  END

;

updateJobs(empl,works) =
  PRE
    empl : employees &
    works : POW(engagedIn(belongsTo(empl)))
  THEN
    empClass(empl):= (empName(empl),belongsTo(empl),works)
  END

ASN

 |* Definitionen zur Datenstrukturierung: \\ *| ;

  compName := %x.(x : companies | x);
  empId := %x.(x : employees | x);
  projName := %x.(x : projects & x = (a,b) | a);
```

```
    getsGrantFrom := %x.(x : projects & x = (a,b) | b);
    projId :=  %x.(x : projects | x);
    engagedIn := %x.(x : companies & compClass(x) = (a,b) | a);
    budget := %x.(x : companies & compClass(x) = (a,b) | b);
    empName := %x. (x : employees & empClass(x) = (a,b,c) | a);
    belongsTo := %x. (x : employees & empClass(x) = (a,b,c) | b);
    worksOn := %x. (x : employees & empClass(x) = (a,b,c) | c);
    consortium :=   %x. (x : projects | projClass(x));
    companies := dom(compClass);
    employees := dom(empClass);
    projects := dom(projClass)


 END  /* MCH */
```

C.4 Dritte Verfeinerung

C.4.1 Kontextvereinbarung

```
CTX

  |* Kontext der 3. Verfeinerung:
     \biz
      {
       \bitem  Einf\"uhrung relational strukturierter Basismengen
       \bitem  Erm\"oglichung der statischen Typisierung
      }
     \\  *| ;

  ExtResearchCompaniesCtx4

SETS
  Strings,
  CompNames,
  EmpNames ,
  ProjNames ,
  EMPLOYEES,
  PROJECTS,
  COMPANIES,
  MONEY,
  EmpIds,
  ProjIdRecType,
  ProjName,
```

```
  GetsGrantFrom,
  ProjIdRelType,
  CompRecType,
  CompName,
  EngagedIn,
  Budget,
  CompRelType,
  EmpRecType,
  EmpId,
  EmpName,
  BelongsTo,
  WorksOn,
  EmpRelType,
  ProjRecType,
  ProjId,
  Consortium,
  ProjRelType

ENUM

  Agencies = { ESPRIT, DFG, NSF }

PRP

  CompNames = Strings;
  EmpNames = Strings;
  ProjNames = Strings;
  COMPANIES = CompNames;
  EMPLOYEES = EmpIds;
  PROJECTS = ProjIdRecType;

  ProjIdRecType = ProjNames * Agencies;
  ProjName = %x.(x:ProjIdRecType & x = (a,b) | a);
  GetsGrantFrom = %x.(x:ProjIdRecType & x = (a,b) | b);
  ProjIdRelType = POW (ProjIdRecType);

  CompRecType = CompNames * ProjIdRelType * MONEY;
  CompName = %x.(x:CompRecType & x = (a,b,c) | a);
  EngagedIn = %x.(x:CompRecType & x = (a,b,c) | b);
  Budget = %x.(x:CompRecType & x = (a,b,c) | c);
  CompRelType = CompNames +-> ProjIdRelType * MONEY;

  EmpRecType = EmpIds * EmpNames * CompNames * ProjIdRelType;
  EmpId = %x.(x:EmpRecType & x = (a,b,c,d) | a);
  EmpName = %x.(x:EmpRecType & x = (a,b,c,d) | b);
  BelongsTo = %x.(x:EmpRecType & x = (a,b,c,d) | c);
  WorksOn = %x.(x:EmpRecType & x = (a,b,c,d) | d);
  EmpRelType = EmpIds +-> EmpNames * CompNames * ProjIdRelType;
```

```
  ProjRecType = ProjIdRecType * POW (CompNames);
  ProjId = %x.(x:ProjRecType & x =(a,b) | a);
  Consortium = %x.(x:ProjRecType & x =(a,b) | b);
  ProjRelType = ProjIdRecType +-> POW (CompNames)


END
```

C.4.2 Dritte Verfeinerungsmaschine

```
REF

   |* 3. Verfeinerungsmaschine der erweiterten
      ResearchCcompanies-Beispiels.\\
      Roland Stuckardt, 17.11.90 \\                          *| ;

   ExtResearchCompaniesMch

SEE

   ExtResearchCompaniesCtx4

VRB

   companies, compName, engagedIn, budget,
   projects, projName, getsGrantFrom, consortium,
   employees, empName, belongsTo, worksOn,
   empId, projId, tEmpId,
   compRel, empRel, projRel,
   compRelDes, empRelDes, projRelDes


CHG

   |* Datenverfeinerung: Einf\"uhrung statisch typisierter
      Datenstrukturen:\\ *| ;

   compRel : CompRelType;
   compRel = compClass;

   empRel : EmpRelType;
   empRel = empClass;
```

```
   projRel : ProjRelType;
   projRel = projClass;

   tEmpId : EmpIds


  INI

   companies, compName, engagedIn, budget := {},{},{},{} ||
   employees, empName, belongsTo := {},{},{} ||
   worksOn, projects, projName := {},{},{} ||
   getsGrantFrom, consortium, empId := {},{},{} ||
   projId, tEmpId, compRel := {},{},{} ||
   empRel, projRel, compRelDes := {},{},{} ||
   empRelDes, projRelDes := {},{}


OPN

  |* 3. Verfeinerungen der L\"osch-Operationen: \\ *| ;


  removeProject(proj) =
   PRE
     proj : ProjIdRecType
   THEN
     IF
       proj : projects &
       consortium(projRelDes(proj)) = {}
     THEN
       projRel:= {proj} <<| projRel
     END
   END

 ;


 removeEmployee(empl) =
   PRE
     empl : EmpIds
   THEN
     IF
      empl : employees
     THEN
       empRel:= {empl} <<| empRel
     END
   END
```

```
  ;


   |* 3. Verfeinerungen der Einf\"uge-Operationen: \\ *| ;

   c <-- newCompany(newname,ventures,budg) =
    PRE
      newname : CompNames &
      ventures : ProjIdRelType &
      budg : MONEY

    THEN
      IF
        newname : CompNames - ran(compName) &
        ventures : POW(projects)
      THEN
        projRel:= projRel<+(%x.(x : ventures | projRel(x) \/ {newname}));
        compRel:= compRel \/ {(newname,ventures,budg)} ;
        c:= newname
      ELSE
        c:= nilCompName
      END
    END

  ;

    e <-- newEmployee(newname,belongs,works)  =
  PRE
    newname   : EmpNames &
    belongs   : CompNames &
    works   :   ProjIdRelType
  THEN
    IF
      belongs : companies &
      works : POW (engagedIn(compRelDes(belongs)))
    THEN
      tEmpId:= newEmpId;
      empRel := empRel \/ {(tEmpId, newname, belongs, works)};
      e := tEmpId
    ELSE e := nilEmpId
    END
  END
;

   |* 3. Verfeinerungen der \"Anderungs-Operationen: \\ *| ;


  updateEmployer(empl,belongs,works) =
```

```
    PRE
      empl : EmpIds &
      belongs : CompNames &
      works : ProjIdRelType
    THEN
      IF
        empl : employees &
        belongs : companies &
        works: POW(engagedIn(compRelDes(belongs)))
      THEN
        empRel(empl):= (empName(empRelDes(empl)),belongs,works)
      END
    END

  ;

  updateJobs(empl,works) =
    PRE
      empl : EmpIds &
      works : ProjIdRelType
    THEN
      IF
        empl : employees &
        works : POW(engagedIn(compRelDes(belongsTo(empRelDes(empl)))))
      THEN
        empRel(empl):=(empName(empRelDes(empl)),belongsTo(empRelDes(empl)),works)
      END
    END


ASN

   |* Definitionen zur statischen Datentypisierung: \\ *| ;

  compRelDes := %x.(x : dom(compRel) & compRel(x) = (a,b) |(x,a,b));
  empRelDes := %x.(x : dom(empRel) & empRel(x) = (a,b,c) |(x,a,b,c));
  projRelDes := %x.(x : dom(projRel) & projRel(x) = a |(x,a));

  compName := compRel <| CompName;
  engagedIn := compRel <| EngagedIn;
  budget := compRel <| Budget ;

  empId := empRel <| EmpId;
  empName := empRel <| EmpName;
  belongsTo := empRel <| BelongsTo;
  worksOn := empRel <| WorksOn;

  projId := projRel <| ProjId;
```

```
  projName := dom(projRel) <| ProjName;
  getsGrantFrom := dom(projRel) <| GetsGrantFrom;
  consortium := projRel <| Consortium;

  companies := dom(compRel);
  employees := dom(empRel);
  projects := dom(projRel)



END  /* MCH */
```

Anhang D

Die aus der Beispielverfeinerung extrahierte Information

Dieser Anhang gibt eine Auflistung der "Theorien", in denen die während der Verarbeitung abstrakter Maschinen durch `devB` extrahierte Information abgelegt wird. Hiermit ist es möglich, die durchgeführten Beweise (vgl. Anhang G) nachzuvollziehen, da die dort verwendeten Taktiken diese Theorien referenzieren.

```
Extrahierte Information des erweiterten Verfeinerungsbeispiels
"ExtResearchCompanies ":

  92: GetProofMethodX (47)
  93: GetLevelX (4)
  94: GetVersionX (1)
  95: GetEnumX (3)
  96: GetParamRenameX (32)
  97: GetRenameX (8)
  98: GetStoredX (4)
  99: GetContextListX (8)
  100: GetContextClausesX (6)
  101: GetVariableListX (34)
  102: GetOperationListsX (8)
  103: GetOperationX (24)
  104: GetInitialisationX (8)
  105: GetInvariantX (8)
  106: GetInvariantUsedX (0)
  107: GetPredicateX (35)
  108: GetDefinitionX (36)
  109: SemanticsX (0)
  110: FwdSemanticsX (124)
```

```
  117: GetInvDefinitionX (36)


Theory 92 "GetProofMethodX":
----------------------------

*  1  filterflag == off
*  2  autopflag == off
*  3  lemsepflag == off
*  4  reflemmaftac == (GetPredicateX~;
      FwdSemanticsX~;FEQL)
*  5  reflemmatac == (RefinementLemmaX;DED;(
      PrepareConjectureX;REV;(PrepareConjectureX;
      ARI)~;FLAT)~;GetOperationX~;
      GetInitialisationX~;TransCondX~;
      GetInvariantX;GetDefinitionX~;
      BuildConjectureX;FLAT~;(((GEN;DED)~;
      CalculusX)~;PrepareLemma1X)~)
*  6  invlemmaftac == (GetPredicateX~;
      FwdSemanticsX~;FEQL)
*  7  invlemmatac == (InvariantLemmaX;DED;
      GetOperationX~;FLAT;GetInitialisationX;
      GetInvariantX~;GetDefinitionX~;
      BuildConjectureX;(((GEN;DED)~;CalculusX)~;
      PrepareLemma1X)~)
*  8  asnlemmaftac == (GetPredicateX~;
      FwdSemanticsX~;FEQL)
*  9  asnlemmatac == (DED;GEN;HYP)~
*  10 mathlemmatac == (DED;GEN;HYP)~
*  11 mathlemmaftac == (GetPredicateX~;
      FwdSemanticsX~;FEQL)
*  12 typelemmatac == (DED;GEN;HYP)~
*  13 typelemmaftac == (GetPredicateX~;
      FwdSemanticsX~;FEQL)
*  14 mathlemmataccall(g) == ((DED;GEN;HYP)~,(
      GetPredicateX~;FwdSemanticsX~;FEQL): g)
*  15 typelemmataccall(g) == ((DED;GEN;HYP)~,(
      GetPredicateX~;FwdSemanticsX~;FEQL): g)
*  16 autopflag == on
*  17 lemsepflag == off
*  18 asnlemmatac == (DED;AsnDefLemmasX)
*  19 reflemmatac == (DED;RefinementLemmaTacticsX
      )
*  20 currmch == ExtResearchCompaniesMch
*  21 currmch == ExtResearchCompaniesMch
*  22 currmch == ExtResearchCompaniesMch
*  23 currmch == ExtResearchCompaniesMch
*  24 currmch == ExtResearchCompaniesMch
```

```
*  25 currmch == ExtResearchCompaniesMch
*  26 currmch == ExtResearchCompaniesMch
*  27 currmch == ExtResearchCompaniesMch
*  28 currmch == ExtResearchCompaniesMch
*  29 currmch == ExtResearchCompaniesMch
*  30 currmch == ExtResearchCompaniesMch
*  31 currmch == ExtResearchCompaniesMch
*  32 currmch == ExtResearchCompaniesMch
*  33 currmch == ExtResearchCompaniesMch
*  34 currmch == ExtResearchCompaniesMch
*  35 currmch == ExtResearchCompaniesMch
*  36 currmch == ExtResearchCompaniesMch
*  37 currmch == ExtResearchCompaniesMch
*  38 currmch == ExtResearchCompaniesMch
*  39 currmch == ExtResearchCompaniesMch
*  40 currmch == ExtResearchCompaniesMch
*  41 currmch == ExtResearchCompaniesMch
*  42 currmch == ExtResearchCompaniesMch
*  43 currmch == ExtResearchCompaniesMch
*  44 currmch == ExtResearchCompaniesMch
*  45 currmch == ExtResearchCompaniesMch
*  46 currmch == ExtResearchCompaniesMch
*  47 currmch == ExtResearchCompaniesMch


Theory 93 "GetLevelX":
----------------------

*  1  level(ExtResearchCompaniesMch) == 1
*  2  level(ExtResearchCompaniesMch) == 2
*  3  level(ExtResearchCompaniesMch) == 3
*  4  level(ExtResearchCompaniesMch) == 4


Theory 94 "GetVersionX":
------------------------

*  1  version(ExtResearchCompaniesMch_1) == 1


Theory 95 "GetEnumX":
---------------------

*  1  inhyp(ctx(ExtResearchCompaniesCtx)) =>
      Agencies == {ESPRIT,DFG,NSF}
*  2  inhyp(ctx(ExtResearchCompaniesCtx2)) =>
      Agencies == {ESPRIT,DFG,NSF}
*  3  inhyp(ctx(ExtResearchCompaniesCtx4)) =>
```

```
      Agencies == {ESPRIT,DFG,NSF}


Theory 96 "GetParamRenameX":
----------------------------

*  1  wQxUzJKparamw,A == wQxUzJKparamw,a
*  2  wQxUzJKparamw,A,B == wQxUzJKparamw,a,b
*  3  wQxUzJKparamw,A,B,C == wQxUzJKparamw,a,b,c
*  4  wQxUzJKparamw,A,B,C,D == wQxUzJKparamw,a,b,
      c,d
*  5  wQxUzJKparamw,A,B,C,D,E == wQxUzJKparamw,a,
      b,c,d,e
*  6  wQxUzJKparamw,A,B,C,D,E,F == wQxUzJKparamw,
      a,b,c,d,e,f
*  7  wQxUzJKparamw,A,B,C,D,E,F,G ==
      wQxUzJKparamw,a,b,c,d,e,f,g
*  8  wQxUzJKparamw,A,B,C,D,E,F,G,H ==
      wQxUzJKparamw,a,b,c,d,e,f,g,h
*  9  wQxUzJKparamw,A,B,C,D,E,F,G,H,I ==
      wQxUzJKparamw,a,b,c,d,e,f,g,h,i
*  10 wQxUzJKparamw,A,B,C,D,E,F,G,H,I,J ==
      wQxUzJKparamw,a,b,c,d,e,f,g,h,i,j
*  11 wQxUzJKparamw,A,B,C,D,E,F,G,H,I,J,K ==
      wQxUzJKparamw,a,b,c,d,e,f,g,h,i,j,k
*  12 wQxUzJKparamw,A,B,C,D,E,F,G,H,I,J,K,L ==
      wQxUzJKparamw,a,b,c,d,e,f,g,h,i,j,k,l
*  13 wQxUzJKparamw,A,B,C,D,E,F,G,H,I,J,K,L,M ==
      wQxUzJKparamw,a,b,c,d,e,f,g,h,i,j,k,l.m
*  14 wQxUzJKparamw,A,B,C,D,E,F,G,H,I,J,K,L,M,N
      == wQxUzJKparamw,a,b,c,d,e,f,g,h,i,j,k,l.m,
      n
*  15 wQxUzJKparamw,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O
      == wQxUzJKparamw,a,b,c,d,e,f,g,h,i,j,k,l.m,
      n,o
*  16 wQxUzJKparamw,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O
      ,P == wQxUzJKparamw,a,b,c,d,e,f,g,h,i,j,k,l
      .m,n,o,p
*  17 wQxUzJKparamw,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O
      ,P,Q == wQxUzJKparamw,a,b,c,d,e,f,g,h,i,j,k
      ,l.m,n,o,p,q
*  18 wQxUzJKparamw,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O
      ,P,Q,R == wQxUzJKparamw,a,b,c,d,e,f,g,h,i,j
      ,k,l.m,n,o,p,q,r
*  19 wQxUzJKparamw,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O
      ,P,Q,R,S == wQxUzJKparamw,a,b,c,d,e,f,g,h,i
      ,j,k,l.m,n,o,p,q,r,s
*  20 wQxUzJKparamw,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O
```

```
      ,P,Q,R,S,T == wQxUzJKparamw,a,b,c,d,e,f,g,h
      ,i,j,k,l.m,n,o,p,q,r,s,t
*  21 wQxUzJKparamw,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O
      ,P,Q,R,S,T,U == wQxUzJKparamw,a,b,c,d,e,f,g
      ,h,i,j,k,l.m,n,o,p,q,r,s,t,u
*  22 wQxUzJKparamw,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O
      ,P,Q,R,S,T,U,V == wQxUzJKparamw,a,b,c,d,e,f
      ,g,h,i,j,k,l.m,n,o,p,q,r,s,t,u,v
*  23 wQxUzJKparamw,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O
      ,P,Q,R,S,T,U,V,W == wQxUzJKparamw,a,b,c,d,e
      ,f,g,h,i,j,k,l.m,n,o,p,q,r,s,t,u,v,w
*  24 wQxUzJKparamw,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O
      ,P,Q,R,S,T,U,V,W,X == wQxUzJKparamw,a,b,c,d
      ,e,f,g,h,i,j,k,l.m,n,o,p,q,r,s,t,u,v,w,x
*  25 wQxUzJKparamw,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O
      ,P,Q,R,S,T,U,V,W,X,Y == wQxUzJKparamw,a,b,c
      ,d,e,f,g,h,i,j,k,l.m,n,o,p,q,r,s,t,u,v,w,x,
      y
*  26 wQxUzJKparamw,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O
      ,P,Q,R,S,T,U,V,W,X,Y,Z == wQxUzJKparamw,a,b
      ,c,d,e,f,g,h,i,j,k,l.m,n,o,p,q,r,s,t,u,v,w,
      x,y,z
*  27 paramrename(
      ExtResearchCompaniesMchremoveProject) ==
      wQxUzJKparamw,proj:=wQxUzJKparamw,a
*  28 paramrename(
      ExtResearchCompaniesMchremoveEmployee) ==
      wQxUzJKparamw,empl:=wQxUzJKparamw,a
*  29 paramrename(
      ExtResearchCompaniesMchnewCompany) ==
      wQxUzJKparamw,c,budg,ventures,newname:=
      wQxUzJKparamw,a,b,c,d
*  30 paramrename(
      ExtResearchCompaniesMchnewEmployee) ==
      wQxUzJKparamw,e,works,belongs,newname:=
      wQxUzJKparamw,a,b,c,d
*  31 paramrename(
      ExtResearchCompaniesMchupdateEmployer) ==
      wQxUzJKparamw,works,belongs,empl:=
      wQxUzJKparamw,a,b,c
*  32 paramrename(
      ExtResearchCompaniesMchupdateJobs) ==
      wQxUzJKparamw,works,empl:=wQxUzJKparamw,a,b


Theory 97 "GetRenameX":
-----------------------
```

```
*  1  vrbrename(ExtResearchCompaniesMch_1) == (
      companies,compName,engagedIn,budget,
      projects,projName,getsGrantFrom,consortium,
      employees,empName,belongsTo,worksOn =
      companies_1,compName_1,engagedIn_1,budget_1
      ,projects_1,projName_1,getsGrantFrom_1,
      consortium_1,employees_1,empName_1,
      belongsTo_1,worksOn_1)
*  2  completespecrename(ExtResearchCompaniesMch_
      1) == (companies,compName,engagedIn,budget,
      projects,projName,getsGrantFrom,consortium,
      employees,empName,belongsTo,worksOn =
      companies_1,compName_1,engagedIn_1,budget_1
      ,projects_1,projName_1,getsGrantFrom_1,
      consortium_1,employees_1,empName_1,
      belongsTo_1,worksOn_1)
*  3  vrbrename(ExtResearchCompaniesMch_2) == (
      empId,projId,companies_0,compName_0,
      engagedIn_0,budget_0,projects_0,projName_0,
      getsGrantFrom_0,consortium_0,employees_0,
      empName_0,belongsTo_0,worksOn_0,companies,
      compName,engagedIn,budget,projects,projName
      ,getsGrantFrom,consortium,employees,empName
      ,belongsTo,worksOn = empId_2,projId_2,
      companies_1,compName_1,engagedIn_1,budget_1
      ,projects_1,projName_1,getsGrantFrom_1,
      consortium_1,employees_1,empName_1,
      belongsTo_1,worksOn_1,companies_2,compName_
      2,engagedIn_2,budget_2,projects_2,projName_
      2,getsGrantFrom_2,consortium_2,employees_2,
      empName_2,belongsTo_2,worksOn_2)
*  4  completerefrename(ExtResearchCompaniesMch_2
      ) == (empId,projId,companies_0,compName_0,
      engagedIn_0,budget_0,projects_0,projName_0,
      getsGrantFrom_0,consortium_0,employees_0,
      empName_0,belongsTo_0,worksOn_0,companies,
      compName,engagedIn,budget,projects,projName
      ,getsGrantFrom,consortium,employees,empName
      ,belongsTo,worksOn = empId_2,projId_2,
      companies_1,compName_1,engagedIn_1,budget_1
      ,projects_1,projName_1,getsGrantFrom_1,
      consortium_1,employees_1,empName_1,
      belongsTo_1,worksOn_1,companies_2,compName_
      2,engagedIn_2,budget_2,projects_2,projName_
      2,getsGrantFrom_2,consortium_2,employees_2,
      empName_2,belongsTo_2,worksOn_2)
*  5  vrbrename(ExtResearchCompaniesMch_3) == (
      compClass,empClass,projClass,tEmpId,
```

```
      companies_0,compName_0,engagedIn_0,budget_0
      ,projects_0,projName_0,getsGrantFrom_0,
      consortium_0,employees_0,empName_0,
      belongsTo_0,worksOn_0,empId_0,projId_0,
      companies,compName,engagedIn,budget,
      projects,projName,getsGrantFrom,consortium,
      employees,empName,belongsTo,worksOn,empId,
      projId = compClass_3,empClass_3,projClass_3
      ,tEmpId_3,companies_2,compName_2,engagedIn_
      2,budget_2,projects_2,projName_2,
      getsGrantFrom_2,consortium_2,employees_2,
      empName_2,belongsTo_2,worksOn_2,empId_2,
      projId_2,companies_3,compName_3,engagedIn_3
      ,budget_3,projects_3,projName_3,
      getsGrantFrom_3,consortium_3,employees_3,
      empName_3,belongsTo_3,worksOn_3,empId_3,
      projId_3)
*  6  completerefrename(ExtResearchCompaniesMch_3
      ) == (compClass,empClass,projClass,tEmpId,
      companies_0,compName_0,engagedIn_0,budget_0
      ,projects_0,projName_0,getsGrantFrom_0,
      consortium_0,employees_0,empName_0,
      belongsTo_0,worksOn_0,empId_0,projId_0,
      companies,compName,engagedIn,budget,
      projects,projName,getsGrantFrom,consortium,
      employees,empName,belongsTo,worksOn,empId,
      projId = compClass_3,empClass_3,projClass_3
      ,tEmpId_3,companies_2,compName_2,engagedIn_
      2,budget_2,projects_2,projName_2,
      getsGrantFrom_2,consortium_2,employees_2,
      empName_2,belongsTo_2,worksOn_2,empId_2,
      projId_2,companies_3,compName_3,engagedIn_3
      ,budget_3,projects_3,projName_3,
      getsGrantFrom_3,consortium_3,employees_3,
      empName_3,belongsTo_3,worksOn_3,empId_3,
      projId_3)
*  7  vrbrename(ExtResearchCompaniesMch_4) == (
      compRel,empRel,projRel,compRelDes,empRelDes
      ,projRelDes,companies_0,compName_0,
      engagedIn_0,budget_0,projects_0,projName_0,
      getsGrantFrom_0,consortium_0,employees_0,
      empName_0,belongsTo_0,worksOn_0,empId_0,
      projId_0,tEmpId_0,companies,compName,
      engagedIn,budget,projects,projName,
      getsGrantFrom,consortium,employees,empName,
      belongsTo,worksOn,empId,projId,tEmpId,
      compClass,empClass,projClass = compRel_4,
      empRel_4,projRel_4,compRelDes_4,empRelDes_4
```

```
      ,projRelDes_4,companies_3,compName_3,
      engagedIn_3,budget_3,projects_3,projName_3,
      getsGrantFrom_3,consortium_3,employees_3,
      empName_3,belongsTo_3,worksOn_3,empId_3,
      projId_3,tEmpId_3,companies_4,compName_4,
      engagedIn_4,budget_4,projects_4,projName_4,
      getsGrantFrom_4,consortium_4,employees_4,
      empName_4,belongsTo_4,worksOn_4,empId_4,
      projId_4,tEmpId_4,compClass_3,empClass_3,
      projClass_3)
*  8  completerefrename(ExtResearchCompaniesMch_4
      ) == (compRel,empRel,projRel,compRelDes,
      empRelDes,projRelDes,companies_0,compName_0
      ,engagedIn_0,budget_0,projects_0,projName_0
      ,getsGrantFrom_0,consortium_0,employees_0,
      empName_0,belongsTo_0,worksOn_0,empId_0,
      projId_0,tEmpId_0,companies,compName,
      engagedIn,budget,projects,projName,
      getsGrantFrom,consortium,employees,empName,
      belongsTo,worksOn,empId,projId,tEmpId,
      compClass,empClass,projClass = compRel_4,
      empRel_4,projRel_4,compRelDes_4,empRelDes_4
      ,projRelDes_4,companies_3,compName_3,
      engagedIn_3,budget_3,projects_3,projName_3,
      getsGrantFrom_3,consortium_3,employees_3,
      empName_3,belongsTo_3,worksOn_3,empId_3,
      projId_3,tEmpId_3,companies_4,compName_4,
      engagedIn_4,budget_4,projects_4,projName_4,
      getsGrantFrom_4,consortium_4,employees_4,
      empName_4,belongsTo_4,worksOn_4,empId_4,
      projId_4,tEmpId_4,compClass_3,empClass_3,
      projClass_3)



Theory 98 "GetStoredX":
-----------------------

*  1  ctx(ExtResearchCompaniesCtx) == stored
*  2  mch(ExtResearchCompaniesMch) == stored
*  3  ctx(ExtResearchCompaniesCtx2) == stored
*  4  ctx(ExtResearchCompaniesCtx4) == stored


Theory 99 "GetContextListX":
----------------------------

*  1  ctx(ExtResearchCompaniesMch_1_1) ==
```

```
      ExtResearchCompaniesCtx
*  2  ctx(ExtResearchCompaniesMch_1) ==
      ExtResearchCompaniesCtx
*  3  ctx(ExtResearchCompaniesMch_2_1) ==
      ExtResearchCompaniesCtx2
*  4  ctx(ExtResearchCompaniesMch_2) ==
      ExtResearchCompaniesCtx2
*  5  ctx(ExtResearchCompaniesMch_3_2) ==
      ExtResearchCompaniesCtx2
*  6  ctx(ExtResearchCompaniesMch_3) ==
      ExtResearchCompaniesCtx2
*  7  ctx(ExtResearchCompaniesMch_4_3) ==
      ExtResearchCompaniesCtx4
*  8  ctx(ExtResearchCompaniesMch_4) ==
      ExtResearchCompaniesCtx4


Theory 100 "GetContextClausesX":
--------------------------------

*  1  set(ExtResearchCompaniesCtx) == Strings,
      COMPANIES,PROJECTS,EMPLOYEES,CompNames,
      MONEY,EmpNames,ProjNames
*  2  prp(ExtResearchCompaniesCtx) == (CompNames
      = Strings;EmpNames = Strings;ProjNames =
      Strings;FIN(ProjNames);ProjNames/={};FIN(
      EmpNames);EmpNames/={};FIN(MONEY);MONEY/={}
      ;FIN(CompNames);CompNames/={};FIN(EMPLOYEES
      );EMPLOYEES/={};FIN(PROJECTS);PROJECTS/={};
      FIN(COMPANIES);COMPANIES/={};FIN(Strings);
      Strings/={})
*  3  set(ExtResearchCompaniesCtx2) == Strings,
      CompNames,EmpNames,ProjNames,EMPLOYEES,
      PROJECTS,COMPANIES,MONEY,EmpIds
*  4  prp(ExtResearchCompaniesCtx2) == (CompNames
      = Strings;EmpNames = Strings;ProjNames =
      Strings;COMPANIES = CompNames;EMPLOYEES =
      EmpIds;PROJECTS = ProjNames*Agencies;FIN(
      EmpIds);EmpIds/={};FIN(MONEY);MONEY/={};FIN
      (COMPANIES);COMPANIES/={};FIN(PROJECTS);
      PROJECTS/={};FIN(EMPLOYEES);EMPLOYEES/={};
      FIN(ProjNames);ProjNames/={};FIN(EmpNames);
      EmpNames/={};FIN(CompNames);CompNames/={};
      FIN(Strings);Strings/={})
*  5  set(ExtResearchCompaniesCtx4) == Strings,
      CompNames,EmpNames,ProjNames,EMPLOYEES,
      PROJECTS,COMPANIES,MONEY,EmpIds,
      ProjIdRecType,ProjName,GetsGrantFrom,
```

```
      ProjIdRelType,CompRecType,CompName,
      EngagedIn,Budget,CompRelType,EmpRecType,
      EmpId,EmpName,BelongsTo,WorksOn,EmpRelType,
      ProjRecType,ProjId,Consortium,ProjRelType
*  6  prp(ExtResearchCompaniesCtx4) == (CompNames
      = Strings;EmpNames = Strings;ProjNames =
      Strings;COMPANIES = CompNames;EMPLOYEES =
      EmpIds;PROJECTS = ProjIdRecType;
      ProjIdRecType = ProjNames*Agencies;ProjName
      = %x.(x: ProjIdRecType & x = a,b | a);
      GetsGrantFrom = %x.(x: ProjIdRecType & x =
      a,b | b);ProjIdRelType = POW(ProjIdRecType)
      ;CompRecType = CompNames*ProjIdRelType*
      MONEY;CompName = %x.(x: CompRecType & x = a
      ,b,c | a);EngagedIn = %x.(x: CompRecType &
      x = a,b,c | b);Budget = %x.(x: CompRecType
      & x = a,b,c | c);CompRelType = CompNames
      +-> ProjIdRelType*MONEY;EmpRecType = EmpIds
      *EmpNames*CompNames*ProjIdRelType;EmpId = %
      x.(x: EmpRecType & x = a,b,c,d | a);EmpName
      = %x.(x: EmpRecType & x = a,b,c,d | b);
      BelongsTo = %x.(x: EmpRecType & x = a,b,c,d
      | c);WorksOn = %x.(x: EmpRecType & x = a,b,
      c,d | d);EmpRelType = EmpIds +-> EmpNames*
      CompNames*ProjIdRelType;ProjRecType =
      ProjIdRecType*POW(CompNames);ProjId = %x.(x
      : ProjRecType & x = a,b | a);Consortium = %
      x.(x: ProjRecType & x = a,b | b);
      ProjRelType = ProjIdRecType +-> POW(
      CompNames);FIN(ProjRelType);ProjRelType/={}
      ;FIN(Consortium);Consortium/={};FIN(ProjId)
      ;ProjId/={};FIN(ProjRecType);ProjRecType/=
      {};FIN(EmpRelType);EmpRelType/={};FIN(
      WorksOn);WorksOn/={};FIN(BelongsTo);
      BelongsTo/={};FIN(EmpName);EmpName/={};FIN(
      EmpId);EmpId/={};FIN(EmpRecType);EmpRecType
      /={};FIN(CompRelType);CompRelType/={};FIN(
      Budget);Budget/={};FIN(EngagedIn);EngagedIn
      /={};FIN(CompName);CompName/={};FIN(
      CompRecType);CompRecType/={};FIN(
      ProjIdRelType);ProjIdRelType/={};FIN(
      GetsGrantFrom);GetsGrantFrom/={};FIN(
      ProjName);ProjName/={};FIN(ProjIdRecType);
      ProjIdRecType/={};FIN(EmpIds);EmpIds/={};
      FIN(MONEY);MONEY/={};FIN(COMPANIES);
      COMPANIES/={};FIN(PROJECTS);PROJECTS/={};
      FIN(EMPLOYEES);EMPLOYEES/={};FIN(ProjNames)
      ;ProjNames/={};FIN(EmpNames);EmpNames/={};
```

```
      FIN(CompNames);CompNames/={};FIN(Strings);
      Strings/={})


