Universität-Gesamthochschule Paderborn

Fachbereich 17 - Mathematik/Informatik
Arbeitsgruppe Softwaretechnik
Warburger Straße 100
33098 Paderborn

Inkrementelle Konsistenzerhaltung in der transformationsbasierten Datenbankmigration

Diplomarbeit
für den integrierten Studiengang Informatik
im Rahmen des Hauptstudiums II

von

Jörg P. Wadsack
Querweg 38
33098 Paderborn

vorgelegt bei
Prof. Dr. Wilhelm Schäfer

und
Prof. Dr. Gregor Engels

Paderborn, September 1998

Danksagung

Diese Arbeit wurde am Lehrstuhl Softwaretechnik der Universität-Gesamthochschule Paderborn erstellt. Mein Dank gilt Herrn Dipl. Inform. Jens Jahnke für die interessante Aufgabenstellung und seine wertvollen Anregungen und Hinweise, sowie Herrn Prof. Dr. W. Schäfer für die begleitende Betreuung und Herrn Prof. Dr. G. Engels für die Zweitkorrektur der Arbeit. Außerdem danke ich allen, die mir im Verlauf dieser Arbeit zur Seite standen.

Eidesstattliche Erklärung

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe, und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Paderborn, den 11. September 1998

(Jörg P. Wadsack)

Inhaltsverzeichnis

Kapitel 1

Einleitung

Um den ständig komplexer werdenden Anforderungen an Datenbanken nachkommen zu können, haben die objektorientierten Datenbanken immer mehr an Bedeutung gewonnen. Daten aus unterschiedlichen Anwendungsbereichen, wie *CAD* (computer aided design, computerunterstützte Konstruktion), *CIM* (computer integrated manufacturing, computerunterstützte Fertigung) oder *SE* (software engineering, Softwaretechnik), müssen mit vorhandenen Datenbanken betriebswirtschaftlicher Natur integriert werden. Neuere Anwendungsbereiche wie integrierte Planung und Produktion, Management Informationssysteme, Electronic Commerce (Web), usw., benötigen eine föderierte Informationsverwaltung.

Die vorhandenen Informationssysteme (IS) können den heutigen Anforderungen oft nicht nachkommen. Sie müssen modernisiert werden, was viel Zeit und Kosten verursacht. Die heute vorhandenen Datenbanken sind häufig veraltet und schon lange im Einsatz. Diese sogenannten *Legacy-Datenbanksysteme* haben meistens wenig Dokumentation, die ihren technischen Aufbau beschreiben. Viele Betriebe sind von Datenbanken abhängig und können nicht ohne sie arbeiten. Sie sind wertvoll und wichtig für einen Betrieb, weil dieser täglich die gespeicherten Daten aus der Datenbank braucht. Deshalb ist das *Reengineering* von Datenbanken wichtig geworden.

1.1 Datenbank-Reengineering

Das *konzeptionelle* Schema einer Datenbank strukturiert die (Daten-)Objekte und ihre Beziehungen untereinander. Einer der wichtigsten Schritte des Datenbankentwurfs ist die Erstellung des konzeptionellen Schemas. Somit ist dieses bei der Wartung und Änderung von Datenbanken von entscheidender Bedeutung.

Die Anzahl der Entwickler, die am Design und anschließend an der Implementierung einer Datenbank teilgenommen haben, ist groß. Diese nach den vorgenommenen Veränderungen zu befragen, ist ein aussichtsloses Unterfangen. Wenn der betroffene Entwickler überhaupt noch vor Ort ist, ist es unwahrscheinlich, daß dieser sich noch an alles erinnern kann.

Außerdem fehlt bei einer Datenbank häufig eine vollständige Dokumentation über die Entwicklung der Veränderungen. Die vorgenommenen Modifikationen beschränken sich meistens auf das *physikalische* Schema und auf den *Applikationscode*. Im Fall von relationalen Datendanken fügt ein Entwickler beispielsweise eine neue Relation in das physikalische Schema ein

und formuliert die gewünschten Anfragen im Applikationscode. Dabei wird jedoch selten die Zeit aufgewendet, diese Änderungen im konzeptionellen Schema und in der Dokumentation ebenfalls zu erfassen.

Zudem haben die wechselnden Anforderungen und Möglichkeiten über mehrere Jahre häufige Anpassungen hervorgerufen. Der Zeitdruck bei solchen Anpassungen ist in der heutigen Gesellschaft meistens hoch. Diese Punkte verschärfen das Problem der Inkonsistenz zwischen der Dokumentation, dem physikalischen und dem konzeptionellen Schema.

Bei der Modernisierung alter Systeme (neue, komplexere Anforderungen) ist das konzeptionelle Schema ebenfalls notwendig. Die immer präsenter werdende Sprache Java bietet einen leichten Zugang zum Internet, deshalb wird sie bei vielen neuen/erweiternden Implementierungen eingesetzt. Die Realisierung einer Datenbank durch ein Client/Server-System gehört ebenfalls dazu.

Die Anbindung von Datenbanken an das Internet ist aktuell und hat den Vorteil, daß die Datenbank auf einem Server liegt und die Benutzer über das Web auf sie zugreifen können. Die Datenbank kann zentral verwaltet werden und muß nicht mehr in allen verschiedenen Außenstellen vorhanden sein. Somit ist sie leichter und für eine größere Anzahl von Benutzern zugänglich. Dadurch wird außerdem eine Plattformunabhängigkeit erreicht, d.h. jeder der einen Internet Zugang besitzt, kann auf die Datenbank zugreifen, wenn er dazu berechtigt ist.

Weitere Anwendungsgebiete, für die das konzeptionelle Schema gebraucht wird, sind die *Erweiterung* und *Integration* von Datenbanken, die *Föderierung* von Datenbanken, die *Migration* von relationalen zu objektorientierten Datenbanken usw.. Mit der Erweiterung von Datenbanken ist unter anderem das Hinzufügen von Relationen, Attributen oder Beziehungen zwischen solchen in das konzeptionelle Schema gemeint. Die Integration soll einen Datenaustausch zwischen Datenbanken ermöglichen. Der Datenaustausch kann auch zwischen Datenbanken mit unterschiedlichen Datenmodellen, beispielsweise zwischen einer relationalen und einer objektorientierten Datenbank stattfinden. Der Zugriff auf solche heterogenen Datenbanken über eine gemeinsame Zugriffsschicht wird Föderierung von Datenbanken genannt. Die Datenbankmigration besteht aus drei Schritten, der erste ist die *Analyse*. Sie extrahiert Informationen aus der relationalen Datenbank. Der zweite Schritt ist die *Schemamigration*. Sie übersetzt das relationale in das objektorientierten Schema. Der letzte ist die *Applikationsmigration*. Sie besteht aus der *Datenmigration* und der *Anfragemigration*. Die Datenmigration überführt die Ausprägung der Datenbank. Die Anfragenmigration übersetzt die Datenbankanfragen.

Legacy-Datenbanksysteme sind weit verbreitet und oftmals komplex. Diese Umstände lassen das Datenbank-Reengineering immer wichtiger werden.

Definition des Datenbank-Reengineering nach [CC90]:

> Die Analyse einer „alten“ Datenbank, um sie in neuer Form wiederherzustellen, und die Implementierung dieser neuen Form werden Datenbank-Reengineering (DBRE) genannt.
> Die wichtigsten Aktivitäten des DBRE können in *Reverse-Engineering* (Analyse) und *Forward-Engineering* (Migration) klassifiziert werden.

Das Reverse-Engineering hat die Aufgabe, (Meta-)Informationen aus einer vorhandenen Datenbank zu gewinnen. Das Forward-Engineering ist die Implementierung der gewonnenen Informationen in neuer Form.

1.2 Ziel der Arbeit

In dieser Arbeit wird das Reengineering von Datenbanken für die Migration von relationalen zu objektorientierten Datenbanken untersucht. Auf die Analyse der relationalen Datenbank wird hier nicht eingegangen. Ausgehend von einer analysierten und angereicherten Datenbank liegt der Schwerpunkt auf dem Forward-Engineering.

Ein iterativer, explorativer Ansatz

Diese Arbeit befaßt sich mit dem zweiten Schritt der Datenbankmigration, der Schemamigration. Während des Migrationsprozesses ist eine Interaktion des Benutzers notwendig. Nachdem das relationale Schema in das objektorientierte überführt wurde, können Erkenntnisse auftreten, die einen Rückschritt erfordern. Erweiterungen oder Nachbesserungen des relationalen Schemas müssen durchgeführt werden. Diese sollen dann wieder in das objektorientierte Schema einfließen. Diese Arbeit beschäftigt sich mit der Konsistenzsicherung bei solchen Rückschritten, um einen *iterativen, explorativen* Reengineering Prozeß zu ermöglichen [JSZ96].

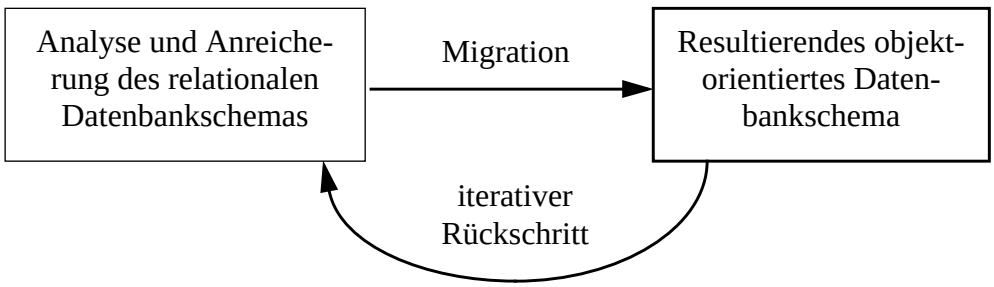


Abb. 1.1: Ein iterativer, explorativer Ansatz

Die Idee ist, die Migration in zwei Schritten durchzuführen. Als erstes wird das erweiterte (analysierte und angereicherte) relationale Schema automatisch mit einer (initialen) Abbildung in ein objektorientiertes Schema übersetzt. Dieses „erste“ objektorientierte Schema sieht dem relationalen ähnlich und nutzt die objektorientierten Konzepte kaum. Deshalb besteht der zweite Schritt in der Restrukturierung des objektorientierten Schemas. Dem Benutzer werden Transformationen zur Verfügung gestellt, die es ihm erlauben, das objektorientierte Schema zu strukturieren. Durch die Aufspaltung des Migrationsprozesses erhält der Benutzer den Vorteil, daß er nicht schon im relationalen Schema alle Entscheidungen für die Restrukturierung treffen muß. Er bekommt eine objektorientierte Sicht des Schemas und kann sich anschließend entscheiden. Außerdem wird ihm die Möglichkeit gegeben, das relationale Schema erneut zu analysieren und es mit zusätzlichen Informationen anzureichern. Die objektorientierte Sicht liefert oft neue Erkenntnisse über das Ausgangsschema. Deshalb ist diese Möglichkeit ein wichtiger Schritt während der Datenbankmigration.

Das Ziel dieser Arbeit besteht darin, die Konsistenz zwischen den Schemata während des gesamten Schemamigrationsprozesses zu erhalten. Ein Mechanismus soll entworfen und implementiert werden, der eine inkrementelle Konsistenzerhaltung während der Migration bereitstellt.

1.3 Gliederung der Arbeit

In Kapitel 2 werden verwandte Arbeiten vorgestellt. Die verschiedenen Ansätze werden untersucht und die Vor- bzw. Nachteile diskutiert.

In Kapitel 3 wird die Datenbankmigration in *VARLET* beschrieben, ein iterativ-explorativer Ansatz. Es wird die Architektur dieses Ansatzes dargelegt und ein Beispiel-Szenario erklärt.

Anschließend, in Kapitel 4, wird das relationale und das nach dem *ODMG-2.0* Datenmodell geprägte Migrations-Meta-Modell (M^3) vorgestellt. Dazu werden die benutzten Tripelgraphgrammatiken und Restrukturierungstransformationen detaillierter betrachtet.

Der Ansatz zur Konsistenzerhaltung auf der Basis von *Nachtransformationen* wird in Kapitel 5 präsentiert. Hier werden die verschiedenen Möglichkeiten, der Aufbau und die Fehlerbehandlung bei der Konsistenzerhaltung im Vordergrund stehen.

Die Implementierung der beschriebenen Konzepte und der Prototyp werden in Kapitel 6 vorgestellt. Ein Ablauf des Beispiel-Szenarios in der VARLET-Umgebung gehört ebenfalls zu diesem Kapitel.

Das Kapitel 7 besteht aus der Zusammenfassung des vorgestellten Ansatzes und den Schwerpunkten dieser Arbeit. Außerdem wird ein Ausblick präsentiert und zukünftige Erweiterungen werden vorgeschlagen.

Kapitel 2

Verwandte Arbeiten

Vollautomatische Ansätze

Viele Arbeiten beschäftigen sich mit der Migration von einem relationalen zu einem objektorientierten Schema bzw. dem (Extended-)Entity-Relationship Modell. Die meisten Ansätze sind *vollautomatisch*, d.h. der Benutzer kann in den Migrationsprozeß nicht eingreifen [DA87, MM90, JK90, SK90, ACM94, And94, FV95, RH96, RH97, Fong97].

Sie unterscheiden sich aber durch verschiedene Anforderungen an das relationale Schema. Diese Anforderungen sind jedoch oftmals unrealistisch, da *Legacy-Systeme* häufig nicht unter Anwendung bestimmter Entwurfsprinzipien (weiter-)entwickelt wurden. In [DA87] und [RH96] wird die dritte Noramalform, in [RH97] die zweite Normalform vorausgesetzt. Zusätzlich zu der dritten Normalform wird eine Klassifizierung der Relationen in [JK90] vorausgesetzt. Das Resultat der Migration ist das Entity-Relationship (ER) Modell [DA87], ein erweitertes ER Model [JK90] oder ein objektorientiertes Schema [RH96, RH97].

In [SK90] wird angenommen, daß das relationale Schema nach dem ER Modell entworfen wird. Hier wird mit Hilfe eines *kanonisch-relationalen* Schemas CRS (Canonical Relational Schema) ein ER-Diagramm konstruiert.

Ein Extended-ER (EER) Schema kann durch ein relationales Schema in BCNF (Boyce-Codd Normalform), welches Schlüssel-Abhängigkeiten und schüsselbasierte Inklusionsabhängigkeiten unterstützt, dargestellt werden. In [MM90] muß das relationale Schema in dieser Form sein und mit einem EER Modell verknüpft werden können (EER-*convertible*). Das resultierende Schema ist dann in eine, von den Autoren definierten, Normalform *EER/OSNF* (EER object structure normal form).

Andere Ansätze sind nur anwendbar, wenn das zugrundeliegende Schema vorher in eine genau vordefinierte Form gebracht wurde. In [And94] wird ein *Connection Diagram* definiert, in dem die gewonnenen Informationen aus dem Schema und dem Applikationscode gespeichert werden. Anschließend wird dieses in ein von Andersson erweitertes ER Modell überführt.

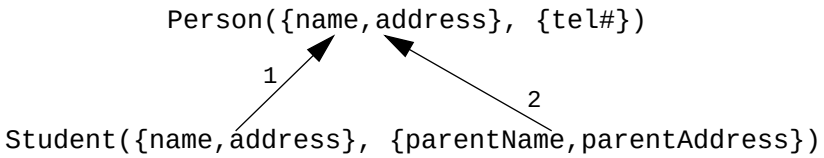


Abb. 2.1: Connection Diagram mit Schlüsseln und Inklusionsabhängigkeiten

Ein anderer Ansatz überführt das relationale Schema in eine objektorientierte Form. In [FV95] wird es *Relational Completion* genannt. Mit Hilfe einer Analyse des Schemas und des Applikationscodes wird das zugrundeliegende Schema vervollständigt. Danach wird es in ein objektorientiertes Schema gemäß dem ODMG-93 Standard transformiert.

Zwei Schemata sind *äquivalent*, wenn eine Abbildung zwischen ihnen existiert, so daß alle Eigenschaften, die im Kontext des ersten Schemas essentiell sind, im zweiten erhalten bleiben. Dies ist die zugrundeliegende Definition in [Fong97]. Das relationale Schema wird in ein objektorientiertes übersetzt, und anschließend werden die Daten migriert. Der Schwerpunkt liegt hier auf einer Verbindung der Schema- und Datenmigration.

Das objektorientierte Schema ist bei allen hier vorgestellten Ansätzen relational „geprägt“. Damit das resultierende Schema auch die gewünschten objektorientierten Eigenschaften besitzt, muß der Benutzer *aktiv* in den Migrationsprozeß eingreifen. Die sogenannten *halbautomatischen* Ansätze sollen dies gewährleisten.

Halbautomatische Ansätze

[HTJC93] stellt eine Reverse-Engineering Methode für Datenbanken vor. In einem ersten Schritt werden die Strukturen des Schemas extrahiert. Danach wird die Semantik des Schemas wiederhergestellt. Dafür werden dem Benutzer Transformationen angeboten, um ein konzeptionelles Schema konstruieren zu können. Die Transformationen sollen *reversible* sein, d.h. das Quellschema und das Zielschema sollen die gleiche Mächtigkeit in Bezug auf ihre Ausdrucksfähigkeit (*descriptive Power*) haben und den Kontext wahren.

Im Ansatz von [JJ94] wird ein *Meta Modell* für die Migration benutzt. Das Meta Modell ist eine Hierarchie, die Bausteine von Datenmodellen klassifiziert. Das zugrundeliegende Schema wird in das Meta Modell umgewandelt. Der Benutzer kann dann das konzeptionelle Schema aus dem Meta Modell erzeugen.

Das Reverse-Engineering von Datenbanken wird auch bei [SLGC94] in zwei Phasen durchgeführt. In der ersten werden mittels einer Heuristik *Indikatoren* in der relationalen Datenbank gesucht. Indikatoren sind Hinweise auf semantische Eigenschaften eines Schemas wie z.B. Fremdschlüssel. Der Benutzer kann dann mit ihrer Hilfe in einem zweiten Schritt das ER Schema konstruieren.

Ein halbautomatischer Ansatz, der das Schema und die Daten migriert, ist [BGD97]. Wie zusätzliche Informationen aus der relationalen Datenbank gewonnen werden, wird nicht beschrieben. Der interaktive Teil des Migrationsprozesses ist die Schematransformation. Der Benutzer konstruiert das objektorientierte Schema mit einer Menge von vordefinierten Transformationen. Die Datenmigration wird anschließend automatisch vollzogen.

Der Nachteil der halbautomatischen und vollautomatischen Ansätze zeigt sich vorallem darin, daß sie wasserfall-orientiert sind. Damit ist gemeint, daß der Prozeß keine Rückschritte erlaubt.

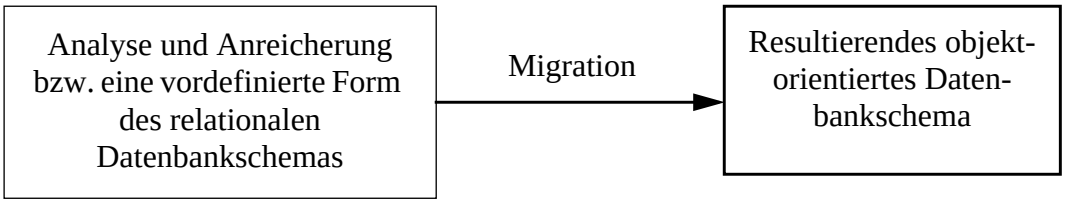


Abb. 2.2: Wasserfall-orientiertes Modell

Falls Informationen aus dem relationalen Schema fehlen oder unvollständig sind, können jedoch Rückschritte gebraucht werden. Bei den vorgestellten Ansätzen ist es nicht möglich, Korrekturen nachträglich in den Migrationsprozeß zu integrieren. Er müßte von neuem gestartet werden.

Ein iterativer / halbautomatischer Ansatz

In [PB94] wird dieses Problem berücksichtigt. Die dort präsentierten Schritte ermöglichen dem Benutzer mit *backtracking* und einer *Neuordnung* der Schritte einen *iterativen / halbautomatischen* Prozeß. Der Benutzer hat die Möglichkeit, seine Design-Entscheidungen für das objektorientierte Schema rückgängig zu machen. Nur ein Teil des Reverse-Engineering Prozesses ist automatisiert, und einfache Werkzeuge wie in UNIX grep, awk, usw. werden benutzt. Die Konsistenz zwischen den Schemata muß allerdings manuell erhalten werden. Nach der Schematransformation sind nicht genug Informationen für eine automatische Datenmigration vorhanden.

ONTOS

Die Firma ONTOS ist Anbieter der Objektdatenbank OIS (Object Integration Server). OIS ist ein kommerziell verfügbares Werkzeug und enthält unter anderem einen *Schema Mapper* (vgl. [ONTOS]), der Reengineering erlaubt.

Für jede Klasse im OIS Schema wird ein *type mapping* angelegt. Die Beziehungen zwischen der relationalen Datenbank und dem OIS Schema werden darin gespeichert. Klassen heißen in OIS *types*. Die Abbildung zwischen einer Relation und einer Klasse (type) wird durch eine Beziehung *type relationship* dargestellt.

Zu einem type mapping gehört auch die Information der Abbildung der Spalten einer Relation auf die Attribute einer Klasse. Die Abbildung der Attribute wird in OIS *property map* genannt. Die Abbildung in OIS bietet insgesamt wenig Variationsmöglichkeiten. Die Reihenfolge von möglichen Editoroperationen ist strikt festgelegt, was das Editieren sehr umständlich macht. Eine neue Oberklasse kann beispielsweise erst dann auf eine Tabelle abgebildet werden, wenn alle Unterklassen schon eingeführt wurden, aber noch keine relationale Abbildung aufweisen. Eine semantische Anreicherung des relationalen Schemas ist über einige Umwege nur schwer und auch nur eingeschränkt realisierbar.

Es werden Annahmen für das Design des relationalen Schemas gemacht, die nur ein unflexibles Arbeiten erlauben und in der Praxis selten erfüllt sind.

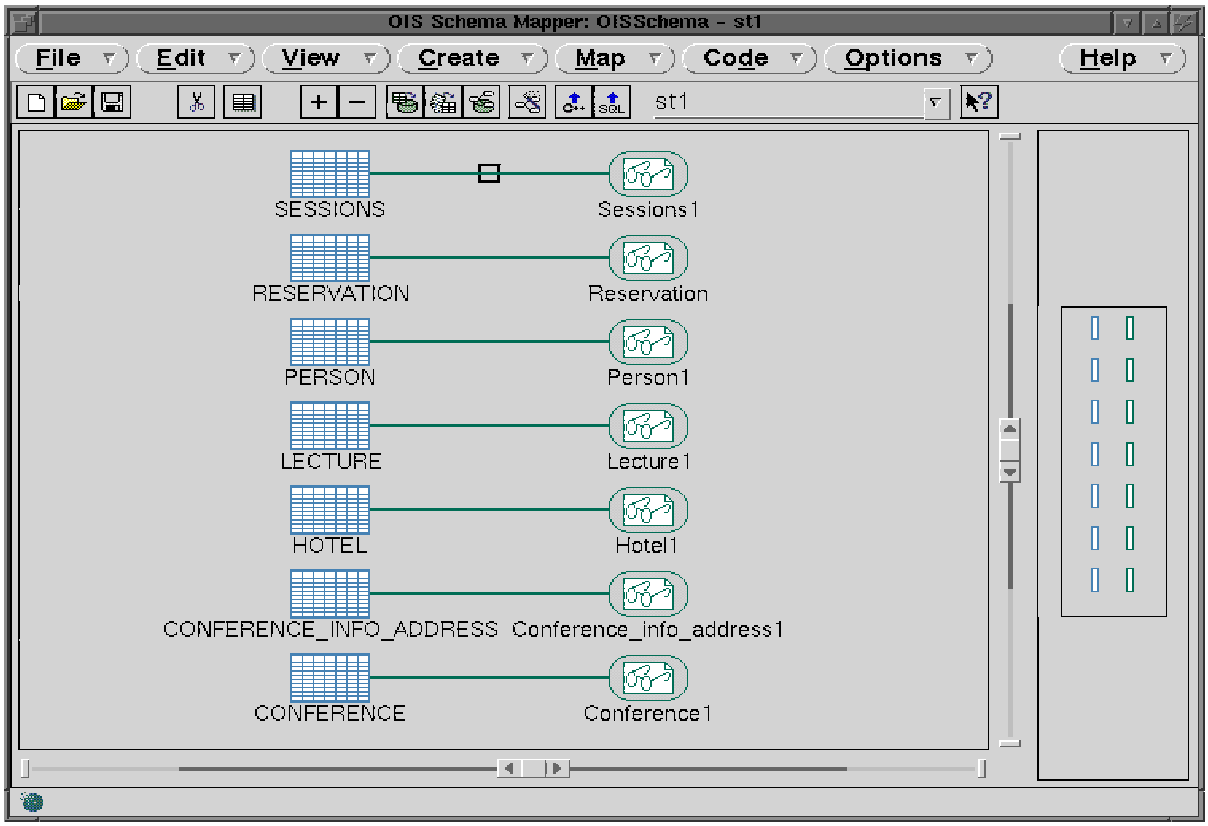


Abb. 2.3 : Der OIS Schema Mapper

Ein Fremdschlüssel muß immer auf einen Primärschlüssel referenzieren und mit diesem einen genau identischen Typ aufweisen. Außerdem ist es Voraussetzung, daß alle Schlüsselkandidaten und Fremdschlüssel der Relationen im Schema explizit definiert sind. In Legacy-Datenbanken ist dies aber selten anzutreffen. Ältere Datenbanksprachen bieten diese Möglichkeit gar nicht. Ein Fremdschlüssel kann auch auf Attribute referenzieren, die nicht den Primärschlüssel bilden. Selbst wenn diese Bedingungen noch erfüllt sind, können leicht kleine Abweichungen in der Namensgebung, zum Beispiel Groß- und Kleinschreibung, und den Datentypen der Fremd- und Primärschlüssel auftreten. Zwei Zeichenketten (`char`) brauchen nur mit verschiedener Länge definiert zu sein und schon gelten sie nicht mehr als identisch.

Die Annahme, daß Fremdschlüssel nur auf genau identische Primärschlüssel referenziert werden dürfen, schränkt das Reengineering auf wenige Datenbanken ein. Darüberhinaus wird dem Benutzer keine Möglichkeit gegeben, semantische Hinweise oder Vermutungen - zum Beispiel über die Bedeutung von Fremdschlüsselbeziehungen zwischen den Relationen - zu kommentieren.

Die Abbildung ist nicht immer korrekt. Der Schema Mapper hat beispielsweise Probleme bei der Abbildung einer m:n-Beziehung, wenn die Beziehungsrelation noch zusätzliche Attribute enthält. Diese zusätzlichen Attribute werden bei der Abbildung in das OIS Schema vergessen. Außerdem läßt der Schema Mapper in einigen Fällen Abbildungen zu, die nicht sinnvoll sind. Eine ausreichende Überprüfung findet nicht statt.

Kapitel 3

Datenbankmigration in VARLET

In diesem Kapitel werden die Datenbankmigrations-Umgebung VARLET (Verified Analysis and Reengineering of Legacy database systems using Equivalence Transformations) und ein Beispiel zur Motivation vorgestellt.

3.1 Datenbankmigrations-Ansatz

Die Hauptaufgabe von VARLET ist die Migration einer relationalen Datenbank in eine objektorientierte bzw. objekt-relationale Datenbank. Die konkreten Datenmodelle sind verschieden, deshalb wird ein gemeinsames objektorientiertes Datenmodell gebraucht. Das Migrations-Meta-Modell (M^3) basiert auf dem ODMG-2.0 (*Objekt Data Management Group*) Datenmodell und ist ein solches Modell. Außerdem wird ein Abbildungsschema verwaltet, das eine eindeutige Zuordnung zwischen den Schemata der Datenmodelle gewährleisten soll.

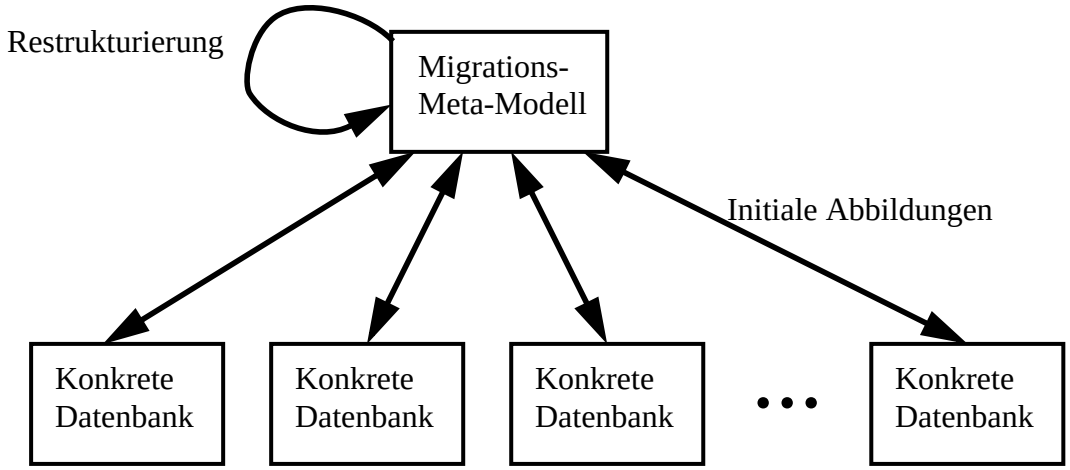


Abb. 3.1: Der Datenbankmigrations-Ansatz in VARLET

Aus einer konkreten Datenbank wird mittels einer *initialen Abbildung* ein äquivalentes Migrations-Meta-Schema konstruiert. Dieses Schema kann dann restrukturiert werden und mit einer weiteren initialen Abbildung in die gewünschte konkrete Datenbank abgebildet werden. Es ist auch möglich, aus dem M^3 wieder in das Ausgangsdatenmodell zurück abzubilden.

Die existierenden relationalen Datenbanken beinhalten selten alle *semantischen* Informationen, die für die Abbildung gebraucht werden. Zudem ist das Schema meistens unvollständig und nicht eindeutig. Schlüssel und Fremdschlüssel sind häufig nicht gekennzeichnet. Deshalb muß das relationale Datenbankschema in einer *Analyse-Phase* angereichert werden.

In VARLET ist das Wissen explizit spezifizierbar und flexibel an veränderte Bedingungen anpaßbar, wie beispielsweise das Hinzukommen neuen Wissens. Dieses Wissen wird mit Konfidenzfaktoren gewichtet (siehe [JSZ97, JH98]), es bekommt Werte zwischen null (*ausgeschlossen*) und eins (*sicher*).