Theory 101 "GetVariableListX":
------------------------------

*  1  undec_varb(ExtResearchCompaniesMch_1_1) ==
      companies,compName,engagedIn,budget,
      projects,projName,getsGrantFrom,consortium,
      employees,empName,belongsTo,worksOn
*  2  undec_varb(ExtResearchCompaniesMch_1) ==
      companies,compName,engagedIn,budget,
      projects,projName,getsGrantFrom,consortium,
      employees,empName,belongsTo,worksOn
*  3  varb(ExtResearchCompaniesMch_1_1__m) ==
      companies_m,compName_m,engagedIn_m,budget_m
      ,projects_m,projName_m,getsGrantFrom_m,
      consortium_m,employees_m,empName_m,
      belongsTo_m,worksOn_m
*  4  varb(ExtResearchCompaniesMch_1__m) ==
      companies_m,compName_m,engagedIn_m,budget_m
      ,projects_m,projName_m,getsGrantFrom_m,
      consortium_m,employees_m,empName_m,
      belongsTo_m,worksOn_m
*  5  undec_varb(ExtResearchCompaniesMch_2_1) ==
      companies,compName,engagedIn,budget,
      projects,projName,getsGrantFrom,consortium,
      employees,empName,belongsTo,worksOn,empId,
      projId
*  6  undec_varb(ExtResearchCompaniesMch_2) ==
      companies,compName,engagedIn,budget,
      projects,projName,getsGrantFrom,consortium,
      employees,empName,belongsTo,worksOn,empId,
      projId
*  7  varb(ExtResearchCompaniesMch_2_1__m) ==
      companies_m,compName_m,engagedIn_m,budget_m
      ,projects_m,projName_m,getsGrantFrom_m,
      consortium_m,employees_m,empName_m,
      belongsTo_m,worksOn_m,empId_m,projId_m
*  8  varb(ExtResearchCompaniesMch_2__m) ==
      companies_m,compName_m,engagedIn_m,budget_m
      ,projects_m,projName_m,getsGrantFrom_m,
      consortium_m,employees_m,empName_m,
      belongsTo_m,worksOn_m,empId_m,projId_m
*  9  undec_overlapvarb(ExtResearchCompaniesMch_2
      ) == companies,compName,engagedIn,budget,
      projects,projName,getsGrantFrom,consortium,
```

```
      employees,empName,belongsTo,worksOn
*  10 overlapvarb(ExtResearchCompaniesMch_2__m)
      == companies_m,compName_m,engagedIn_m,
      budget_m,projects_m,projName_m,
      getsGrantFrom_m,consortium_m,employees_m,
      empName_m,belongsTo_m,worksOn_m
*  11 undec_residualvarb(ExtResearchCompaniesMch_
      2) == empId,projId
*  12 residualvarb(ExtResearchCompaniesMch_2__m)
      == empId_m,projId_m
*  13 undec_residualvarb(ExtResearchCompaniesMch_
      1) == wQxUzJ
*  14 residualvarb(ExtResearchCompaniesMch_1__m)
      == wQxUzJ
*  15 undec_varb(ExtResearchCompaniesMch_3_2) ==
      companies,compName,engagedIn,budget,
      projects,projName,getsGrantFrom,consortium,
      employees,empName,belongsTo,worksOn,empId,
      projId,compClass,empClass,projClass,tEmpId
*  16 undec_varb(ExtResearchCompaniesMch_3) ==
      companies,compName,engagedIn,budget,
      projects,projName,getsGrantFrom,consortium,
      employees,empName,belongsTo,worksOn,empId,
      projId,compClass,empClass,projClass,tEmpId
*  17 varb(ExtResearchCompaniesMch_3_2__m) ==
      companies_m,compName_m,engagedIn_m,budget_m
      ,projects_m,projName_m,getsGrantFrom_m,
      consortium_m,employees_m,empName_m,
      belongsTo_m,worksOn_m,empId_m,projId_m,
      compClass_m,empClass_m,projClass_m,tEmpId_m
*  18 varb(ExtResearchCompaniesMch_3__m) ==
      companies_m,compName_m,engagedIn_m,budget_m
      ,projects_m,projName_m,getsGrantFrom_m,
      consortium_m,employees_m,empName_m,
      belongsTo_m,worksOn_m,empId_m,projId_m,
      compClass_m,empClass_m,projClass_m,tEmpId_m
*  19 undec_overlapvarb(ExtResearchCompaniesMch_3
      ) == companies,compName,engagedIn,budget,
      projects,projName,getsGrantFrom,consortium,
      employees,empName,belongsTo,worksOn,empId,
      projId
*  20 overlapvarb(ExtResearchCompaniesMch_3__m)
      == companies_m,compName_m,engagedIn_m,
      budget_m,projects_m,projName_m,
      getsGrantFrom_m,consortium_m,employees_m,
      empName_m,belongsTo_m,worksOn_m,empId_m,
      projId_m
*  21 undec_residualvarb(ExtResearchCompaniesMch_
```

```
      3) == compClass,empClass,projClass,tEmpId
*  22 residualvarb(ExtResearchCompaniesMch_3__m)
      == compClass_m,empClass_m,projClass_m,
      tEmpId_m
*  23 undec_residualvarb(ExtResearchCompaniesMch_
      2) == wQxUzJ
*  24 residualvarb(ExtResearchCompaniesMch_2__m)
      == wQxUzJ
*  25 undec_varb(ExtResearchCompaniesMch_4_3) ==
      companies,compName,engagedIn,budget,
      projects,projName,getsGrantFrom,consortium,
      employees,empName,belongsTo,worksOn,empId,
      projId,tEmpId,compRel,empRel,projRel,
      compRelDes,empRelDes,projRelDes
*  26 undec_varb(ExtResearchCompaniesMch_4) ==
      companies,compName,engagedIn,budget,
      projects,projName,getsGrantFrom,consortium,
      employees,empName,belongsTo,worksOn,empId,
      projId,tEmpId,compRel,empRel,projRel,
      compRelDes,empRelDes,projRelDes
*  27 varb(ExtResearchCompaniesMch_4_3__m) ==
      companies_m,compName_m,engagedIn_m,budget_m
      ,projects_m,projName_m,getsGrantFrom_m,
      consortium_m,employees_m,empName_m,
      belongsTo_m,worksOn_m,empId_m,projId_m,
      tEmpId_m,compRel_m,empRel_m,projRel_m,
      compRelDes_m,empRelDes_m,projRelDes_m
*  28 varb(ExtResearchCompaniesMch_4__m) ==
      companies_m,compName_m,engagedIn_m,budget_m
      ,projects_m,projName_m,getsGrantFrom_m,
      consortium_m,employees_m,empName_m,
      belongsTo_m,worksOn_m,empId_m,projId_m,
      tEmpId_m,compRel_m,empRel_m,projRel_m,
      compRelDes_m,empRelDes_m,projRelDes_m
*  29 undec_overlapvarb(ExtResearchCompaniesMch_4
      ) == companies,compName,engagedIn,budget,
      projects,projName,getsGrantFrom,consortium,
      employees,empName,belongsTo,worksOn,empId,
      projId,tEmpId
*  30 overlapvarb(ExtResearchCompaniesMch_4__m)
      == companies_m,compName_m,engagedIn_m,
      budget_m,projects_m,projName_m,
      getsGrantFrom_m,consortium_m,employees_m,
      empName_m,belongsTo_m,worksOn_m,empId_m,
      projId_m,tEmpId_m
*  31 undec_residualvarb(ExtResearchCompaniesMch_
      4) == compRel,empRel,projRel,compRelDes,
      empRelDes,projRelDes
```

```
*  32 residualvarb(ExtResearchCompaniesMch_4__m)
      == compRel_m,empRel_m,projRel_m,compRelDes_
      m,empRelDes_m,projRelDes_m
*  33 undec_residualvarb(ExtResearchCompaniesMch_
      3) == compClass,empClass,projClass
*  34 residualvarb(ExtResearchCompaniesMch_3__m)
      == compClass_m,empClass_m,projClass_m


Theory 102 "GetOperationListsX":
--------------------------------

*  1  opnlist(ExtResearchCompaniesMch_1_1) ==
      ExtResearchCompaniesMchremoveProject_1(proj
      ),ExtResearchCompaniesMchremoveEmployee_1(
      empl),(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)),(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)),
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works),
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)
*  2  opnnamelist(ExtResearchCompaniesMch_1_1) ==
      removeProject,removeEmployee,newCompany,
      newEmployee,updateEmployer,updateJobs
*  3  opnlist(ExtResearchCompaniesMch_1) ==
      ExtResearchCompaniesMchremoveProject_1(proj
      ),ExtResearchCompaniesMchremoveEmployee_1(
      empl),(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)),(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)),
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works),
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)
*  4  opnnamelist(ExtResearchCompaniesMch_1) ==
      removeProject,removeEmployee,newCompany,
      newEmployee,updateEmployer,updateJobs
*  5  unrenameopn(ExtResearchCompaniesMch) ==
      ExtResearchCompaniesMchremoveProject,
      ExtResearchCompaniesMchremoveEmployee,
      ExtResearchCompaniesMchnewCompany,
      ExtResearchCompaniesMchnewEmployee,
      ExtResearchCompaniesMchupdateEmployer,
```

```
      ExtResearchCcompaniesMchupdateJobs:=
      removeProject,removeEmployee,newCompany,
      newEmployee,updateEmployer,updateJobs
*  6  absopresidue(ExtResearchCompaniesMch_2) ==
      wQxUzJ
*  7  absopresidue(ExtResearchCompaniesMch_3) ==
      wQxUzJ
*  8  absopresidue(ExtResearchCompaniesMch_4) ==
      wQxUzJ
```

```
Theory 103 "GetOperationX":
---------------------------

*  1  ExtResearchCompaniesMchremoveProject_1(a)
      == (a: projects_1 & consortium_1(a) = {} |
      projName_1,consortium_1,getsGrantFrom_1,
      projects_1:={a}<<|projName_1,{a}<<|
      consortium_1,{a}<<|getsGrantFrom_1,projects
      _1-{a})
*  2  ExtResearchCompaniesMchremoveEmployee_1(a)
      == (a: employees_1 | empName_1,worksOn_1,
      belongsTo_1,employees_1:={a}<<|empName_1,{a
      }<<|worksOn_1,{a}<<|belongsTo_1,employees_1
      -{a})
*  3  a <-- ExtResearchCompaniesMchnewCompany_1(d
      ,c,b) == (d: CompNames-ran(compName_1) & c:
      POW(projects_1) & b: MONEY | @co.(co:
      COMPANIES-companies_1 ==> engagedIn_1,
      companies_1,a,consortium_1,budget_1,
      compName_1:=engagedIn_1<+{co|->c},companies
      _1\/{co},co,consortium_1<+%x.(x: c |
      consortium_1(x)\/{co}),budget_1<+{co|->b},
      compName_1<+{co|->d}))
*  4  a <-- ExtResearchCompaniesMchnewEmployee_1(
      d,c,b) == (d: EmpNames & c: companies_1 & b
      : POW(engagedIn_1(c)) | @ee.(ee: EMPLOYEES-
      employees_1 ==> worksOn_1,employees_1,a,
      belongsTo_1,empName_1:=worksOn_1<+{ee|->b},
      employees_1\/{ee},ee,belongsTo_1<+{ee|->c},
      empName_1<+{ee|->d}))
*  5  ExtResearchCompaniesMchupdateEmployer_1(c,b
      ,a) == (c: employees_1 & b: companies_1 & a
      : POW(engagedIn_1(b)) | worksOn_1,belongsTo
      _1:=worksOn_1<+{c|->a},belongsTo_1<+{c|->b}
      )
```

```
*  6  ExtResearchCompaniesMchupdateJobs_1(b,a) ==
      (b: employees_1 & a: POW(engagedIn_1(
      belongsTo_1(b))) | worksOn_1:=worksOn_1<+{b
      |->a})
*  7  ExtResearchCompaniesMchremoveProject_2(a)
      == (a: projects_2 & consortium_2(a) = {} |
      consortium_2,projects_2:={a}<<|consortium_2
      ,projects_2-{a})
*  8  ExtResearchCompaniesMchremoveEmployee_2(a)
      == (a: employees_2 | empName_2,worksOn_2,
      belongsTo_2,employees_2:={a}<<|empName_2,{a
      }<<|worksOn_2,{a}<<|belongsTo_2,employees_2
      -{a})
*  9  a <-- ExtResearchCompaniesMchnewCompany_2(d
      ,c,b) == (d: CompNames-ran(compName_2) & c:
      POW(projects_2) & b: MONEY | budget_2,
      consortium_2,a,companies_2,engagedIn_2:=
      budget_2<+{d|->b},consortium_2<+%x.(x: c |
      consortium_2(x)\/{d}),d,companies_2\/{d},
      engagedIn_2<+{d|->c})
*  10 a <-- ExtResearchCompaniesMchnewEmployee_2(
      d,c,b) == (d: EmpNames & c: companies_2 & b
      : POW(engagedIn_2(c)) | worksOn_2,employees
      _2,a,belongsTo_2,empName_2:=worksOn_2<+{
      newEmpId|->b},employees_2\/{newEmpId},
      newEmpId,belongsTo_2<+{newEmpId|->c},
      empName_2<+{newEmpId|->d})
*  11 ExtResearchCompaniesMchupdateEmployer_2(c,b
      ,a) == (c: employees_2 & b: companies_2 & a
      : POW(engagedIn_2(b)) | worksOn_2,belongsTo
      _2:=worksOn_2<+{c|->a},belongsTo_2<+{c|->b}
      )
*  12 ExtResearchCompaniesMchupdateJobs_2(b,a) ==
      (b: employees_2 & a: POW(engagedIn_2(
      belongsTo_2(b))) | worksOn_2:=worksOn_2<+{b
      |->a})
*  13 ExtResearchCompaniesMchremoveProject_3(a)
      == (a: projects_3 & consortium_3(a) = {} |
      projClass_3:={a}<<|projClass_3)
*  14 ExtResearchCompaniesMchremoveEmployee_3(a)
      == (a: employees_3 | empClass_3:={a}<<|
      empClass_3)
*  15 a <-- ExtResearchCompaniesMchnewCompany_3(d
      ,c,b) == (d: CompNames-ran(compName_3) & c:
      POW(projects_3) & b: MONEY | projClass_3:=
      projClass_3<+%x.(x: c | projClass_3(x)\/{d}
      );compClass_3:=compClass_3\/{d,c,b};a:=d)
*  16 a <-- ExtResearchCompaniesMchnewEmployee_3(
```

```
      d,c,b) == (d: EmpNames & c: companies_3 & b
      : POW(engagedIn_3(c)) | tEmpId_3:=newEmpId;
      empClass_3:=empClass_3\/{tEmpId_3,d,c,b};a
      :=tEmpId_3)
*  17 ExtResearchCompaniesMchupdateEmployer_3(c,b
      ,a) == (c: employees_3 & b: companies_3 & a
      : POW(engagedIn_3(b)) | empClass_3:=
      empClass_3<+{c|->(empName_3(c),b,a)})
*  18 ExtResearchCompaniesMchupdateJobs_3(b,a) ==
      (b: employees_3 & a: POW(engagedIn_3(
      belongsTo_3(b))) | empClass_3:=empClass_3<+
      {b|->(empName_3(b),belongsTo_3(b),a)})
*  19 ExtResearchCompaniesMchremoveProject_4(a)
      == (a: ProjIdRecType | a: projects_4 &
      consortium_4(projRelDes_4(a)) = {} ==>
      projRel_4:={a}<<|projRel_4 [] not(a:
      projects_4 & consortium_4(projRelDes_4(a))
      = {}) ==> skip)
*  20 ExtResearchCompaniesMchremoveEmployee_4(a)
      == (a: EmpIds | a: employees_4 ==> empRel_4
      :={a}<<|empRel_4 [] not(a: employees_4) ==>
      skip)
*  21 a <-- ExtResearchCompaniesMchnewCompany_4(d
      ,c,b) == (d: CompNames & c: ProjIdRelType &
      b: MONEY | d: CompNames-ran(compName_4) & c
      : POW(projects_4) ==> (projRel_4:=projRel_4
      <+%x.(x: c | projRel_4(x)\/{d});compRel_4:=
      compRel_4\/{d,c,b};a:=d) [] not(d:
      CompNames-ran(compName_4) & c: POW(projects
      _4)) ==> a:=nilCompName)
*  22 a <-- ExtResearchCompaniesMchnewEmployee_4(
      d,c,b) == (d: EmpNames & c: CompNames & b:
      ProjIdRelType | c: companies_4 & b: POW(
      engagedIn_4(compRelDes_4(c))) ==> (tEmpId_4
      :=newEmpId;empRel_4:=empRel_4\/{tEmpId_4,d,
      c,b};a:=tEmpId_4) [] not(c: companies_4 & b
      : POW(engagedIn_4(compRelDes_4(c)))) ==> a
      :=nilEmpId)
*  23 ExtResearchCompaniesMchupdateEmployer_4(c,b
      ,a) == (c: EmpIds & b: CompNames & a:
      ProjIdRelType | c: employees_4 & b:
      companies_4 & a: POW(engagedIn_4(compRelDes
      _4(b))) ==> empRel_4:=empRel_4<+{c|->(
      empName_4(empRelDes_4(c)),b,a)} [] not(c:
      employees_4 & b: companies_4 & a: POW(
      engagedIn_4(compRelDes_4(b)))) ==> skip)
*  24 ExtResearchCompaniesMchupdateJobs_4(b,a) ==
      (b: EmpIds & a: ProjIdRelType | b:
```

```
      employees_4 & a: POW(engagedIn_4(compRelDes
      _4(belongsTo_4(empRelDes_4(b))))) ==>
      empRel_4:=empRel_4<+{b|->(empName_4(
      empRelDes_4(b)),belongsTo_4(empRelDes_4(b))
      ,a)} [] not(b: employees_4 & a: POW(
      engagedIn_4(compRelDes_4(belongsTo_4(
      empRelDes_4(b)))))) ==> skip)



Theory 104 "GetInitialisationX":
--------------------------------

*  1  ini(ExtResearchCompaniesMch_1_1) ==
      compName_1,budget_1,projName_1,consortium_1
      ,empName_1,worksOn_1,belongsTo_1,employees_
      1,getsGrantFrom_1,projects_1,engagedIn_1,
      companies_1:={},{},{},{},{},{},{},{},{},{},
      {},{}
*  2  ini(ExtResearchCompaniesMch_1) == compName_
      1,budget_1,projName_1,consortium_1,empName_
      1,worksOn_1,belongsTo_1,employees_1,
      getsGrantFrom_1,projects_1,engagedIn_1,
      companies_1:={},{},{},{},{},{},{},{},{},{},
      {},{}
*  3  ini(ExtResearchCompaniesMch_2_1) ==
      employees_2,empName_2,belongsTo_2,worksOn_2
      ,consortium_2,empId_2,projId_2,
      getsGrantFrom_2,projName_2,projects_2,
      budget_2,engagedIn_2,compName_2,companies_2
      :={},{},{},{},{},{},{},{},{},{},{},{},{},{}
*  4  ini(ExtResearchCompaniesMch_2) == employees
      _2,empName_2,belongsTo_2,worksOn_2,
      consortium_2,empId_2,projId_2,getsGrantFrom
      _2,projName_2,projects_2,budget_2,engagedIn
      _2,compName_2,companies_2:={},{},{},{},{},
      {},{},{},{},{},{},{},{},{}
*  5  ini(ExtResearchCompaniesMch_3_2) ==
      employees_3,empName_3,belongsTo_3,
      getsGrantFrom_3,consortium_3,empId_3,
      projClass_3,tEmpId_3,empClass_3,compClass_3
      ,projId_3,projName_3,projects_3,worksOn_3,
      budget_3,engagedIn_3,compName_3,companies_3
      :={},{},{},{},{},{},{},{},{},{},{},{},{},{}
      ,{},{},{},{}
*  6  ini(ExtResearchCompaniesMch_3) == employees
      _3,empName_3,belongsTo_3,getsGrantFrom_3,
```

```
      consortium_3,empId_3,projClass_3,tEmpId_3,
      empClass_3,compClass_3,projId_3,projName_3,
      projects_3,worksOn_3,budget_3,engagedIn_3,
      compName_3,companies_3:={},{},{},{},{},{},
      {},{},{},{},{},{},{},{},{},{},{},{},{}
*  7  ini(ExtResearchCompaniesMch_4_3) ==
      employees_4,empName_4,belongsTo_4,
      getsGrantFrom_4,consortium_4,empId_4,empRel
      _4,projRel_4,compRelDes_4,projRelDes_4,
      empRelDes_4,compRel_4,tEmpId_4,projId_4,
      projName_4,projects_4,worksOn_4,budget_4,
      engagedIn_4,compName_4,companies_4:={},{},
      {},{},{},{},{},{},{},{},{},{},{},{},{},{},
      {},{},{},{},{}
*  8  ini(ExtResearchCompaniesMch_4) == employees
      _4,empName_4,belongsTo_4,getsGrantFrom_4,
      consortium_4,empId_4,empRel_4,projRel_4,
      compRelDes_4,projRelDes_4,empRelDes_4,
      compRel_4,tEmpId_4,projId_4,projName_4,
      projects_4,worksOn_4,budget_4,engagedIn_4,
      compName_4,companies_4:={},{},{},{},{},{},
      {},{},{},{},{},{},{},{},{},{},{},{},{},{},
      {}


Theory 105 "GetInvariantX":
---------------------------

*  1  inv(ExtResearchCompaniesMch_1_1) == (
      companies_1: POW(COMPANIES) & compName_1:
      companies_1 --> CompNames & engagedIn_1:
      companies_1 --> POW(projects_1) & budget_1:
      companies_1 --> MONEY & projects_1: POW(
      PROJECTS) & projName_1: projects_1 -->
      ProjNames & getsGrantFrom_1: projects_1 -->
      Agencies & consortium_1: projects_1 --> POW
      (companies_1) & employees_1: POW(EMPLOYEES)
      & empName_1: employees_1 --> EmpNames &
      belongsTo_1: employees_1 --> companies_1 &
      worksOn_1: employees_1 --> POW(projects_1)
      & !x.!y.(x,y: projects_1 => (projName_1(x)
      = projName_1(y) & getsGrantFrom_1(x) =
      getsGrantFrom_1(y) => x = y)) & !x.!y.(x,y:
      companies_1 => (compName_1(x) = compName_1(
      y) => x = y)) & !x.(x: employees_1 =>
      worksOn_1(x) incl engagedIn_1(belongsTo_1(x
      ))) & !x.(x: projects_1 => consortium_1(x)
      = {y | y: companies_1 & x: engagedIn_1(y)})
```

```
    )
*   2  inv(ExtResearchCompaniesMch_1) == (
       companies_1: POW(COMPANIES) & compName_1:
       companies_1 --> CompNames & engagedIn_1:
       companies_1 --> POW(projects_1) & budget_1:
       companies_1 --> MONEY & projects_1: POW(
       PROJECTS) & projName_1: projects_1 -->
       ProjNames & getsGrantFrom_1: projects_1 -->
       Agencies & consortium_1: projects_1 --> POW
       (companies_1) & employees_1: POW(EMPLOYEES)
       & empName_1: employees_1 --> EmpNames &
       belongsTo_1: employees_1 --> companies_1 &
       worksOn_1: employees_1 --> POW(projects_1)
       & !x.!y.(x,y: projects_1 => (projName_1(x)
       = projName_1(y) & getsGrantFrom_1(x) =
       getsGrantFrom_1(y) => x = y)) & !x.!y.(x,y:
       companies_1 => (compName_1(x) = compName_1(
       y) => x = y)) & !x.(x: employees_1 =>
       worksOn_1(x) incl engagedIn_1(belongsTo_1(x
       ))) & !x.(x: projects_1 => consortium_1(x)
       = {y | y: companies_1 & x: engagedIn_1(y)})
       )
*   3  inv(ExtResearchCompaniesMch_2_1) == (
       worksOn_1 = worksOn_2 & belongsTo_1 =
       belongsTo_2 & empName_1 = empName_2 &
       employees_1 = employees_2 & consortium_1 =
       consortium_2 & getsGrantFrom_1 =
       getsGrantFrom_2 & projName_1 = projName_2 &
       projects_1 = projects_2 & budget_1 = budget
       _2 & engagedIn_1 = engagedIn_2 & compName_1
       = compName_2 & companies_1 = companies_2)
*   4  inv(ExtResearchCompaniesMch_2) == (worksOn_
       1 = worksOn_2 & belongsTo_1 = belongsTo_2 &
       empName_1 = empName_2 & employees_1 =
       employees_2 & consortium_1 = consortium_2 &
       getsGrantFrom_1 = getsGrantFrom_2 &
       projName_1 = projName_2 & projects_1 =
       projects_2 & budget_1 = budget_2 &
       engagedIn_1 = engagedIn_2 & compName_1 =
       compName_2 & companies_1 = companies_2)
*   5  inv(ExtResearchCompaniesMch_3_1) == (
       compClass_3: companies_3 --> POW(projects_3
       )*MONEY & empClass_3: employees_3 -->
       EmpNames*companies_3*POW(projects_3) &
       projClass_3: projects_3 --> POW(companies_3
       ) & tEmpId_3: EmpIds & projId_2 = projId_3
       & empId_2 = empId_3 & worksOn_2 = worksOn_3
       & belongsTo_2 = belongsTo_3 & empName_2 =
```

```
      empName_3 & employees_2 = employees_3 &
      consortium_2 = consortium_3 & getsGrantFrom
      _2 = getsGrantFrom_3 & projName_2 =
      projName_3 & projects_2 = projects_3 &
      budget_2 = budget_3 & engagedIn_2 =
      engagedIn_3 & compName_2 = compName_3 &
      companies_2 = companies_3)
*  6  inv(ExtResearchCompaniesMch_3) == (
      compClass_3: companies_3 --> POW(projects_3
      )*MONEY & empClass_3: employees_3 -->
      EmpNames*companies_3*POW(projects_3) &
      projClass_3: projects_3 --> POW(companies_3
      ) & tEmpId_3: EmpIds & projId_2 = projId_3
      & empId_2 = empId_3 & worksOn_2 = worksOn_3
      & belongsTo_2 = belongsTo_3 & empName_2 =
      empName_3 & employees_2 = employees_3 &
      consortium_2 = consortium_3 & getsGrantFrom
      _2 = getsGrantFrom_3 & projName_2 =
      projName_3 & projects_2 = projects_3 &
      budget_2 = budget_3 & engagedIn_2 =
      engagedIn_3 & compName_2 = compName_3 &
      companies_2 = companies_3)
*  7  inv(ExtResearchCompaniesMch_4_1) == (
      compRel_4: CompRelType & compRel_4 =
      compClass_3 & empRel_4: EmpRelType & empRel
      _4 = empClass_3 & projRel_4: ProjRelType &
      projRel_4 = projClass_3 & tEmpId_4: EmpIds
      & tEmpId_3 = tEmpId_4 & projId_3 = projId_4
      & empId_3 = empId_4 & worksOn_3 = worksOn_4
      & belongsTo_3 = belongsTo_4 & empName_3 =
      empName_4 & employees_3 = employees_4 &
      consortium_3 = consortium_4 & getsGrantFrom
      _3 = getsGrantFrom_4 & projName_3 =
      projName_4 & projects_3 = projects_4 &
      budget_3 = budget_4 & engagedIn_3 =
      engagedIn_4 & compName_3 = compName_4 &
      companies_3 = companies_4)
*  8  inv(ExtResearchCompaniesMch_4) == (compRel_
      4: CompRelType & compRel_4 = compClass_3 &
      empRel_4: EmpRelType & empRel_4 = empClass_
      3 & projRel_4: ProjRelType & projRel_4 =
      projClass_3 & tEmpId_4: EmpIds & tEmpId_3 =
      tEmpId_4 & projId_3 = projId_4 & empId_3 =
      empId_4 & worksOn_3 = worksOn_4 & belongsTo
      _3 = belongsTo_4 & empName_3 = empName_4 &
      employees_3 = employees_4 & consortium_3 =
      consortium_4 & getsGrantFrom_3 =
      getsGrantFrom_4 & projName_3 = projName_4 &
```

```
      projects_3 = projects_4 & budget_3 = budget
      _4 & engagedIn_3 = engagedIn_4 & compName_3
      = compName_4 & companies_3 = companies_4)


Theory 106 "GetInvariantUsedX":
-------------------------------

  (leer)


Theory 107 "GetPredicateX":
---------------------------

*  1  ctx(ExtResearchCompaniesMch) => ctx(
      ExtResearchCompaniesCtx)
*  2  inv(ExtResearchCompaniesMch_1) => companies
      _1: POW(COMPANIES) & compName_1: companies_
      1 --> CompNames & engagedIn_1: companies_1
      --> POW(projects_1) & budget_1: companies_1
      --> MONEY & projects_1: POW(PROJECTS) &
      projName_1: projects_1 --> ProjNames &
      getsGrantFrom_1: projects_1 --> Agencies &
      consortium_1: projects_1 --> POW(companies_
      1) & employees_1: POW(EMPLOYEES) & empName_
      1: employees_1 --> EmpNames & belongsTo_1:
      employees_1 --> companies_1 & worksOn_1:
      employees_1 --> POW(projects_1) & !x.!y.(x,
      y: projects_1 => (projName_1(x) = projName_
      1(y) & getsGrantFrom_1(x) = getsGrantFrom_1
      (y) => x = y)) & !x.!y.(x,y: companies_1 =>
      (compName_1(x) = compName_1(y) => x = y)) &
      !x.(x: employees_1 => worksOn_1(x) incl
      engagedIn_1(belongsTo_1(x))) & !x.(x:
      projects_1 => consortium_1(x) = {y | y:
      companies_1 & x: engagedIn_1(y)})
*  3  pre(ExtResearchCompaniesMchremoveProject_1(
      a)) => a: projects_1 & consortium_1(a) = {}
*  4  pre(ExtResearchCompaniesMchremoveEmployee_1
      (a)) => a: employees_1
*  5  pre(a <-- ExtResearchCompaniesMchnewCompany
      _1(d,c,b)) => d: CompNames-ran(compName_1)
      & c: POW(projects_1) & b: MONEY
*  6  pre(a <--
      ExtResearchCompaniesMchnewEmployee_1(d,c,b)
      ) => d: EmpNames & c: companies_1 & b: POW(
      engagedIn_1(c))
*  7  pre(ExtResearchCompaniesMchupdateEmployer_1
```

```
      (c,b,a)) => c: employees_1 & b: companies_1
      & a: POW(engagedIn_1(b))
*  8  pre(ExtResearchCompaniesMchupdateJobs_1(b,a
      )) => b: employees_1 & a: POW(engagedIn_1(
      belongsTo_1(b)))
*  9  ctx(ExtResearchCompaniesMch) => ctx(
      ExtResearchCompaniesCtx2)
*  10 inv(ExtResearchCompaniesMch_2) => worksOn_1
      = worksOn_2 & belongsTo_1 = belongsTo_2 &
      empName_1 = empName_2 & employees_1 =
      employees_2 & consortium_1 = consortium_2 &
      getsGrantFrom_1 = getsGrantFrom_2 &
      projName_1 = projName_2 & projects_1 =
      projects_2 & budget_1 = budget_2 &
      engagedIn_1 = engagedIn_2 & compName_1 =
      compName_2 & companies_1 = companies_2
*  11 inv(ExtResearchCompaniesMch_2) => inv(
      ExtResearchCompaniesMch_1)
*  12 pre(ExtResearchCompaniesMchremoveProject_2(
      a)) => a: projects_2 & consortium_2(a) = {}
*  13 pre(ExtResearchCompaniesMchremoveEmployee_2
      (a)) => a: employees_2
*  14 pre(a <-- ExtResearchCompaniesMchnewCompany
      _2(d,c,b)) => d: CompNames-ran(compName_2)
      & c: POW(projects_2) & b: MONEY
*  15 pre(a <--
      ExtResearchCompaniesMchnewEmployee_2(d,c,b)
      ) => d: EmpNames & c: companies_2 & b: POW(
      engagedIn_2(c))
*  16 pre(ExtResearchCompaniesMchupdateEmployer_2
      (c,b,a)) => c: employees_2 & b: companies_2
      & a: POW(engagedIn_2(b))
*  17 pre(ExtResearchCompaniesMchupdateJobs_2(b,a
      )) => b: employees_2 & a: POW(engagedIn_2(
      belongsTo_2(b)))
*  18 ctx(ExtResearchCompaniesMch) => ctx(
      ExtResearchCompaniesCtx2)
*  19 inv(ExtResearchCompaniesMch_3) => compClass
      _3: companies_3 --> POW(projects_3)*MONEY &
      empClass_3: employees_3 --> EmpNames*
      companies_3*POW(projects_3) & projClass_3:
      projects_3 --> POW(companies_3) & tEmpId_3:
      EmpIds & projId_2 = projId_3 & empId_2 =
      empId_3 & worksOn_2 = worksOn_3 & belongsTo
      _2 = belongsTo_3 & empName_2 = empName_3 &
      employees_2 = employees_3 & consortium_2 =
      consortium_3 & getsGrantFrom_2 =
      getsGrantFrom_3 & projName_2 = projName_3 &
```