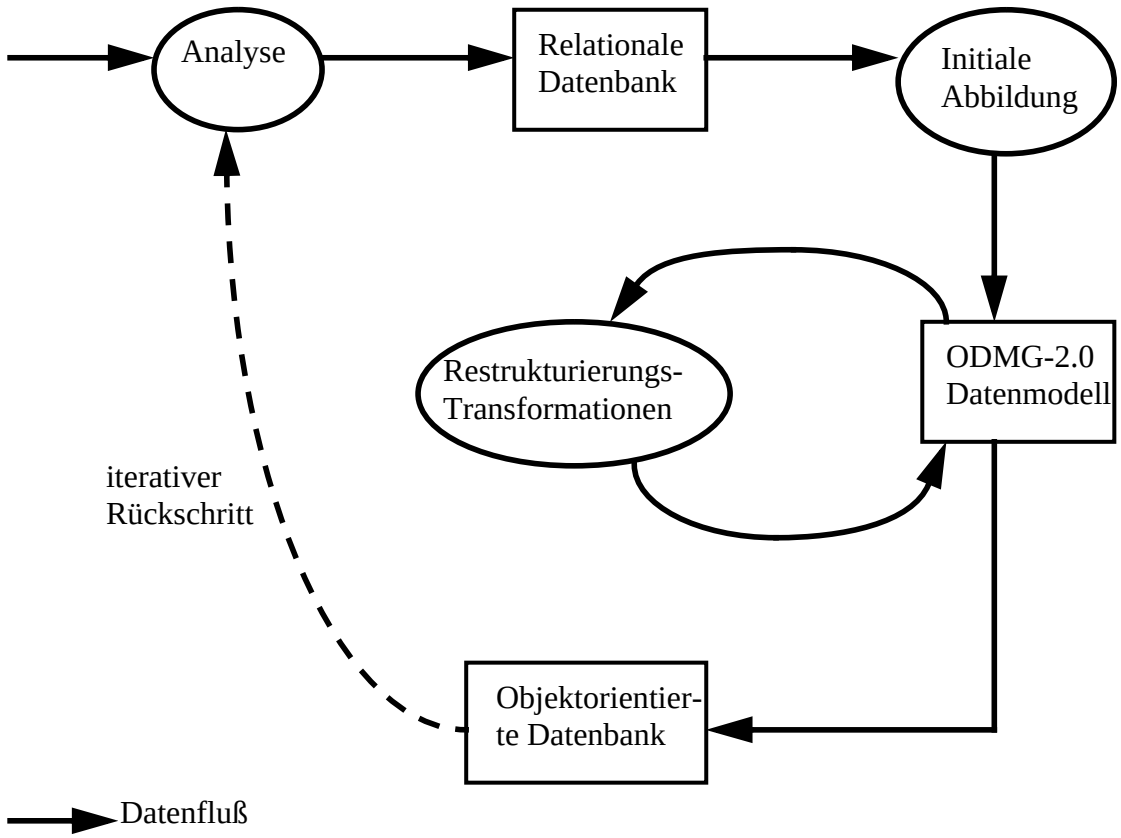


Abb. 3.2: Datenbankmigration in VARLET

Initiale Abbildung

Das relationale Schema wird dann mit der initialen Abbildung in das M^3 Schema übersetzt, ähnlich wie es schon in [JSZ96] beschrieben worden ist. Das resultierende Schema wird graphisch angezeigt und ist objektorientiert, sieht dem relationalen jedoch sehr ähnlich. Diese Abbildung geschieht automatisch mit Hilfe von Tripelgraphgrammatiken (vgl. [Lef95, JSZ96, Hol97]).

Restrukturierung

Anschließend kann das vorhandene objektorientierte Schema mit Transformationen restrukturiert werden, um dieses zu optimieren und die Vorteile der objektorientierten Konzepte (Datenbanken) zu nutzen. Das resultierende Schema ist dann immer noch graphisch

veranschaulicht. Es kann aber auch textuell ausgegeben und gespeichert werden. Dieser Text kann anschließend in eine objektorientierte Datenbank eingelesen werden. Dieser Vorgang entspricht einer Abbildung des M^3 Schemas in ein konkretes Objektmodell, z.B. das Java-ODMG Binding.

Rückschritt

Soweit unterscheidet sich dieser Ansatz noch nicht wesentlich von anderen. Die anderen Ansätze sind strikt phasenorientiert (Wasserfall-orientierte Modelle). Erkenntnisse, die nach der Restrukturierung gewonnen werden, können jedoch wichtig für die Analyse sein. Deshalb ist es an diesem Punkt möglich, die Analyse mit den neuen Informationen erneut auszuführen. Vorher kann das relationale Schema an das objektorientierte Schema automatisch angeglichen werden. Danach ist eine *Nachtransformation* nötig, da sich das relationale Schema geändert hat.

Änderungen des relationalen Schemas wirken sich auf das objektorientierte Schema aus. Einerseits ist das „Ausgangsschema“ im M^3 davon betroffen. Wäre die initiale Abbildung nicht schon vorher durchgeführt worden, hätte sie ein anderes Schema erzeugt. Andererseits hätten dann eventuell einige Restrukturierungstransformationen nicht ausgeführt werden dürfen. Deshalb müssen die initiale Abbildung und die Restrukturierung wiederholt werden. Allerdings nur auf den veränderten Schemakomponenten, um den Aufwand in Grenzen zu halten.

Inkrementelle Konsistenzerhaltung

Der Reengineer möchte bzw. kann nicht alle schon durchgeführten Restrukturierungstransformationen erneut ausführen. Er müßte alle getroffenen Entscheidungen und Erkenntnisse behalten bzw. neu treffen. Die initiale Abbildung und die Restrukturierung werden *inkrementell* wiederholt. Bei jedem Iterationsschritt werden so viele manuell durchgeführte Restrukturierungstransformationen automatisch wiederhergestellt, wie noch möglich sind, ohne die Konsistenz zwischen Ursprungs- und Zielschema zu zerstören. Das Schema im M^3 soll ähnlich dem vor der erneuten Analyse sein.

Alle Transformationen sind nicht immer neu ausführbar. Hier ist die Konsistenzerhaltung ein wichtiger Punkt. Sie garantiert dem Benutzer ein zum relationalen konsistentes, objektorientiertes Schema. Nur diejenigen Transformationen dürfen erneut ausgeführt werden, die aufgrund der neuen Analyseinformationen noch zulässig sind.

Ein Logbuch zeigt die nicht durchgeführten Restrukturierungstransformationen an. Der Benutzer kann auf diese Weise die nicht wiederholten Transformationen begutachten. Die nötigen Schritte, um das Schema in die gewünschte Form zu bringen, werden dadurch erleichtert.

Als nächstes folgt ein Beispiel-Szenario, das diese Schritte deutlicher machen soll.

3.2 Beispiel-Szenario

Der folgende Auszug aus einem SQL Schema sei gewählt:

```
CREATE TABLE Person (
Name VARCHAR (50) NOT NULL,
Geburtstag DATETIME,
Strasse VARCHAR (65),
Stadt VARCHAR (60),
PLZ INTEGER,
Tel INTEGER,
Raum VARCHAR (10),
Nr INTEGER NOT NULL,
FachSem INTEGER,
PRIMARY KEY (Nr) )

CREATE TABLE Vorlesung (
Titel VARCHAR (50),
DozentNr INTEGER NOT NULL,
RefNr INTEGER NOT NULL,
PRIMARY KEY (RefNr),
FOREIGN KEY (DozentNr) REFERENCES Person (Nr) )

CREATE TABLE StudVorl (
MatrNr INTEGER NOT NULL,
RefNr INTEGER NOT NULL REFERENCES Vorlesung,
PRIMARY KEY (MatrNr,RefNr),
FOREIGN KEY (MatrNr) REFERENCES Person (Nr) )

CREATE TABLE Abschlusspruefung (
Faecher VARCHAR (80),
MatrNr INTEGER NOT NULL,
Datum DATETIME NOT NULL,
Ort VARCHAR (30) NOT NULL,
Punkte INTEGER,
Note NUMERIC,
FOREIGN KEY (MatrNr) REFERENCES Person (Nr) )

CREATE TABLE Schein (
RefNr INTEGER NOT NULL REFERENCES Vorlesung (RefNr),
MatrNr INTEGER NOT NULL,
Datum DATETIME NOT NULL,
Ort VARCHAR (30) NOT NULL,
Punkte INTEGER,
Bestanden BOOLEAN,
PRIMARY KEY (RefNr,MatrNr,Datum),
FOREIGN KEY (MatrNr) REFERENCES Person (Nr) )
```

Abb. 3.3: Ein relationales Teilschema einer Universitäts-Datenbank

Das relationale Schema ist unvollständig. Der Tabelle `Abschlusspruefung` fehlt die Angabe eines *Schlüssels*. Hier sind die Fremdschlüsselbeziehungen schon angegeben. Oft ist dies in alten Datenbanken nicht der Fall, weil es in alten Datenbanksprachen, z.B. in den Anfängen von SQL, nicht erlaubt war. Ab SQL-92 können die Fremdschlüsselbeziehungen auf verschiedene Weisen angegeben werden. Entweder direkt nach einem Attribut oder explizit mit `FOREIGN KEY`. Siehe „RefNr“ und „MatrNr“ in der Tabelle `Schein`. Die Attribute von Fremdschlüsseln werden nicht immer angegeben, wie z.B. in `StudVorl` für `RefNr INTEGER NOT NULL REFERENCES Vorlesung`.

Außerdem wird Redundanz im Schema entdeckt. Die Tabellen `Abschlusspruefung` und `Schein` haben beispielsweise beide die Attribute „MatrNr“, „Datum“, „Ort“ und „Punkte“.

In der Analyse-Phase werden zwei Varianten für die Tabelle `Person` identifiziert. Die Varianten werden während der Analyse des vorhandenen Datenbestandes (über NULL-Werte)

entdeckt. Die Tabelle wird auf eine abstrakte Klasse *Person* und zwei Unterklassen Person#1 und Person#2 abgebildet. Da die initiale Abbildung automatisch durchgeführt wird, kann keine sinnvolle Namensgebung vollzogen werden. Deshalb sind in der Abb. 3.4 die zwei Unterklassen schon in Professor und Student umbenannt worden. Außerdem sollte die Analyse den fehlenden *Schlüssel* (primary key) für Abschlusspruefung liefern.

Die Attribute auf der linken Seite einer Fremdschüsselbeziehung werden ins objektorientierten Schema nicht abgebildet. Die Definition der *Assoziation* impliziert, daß Attribute auf ihrer linken Seite von der Klasse auf der rechten Seite sichtbar sind.

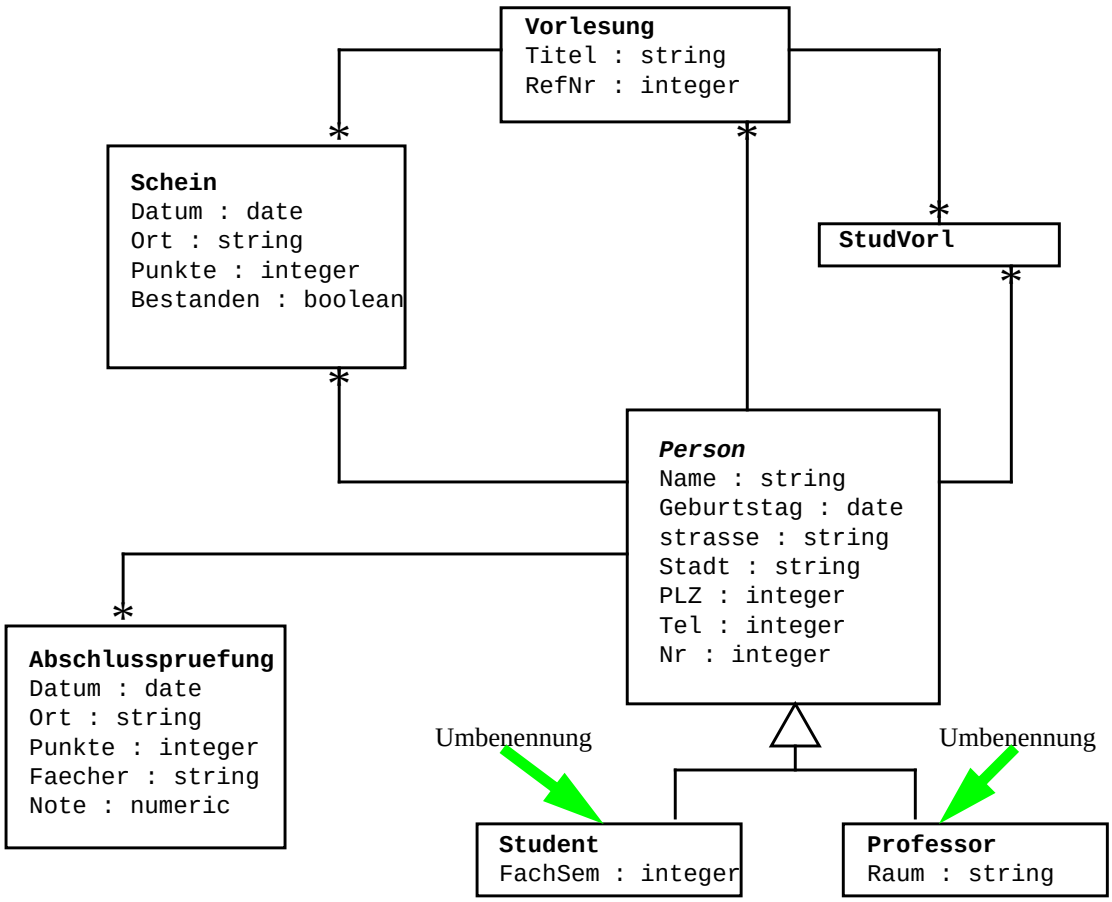


Abb. 3.4: Das objektorientierte Teilschema - dem relationalen ähnlich

Abb. 3.4 bis 3.7 benutzen die OMT-Notation (vgl. [OMT]), d.h. Beziehungen werden durch einfache Linien dargestellt. Die Vererbungen sind mit einem Dreieck und Aggregationen mit einer Raute dargestellt. Die Kardinalitäten für die Assoziationen werden mit einem Stern für [0-n], einem Plus für [1-n], einem Kreis für [0-1] und einem Strich für [1-1] dargestellt.

Das objektorientierte Teilschema sieht nach der initialen Abbildung dem relationalen noch sehr ähnlich. Die Vererbung in der Tabelle Person wird erkannt, aber die objektorientierten Konzepte lassen viele äquivalente, anders strukturierte Schemata zu.

Mehrere Restrukturierungen sind für dieses Schema angebracht. Die Klasse StudVorl wird in eine Assoziation umgewandelt. Sie resultiert aus einer Tabelle, die eine n:m-Beziehung repräsentiert. Von der Klasse *Person* wird die Klasse Adresse abgespalten und *aggregiert*.

Für die Klassen `Abschlusspruefung` und `Schein` wird eine abstrakte Oberklasse `Pruefung`, mit den Attributen „Datum“, „Ort“ und „Punkte“ angelegt. Hinzu kommt die Spezialisierung des Attributes „Nr“ aus *`Person`* in „MatrNr“ für `Student`. Dies ermöglicht die Spezialisierung einiger Assoziationen.

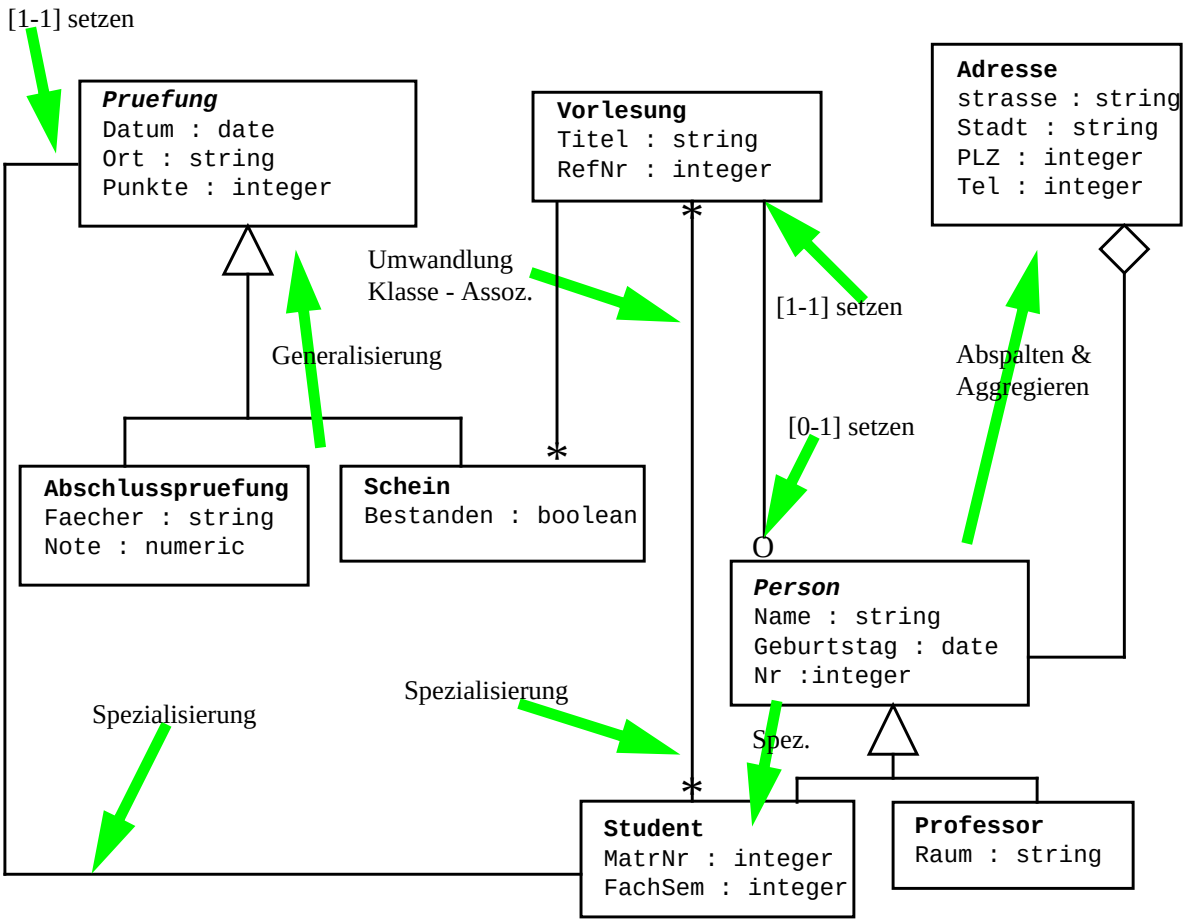


Abb. 3.5: Das objektorientierte Teilschema

Die umgewandelte Assoziation zwischen `Vorlesung` und *`Person`* wird nach `Student` spezialisiert. Die andere Assoziation zwischen `Vorlesung` und *`Person`* resultiert aus der Fremdschlüsselbeziehung von „DozentNr“ nach „Nr“ im relationalen Schema. Sie wird in eine 1:1-Assoziation umgewandelt, weil jede Vorlesung nur von einem Professor gelesen wird.

Die Assoziationen zwischen `Abschlusspruefung` und *`Person`* und die zwischen `Schein` und *`Person`* sind redundant. Wegen der eingeführten Oberklasse *`Pruefung`* reicht eine 1:1-Assoziation zwischen *`Pruefung`* und `Student`, um einen Schein oder eine Abschlußprüfung einem Studenten zuzuordnen.

Weitere Optimierungen des Schemas sind sinnvoll. Die Verschiebung des Attributes „Nr“ aus *`Person`* nach „PersNr“ in `Professor` ist angebracht. Die Spezialisierung dieses Attributes für `Student` ist bereits erfolgt. Die Spezialisierung der Assoziation zwischen `Vorlesung` und *`Person`* zu einer Assoziation zwischen `Vorlesung` und `Professor` steht noch aus.

Damit sind die Analyse und die Restrukturierung vorerst abgeschlossen. Angenommen, daß alle Abschlußprüfungen mündlich sind und dem Reengineer dieses Bereichswissen bekannt ist. Das Attribut „Punkte“ in der Tabelle `Abschlusspruefung` ist dann überflüssig. Das heißt, dieses Attribut könnte von der Klasse *`Pruefung`* in die Klasse `Schein` verschoben werden.

An dieser Stelle müßte dann ein Rückschritt erfolgen. Die Analyse müßte prüfen, ob „Punkte“ nie in den relationalen Daten für `Abschlusspruefung` belegt worden ist (d.h. einen NULL-Wert hat). Ist dies der Fall, kann im relationalen Schema das Attribut „Punkte“ in der Tabelle `Abschlusspruefung` NULL gesetzt werden. Daraus ergeben sich zwei Varianten. Warum diese Varianten nicht früher gefunden wurden, liegt an fehlenden Informationen während der voherigen Analyse. Durch das Bereichswissen des Reengineers ist dies nun möglich.

Bei der Nachtransformation würden dann die Klassen `Abschlusspruefung` und `Abschlusspruefung#1` entstehen. Die Generalisierung von `Schein` müßte allerdings erneut durch den Benutzer vollzogen werden, da sich die Ausgangstabellen und dementsprechend die Ausgangsklassen geändert haben. Außerdem sollte der Benutzer die Klassen `Abschlusspruefung` in `Pruefung` und `Abschlusspruefung#1` in `Abschlusspruefung` umbenennen.

Eine Verschiebung der Attribute „Faecher“ und „Note“ von `Pruefung` nach `Abschlusspruefung` und das Löschen des Attributes „Punkte“ aus `Abschlusspruefung` ist für dieses Beispiel sinnvoll.

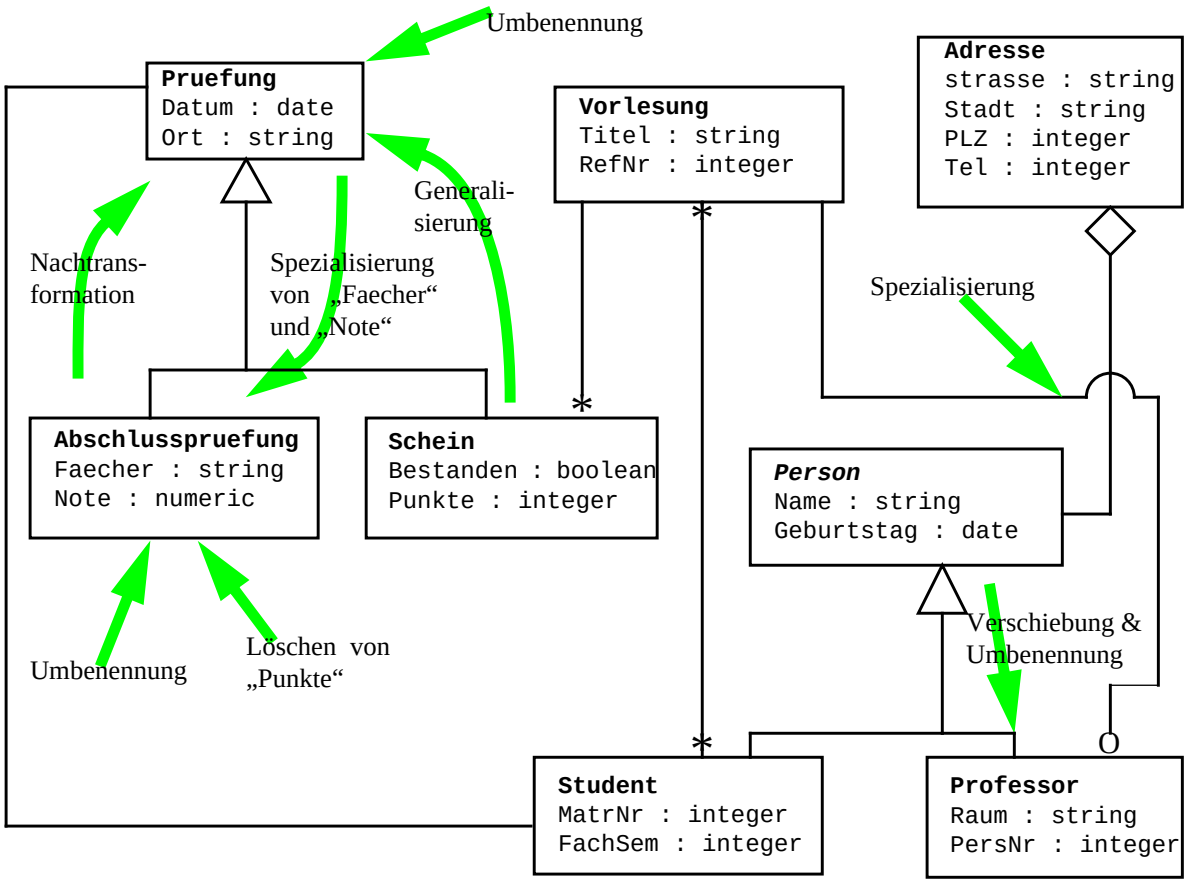


Abb. 3.6: Das endgültige objektorientierte Teilschema

Kapitel 4

Objekt-relationale Abbildung

In diesem Kapitel wird der Einsatz der Graphentechnik [Zün95] für die objekt-relationale Abbildung beschrieben. In einem ersten Abschnitt wird das relationale Datenmodell kurz dargestellt. Danach wird das auf dem ODMG-2.0 Datenmodell basierende Migrations-Meta-Modell (M^3) vorgestellt. Der dritte Abschnitt befaßt sich mit den semantischen Anreicherungen, die während der Abbildung ins M^3 berücksichtigt werden. Die initiale Abbildung von dem relationalen Datenbankschema in ein konzeptionelles Schema (das M^3) wird in einem vierten Abschnitt vorgestellt. Anschließend werden dem Leser die Restrukturierungstransformationen näher gebracht. Diese ermöglichen, das nun existierende konzeptionelle Schema in eine Form zu bringen, die die objektorientierten Eigenschaften ausnutzt.

4.1 Das relationale Datenmodell

Ein Datenmodell ist eine Sammlung von Konzepten zur Datenbeschreibung und -strukturierung. In dieser Arbeit wird nur der *statische* Anteil des Datenbankschemas vorgestellt. Die Beispiele sind aus dem Szenario und folglich in SQL. Die Operationen sind für die Analyse von Bedeutung aber nicht für die Schemamigration selber. Näheres dazu findet man in [Schä96], woran sich die Beschreibung des Datenmodells anlehnt.

Attribute

Ein Attribut ist die kleinste logische Einheit im relationalen Datenmodell. Jedem Attribut kann ein *Datentyp* zugeordnet werden. Der Datentyp legt die Menge aller zulässigen Werte für das Attribut fest. Standard-Datentypen sind beispielsweise `integer`, `varchar`, usw.. Ein Datentyp ist z.B. auch `datetime`, das ein Datum mit Uhrzeit speichern kann.

Relationen

Ein *Relationenschema* $R(A_1:D_1, \ldots, A_n:D_n)$ besteht aus dem Relationen-Namen R und einer Liste von Attributen A_i mit ihren respektiven Datentypen D_i, z.B.:

```
Vorlesung (RefNr : INTEGER, Titel : VARCHAR (50), DozentNr : INTEGER)
```

Attribute können auch mit Null-Werten besetzt werden, wenn dies nicht explizit ausgeschlossen wird (`NOT NULL`, siehe Abb. 3.3).

Die Menge aller *Tupel*, Ausprägungen der Relationenschemata, sind in den Daten enthalten. Sie werden mit Tabellen dargestellt.

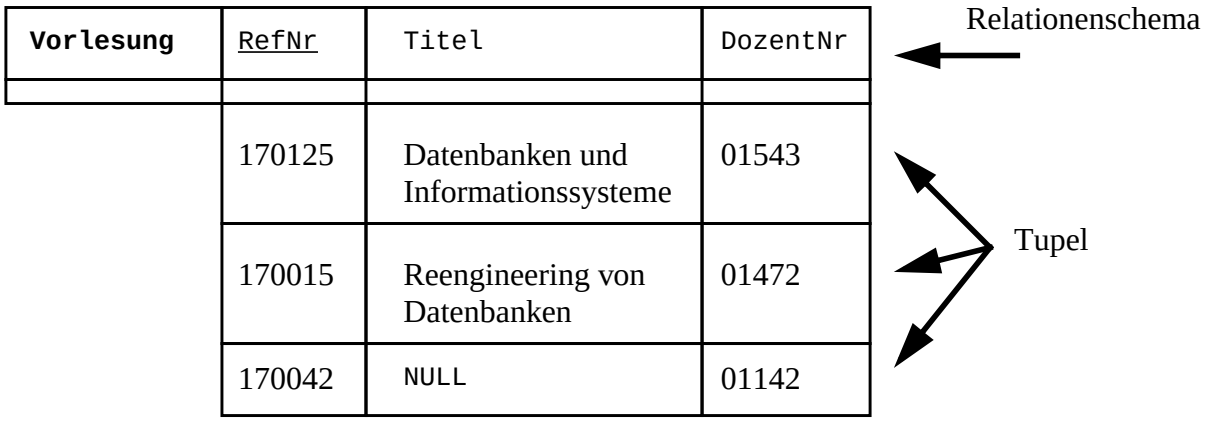


Abb. 4.1: Repräsentation einer Relation durch eine Tabelle

Eine Datenbank besteht aus vielen Relationen, deren Tupel miteinander verknüpft sind. Das Datenbankschema ist die Menge der Relationen und Schlüssel.

Schlüssel (KEY)

Alle Tupel einer Relation müssen verschieden sein. Deshalb werden sogenannte S*chlüssel* gebraucht, um die Tupel unterscheiden zu können. Ein Schüssel ist eine Attributmenge, die jedes Tupel eindeutig identifiziert. Es kann mehrere Schlüssel für eine Relation geben, die diese Eindeutigkeit erfüllen. Sie sind dann alle *Schlüsselkandidaten*. Einer von ihnen wird als *Primärschlüssel* (`PRIMARY KEY`) ausgezeichnet, die anderen sind *Sekundärschlüssel*. Sekundärschlüssel können in SQL mit `UNIQUE` angegeben werden. In einer Relation wird der Primärschlüssel unterstrichen (z.B. `RefNr`), im Schema wird er meistens explizit (siehe `primary key (RefNr)` in Abb. 3.3) angegeben. Das Attribut bzw. die Attribute, die den Schlüssel bilden, sollten als `NOT NULL` deklariert sein. Null-Werte in Schlüsseln würden der Eindeutigkeit entgegenwirken.

Fremdschlüssel (FOREIGN KEY)

Ein *Fremdschlüssel* einer Relation R1 auf eine Relation R2 ist eine Attributmenge, die folgende Bedingungen erfüllt:

- Die Attribute des Fremdschlüssels von R1 haben denselben Wertebereich wie die Attribute des Schlüssels von R2.
- Der Wert eines Fremdschlüssels in einem Tupel t1 in R1 kommt entweder als Wert eines Tupels t2 in R2 vor oder ist Null.

Fremdschlüssel können ebenfalls auf dieselbe Relation verweisen (R1 = R2).

4.2 Semantische Anreicherung

Oftmals sind wichtige semantische Informationen wie Integritätsbedingungen und Kontextbedingungen nicht explizit in der Schemadefinition einer Datenbankanwendung vorhanden. Deshalb muß das Schema semantisch angereichert werden. Diese Anreicherungen betreffen alle Komponenten des relationalen Datenmodells.

Varianten

Eine Relation kann aus mehreren „virtuellen“ Relationen bestehen. Eine Aufgabe der Analyse ist es diese herauszufinden. Diese „virtuellen“ Relationen einer Relation werden *Varianten* genannt. In Abb. 3.3 wird die Relation `Person` mit den zwei Varianten `Student` und `Professor` betrachtet. Ein `Professor` hat keine Anzahl an Fachsemestern (`FachSem`). Varianten sind die relationale Form der Generalisierung.

Schlüssel

Schlüssel sind nicht immer explizit angegeben und müssen deshalb identifiziert werden, siehe in Abb. 3.3 die Relation `Abschlusspruefung`. Aus Schlüsselkandidaten muß ein Primärschlüssel für eine Relation bestimmt werden. Oftmals sind nur die Primärschlüssel, aber keine alternativen Schlüssel angegeben, diese werden dann indentifiziert.

Fremdschlüssel

Im M^3 werden wie bei [FV95] drei Arten von Fremdschlüsselbeziehungen (IND, Inclusion Dependencies vgl. [Vos94]) unterstützt:

- Erstens die sogenannte *R-IND*, die eine Beziehung beschreibt, die *nicht* durch eine separate Relation (wie in Abb. 3.3 `StudVorl`) dargestellt ist. Im objektorientierten Schema wird sie zu einer [1-1]:[0-n] Assoziation.
- Dann die sogenannte *C-IND*, die auf der „Fremdschlüsselseite“ total ist, d.h. eine [1-1]:[1-n] Assoziation.
- Und als letztes die sogenannte *IsA-IND* (IIND), die Vererbungsbeziehungen im relationalen Schema ausdrücken.

Es gibt IND, die durch Transitivität redundant sind (vgl. [FV95]), beispielsweise existiert eine IND von einer Relation R_1 zu einer Relation R_2, eine weitere von R_2 zu einer Relation R_3 und eine von R_1 zu R_3. Wenn die Fremdschlüssel der jeweiligen IND die gleichen Informationen besitzen, dann ist die IND von R_1 zu R_3 überflüssig.