```
      projects_2 = projects_3 & budget_2 = budget
      _3 & engagedIn_2 = engagedIn_3 & compName_2
      = compName_3 & companies_2 = companies_3
*  20 inv(ExtResearchCompaniesMch_3) => inv(
      ExtResearchCompaniesMch_2)
*  21 pre(ExtResearchCompaniesMchremoveProject_3(
      a)) => a: projects_3 & consortium_3(a) = {}
*  22 pre(ExtResearchCompaniesMchremoveEmployee_3
      (a)) => a: employees_3
*  23 pre(a <-- ExtResearchCompaniesMchnewCompany
      _3(d,c,b)) => d: CompNames-ran(compName_3)
      & c: POW(projects_3) & b: MONEY
*  24 pre(a <--
      ExtResearchCompaniesMchnewEmployee_3(d,c,b)
      ) => d: EmpNames & c: companies_3 & b: POW(
      engagedIn_3(c))
*  25 pre(ExtResearchCompaniesMchupdateEmployer_3
      (c,b,a)) => c: employees_3 & b: companies_3
      & a: POW(engagedIn_3(b))
*  26 pre(ExtResearchCompaniesMchupdateJobs_3(b,a
      )) => b: employees_3 & a: POW(engagedIn_3(
      belongsTo_3(b)))
*  27 ctx(ExtResearchCompaniesMch) => ctx(
      ExtResearchCompaniesCtx4)
*  28 inv(ExtResearchCompaniesMch_4) => compRel_4
      : CompRelType & compRel_4 = compClass_3 &
      empRel_4: EmpRelType & empRel_4 = empClass_
      3 & projRel_4: ProjRelType & projRel_4 =
      projClass_3 & tEmpId_4: EmpIds & tEmpId_3 =
      tEmpId_4 & projId_3 = projId_4 & empId_3 =
      empId_4 & worksOn_3 = worksOn_4 & belongsTo
      _3 = belongsTo_4 & empName_3 = empName_4 &
      employees_3 = employees_4 & consortium_3 =
      consortium_4 & getsGrantFrom_3 =
      getsGrantFrom_4 & projName_3 = projName_4 &
      projects_3 = projects_4 & budget_3 = budget
      _4 & engagedIn_3 = engagedIn_4 & compName_3
      = compName_4 & companies_3 = companies_4
*  29 inv(ExtResearchCompaniesMch_4) => inv(
      ExtResearchCompaniesMch_3)
*  30 pre(ExtResearchCompaniesMchremoveProject_4(
      a)) => a: ProjIdRecType
*  31 pre(ExtResearchCompaniesMchremoveEmployee_4
      (a)) => a: EmpIds
*  32 pre(a <-- ExtResearchCompaniesMchnewCompany
      _4(d,c,b)) => d: CompNames & c:
      ProjIdRelType & b: MONEY
*  33 pre(a <--
```

```
      ExtResearchCompaniesMchnewEmployee_4(d,c,b)
      ) => d: EmpNames & c: CompNames & b:
      ProjIdRelType
*  34 pre(ExtResearchCompaniesMchupdateEmployer_4
      (c,b,a)) => c: EmpIds & b: CompNames & a:
      ProjIdRelType
*  35 pre(ExtResearchCompaniesMchupdateJobs_4(b,a
      )) => b: EmpIds & a: ProjIdRelType



Theory 108 "GetDefinitionX":
----------------------------

*  1  projId_2 = %x.(x: projects_2 | x)
*  2  getsGrantFrom_2 = %x.(x: projects_2 & x = a
      ,b | b)
*  3  projName_2 = %x.(x: projects_2 & x = a,b |
      a)
*  4  empId_2 = %x.(x: employees_2 | x)
*  5  compName_2 = %x.(x: companies_2 | x)
*  6  projects_3 = dom(projClass_3)
*  7  employees_3 = dom(empClass_3)
*  8  companies_3 = dom(compClass_3)
*  9  consortium_3 = %x.(x: projects_3 |
      projClass_3(x))
*  10 worksOn_3 = %x.(x: employees_3 & empClass_3
      (x) = a,b,c | c)
*  11 belongsTo_3 = %x.(x: employees_3 & empClass
      _3(x) = a,b,c | b)
*  12 empName_3 = %x.(x: employees_3 & empClass_3
      (x) = a,b,c | a)
*  13 budget_3 = %x.(x: companies_3 & compClass_3
      (x) = a,b | b)
*  14 engagedIn_3 = %x.(x: companies_3 &
      compClass_3(x) = a,b | a)
*  15 projId_3 = %x.(x: projects_3 | x)
*  16 getsGrantFrom_3 = %x.(x: projects_3 & x = a
      ,b | b)
*  17 projName_3 = %x.(x: projects_3 & x = a,b |
      a)
*  18 empId_3 = %x.(x: employees_3 | x)
*  19 compName_3 = %x.(x: companies_3 | x)
*  20 projects_4 = dom(projRel_4)
*  21 employees_4 = dom(empRel_4)
*  22 companies_4 = dom(compRel_4)
*  23 consortium_4 = projRel_4<|Consortium
```

```
*  24 getsGrantFrom_4 = dom(projRel_4)<|
      GetsGrantFrom
*  25 projName_4 = dom(projRel_4)<|ProjName
*  26 projId_4 = projRel_4<|ProjId
*  27 worksOn_4 = empRel_4<|WorksOn
*  28 belongsTo_4 = empRel_4<|BelongsTo
*  29 empName_4 = empRel_4<|EmpName
*  30 empId_4 = empRel_4<|EmpId
*  31 budget_4 = compRel_4<|Budget
*  32 engagedIn_4 = compRel_4<|EngagedIn
*  33 compName_4 = compRel_4<|CompName
*  34 projRelDes_4 = %x.(x: dom(projRel_4) &
      projRel_4(x) = a | x,a)
*  35 empRelDes_4 = %x.(x: dom(empRel_4) & empRel
      _4(x) = a,b,c | x,a,b,c)
*  36 compRelDes_4 = %x.(x: dom(compRel_4) &
      compRel_4(x) = a,b | x,a,b)


Theory 109 "SemanticsX":
------------------------

  (leer)


Theory 110 "FwdSemanticsX":
---------------------------

*  1  ctx(ExtResearchCompaniesCtx) => FIN(Strings
      )
*  2  ctx(ExtResearchCompaniesCtx) => Strings/={}
*  3  ctx(ExtResearchCompaniesCtx) => FIN(
      COMPANIES)
*  4  ctx(ExtResearchCompaniesCtx) => COMPANIES/=
      {}
*  5  ctx(ExtResearchCompaniesCtx) => FIN(
      PROJECTS)
*  6  ctx(ExtResearchCompaniesCtx) => PROJECTS/=
      {}
*  7  ctx(ExtResearchCompaniesCtx) => FIN(
      EMPLOYEES)
*  8  ctx(ExtResearchCompaniesCtx) => EMPLOYEES/=
      {}
*  9  ctx(ExtResearchCompaniesCtx) => FIN(
      CompNames)
*  10 ctx(ExtResearchCompaniesCtx) => CompNames/=
      {}
*  11 ctx(ExtResearchCompaniesCtx) => FIN(MONEY)
```

```
*  12 ctx(ExtResearchCompaniesCtx) => MONEY/={}
*  13 ctx(ExtResearchCompaniesCtx) => FIN(
      EmpNames)
*  14 ctx(ExtResearchCompaniesCtx) => EmpNames/=
      {}
*  15 ctx(ExtResearchCompaniesCtx) => FIN(
      ProjNames)
*  16 ctx(ExtResearchCompaniesCtx) => ProjNames/=
      {}
*  17 ctx(ExtResearchCompaniesCtx) => CompNames =
      Strings
*  18 ctx(ExtResearchCompaniesCtx) => EmpNames =
      Strings
*  19 ctx(ExtResearchCompaniesCtx) => ProjNames =
      Strings
*  20 ctx(ExtResearchCompaniesCtx2) => FIN(
      Strings)
*  21 ctx(ExtResearchCompaniesCtx2) => Strings/=
      {}
*  22 ctx(ExtResearchCompaniesCtx2) => FIN(
      CompNames)
*  23 ctx(ExtResearchCompaniesCtx2) => CompNames
      /={}
*  24 ctx(ExtResearchCompaniesCtx2) => FIN(
      EmpNames)
*  25 ctx(ExtResearchCompaniesCtx2) => EmpNames/=
      {}
*  26 ctx(ExtResearchCompaniesCtx2) => FIN(
      ProjNames)
*  27 ctx(ExtResearchCompaniesCtx2) => ProjNames
      /={}
*  28 ctx(ExtResearchCompaniesCtx2) => FIN(
      EMPLOYEES)
*  29 ctx(ExtResearchCompaniesCtx2) => EMPLOYEES
      /={}
*  30 ctx(ExtResearchCompaniesCtx2) => FIN(
      PROJECTS)
*  31 ctx(ExtResearchCompaniesCtx2) => PROJECTS/=
      {}
*  32 ctx(ExtResearchCompaniesCtx2) => FIN(
      COMPANIES)
*  33 ctx(ExtResearchCompaniesCtx2) => COMPANIES
      /={}
*  34 ctx(ExtResearchCompaniesCtx2) => FIN(MONEY)
*  35 ctx(ExtResearchCompaniesCtx2) => MONEY/={}
*  36 ctx(ExtResearchCompaniesCtx2) => FIN(EmpIds
      )
```

```
*  37 ctx(ExtResearchCompaniesCtx2) => EmpIds/={}
*  38 ctx(ExtResearchCompaniesCtx2) => CompNames
      = Strings
*  39 ctx(ExtResearchCompaniesCtx2) => EmpNames =
      Strings
*  40 ctx(ExtResearchCompaniesCtx2) => ProjNames
      = Strings
*  41 ctx(ExtResearchCompaniesCtx2) => COMPANIES
      = CompNames
*  42 ctx(ExtResearchCompaniesCtx2) => EMPLOYEES
      = EmpIds
*  43 ctx(ExtResearchCompaniesCtx2) => PROJECTS =
      ProjNames*Agencies
*  44 ctx(ExtResearchCompaniesCtx4) => FIN(
      Strings)
*  45 ctx(ExtResearchCompaniesCtx4) => Strings/=
      {}
*  46 ctx(ExtResearchCompaniesCtx4) => FIN(
      CompNames)
*  47 ctx(ExtResearchCompaniesCtx4) => CompNames
      /={}
*  48 ctx(ExtResearchCompaniesCtx4) => FIN(
      EmpNames)
*  49 ctx(ExtResearchCompaniesCtx4) => EmpNames/=
      {}
*  50 ctx(ExtResearchCompaniesCtx4) => FIN(
      ProjNames)
*  51 ctx(ExtResearchCompaniesCtx4) => ProjNames
      /={}
*  52 ctx(ExtResearchCompaniesCtx4) => FIN(
      EMPLOYEES)
*  53 ctx(ExtResearchCompaniesCtx4) => EMPLOYEES
      /={}
*  54 ctx(ExtResearchCompaniesCtx4) => FIN(
      PROJECTS)
*  55 ctx(ExtResearchCompaniesCtx4) => PROJECTS/=
      {}
*  56 ctx(ExtResearchCompaniesCtx4) => FIN(
      COMPANIES)
*  57 ctx(ExtResearchCompaniesCtx4) => COMPANIES
      /={}
*  58 ctx(ExtResearchCompaniesCtx4) => FIN(MONEY)
*  59 ctx(ExtResearchCompaniesCtx4) => MONEY/={}
*  60 ctx(ExtResearchCompaniesCtx4) => FIN(EmpIds
      )
*  61 ctx(ExtResearchCompaniesCtx4) => EmpIds/={}
*  62 ctx(ExtResearchCompaniesCtx4) => FIN(
      ProjIdRecType)
```

```
*  63 ctx(ExtResearchCompaniesCtx4) =>
      ProjIdRecType/={}
*  64 ctx(ExtResearchCompaniesCtx4) => FIN(
      ProjName)
*  65 ctx(ExtResearchCompaniesCtx4) => ProjName/=
      {}
*  66 ctx(ExtResearchCompaniesCtx4) => FIN(
      GetsGrantFrom)
*  67 ctx(ExtResearchCompaniesCtx4) =>
      GetsGrantFrom/={}
*  68 ctx(ExtResearchCompaniesCtx4) => FIN(
      ProjIdRelType)
*  69 ctx(ExtResearchCompaniesCtx4) =>
      ProjIdRelType/={}
*  70 ctx(ExtResearchCompaniesCtx4) => FIN(
      CompRecType)
*  71 ctx(ExtResearchCompaniesCtx4) =>
      CompRecType/={}
*  72 ctx(ExtResearchCompaniesCtx4) => FIN(
      CompName)
*  73 ctx(ExtResearchCompaniesCtx4) => CompName/=
      {}
*  74 ctx(ExtResearchCompaniesCtx4) => FIN(
      EngagedIn)
*  75 ctx(ExtResearchCompaniesCtx4) => EngagedIn
      /={}
*  76 ctx(ExtResearchCompaniesCtx4) => FIN(Budget
      )
*  77 ctx(ExtResearchCompaniesCtx4) => Budget/={}
*  78 ctx(ExtResearchCompaniesCtx4) => FIN(
      CompRelType)
*  79 ctx(ExtResearchCompaniesCtx4) =>
      CompRelType/={}
*  80 ctx(ExtResearchCompaniesCtx4) => FIN(
      EmpRecType)
*  81 ctx(ExtResearchCompaniesCtx4) => EmpRecType
      /={}
*  82 ctx(ExtResearchCompaniesCtx4) => FIN(EmpId)
*  83 ctx(ExtResearchCompaniesCtx4) => EmpId/={}
*  84 ctx(ExtResearchCompaniesCtx4) => FIN(
      EmpName)
*  85 ctx(ExtResearchCompaniesCtx4) => EmpName/=
      {}
*  86 ctx(ExtResearchCompaniesCtx4) => FIN(
      BelongsTo)
*  87 ctx(ExtResearchCompaniesCtx4) => BelongsTo
      /={}
*  88 ctx(ExtResearchCompaniesCtx4) => FIN(
```

```
      WorksOn)
*  89 ctx(ExtResearchCompaniesCtx4) => WorksOn/=
      {}
*  90 ctx(ExtResearchCompaniesCtx4) => FIN(
      EmpRelType)
*  91 ctx(ExtResearchCompaniesCtx4) => EmpRelType
      /={}
*  92 ctx(ExtResearchCompaniesCtx4) => FIN(
      ProjRecType)
*  93 ctx(ExtResearchCompaniesCtx4) =>
      ProjRecType/={}
*  94 ctx(ExtResearchCompaniesCtx4) => FIN(ProjId
      )
*  95 ctx(ExtResearchCompaniesCtx4) => ProjId/={}
*  96 ctx(ExtResearchCompaniesCtx4) => FIN(
      Consortium)
*  97 ctx(ExtResearchCompaniesCtx4) => Consortium
      /={}
*  98 ctx(ExtResearchCompaniesCtx4) => FIN(
      ProjRelType)
*  99 ctx(ExtResearchCompaniesCtx4) =>
      ProjRelType/={}
*  100 ctx(ExtResearchCompaniesCtx4) => CompNames
      = Strings
*  101 ctx(ExtResearchCompaniesCtx4) => EmpNames
      = Strings
*  102 ctx(ExtResearchCompaniesCtx4) => ProjNames
      = Strings
*  103 ctx(ExtResearchCompaniesCtx4) => COMPANIES
      = CompNames
*  104 ctx(ExtResearchCompaniesCtx4) => EMPLOYEES
      = EmpIds
*  105 ctx(ExtResearchCompaniesCtx4) => PROJECTS
      = ProjIdRecType
*  106 ctx(ExtResearchCompaniesCtx4) =>
      ProjIdRecType = ProjNames*Agencies
*  107 ctx(ExtResearchCompaniesCtx4) => ProjName
      = %x.(x: ProjIdRecType & x = a,b | a)
*  108 ctx(ExtResearchCompaniesCtx4) =>
      GetsGrantFrom = %x.(x: ProjIdRecType & x =
      a,b | b)
*  109 ctx(ExtResearchCompaniesCtx4) =>
      ProjIdRelType = POW(ProjIdRecType)
*  110 ctx(ExtResearchCompaniesCtx4) =>
      CompRecType = CompNames*ProjIdRelType*MONEY
*  111 ctx(ExtResearchCompaniesCtx4) => CompName
      = %x.(x: CompRecType & x = a,b,c | a)
*  112 ctx(ExtResearchCompaniesCtx4) => EngagedIn
```

```
      = %x.(x: CompRecType & x = a,b,c | b)
*  113 ctx(ExtResearchCompaniesCtx4) => Budget =
      %x.(x: CompRecType & x = a,b,c | c)
*  114 ctx(ExtResearchCompaniesCtx4) =>
      CompRelType = CompNames +-> ProjIdRelType*
      MONEY
*  115 ctx(ExtResearchCompaniesCtx4) =>
      EmpRecType = EmpIds*EmpNames*CompNames*
      ProjIdRelType
*  116 ctx(ExtResearchCompaniesCtx4) => EmpId = %
      x.(x: EmpRecType & x = a,b,c,d | a)
*  117 ctx(ExtResearchCompaniesCtx4) => EmpName =
      %x.(x: EmpRecType & x = a,b,c,d | b)
*  118 ctx(ExtResearchCompaniesCtx4) => BelongsTo
      = %x.(x: EmpRecType & x = a,b,c,d | c)
*  119 ctx(ExtResearchCompaniesCtx4) => WorksOn =
      %x.(x: EmpRecType & x = a,b,c,d | d)
*  120 ctx(ExtResearchCompaniesCtx4) =>
      EmpRelType = EmpIds +-> EmpNames*CompNames*
      ProjIdRelType
*  121 ctx(ExtResearchCompaniesCtx4) =>
      ProjRecType = ProjIdRecType*POW(CompNames)
*  122 ctx(ExtResearchCompaniesCtx4) => ProjId =
      %x.(x: ProjRecType & x = a,b | a)
*  123 ctx(ExtResearchCompaniesCtx4) =>
      Consortium = %x.(x: ProjRecType & x = a,b |
      b)
*  124 ctx(ExtResearchCompaniesCtx4) =>
      ProjRelType = ProjIdRecType +-> POW(
      CompNames)


Theory 117 "GetInvDefinitionX":
-------------------------------

*  1  %x.(x: projects_2 | x) = projId_2
*  2  %x.(x: projects_2 & x = a,b | b) =
      getsGrantFrom_2
*  3  %x.(x: projects_2 & x = a,b | a) = projName
      _2
*  4  %x.(x: employees_2 | x) = empId_2
*  5  %x.(x: companies_2 | x) = compName_2
*  6  dom(projClass_3) = projects_3
*  7  dom(empClass_3) = employees_3
*  8  dom(compClass_3) = companies_3
*  9  %x.(x: projects_3 | projClass_3(x)) =
      consortium_3
*  10 %x.(x: employees_3 & empClass_3(x) = a,b,c
```

```
      | c) = worksOn_3
*  11 %x.(x: employees_3 & empClass_3(x) = a,b,c
      | b) = belongsTo_3
*  12 %x.(x: employees_3 & empClass_3(x) = a,b,c
      | a) = empName_3
*  13 %x.(x: companies_3 & compClass_3(x) = a,b |
      b) = budget_3
*  14 %x.(x: companies_3 & compClass_3(x) = a,b |
      a) = engagedIn_3
*  15 %x.(x: projects_3 | x) = projId_3
*  16 %x.(x: projects_3 & x = a,b | b) =
      getsGrantFrom_3
*  17 %x.(x: projects_3 & x = a,b | a) = projName
      _3
*  18 %x.(x: employees_3 | x) = empId_3
*  19 %x.(x: companies_3 | x) = compName_3
*  20 dom(projRel_4) = projects_4
*  21 dom(empRel_4) = employees_4
*  22 dom(compRel_4) = companies_4
*  23 projRel_4<|Consortium = consortium_4
*  24 dom(projRel_4)<|GetsGrantFrom =
      getsGrantFrom_4
*  25 dom(projRel_4)<|ProjName = projName_4
*  26 projRel_4<|ProjId = projId_4
*  27 empRel_4<|WorksOn = worksOn_4
*  28 empRel_4<|BelongsTo = belongsTo_4
*  29 empRel_4<|EmpName = empName_4
*  30 empRel_4<|EmpId = empId_4
*  31 compRel_4<|Budget = budget_4
*  32 compRel_4<|EngagedIn = engagedIn_4
*  33 compRel_4<|CompName = compName_4
*  34 %x.(x: dom(projRel_4) & projRel_4(x) = a |
      x,a) = projRelDes_4
*  35 %x.(x: dom(empRel_4) & empRel_4(x) = a,b,c
      | x,a,b,c) = empRelDes_4
*  36 %x.(x: dom(compRel_4) & compRel_4(x) = a,b
      | x,a,b) = compRelDes_4
```

Anhang E

Die Lemmas des Verfeinerungsbeispiels

Es werden die für das Verfeinerungsbeispiel erzeugten *mathematischen* und *Typlemmas*, aufgegliedert nach den unterschiedlichen Verfeinerungsebenen, gezeigt.

E.1 Typlemmas

E.1.1 Typlemmas der Konsistenznachweise

```
*L 1  ctx(ExtResearchCompaniesMch) => {}: POW(
      COMPANIES)
*L 2  ctx(ExtResearchCompaniesMch) => {}: {} -->
      CompNames
*L 3  ctx(ExtResearchCompaniesMch) => {}: {} -->
      POW({})
*L 4  ctx(ExtResearchCompaniesMch) => {}: {} -->
      MONEY
*L 5  ctx(ExtResearchCompaniesMch) => {}: POW(
      PROJECTS)
*L 6  ctx(ExtResearchCompaniesMch) => {}: {} -->
      ProjNames
*L 7  ctx(ExtResearchCompaniesMch) => {}: {} -->
      Agencies
*L 8  ctx(ExtResearchCompaniesMch) => {}: {} -->
      POW({})
*L 9  ctx(ExtResearchCompaniesMch) => {}: POW(
      EMPLOYEES)
*L 10 ctx(ExtResearchCompaniesMch) => {}: {} -->
```

```
      EmpNames
*L 11 ctx(ExtResearchCompaniesMch) => {}: {} -->
      {}
*L 12 ctx(ExtResearchCompaniesMch) => {}: {} -->
      POW({})
*L 13 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_1) =>
      engagedIn_1: companies_1 --> POW(projects_1
      -{proj})
*L 14 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_1) =>
      projects_1-{proj}: POW(PROJECTS)
*L 15 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_1) => {
      proj}<<|projName_1: projects_1-{proj} -->
      ProjNames
*L 16 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_1) => {
      proj}<<|getsGrantFrom_1: projects_1-{proj}
      --> Agencies
*L 17 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_1) => {
      proj}<<|consortium_1: projects_1-{proj} -->
      POW(companies_1)
*L 18 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_1) =>
      worksOn_1: employees_1 --> POW(projects_1-{
      proj})
*L 19 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_1) =>
      employees_1-{empl}: POW(EMPLOYEES)
*L 20 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
```

```
      empl)) & asn(ExtResearchCompaniesMch_1) =>
      {empl}<<|empName_1: employees_1-{empl} -->
      EmpNames
*L 21 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_1) =>
      {empl}<<|belongsTo_1: employees_1-{empl}
      --> companies_1
*L 22 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_1) =>
      {empl}<<|worksOn_1: employees_1-{empl} -->
      POW(projects_1)
*L 23 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_1) & co: COMPANIES-
      companies_1 => companies_1\/{co}: POW(
      COMPANIES)
*L 24 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_1) & co: COMPANIES-
      companies_1 => compName_1<+{co|->newname}:
      companies_1\/{co} --> CompNames
*L 25 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_1) & co: COMPANIES-
      companies_1 => engagedIn_1<+{co|->ventures}
      : companies_1\/{co} --> POW(projects_1)
*L 26 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_1) & co: COMPANIES-
      companies_1 => budget_1<+{co|->budg}:
      companies_1\/{co} --> MONEY
*L 27 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_1) & co: COMPANIES-
```

```
      companies_1 => consortium_1<+%x.(x:
      ventures | consortium_1(x)\/{co}): projects
      _1 --> POW(companies_1\/{co})
*L 28 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_1) & co: COMPANIES-
      companies_1 => belongsTo_1: employees_1 -->
      (companies_1\/{co})
*L 29 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_1) & ee: EMPLOYEES-
      employees_1 => employees_1\/{ee}: POW(
      EMPLOYEES)
*L 30 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_1) & ee: EMPLOYEES-
      employees_1 => empName_1<+{ee|->newname}:
      employees_1\/{ee} --> EmpNames
*L 31 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_1) & ee: EMPLOYEES-
      employees_1 => belongsTo_1<+{ee|->belongs}:
      employees_1\/{ee} --> companies_1
*L 32 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_1) & ee: EMPLOYEES-
      employees_1 => worksOn_1<+{ee|->works}:
      employees_1\/{ee} --> POW(projects_1)
*L 33 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_1) => belongsTo_1<+
      {empl|->belongs}: employees_1 --> companies
      _1
*L 34 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
```

```
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_1) => worksOn_1<+{
      empl|->works}: employees_1 --> POW(projects
      _1)
*L 35 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_1) =>
      worksOn_1<+{empl|->works}: employees_1 -->
      POW(projects_1)
```

E.1.2 Typlemmas der ersten Verfeinerung

```
*L 36 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_2) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_2) => newname:
      Strings-companies_2
*L 37 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_2) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_2) => newEmpId:
      EmpIds-employees_2
```

E.1.3 Typlemmas der zweiten Verfeinerung

```
*L 38 ctx(ExtResearchCompaniesMch) => {}: {} -->
      POW({})*MONEY
*L 39 ctx(ExtResearchCompaniesMch) => {}: {} -->
      Strings*{}*POW({})
*L 40 ctx(ExtResearchCompaniesMch) => {}: {} -->
      POW({})
*L 41 ctx(ExtResearchCompaniesMch) => {}: EmpIds
*L 42 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_3) =>
      compClass_3: companies_3 --> POW(dom({proj}
      <<|projClass_3))*MONEY
*L 43 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
```

```
      )) & asn(ExtResearchCompaniesMch_3) =>
      empClass_3: dom(empClass_3) --> Strings*dom
      (compClass_3)*POW(dom({proj}<<|projClass_3)
      )
*L 44 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_3) => {
      proj}<<|projClass_3: dom({proj}<<|projClass
      _3) --> POW(companies_3)
*L 45 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_3) =>
      {empl}<<|empClass_3: dom({empl}<<|empClass_
      3) --> Strings*dom(compClass_3)*POW(dom(
      projClass_3))
*L 46 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_3) => compClass_3\/
      {newname,ventures,budg}: dom(compClass_3\/{
      newname,ventures,budg}) --> POW(dom(
      projClass_3<+%x.(x: ventures | projClass_3(
      x)\/{newname})))*MONEY
*L 47 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_3) => empClass_3:
      dom(empClass_3) --> Strings*dom(compClass_3
      \/{newname,ventures,budg})*POW(dom(
      projClass_3<+%x.(x: ventures | projClass_3(
      x)\/{newname})))
*L 48 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_3) => projClass_3<+
      %x.(x: ventures | projClass_3(x)\/{newname}
      ): dom(projClass_3<+%x.(x: ventures |
      projClass_3(x)\/{newname})) --> POW(dom(
      compClass_3\/{newname,ventures,budg}))
*L 49 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
```

```
      ExtResearchCompaniesMch_3) => x: dom(
      projClass_3)
*L 50 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_3) => empClass_3\/{
      newEmpId,newname,belongs,works}: dom(
      empClass_3\/{newEmpId,newname,belongs,works
      }) --> Strings*dom(compClass_3)*POW(dom(
      projClass_3))
*L 51 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_3) => newEmpId:
      EmpIds
*L 52 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_3) => empClass_3<+{
      empl|->((%x.(x: dom(empClass_3) & empClass_
      3(x) = a,b,c | a))(empl),belongs,works)}:
      dom(empClass_3<+{empl|->((%x.(x: dom(
      empClass_3) & empClass_3(x) = a,b,c | a))(
      empl),belongs,works)}) --> Strings*dom(
      compClass_3)*POW(dom(projClass_3))
*L 53 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_3) =>
      empClass_3<+{empl|->((%x.(x: dom(empClass_3
      ) & empClass_3(x) = a,b,c | a))(empl),(%x.(
      x: dom(empClass_3) & empClass_3(x) = a,b,c
      | b))(empl),works)}: dom(empClass_3<+{empl
      |->((%x.(x: dom(empClass_3) & empClass_3(x)
      = a,b,c | a))(empl),(%x.(x: dom(empClass_3)
      & empClass_3(x) = a,b,c | b))(empl),works)}
      ) --> Strings*dom(compClass_3)*POW(dom(
      projClass_3))
```

E.1.4 Typlemmas der dritten Verfeinerung

```
*L 54 ctx(ExtResearchCompaniesMch) => {}: Strings
      +-> POW(Strings*Agencies)*MONEY
```

```
*L 55 ctx(ExtResearchCompaniesMch) => {}: EmpIds
      +-> Strings*Strings*POW(Strings*Agencies)
*L 56 ctx(ExtResearchCompaniesMch) => {}: Strings
      *Agencies +-> POW(Strings)
*L 57 ctx(ExtResearchCompaniesMch) => {}: EmpIds
*L 58 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_4) & proj:
      dom(projRel_4) & (projRel_4<|Consortium)((%
      x.(x: dom(projRel_4) & projRel_4(x) = a | x
      ,a))(proj)) = {} => {proj}<<|projClass_3:
      Strings*Agencies +-> POW(Strings)
*L 59 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_4) => proj
      : Strings*Agencies
*L 60 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_4) &
      empl: dom(empRel_4) => {empl}<<|empClass_3:
      EmpIds +-> Strings*Strings*POW(Strings*
      Agencies)
*L 61 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_4) =>
      empl: EmpIds
*L 62 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) & newname:
      CompNames-ran(compRel_4<|CompName) &
      ventures: POW(dom(projRel_4)) => compClass_
      3\/{newname,ventures,budg}: Strings +-> POW
      (Strings*Agencies)*MONEY
*L 63 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) & newname:
      CompNames-ran(compRel_4<|CompName) &
      ventures: POW(dom(projRel_4)) => projClass_
      3<+%x.(x: ventures | projClass_3(x)\/{
      newname}): Strings*Agencies +-> POW(Strings
```

```
      )
*L 64 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) => newname:
      Strings
*L 65 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) => ventures: POW
      (Strings*Agencies)
*L 66 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & belongs: dom(
      compRel_4) & works: POW((compRel_4<|
      EngagedIn)((%x.(x: dom(compRel_4) & compRel
      _4(x) = a,b | x,a,b))(belongs))) =>
      empClass_3\/{newEmpId,newname,belongs,works
      }: EmpIds +-> Strings*Strings*POW(Strings*
      Agencies)
*L 67 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & belongs: dom(
      compRel_4) & works: POW((compRel_4<|
      EngagedIn)((%x.(x: dom(compRel_4) & compRel
      _4(x) = a,b | x,a,b))(belongs))) =>
      newEmpId: EmpIds
*L 68 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) => belongs:
      Strings
*L 69 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) => works: POW(
      Strings*Agencies)
*L 70 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
```

```
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & empl: dom(
      empRel_4) & belongs: dom(compRel_4) & works
      : POW((compRel_4<|EngagedIn)((%x.(x: dom(
      compRel_4) & compRel_4(x) = a,b | x,a,b))(
      belongs))) => empClass_3<+{empl|->((
      empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | b))((
      %x.(x: dom(empClass_3) & empClass_3(x) = a,
      b,c | x,a,b,c))(empl)),belongs,works)}:
      EmpIds +-> Strings*Strings*POW(Strings*
      Agencies)
*L 71 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) => empl: EmpIds
*L 72 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) => belongs:
      Strings
*L 73 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) => works: POW(
      Strings*Agencies)
*L 74 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_4) &
      empl: dom(empRel_4) & works: POW((compRel_4
      <|EngagedIn)((%x.(x: dom(compRel_4) &
      compRel_4(x) = a,b | x,a,b))((empRel_4<|
      BelongsTo)((%x.(x: dom(empRel_4) & empRel_4
      (x) = a,b,c | x,a,b,c))(empl))))) =>
      empClass_3<+{empl|->((empClass_3<|%x.(x:
      EmpIds*Strings*Strings*POW(Strings*Agencies
      ) & x = a,b,c,d | b))((%x.(x: dom(empClass_
      3) & empClass_3(x) = a,b,c | x,a,b,c))(empl
      )),(empClass_3<|%x.(x: EmpIds*Strings*
      Strings*POW(Strings*Agencies) & x = a,b,c,d
      | c))((%x.(x: dom(empClass_3) & empClass_3(
      x) = a,b,c | x,a,b,c))(empl)),works)}:
      EmpIds +-> Strings*Strings*POW(Strings*
      Agencies)
```

```
*L 75 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_4) =>
      empl: EmpIds
*L 76 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_4) =>
      works: POW(Strings*Agencies)
```

E.2 Mathematische Lemmas

E.2.1 Mathematische Lemmas der Konsistenznachweise

```
*L 1  ctx(ExtResearchCompaniesMch) & x,y: {} & {}
      (x) = {}(y) & {}(x) = {}(y) => x = y
*L 2  ctx(ExtResearchCompaniesMch) & x,y: {} & {}
      (x) = {}(y) => x = y
*L 3  ctx(ExtResearchCompaniesMch) & x: {} => {}(
      x) incl {}({}(x))
*L 4  ctx(ExtResearchCompaniesMch) & x: {} => {}(
      x) = {y | y: {} & x: {}(y)}
*L 5  ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_1) & x,y:
      projects_1-{proj} & ({proj}<<|projName_1)(x
      ) = ({proj}<<|projName_1)(y) & ({proj}<<|
      getsGrantFrom_1)(x) = ({proj}<<|
      getsGrantFrom_1)(y) => x = y
*L 6  ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_1) & x:
      projects_1-{proj} => ({proj}<<|consortium_1
      )(x) = {y | y: companies_1 & x: engagedIn_1
      (y)}
*L 7  ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_1) & x
      : employees_1-{empl} => ({empl}<<|worksOn_1
      )(x) incl engagedIn_1(({empl}<<|belongsTo_1
```

```
      )(x))
*L 8  ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_1) & co: COMPANIES-
      companies_1 & x,y: companies_1\/{co} & (
      compName_1<+{co|->newname})(x) = (compName_
      1<+{co|->newname})(y) => x = y
*L 9  ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_1) & co: COMPANIES-
      companies_1 & x: employees_1 => worksOn_1(x
      ) incl (engagedIn_1<+{co|->ventures})(
      belongsTo_1(x))
*L 10 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_1) & co: COMPANIES-
      companies_1 & x: projects_1 => (consortium_
      1<+%x.(x: ventures | consortium_1(x)\/{co})
      )(x) = {y | y: companies_1\/{co} & x: (
      engagedIn_1<+{co|->ventures})(y)}
*L 11 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_1) & ee: EMPLOYEES-
      employees_1 & x: employees_1\/{ee} => (
      worksOn_1<+{ee|->works})(x) incl engagedIn_
      1((belongsTo_1<+{ee|->belongs})(x))
*L 12 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_1) & x: employees_1
      => (worksOn_1<+{empl|->works})(x) incl
      engagedIn_1((belongsTo_1<+{empl|->belongs})
      (x))
*L 13 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_1) &
      x: employees_1 => (worksOn_1<+{empl|->works
      })(x) incl engagedIn_1(belongsTo_1(x))
```

E.2.2 Mathematische Lemmas der ersten Verfeinerung

```
*L 14 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_2) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_2) => {
      proj}<<|%x.(x: projects_2 & x = a,b | b) =
      %x.(x: projects_2-{proj} & x = a,b | b)
*L 15 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_2) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_2) => {
      proj}<<|%x.(x: projects_2 & x = a,b | a) =
      %x.(x: projects_2-{proj} & x = a,b | a)
*L 16 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_2) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_2) => %x.(x:
      companies_2 | x)<+{newname|->newname} = %x.
      (x: companies_2\/{newname} | x)
```

E.2.3 Mathematische Lemmas der zweiten Verfeinerung

```
*L 17 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_3) => %x.(
      x: dom(projClass_3)-{proj} | x) = %x.(x:
      dom({proj}<<|projClass_3) | x)
*L 18 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_3) => {
      proj}<<|%x.(x: dom(projClass_3) | projClass
      _3(x)) = %x.(x: dom({proj}<<|projClass_3) |
      ({proj}<<|projClass_3)(x))
*L 19 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_3) => %x.(
      x: dom(projClass_3)-{proj} & x = a,b | b) =
      %x.(x: dom({proj}<<|projClass_3) & x = a,b
      | b)
*L 20 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
```

```
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_3) => %x.(
      x: dom(projClass_3)-{proj} & x = a,b | a) =
      %x.(x: dom({proj}<<|projClass_3) & x = a,b
      | a)
*L 21 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_3) => dom(
      projClass_3)-{proj} = dom({proj}<<|
      projClass_3)
*L 22 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_3) =>
      %x.(x: dom(empClass_3)-{empl} | x) = %x.(x:
      dom({empl}<<|empClass_3) | x)
*L 23 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_3) =>
      {empl}<<|%x.(x: dom(empClass_3) & empClass_
      3(x) = a,b,c | c) = %x.(x: dom({empl}<<|
      empClass_3) & ({empl}<<|empClass_3)(x) = a,
      b,c | c)
*L 24 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_3) =>
      {empl}<<|%x.(x: dom(empClass_3) & empClass_
      3(x) = a,b,c | b) = %x.(x: dom({empl}<<|
      empClass_3) & ({empl}<<|empClass_3)(x) = a,
      b,c | b)
*L 25 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_3) =>
      {empl}<<|%x.(x: dom(empClass_3) & empClass_
      3(x) = a,b,c | a) = %x.(x: dom({empl}<<|
      empClass_3) & ({empl}<<|empClass_3)(x) = a,
      b,c | a)
*L 26 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_3) =>
      dom(empClass_3)-{empl} = dom({empl}<<|
      empClass_3)
*L 27 ctx(ExtResearchCompaniesMch) & inv(
```

```
      ExtResearchCompaniesMch_3) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_3) => %x.(x: dom(
      projClass_3) | x) = %x.(x: dom(projClass_3
      <+%x.(x: ventures | projClass_3(x)\/{
      newname})) | x)
*L 28 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_3) => %x.(x: dom(
      projClass_3) | projClass_3(x))<+%x.(x:
      ventures | {y | y: dom(compClass_3) & x: (%
      x.(x: dom(compClass_3) & compClass_3(x) = a
      ,b | a))(y)}\/{newname}) = %x.(x: dom(
      projClass_3<+%x.(x: ventures | projClass_3(
      x)\/{newname})) | (projClass_3<+%x.(x:
      ventures | projClass_3(x)\/{newname}))(x))
*L 29 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_3) => %x.(x: dom(
      projClass_3) & x = a,b | b) = %x.(x: dom(
      projClass_3<+%x.(x: ventures | projClass_3(
      x)\/{newname})) & x = a,b | b)
*L 30 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_3) => %x.(x: dom(
      projClass_3) & x = a,b | a) = %x.(x: dom(
      projClass_3<+%x.(x: ventures | projClass_3(
      x)\/{newname})) & x = a,b | a)
*L 31 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_3) => dom(projClass
      _3) = dom(projClass_3<+%x.(x: ventures |
      projClass_3(x)\/{newname}))
*L 32 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_3) => %x.(x: dom(
      compClass_3) & compClass_3(x) = a,b | b)<+{
```

```
      newname|->budg} = %x.(x: dom(compClass_3\/{
      newname,ventures,budg}) & (compClass_3\/{
      newname,ventures,budg})(x) = a,b | b)
*L 33 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_3) => %x.(x: dom(
      compClass_3) & compClass_3(x) = a,b | a)<+{
      newname|->ventures} = %x.(x: dom(compClass_
      3\/{newname,ventures,budg}) & (compClass_3
      \/{newname,ventures,budg})(x) = a,b | a)
*L 34 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_3) => %x.(x: dom(
      compClass_3)\/{newname} | x) = %x.(x: dom(
      compClass_3\/{newname,ventures,budg}) | x)
*L 35 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_3) => dom(compClass
      _3)\/{newname} = dom(compClass_3\/{newname,
      ventures,budg})
*L 36 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_3) => %x.(x: dom(
      empClass_3)\/{newEmpId} | x) = %x.(x: dom(
      empClass_3\/{newEmpId,newname,belongs,works
      }) | x)
*L 37 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_3) => %x.(x: dom(
      empClass_3) & empClass_3(x) = a,b,c | c)<+{
      newEmpId|->works} = %x.(x: dom(empClass_3\/
      {newEmpId,newname,belongs,works}) & (
      empClass_3\/{newEmpId,newname,belongs,works
      })(x) = a,b,c | c)
*L 38 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
```

```
      ExtResearchCompaniesMch_3) => %x.(x: dom(
      empClass_3) & empClass_3(x) = a,b,c | b)<+{
      newEmpId|->belongs} = %x.(x: dom(empClass_3
      \/{newEmpId,newname,belongs,works}) & (
      empClass_3\/{newEmpId,newname,belongs,works
      })(x) = a,b,c | b)
*L 39 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_3) => %x.(x: dom(
      empClass_3) & empClass_3(x) = a,b,c | a)<+{
      newEmpId|->newname} = %x.(x: dom(empClass_3
      \/{newEmpId,newname,belongs,works}) & (
      empClass_3\/{newEmpId,newname,belongs,works
      })(x) = a,b,c | a)
*L 40 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_3) => dom(empClass_
      3)\/{newEmpId} = dom(empClass_3\/{newEmpId,
      newname,belongs,works})
*L 41 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_3) => %x.(x: dom(
      empClass_3) | x) = %x.(x: dom(empClass_3<+{
      empl|->((%x.(x: dom(empClass_3) & empClass_
      3(x) = a,b,c | a))(empl),belongs,works)}) |
      x)
*L 42 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_3) => %x.(x: dom(
      empClass_3) & empClass_3(x) = a,b,c | c)<+{
      empl|->works} = %x.(x: dom(empClass_3<+{
      empl|->((%x.(x: dom(empClass_3) & empClass_
      3(x) = a,b,c | a))(empl),belongs,works)}) &
      (empClass_3<+{empl|->((%x.(x: dom(empClass_
      3) & empClass_3(x) = a,b,c | a))(empl),
      belongs,works)})(x) = a,b,c | c)
*L 43 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
```

```
      ExtResearchCompaniesMch_3) => %x.(x: dom(
      empClass_3) & empClass_3(x) = a,b,c | b)<+{
      empl|->belongs} = %x.(x: dom(empClass_3<+{
      empl|->((%x.(x: dom(empClass_3) & empClass_
      3(x) = a,b,c | a))(empl),belongs,works)}) &
      (empClass_3<+{empl|->((%x.(x: dom(empClass_
      3) & empClass_3(x) = a,b,c | a))(empl),
      belongs,works)})(x) = a,b,c | b)
*L 44 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_3) => %x.(x: dom(
      empClass_3) & empClass_3(x) = a,b,c | a) =
      %x.(x: dom(empClass_3<+{empl|->((%x.(x: dom
      (empClass_3) & empClass_3(x) = a,b,c | a))(
      empl),belongs,works)}) & (empClass_3<+{empl
      |->((%x.(x: dom(empClass_3) & empClass_3(x)
      = a,b,c | a))(empl),belongs,works)})(x) = a
      ,b,c | a)
*L 45 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_3) => dom(empClass_
      3) = dom(empClass_3<+{empl|->((%x.(x: dom(
      empClass_3) & empClass_3(x) = a,b,c | a))(
      empl),belongs,works)})
*L 46 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_3) =>
      %x.(x: dom(empClass_3) | x) = %x.(x: dom(
      empClass_3<+{empl|->((%x.(x: dom(empClass_3
      ) & empClass_3(x) = a,b,c | a))(empl),(%x.(
      x: dom(empClass_3) & empClass_3(x) = a,b,c
      | b))(empl),works)}) | x)
*L 47 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_3) =>
      %x.(x: dom(empClass_3) & empClass_3(x) = a,
      b,c | c)<+{empl|->works} = %x.(x: dom(
      empClass_3<+{empl|->((%x.(x: dom(empClass_3
      ) & empClass_3(x) = a,b,c | a))(empl),(%x.(
      x: dom(empClass_3) & empClass_3(x) = a,b,c
      | b))(empl),works)}) & (empClass_3<+{empl
      |->((%x.(x: dom(empClass_3) & empClass_3(x)
```

```
      = a,b,c | a))(empl),(%x.(x: dom(empClass_3)
      & empClass_3(x) = a,b,c | b))(empl),works)}
      )(x) = a,b,c | c)
*L 48 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_3) =>
      %x.(x: dom(empClass_3) & empClass_3(x) = a,
      b,c | b) = %x.(x: dom(empClass_3<+{empl|->(
      (%x.(x: dom(empClass_3) & empClass_3(x) = a
      ,b,c | a))(empl),(%x.(x: dom(empClass_3) &
      empClass_3(x) = a,b,c | b))(empl),works)})
      & (empClass_3<+{empl|->((%x.(x: dom(
      empClass_3) & empClass_3(x) = a,b,c | a))(
      empl),(%x.(x: dom(empClass_3) & empClass_3(
      x) = a,b,c | b))(empl),works)})(x) = a,b,c
      | b)
*L 49 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_3) =>
      %x.(x: dom(empClass_3) & empClass_3(x) = a,
      b,c | a) = %x.(x: dom(empClass_3<+{empl|->(
      (%x.(x: dom(empClass_3) & empClass_3(x) = a
      ,b,c | a))(empl),(%x.(x: dom(empClass_3) &
      empClass_3(x) = a,b,c | b))(empl),works)})
      & (empClass_3<+{empl|->((%x.(x: dom(
      empClass_3) & empClass_3(x) = a,b,c | a))(
      empl),(%x.(x: dom(empClass_3) & empClass_3(
      x) = a,b,c | b))(empl),works)})(x) = a,b,c
      | a)
*L 50 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_3) =>
      dom(empClass_3) = dom(empClass_3<+{empl|->(
      (%x.(x: dom(empClass_3) & empClass_3(x) = a
      ,b,c | a))(empl),(%x.(x: dom(empClass_3) &
      empClass_3(x) = a,b,c | b))(empl),works)})
```

E.2.4 Mathematische Lemmas der dritten Verfeinerung

```
*L 51 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_4) & proj:
```

```
      dom(projRel_4) & (projRel_4<|Consortium)((%
      x.(x: dom(projRel_4) & projRel_4(x) = a | x
      ,a))(proj)) = {} => %x.(x: dom({proj}<<|
      projClass_3) | x) = {proj}<<|projClass_3<|%
      x.(x: Strings*Agencies*POW(Strings) & x = a
      ,b | a)
*L 52 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_4) & proj:
      dom(projRel_4) & (projRel_4<|Consortium)((%
      x.(x: dom(projRel_4) & projRel_4(x) = a | x
      ,a))(proj)) = {} => %x.(x: dom({proj}<<|
      projClass_3) | ({proj}<<|projClass_3)(x)) =
      {proj}<<|projClass_3<|%x.(x: Strings*
      Agencies*POW(Strings) & x = a,b | b)
*L 53 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_4) & proj:
      dom(projRel_4) & (projRel_4<|Consortium)((%
      x.(x: dom(projRel_4) & projRel_4(x) = a | x
      ,a))(proj)) = {} => %x.(x: dom({proj}<<|
      projClass_3) & x = a,b | b) = dom({proj}<<|
      projClass_3)<|%x.(x: Strings*Agencies & x =
      a,b | b)
*L 54 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_4) & proj:
      dom(projRel_4) & (projRel_4<|Consortium)((%
      x.(x: dom(projRel_4) & projRel_4(x) = a | x
      ,a))(proj)) = {} => %x.(x: dom({proj}<<|
      projClass_3) & x = a,b | a) = dom({proj}<<|
      projClass_3)<|%x.(x: Strings*Agencies & x =
      a,b | a)
*L 55 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_4) & not(
      proj: dom(projRel_4) & (projRel_4<|
      Consortium)((%x.(x: dom(projRel_4) &
      projRel_4(x) = a | x,a))(proj)) = {}) =>
      projClass_3 = {proj}<<|projClass_3
*L 56 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_4) & not(
```

```
      proj: dom(projRel_4) & (projRel_4<|
      Consortium)((%x.(x: dom(projRel_4) &
      projRel_4(x) = a | x,a))(proj)) = {}) => %x
      .(x: dom({proj}<<|projClass_3) | x) =
      projId_4
*L 57 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_4) & not(
      proj: dom(projRel_4) & (projRel_4<|
      Consortium)((%x.(x: dom(projRel_4) &
      projRel_4(x) = a | x,a))(proj)) = {}) => %x
      .(x: dom({proj}<<|projClass_3) | ({proj}<<|
      projClass_3)(x)) = consortium_4
*L 58 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_4) & not(
      proj: dom(projRel_4) & (projRel_4<|
      Consortium)((%x.(x: dom(projRel_4) &
      projRel_4(x) = a | x,a))(proj)) = {}) => %x
      .(x: dom({proj}<<|projClass_3) & x = a,b |
      b) = getsGrantFrom_4
*L 59 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_4) & not(
      proj: dom(projRel_4) & (projRel_4<|
      Consortium)((%x.(x: dom(projRel_4) &
      projRel_4(x) = a | x,a))(proj)) = {}) => %x
      .(x: dom({proj}<<|projClass_3) & x = a,b |
      a) = projName_4


*L 60 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_4) & not(
      proj: dom(projRel_4) & (projRel_4<|
      Consortium)((%x.(x: dom(projRel_4) &
      projRel_4(x) = a | x,a))(proj)) = {}) =>
      dom({proj}<<|projClass_3) = projects_4
*L 61 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_4) &
      empl: dom(empRel_4) => %x.(x: dom({empl}<<|
      empClass_3) | x) = {empl}<<|empClass_3<|%x.
```

```
      (x: EmpIds*Strings*Strings*POW(Strings*
      Agencies) & x = a,b,c,d | a)
*L 62 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_4) &
      empl: dom(empRel_4) => %x.(x: dom({empl}<<|
      empClass_3) & ({empl}<<|empClass_3)(x) = a,
      b,c | c) = {empl}<<|empClass_3<|%x.(x:
      EmpIds*Strings*Strings*POW(Strings*Agencies
      ) & x = a,b,c,d | d)
*L 63 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_4) &
      empl: dom(empRel_4) => %x.(x: dom({empl}<<|
      empClass_3) & ({empl}<<|empClass_3)(x) = a,
      b,c | b) = {empl}<<|empClass_3<|%x.(x:
      EmpIds*Strings*Strings*POW(Strings*Agencies
      ) & x = a,b,c,d | c)
*L 64 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_4) &
      empl: dom(empRel_4) => %x.(x: dom({empl}<<|
      empClass_3) & ({empl}<<|empClass_3)(x) = a,
      b,c | a) = {empl}<<|empClass_3<|%x.(x:
      EmpIds*Strings*Strings*POW(Strings*Agencies
      ) & x = a,b,c,d | b)
*L 65 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_4) &
      not(empl: dom(empRel_4)) => empClass_3 = {
      empl}<<|empClass_3
*L 66 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_4) &
      not(empl: dom(empRel_4)) => %x.(x: dom({
      empl}<<|empClass_3) | x) = empId_4
*L 67 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_4) &
      not(empl: dom(empRel_4)) => %x.(x: dom({
      empl}<<|empClass_3) & ({empl}<<|empClass_3)
      (x) = a,b,c | c) = worksOn_4
```

```
*L 68 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_4) &
      not(empl: dom(empRel_4)) => %x.(x: dom({
      empl}<<|empClass_3) & ({empl}<<|empClass_3)
      (x) = a,b,c | b) = belongsTo_4
*L 69 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_4) &
      not(empl: dom(empRel_4)) => %x.(x: dom({
      empl}<<|empClass_3) & ({empl}<<|empClass_3)
      (x) = a,b,c | a) = empName_4
*L 70 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_4) &
      not(empl: dom(empRel_4)) => dom({empl}<<|
      empClass_3) = employees_4
*L 71 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) & newname:
      CompNames-ran(compRel_4<|CompName) &
      ventures: POW(dom(projRel_4)) => %x.(x: dom
      (projClass_3<+%x.(x: ventures | projClass_3
      (x)\/{newname})) | x) = projClass_3<+%x.(x:
      ventures | projClass_3(x)\/{newname})<|%x.(
      x: Strings*Agencies*POW(Strings) & x = a,b
      | a)
*L 72 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) & newname:
      CompNames-ran(compRel_4<|CompName) &
      ventures: POW(dom(projRel_4)) => %x.(x: dom
      (projClass_3<+%x.(x: ventures | projClass_3
      (x)\/{newname})) | (projClass_3<+%x.(x:
      ventures | projClass_3(x)\/{newname}))(x))
      = projClass_3<+%x.(x: ventures | projClass_
      3(x)\/{newname})<|%x.(x: Strings*Agencies*
      POW(Strings) & x = a,b | b)
*L 73 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
```

```
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) & newname:
      CompNames-ran(compRel_4<|CompName) &
      ventures: POW(dom(projRel_4)) => %x.(x: dom
      (projClass_3<+%x.(x: ventures | projClass_3
      (x)\/{newname})) & x = a,b | b) = dom(
      projClass_3<+%x.(x: ventures | projClass_3(
      x)\/{newname}))<|%x.(x: Strings*Agencies &
      x = a,b | b)
*L 74 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) & newname:
      CompNames-ran(compRel_4<|CompName) &
      ventures: POW(dom(projRel_4)) => %x.(x: dom
      (projClass_3<+%x.(x: ventures | projClass_3
      (x)\/{newname})) & x = a,b | a) = dom(
      projClass_3<+%x.(x: ventures | projClass_3(
      x)\/{newname}))<|%x.(x: Strings*Agencies &
      x = a,b | a)
*L 75 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) & newname:
      CompNames-ran(compRel_4<|CompName) &
      ventures: POW(dom(projRel_4)) => %x.(x: dom
      (compClass_3\/{newname,ventures,budg}) & (
      compClass_3\/{newname,ventures,budg})(x) =
      a,b | b) = compClass_3\/{newname,ventures,
      budg}<|%x.(x: Strings*POW(Strings*Agencies)
      *MONEY & x = a,b,c | c)
*L 76 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) & newname:
      CompNames-ran(compRel_4<|CompName) &
      ventures: POW(dom(projRel_4)) => %x.(x: dom
      (compClass_3\/{newname,ventures,budg}) & (
      compClass_3\/{newname,ventures,budg})(x) =
      a,b | a) = compClass_3\/{newname,ventures,
      budg}<|%x.(x: Strings*POW(Strings*Agencies)
      *MONEY & x = a,b,c | b)
*L 77 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
```

```
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) & newname:
      CompNames-ran(compRel_4<|CompName) &
      ventures: POW(dom(projRel_4)) => %x.(x: dom
      (compClass_3\/{newname,ventures,budg}) | x)
      = compClass_3\/{newname,ventures,budg}<|%x.
      (x: Strings*POW(Strings*Agencies)*MONEY & x
      = a,b,c | a)
*L 78 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) & not(newname:
      CompNames-ran(compRel_4<|CompName) &
      ventures: POW(dom(projRel_4))) => compClass
      _3 = compClass_3\/{newname,ventures,budg}
*L 79 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) & not(newname:
      CompNames-ran(compRel_4<|CompName) &
      ventures: POW(dom(projRel_4))) => projClass
      _3 = projClass_3<+%x.(x: ventures |
      projClass_3(x)\/{newname})
*L 80 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) & not(newname:
      CompNames-ran(compRel_4<|CompName) &
      ventures: POW(dom(projRel_4))) => %x.(x:
      dom(projClass_3<+%x.(x: ventures |
      projClass_3(x)\/{newname})) | x) = projId_4
*L 81 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) & not(newname:
      CompNames-ran(compRel_4<|CompName) &
      ventures: POW(dom(projRel_4))) => %x.(x:
      dom(projClass_3<+%x.(x: ventures |
      projClass_3(x)\/{newname})) | (projClass_3
      <+%x.(x: ventures | projClass_3(x)\/{
      newname}))(x)) = consortium_4
*L 82 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
```

```
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) & not(newname:
      CompNames-ran(compRel_4<|CompName) &
      ventures: POW(dom(projRel_4))) => %x.(x:
      dom(projClass_3<+%x.(x: ventures |
      projClass_3(x)\/{newname})) & x = a,b | b)
      = getsGrantFrom_4
*L 83 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) & not(newname:
      CompNames-ran(compRel_4<|CompName) &
      ventures: POW(dom(projRel_4))) => %x.(x:
      dom(projClass_3<+%x.(x: ventures |
      projClass_3(x)\/{newname})) & x = a,b | a)
      = projName_4
*L 84 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) & not(newname:
      CompNames-ran(compRel_4<|CompName) &
      ventures: POW(dom(projRel_4))) => dom(
      projClass_3<+%x.(x: ventures | projClass_3(
      x)\/{newname})) = projects_4
*L 85 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) & not(newname:
      CompNames-ran(compRel_4<|CompName) &
      ventures: POW(dom(projRel_4))) => %x.(x:
      dom(compClass_3\/{newname,ventures,budg}) &
      (compClass_3\/{newname,ventures,budg})(x) =
      a,b | b) = budget_4
*L 86 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) & not(newname:
      CompNames-ran(compRel_4<|CompName) &
      ventures: POW(dom(projRel_4))) => %x.(x:
      dom(compClass_3\/{newname,ventures,budg}) &
      (compClass_3\/{newname,ventures,budg})(x) =
      a,b | a) = engagedIn_4
*L 87 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
```

```
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) & not(newname:
      CompNames-ran(compRel_4<|CompName) &
      ventures: POW(dom(projRel_4))) => %x.(x:
      dom(compClass_3\/{newname,ventures,budg}) |
      x) = compName_4
*L 88 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) & not(newname:
      CompNames-ran(compRel_4<|CompName) &
      ventures: POW(dom(projRel_4))) => dom(
      compClass_3\/{newname,ventures,budg}) =
      companies_4
*L 89 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_4) & not(newname:
      CompNames-ran(compRel_4<|CompName) &
      ventures: POW(dom(projRel_4))) =>
      nilCompName = newname
*L 90 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & belongs: dom(
      compRel_4) & works: POW((compRel_4<|
      EngagedIn)((%x.(x: dom(compRel_4) & compRel
      _4(x) = a,b | x,a,b))(belongs))) => %x.(x:
      dom(empClass_3\/{newEmpId,newname,belongs,
      works}) | x) = empClass_3\/{newEmpId,
      newname,belongs,works}<|%x.(x: EmpIds*
      Strings*Strings*POW(Strings*Agencies) & x =
      a,b,c,d | a)
*L 91 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & belongs: dom(
      compRel_4) & works: POW((compRel_4<|
      EngagedIn)((%x.(x: dom(compRel_4) & compRel
      _4(x) = a,b | x,a,b))(belongs))) => %x.(x:
      dom(empClass_3\/{newEmpId,newname,belongs,
      works}) & (empClass_3\/{newEmpId,newname,
      belongs,works})(x) = a,b,c | c) = empClass_
```

```
      3\/{newEmpId,newname,belongs,works}<|%x.(x:
      EmpIds*Strings*Strings*POW(Strings*Agencies
      ) & x = a,b,c,d | d)
*L 92 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & belongs: dom(
      compRel_4) & works: POW((compRel_4<|
      EngagedIn))((%x.(x: dom(compRel_4) & compRel
      _4(x) = a,b | x,a,b))(belongs))) => %x.(x:
      dom(empClass_3\/{newEmpId,newname,belongs,
      works}) & (empClass_3\/{newEmpId,newname,
      belongs,works})(x) = a,b,c | b) = empClass_
      3\/{newEmpId,newname,belongs,works}<|%x.(x:
      EmpIds*Strings*Strings*POW(Strings*Agencies
      ) & x = a,b,c,d | c)
*L 93 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & belongs: dom(
      compRel_4) & works: POW((compRel_4<|
      EngagedIn))((%x.(x: dom(compRel_4) & compRel
      _4(x) = a,b | x,a,b))(belongs))) => %x.(x:
      dom(empClass_3\/{newEmpId,newname,belongs,
      works}) & (empClass_3\/{newEmpId,newname,
      belongs,works})(x) = a,b,c | a) = empClass_
      3\/{newEmpId,newname,belongs,works}<|%x.(x:
      EmpIds*Strings*Strings*POW(Strings*Agencies
      ) & x = a,b,c,d | b)
*L 94 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & not(belongs:
      dom(compRel_4) & works: POW((compRel_4<|
      EngagedIn))((%x.(x: dom(compRel_4) & compRel
      _4(x) = a,b | x,a,b))(belongs)))) =>
      empClass_3 = empClass_3\/{newEmpId,newname,
      belongs,works}
*L 95 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & not(belongs:
      dom(compRel_4) & works: POW((compRel_4<|
      EngagedIn))((%x.(x: dom(compRel_4) & compRel
```

```
      _4(x) = a,b | x,a,b))(belongs)))) =>
      newEmpId = tEmpId_4
*L 96 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & not(belongs:
      dom(compRel_4) & works: POW((compRel_4<|
      EngagedIn)((%x.(x: dom(compRel_4) & compRel
      _4(x) = a,b | x,a,b))(belongs)))) => %x.(x:
      dom(empClass_3\/{newEmpId,newname,belongs,
      works}) | x) = empId_4
*L 97 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & not(belongs:
      dom(compRel_4) & works: POW((compRel_4<|
      EngagedIn)((%x.(x: dom(compRel_4) & compRel
      _4(x) = a,b | x,a,b))(belongs)))) => %x.(x:
      dom(empClass_3\/{newEmpId,newname,belongs,
      works}) & (empClass_3\/{newEmpId,newname,
      belongs,works})(x) = a,b,c | c) = worksOn_4
*L 98 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & not(belongs:
      dom(compRel_4) & works: POW((compRel_4<|
      EngagedIn)((%x.(x: dom(compRel_4) & compRel
      _4(x) = a,b | x,a,b))(belongs)))) => %x.(x:
      dom(empClass_3\/{newEmpId,newname,belongs,
      works}) & (empClass_3\/{newEmpId,newname,
      belongs,works})(x) = a,b,c | b) = belongsTo
      _4
*L 99 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & not(belongs:
      dom(compRel_4) & works: POW((compRel_4<|
      EngagedIn)((%x.(x: dom(compRel_4) & compRel
      _4(x) = a,b | x,a,b))(belongs)))) => %x.(x:
      dom(empClass_3\/{newEmpId,newname,belongs,
      works}) & (empClass_3\/{newEmpId,newname,
      belongs,works})(x) = a,b,c | a) = empName_4
```

```
*L 100 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & not(belongs:
      dom(compRel_4) & works: POW((compRel_4<|
      EngagedIn)((%x.(x: dom(compRel_4) & compRel
      _4(x) = a,b | x,a,b))(belongs)))) => dom(
      empClass_3\/{newEmpId,newname,belongs,works
      }) = employees_4
*L 101 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & not(belongs:
      dom(compRel_4) & works: POW((compRel_4<|
      EngagedIn)((%x.(x: dom(compRel_4) & compRel
      _4(x) = a,b | x,a,b))(belongs)))) =>
      nilEmpId = newEmpId
*L 102 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & empl: dom(
      empRel_4) & belongs: dom(compRel_4) & works
      : POW((compRel_4<|EngagedIn)((%x.(x: dom(
      compRel_4) & compRel_4(x) = a,b | x,a,b))(
      belongs))) => empClass_3<+{empl|->((
      empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | b))((
      %x.(x: dom(empClass_3) & empClass_3(x) = a,
      b,c | x,a,b,c))(empl)),belongs,works)} =
      empClass_3<+{empl|->((empClass_3<|%x.(x:
      EmpIds*Strings*Strings*POW(Strings*Agencies
      ) & x = a,b,c,d | b))(empl),belongs,works)}
*L 103 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & empl: dom(
      empRel_4) & belongs: dom(compRel_4) & works
      : POW((compRel_4<|EngagedIn)((%x.(x: dom(
      compRel_4) & compRel_4(x) = a,b | x,a,b))(
      belongs))) => %x.(x: dom(empClass_3<+{empl
      |->((empClass_3<|%x.(x: EmpIds*Strings*
      Strings*POW(Strings*Agencies) & x = a,b,c,d
      | b))(empl),belongs,works)}) | x) =
      empClass_3<+{empl|->((empClass_3<|%x.(x:
```

```
      EmpIds*Strings*Strings*POW(Strings*Agencies
      ) & x = a,b,c,d | b))((%x.(x: dom(empClass_
      3) & empClass_3(x) = a,b,c | x,a,b,c))(empl
      )),belongs,works)}<|%x.(x: EmpIds*Strings*
      Strings*POW(Strings*Agencies) & x = a,b,c,d
      | a)
*L 104 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & empl: dom(
      empRel_4) & belongs: dom(compRel_4) & works
      : POW((compRel_4<|EngagedIn)((%x.(x: dom(
      compRel_4) & compRel_4(x) = a,b | x,a,b))(
      belongs))) => %x.(x: dom(empClass_3<+{empl
      |->((empClass_3<|%x.(x: EmpIds*Strings*
      Strings*POW(Strings*Agencies) & x = a,b,c,d
      | b))(empl),belongs,works)}) & (empClass_3
      <+{empl|->((empClass_3<|%x.(x: EmpIds*
      Strings*Strings*POW(Strings*Agencies) & x =
      a,b,c,d | b))(empl),belongs,works)})(x) = a
      ,b,c | c) = empClass_3<+{empl|->((empClass_
      3<|%x.(x: EmpIds*Strings*Strings*POW(
      Strings*Agencies) & x = a,b,c,d | b))((%x.(
      x: dom(empClass_3) & empClass_3(x) = a,b,c
      | x,a,b,c))(empl)),belongs,works)}<|%x.(x:
      EmpIds*Strings*Strings*POW(Strings*Agencies
      ) & x = a,b,c,d | d)


*L 105 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & empl: dom(
      empRel_4) & belongs: dom(compRel_4) & works
      : POW((compRel_4<|EngagedIn)((%x.(x: dom(
      compRel_4) & compRel_4(x) = a,b | x,a,b))(
      belongs))) => %x.(x: dom(empClass_3<+{empl
      |->((empClass_3<|%x.(x: EmpIds*Strings*
      Strings*POW(Strings*Agencies) & x = a,b,c,d
      | b))(empl),belongs,works)}) & (empClass_3
      <+{empl|->((empClass_3<|%x.(x: EmpIds*
      Strings*Strings*POW(Strings*Agencies) & x =
      a,b,c,d | b))(empl),belongs,works)})(x) = a
      ,b,c | b) = empClass_3<+{empl|->((empClass_
      3<|%x.(x: EmpIds*Strings*Strings*POW(
      Strings*Agencies) & x = a,b,c,d | b))((%x.(
```

```
      x: dom(empClass_3) & empClass_3(x) = a,b,c
      | x,a,b,c))(empl)),belongs,works)}<|%x.(x:
      EmpIds*Strings*Strings*POW(Strings*Agencies
      ) & x = a,b,c,d | c)
*L 106 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & empl: dom(
      empRel_4) & belongs: dom(compRel_4) & works
      : POW((compRel_4<|EngagedIn)((%x.(x: dom(
      compRel_4) & compRel_4(x) = a,b | x,a,b))(
      belongs))) => %x.(x: dom(empClass_3<+{empl
      |->((empClass_3<|%x.(x: EmpIds*Strings*
      Strings*POW(Strings*Agencies) & x = a,b,c,d
      | b))(empl),belongs,works)}) & (empClass_3
      <+{empl|->((empClass_3<|%x.(x: EmpIds*
      Strings*Strings*POW(Strings*Agencies) & x =
      a,b,c,d | b))(empl),belongs,works)})(x) = a
      ,b,c | a) = empClass_3<+{empl|->((empClass_
      3<|%x.(x: EmpIds*Strings*Strings*POW(
      Strings*Agencies) & x = a,b,c,d | b))((%x.(
      x: dom(empClass_3) & empClass_3(x) = a,b,c
      | x,a,b,c))(empl)),belongs,works)}<|%x.(x:
      EmpIds*Strings*Strings*POW(Strings*Agencies
      ) & x = a,b,c,d | b)
*L 107 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & empl: dom(
      empRel_4) & belongs: dom(compRel_4) & works
      : POW((compRel_4<|EngagedIn)((%x.(x: dom(
      compRel_4) & compRel_4(x) = a,b | x,a,b))(
      belongs))) => dom(empClass_3<+{empl|->((
      empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | b))(
      empl),belongs,works)}) = dom(empClass_3<+{
      empl|->((empClass_3<|%x.(x: EmpIds*Strings*
      Strings*POW(Strings*Agencies) & x = a,b,c,d
      | b))((%x.(x: dom(empClass_3) & empClass_3(
      x) = a,b,c | x,a,b,c))(empl)),belongs,works
      )})
*L 108 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & not(empl: dom(
```