Optimierungen

Es gibt viele Optimierungen, die durch semantische Anreicherung aufgelöst werden können. Als Beispiel sei folgende Optimierung gewählt:

Die meisten IND sind *schlüsselbasiert* (vgl. [Vos94]), d.h. auf der rechten Seite ist ein Primärschlüssel. Nicht-schlüsselbasierte IND weisen auf eine Optimierungsstruktur hin (vgl. [FV95]). Die Attribute auf der rechten Seite beschreiben dann einen Primärschlüssel einer neuen Relation. Entweder muß die Relation dann separat als solche dargestellt werden oder die Optimierung muß aufgelöst werden.

4.3 Das Migrations-Meta-Modell

Das Migrations-Meta-Modell (M^3) basiert auf dem ODMG-2.0 Standard. Es ist ein objektorientiertes Schema mit den erwünschten Beziehungen (Assoziation und Aggregation) und Schüsseln, um ein *Forward-Engineering* in ein Schema, das solche Informationen braucht, zu gewährleisten.

Das ODMG-2.0 Objekt-Modell ist in [ODMG97] folgendermaßen zusammengefaßt:

- Die Basiseinheit ist das Objekt.
- Objekte können in Typen kategorisiert werden. Alle Objekte eines Typs haben dasselbe Verhalten und denselben Wertebereich für Zustände.
- Das Verhalten der Objekte wird durch eine Menge von Operationen bestimmt. Die Operationen sind im Objekttyp definiert.
- Der Zustand eines Objekts wird durch eine Menge von Eigenschaften definiert. Eigenschaften sind entweder Attribute von einem Objekt oder Beziehungen zwischen Objekten.

Im M^3 sind Objekttypen *Klassen*. Die Klasseneigenschaften sind *Attribute* und *Traversierungspfade*. *Schlüssel*, Beziehungen (*Assoziation, Aggregation)* und *Generalisierung* wie im ODMG-2.0 Objekt-Modell werden ebenfalls berücksichtigt.

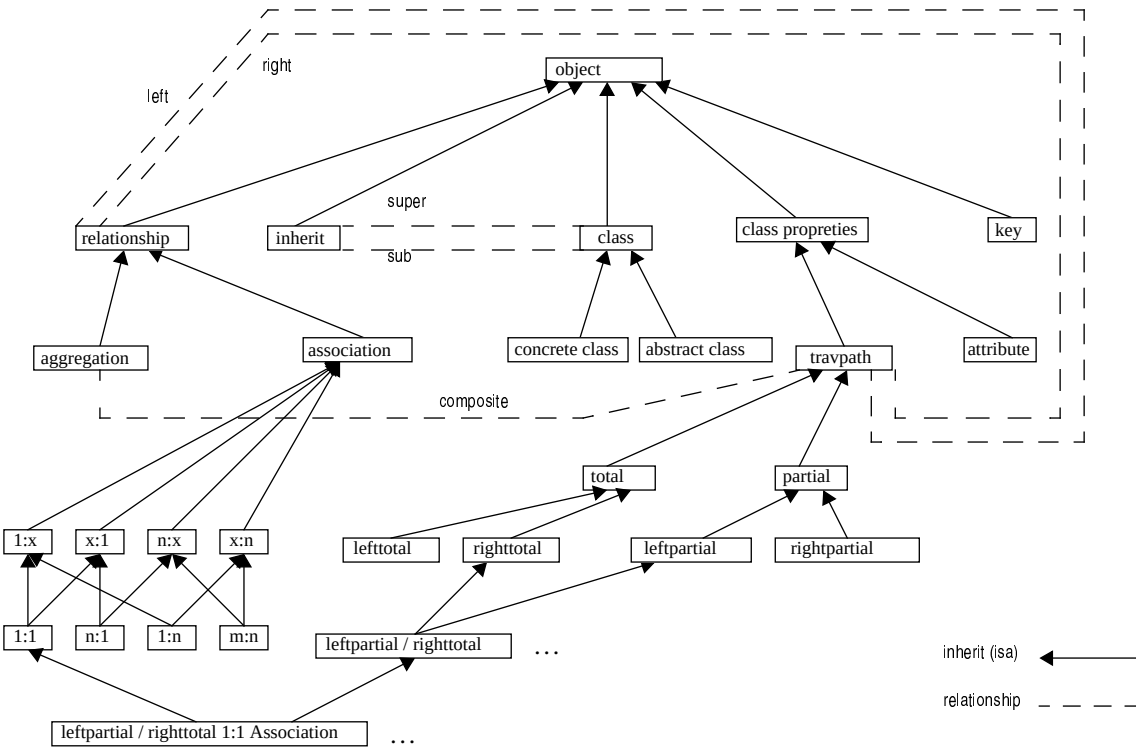


Abb. 4.2: Das Migrations-Meta-Modell

Die verschiedenen Elemente des M^3 werden im folgenden erklärt:

Klassen

Eine Klasse ist eine Schablone, nach der Objekte erzeugt werden können. Man unterscheidet konkrete und *abstrakte* Klassen. Von abstrakten Klassen werden keine Instanzen abgeleitet, sie dienen einer besseren Strukturierung des Schemas (siehe `Pruefung` im Beispiel-Szenario).

Klasseneigenschaften

- Attribute

Attribute werden für eine Klasse definiert, d.h. in einer Klasse dürfen nicht zwei gleiche Attribute vorkommen. In zwei verschiedenen Klassen dagegen ist dies erlaubt, da die Attribute dann über die Klasse eindeutig identifizierbar sind. Jedem Attribut wird ein Typ zugeordnet. Der Typ legt die Menge aller zulässigen Werte für das Attribut fest. Die Standard-Typen sind beispielsweise `integer`, `string`, usw.. Ein Typ kann aber auch komplexer (z.B. `Date`) oder eine Klasse sein.

- Traversierungspfade

Klassen haben Traversierungspfade, denen die Kardinalität der zugehörigen Assoziation mitgegeben werden. Beziehungen können auf einer Seite *total* oder *partiell* sein. Es wird zwischen den Kardinalitäten *Eins* (total [1-1] und partiel [0-1]) und *Viel* (total [1-n] und partiel [0-n]) unterschieden.

Schlüssel

Schlüssel im ODMG-2.0 Objekt-Modell haben - wie die Primärschlüssel im relationalen Datenmodell - die Aufgabe der eindeutigen Identifizierung von Objekten. Im M^3 werden sie nur für Abbildungen in Schemata gebraucht, die ohne Schlüssel nicht auskommen, wie beispielsweise relationale Schemata.

Beziehungen

- Assoziation

Eine Assoziation ist eine Beziehung zwischen Klassen. Sie ist eine logische Verbindung, ein Paar von Traversierungspfaden. Im M^3 werden 1:1, 1:n, n:1 und n:m-Beziehungen unterstützt.

- Aggregation

Bei der Aggregation werden Schemakomponenten zu einer neuen Schemakomponenten zusammengefaßt. In Abb. 3.6 gibt es eine Aggregation zwischen `Person` und `Adresse`. Die Klassen *`Person`* aggregiert zusätzlich zu einigen Attributen noch ein Objekt der Klasse `Adresse`.

Generalisierung

Bei der Generalisierung werden Klassen zu Obermengen zusammengefaßt, siehe Abb. 3.6 `Schein`, `Abschlusspruefung` und *`Pruefung`*. Die untergeordnete Klasse (*Unterklasse*) erbt alle Attribute der *Oberklasse*. Die Umkehrung wird *Spezialisierung* genannt.

4.4 (Initiale) Abbildung durch Tripelgraphgrammatiken

Die initiale Abbildung wird mit den von [Lef95] eingeführten Tripelgraphgrammatiken (TGG) modelliert, da diese *Transformation* und *Konsistenzerhaltung* ermöglichen.

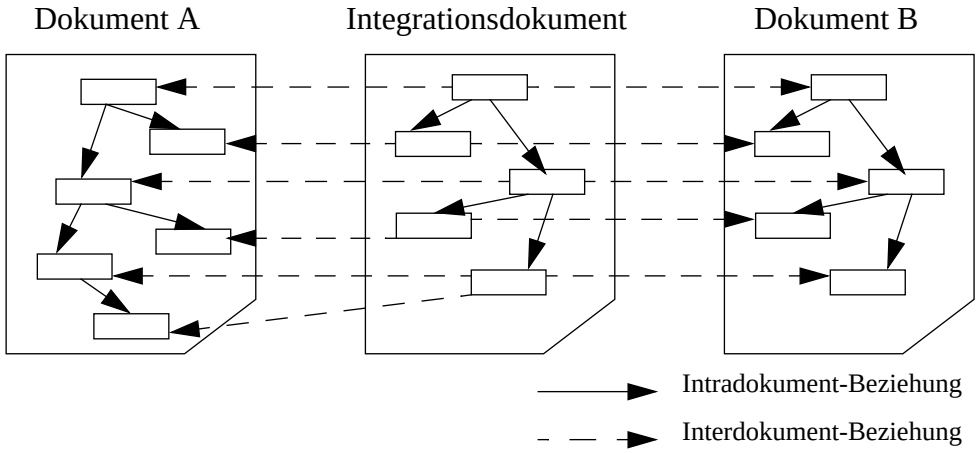


Abb. 4.3: Integration durch Tripelgraphgrammatiken

Die grundlegende Idee der Tripelgraphgrammatiken basiert auf der *Integration* von zwei Dokumenten mit Hilfe eines dritten sogenannten Integrationsdokuments. Durch die Angabe einer Menge von Tripelregeln können die zu integrierenden Dokumente eindeutig aufeinander abgebildet werden. Diese Abbildung zwischen den korrespondierenden Inkrementen wird durch das Integrationsdokument verwaltet.

Das Abbildungsschema entspricht dem Integrationsdokument und verwaltet die Abhängigkeit zwischen den beiden Schemata (SQL und M^3). Abb. 4.4 zeigt, wie die beiden Schemata und das Abbildungsschema miteinander verbunden sind. Es wurde wieder die OMT-Notation [OMT] benutzt. (MapVar steht für MapVariant, src für source, dst für destination, l_map für left mapping und r_map für right mapping.)

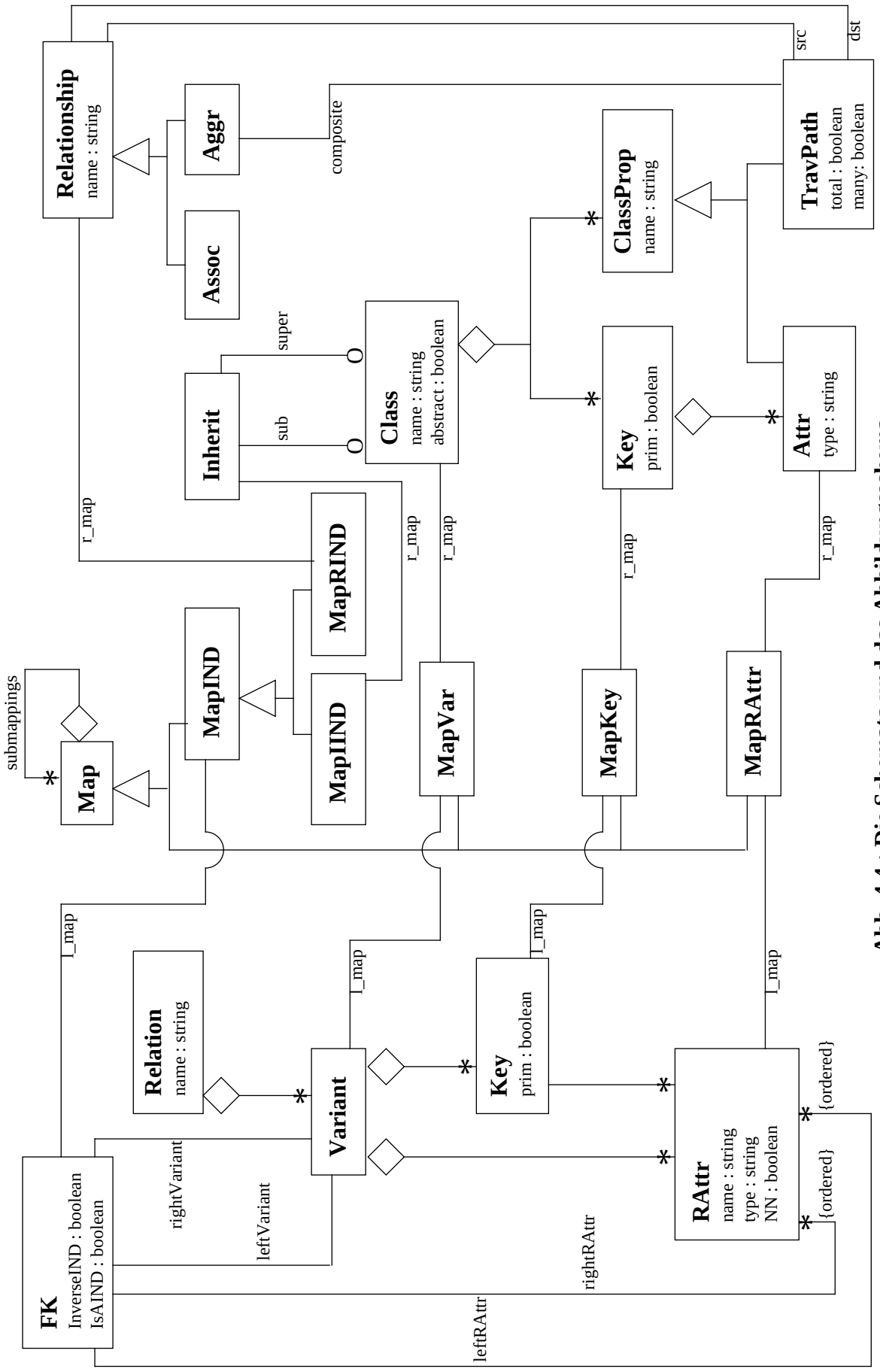


Abb. 4.4 : Die Schemata und das Abbildungsschema

Intern wird die Struktur in einem Syntaxgraph dargestellt. Abb. 4.5 zeigt einen Auszug der internen Struktur, die mit dem einfachen *Tripelregelsatz* aufgebaut wird. Die Grundlage für die Beschreibung des Syntaxgraphen stellt ein mathematisches Modell: die ***g**erichteten, **a**ttributierten, **k**noten- und **k**antenmarkierte Graphen* (gakk-Graphen). Ein gakk-Graph besteht aus unterschiedlich markierten Knoten, um Objekte verschiedener Typen zu modellieren, unterschiedlich markierten Kanten, um die Struktur und die Beziehungen zwischen solchen Objekten zu beschreiben, und als letztes aus einfachen Knotenattributen, um unstrukturierte Daten (z.B. Zahlen oder Zeichenketten) darzustellen.