```
      empRel_4) & belongs: dom(compRel_4) & works
      : POW((compRel_4<|EngagedIn)((%x.(x: dom(
      compRel_4) & compRel_4(x) = a,b | x,a,b))(
      belongs)))) => empClass_3 = empClass_3<+{
      empl|->((empClass_3<|%x.(x: EmpIds*Strings*
      Strings*POW(Strings*Agencies) & x = a,b,c,d
      | b))(empl),belongs,works)}
*L 109 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & not(empl: dom(
      empRel_4) & belongs: dom(compRel_4) & works
      : POW((compRel_4<|EngagedIn)((%x.(x: dom(
      compRel_4) & compRel_4(x) = a,b | x,a,b))(
      belongs)))) => %x.(x: dom(empClass_3<+{empl
      |->((empClass_3<|%x.(x: EmpIds*Strings*
      Strings*POW(Strings*Agencies) & x = a,b,c,d
      | b))(empl),belongs,works)}) | x) =
      empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | a)
*L 110 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & not(empl: dom(
      empRel_4) & belongs: dom(compRel_4) & works
      : POW((compRel_4<|EngagedIn)((%x.(x: dom(
      compRel_4) & compRel_4(x) = a,b | x,a,b))(
      belongs)))) => %x.(x: dom(empClass_3<+{empl
      |->((empClass_3<|%x.(x: EmpIds*Strings*
      Strings*POW(Strings*Agencies) & x = a,b,c,d
      | b))(empl),belongs,works)}) & (empClass_3
      <+{empl|->((empClass_3<|%x.(x: EmpIds*
      Strings*Strings*POW(Strings*Agencies) & x =
      a,b,c,d | b))(empl),belongs,works)})(x) = a
      ,b,c | c) = empClass_3<|%x.(x: EmpIds*
      Strings*Strings*POW(Strings*Agencies) & x =
      a,b,c,d | d)
*L 111 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & not(empl: dom(
      empRel_4) & belongs: dom(compRel_4) & works
      : POW((compRel_4<|EngagedIn)((%x.(x: dom(
      compRel_4) & compRel_4(x) = a,b | x,a,b))(
      belongs)))) => %x.(x: dom(empClass_3<+{empl
```

```
      |->((empClass_3<|%x.(x: EmpIds*Strings*
      Strings*POW(Strings*Agencies) & x = a,b,c,d
      | b))(empl),belongs,works)}) & (empClass_3
      <+{empl|->((empClass_3<|%x.(x: EmpIds*
      Strings*Strings*POW(Strings*Agencies) & x =
      a,b,c,d | b))(empl),belongs,works)})(x) = a
      ,b,c | b) = empClass_3<|%x.(x: EmpIds*
      Strings*Strings*POW(Strings*Agencies) & x =
      a,b,c,d | c)
*L 112 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & not(empl: dom(
      empRel_4) & belongs: dom(compRel_4) & works
      : POW((compRel_4<|EngagedIn)((%x.(x: dom(
      compRel_4) & compRel_4(x) = a,b | x,a,b))(
      belongs)))) => %x.(x: dom(empClass_3<+{empl
      |->((empClass_3<|%x.(x: EmpIds*Strings*
      Strings*POW(Strings*Agencies) & x = a,b,c,d
      | b))(empl),belongs,works)}) & (empClass_3
      <+{empl|->((empClass_3<|%x.(x: EmpIds*
      Strings*Strings*POW(Strings*Agencies) & x =
      a,b,c,d | b))(empl),belongs,works)})(x) = a
      ,b,c | a) = empClass_3<|%x.(x: EmpIds*
      Strings*Strings*POW(Strings*Agencies) & x =
      a,b,c,d | b)
*L 113 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_4) & not(empl: dom(
      empRel_4) & belongs: dom(compRel_4) & works
      : POW((compRel_4<|EngagedIn)((%x.(x: dom(
      compRel_4) & compRel_4(x) = a,b | x,a,b))(
      belongs)))) => dom(empClass_3<+{empl|->((
      empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | b))(
      empl),belongs,works)}) = dom(empClass_3)
*L 114 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_4) &
      empl: dom(empRel_4) & works: POW((compRel_4
      <|EngagedIn)((%x.(x: dom(compRel_4) &
      compRel_4(x) = a,b | x,a,b))((empRel_4<|
      BelongsTo)((%x.(x: dom(empRel_4) & empRel_4
      (x) = a,b,c | x,a,b,c))(empl))))) =>
```

```
      empClass_3<+{empl|->((empClass_3<|%x.(x:
      EmpIds*Strings*Strings*POW(Strings*Agencies
      ) & x = a,b,c,d | b))((%x.(x: dom(empClass_
      3) & empClass_3(x) = a,b,c | x,a,b,c))(empl
      )),(empClass_3<|%x.(x: EmpIds*Strings*
      Strings*POW(Strings*Agencies) & x = a,b,c,d
      | c))((%x.(x: dom(empClass_3) & empClass_3(
      x) = a,b,c | x,a,b,c))(empl)),works)} =
      empClass_3<+{empl|->((empClass_3<|%x.(x:
      EmpIds*Strings*Strings*POW(Strings*Agencies
      ) & x = a,b,c,d | b))(empl),(empClass_3<|%x
      .(x: EmpIds*Strings*Strings*POW(Strings*
      Agencies) & x = a,b,c,d | c))(empl),works)}
*L 115 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_4) &
      empl: dom(empRel_4) & works: POW((compRel_4
      <|EngagedIn)((%x.(x: dom(compRel_4) &
      compRel_4(x) = a,b | x,a,b))((empRel_4<|
      BelongsTo)((%x.(x: dom(empRel_4) & empRel_4
      (x) = a,b,c | x,a,b,c))(empl))))) => %x.(x:
      dom(empClass_3<+{empl|->((empClass_3<|%x.(x
      : EmpIds*Strings*Strings*POW(Strings*
      Agencies) & x = a,b,c,d | b))(empl),(
      empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | c))(
      empl),works)}) | x) = empClass_3<+{empl|->(
      (empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | b))((
      %x.(x: dom(empClass_3) & empClass_3(x) = a,
      b,c | x,a,b,c))(empl)),(empClass_3<|%x.(x:
      EmpIds*Strings*Strings*POW(Strings*Agencies
      ) & x = a,b,c,d | c))((%x.(x: dom(empClass_
      3) & empClass_3(x) = a,b,c | x,a,b,c))(empl
      )),works)}<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | a)
*L 116 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_4) &
      empl: dom(empRel_4) & works: POW((compRel_4
      <|EngagedIn)((%x.(x: dom(compRel_4) &
      compRel_4(x) = a,b | x,a,b))((empRel_4<|
      BelongsTo)((%x.(x: dom(empRel_4) & empRel_4
      (x) = a,b,c | x,a,b,c))(empl))))) => %x.(x:
      dom(empClass_3<+{empl|->((empClass_3<|%x.(x
      : EmpIds*Strings*Strings*POW(Strings*
```

```
      Agencies) & x = a,b,c,d | b))(empl),(
      empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | c))(
      empl),works)}) & (empClass_3<+{empl|->((
      empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | b))(
      empl),(empClass_3<|%x.(x: EmpIds*Strings*
      Strings*POW(Strings*Agencies) & x = a,b,c,d
      | c))(empl),works)})(x) = a,b,c | c) =
      empClass_3<+{empl|->((empClass_3<|%x.(x:
      EmpIds*Strings*Strings*POW(Strings*Agencies
      ) & x = a,b,c,d | b))((%x.(x: dom(empClass_
      3) & empClass_3(x) = a,b,c | x,a,b,c))(empl
      )),(empClass_3<|%x.(x: EmpIds*Strings*
      Strings*POW(Strings*Agencies) & x = a,b,c,d
      | c))((%x.(x: dom(empClass_3) & empClass_3(
      x) = a,b,c | x,a,b,c))(empl)),works)}<|%x.(
      x: EmpIds*Strings*Strings*POW(Strings*
      Agencies) & x = a,b,c,d | d)
*L 117 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_4) &
      empl: dom(empRel_4) & works: POW((compRel_4
      <|EngagedIn)((%x.(x: dom(compRel_4) &
      compRel_4(x) = a,b | x,a,b))((empRel_4<|
      BelongsTo)((%x.(x: dom(empRel_4) & empRel_4
      (x) = a,b,c | x,a,b,c))(empl))))) => %x.(x:
      dom(empClass_3<+{empl|->((empClass_3<|%x.(x
      : EmpIds*Strings*Strings*POW(Strings*
      Agencies) & x = a,b,c,d | b))(empl),(
      empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | c))(
      empl),works)}) & (empClass_3<+{empl|->((
      empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | b))(
      empl),(empClass_3<|%x.(x: EmpIds*Strings*
      Strings*POW(Strings*Agencies) & x = a,b,c,d
      | c))(empl),works)})(x) = a,b,c | b) =
      empClass_3<+{empl|->((empClass_3<|%x.(x:
      EmpIds*Strings*Strings*POW(Strings*Agencies
      ) & x = a,b,c,d | b))((%x.(x: dom(empClass_
      3) & empClass_3(x) = a,b,c | x,a,b,c))(empl
      )),(empClass_3<|%x.(x: EmpIds*Strings*
      Strings*POW(Strings*Agencies) & x = a,b,c,d
      | c))((%x.(x: dom(empClass_3) & empClass_3(
      x) = a,b,c | x,a,b,c))(empl)),works)}<|%x.(
      x: EmpIds*Strings*Strings*POW(Strings*
```

```
      Agencies) & x = a,b,c,d | c)


*L 118 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_4) &
      empl: dom(empRel_4) & works: POW((compRel_4
      <|EngagedIn)((%x.(x: dom(compRel_4) &
      compRel_4(x) = a,b | x,a,b))((empRel_4<|
      BelongsTo)((%x.(x: dom(empRel_4) & empRel_4
      (x) = a,b,c | x,a,b,c))(empl))))) => %x.(x:
      dom(empClass_3<+{empl|->((empClass_3<|%x.(x
      : EmpIds*Strings*Strings*POW(Strings*
      Agencies) & x = a,b,c,d | b))(empl),(
      empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | c))(
      empl),works)}) & (empClass_3<+{empl|->((
      empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | b))(
      empl),(empClass_3<|%x.(x: EmpIds*Strings*
      Strings*POW(Strings*Agencies) & x = a,b,c,d
      | c))(empl),works)})(x) = a,b,c | a) =
      empClass_3<+{empl|->((empClass_3<|%x.(x:
      EmpIds*Strings*Strings*POW(Strings*Agencies
      ) & x = a,b,c,d | b))((%x.(x: dom(empClass_
      3) & empClass_3(x) = a,b,c | x,a,b,c))(empl
      )),(empClass_3<|%x.(x: EmpIds*Strings*
      Strings*POW(Strings*Agencies) & x = a,b,c,d
      | c))((%x.(x: dom(empClass_3) & empClass_3(
      x) = a,b,c | x,a,b,c))(empl)),works)}<|%x.(
      x: EmpIds*Strings*Strings*POW(Strings*
      Agencies) & x = a,b,c,d | b)
*L 119 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_4) &
      empl: dom(empRel_4) & works: POW((compRel_4
      <|EngagedIn)((%x.(x: dom(compRel_4) &
      compRel_4(x) = a,b | x,a,b))((empRel_4<|
      BelongsTo)((%x.(x: dom(empRel_4) & empRel_4
      (x) = a,b,c | x,a,b,c))(empl))))) => dom(
      empClass_3<+{empl|->((empClass_3<|%x.(x:
      EmpIds*Strings*Strings*POW(Strings*Agencies
      ) & x = a,b,c,d | b))(empl),(empClass_3<|%x
      .(x: EmpIds*Strings*Strings*POW(Strings*
      Agencies) & x = a,b,c,d | c))(empl),works)}
      ) = dom(empClass_3<+{empl|->((empClass_3<|%
```

```
      x.(x: EmpIds*Strings*Strings*POW(Strings*
      Agencies) & x = a,b,c,d | b))((%x.(x: dom(
      empClass_3) & empClass_3(x) = a,b,c | x,a,b
      ,c))(empl)),(empClass_3<|%x.(x: EmpIds*
      Strings*Strings*POW(Strings*Agencies) & x =
      a,b,c,d | c))((%x.(x: dom(empClass_3) &
      empClass_3(x) = a,b,c | x,a,b,c))(empl)),
      works)})
*L 120 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_4) &
      not(empl: dom(empRel_4) & works: POW((
      compRel_4<|EngagedIn))((%x.(x: dom(compRel_4
      ) & compRel_4(x) = a,b | x,a,b))((empRel_4
      <|BelongsTo))((%x.(x: dom(empRel_4) & empRel
      _4(x) = a,b,c | x,a,b,c))(empl)))))) =>
      empClass_3 = empClass_3<+{empl|->((empClass
      _3<|%x.(x: EmpIds*Strings*Strings*POW(
      Strings*Agencies) & x = a,b,c,d | b))(empl)
      ,(empClass_3<|%x.(x: EmpIds*Strings*Strings
      *POW(Strings*Agencies) & x = a,b,c,d | c))(
      empl),works)}
*L 121 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_4) &
      not(empl: dom(empRel_4) & works: POW((
      compRel_4<|EngagedIn))((%x.(x: dom(compRel_4
      ) & compRel_4(x) = a,b | x,a,b))((empRel_4
      <|BelongsTo))((%x.(x: dom(empRel_4) & empRel
      _4(x) = a,b,c | x,a,b,c))(empl)))))) => %x.
      (x: dom(empClass_3<+{empl|->((empClass_3<|%
      x.(x: EmpIds*Strings*Strings*POW(Strings*
      Agencies) & x = a,b,c,d | b))(empl),(
      empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | c))(
      empl),works)}) | x) = empClass_3<|%x.(x:
      EmpIds*Strings*Strings*POW(Strings*Agencies
      ) & x = a,b,c,d | a)
*L 122 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_4) &
      not(empl: dom(empRel_4) & works: POW((
      compRel_4<|EngagedIn))((%x.(x: dom(compRel_4
      ) & compRel_4(x) = a,b | x,a,b))((empRel_4
      <|BelongsTo))((%x.(x: dom(empRel_4) & empRel
```

```
      _4(x) = a,b,c | x,a,b,c))(empl)))))) => %x.
      (x: dom(empClass_3<+{empl|->((empClass_3<|%
      x.(x: EmpIds*Strings*Strings*POW(Strings*
      Agencies) & x = a,b,c,d | b))(empl),(
      empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | c))(
      empl),works)}) & (empClass_3<+{empl|->((
      empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | b))(
      empl),(empClass_3<|%x.(x: EmpIds*Strings*
      Strings*POW(Strings*Agencies) & x = a,b,c,d
      | c))(empl),works)})(x) = a,b,c | c) =
      empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | d)
*L 123 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_4) &
      not(empl: dom(empRel_4) & works: POW((
      compRel_4<|EngagedIn)((%x.(x: dom(compRel_4
      ) & compRel_4(x) = a,b | x,a,b))((empRel_4
      <|BelongsTo)((%x.(x: dom(empRel_4) & empRel
      _4(x) = a,b,c | x,a,b,c))(empl)))))) => %x.
      (x: dom(empClass_3<+{empl|->((empClass_3<|%
      x.(x: EmpIds*Strings*Strings*POW(Strings*
      Agencies) & x = a,b,c,d | b))(empl),(
      empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | c))(
      empl),works)}) & (empClass_3<+{empl|->((
      empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | b))(
      empl),(empClass_3<|%x.(x: EmpIds*Strings*
      Strings*POW(Strings*Agencies) & x = a,b,c,d
      | c))(empl),works)})(x) = a,b,c | b) =
      empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | c)
*L 124 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_4) &
      not(empl: dom(empRel_4) & works: POW((
      compRel_4<|EngagedIn)((%x.(x: dom(compRel_4
      ) & compRel_4(x) = a,b | x,a,b))((empRel_4
      <|BelongsTo)((%x.(x: dom(empRel_4) & empRel
      _4(x) = a,b,c | x,a,b,c))(empl)))))) => %x.
      (x: dom(empClass_3<+{empl|->((empClass_3<|%
      x.(x: EmpIds*Strings*Strings*POW(Strings*
      Agencies) & x = a,b,c,d | b))(empl),(
```

```
      empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | c))(
      empl),works)}) & (empClass_3<+{empl|->((
      empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | b))(
      empl),(empClass_3<|%x.(x: EmpIds*Strings*
      Strings*POW(Strings*Agencies) & x = a,b,c,d
      | c))(empl),works)})(x) = a,b,c | a) =
      empClass_3<|%x.(x: EmpIds*Strings*Strings*
      POW(Strings*Agencies) & x = a,b,c,d | b)
*L 125 ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_4) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_4) &
      not(empl: dom(empRel_4) & works: POW((
      compRel_4<|EngagedIn)((%x.(x: dom(compRel_4
      ) & compRel_4(x) = a,b | x,a,b))((empRel_4
      <|BelongsTo)((%x.(x: dom(empRel_4) & empRel
      _4(x) = a,b,c | x,a,b,c))(empl)))))) => dom
      (empClass_3<+{empl|->((empClass_3<|%x.(x:
      EmpIds*Strings*Strings*POW(Strings*Agencies
      ) & x = a,b,c,d | b))(empl),(empClass_3<|%x
      .(x: EmpIds*Strings*Strings*POW(Strings*
      Agencies) & x = a,b,c,d | c))(empl),works)}
      ) = dom(empClass_3)
```

Anhang F

Die entwickelte Beweismethode

Dieser Anhang zeigt die *Beweismethode*, die die Theorien und Taktiken zum Beweis der Lemmas enthält und das Definitionskonzept realisiert.

```
 PMD

     ExtResearchCompaniesPmd

/*
   Beweismethode fuer die Beispielverfeinerung "ExtResearchCompanies".
      enthaelt:
           1)  spezielle Taktiken zur Realisierung eines
               Definitionskonzepts in Verfeinerungsmaschinen
           2)  spezielle Taktiken zum Beweis der mathematischen
               und Typlemmas, getrennt fuer Konsistenz- und
               Verfeinerungsnachweise.

   Autor:   Roland Stuckardt, Universitaet Frankfurt
   Version: 13. Januar 1991
*/

 AUTOP

     off



 LEMSEP

     off
```

```
 REFTAC

/*
   Adaptierung der Taktik fuer die Verarbeitung der Verfeinerungs-
   maschine: eine die Definitionen beruecksichtigende Taktik wird
   in der Theorie "RefinementLemmaTacticsX" mittels bcall zur
   Verfuegung gestellt.
*/

    (DED;RefinementLemmaTacticsX)


 ASNTAC


/*
   Taktik zur unmittelbaren Reduktion der fuer Definitionen
   resultierenden ASN-Lemmas
*/

    (DED;AsnDefLemmasX)


 TYPETAC

/*
   Kombination von Vorwaerts-und Rueckwaertstaktik
   fuer den Beweis der Typlemmas des Konsistenznachweises:
*/

   (DED;empty~;hypdown;SUB;(DED;exfalsoquodlibet;support;HYP;hypdown;SUB)~)
  ,
   (fempty~;GetPredicateX~;fsupport~;specialize~)~


/*
   alternativ:
   Kombination von Vorwaerts-und Rueckwaertstaktik
   fuer den Beweis der Typlemmas des Verfeinerungsnachweises:

   (DED;empty~;SUB;refinementsupport~;
   (exfalsoquodlibet;support;GetInvDefinitionX~;HYP~;SUB)~;
   (GetDefinitionX~;refinementsupport~)~;refinementEqualities~)~
  ,
   (fempty~;GetPredicateX~;fsupport~;FwdSemanticsX~;specialize~)~
```

```
*/


 MATHTAC

/*
   Kombination von Vorwaerts-und Rueckwaertstaktik fuer den
   Beweis der mathematischen Lemmas des Konsistenznachweises:
*/

   (DED;empty~;hypdown;SUB;(DED;exfalsoquodlibet;support;HYP;hypdown;SUB)~)
  ,
   (fempty~;GetPredicateX~;fsupport~;specialize~)~

/*
   alternativ:
   Kombination von Vorwaerts-und Rueckwaertstaktik fuer den
   Beweis der mathematischen Lemmas des Verfeinerungsnachweises:

   (DED;(refinementEqualities~;GetInvDefinitionX~;HYP~)~)
  ,
   (fempty~;GetPredicateX~;fsupport~;GetDefinitionX~)~

*/


 THYS

/*
   Hilfstheorien zur Generierung korrekter Lemmas:
   mit Respektierung von Definitionen im ASN-Paragraph.
*/

  THEORY  RefinementLemmaTacticsX  IS

/*
   Unterschiedliche Taktiken fuer Initialisierungs- und
   Verfeinerungslemmas der Verfeinerungsbeweisverpflichtungen:
   ohne bzw. mit Definitionsanwendungen vor Substitution.
   Jeweils fuer Autoproof-Beweise (Bezugname auf interne Syntax)
   sowie Beweise bei Zwischenspeicherung in RefinementLemmaX
   (Bezugnahme auf externe Syntax) realisiert.
*/

    bwrite("\n Illegal Proof Obligation Syntax\n ")  &  bhalt
      => M

  ;
```

```
/*
   fuer interne Darstellung:
*/


    bwrite("\n processing Refinement Proof Obligation \n ") &
    bcall(((PrepareConjectureX;REV;
           (PrepareConjectureX;ARI)~;FLAT)~;GetOperationX~;
           GetInitialisationX~;TransCondX~;GetInvariantX;
           GetDefinitionX~;
           BuildConjectureX;FLAT~;(((GEN;DED)~;CalculusX)~;
           MathTypeLemmaGenerationX)~) : (chkref(inv(A),B,C)) )
       =>
    chkref(inv(A),B,C)

 ;

    bwrite("\n processing Initialisation Proof Obligation \n ") &
    bcall(((PrepareConjectureX;REV;
           (PrepareConjectureX;ARI)~;FLAT)~;GetOperationX~;
           GetInitialisationX~;TransCondX~;GetInvariantX;
           BuildConjectureX;FLAT~;(((GEN;DED)~;CalculusX)~;
           InitMathTypeLemmaGenerationX)~) : (chkref(inv(A),ini(B),C)) )
       =>
    chkref(inv(A),ini(B),C)

;

/*
   fuer externe Darstellung:
*/

    bwrite("\n processing Initialisation Proof Obligation \n ") &
    bcall((RefinementLemmaX;DED;(PrepareConjectureX;REV;
           (PrepareConjectureX;ARI)~;FLAT)~;GetOperationX~;
           GetInitialisationX~;TransCondX~;GetInvariantX;
           BuildConjectureX;FLAT~;(((GEN;DED)~;CalculusX)~;
           InitMathTypeLemmaGenerationX)~) : (Check_initialisation: M) )
       =>
    Check_initialisation: M

  ;

/*
   fuer den Fall von Verfeinerungslemmas "gewoehnlicher"
   Operationen werden VOR Ausfuehrung der Substitutionen
   die in der Theorie GetDefinitionX als Gleichungen erfassten
```

```
   Definitionen expandiert.
   Nach Auswertung gemaess den Beziehungen des Substitutions-
   kalkuels werden (vgl.u., Theorie MathTypeLemmaGenerationX)
   weitere Umformungen vorgenommmen, in denen, sofern moeglich,
   Definitionen invers angewandt werden (GetInvDefinitionX)
   und die in der Hypothese stehenden CHANGE-Gleichungen
   angewandt werden, um eine leichter handhabbare Darstellung
   der anschliessend generierten Lemmas zu erreichen.
*/

    bwrite("\n processing Refinement Proof Obligation \n ") &
    bcall((RefinementLemmaX;DED;(PrepareConjectureX;REV;
           (PrepareConjectureX;ARI)~;FLAT)~;GetOperationX~;
           GetInitialisationX~;TransCondX~;GetInvariantX;
           GetDefinitionX~;
           BuildConjectureX;FLAT~;(((GEN;DED)~;CalculusX)~;
           MathTypeLemmaGenerationX)~) : (Check_refinement: M) )
       =>
    Check_refinement: M

  END

&

  THEORY  InitMathTypeLemmaGenerationX  IS

    bcall((HYP~; PrepareLemma1X): A)
       =>
    A

  END

&

/*
   Vornahme bestimmter Vereinfachungen vor Lemmagenerierung
   (vgl. Beschreibung oben)
*/

  THEORY  MathTypeLemmaGenerationX  IS

    bcall((GetInvDefinitionX~; HYP~; GetDefinitionX~;
          HYP~; GetDefinitionX~; PrepareLemma1X): A)
       =>
    A

  END
```

```
&

  THEORY AsnDefLemmasX  IS

    A := B

  END

&

/*
   Theorien, die den Beweis der mathematischen und Typlemmas
   der Konsistenz- und Verfeinerungsnachweise unterst\"utzen:
*/


  THEORY  support  IS



/*
   Spezielle Regeln zum Beweis von Lemmas der Klasse MSL
*/


   inhyp( !x.!y.(x,y: T => ( (F(x) = F(y)) & (G(x) = G(y)) =>  (x = y) ) ) )
         &
   inhyp(x,y : T - U)
         &
   inhyp((U <<| F)(x) = (U <<| F)(y))
         &
   inhyp((U <<| G)(x) = (U <<| G)(y))
            =>
   (x = y)

       ;

   inhyp( !x.!y.(x,y: T => ( (F(x) = F(y)) =>  (x = y) ) ) )
         &
   inhyp(x,y : T - U)
         &
   inhyp((U <<| F)(x) = (U <<| F)(y))
       =>
   (x = y)

       ;

/*
```

```
   Spezielle Regeln zum Beweis von Lemmas der Klasse MSE
*/

   inhyp( !x.!y.(x,y: T => ( (F(x) = F(y)) =>  (x = y) ) ) )
         &
   inhyp(x,y : T \/ {z})
         &
   inhyp(F : T  --> S)
         &
   inhyp(n : S - ran(F))
         &
   inhyp((F<+{z|->n})(x) = (F<+{z|->n})(y))
            =>
   (x = y)

       ;

   inhyp( !x.!y.(x,y: T => ( (F(x) = F(y))  & (G(x) = G(y)) =>  (x = y) ) ) )
         &
   inhyp(x,y : T \/ {z})
         &
   inhyp(F : T  --> S)
         &
   inhyp(G : T  --> U)
         &
   inhyp(n : S - ran(F))
         &
   inhyp(m : U - ran(G))
         &
   inhyp((F<+{z|->n})(x) = (F<+{z|->n})(y))
         &
   inhyp((G<+{z|->m})(x) = (G<+{z|->m})(y))
            =>
   (x = y)

       ;

   inhyp(E : U - S)  &  ran(F) incl S
        =>
    (G<+{E|->D})(F(A)) = G(F(A))

        ;

   inhyp(X : T)
        &
   ((X = E)  =>  N incl G(H(E)))
        &
   (X : T - {E}  =>  F(X) incl G(H(X)))
```

```
     =>
((F<+{E|-> N})(X) incl G(H(X)))

     ;

inhyp(X : S \/ T)
     &
(X : S  =>  F(X) incl G(H(X)))
     &
(X : T  =>  F(X) incl G(H(X)))
     =>
 (F(X) incl G(H(X)))

     ;

 F : T --> S  &  E : T  &  D : S
    =>
 F<+{E |-> D} : T --> S

      ;

 F : T +-> S  &  E : T  &  D : S
    =>
 F<+{E |-> D} : T +-> S

      ;


inhyp(A : R - T)  &  inhyp(T : POW(R))
 &  B : S  &  F : T --> S
    =>
 F<+{A|-> B} : (T \/ {A}) --> S

      ;

inhyp(A : R - T)  &  inhyp(T : POW(R))
 &  B : S  &  F : T +-> S
    =>
 F<+{A|-> B} : (T \/ {A}) +-> S

      ;

 inhyp({} = { Y | Y : T  &  X : F(Y)})  &   F : T --> POW(S)  &  X : S
    =>
 F : T --> POW(S - {X})

      ;
 inhyp({} = { Y | Y : T  &  X : F(Y)})  &   F : T +-> POW(S)  &  X : S
```

```
      =>
    F : T +-> POW(S - {X})

        ;

    inhyp(F : T --> U)  &  U incl S  &  B : S  &  A : POW(T)
      =>
    F<+%x.(x : A | B) : T --> S

        ;

    inhyp(F : T +-> U)  &  U incl S  &  B : S  &  A : POW(T)
      =>
    F<+%x.(x : A | B) : T +-> S

        ;

    F : T --> S
      =>
    (D <<| F) : (T - D) --> S

        ;

    F : T +-> S
      =>
    (D <<| F) : (T - D) +-> S

        ;

    F : T --> S
      =>
    F : T --> (S \/ U)

        ;

    F : T +-> S
      =>
    F : T +-> (S \/ U)

        ;

/*
   allgemeine mengentheoretische Beziehungen:
*/

    S incl T
      =>
    S incl (T \/ U)
```

```
     ;

S incl T
   =>
POW(S) incl POW(T)

     ;

S incl (T \/ S)

     ;

S incl S

     ;

S incl T
   =>
S  :  POW(T)
     ;

E : S
   =>
E : (S \/ T)

     ;

E : S
   =>
{E} : POW(S)

     ;

E : (S \/ {E})

     ;

E : {E}

     ;

S : POW(S)

     ;

inhyp(F : T --> U) &   U incl S  /* &  E : T  */
   =>
```

```
     F(E) : S

          ;
    inhyp(F : T --> S)   /* &  E : T  */
       =>
     F(E) : S

          ;

    S : T  &  R : T
       =>
    (R \/ S) : T

          ;

/*
    folgende Regel ist redundant. Sie ermoeglicht in
    bestimmmten Faellen eine Beweisbeschleunigung.
*/

    E : T  &  S : POW(T)
       =>
    (S \/ {E}) : POW(T)

          ;


    E : T  =>  E - U : T


  END

 &

  THEORY  empty  IS

    {} :  S +-> T

          ;

    {} :  {} --> T

          ;

    {} : POW(S)

  END
```

```
&

 FWDTHEORY   fempty  IS

  a : {}
    =>
  FALSE

 END

&

 FWDTHEORY  fsupport  IS


  E : T
   =>
  {E} incl T

     ;


  F : T --> S
      =>
  dom(F) = T

      ;

  F : T --> S
      =>
  ran(F) incl S

      ;

  F : T +-> S
      =>
  dom(F) incl T

      ;

  F : T +-> S
      =>
  ran(F) incl S

      ;

  E : T  &   F : T --> S
      =>
```

```
F(E) : S

    ;

 X : T - U  &  dom(F) = T
    =>
(U <<| F)(X) = F(X)

    ;

(X : T) &  (E : U - T) &  (F : T --> S) & (N : S)
    =>
(F<+{E |-> N})(X) =  F(X)

     ;

(X : T) &  (E : U - T) &  (F : T +-> S) & (N : S)
    =>
(F<+{E |-> N})(X) =  F(X)

     ;

(X : T - {E})  &  (E : T)  &  (F : T --> S) & (N : S)
    =>
(F<+{E |-> N})(X) =  F(X)

     ;

(X : T - {E})  &  (E : T)  &  (F : T +-> S) & (N : S)
    =>
(F<+{E |-> N})(X) =  F(X)

     ;


A : POW(S)  &  T : POW(A)
    =>
T : POW(S)

     ;

F : T --> POW(S)  &  E : T
    =>
F(E) : POW(S)

     ;

E : dom(F)  &   F : D --> R
```

```
      =>
   E : D

;

   E : dom(F)  &   F : D +-> R
      =>
   E : D

;

   E : dom(F)  &  F = G
      =>
   E : dom(G)

;

   (X : {E}) &  (N : S) &  (F : T --> S)
      =>
   (F<+{E |-> N})(X) = N

       ;

   (F<+{E |-> N})(E) = N

       ;

   not(X = E)  &  X : dom(F)
       =>
   (F<+{E |-> N})(X) = F(X)

       ;

   X : T  &  not(X = E)
       =>
   X : T - {E}

        ;

   E : T - S
      =>
   E : T

;

   S incl T
      =>
   (S \/ T) = T
```

```
      ;

   (S incl T)
      =>
   (T \/ S) = T

      ;

   S incl T
      =>
   S : POW(T)
      ;


    S : POW(T)
      =>
    S incl T
      ;


   S = T  &  S incl U
      =>
   T incl U

      ;

   S = T & U incl T
      =>
   U incl S

      ;


   F(A) : S  &  S = T
      =>
   F(A) : T

;

   F(A) : S  &   F = G
      =>
   G(A) : S

;

   F(A) : S   &   F = G
      =>
```

```
   F(A) = G(A)

;

/*
   Folgende zwei Regeln sind bei Verfeinerungslemmabeweisen
   abzuschalten, da sie in einigen Faellen zu Endlosschleifen
   beim FWD-Schliessen fuehren.
*/

   X = E  &   F : T --> S
       =>
   F(X) = F(E)

        ;

   X = E  &   F : T +-> S
       =>
   F(X) = F(E)

        ;

   S = T  &   E : POW(S)
       =>
   E : POW(T)

;



   E : S  &  S = T
       =>
   E : T

;


    T = U  &  S = T
       =>
    S = U

;

/*
   Folgende Regel sollte ebenfalls waehrend dem Beweis von
   Verfeinerungslemmas abgeschaltet werden.
*/
```

```
   T = U  &  T = S
      =>
   S = U

       ;


   A  &  (A => B)
      =>
   B

       ;

   A,B : C  =  (A : C  &  B : C)

 END

&

 FWDTHEORY  specialize  IS


  (X : {E}) &  (N : S) &  (F : T --> S)
      =>
  (F<+{E |-> N})(X) = N

       ;


 !x.(x : T => P)   &  (Y : T)
      =>
   [x:= Y]P

       ;

!x.(x : T => P)   &  (Y : T - U)
      =>
   [x:= Y]P

       ;