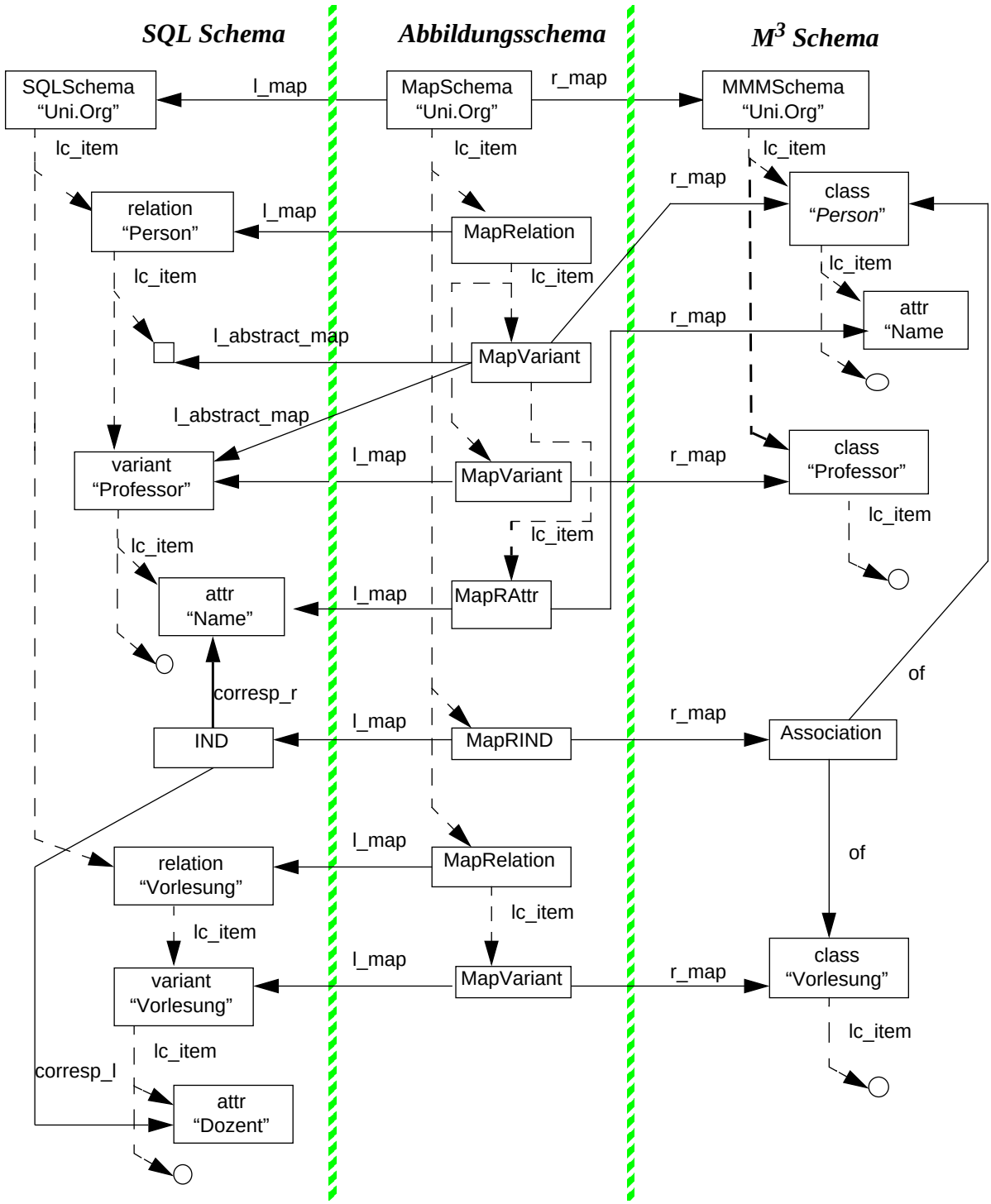


Abb. 4.5: Interner Syntaxgraph nach der initialen Abbildung

Die Schemata sind in Listen gespeichert, deshalb werden die obersten Ebene (SQL Schema, M^3 Schema) aufeinander abgebildet.

Aus der Relation `Person`, mit den Varianten `Professor` und `Student`, wird eine abstrakte Klasse *`Person`* erzeugt, die die gemeinsamen Attribute der beiden Varianten enthält. Das Viereck, das mit der Kante `lc_item` mit der Relation `Person` verbunden ist, soll die Variante `Student` darstellen. Das Attribut `Name` ist auch Teil dieser Variante, d.h. eine `lc_item` Kante verbindet auch hier die beiden Knoten. Der `MapRelation`-Knoten wird als interner Listenkopf im Abbildungsschema genutzt. Die Kreise sollen weitere Attribute symbolisieren, beim Abbildungsschema sind weitere Knoten aus Platzgründen nicht dargestellt worden.

Die Attribute werden in ihre entsprechende Klasse abgebildet, außer sie gehören zu der linken Seite einer IND, wie bei `Dozent` in `Vorlesung`. Dementsprechend werden dann die IND auf Assoziationen bzw. Generalisierungen (*Inherit*) abgebildet und den Klassen zugeordnet, die die korrespondierenden Attribute enthalten bzw. enthalten würden. Die Abbildung der Schlüssel ist hier nicht aufgeführt, die Zuordnung geht von jeder Variante zur entsprechenden Klasse.

Graphgrammatiken und Graphersetzungsregeln

Die Spezifizierung der Tripelgraphgrammatiken [Lef95] wurde in VARLET mit *Graphgrammatiken* vorgenommen. Die Grundlage für die Beschreibung dieser graphartigen Datenstruktur stellen die gakk-Graphen. Im Rahmen dieser Diplomarbeit wurde PROGRES (PROgrammierte GRaphErsetzungsSysteme) eingesetzt, um diese Graphen zu beschreiben und zu verändern. Als Grundlage dient ein *Graphschema*, das die Eigenschaften der Graphen festlegt. Als graphverändernde Operationen werden *Graphersetzungsregeln* spezifiziert. Eine Graphersetzungsregel besteht aus einer linken Regelseite, die den zu modifizierenden Teilgraphen beschreibt, d.h. im Graphen wird dieser Teilgraph gesucht. Ersetzt wird er durch den auf der rechten Regelseite beschriebenen Teilgraphen. Ein Rahmenwerk kann um diese Regeln implementiert werden - wo man sie parametrisiert mit den gewöhnlichen Kontrollstrukturen - aufrufen kann. Eine ausführliche Beschreibung dieses Graphentechnik-Ansatzes findet man in [Zün95].

Eine Seite einer Graphersetzungsregel kann folgende Konstrukte enthalten:

- **Knoten**
  Knoten werden durch Vierecke mit einem Knotenbezeichner und einem Typ (`‘4: Class_T` in Abb. 4.11) bzw. übergebenen Knotennamen (`‘1 = oldclass` in Abb. 4.11) dargestellt. Man unterscheidet zwischen *obligaten* (durchgezogene Linien) und *optionalen* Knoten (gestrichelte Linien). Optionale Knoten treten nur auf der linken Seite der Graphersetzungsregel auf. Knoten auf der rechten Seite, die unverändert von der linken übernommen werden, stellt man mit `1‘ = ‘1` (siehe Abb. 4.11) dar. `1‘` ist ein Knotenbezeichner. `‘1` repräsentiert den Knoten der linken Regelseite mit Knotenbezeichner `‘1`, beispielsweise könnte man auch `9‘ = ‘1` schreiben.

- **Knotenmengen**
  Eine Knotenmenge wird durch zwei Vierecke dargestellt, man unterscheidet auch hier zwischen obligaten und optionalen Knotenmengen. (siehe z.B. `‘2 = cps` in Abb. 4.11)

- **Kanten**
  Die Kanten sind gerichtet und werden mit einem Pfeil zwischen zwei Knoten(mengen) dargestellt. Die Kanten sind beschriftet (siehe `of` in Abb. 4.11).

- **Pfade**
  Ein Pfad ist eine Konkatenation von Kanten und ist ebenfalls beschriftet. Er wird durch einen Doppelpfeil zwischen zwei Knoten(mengen) dargestellt. Pfade treten nur auf der linken Seite der Graphersetzungsregel auf (siehe `lc_item` in Abb. 4.11).

- **Restriktionen**
  Eine Restriktion ist eine boolsche Bedingung, die an eine Knoten(menge) geknüpft wird. Dargestellt werden sie durch einen Doppelpfeil, der auf eine Knoten(menge) zeigt und mit der boolschen Bedingung annotiert ist. Restriktionen treten nur auf der linken Seite der Graphersetzungsregel auf.

- **Knotenattribute**
  Die Knotenattribute können unter der rechten Regelseite nach dem Schlüsselwort `transfer` gesetzt werden (siehe `4'.Name := '3.Name;` in Abb. 4.11).

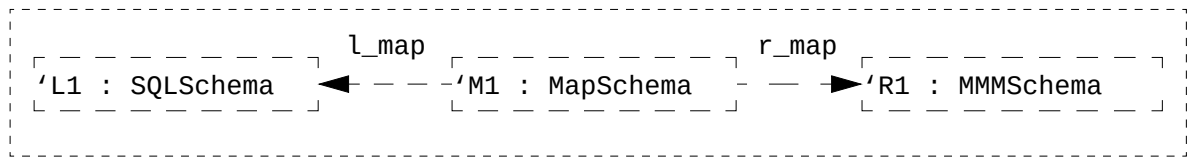
Einer Graphersetzungsregel können Knoten(mengen) übergeben und von ihr zurückgeliefert werden. Die übergebenen Knoten(mengen) werden durch ihren Namen identifiziert. Die Rückgabeknoten werden nach dem Schlüsselwort `return` aufgeführt und bekommen den entsprechenden Knotenbezeichner zugewiesen.

Tripelregeln sind *monotone* Graphersetzungsregeln, d.h. alle Knoten und Kanten der linken Regelseite werden in der rechten identisch ersetzt und mindestens ein neuer Knoten wird hinzugefügt (vgl [Lef95]). Diese Einschränkung ermöglicht eine abgekürzte PROGRES-Notation. Beide Seiten lassen sich zu einer verschmelzen, die linke Seite der Tripelregel wird durch durchgezogene Linien und die rechte mit gestrichelten Linien dargestellt (vgl.[Lef95]).

Die Schema-Abbildungsregel

Das M^3 Schema wird durch die initiale Abbildung aufgebaut. Die Abbildungsvorschrift wird durch Tripelregeln formalisiert. Die `Map_Schema`-Regel wird als erstes vorgestellt.

```
mapping Map_Schema( out relschema : SQLSchema ; out ooschema : MMMSchema ;
                    out mapschema : MapSchema) =
```




```
   transfer 'R1.Name := 'L1.Name;
   return relschema := 'L1;
          ooschema := 'R1;
          mapschema := 'M1;
end;
```

Abb. 4.6: Die Tripelregel für die Schema-Abbildung

Die Regel besagt, daß zu einem `SQLSchema` ein `MMMSchema` gehört. Der `MapSchema`-Knoten stellt die Korrespondenz zwischen ihnen her. Die linke Regelseite ist leer. Diese Regel stellt die simultane Erzeugung beider Schemata dar. Aus ihr können automatisch eine Vorwärtstransformtion, eine Rückwärtstransformation und eine Konsistenzprüfung abgeleitet werden (vgl. [Lef95]). Abb. 4.7 zeigt die abgeleiteten Regeln zur `Map_Schema`-Regel.

Zu einem existierenden SQL Schema erzeugt die Vorwärtstransformtion (`forward rule`) ein M^3 Schema und die Korrespondenz. Zu einem M^3 Schema erzeugt die Rückwärtstransformation (`backward rule`) ein SQL Schema und die Korrespondenz. Die Konsistenzprüfung (`relating rule`) richtet eine Beziehung zwischen den existierenden Schemata ein.

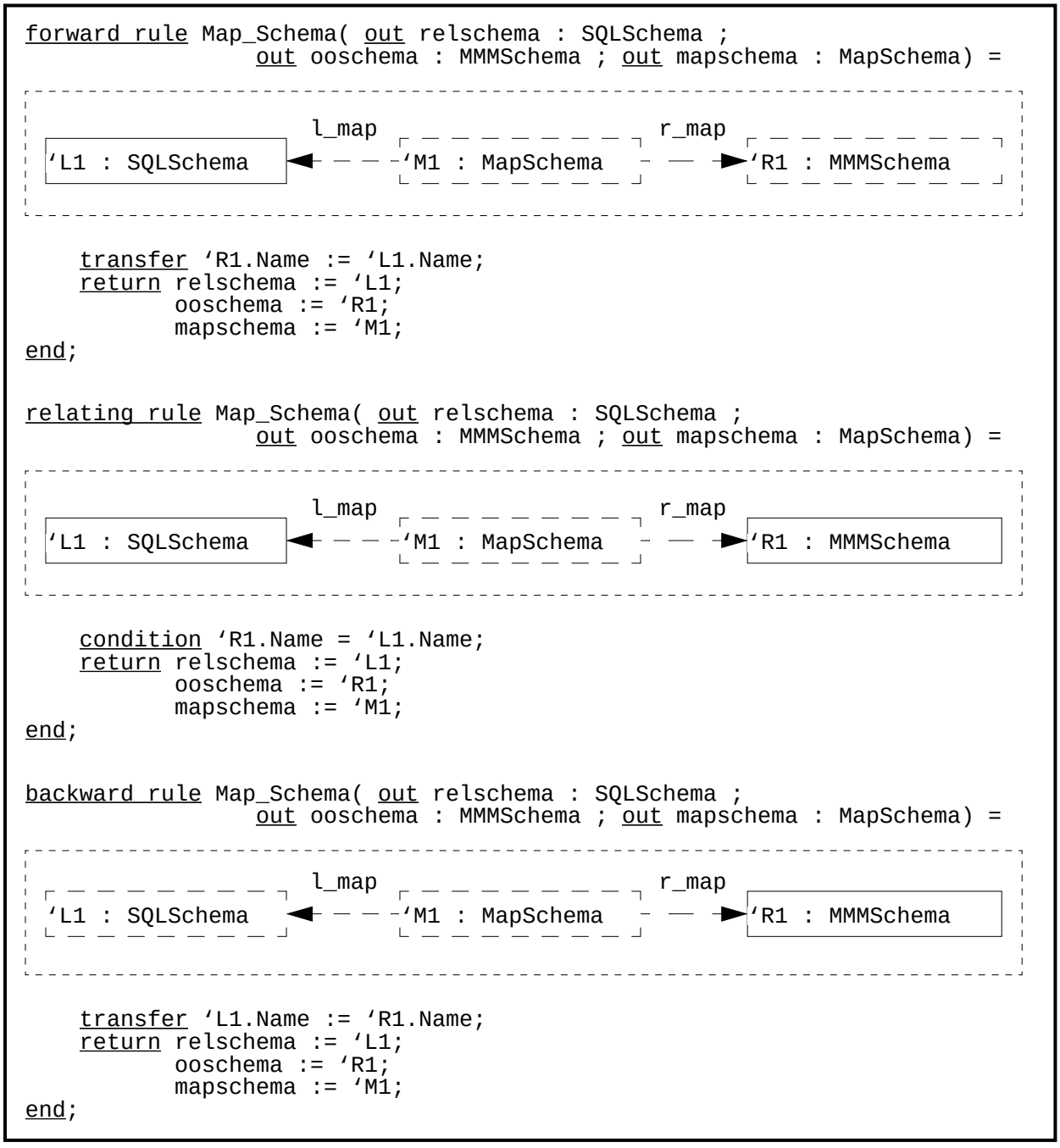


Abb. 4.7: Die abgeleiteten Tripelregeln für die Schema-Abbildung

Die initiale Abbildung erzeugt zu einem SQL Schema ein M^3 Schema, deshalb werden im folgenden nur noch die Vorwärtstransformationen (`forward rule`) vorgestellt.

In `forward rule Map_Schema` wird der `SQLSchema`-Knoten in dem Graphen gesucht. Es wird davon ausgegangen, daß nicht zwei relationale Schemata in ein entsprechendes objektorientiertes abgebildet werden sollen und daß deshalb *genau ein* `SQLSchema`-Knoten existiert. Dieser muß obligat sein, was durch einen Knoten mit durchgezogener Linie gekennzeichnet wird. Die gestrichelten Knoten und Kanten bedeuten, daß diese bei der Regelausführung neu erzeugt werden. Das `MMMSchema` bekommt den gleichen Namen wie das `SQLSchema` und alle drei Knoten werden zurückgegeben. Dadurch kann das komplette relationale Schema erreicht werden. Die beiden anderen Strukturen (Schemata) werden ausgehend von den Schema-Knoten aufgebaut.

Im folgenden werden negative Knoten in Tripelgraphgrammatiken angewendet, die im Allgemeinen nicht zugelassen sind. Negative Knoten in Tripelgraphgrammatiken erschweren das Parsen von Graphen, und bei der inkrementellen Konsistenzprüfung wird der Aufwand unvertretbar, siehe [Schü94].

Während der initialen Abbildung werden nur wenige Tripelregeln eingesetzt. Die Einschränkung der negativen Knoten ist leicht überschaubar. Der Aufwand beim Parsen und der Konsistenzprüfung ändert sich nicht wesentlich.

Die Relationen-Abbildungsregel

Die Regel `Map_Relation` gestaltet sich ähnlich, außer daß kein entsprechender Knoten im M^3 Schema erzeugt wird. Davon ausgehend werden die Variante(n) abgebildet.