 !x.(x : T => P)   &  (x : T)
      =>
      P

      ;
```

```
  !x.(x : T => P)   &  (x : T - U)
       =>
       P


   END

 &

  THEORY  hypdown IS

    inhyp([x:= E]P)  &   (([x:= E]P) => Q)
       =>
     Q

  END

&

  THEORY  exfalsoquodlibet  IS

    inhyp(FALSE)
       =>
    P

  END

&

  THEORY  refinementEqualities  IS

    (S <<| (%x.(x : T & P | C)))  =  (%x.(x : (T - S)  &  P | C))

      ;

    (S <<| (%x.(x : T | C)))  =  (%x.(x : (T - S)  | C))

      ;

    (S <<| (%x.(x : dom(F) & (F(x) = A) | B)))
      =
    (%x.(x : dom(S <<| F)  & ((S <<| F)(x) = A) | B))

      ;

    ((%x.(x : T | x))<+{B |-> B})  =   (%x.(x : (T \/ {B}) | x))

      ;
```

```
    dom(F <+ G)  =  (dom(F) \/ dom(G))

      ;

    dom({E |-> A}) = {E}

      ;

    dom(%x.(x : T | A)) = T

      ;

    (dom(F) - S)   =  (dom(S <<| F))

      ;

    (dom(F \/ {(A,B)})) = (dom(F) \/ {first(A)})

      ;

/*
   Spezielle Regeln zum Beweis von Lemmas der Klasse MV2E2:
*/

    (%x.(x : dom(F \/ {(A,B)}) & (F \/ {(A,B)})(x) = G | G))
      =
    ((%x.(x : dom(F) & F(x) = G | G))<+{A |-> B})

      ;


    (%x.(x : dom(F \/ {(A,B,C)}) & (F \/ {(A,B,C)})(x) = (G,H) | G))
      =
    ((%x.(x : dom(F) & F(x) = (G,H) | G))<+{A |-> B})
      ;
    (%x.(x : dom(F \/ {(A,B,C)}) & (F \/ {(A,B,C)})(x) = (G,H) | H))
      =
    ((%x.(x : dom(F) & F(x) = (G,H) | H))<+{A |-> C})

      ;


    (%x.(x : dom(F \/ {(A,B,C,D)}) & (F \/ {(A,B,C,D)})(x) = (G,H,I) | G))
      =
    ((%x.(x : dom(F) & F(x) = (G,H,I) | G))<+{A |-> B})
      ;
    (%x.(x : dom(F \/ {(A,B,C,D)}) & (F \/ {(A,B,C,D)})(x) = (G,H,I) | H))
```

```
  =
((%x.(x : dom(F) & F(x) = (G,H,I) | H))<+{A |-> C})
  ;
(%x.(x : dom(F \/ {(A,B,C,D)}) & (F \/ {(A,B,C,D)})(x) = (G,H,I) | I))
  =
((%x.(x : dom(F) & F(x) = (G,H,I) | I))<+{A |-> D})

  ;


/*
   Spezielle Regeln zum Beweis von Lemmas der Klasse MV2Ae1:
    - formuliert fuer maximal vier abhaengige Klassenattribute
    - von unten nach oben: Regeln fuer
                           > viertletzte Attributstelle
                           > drittletzte Attributstelle
                            ....
                           > letzte Attributstelle
      jeweils fuer die Faelle > Aenderung
                              > keine Aenderung
    - Wegen des geschickten Ausnutzens der Linksassoziativitaet des
      Tupelseparators ',' reichen O(n) Regeln fuer die
      Beruecksichtigung von n abhaengigen Attributstellen
      anstelle der aus der Trivialloesung resultierenden
      O(n*n) Regeln.
*/

    (%x.(x : dom(F<+{S |-> A })
         & ((F<+{S |-> A })(x) = G) | G))
      =
    ((%x.(x : dom(F) & (F(x) = G) | G))<+{S |-> A})

      ;



    (%x.(x : dom(F<+{S |->
                 (%x.(x : dom(F) & (F(x) = G) | G))(S)})
         & ((F<+{S |->
              (%x.(x : dom(F) & (F(x) = G) | G))(S)})(x) = G) | G))
      =
    (%x.(x : dom(F) & (F(x) = G) | G))

      ;


    (%x.(x : dom(F<+{S |-> (A,B)})
         & ((F<+{S |-> (A,B)})(x) = (G,H)) | G))
```

```
  =
((%x.(x : dom(F) & (F(x) = (G,H)) | G))<+{S |-> A})

  ;


(%x.(x : dom(F<+{S |->
             ((%x.(x : dom(F) & (F(x) = (G,H)) | G))(S),
              B)})
     & ((F<+{S |->
        ((%x.(x : dom(F) & (F(x) = (G,H)) | G))(S),
         B)})(x) = (G,H)) | G))
  =
(%x.(x : dom(F) & (F(x) = (G,H)) | G))

  ;

(%x.(x : dom(F<+{S |-> (A,B)})
     & ((F<+{S |-> (A,B)})(x) = (G,H)) | H))
  =
((%x.(x : dom(F) & (F(x) = (G,H)) | H))<+{S |-> B})

  ;


(%x.(x : dom(F<+{S |->
             (A,
              (%x.(x : dom(F) & (F(x) = (G,H)) | H))(S)
              )})
     & ((F<+{S |->
         (A,
          (%x.(x : dom(F) & (F(x) = (G,H)) | H))(S)
         )})(x) = (G,H)) | H))
  =
(%x.(x : dom(F) & (F(x) = (G,H)) | H))

  ;


(%x.(x : dom(F<+{S |-> (A,B,C)})
     & ((F<+{S |-> (A,B,C)})(x) = (G,H,I)) | G))
  =
((%x.(x : dom(F) & (F(x) = (G,H,I)) | G))<+{S |-> A})

  ;


(%x.(x : dom(F<+{S |->
```

```
          ((%x.(x : dom(F) & (F(x) = (G,H,I)) | G))(S),
           B,C)})
     & ((F<+{S |->
        ((%x.(x : dom(F) & (F(x) = (G,H,I)) | G))(S),
        B,C)})(x) = (G,H,I)) | G))
  =
(%x.(x : dom(F) & (F(x) = (G,H,I)) | G))

  ;

(%x.(x : dom(F<+{S |-> (A,B,C)})
   & ((F<+{S |->
      (A,B,C)})(x) = (G,H,I)) | H))
  =
((%x.(x : dom(F) & (F(x) = (G,H,I)) | H))<+{S |-> B})

  ;


(%x.(x : dom(F<+{S |->
        (A,
         (%x.(x : dom(F) & (F(x) = (G,H,I)) | H))(S),
         C)})
     & ((F<+{S |->
         (A,
          (%x.(x : dom(F) & (F(x) = (G,H,I)) | H))(S),
          C)})(x) = (G,H,I)) | H))
  =
(%x.(x : dom(F) & (F(x) = (G,H,I)) | H))

  ;


(%x.(x : dom(F<+{S |-> (A,B,C,D)})
     & ((F<+{S |->
        (A,B,C,D)})(x) = (G,H,I,J)) | G))
  =
((%x.(x : dom(F) & (F(x) = (G,H,I,J)) | G))<+{S |-> A})

  ;


(%x.(x : dom(F<+{S |->
           ((%x.(x : dom(F) & (F(x) = (G,H,I,J)) | G))(S),
            B,C,D)})
     & ((F<+{S |->
        ((%x.(x : dom(F) & (F(x) = (G,H,I,J)) | G))(S),
        B,C,D)})(x) = (G,H,I,J)) | G))
```

```
      =
    (%x.(x : dom(F) & (F(x) = (G,H,I,J)) | G))

      ;

    (%x.(x : dom(F<+{S |-> (A,B,C,D)})
       & ((F<+{S |->
           (A,B,C,D)})(x) = (G,H,I,J)) | H))
      =
    ((%x.(x : dom(F) & (F(x) = (G,H,I,J)) | H))<+{S |-> B})

      ;


    (%x.(x : dom(F<+{S |->
                (A,(%x.(x : dom(F) & (F(x) = (G,H,I,J)) | H))(S),
                 C,D)})
          & ((F<+{S |->
          (A,
           (%x.(x : dom(F) & (F(x) = (G,H,I,J)) | H))(S),
           C,D)})(x) = (G,H,I,J)) | H))
      =
    (%x.(x : dom(F) & (F(x) = (G,H,I,J)) | H))

      ;


    first((A)) = A
      ;
    first((A,B)) = first((A))


  END

&




  THEORY  refinementsupport  IS

/*
    enthaelt die fuer den Beweis der Typlemmas der
    Verfeinerungsnachweise benoetigten Regeln spezieller Gestalt,
    die aus Effizienzgruenden nicht in die allgemeine Theorie
    "support" eingegliedert werden sollen.
*/
```

```
/*
    "rekursive" Axiomatisierung von Tupelgleichheit,
    Gleichheit karteasischer Produkte
    und Tupelelementbeziehungen:
*/


     (A = L)  &  (B = M)
         =>
     (A,B) = (L,M)

         ;

     (A = L)  &  (B = M)
         =>
     (A * B) = (L * M)

          ;

    (A : S)  &  (B : T)
         =>
     (A,B) : (S * T)

          ;


     inhyp(E : U - R)  &   (U = T)  &  (R = S)
       =>
     E : T - S

      ;


     dom(S <<| F) =  (dom(F) - S)

       ;


    inhyp(F : V --> A * B)
    & (V = W) & (A * B) = (R * S)
      =>
    (F : W --> R * S)

       ;

/*
   Spezielle Regel zum Beweis von Lemmas der Klasse TV2E:
```

```
     (eine Regel reicht fuer beliebige Anzahl abhaengiger
      Attribute!)
*/

    inhyp(F : W --> A * B)
    & (W = dom(F)) & (A * B) = (R * S)
                   & (tail(G,H,I) : (A * B))
       =>
    (F \/ {(G,H,I)}) : dom(F \/ {(G,H,I)})
          --> R * S
        ;

/*
   Spezielle Regeln zum Beweis von Lemmas der Klasse TV2Ae:
      - okproj(E) beschreibt diejenigen abhaengigen Attribute,
        deren Wert erhalten bleibt. Fuer diese Attribute ent-
        faellt der Typnachweis (Rolle des Axioms okproj(E) : T).
      - nach Durchfuehrung aller okproj-Vereinfachungen wird
        die oberste Regel angewandt, die die Typnachweise
        fuer die individuellen Attribute generiert.
      - O(n) Regeln reichen fuer n abhaengige Attribute
*/

    inhyp(F : Z --> A * B )
     & (Z = dom(F)) & (A*B) = (V*W)
                    & (R,S) : (A*B)
        =>
     F<+{E |-> (R,S)} :
       dom(F<+{E |-> (R,S)}) --> V * W

        ;



     F<+{E |->
       (R,okproj(E))}
       : dom(F<+{E |->
            (R,
             okproj(E))})
         --> V * W
        =>
     F<+{E |->
       (R,
        (%x.(x : dom(F) & F(x) = (A,B) | B))(E))}
       : dom(F<+{E |->
            (R,
             (%x.(x : dom(F) & F(x) = (A,B) | B))(E)) })
             --> V * W
```

```
    ;


F<+{E |->
  (okproj(E),S)}
  : dom(F<+{E |->
       (okproj(E),
        S)}) --> V * W
   =>
F<+{E |->
  ((%x.(x : dom(F) & F(x) = (A,B) | A))(E),
    S)}
  : dom(F<+{E |->
       ((%x.(x : dom(F) & F(x) = (A,B) | A))(E),
        S)}) --> V * W
    ;

F<+{E |->
  (R,okproj(E),T)}
  : dom(F<+{E |->
       (R,
        okproj(E),
        T)}) --> V * W * X
   =>
F<+{E |->
  (R,
   (%x.(x : dom(F) & F(x) = (A,B,C) | B))(E),
    T)}
  : dom(F<+{E |->
       (R,
        (%x.(x : dom(F) & F(x) = (A,B,C) | B))(E),
        T)}) --> V * W * X
    ;


F<+{E |->
  (okproj(E),S,T)}
  : dom(F<+{E |->
       (okproj(E),
        S,T)}) --> V * W * X
   =>
F<+{E |->
  ((%x.(x : dom(F) & F(x) = (A,B,C) | A))(E),
    S,T)}
  : dom(F<+{E |->
       ((%x.(x : dom(F) & F(x) = (A,B,C) | A))(E),
        S,T)}) --> V * W * X
    ;
```

```
    F<+{E |->
      (R,okproj(E),T,U)}
      : dom(F<+{E |->
           (R,
            okproj(E),
            T,U)}) --> V * W * X * Y
       =>
    F<+{E |->
      (R,
       (%x.(x : dom(F) & F(x) = (A,B,C,D) | B))(E),
        T,U)}
      : dom(F<+{E |->
           (R,
            (%x.(x : dom(F) & F(x) = (A,B,C,D) | B))(E),
            T,U)}) --> V * W * X * Y
        ;


    F<+{E |->
      (okproj(E),S,T,U)}
      : dom(F<+{E |->
           (okproj(E),
            S,T,U)}) --> V * W * X * Y
       =>
    F<+{E |->
      ((%x.(x : dom(F) & F(x) = (A,B,C,D) | A))(E),
        S,T,U)}
      : dom(F<+{E |->
           ((%x.(x : dom(F) & F(x) = (A,B,C,D) | A))(E),
            S,T,U)}) --> V * W * X * Y
        ;

    okproj(E) : T

        ;

    ran(%x.(x : T | x)) = T

        ;

/*
    folgende Axiome werden fuer die allgemeine
    Verarbeitung kartesischer Produktnotationen und
    Tupel benoetigt:
*/

   tail(R,S) = S
```

```
        ;
    tail(R,S,T) = (tail(R,S),T)
         ;
    tail(R * S) = S
        ;
    tail(R * S * T) = tail(R * S) * T

  END

END
```

Anhang G

Beispielbeweise

Anhang G zeigt *Beispielbeweise* von mathematischen und Typlemmas, die mit Hilfe der vorgestellten Beweismethode durchgeführt wurden, sowie die Verarbeitung einer Konsistenzbeweisverpflichtung durch die Theoriensammlung `devB`.

G.1 Verarbeitung einer Beweisverpflichtung

Verarbeitung der Konsistenzbeweisverpflichtung der Operation `removeProject` im Autobeweismodus und unter Verwendung der von `devB` vorgegebenen Taktik.

```
PROOF

  1  ctx(ExtResearchCompaniesMch)                HYP
  2  inv(ExtResearchCompaniesMch_1)              HYP
  3    ctx(ExtResearchCompaniesCtx4)             1 GetPredicateX.27
  4    companies_1: POW(COMPANIES)               2 GetPredicateX.2
  5    compName_1: companies_1 --> CompNames     2 GetPredicateX.2
  6    budget_1: companies_1 --> MONEY           2 GetPredicateX.2
  7    employees_1: POW(EMPLOYEES)               2 GetPredicateX.2
  8    empName_1: employees_1 --> EmpNames       2 GetPredicateX.2
  9    belongsTo_1: employees_1 --> companies_1  2 GetPredicateX.2
 10    !x.!y.(x,y: companies_1 => (compName_1(x)
    = compName_1(y) => x = y))                   2 GetPredicateX.2
 11    !x.(x: employees_1 => worksOn_1(x) incl
    engagedIn_1(belongsTo_1(x)))                 2 GetPredicateX.2
 12    ProjIdRecType = ProjNames*Agencies        3 FwdSemanticsX.106
 13    PROJECTS = ProjIdRecType                  3 FwdSemanticsX.105
 14    ProjNames = Strings                       3 FwdSemanticsX.102
 15    delaylemma4((engagedIn_1: companies_1 -->
```

```
      POW(projects_1-{proj})),TypeLemma,(DED;
      GEN;HYP)~,(GetPredicateX~;FwdSemanticsX~;
      FEQL)~)                                  PrepareLemma3X.1
 16   delaylemma4((engagedIn_1: companies_1 -->
      POW(projects_1-{proj})),TypeLemma,
      typelemmatac,(GetPredicateX~;
      FwdSemanticsX~;FEQL)~)                   15 GetProofMethodX.57
 17   delaylemma4((engagedIn_1: companies_1 -->
      POW(projects_1-{proj})),TypeLemma,
      typelemmatac,typelemmaftac)              16 GetProofMethodX.58
 18   delaylemma3((engagedIn_1: companies_1 -->
      POW(projects_1-{proj})),
      ExtResearchCompaniesMch,off)             17 PrepareLemma3X.5
 19   bcall((DED;GEN;HYP)~,(GetPredicateX~;
      FwdSemanticsX~;FEQL)~: (engagedIn_1:
      companies_1 --> POW(projects_1-{proj})) |
      (GetProofMethodX~;PrepareLemma3X)~:
      delaylemma3((engagedIn_1: companies_1 -->
      POW(projects_1-{proj})),
      ExtResearchCompaniesMch,off))            18 CAL
 20   bcalldelay((DED;GEN;HYP)~,(GetPredicateX~
      ;FwdSemanticsX~;FEQL)~: (engagedIn_1:
      companies_1 --> POW(projects_1-{proj})) |
      (GetProofMethodX~;PrepareLemma3X)~:
      delaylemma3((engagedIn_1: companies_1 -->
      POW(projects_1-{proj})),
      ExtResearchCompaniesMch,off))            19 PrepareLemma2X.1
 21   bcalldelay((DED;GEN;HYP)~,(GetPredicateX~
      ;FwdSemanticsX~;FEQL)~: (engagedIn_1:
      companies_1 --> POW(projects_1-{proj})) |
      (GetProofMethodX~;PrepareLemma3X)~:
      delaylemma3((engagedIn_1: companies_1 -->
      POW(projects_1-{proj})),currmch,off))    20 GetProofMethodX.51
 22   bcalldelay((DED;GEN;HYP)~,(GetPredicateX~
      ;FwdSemanticsX~;FEQL)~: (engagedIn_1:
      companies_1 --> POW(projects_1-{proj})) |
      (GetProofMethodX~;PrepareLemma3X)~:
      delaylemma3((engagedIn_1: companies_1 -->
      POW(projects_1-{proj})),currmch,
      lemsepflag))                             21 GetProofMethodX.53
 23   bcalldelay(typelemmataccall(engagedIn_1:
      companies_1 --> POW(projects_1-{proj})) |
      (GetProofMethodX~;PrepareLemma3X)~:
      delaylemma3((engagedIn_1: companies_1 -->
      POW(projects_1-{proj})),currmch,
      lemsepflag))                             22 GetProofMethodX.59
 24   delaylemma1(engagedIn_1: companies_1 -->
      POW(projects_1-{proj}))                  23 PrepareLemma2X.5
```

```
25  delaylemma((engagedIn_1: companies_1 -->
    POW(projects_1-{proj})),on)                 24 PrepareLemma2X.6
26  delaylemma((engagedIn_1: companies_1 -->
    POW(projects_1-{proj})),autopflag)          25 GetProofMethodX.52
27  bcall((GetProofMethodX~;TempX;
    PrepareLemma2X)~: delaylemma((engagedIn_1
    : companies_1 --> POW(projects_1-{proj}))
    ,autopflag))                                26 CAL
28  engagedIn_1: companies_1 --> POW(projects
    _1-{proj})                                  27 PrepareLemma1X.1
29  delaylemma4((projects_1-{proj}: POW(
    PROJECTS)),TypeLemma,(DED;GEN;HYP)~,(
    GetPredicateX~;FwdSemanticsX~;FEQL)~)       PrepareLemma3X.1
30  delaylemma4((projects_1-{proj}: POW(
    PROJECTS)),TypeLemma,typelemmatac,(
    GetPredicateX~;FwdSemanticsX~;FEQL)~)       29 GetProofMethodX.57
31  delaylemma4((projects_1-{proj}: POW(
    PROJECTS)),TypeLemma,typelemmatac,
    typelemmaftac)                              30 GetProofMethodX.58
32  delaylemma3((projects_1-{proj}: POW(
    PROJECTS)),ExtResearchCompaniesMch,off)     31 PrepareLemma3X.5
33  bcall((DED;GEN;HYP)~,(GetPredicateX~;
    FwdSemanticsX~;FEQL)~: (projects_1-{proj}
    : POW(PROJECTS)) | (GetProofMethodX~;
    PrepareLemma3X)~: delaylemma3((projects_1
    -{proj}: POW(PROJECTS)),
    ExtResearchCompaniesMch,off))               32 CAL
34  bcalldelay((DED;GEN;HYP)~,(GetPredicateX~
    ;FwdSemanticsX~;FEQL)~: (projects_1-{proj
    }: POW(PROJECTS)) | (GetProofMethodX~;
    PrepareLemma3X)~: delaylemma3((projects_1
    -{proj}: POW(PROJECTS)),
    ExtResearchCompaniesMch,off))               33 PrepareLemma2X.1
35  bcalldelay((DED;GEN;HYP)~,(GetPredicateX~
    ;FwdSemanticsX~;FEQL)~: (projects_1-{proj
    }: POW(PROJECTS)) | (GetProofMethodX~;
    PrepareLemma3X)~: delaylemma3((projects_1
    -{proj}: POW(PROJECTS)),currmch,off))       34 GetProofMethodX.51
36  bcalldelay((DED;GEN;HYP)~,(GetPredicateX~
    ;FwdSemanticsX~;FEQL)~: (projects_1-{proj
    }: POW(PROJECTS)) | (GetProofMethodX~;
    PrepareLemma3X)~: delaylemma3((projects_1
    -{proj}: POW(PROJECTS)),currmch,
    lemsepflag))                                35 GetProofMethodX.53
37  bcalldelay(typelemmataccall(projects_1-{
    proj}: POW(PROJECTS)) | (GetProofMethodX~
    ;PrepareLemma3X)~: delaylemma3((projects_
    1-{proj}: POW(PROJECTS)),currmch,
```

```
      lemsepflag))                                    36 GetProofMethodX.59
 38   delaylemma1(projects_1-{proj}: POW(
      PROJECTS))                                      37 PrepareLemma2X.5
 39   delaylemma((projects_1-{proj}: POW(
      PROJECTS)),on)                                  38 PrepareLemma2X.6
 40   delaylemma((projects_1-{proj}: POW(
      PROJECTS)),autopflag)                           39 GetProofMethodX.52
 41   bcall((GetProofMethodX~;TempX;
      PrepareLemma2X)~: delaylemma((projects_1-
      {proj}: POW(PROJECTS)),autopflag))              40 CAL
 42   projects_1-{proj}: POW(PROJECTS)                41 PrepareLemma1X.1
 43   delaylemma4(({proj}<<|projName_1:
      projects_1-{proj} --> ProjNames),
      TypeLemma,(DED;GEN;HYP)~,(GetPredicateX~;
      FwdSemanticsX~;FEQL)~)                          PrepareLemma3X.1
 44   delaylemma4(({proj}<<|projName_1:
      projects_1-{proj} --> ProjNames),
      TypeLemma,typelemmatac,(GetPredicateX~;
      FwdSemanticsX~;FEQL)~)                          43 GetProofMethodX.57
 45   delaylemma4(({proj}<<|projName_1:
      projects_1-{proj} --> ProjNames),
      TypeLemma,typelemmatac,typelemmaftac)           44 GetProofMethodX.58
 46   delaylemma3(({proj}<<|projName_1:
      projects_1-{proj} --> ProjNames),
      ExtResearchCompaniesMch,off)                    45 PrepareLemma3X.5
 47   bcall((DED;GEN;HYP)~,(GetPredicateX~;
      FwdSemanticsX~;FEQL)~: ({proj}<<|projName
      _1: projects_1-{proj} --> ProjNames) | (
      GetProofMethodX~;PrepareLemma3X)~:
      delaylemma3(({proj}<<|projName_1:
      projects_1-{proj} --> ProjNames),
      ExtResearchCompaniesMch,off))                   46 CAL
 48   bcalldelay((DED;GEN;HYP)~,(GetPredicateX~
      ;FwdSemanticsX~;FEQL)~: ({proj}<<|
      projName_1: projects_1-{proj} -->
      ProjNames) | (GetProofMethodX~;
      PrepareLemma3X)~: delaylemma3(({proj}<<|
      projName_1: projects_1-{proj} -->
      ProjNames),ExtResearchCompaniesMch,off))        47 PrepareLemma2X.1
 49   bcalldelay((DED;GEN;HYP)~,(GetPredicateX~
      ;FwdSemanticsX~;FEQL)~: ({proj}<<|
      projName_1: projects_1-{proj} -->
      ProjNames) | (GetProofMethodX~;
      PrepareLemma3X)~: delaylemma3(({proj}<<|
      projName_1: projects_1-{proj} -->
      ProjNames),currmch,off))                        48 GetProofMethodX.51
 50   bcalldelay((DED;GEN;HYP)~,(GetPredicateX~
      ;FwdSemanticsX~;FEQL)~: ({proj}<<|
```

```
      projName_1: projects_1-{proj} -->
      ProjNames) | (GetProofMethodX~;
      PrepareLemma3X)~: delaylemma3(({proj}<<|
      projName_1: projects_1-{proj} -->
      ProjNames),currmch,lemsepflag))          49 GetProofMethodX.53
 51   bcalldelay(typelemmataccall({proj}<<|
      projName_1: projects_1-{proj} -->
      ProjNames) | (GetProofMethodX~;
      PrepareLemma3X)~: delaylemma3(({proj}<<|
      projName_1: projects_1-{proj} -->
      ProjNames),currmch,lemsepflag))          50 GetProofMethodX.59
 52   delaylemma1({proj}<<|projName_1: projects
      _1-{proj} --> ProjNames)                 51 PrepareLemma2X.5
 53   delaylemma(({proj}<<|projName_1: projects
      _1-{proj} --> ProjNames),on)             52 PrepareLemma2X.6
 54   delaylemma(({proj}<<|projName_1: projects
      _1-{proj} --> ProjNames),autopflag)      53 GetProofMethodX.52
 55   bcall((GetProofMethodX~;TempX;
      PrepareLemma2X)~: delaylemma(({proj}<<|
      projName_1: projects_1-{proj} -->
      ProjNames),autopflag))                   54 CAL
 56   {proj}<<|projName_1: projects_1-{proj}
      --> ProjNames                            55 PrepareLemma1X.1
 57   delaylemma4(({proj}<<|getsGrantFrom_1:
      projects_1-{proj} --> Agencies),TypeLemma
      ,(DED;GEN;HYP)~,(GetPredicateX~;
      FwdSemanticsX~;FEQL)~)                   PrepareLemma3X.1
 58   delaylemma4(({proj}<<|getsGrantFrom_1:
      projects_1-{proj} --> Agencies),TypeLemma
      ,typelemmatac,(GetPredicateX~;
      FwdSemanticsX~;FEQL)~)                   57 GetProofMethodX.57
 59   delaylemma4(({proj}<<|getsGrantFrom_1:
      projects_1-{proj} --> Agencies),TypeLemma
      ,typelemmatac,typelemmaftac)             58 GetProofMethodX.58
 60   delaylemma3(({proj}<<|getsGrantFrom_1:
      projects_1-{proj} --> Agencies),
      ExtResearchCompaniesMch,off)             59 PrepareLemma3X.5
 61   bcall((DED;GEN;HYP)~,(GetPredicateX~;
      FwdSemanticsX~;FEQL)~: ({proj}<<|
      getsGrantFrom_1: projects_1-{proj} -->
      Agencies) | (GetProofMethodX~;
      PrepareLemma3X)~: delaylemma3(({proj}<<|
      getsGrantFrom_1: projects_1-{proj} -->
      Agencies),ExtResearchCompaniesMch,off))  60 CAL
 62   bcalldelay((DED;GEN;HYP)~,(GetPredicateX~
      ;FwdSemanticsX~;FEQL)~: ({proj}<<|
      getsGrantFrom_1: projects_1-{proj} -->
      Agencies) | (GetProofMethodX~;
```

```
      PrepareLemma3X)~: delaylemma3(({proj}<<|
      getsGrantFrom_1: projects_1-{proj} -->
      Agencies),ExtResearchCompaniesMch,off))   61 PrepareLemma2X.1
 63   bcalldelay((DED;GEN;HYP)~,(GetPredicateX~
      ;FwdSemanticsX~;FEQL)~: ({proj}<<|
      getsGrantFrom_1: projects_1-{proj} -->
      Agencies) | (GetProofMethodX~;
      PrepareLemma3X)~: delaylemma3(({proj}<<|
      getsGrantFrom_1: projects_1-{proj} -->
      Agencies),currmch,off))                   62 GetProofMethodX.51
 64   bcalldelay((DED;GEN;HYP)~,(GetPredicateX~
      ;FwdSemanticsX~;FEQL)~: ({proj}<<|
      getsGrantFrom_1: projects_1-{proj} -->
      Agencies) | (GetProofMethodX~;
      PrepareLemma3X)~: delaylemma3(({proj}<<|
      getsGrantFrom_1: projects_1-{proj} -->
      Agencies),currmch,lemsepflag))            63 GetProofMethodX.53
 65   bcalldelay(typelemmataccall({proj}<<|
      getsGrantFrom_1: projects_1-{proj} -->
      Agencies) | (GetProofMethodX~;
      PrepareLemma3X)~: delaylemma3(({proj}<<|
      getsGrantFrom_1: projects_1-{proj} -->
      Agencies),currmch,lemsepflag))            64 GetProofMethodX.59
 66   delaylemma1({proj}<<|getsGrantFrom_1:
      projects_1-{proj} --> Agencies)           65 PrepareLemma2X.5
 67   delaylemma(({proj}<<|getsGrantFrom_1:
      projects_1-{proj} --> Agencies),on)       66 PrepareLemma2X.6
 68   delaylemma(({proj}<<|getsGrantFrom_1:
      projects_1-{proj} --> Agencies),autopflag
      )                                         67 GetProofMethodX.52
 69   bcall((GetProofMethodX~;TempX;
      PrepareLemma2X)~: delaylemma(({proj}<<|
      getsGrantFrom_1: projects_1-{proj} -->
      Agencies),autopflag))                     68 CAL
 70   {proj}<<|getsGrantFrom_1: projects_1-{
      proj} --> Agencies                        69 PrepareLemma1X.1
 71   delaylemma4(({proj}<<|consortium_1:
      projects_1-{proj} --> POW(companies_1)),
      TypeLemma,(DED;GEN;HYP)~,(GetPredicateX~;
      FwdSemanticsX~;FEQL)~)                    PrepareLemma3X.1
 72   delaylemma4(({proj}<<|consortium_1:
      projects_1-{proj} --> POW(companies_1)),
      TypeLemma,typelemmatac,(GetPredicateX~;
      FwdSemanticsX~;FEQL)~)                    71 GetProofMethodX.57
 73   delaylemma4(({proj}<<|consortium_1:
      projects_1-{proj} --> POW(companies_1)),
      TypeLemma,typelemmatac,typelemmaftac)     72 GetProofMethodX.58
 74   delaylemma3(({proj}<<|consortium_1:
```

```
      projects_1-{proj} --> POW(companies_1)),
      ExtResearchCompaniesMch,off)                73 PrepareLemma3X.5
 75   bcall((DED;GEN;HYP)~,(GetPredicateX~;
      FwdSemanticsX~;FEQL)~: ({proj}<<|
      consortium_1: projects_1-{proj} --> POW(
      companies_1)) | (GetProofMethodX~;
      PrepareLemma3X)~: delaylemma3(({proj}<<|
      consortium_1: projects_1-{proj} --> POW(
      companies_1)),ExtResearchCompaniesMch,off
      ))                                          74 CAL
 76   bcalldelay((DED;GEN;HYP)~,(GetPredicateX~
      ;FwdSemanticsX~;FEQL)~: ({proj}<<|
      consortium_1: projects_1-{proj} --> POW(
      companies_1)) | (GetProofMethodX~;
      PrepareLemma3X)~: delaylemma3(({proj}<<|
      consortium_1: projects_1-{proj} --> POW(
      companies_1)),ExtResearchCompaniesMch,off
      ))                                          75 PrepareLemma2X.1
 77   bcalldelay((DED;GEN;HYP)~,(GetPredicateX~
      ;FwdSemanticsX~;FEQL)~: ({proj}<<|
      consortium_1: projects_1-{proj} --> POW(
      companies_1)) | (GetProofMethodX~;
      PrepareLemma3X)~: delaylemma3(({proj}<<|
      consortium_1: projects_1-{proj} --> POW(
      companies_1)),currmch,off))                 76 GetProofMethodX.51
 78   bcalldelay((DED;GEN;HYP)~,(GetPredicateX~
      ;FwdSemanticsX~;FEQL)~: ({proj}<<|
      consortium_1: projects_1-{proj} --> POW(
      companies_1)) | (GetProofMethodX~;
      PrepareLemma3X)~: delaylemma3(({proj}<<|
      consortium_1: projects_1-{proj} --> POW(
      companies_1)),currmch,lemsepflag))          77 GetProofMethodX.53
 79   bcalldelay(typelemmataccall({proj}<<|
      consortium_1: projects_1-{proj} --> POW(
      companies_1)) | (GetProofMethodX~;
      PrepareLemma3X)~: delaylemma3(({proj}<<|
      consortium_1: projects_1-{proj} --> POW(
      companies_1)),currmch,lemsepflag))          78 GetProofMethodX.59
 80   delaylemma1({proj}<<|consortium_1:
      projects_1-{proj} --> POW(companies_1))     79 PrepareLemma2X.5


 81   delaylemma(({proj}<<|consortium_1:
      projects_1-{proj} --> POW(companies_1)),
      on)                                         80 PrepareLemma2X.6
 82   delaylemma(({proj}<<|consortium_1:
      projects_1-{proj} --> POW(companies_1)),
      autopflag)                                  81 GetProofMethodX.52
```

```
83   bcall((GetProofMethodX~;TempX;
     PrepareLemma2X)~: delaylemma(({proj}<<|
     consortium_1: projects_1-{proj} --> POW(
     companies_1)),autopflag))                  82 CAL
84   {proj}<<|consortium_1: projects_1-{proj}
     --> POW(companies_1)                       83 PrepareLemma1X.1
85   delaylemma4((worksOn_1: employees_1 -->
     POW(projects_1-{proj})),TypeLemma,(DED;
     GEN;HYP)~,(GetPredicateX~;FwdSemanticsX~;
     FEQL)~)                                    PrepareLemma3X.1
86   delaylemma4((worksOn_1: employees_1 -->
     POW(projects_1-{proj})),TypeLemma,
     typelemmatac,(GetPredicateX~;
     FwdSemanticsX~;FEQL)~)                     85 GetProofMethodX.57
87   delaylemma4((worksOn_1: employees_1 -->
     POW(projects_1-{proj})),TypeLemma,
     typelemmatac,typelemmaftac)                86 GetProofMethodX.58
88   delaylemma3((worksOn_1: employees_1 -->
     POW(projects_1-{proj})),
     ExtResearchCompaniesMch,off)               87 PrepareLemma3X.5
89   bcall((DED;GEN;HYP)~,(GetPredicateX~;
     FwdSemanticsX~;FEQL)~: (worksOn_1:
     employees_1 --> POW(projects_1-{proj})) |
     (GetProofMethodX~;PrepareLemma3X)~:
     delaylemma3((worksOn_1: employees_1 -->
     POW(projects_1-{proj})),
     ExtResearchCompaniesMch,off))              88 CAL
90   bcalldelay((DED;GEN;HYP)~,(GetPredicateX~
     ;FwdSemanticsX~;FEQL)~: (worksOn_1:
     employees_1 --> POW(projects_1-{proj})) |
     (GetProofMethodX~;PrepareLemma3X)~:
     delaylemma3((worksOn_1: employees_1 -->
     POW(projects_1-{proj})),
     ExtResearchCompaniesMch,off))              89 PrepareLemma2X.1
91   bcalldelay((DED;GEN;HYP)~,(GetPredicateX~
     ;FwdSemanticsX~;FEQL)~: (worksOn_1:
     employees_1 --> POW(projects_1-{proj})) |
     (GetProofMethodX~;PrepareLemma3X)~:
     delaylemma3((worksOn_1: employees_1 -->
     POW(projects_1-{proj})),currmch,off))      90 GetProofMethodX.51
92   bcalldelay((DED;GEN;HYP)~,(GetPredicateX~
     ;FwdSemanticsX~;FEQL)~: (worksOn_1:
     employees_1 --> POW(projects_1-{proj})) |
     (GetProofMethodX~;PrepareLemma3X)~:
     delaylemma3((worksOn_1: employees_1 -->
     POW(projects_1-{proj})),currmch,
     lemsepflag))                               91 GetProofMethodX.53
93   bcalldelay(typelemmataccall(worksOn_1:
```

```
      employees_1 --> POW(projects_1-{proj})) |
      (GetProofMethodX~;PrepareLemma3X)~:
      delaylemma3((worksOn_1: employees_1 -->
      POW(projects_1-{proj})),currmch,
      lemsepflag))                                92 GetProofMethodX.59
 94   delaylemma1(worksOn_1: employees_1 -->
      POW(projects_1-{proj}))                     93 PrepareLemma2X.5
 95   delaylemma((worksOn_1: employees_1 -->
      POW(projects_1-{proj})),on)                 94 PrepareLemma2X.6
 96   delaylemma((worksOn_1: employees_1 -->
      POW(projects_1-{proj})),autopflag)          95 GetProofMethodX.52
 97   bcall((GetProofMethodX~;TempX;
      PrepareLemma2X)~: delaylemma((worksOn_1:
      employees_1 --> POW(projects_1-{proj})),
      autopflag))                                 96 CAL
 98   worksOn_1: employees_1 --> POW(projects_1
      -{proj})                                    97 PrepareLemma1X.1
 99       delaylemma4((x = y),MathLemma,(DED;
          GEN;HYP)~,(GetPredicateX~;
          FwdSemanticsX~;FEQL)~)                  PrepareLemma3X.1
100        delaylemma4((x = y),MathLemma,
          mathlemmatac,(GetPredicateX~;
          FwdSemanticsX~;FEQL)~)                  99 GetProofMethodX.54
101        delaylemma4((x = y),MathLemma,
          mathlemmatac,mathlemmaftac)             100 GetProofMethodX.55
102        delaylemma3((x = y),
          ExtResearchCompaniesMch,off)            101 PrepareLemma3X.4
103        bcall((DED;GEN;HYP)~,(GetPredicateX~
          ;FwdSemanticsX~;FEQL)~: (x = y) | (
          GetProofMethodX~;PrepareLemma3X)~:
          delaylemma3((x = y),
          ExtResearchCompaniesMch,off))           102 CAL
104        bcalldelay((DED;GEN;HYP)~,(
          GetPredicateX~;FwdSemanticsX~;FEQL)~:
          (x = y) | (GetProofMethodX~;
          PrepareLemma3X)~: delaylemma3((x = y)
          ,ExtResearchCompaniesMch,off))          103 PrepareLemma2X.1
105        bcalldelay((DED;GEN;HYP)~,(
          GetPredicateX~;FwdSemanticsX~;FEQL)~:
          (x = y) | (GetProofMethodX~;
          PrepareLemma3X)~: delaylemma3((x = y)
          ,currmch,off))                          104 GetProofMethodX.51
106        bcalldelay((DED;GEN;HYP)~,(
          GetPredicateX~;FwdSemanticsX~;FEQL)~:
          (x = y) | (GetProofMethodX~;
          PrepareLemma3X)~: delaylemma3((x = y)
          ,currmch,lemsepflag))                   105 GetProofMethodX.53
107        bcalldelay(mathlemmataccall(x = y) |
```

```
          (GetProofMethodX~;PrepareLemma3X)~:
          delaylemma3((x = y),currmch,
          lemsepflag))                              106 GetProofMethodX.56
108        delaylemma1(x = y)                       107 PrepareLemma2X.3
109        delaylemma((x = y),on)                   108 PrepareLemma2X.4
110        delaylemma((x = y),autopflag)            109 GetProofMethodX.52
111        bcall((GetProofMethodX~;TempX;
          PrepareLemma2X)~: delaylemma((x = y),
          autopflag))                               110 CAL
112        x = y                                    111 PrepareLemma1X.1
113       ({proj}<<|projName_1)(x) = ({proj}<<|
        projName_1)(y) & ({proj}<<|
        getsGrantFrom_1)(x) = ({proj}<<|
        getsGrantFrom_1)(y) => x = y                DED
114     x,y: projects_1-{proj} => (({proj}<<|
      projName_1)(x) = ({proj}<<|projName_1)(y)
      & ({proj}<<|getsGrantFrom_1)(x) = ({proj}
      <<|getsGrantFrom_1)(y) => x = y)              DED
115     !y.(x,y: projects_1-{proj} => (({proj}
      <<|projName_1)(x) = ({proj}<<|projName_1)
      (y) & ({proj}<<|getsGrantFrom_1)(x) = ({
      proj}<<|getsGrantFrom_1)(y) => x = y))        114 GEN
116     !x.!y.(x,y: projects_1-{proj} => (({proj
      }<<|projName_1)(x) = ({proj}<<|projName_1
      )(y) & ({proj}<<|getsGrantFrom_1)(x) = ({
      proj}<<|getsGrantFrom_1)(y) => x = y))        115 GEN
117       delaylemma4((({proj}<<|consortium_1)(x
        ) = {y | y: companies_1 & x: engagedIn_
        1(y)}),MathLemma,(DED;GEN;HYP)~,(
        GetPredicateX~;FwdSemanticsX~;FEQL)~)       PrepareLemma3X.1
118       delaylemma4((({proj}<<|consortium_1)(x
        ) = {y | y: companies_1 & x: engagedIn_
        1(y)}),MathLemma,mathlemmatac,(
        GetPredicateX~;FwdSemanticsX~;FEQL)~)       117 GetProofMethodX.54
119       delaylemma4((({proj}<<|consortium_1)(x
        ) = {y | y: companies_1 & x: engagedIn_
        1(y)}),MathLemma,mathlemmatac,
        mathlemmaftac)                              118 GetProofMethodX.55
120       delaylemma3((({proj}<<|consortium_1)(x
        ) = {y | y: companies_1 & x: engagedIn_
        1(y)}),ExtResearchCompaniesMch,off)         119 PrepareLemma3X.4
121       bcall((DED;GEN;HYP)~,(GetPredicateX~;
        FwdSemanticsX~;FEQL)~: (({proj}<<|
        consortium_1)(x) = {y | y: companies_1
        & x: engagedIn_1(y)}) | (
        GetProofMethodX~;PrepareLemma3X)~:
        delaylemma3((({proj}<<|consortium_1)(x)
        = {y | y: companies_1 & x: engagedIn_1(
```

```
        y)}),ExtResearchCompaniesMch,off))      120 CAL
122      bcalldelay((DED;GEN;HYP)~,(
        GetPredicateX~;FwdSemanticsX~;FEQL)~: (
        ({proj}<<|consortium_1)(x) = {y | y:
        companies_1 & x: engagedIn_1(y)}) | (
        GetProofMethodX~;PrepareLemma3X)~:
        delaylemma3((({proj}<<|consortium_1)(x)
        = {y | y: companies_1 & x: engagedIn_1(
        y)}),ExtResearchCompaniesMch,off))      121 PrepareLemma2X.1
123      bcalldelay((DED;GEN;HYP)~,(
        GetPredicateX~;FwdSemanticsX~;FEQL)~: (
        ({proj}<<|consortium_1)(x) = {y | y:
        companies_1 & x: engagedIn_1(y)}) | (
        GetProofMethodX~;PrepareLemma3X)~:
        delaylemma3((({proj}<<|consortium_1)(x)
        = {y | y: companies_1 & x: engagedIn_1(
        y)}),currmch,off))                      122 GetProofMethodX.51
124      bcalldelay((DED;GEN;HYP)~,(
        GetPredicateX~;FwdSemanticsX~;FEQL)~: (
        ({proj}<<|consortium_1)(x) = {y | y:
        companies_1 & x: engagedIn_1(y)}) | (
        GetProofMethodX~;PrepareLemma3X)~:
        delaylemma3((({proj}<<|consortium_1)(x)
        = {y | y: companies_1 & x: engagedIn_1(
        y)}),currmch,lemsepflag))               123 GetProofMethodX.53
125      bcalldelay(mathlemmataccall(({proj}<<|
        consortium_1)(x) = {y | y: companies_1
        & x: engagedIn_1(y)}) | (
        GetProofMethodX~;PrepareLemma3X)~:
        delaylemma3((({proj}<<|consortium_1)(x)
        = {y | y: companies_1 & x: engagedIn_1(
        y)}),currmch,lemsepflag))               124 GetProofMethodX.56
126      delaylemma1(({proj}<<|consortium_1)(x)
        = {y | y: companies_1 & x: engagedIn_1(
        y)})                                    125 PrepareLemma2X.3
127      delaylemma((({proj}<<|consortium_1)(x)
        = {y | y: companies_1 & x: engagedIn_1(
        y)}),on)                                126 PrepareLemma2X.4
128      delaylemma((({proj}<<|consortium_1)(x)
        = {y | y: companies_1 & x: engagedIn_1(
        y)}),autopflag)                         127 GetProofMethodX.52
129      bcall((GetProofMethodX~;TempX;
        PrepareLemma2X)~: delaylemma((({proj}
        <<|consortium_1)(x) = {y | y: companies
        _1 & x: engagedIn_1(y)}),autopflag))    128 CAL
130      ({proj}<<|consortium_1)(x) = {y | y:
        companies_1 & x: engagedIn_1(y)}        129 PrepareLemma1X.1
131    x: projects_1-{proj} => ({proj}<<|
```

```
      consortium_1)(x) = {y | y: companies_1 &
      x: engagedIn_1(y)}                          DED
 132   !x.(x: projects_1-{proj} => ({proj}<<|
      consortium_1)(x) = {y | y: companies_1 &
      x: engagedIn_1(y)})                         131 GEN
 133   companies_1: POW(COMPANIES) & compName_1
      : companies_1 --> CompNames & engagedIn_1
      : companies_1 --> POW(projects_1-{proj})
      & budget_1: companies_1 --> MONEY &
      projects_1-{proj}: POW(PROJECTS) & {proj}
      <<|projName_1: projects_1-{proj} -->
      ProjNames & {proj}<<|getsGrantFrom_1:
      projects_1-{proj} --> Agencies & {proj}
      <<|consortium_1: projects_1-{proj} -->
      POW(companies_1) & employees_1: POW(
      EMPLOYEES) & empName_1: employees_1 -->
      EmpNames & belongsTo_1: employees_1 -->
      companies_1 & worksOn_1: employees_1 -->
      POW(projects_1-{proj}) & !x.!y.(x,y:
      projects_1-{proj} => (({proj}<<|projName_
      1)(x) = ({proj}<<|projName_1)(y) & ({proj
      }<<|getsGrantFrom_1)(x) = ({proj}<<|
      getsGrantFrom_1)(y) => x = y)) & !x.!y.(x
      ,y: companies_1 => (compName_1(x) =
      compName_1(y) => x = y)) & !x.(x:
      employees_1 => worksOn_1(x) incl
      engagedIn_1(belongsTo_1(x))) & !x.(x:
      projects_1-{proj} => ({proj}<<|consortium
      _1)(x) = {y | y: companies_1 & x:
      engagedIn_1(y)})                            4 5 28 6 42 56 70 84 7
                                                  8 9 98 116 10 11 132 AND
 134   [projName_1,consortium_1,getsGrantFrom_1
      ,projects_1:={proj}<<|projName_1,{proj}
      <<|consortium_1,{proj}<<|getsGrantFrom_1,
      projects_1-{proj}](companies_1: POW(
      COMPANIES) & compName_1: companies_1 -->
      CompNames & engagedIn_1: companies_1 -->
      POW(projects_1) & budget_1: companies_1
      --> MONEY & projects_1: POW(PROJECTS) &
      projName_1: projects_1 --> ProjNames &
      getsGrantFrom_1: projects_1 --> Agencies
      & consortium_1: projects_1 --> POW(
      companies_1) & employees_1: POW(EMPLOYEES
      ) & empName_1: employees_1 --> EmpNames &
      belongsTo_1: employees_1 --> companies_1
      & worksOn_1: employees_1 --> POW(projects
      _1) & !x.!y.(x,y: projects_1 => (projName
      _1(x) = projName_1(y) & getsGrantFrom_1(x
```

```
      ) = getsGrantFrom_1(y) => x = y)) & !x.!y
      .(x,y: companies_1 => (compName_1(x) =
      compName_1(y) => x = y)) & !x.(x:
      employees_1 => worksOn_1(x) incl
      engagedIn_1(belongsTo_1(x))) & !x.(x:
      projects_1 => consortium_1(x) = {y | y:
      companies_1 & x: engagedIn_1(y)}))        133 SUB
135    chkinv((companies_1: POW(COMPANIES) &
      compName_1: companies_1 --> CompNames &
      engagedIn_1: companies_1 --> POW(projects
      _1) & budget_1: companies_1 --> MONEY &
      projects_1: POW(PROJECTS) & projName_1:
      projects_1 --> ProjNames & getsGrantFrom_
      1: projects_1 --> Agencies & consortium_1
      : projects_1 --> POW(companies_1) &
      employees_1: POW(EMPLOYEES) & empName_1:
      employees_1 --> EmpNames & belongsTo_1:
      employees_1 --> companies_1 & worksOn_1:
      employees_1 --> POW(projects_1) & !x.!y.(
      x,y: projects_1 => (projName_1(x) =
      projName_1(y) & getsGrantFrom_1(x) =
      getsGrantFrom_1(y) => x = y)) & !x.!y.(x,
      y: companies_1 => (compName_1(x) =
      compName_1(y) => x = y)) & !x.(x:
      employees_1 => worksOn_1(x) incl
      engagedIn_1(belongsTo_1(x))) & !x.(x:
      projects_1 => consortium_1(x) = {y | y:
      companies_1 & x: engagedIn_1(y)})),(proj:
      projects_1 & consortium_1(proj) = {} |
      projName_1,consortium_1,getsGrantFrom_1,
      projects_1:={proj}<<|projName_1,{proj}<<|
      consortium_1,{proj}<<|getsGrantFrom_1,
      projects_1-{proj}))                       134 BuildConjectureX.2
136    chkinv(inv(ExtResearchCompaniesMch_1),(
      proj: projects_1 & consortium_1(proj) =
      {} | projName_1,consortium_1,
      getsGrantFrom_1,projects_1:={proj}<<|
      projName_1,{proj}<<|consortium_1,{proj}
      <<|getsGrantFrom_1,projects_1-{proj}))    135 GetInvariantX.2
137    chkinv(inv(ExtResearchCompaniesMch_1),
      ExtResearchCompaniesMchremoveProject_1(
      proj))                                    136 GetOperationX.1
138  ctx(ExtResearchCompaniesMch) & inv(
    ExtResearchCompaniesMch_1) & pre(
    ExtResearchCompaniesMchremoveProject_1(proj
    )) & asn(ExtResearchCompaniesMch_1) =>
    chkinv(inv(ExtResearchCompaniesMch_1),
    ExtResearchCompaniesMchremoveProject_1(proj
```

```
     ))                                          DED
  139  Check_operation: (ExtResearchCompaniesMch_
     1: removeProject)                           138 InvariantLemmaX.2

END OF PROOF
```

G.2 Beweise ausgesuchter Lemmas

Für die abgebildeten Lemmabeweise wird jeweils angegeben:

- die *generische Klasse* des bewiesenen Lemmas (vgl. Abschnitt 7.5), sofern eine Einordnung vorliegt.
- die verwendete, hinreichende FWD-Quote (Anzahl der nach jeder HYP-Anwendung gemäß der Vorwärtstaktik durchgeführten Vorwärtsregelanwendungen), sofern von Belang

Falls der Beweis unter Benutzerassistenz durchgeführt wurde, so wird dies ebenfalls erwähnt. Die Beweislistings zeigen jeweils nur die Hypothesenformeln ang, die zu dem entsprechenden Beweis *beitragen*.

G.2.1 Typlemmas

G.2.1.1 Typlemma Nr. 2

Lemmaklasse: **TI**.

```
PROOF

   1    {}: {} --> CompNames                    empty.2
   2  ctx(ExtResearchCompaniesMch) => {}: {} -->
      CompNames                                 DED

END OF PROOF
```

G.2.1.2 Typlemma Nr. 14

Lemmaklasse: **TL1**.
FWD-Quote: 100

```
PROOF

   1  inv(ExtResearchCompaniesMch_1)              HYP
   2    projects_1: POW(PROJECTS)                 1 GetPredicateX.2
   3    projects_1-{proj}: POW(PROJECTS)          2 support.34
   4  ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_1) =>
      projects_1-{proj}: POW(PROJECTS)           DED

END OF PROOF
```

G.2.1.3 Typlemma Nr. 20

Lemmaklasse: **TL2**.
FWD-Quote: 50

```
PROOF

   1  inv(ExtResearchCompaniesMch_1)              HYP
   2    empName_1: employees_1 --> EmpNames       1 GetPredicateX.2
   3    {empl}<<|empName_1: employees_1-{empl}
        --> EmpNames                              2 support.16
   4  ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_1) =>
      {empl}<<|empName_1: employees_1-{empl} -->
      EmpNames                                    DED

END OF PROOF
```

G.2.1.4 Typlemma Nr. 23

Lemmaklasse: **TE1**.
FWD-Quote: 50

```
PROOF

   1  inv(ExtResearchCompaniesMch_1)              HYP
   2  co: COMPANIES-companies_1                   HYP
   3    companies_1: POW(COMPANIES)               1 GetPredicateX.2
   4    co: COMPANIES                             2 fsupport.40
   5    companies_1\/{co}: POW(COMPANIES)         4 3 support.33
   6  ctx(ExtResearchCcompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_1) & co: COMPANIES-
      companies_1 => companies_1\/{co}: POW(
      COMPANIES)                                  DED

END OF PROOF
```

G.2.1.5 Typlemma Nr. 32

Lemmaklasse: **TE2**.
FWD-Quote: 150

```
PROOF

   1  inv(ExtResearchCompaniesMch_1)              HYP
   2  pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works))                     HYP
   3  ee: EMPLOYEES-employees_1                   HYP
   4    engagedIn_1: companies_1 --> POW(projects
      _1)                                         1 GetPredicateX.2
   5    employees_1: POW(EMPLOYEES)               1 GetPredicateX.2
   6    worksOn_1: employees_1 --> POW(projects_1
      )                                           1 GetPredicateX.2
   7    belongs: companies_1                      2 GetPredicateX.6
   8    works: POW(engagedIn_1(belongs))          2 GetPredicateX.6
   9    engagedIn_1(belongs): POW(projects_1)     4 7 fsupport.24
  10    works: POW(projects_1)                    8 9 fsupport.23
  11    worksOn_1<+{ee|->works}: employees_1\/{ee
        } --> POW(projects_1)                     3 5 10 6 support.10
  12  ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
```

```
      ExtResearchCompaniesMch_1) & ee: EMPLOYEES-
      employees_1 => worksOn_1<+{ee|->works}:
      employees_1\/{ee} --> POW(projects_1)      DED

END OF PROOF
```

G.2.1.6 Typlemma Nr. 34

Lemmaklasse: **TÄ**.
FWD-Quote: 150

```
PROOF

   1  inv(ExtResearchCompaniesMch_1)             HYP
   2  pre(ExtResearchCompaniesMchupdateEmployer_1
      (empl,belongs,works))                      HYP
   3    engagedIn_1: companies_1 --> POW(projects
      _1)                                        1 GetPredicateX.2
   4    worksOn_1: employees_1 --> POW(projects_1
      )                                          1 GetPredicateX.2
   5    empl: employees_1                         2 GetPredicateX.7
   6    belongs: companies_1                      2 GetPredicateX.7
   7    works: POW(engagedIn_1(belongs))          2 GetPredicateX.7
   8    engagedIn_1(belongs): POW(projects_1)     3 6 fsupport.24
   9    works: POW(projects_1)                    7 8 fsupport.23
  10    worksOn_1<+{empl|->works}: employees_1
        --> POW(projects_1)                       4 5 9 support.8
  11  ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_1) => worksOn_1<+{
      empl|->works}: employees_1 --> POW(projects
      _1)                                        DED

END OF PROOF
```

G.2.1.7 Typlemma Nr. 44

Lemmaklasse: **TV2L**.
FWD-Quote: unbedeutend, 10 reicht

```
PROOF

   1  inv(ExtResearchCompaniesMch_3)              HYP
   2    projClass_3: projects_3 --> POW(companies
      _3)                                        1 GetPredicateX.19
   3    projClass_3: dom(projClass_3) --> POW(
        companies_3)                             2 GetInvDefinitionX.6
   4    {proj}<<|projClass_3: dom(projClass_3)-{
        proj} --> POW(companies_3)               3 support.16
   5    {proj}<<|projClass_3: dom({proj}<<|
        projClass_3) --> POW(companies_3)        4 refinementsupport.5
   6  ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_3) => {
      proj}<<|projClass_3: dom({proj}<<|projClass
      _3) --> POW(companies_3)                   DED

END OF PROOF
```

G.2.1.8 Typlemma Nr. 50

Lemmaklasse: **TV2E**.
Aus Effizienzgründen wurde ein für dieses Lemma erzeugtes Teilziel "von Hand" als wahr reduziert (Beweiszeile 23). Es ist zwar prinzipiell möglich, dieses Teilziel per FWD-Deduktion aus der Hypothese abzuleiten, doch erfordert dies eine hohe FWD-Quote (500 Schritte reichten *nicht*) und viel Rechenzeit.

```
PROOF

   1  ctx(ExtResearchCompaniesMch)                HYP
   2  inv(ExtResearchCompaniesMch_3)              HYP
   3  pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works))                    HYP
   4    ctx(ExtResearchCompaniesCtx4)             1 GetPredicateX.27
   5    inv(ExtResearchCompaniesMch_2)            2 GetPredicateX.20
   6    empClass_3: employees_3 --> EmpNames*
      companies_3*POW(projects_3)                2 GetPredicateX.19
   7    companies_2 = companies_3                 2 GetPredicateX.19
   8    newname: EmpNames                         3 GetPredicateX.6
   9    belongs: companies_1                      3 GetPredicateX.6
  10    EmpNames = Strings                        4 FwdSemanticsX.101
```

```
11    companies_1 = companies_2                 5 GetPredicateX.10
12    companies_1 = companies_3                 7 11 fsupport.63
13    newname: Strings                          8 10 fsupport.61
14    belongs: companies_3                      9 12 fsupport.61
15    employees_3 = employees_3                 EQL
16    employees_3 = dom(empClass_3)             15 GetInvDefinitionX.7
17    Strings*companies_3*POW(projects_3) =
      Strings*companies_3*POW(projects_3)       EQL
18    EmpNames*companies_3*POW(projects_3) =
      Strings*companies_3*POW(projects_3)       17 HYP.10
19    EmpNames*companies_3*POW(projects_3) =
      Strings*companies_3*POW(dom(projClass_3)) 18 GetInvDefinitionX.6
20    EmpNames*companies_3*POW(projects_3) =
      Strings*dom(compClass_3)*POW(dom(
      projClass_3))                             19 GetInvDefinitionX.8
21    belongs: dom(compClass_3)                 14 GetInvDefinitionX.8
22    newname,belongs: Strings*dom(compClass_3) 13 21 refinementsupport.3
23    works: POW(projects_3)                    LEM
24    works: POW(dom(projClass_3))              23 GetInvDefinitionX.6
25    newname,belongs,works: Strings*dom(
      compClass_3)*POW(dom(projClass_3))        22 24 refinementsupport.3
26    tail(newEmpId,newname),belongs,works:
      Strings*dom(compClass_3)*POW(dom(
      projClass_3))                             25 refinementsupport.17
27    tail(newEmpId,newname,belongs),works:
      Strings*dom(compClass_3)*POW(dom(
      projClass_3))                             26 refinementsupport.18
28    tail(newEmpId,newname,belongs,works):
      Strings*dom(compClass_3)*POW(dom(
      projClass_3))                             27 refinementsupport.18
29    tail(newEmpId,newname,belongs,works):
      Strings*dom(compClass_3)*POW(projects_3)  28 GetDefinitionX.6
30    tail(newEmpId,newname,belongs,works):
      Strings*companies_3*POW(projects_3)       29 GetDefinitionX.8
31    tail(newEmpId,newname,belongs,works):
      EmpNames*companies_3*POW(projects_3)      30 HYP.10
32    empClass_3\/{newEmpId,newname,belongs,
      works}: dom(empClass_3\/{newEmpId,newname
      ,belongs,works}) --> Strings*dom(
      compClass_3)*POW(dom(projClass_3))        6 16 20 31 refinementsupport.7
33  ctx(ExtResearchCompaniesMch) & inv(
    ExtResearchCompaniesMch_3) & pre(e <--
    ExtResearchCompaniesMchnewEmployee_1(
    newname,belongs,works)) & asn(
    ExtResearchCompaniesMch_3) => empClass_3\/{
    newEmpId,newname,belongs,works}: dom(
    empClass_3\/{newEmpId,newname,belongs,works
    }) --> Strings*dom(compClass_3)*POW(dom(
```

```
      projClass_3))                                    DED

END OF PROOF
```

G.2.1.9 Typlemma Nr. 52

Lemmaklasse: **TV2Ä**.
FWD-Quote: 500.

```
PROOF

   1  ctx(ExtResearchCompaniesMch)                    HYP
   2  inv(ExtResearchCompaniesMch_3)                  HYP
   3  pre(ExtResearchCompaniesMchupdateEmployer_1
      (empl,belongs,works))                           HYP
   4    ctx(ExtResearchCompaniesCtx4)                 1 GetPredicateX.27
   5    inv(ExtResearchCompaniesMch_2)                2 GetPredicateX.20
   6    empClass_3: employees_3 --> EmpNames*
      companies_3*POW(projects_3)                     2 GetPredicateX.19
   7    projects_2 = projects_3                       2 GetPredicateX.19
   8    companies_2 = companies_3                     2 GetPredicateX.19
   9    belongs: companies_1                          3 GetPredicateX.7
  10    works: POW(engagedIn_1(belongs))              3 GetPredicateX.7
  11    EmpNames = Strings                            4 FwdSemanticsX.101
  12    inv(ExtResearchCompaniesMch_1)                5 GetPredicateX.11
  13    projects_1 = projects_2                       5 GetPredicateX.10
  14    companies_1 = companies_2                     5 GetPredicateX.10
  15    projects_1 = projects_3                       7 13 fsupport.63
  16    companies_1 = companies_3                     8 14 fsupport.63
  17    belongs: companies_3                          9 16 fsupport.61
  18    engagedIn_1: companies_1 --> POW(projects
      _1)                                             12 GetPredicateX.2
  19    engagedIn_1(belongs): POW(projects_1)         18 9 fsupport.24
  20    engagedIn_1(belongs): POW(projects_3)         19 15 fsupport.60
  21    works: POW(projects_3)                        20 10 fsupport.22
  22    employees_3 = employees_3                     EQL
  23    employees_3 = dom(empClass_3)                 22 GetInvDefinitionX.7
  24    Strings*companies_3*POW(projects_3) =
        Strings*companies_3*POW(projects_3)           EQL
  25    EmpNames*companies_3*POW(projects_3) =
        Strings*companies_3*POW(projects_3)           24 HYP.11
  26    EmpNames*companies_3*POW(projects_3) =
        Strings*companies_3*POW(dom(projClass_3))     25 GetInvDefinitionX.6
  27    EmpNames*companies_3*POW(projects_3) =
```

```
       Strings*dom(compClass_3)*POW(dom(
       projClass_3))                                26 GetInvDefinitionX.8
  28   okproj(empl): Strings                        refinementsupport.15
  29   belongs: dom(compClass_3)                    17 GetInvDefinitionX.8
  30   okproj(empl),belongs: Strings*dom(
       compClass_3)                                 28 29 refinementsupport.3
  31   works: POW(dom(projClass_3))                 21 GetInvDefinitionX.6
  32   okproj(empl),belongs,works: Strings*dom(
       compClass_3)*POW(dom(projClass_3))           30 31 refinementsupport.3
  33   okproj(empl),belongs,works: Strings*dom(
       compClass_3)*POW(projects_3)                 32 GetDefinitionX.6
  34   okproj(empl),belongs,works: Strings*
       companies_3*POW(projects_3)                  33 GetDefinitionX.8
  35   okproj(empl),belongs,works: EmpNames*
       companies_3*POW(projects_3)                  34 HYP.11
  36   empClass_3<+{empl|->(okproj(empl),belongs
       ,works)}: dom(empClass_3<+{empl|->(okproj
       (empl),belongs,works)}) --> Strings*dom(
       compClass_3)*POW(dom(projClass_3))           6 23 27 35 refinementsupport.8
  37   empClass_3<+{empl|->((%x.(x: dom(empClass
       _3) & empClass_3(x) = a,b,c | a))(empl),
       belongs,works)}: dom(empClass_3<+{empl|->
       ((%x.(x: dom(empClass_3) & empClass_3(x)
       = a,b,c | a))(empl),belongs,works)}) -->
       Strings*dom(compClass_3)*POW(dom(
       projClass_3))                                36 refinementsupport.12
  38   ctx(ExtResearchCompaniesMch) & inv(
       ExtResearchCompaniesMch_3) & pre(
       ExtResearchCompaniesMchupdateEmployer_1(
       empl,belongs,works)) & asn(
       ExtResearchCompaniesMch_3) => empClass_3<+{
       empl|->((%x.(x: dom(empClass_3) & empClass_
       3(x) = a,b,c | a))(empl),belongs,works)}:
       dom(empClass_3<+{empl|->((%x.(x: dom(
       empClass_3) & empClass_3(x) = a,b,c | a))(
       empl),belongs,works)}) --> Strings*dom(
       compClass_3)*POW(dom(projClass_3))           DED

END OF PROOF
```

G.2.1.10 Typlemma Nr. 53

Lemmaklasse: **TV2Ä**.
Der Beweis für dieses Lemma wurde *nicht vollständig automatisch* erzeugt: für das Teilziel `works: POW(projects3)` (Zeile 19) wurde der Beweisweg *benutzerassistiert* ermittelt. Dies zeigt,

daß es selbst innerhalb einer Lemmaklasse zu unterschiedlichem Beweisverhalten kommen kann: der zuvor gezeigte Beweis gehört zu einem Lemma derselben Klasse, läuft jedoch *ohne* Benutzerassistenz.
FWD-Quote: 700

```
PROOF

  1  ctx(ExtResearchCompaniesMch)                HYP
  2  inv(ExtResearchCompaniesMch_3)              HYP
  3  pre(ExtResearchCompaniesMchupdateJobs_1(
     empl,works))                                HYP
  4    ctx(ExtResearchCompaniesCtx4)             1 GetPredicateX.27
  5    inv(ExtResearchCompaniesMch_2)            2 GetPredicateX.20
  6    empClass_3: employees_3 --> EmpNames*
     companies_3*POW(projects_3)                 2 GetPredicateX.19
  7    projects_2 = projects_3                   2 GetPredicateX.19
  8    empl: employees_1                         3 GetPredicateX.8
  9    works: POW(engagedIn_1(belongsTo_1(empl))
     )                                           3 GetPredicateX.8
 10    EmpNames = Strings                        4 FwdSemanticsX.101
 11    inv(ExtResearchCompaniesMch_1)            5 GetPredicateX.11
 12    projects_1 = projects_2                   5 GetPredicateX.10
 13    projects_1 = projects_3                   7 12 fsupport.63
 14    engagedIn_1: companies_1 --> POW(projects
     _1)                                         11 GetPredicateX.2
 15    belongsTo_1: employees_1 --> companies_1  11 GetPredicateX.2
 16    belongsTo_1(empl): companies_1            15 8 fsupport.7
 17    engagedIn_1(belongsTo_1(empl)): POW(
     projects_1)                                 16 14 fsupport.25
 18    engagedIn_1(belongsTo_1(empl)): POW(
     projects_3)                                 17 13 fsupport.60
 19    works: POW(projects_3)                    18 9 fsupport.22
 20    employees_3 = employees_3                 EQL
 21    employees_3 = dom(empClass_3)             20 GetInvDefinitionX.7
 22    Strings*companies_3*POW(projects_3) =
       Strings*companies_3*POW(projects_3)       EQL
 23    EmpNames*companies_3*POW(projects_3) =
       Strings*companies_3*POW(projects_3)       22 HYP.10
 24    EmpNames*companies_3*POW(projects_3) =
       Strings*companies_3*POW(dom(projClass_3)) 23 GetInvDefinitionX.6
 25    EmpNames*companies_3*POW(projects_3) =
       Strings*dom(compClass_3)*POW(dom(
       projClass_3))                             24 GetInvDefinitionX.8
 26    okproj(empl): Strings                     refinementsupport.15
 27    okproj(empl): dom(compClass_3)            refinementsupport.15
```