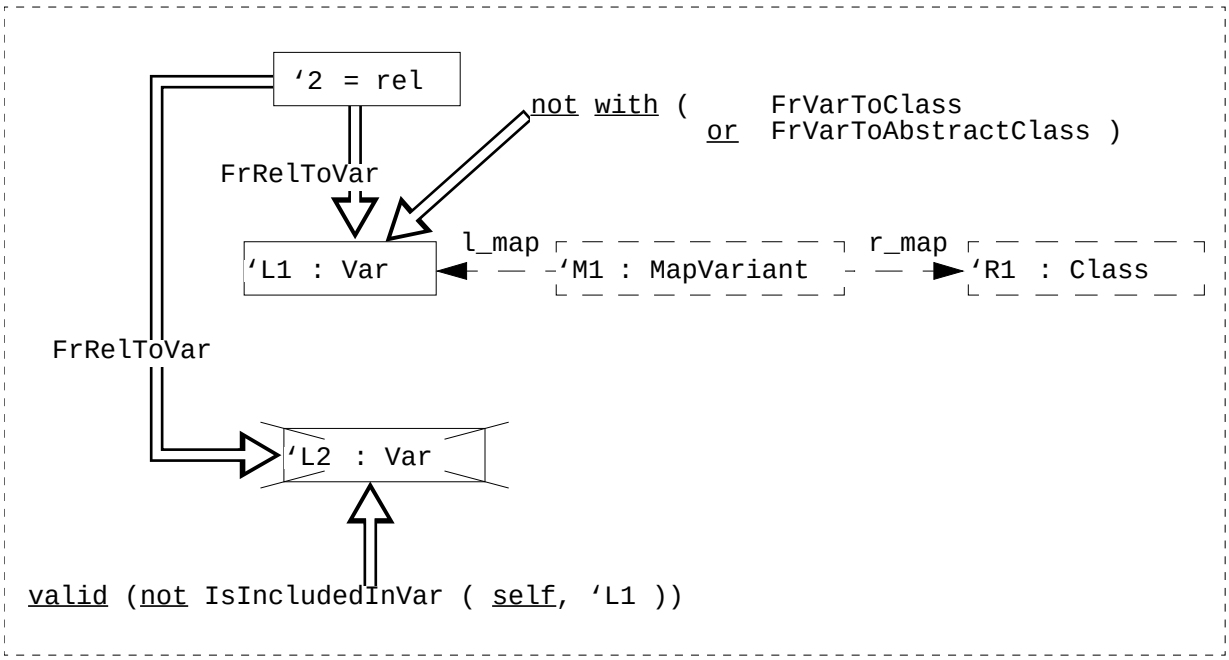
Die Varianten-Abbildungsregeln

In einem Regelwerk (siehe Anhang A) werden nacheinander sechs Tripelregeln solange ausprobiert bis alle Varianten einer Relation abgebildet sind. Hier wird nur eine dieser Regeln vorgestellt, siehe Anhang A für die restlichen.

Die Regel in Abb. 4.8 bildet die kleinste Variante einer Relation auf die Wurzelklasse der Vererbungshierarchie dieser Relation ab. Mit der kleinsten Variante ist die Variante gemeint, die am wenigsten Attribute hat und in allen anderen enthalten ist. Diese Regel wird für alle Relationen mit einer Variante angewendet. Gibt es keine solche Variante, wird eine andere Regel ausgeführt.

Für eine Relation `'2` wird eine Variante `'L1` gesucht, welche noch nicht auf eine (abstrakte) Klasse abgebildet wurde. Diese Variante muß genauso viele Attribute haben, wie die Variante der Relation mit den wenigsten Attributen (`condition`-Teil). Hinzu kommt, daß es keine Variante `'L2` geben darf, gekennzeichnet durch den mit einem Kreuz durchgestrichenen Knoten, die in `'L1` enthalten ist. Die gestrichelten Teile werden bei der Ausführung der Regel im Graphen hinzugefügt. In der `mapping`-Regel gibt es eine zusätzliche Restriktion auf `'R1`, die besagt daß die Klasse keine Oberklasse besitzen darf. Diese Restriktion in der `forward rule` aufzuführen macht keinen Sinn, weil die Klasse neu erzeugt wird.

```
forward rule Map_VarToClassWhichIsRootOfHierarchy( rel : Relation ;
                                out mapvar : MapVariant ; out c : Class) =
```




```
    condition VarWithMinRAttrs ( rel ) = NoOfRAttrsInVar ( 'L1 );
    transfer R1'.Name := '2.Name;
    return mapvar := M1';
           c := R1';
end;
```

Abb. 4.8: Die Tripelregel, die einer Variante eine Klasse ohne Oberklasse zuordnet

Die anderen Regeln gestalten sich ähnlich. Restriktionen sind für all diese Regeln nötig, um die Definition einer Reihenfolge, in der sie angewendet werden müssen, zu vermeiden. Die anderen fünf Regeln bilden Varianten folgendermaßen ab:

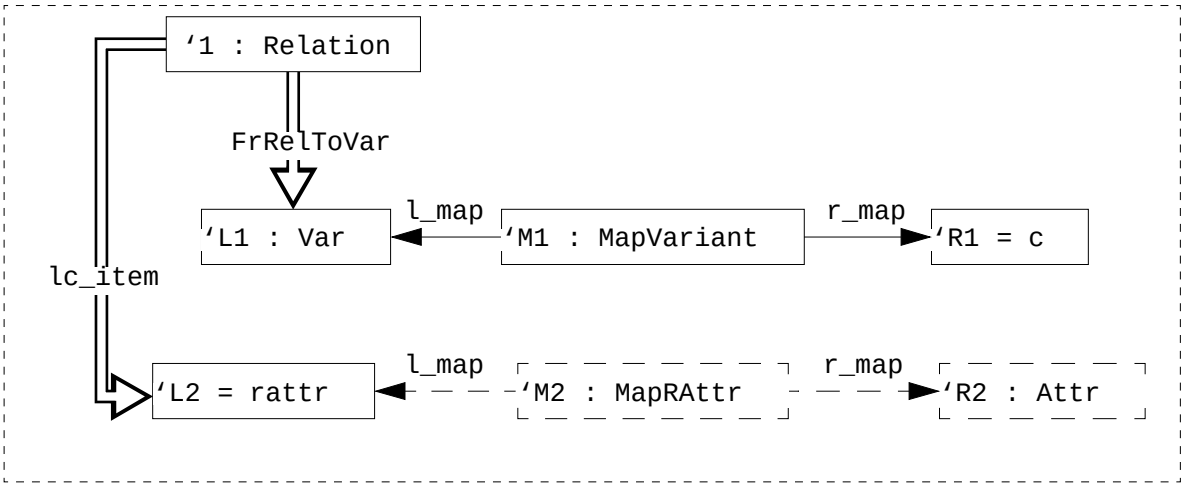
- zwei Varianten auf eine abstrakte Klasse ohne Oberklasse
  (siehe *Person* in Abb. 3.4)
- zwei Varianten auf eine abstrakte Unterklasse
- eine Variante auf eine existierende abstrakte Klasse
- eine Variante auf eine Unterklasse einer abstrakten Klasse
  (siehe Professor und Student in Abb. 3.4)
- eine Variante auf eine Unterklasse einer Klasse
  (siehe Abschlusspruefung in Abb. 3.6)

Mehr Regeln sind nicht nötig, da mehrfach Vererbung in VARLET verboten ist. Dies hat den Grund in einer späteren gewünschten Anbindung an die Programmiersprache *Java*, die dieses nicht unterstützt. Die Regeln sind im Anhang A aufgeführt.

Die Primärschlüssel- und die Attribut-Abbildungsregeln

Die Abbildung der Schlüssel und der Attribute gestaltet sich ebenfalls einfach. Die Schlüsselattribute werden in die zu der Variante korrespondierende (abstrakte) Klasse abgebildet.

```
forward rule Map_RAttr( c : Class ; rattr : RAttr ; out mapattr : MapRAttr ;
                        out attr : Attr) =
```




```
  condition '1.Name = substr( 'R1.Name, 0, pos("#",'R1.Name,0) - 1);
  transfer 'R2.Name := 'L2.Name;
           'R2.Type := 'L2.RType;
  return mapattr := 'M2;
         attr := 'R2;
end;
```

Abb. 4.9: Die Tripelregel für die Attributezuordnung

Bei den Attributen verhält es sich ähnlich, nur daß die auf der linken Seite einer IND stehenden Attribute gar nicht abgebildet werden und dementsprechend nicht übergeben werden.

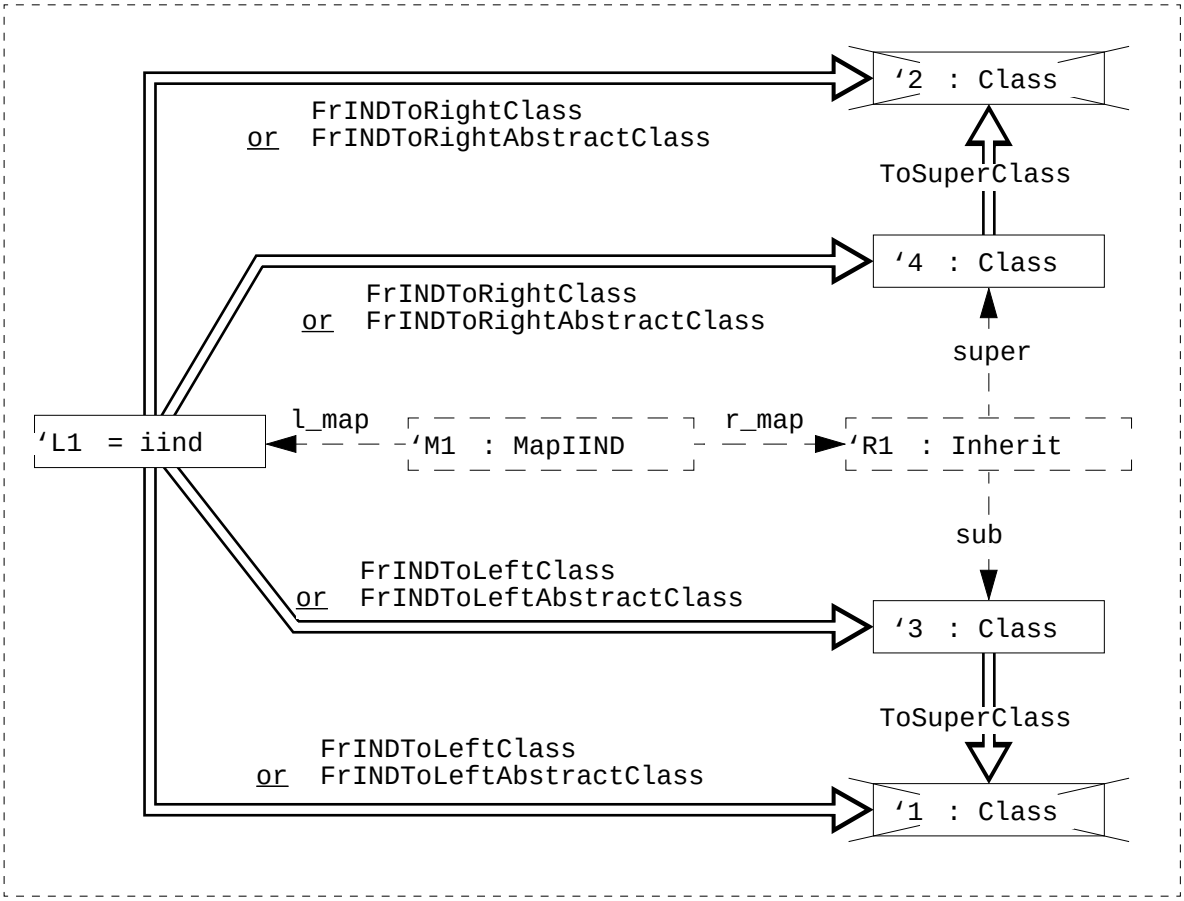
Da das Rahmenwerk keine *sinnvolle* Namensänderungen vornehmen kann, muß dies später durch den Benutzer gemacht werden. Deshalb bekommt jede Klasse erstmal den Namen der entsprechenden Relation (gegebenenfalls mit `#1`, `#2`, ... am Ende). Varianten haben keine Namen, deshalb müssen die Namen der Relation und der Klasse im `condition`-Teil überprüft werden.

`substr( 'R1.Name, 0, pos("#",'R1.Name,0) - 1)` schneidet die Endungen `#1`, `#2`, ... ab.

Die Beziehungs-Abbildungsregeln

Zuletzt müssen noch die Fremdschlüsselbeziehungen abgebildet werden. Da bestehen zwei Möglichkeiten, einmal die Abbildung auf eine Assoziation und andererseits auf eine Vererbungsbeziehung.

```
forward rule Map_IIND( iind : IND ; out mapiind : MapIIND ;
             out class1 : Class ; out class2 : Class ; out inh : Inherit) =
```




```
    return class1 := '3;
           class2 := '4;
           mapiind := 'M1;
           inh := 'R1;
end;
```

Abb. 4.10: Die Tripelregel, die einer IIND eine Vererbungsbeziehung zuordnet

Die Unterscheidung, wann welche Abbildung vollzogen werden muß, ist nicht schwer, da nur die IIND (Is_A-Beziehung) einer Vererbungsbeziehung entsprechen. Die restlichen werden auf eine Assoziation abgebildet, mit entsprechenden Kardinalitäten, was davon abhängt, ob die IND *schlüsselbasiert* oder/und *invers* (C_IND) ist.

Die negativen Knoten, die Klassen repräsentieren, werden gebraucht, um die Vererbungsbeziehung richtig einordnen zu können. Eine Generalisierung wird den „obersten“ Klassen zugeordnet, die von den - an der IIND beteiligten - Relationen abgebildet worden ist. Diese „obersten“ Klassen werden aus den Klassen ausgewählt, die von den Varianten abgebildet wurde, für die die IIND nicht optional ist. Sind diese beiden Klassen indentifiziert, werden sie an die `Inherit` mit einer `sub`-Kante für die linke Klasse und einer `super`-Kante für die rechte Klasse verbunden. Das paßt zu der *nicht-Abbildung* der Attribute, die auf der linken Seite einer IND stehen. Die Unterklasse (linke Klasse) erbt alle Attribute der Oberklasse (rechte Klasse).

Mit diesen wenigen, einmal festgeschriebenen Tripelregeln läßt sich ein relationales Schema

in ein objektorientiertes Schema übersetzen. Dieses besitzt zwar u.U. noch nicht alle wünschenswerten Eigenschaften, dafür geschieht diese Abbildung *vollautomatisch*.

Die Tripelgraphgrammatiken wurden schon in [JSZ96] und [Hol97] erweitert und für die Schemamigration eingesetzt. Anders als bei diesem Ansatz haben sie aber einen Regelsatz konstruiert, der das relationale *direkt* in das endgültige objektorientierte Schema übersetzt. Weiterhin haben sie viele vereinfachende Anforderungen an das relationale Schema gestellt: Fremdschlüssel dürfen nur aus einem Attribut bestehen, Varianten werden nicht berücksichtigt, usw.. Es gibt viele Möglichkeiten relationale Schemakomponenten in äquivalente objektorientierte zu übersetzen. Ein großer Regelsatz ist die Folge, der dementsprechend komplex ist. Außerdem ist nicht sicher, daß alle gewünschten bzw. möglichen Abbildungen berücksichtigt werden. Des Weiteren ist unsicher, ob der Regelsatz immer noch Konsistent und Minimal ist. Es ist nicht auszuschließen, daß redundante Regeln vorhanden sind, d.h. daß einige Regeln können durch andere simuliert werden. Um diese Probleme zu lösen, werden die Tripelgraphgrammatiken mit Restrukturierungstransformationen kombiniert (siehe [JZ98]).

Im VARLET-Ansatz bekommt der Benutzer erst einmal die objektorientierte Sicht des relationalen Schemas. Er muß nicht im relationalen Schema alle gewünschte objektorientierte Eigenschaften für dieses Schema identifizieren. Dementsprechend ist der Regelsatz klein und seine Komplexität verringert sich. Zudem sind die Regeln leicht auf andere Schemata zu übertragen. Das Reengineering bzw. die Migration mit dem M^3 gestaltet sich dann wieder einfach. Hierzu werden die Restrukturierungstransformationen gebraucht, um das Schema zu optimieren und die Vorteile einer objektorientierten Datenbank ausnutzen zu können.