```
 28    okproj(empl),okproj(empl): Strings*dom(
       compClass_3)                                26 27 refinementsupport.3
 29    works: POW(dom(projClass_3))                19 GetInvDefinitionX.6
 30    okproj(empl),okproj(empl),works: Strings*
       dom(compClass_3)*POW(dom(projClass_3))      28 29 refinementsupport.3
 31    okproj(empl),okproj(empl),works: Strings*
       dom(compClass_3)*POW(projects_3)            30 GetDefinitionX.6
 32    okproj(empl),okproj(empl),works: Strings*
       companies_3*POW(projects_3)                 31 GetDefinitionX.8
 33    okproj(empl),okproj(empl),works: EmpNames
       *companies_3*POW(projects_3)                32 HYP.10
 34    empClass_3<+{empl|->(okproj(empl),okproj(
       empl),works)}: dom(empClass_3<+{empl|->(
       okproj(empl),okproj(empl),works)}) -->
       Strings*dom(compClass_3)*POW(dom(
       projClass_3))                               6 21 25 33 refinementsupport.8
 35    empClass_3<+{empl|->(okproj(empl),(%x.(x:
       dom(empClass_3) & empClass_3(x) = a,b,c |
       b))(empl),works)}: dom(empClass_3<+{empl
       |->(okproj(empl),(%x.(x: dom(empClass_3)
       & empClass_3(x) = a,b,c | b))(empl),works
       )}) --> Strings*dom(compClass_3)*POW(dom(
       projClass_3))                               34 refinementsupport.11
 36    empClass_3<+{empl|->((%x.(x: dom(empClass
       _3) & empClass_3(x) = a,b,c | a))(empl),(
       %x.(x: dom(empClass_3) & empClass_3(x) =
       a,b,c | b))(empl),works)}: dom(empClass_3
       <+{empl|->((%x.(x: dom(empClass_3) &
       empClass_3(x) = a,b,c | a))(empl),(%x.(x:
       dom(empClass_3) & empClass_3(x) = a,b,c |
       b))(empl),works)}) --> Strings*dom(
       compClass_3)*POW(dom(projClass_3))          35 refinementsupport.12
 37  ctx(ExtResearchCompaniesMch) & inv(
     ExtResearchCompaniesMch_3) & pre(
     ExtResearchCompaniesMchupdateJobs_1(empl,
     works)) & asn(ExtResearchCompaniesMch_3) =>
     empClass_3<+{empl|->((%x.(x: dom(empClass_3
     ) & empClass_3(x) = a,b,c | a))(empl),(%x.(
     x: dom(empClass_3) & empClass_3(x) = a,b,c
     | b))(empl),works)}: dom(empClass_3<+{empl
     |->((%x.(x: dom(empClass_3) & empClass_3(x)
     = a,b,c | a))(empl),(%x.(x: dom(empClass_3)
     & empClass_3(x) = a,b,c | b))(empl),works)}
     ) --> Strings*dom(compClass_3)*POW(dom(
     projClass_3))                               DED

END OF PROOF
```

G.2.2 Mathematische Lemmas

G.2.2.1 Mathematisches Lemma Nr. 2

Lemmaklasse: **MIW**.
FWD-Quote: 50

```
PROOF

   1  x,y: {}                                        HYP
   2    FALSE                                        1 fempty.1
   3    x = y                                        2 exfalsoquodlibet.1
   4  ctx(ExtResearchCompaniesMch) & x,y: {} & {}
      (x) = {}(y) => x = y                           DED

END OF PROOF
```

G.2.2.2 Mathematisches Lemma Nr. 5

Lemmaklasse: **MSL**.
FWD-Quote: 50

```
PROOF

   1  inv(ExtResearchCompaniesMch_1)                 HYP
   2  x,y: projects_1-{proj}                         HYP
   3  ({proj}<<|projName_1)(x) = ({proj}<<|
      projName_1)(y)                                 HYP
   4  ({proj}<<|getsGrantFrom_1)(x) = ({proj}<<|
      getsGrantFrom_1)(y)                            HYP
   5    !x.!y.(x,y: projects_1 => (projName_1(x)
      = projName_1(y) & getsGrantFrom_1(x) =
      getsGrantFrom_1(y) => x = y))                  1 GetPredicateX.2
   6    x = y                                        5 2 3 4 support.1
   7  ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_1) & x,y:
      projects_1-{proj} & ({proj}<<|projName_1)(x
```

```
      ) = ({proj}<<|projName_1)(y) & ({proj}<<|
      getsGrantFrom_1)(x) = ({proj}<<|
      getsGrantFrom_1)(y) => x = y                DED

END OF PROOF
```

G.2.2.3 Mathematisches Lemma Nr. 7

Lemmaklasse: *nicht klassifiziert, spezielle Gestalt*; das Lemma resultiert aus einem anwendungsspezifischen Integritätsprädikat.
FWD-Quote: 150

```
PROOF

   1  inv(ExtResearchCompaniesMch_1)             HYP
   2  x: employees_1-{empl}                      HYP
   3    belongsTo_1: employees_1 --> companies_1  1 GetPredicateX.2
   4    worksOn_1: employees_1 --> POW(projects_1
      )                                          1 GetPredicateX.2
   5    !x.(x: employees_1 => worksOn_1(x) incl
      engagedIn_1(belongsTo_1(x)))               1 GetPredicateX.2
   6    worksOn_1(x) incl engagedIn_1(belongsTo_1
      (x))                                       2 5 specialize.11
   7    dom(belongsTo_1) = employees_1             3 fsupport.2
   8    dom(worksOn_1) = employees_1               4 fsupport.2
   9    ({empl}<<|belongsTo_1)(x) = belongsTo_1(x
      )                                          7 2 fsupport.9
  10    ({empl}<<|worksOn_1)(x) = worksOn_1(x)     8 2 fsupport.9
  11    ({empl}<<|worksOn_1)(x) incl engagedIn_1(
        belongsTo_1(x))                          6 HYP.10
  12    ({empl}<<|worksOn_1)(x) incl engagedIn_1(
        ({empl}<<|belongsTo_1)(x))               11 HYP.9
  13  ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_1) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_1) & x
      : employees_1-{empl} => ({empl}<<|worksOn_1
      )(x) incl engagedIn_1(({empl}<<|belongsTo_1
      )(x))                                      DED

END OF PROOF
```

G.2.2.4 Mathematisches Lemma Nr. 11

Lemmaklasse: *nicht klassifiziert, spezielle Gestalt*; das Lemma resultiert aus einem anwendungsspezifischen Integritätsprädikat.
Der abgebildete Beweis erfolgte mit teilweiser Benutzerassistenz (Auswahl vom B-Tool vorgeschlagener anwendbarer Regeln durch den Benutzer).

```
PROOF

 1  inv(ExtResearchCompaniesMch_1)                HYP
 2  pre(e <--
    ExtResearchCompaniesMchnewEmployee_1(
    newname,belongs,works))                       HYP
 3  ee: EMPLOYEES-employees_1                     HYP
 4  x: employees_1\/{ee}                          HYP
 5    engagedIn_1: companies_1 --> POW(projects
    _1)                                           1 GetPredicateX.2
 6    belongsTo_1: employees_1 --> companies_1    1 GetPredicateX.2
 7    worksOn_1: employees_1 --> POW(projects_1
    )                                             1 GetPredicateX.2
 8    !x.(x: employees_1 => worksOn_1(x) incl
    engagedIn_1(belongsTo_1(x)))                  1 GetPredicateX.2
 9    belongs: companies_1                        2 GetPredicateX.6
10    works: POW(engagedIn_1(belongs))            2 GetPredicateX.6
11    engagedIn_1(belongs): POW(projects_1)       5 9 fsupport.24
12    works incl engagedIn_1(belongs)             10 fsupport.44
13    works: POW(projects_1)                      10 11 fsupport.23
14    x: employees_1                              HYP
15      (worksOn_1<+{ee|->works})(x) = worksOn_
      1(x)                                        14 3 7 13 fsupport.10
16      (belongsTo_1<+{ee|->belongs})(x) =
      belongsTo_1(x)                              14 3 6 9 fsupport.10
17      worksOn_1(x) incl engagedIn_1(belongsTo
      _1(x))                                      14 8 specialize.9
18      worksOn_1(x) incl engagedIn_1((
        belongsTo_1<+{ee|->belongs})(x))          17 HYP.16
19      (worksOn_1<+{ee|->works})(x) incl
        engagedIn_1((belongsTo_1<+{ee|->belongs
        })(x))                                    18 HYP.15
20    x: employees_1 => (worksOn_1<+{ee|->works
      })(x) incl engagedIn_1((belongsTo_1<+{ee
      |->belongs})(x))                            DED
21    x: {ee}                                     HYP
22      (worksOn_1<+{ee|->works})(x) = works      21 13 7 fsupport.32
23      (belongsTo_1<+{ee|->belongs})(x) =
      belongs                                     21 9 6 fsupport.32
```

```
 24      works incl engagedIn_1((belongsTo_1<+{
         ee|->belongs})(x))                        12 HYP.23
 25      (worksOn_1<+{ee|->works})(x) incl
         engagedIn_1((belongsTo_1<+{ee|->belongs
         })(x))                                    24 HYP.22
 26    x: {ee} => (worksOn_1<+{ee|->works})(x)
       incl engagedIn_1((belongsTo_1<+{ee|->
       belongs})(x))                               DED
 27    (worksOn_1<+{ee|->works})(x) incl
       engagedIn_1((belongsTo_1<+{ee|->belongs})
       (x))                                        4 20 26 support.7
 28  ctx(ExtResearchCompaniesMch) & inv(
     ExtResearchCompaniesMch_1) & pre(e <--
     ExtResearchCompaniesMchnewEmployee_1(
     newname,belongs,works)) & asn(
     ExtResearchCompaniesMch_1) & ee: EMPLOYEES-
     employees_1 & x: employees_1\/{ee} => (
     worksOn_1<+{ee|->works})(x) incl engagedIn_
     1((belongsTo_1<+{ee|->belongs})(x))          DED

END OF PROOF
```

G.2.2.5 Mathematisches Lemma Nr. 12

Lemmaklasse: *nicht klassifiziert, spezielle Gestalt*; das Lemma resultiert aus einem anwendungsspezifischen Integritätsprädikat.
Der abgebildete Beweis erfolgte mit teilweiser Benutzerassistenz (Auswahl vom B-Tool vorgeschlagener anwendbarer Regeln durch den Benutzer).

```
  1  inv(ExtResearchCcompaniesMch_1)              HYP
  2  pre(ExtResearchCompaniesMchupdateEmployer_1
     (empl,belongs,works))                        HYP
  3  x: employees_1                               HYP
  4    belongsTo_1: employees_1 --> companies_1   1 GetPredicateX.2
  5    !x.(x: employees_1 => worksOn_1(x) incl
     engagedIn_1(belongsTo_1(x)))                 1 GetPredicateX.2
  6    empl: employees_1                          2 GetPredicateX.7
  7    belongs: companies_1                       2 GetPredicateX.7
  8    worksOn_1(x) incl engagedIn_1(belongsTo_1
     (x))                                         3 5 specialize.9
  9      works incl engagedIn_1((belongsTo_1<+{
         empl|->belongs})(empl))                  LEM
 10    x = empl => works incl engagedIn_1((
       belongsTo_1<+{empl|->belongs})(empl))      DED
```

```
 11    x: employees_1-{empl}                        HYP
 12      (belongsTo_1<+{empl|->belongs})(x) =
       belongsTo_1(x)                               11 6 4 7 fsupport.16
 13      worksOn_1(x) incl engagedIn_1((
         belongsTo_1<+{empl|->belongs})(x))         8 HYP.12
 14    x: employees_1-{empl} => worksOn_1(x)
       incl engagedIn_1((belongsTo_1<+{empl|->
       belongs})(x))                                DED
 15    (worksOn_1<+{empl|->works})(x) incl
       engagedIn_1((belongsTo_1<+{empl|->belongs
       })(x))                                       3 10 14 support.6
 16  ctx(ExtResearchCompaniesMch) & inv(
     ExtResearchCompaniesMch_1) & pre(
     ExtResearchCompaniesMchupdateEmployer_1(
     empl,belongs,works)) & asn(
     ExtResearchCompaniesMch_1) & x: employees_1
     => (worksOn_1<+{empl|->works})(x) incl
     engagedIn_1((belongsTo_1<+{empl|->belongs})
     (x))                                           DED

END OF PROOF
```

G.2.2.6 Mathematisches Lemma Nr. 15

Lemmaklasse: **MV1L1**

```
PROOF

  1    {proj}<<|%x.(x: projects_2 & x = a,b | a)
       = %x.(x: projects_2-{proj} & x = a,b | a) refinementEqualities.1
  2  ctx(ExtResearchCompaniesMch) & inv(
     ExtResearchCompaniesMch_2) & pre(
     ExtResearchCompaniesMchremoveProject_1(proj
     )) & asn(ExtResearchCompaniesMch_2) => {
     proj}<<|%x.(x: projects_2 & x = a,b | a) =
     %x.(x: projects_2-{proj} & x = a,b | a)     DED

END OF PROOF
```

G.2.2.7 Mathematisches Lemma Nr. 17

Lemmaklasse: **MV2L3**

```
PROOF

   1    %x.(x: dom({proj}<<|projClass_3) | x) = %
        x.(x: dom({proj}<<|projClass_3) | x)      EQL
   2    %x.(x: dom(projClass_3)-{proj} | x) = %x.
        (x: dom({proj}<<|projClass_3) | x)        1 refinementEqualities.8
   3  ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchremoveProject_1(proj
      )) & asn(ExtResearchCompaniesMch_3) => %x.(
      x: dom(projClass_3)-{proj} | x) = %x.(x:
      dom({proj}<<|projClass_3) | x)              DED

END OF PROOF
```

G.2.2.8 Mathematisches Lemma Nr. 23

Lemmaklasse: **MV2L2**

```
PROOF

   1    {empl}<<|%x.(x: dom(empClass_3) &
        empClass_3(x) = a,b,c | c) = %x.(x: dom({
        empl}<<|empClass_3) & ({empl}<<|empClass_
        3)(x) = a,b,c | c)                        refinementEqualities.3
   2  ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchremoveEmployee_1(
      empl)) & asn(ExtResearchCompaniesMch_3) =>
      {empl}<<|%x.(x: dom(empClass_3) & empClass_
      3(x) = a,b,c | c) = %x.(x: dom({empl}<<|
      empClass_3) & ({empl}<<|empClass_3)(x) = a,
      b,c | c)                                    DED

END OF PROOF
```

G.2.2.9 Mathematisches Lemma Nr. 27

Lemmaklasse: *nicht klassifiziert, spezielle Gestalt*
FWD-Quote: 400

```
PROOF

   1  inv(ExtResearchCompaniesMch_3)                HYP
   2  pre(c <-- ExtResearchCompaniesMchnewCompany
      _1(newname,ventures,budg))                    HYP
   3    inv(ExtResearchCompaniesMch_2)              1 GetPredicateX.20
   4    projects_2 = projects_3                     1 GetPredicateX.19
   5    ventures: POW(projects_1)                   2 GetPredicateX.5
   6    projects_1 = projects_2                     3 GetPredicateX.10
   7    projects_1 = projects_3                     4 6 fsupport.63
   8    ventures: POW(projects_3)                   5 7 fsupport.60
   9    ventures incl projects_3                    8 fsupport.44
  10    projects_3\/ventures = projects_3           9 fsupport.42
  11    projId_3 = projId_3                         EQL
  12    projId_3 = %x.(x: projects_3 | x)           11 GetInvDefinitionX.15
  13    projId_3 = %x.(x: projects_3\/ventures |
        x)                                          12 HYP.10
  14    projId_3 = %x.(x: dom(projClass_3)\/
        ventures | x)                               13 GetInvDefinitionX.6
  15    %x.(x: projects_3 | x) = %x.(x: dom(
        projClass_3)\/ventures | x)                 14 GetInvDefinitionX.15
  16    %x.(x: dom(projClass_3) | x) = %x.(x: dom
        (projClass_3)\/ventures | x)                15 GetInvDefinitionX.6
  17    %x.(x: dom(projClass_3) | x) = %x.(x: dom
        (projClass_3)\/dom(%x.(x: ventures |
        projClass_3(x)\/{newname})) | x)            16 refinementEqualities.7
  18    %x.(x: dom(projClass_3) | x) = %x.(x: dom
        (projClass_3<+%x.(x: ventures | projClass
        _3(x)\/{newname})) | x)                     17 refinementEqualities.5
  19  ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_3) => %x.(x: dom(
      projClass_3) | x) = %x.(x: dom(projClass_3
      <+%x.(x: ventures | projClass_3(x)\/{
      newname})) | x)                               DED

END OF PROOF
```

G.2.2.10 Mathematisches Lemma Nr. 29

Lemmaklasse: *nicht klassifiziert, spezielle Gestalt*
FWD-Quote: 300

PROOF

```
 1  inv(ExtResearchCompaniesMch_3)                    HYP
 2  pre(c <-- ExtResearchCompaniesMchnewCompany
    _1(newname,ventures,budg))                        HYP
 3    inv(ExtResearchCompaniesMch_2)                  1 GetPredicateX.20
 4    projects_2 = projects_3                         1 GetPredicateX.19
 5    ventures: POW(projects_1)                       2 GetPredicateX.5
 6    projects_1 = projects_2                         3 GetPredicateX.10
 7    projects_1 = projects_3                         4 6 fsupport.63
 8    ventures: POW(projects_3)                       5 7 fsupport.60
 9    ventures incl projects_3                        8 fsupport.44
10    projects_3\/ventures = projects_3               9 fsupport.42
11    getsGrantFrom_3 = getsGrantFrom_3               EQL
12    getsGrantFrom_3 = %x.(x: projects_3 & x =
      a,b | b)                                        11 GetInvDefinitionX.16
13    getsGrantFrom_3 = %x.(x: projects_3\/
      ventures & x = a,b | b)                         12 HYP.10
14    getsGrantFrom_3 = %x.(x: dom(projClass_3)
      \/ventures & x = a,b | b)                       13 GetInvDefinitionX.6
15    %x.(x: projects_3 & x = a,b | b) = %x.(x:
      dom(projClass_3)\/ventures & x = a,b | b)       14 GetInvDefinitionX.16
16    %x.(x: dom(projClass_3) & x = a,b | b) =
      %x.(x: dom(projClass_3)\/ventures & x = a
      ,b | b)                                         15 GetInvDefinitionX.6
17    %x.(x: dom(projClass_3) & x = a,b | b) =
      %x.(x: dom(projClass_3)\/dom(%x.(x:
      ventures | projClass_3(x)\/{newname})) &
      x = a,b | b)                                    16 refinementEqualities.7
18    %x.(x: dom(projClass_3) & x = a,b | b) =
      %x.(x: dom(projClass_3<+%x.(x: ventures |
      projClass_3(x)\/{newname})) & x = a,b | b
      )                                               17 refinementEqualities.5
19  ctx(ExtResearchCompaniesMch) & inv(
    ExtResearchCompaniesMch_3) & pre(c <--
    ExtResearchCompaniesMchnewCompany_1(newname
    ,ventures,budg)) & asn(
    ExtResearchCompaniesMch_3) => %x.(x: dom(
    projClass_3) & x = a,b | b) = %x.(x: dom(
    projClass_3<+%x.(x: ventures | projClass_3(
    x)\/{newname})) & x = a,b | b)                    DED
```

```
END OF PROOF
```

G.2.2.11 Mathematisches Lemma Nr. 32

Lemmaklasse: **MV2E2**

```
PROOF

   1    %x.(x: dom(compClass_3) & compClass_3(x)
        = a,b | b)<+{newname|->budg} = %x.(x: dom
        (compClass_3) & compClass_3(x) = a,b | b)
        <+{newname|->budg}                          EQL
   2    %x.(x: dom(compClass_3) & compClass_3(x)
        = a,b | b)<+{newname|->budg} = %x.(x: dom
        (compClass_3\/{newname,ventures,budg}) &
        (compClass_3\/{newname,ventures,budg})(x)
        = a,b | b)                                  1 refinementEqualities.12
   3  ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(c <--
      ExtResearchCompaniesMchnewCompany_1(newname
      ,ventures,budg)) & asn(
      ExtResearchCompaniesMch_3) => %x.(x: dom(
      compClass_3) & compClass_3(x) = a,b | b)<+{
      newname|->budg} = %x.(x: dom(compClass_3\/{
      newname,ventures,budg}) & (compClass_3\/{
      newname,ventures,budg})(x) = a,b | b)      DED

END OF PROOF
```

G.2.2.12 Mathematisches Lemma Nr. 34

Lemmaklasse: **MV2E3**

```
PROOF

   1    %x.(x: dom(compClass_3)\/{newname} | x) =
```

```
      %x.(x: dom(compClass_3)\/{newname} | x)   EQL
  2   %x.(x: dom(compClass_3)\/{newname} | x) =
      %x.(x: dom(compClass_3)\/{first(newname)}
      | x)                                      1 refinementEqualities.30
  3   %x.(x: dom(compClass_3)\/{newname} | x) =
      %x.(x: dom(compClass_3)\/{first(newname,
      ventures)} | x)                           2 refinementEqualities.31
  4   %x.(x: dom(compClass_3)\/{newname} | x) =
      %x.(x: dom(compClass_3\/{newname,ventures
      ,budg}) | x)                              3 refinementEqualities.9
  5 ctx(ExtResearchCompaniesMch) & inv(
    ExtResearchCompaniesMch_3) & pre(c <--
    ExtResearchCompaniesMchnewCompany_1(newname
    ,ventures,budg)) & asn(
    ExtResearchCompaniesMch_3) => %x.(x: dom(
    compClass_3)\/{newname} | x) = %x.(x: dom(
    compClass_3\/{newname,ventures,budg}) | x)  DED

END OF PROOF
```

G.2.2.13 Mathematisches Lemma Nr. 38

Lemmaklasse: **MV2E2**

```
PROOF

  1   %x.(x: dom(empClass_3) & empClass_3(x) =
      a,b,c | b)<+{newEmpId|->belongs} = %x.(x:
      dom(empClass_3) & empClass_3(x) = a,b,c |
      b)<+{newEmpId|->belongs}                  EQL
  2   %x.(x: dom(empClass_3) & empClass_3(x) =
      a,b,c | b)<+{newEmpId|->belongs} = %x.(x:
      dom(empClass_3\/{newEmpId,newname,belongs
      ,works}) & (empClass_3\/{newEmpId,newname
      ,belongs,works})(x) = a,b,c | b)          1 refinementEqualities.14
  3 ctx(ExtResearchCompaniesMch) & inv(
    ExtResearchCompaniesMch_3) & pre(e <--
    ExtResearchCompaniesMchnewEmployee_1(
    newname,belongs,works)) & asn(
    ExtResearchCompaniesMch_3) => %x.(x: dom(
    empClass_3) & empClass_3(x) = a,b,c | b)<+{
    newEmpId|->belongs} = %x.(x: dom(empClass_3
    \/{newEmpId,newname,belongs,works}) & (
```

```
      empClass_3\/{newEmpId,newname,belongs,works
      })(x) = a,b,c | b)                          DED

END OF PROOF
```

G.2.2.14 Mathematisches Lemma Nr. 40

Lemmaklasse: **MV2E4**

```
PROOF

   1    dom(empClass_3)\/{newEmpId} = dom(
        empClass_3)\/{newEmpId}                   EQL
   2    dom(empClass_3)\/{newEmpId} = dom(
        empClass_3)\/{first(newEmpId)}            1 refinementEqualities.30
   3    dom(empClass_3)\/{newEmpId} = dom(
        empClass_3)\/{first(newEmpId,newname)}    2 refinementEqualities.31
   4    dom(empClass_3)\/{newEmpId} = dom(
        empClass_3)\/{first(newEmpId,newname,
        belongs)}                                 3 refinementEqualities.31
   5    dom(empClass_3)\/{newEmpId} = dom(
        empClass_3\/{newEmpId,newname,belongs,
        works})                                   4 refinementEqualities.9
   6  ctx(ExtResearchCcompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(e <--
      ExtResearchCompaniesMchnewEmployee_1(
      newname,belongs,works)) & asn(
      ExtResearchCompaniesMch_3) => dom(empClass_
      3)\/{newEmpId} = dom(empClass_3\/{newEmpId,
      newname,belongs,works})                     DED

END OF PROOF
```

G.2.2.15 Mathematisches Lemma Nr. 41

Lemmaklasse: **MV2Ä3**
FWD-Quote: 300

```
PROOF

   1  inv(ExtResearchCompaniesMch_3)                HYP
   2  pre(ExtResearchCompaniesMchupdateEmployer_1
      (empl,belongs,works))                         HYP
   3    inv(ExtResearchCompaniesMch_2)              1 GetPredicateX.20
   4    employees_2 = employees_3                   1 GetPredicateX.19
   5    empl: employees_1                           2 GetPredicateX.7
   6    employees_1 = employees_2                   3 GetPredicateX.10
   7    employees_1 = employees_3                   4 6 fsupport.63
   8    empl: employees_3                           5 7 fsupport.61
   9    {empl} incl employees_3                     8 fsupport.1
  10    employees_3\/{empl} = employees_3           9 fsupport.42
  11    empId_3 = empId_3                           EQL
  12    empId_3 = %x.(x: employees_3 | x)           11 GetInvDefinitionX.18
  13    empId_3 = %x.(x: employees_3\/{empl} | x)   12 HYP.10
  14    empId_3 = %x.(x: dom(empClass_3)\/{empl}
        | x)                                        13 GetInvDefinitionX.7
  15    %x.(x: employees_3 | x) = %x.(x: dom(
        empClass_3)\/{empl} | x)                    14 GetInvDefinitionX.18
  16    %x.(x: dom(empClass_3) | x) = %x.(x: dom(
        empClass_3)\/{empl} | x)                    15 GetInvDefinitionX.7
  17    %x.(x: dom(empClass_3) | x) = %x.(x: dom(
        empClass_3)\/dom({empl|->((%x.(x: dom(
        empClass_3) & empClass_3(x) = a,b,c | a))
        (empl),belongs,works)}) | x)                16 refinementEqualities.6
  18    %x.(x: dom(empClass_3) | x) = %x.(x: dom(
        empClass_3<+{empl|->((%x.(x: dom(empClass
        _3) & empClass_3(x) = a,b,c | a))(empl),
        belongs,works)}) | x)                       17 refinementEqualities.5
  19  ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_3) => %x.(x: dom(
      empClass_3) | x) = %x.(x: dom(empClass_3<+{
      empl|->((%x.(x: dom(empClass_3) & empClass_
      3(x) = a,b,c | a))(empl),belongs,works)}) |
      x)                                            DED

END OF PROOF
```

G.2.2.16 Mathematisches Lemma Nr. 42

Lemmaklasse: **MV2Ä2**

```
PROOF

   1    %x.(x: dom(empClass_3) & empClass_3(x) =
        a,b,c | c)<+{empl|->works} = %x.(x: dom(
        empClass_3) & empClass_3(x) = a,b,c | c)
        <+{empl|->works}                          EQL
   2    %x.(x: dom(empClass_3) & empClass_3(x) =
        a,b,c | c)<+{empl|->works} = %x.(x: dom(
        empClass_3<+{empl|->((%x.(x: dom(empClass
        _3) & empClass_3(x) = a,b,c | a))(empl),
        belongs,works)}) & (empClass_3<+{empl|->(
        (%x.(x: dom(empClass_3) & empClass_3(x) =
        a,b,c | a))(empl),belongs,works)})(x) = a
        ,b,c | c)                                 1 refinementEqualities.20
   3  ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchupdateEmployer_1(
      empl,belongs,works)) & asn(
      ExtResearchCompaniesMch_3) => %x.(x: dom(
      empClass_3) & empClass_3(x) = a,b,c | c)<+{
      empl|->works} = %x.(x: dom(empClass_3<+{
      empl|->((%x.(x: dom(empClass_3) & empClass_
      3(x) = a,b,c | a))(empl),belongs,works)}) &
      (empClass_3<+{empl|->((%x.(x: dom(empClass_
      3) & empClass_3(x) = a,b,c | a))(empl),
      belongs,works)})(x) = a,b,c | c)            DED

END OF PROOF
```

G.2.2.17 Mathematisches Lemma Nr. 46

Lemmaklasse: **MV2Ä3**
FWD-Quote: 300.

```
PROOF

   1  inv(ExtResearchCompaniesMch_3)              HYP
```

```
  2  pre(ExtResearchCompaniesMchupdateJobs_1(
     empl,works))                                    HYP
  3    inv(ExtResearchCompaniesMch_2)                1 GetPredicateX.20
  4    employees_2 = employees_3                     1 GetPredicateX.19
  5    empl: employees_1                             2 GetPredicateX.8
  6    employees_1 = employees_2                     3 GetPredicateX.10
  7    employees_1 = employees_3                     4 6 fsupport.63
  8    empl: employees_3                             5 7 fsupport.61
  9    {empl} incl employees_3                       8 fsupport.1
 10    employees_3\/{empl} = employees_3             9 fsupport.42
 11    empId_3 = empId_3                             EQL
 12    empId_3 = %x.(x: employees_3 | x)             11 GetInvDefinitionX.18
 13    empId_3 = %x.(x: employees_3\/{empl} | x)     12 HYP.10
 14    empId_3 = %x.(x: dom(empClass_3)\/{empl}
       | x)                                          13 GetInvDefinitionX.7
 15    %x.(x: employees_3 | x) = %x.(x: dom(
       empClass_3)\/{empl} | x)                      14 GetInvDefinitionX.18
 16    %x.(x: dom(empClass_3) | x) = %x.(x: dom(
       empClass_3)\/{empl} | x)                      15 GetInvDefinitionX.7
 17    %x.(x: dom(empClass_3) | x) = %x.(x: dom(
       empClass_3)\/dom({empl|->((%x.(x: dom(
       empClass_3) & empClass_3(x) = a,b,c | a))
       (empl),(%x.(x: dom(empClass_3) & empClass
       _3(x) = a,b,c | b))(empl),works)}) | x)       16 refinementEqualities.6
 18    %x.(x: dom(empClass_3) | x) = %x.(x: dom(
       empClass_3<+{empl|->((%x.(x: dom(empClass
       _3) & empClass_3(x) = a,b,c | a))(empl),(
       %x.(x: dom(empClass_3) & empClass_3(x) =
       a,b,c | b))(empl),works)}) | x)               17 refinementEqualities.5
 19  ctx(ExtResearchCompaniesMch) & inv(
     ExtResearchCompaniesMch_3) & pre(
     ExtResearchCompaniesMchupdateJobs_1(empl,
     works)) & asn(ExtResearchCompaniesMch_3) =>
     %x.(x: dom(empClass_3) | x) = %x.(x: dom(
     empClass_3<+{empl|->((%x.(x: dom(empClass_3
     ) & empClass_3(x) = a,b,c | a))(empl),(%x.(
     x: dom(empClass_3) & empClass_3(x) = a,b,c
     | b))(empl),works)}) | x)                       DED

END OF PROOF
```

G.2.2.18 Mathematisches Lemma Nr. 49

Lemmaklasse: **MV2Ä2**

```
PROOF

   1    %x.(x: dom(empClass_3) & empClass_3(x) =
        a,b,c | a) = %x.(x: dom(empClass_3) &
        empClass_3(x) = a,b,c | a)                EQL
   2    %x.(x: dom(empClass_3) & empClass_3(x) =
        a,b,c | a) = %x.(x: dom(empClass_3<+{empl
        |->((%x.(x: dom(empClass_3) & empClass_3(
        x) = a,b,c | a))(empl),(%x.(x: dom(
        empClass_3) & empClass_3(x) = a,b,c | b))
        (empl),works)}) & (empClass_3<+{empl|->((
        %x.(x: dom(empClass_3) & empClass_3(x) =
        a,b,c | a))(empl),(%x.(x: dom(empClass_3)
        & empClass_3(x) = a,b,c | b))(empl),works
        )})(x) = a,b,c | a)                       1 refinementEqualities.23
   3  ctx(ExtResearchCompaniesMch) & inv(
      ExtResearchCompaniesMch_3) & pre(
      ExtResearchCompaniesMchupdateJobs_1(empl,
      works)) & asn(ExtResearchCompaniesMch_3) =>
      %x.(x: dom(empClass_3) & empClass_3(x) = a,
      b,c | a) = %x.(x: dom(empClass_3<+{empl|->(
      (%x.(x: dom(empClass_3) & empClass_3(x) = a
      ,b,c | a))(empl),(%x.(x: dom(empClass_3) &
      empClass_3(x) = a,b,c | b))(empl),works)})
      & (empClass_3<+{empl|->((%x.(x: dom(
      empClass_3) & empClass_3(x) = a,b,c | a))(
      empl),(%x.(x: dom(empClass_3) & empClass_3(
      x) = a,b,c | b))(empl),works)})(x) = a,b,c
      | a)                                        DED

END OF PROOF
```