Die Tripelregeln müssen für verschiedene konkrete Datenmodelle, die in das M^3 abgebildet werden sollen, abgeändert werden. Die Restrukturierungstransformationen formen das M^3 um und müssen nicht angepaßt werden, sondern nur einmal spezifiziert und implementiert werden. Deshalb ist es von Vorteil wenig Tripelregeln zu haben.

4.5 Restrukturierungstransformationen

Die Restrukturierungstransformationen vervollständigen das „erste“ objektorientierte Schema. Eine Transformation ist beispielsweise die Umwandlung einer Klasse in eine Assoziation (siehe `StudVorl` in Abb. 3.4). Zwei weitere sind die Generalisierung (anlegen einer Oberklasse) und Spezialisierung (anlegen einer Unterklasse) von Klassen. Es folgt ein Überblick von Standard-Restrukturierungstransformationen.

- Umwandlung einer Klasse in eine Assoziation
- Umwandlung einer Assoziation in eine Klasse
- Generalisierung
- Spezialisierung
- Abspaltung einer Klasse von einer Klasse
- Verschmelzen von zwei Klassen zu einer Klasse
- Umwandlung einer Assoziation in eine Aggregation

- Umwandlung einer Aggregation in eine Assoziation
- Verschiebung von Attributen in einer Vererbungsstruktur
- Veränderung der Kardinalität der Traversierungspfade
- Erzeugen einer Klasse, eines Attributes oder einer Assoziation
- Umbenennung bzw. Typänderung
- Löschen

Die Restrukturierungstransformationen verändern das Schema und somit auch die Informationskapazität.

Informationskapazität:
Die Informationskapazität eines Schemas ist durch die Menge aller gültigen Zustände der Komponenten dieses Schemas definiert. Das heißt die Menge aller möglichen verschiedenen Konstellationen der Instanzen und ihren Beziehungen untereinander.

Eine Klassifizierung der Transformationen, in Bezug auf den Begriff der Informationskapazität, ist u.a. in [BP96, JZ98] zu finden. Dort wird eine Aufteilung der Transformationen in drei Arten vorgeschlagen.

- **Informationskapazitäts-erhaltende Transformation**
  Die Quell- und die Zielschemakomponenten enthalten genau die gleichen Informationen. Eine Quellschemakomponente gehört zu genau einer Zielschemakomponente und umgekehrt. Diese Transformationen werden auch *Äquivalenztransformationen* genannt.

- **Informationskapazitäts-erweiternde Transformation**
  Alle Quellschemakomponenten können durch die Zielschemakomponenten beschrieben werden. Das Schema gewinnt durch die Transformation an Information.

- **Informationskapazitäts-reduzierende Transformation**
  Alle Zielschemakomponenten können durch die Quellschemakomponenten beschrieben werden. Das Schema verliert durch die Transformation an Information.

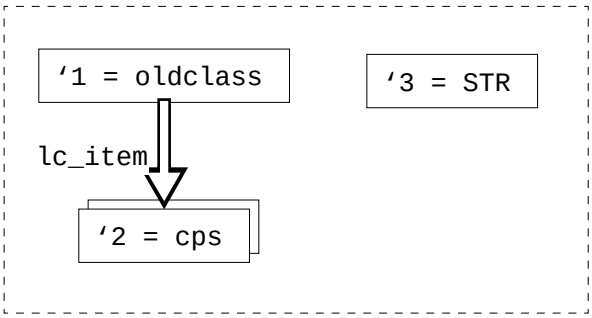
Eine formale Definition dieser Begriffe auf Basis der Graphgrammatik-Theorie findet sich in [JZ98]. Dort ist neben der *Strukturtransformation* auch die *Instanzabbildung* beschrieben. In dieser Arbeit wird nur die Strukturtransformation betrachtet.

Die Spezifikation der Restrukturierungstransformationen wurde in VARLET mit *Graphgrammatiken* vorgenommen. Eine Transformation spezifiziert eine Änderung des Datenbankschemas, wofür sich die Graphersetzungsregeln besonders eignen, da das Schema als Graph betrachtet werden kann.

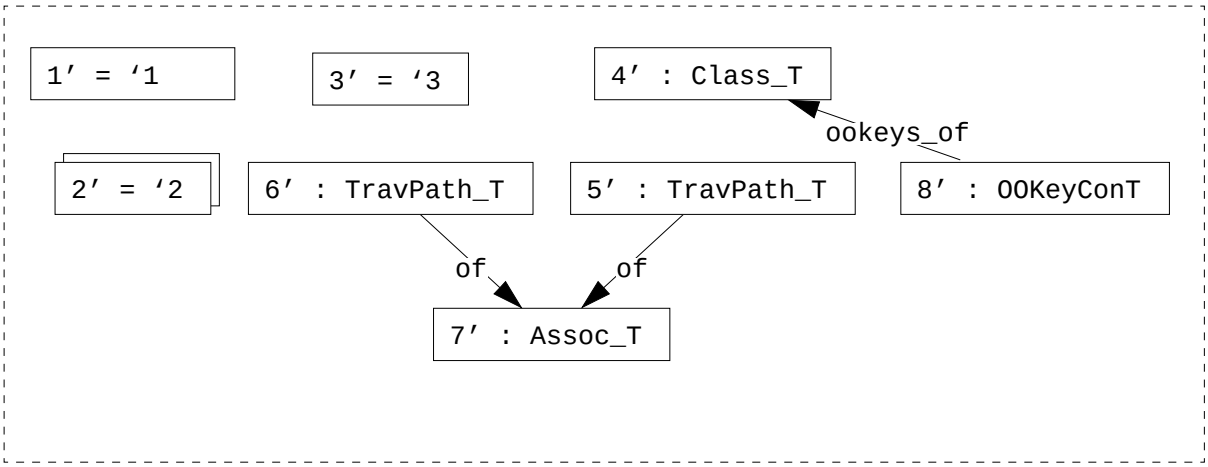
Informationskapazitäts-erhaltende Transformation

Eine Graphersetzungsregel (`production`) besteht aus einer linken und einer rechten Seite (siehe Abb. 4.11). `SplitClass` spaltet eine Klasse. Die Klasse, ihre Klasseneigenschaften (Attribute und Traversierungspfade (`TravPath`)) - die in die neue Klasse sollen - und ein `STRING`-Knoten - der den Namen der neuen Klasse enthält - bilden die linke Seite. Die besondere Rolle des `STRING`-Knoten ‘3 wird im nächsten Kapitel erklärt.

```
production SplitClass( oldclass : Class ; cps : ClassProp [1:n] ;
                  STR : STRING ; out newclass : Class ; out assoc : Assoc ;
                  out tpold, tpnew : TravPath ; out ookeycon : OOKeyCon ) =
```



::=




```
   transfer 4’.Name := ‘3.Name;
            4’.abstract := false;
            6’.NN := true;
            6’.many := false;
            5’.NN := true;
            5’.many := false;
            5’.Name := ‘1.Name;
            6’.Name := ‘3.Name;
   return newclass := 4’;
          assoc := 7’;
          tpold := 6’;
          tpnew := 5’;
          ookeycon := 8’;
end;
```

Abb. 4.11: Die Informationskapazitäts-erhaltende Transformation `SplitClass`

Diese Knoten werden in dem Graphschema gesucht und identifiziert. Außerdem wird der Pfad `lc_item` überprüft, um zu gewährleisten, daß die ausgewählten Klasseneigenschaften auch in der Ausgangsklasse enthalten sind. Ist dies erfüllt, wird die linke durch die rechte Seite ersetzt. Dabei ist anzumerken, daß Pfade (wie `lc_item`) nur auf der linken Seite vorkommen dürfen und somit bei der Ersetzung nicht berücksichtigt werden. Das heißt, daß der Pfad `lc_item` zwar überprüft aber keineswegs gelöscht wird.

Eine Klasse und eine Assoziation mit ihren Traversierungspfaden werden erzeugt und die entsprechenden Attribute werden im `transfer`-Teil belegt. Die Neuordnung der übergebenen Klasseneigenschaften (`'2`), von der alten (`'1`) auf die neue (`4'`) Klasse und die Einordnung der beiden `TravPath` (`5' & 6'`) in die jeweilige Klasse geschieht durch das Rahmenwerk. Der Knoten `OOKeyConT` (`'8`) dient als Listenkopf für die Verwaltung der möglichen Schlüssel, die im M^3 zugelassen sind. Diese Restrukturierungstransformation wurde z.B. für das Abspalten der Klasse `Adresse` von der Klasse *`Person`* in dem Beispiel-Szenario verwendet.

Die Äquivalenztransformationen sind *reversibel*, d.h. jede ausgeführte Transformation kann mit Hilfe seiner Inversen rückgängig gemacht werden. Die inverse Transformation zu `Split-Class` ist `MergeClass`. Sie entsteht, wenn die Regel `SplitClass` einfach nur „umgedreht" wird, linke Seite wird zu rechter Seite und umgekehrt. Die Pfade und das Rahmenwerk müssen angepaßt werden, dies geschieht ebenfalls symmetrisch. Ein formaler Beweis hierzu ist wiederum in [JZ98] zu finden.

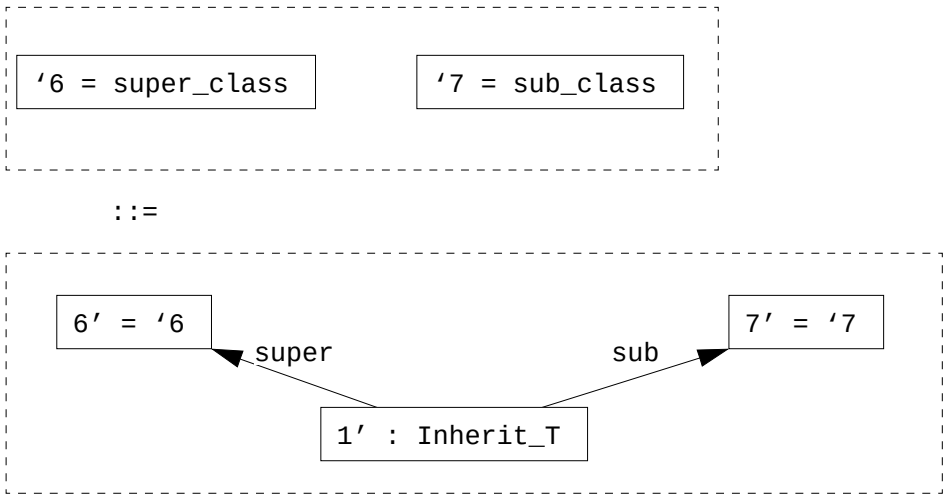
Die folgenden Transformationen sind ebenfalls reversibel:

- `ClassToRelationship`, die eine Klasse in eine Assoziation transformiert, siehe `StudVorl` in Abb. 3.4 & 3.6
- `RelationshipToClass`, die Umkehrung
- `Rename` (Umbenennen eines Objektes), ist zu sich selbst invers

Informationskapazitäts-erweiternde Transformation

`GeneralizeExisting` ist eine solche, siehe Abb. 4.9. Für die übergebene Oberklasse (`'6`) und für die Unterklasse (`'7`) wird ein `Inherit`-Knoten erzeugt, der die Vererbung repräsentiert, und mit den `super-` und `sub-`Kanten verbunden.

```
production DHM_GeneralizeExisting( super_class, sub_class : Class ;
                                   out inherit : Inherit) =
```




```
    return inherit := 1';
end;
```

Abb. 4.12: Die Informationskapazitäts-erweiternde Transformation `GeneralizeExisting`

Eine weitere Informationskapazitäts-erweiternde Transformation ist `Aggregate`, die eine Assoziation in eine Aggregation transformiert, siehe `Adresse` in Abb. 3.5.

Zu dieser Kategorie von Transformationen gehört auch `GeneralizeNew`, die eine Oberklasse für eine gegebene Klasse anlegt, siehe *`Pruefung`*, `Schein` und `Abschlusspruefung` in Abb. 3.5. Dieser Kategorie gehören `SpecializeNew` (legt Unterklasse an), `GeneralizeAttr` (erzeugt ein äquivalentes Attribut in der Oberklasse) und `SpecializeAttr` (erzeugt ein äquivalentes Attribut in der Unterklasse) ebenfalls an.

Weitere ähnliche Transformationen sind `CreateClass` (erzeugt eine neue Klasse), `CreateAttr` (erzeugt ein neues Attribut) und `CreateAssoc` (erzeugt eine neue Assoziation). Hinzu kommt noch die spezielle Operation `SetUnique` die einen Schlüssel konstruiert. Die *kardinalitätsverändernden* Transformationen `SetPartial` (setzt eine `TravPath` auf [0-x]) und `SetMany` (setzt eine `TravPath` auf [x-n]) sind ebenfalls Informationskapazitätserweiternd.

Informationskapazitäts-reduzierende Transformation

`DisAggregate`, die Umkehrung von `Aggregate`, die eine Aggregation zurück in eine Assoziation umwandelt ist eine Informationskapazitäts-reduzierende Transformation.

Transformationen, die ebenfalls in diese Kategorie gehören, sind die *kardinalitätsverändernden* Transformationen `SetTotal` (setzt eine `TravPath` auf [1-x]) und `SetOne` (setzt eine `TravPath` auf [x-1]) bzw. die Löschoperation `Remove`, der alle Objekte übergeben werden dürfen.

Es gibt aber auch Transformationen, die sich etwas anders verhalten, wie zum Beispiel `MoveClassPropsInHierarchy` (verschiebt Attribute in einer Vererbungsstruktur) und `ChangeType` (Typänderung eines Attributes). Es sind zwei Transformationen, die die Informationskapazität sowohl erweitern als auch reduzieren können. Die Zuordnung in die jeweilige Kategorie hängt vom jeweiligen Aufruf ab. `MoveClassPropsInHierarchy` ist für „`Punkte`“ von *`Pruefung`* nach `Schein` in Abb 3.7. Informationskapazitäts-reduzierend. Sie wäre Informationskapazitäts-erweiternd, wenn ein Attribut in der Vererbungshierarchie nach „oben“ geschoben wird.

Zusammenfassung

Die hier kurz vorgestellten Transformationen werden als *primitiv* bezeichnet, weil sie Basisoperationen sind. Komplexere können durch *Konkatenation* solcher Transformationen als Makros konstruiert werden, hierzu siehe wieder [JZ98].

Die Äquivalenztransformationen stellen durch ihre Reversibilität sicher, daß keine Informationen verloren gehen. Im Gegensatz dazu wird die Informationskapazität durch die anderen Transformationen gesteigert oder gemindert. Es wurden hier nicht alle primitiven Restrukturierungstransformationen vorgestellt. Die restlichen Transformationen verhalten sich ähnlich wie die beschriebenen. Ein Satz von Transformationen wird in [Rum98] untersucht, vorgestellt und implementiert. Dieser Satz wird für diese Arbeit vorausgesetzt.

Eine weitere positive Eigenschaft, die Graphersetzungsregeln für die Tripelregeln und die Transformationen einzusetzen, ist erneut die Möglichkeit der Konkatenation zwischen diesen beiden (siehe [JZ98]). Außerdem wird die Komplexität umgangen und die Hilfestellung für den Benutzer erhöht. Des Weiteren können die Transformationen mit anderen Tripelregelsätzen konkateniert werden.

Daraus ergibt sich das Problem, daß die *Konsistenz* zwischen den Schemata nicht mehr gewährleistet ist. Die Tripelregeln sind konsistenzerhaltend, doch die Restrukturierungstransformationen sind dies, so wie sie hier beschrieben wurden, nicht. Dieses Problem spitzt sich zu, wenn wie im Beispiel-Szenario der Rückschritt zur Analyse nötig ist und die bereits vorgenommenen Veränderungen erhalten bleiben sollen. Genau diese beiden Punkte sind Gegenstand des nächsten Kapitels.

Kapitel 5

Konsistenzerhaltung auf der Basis von Nachtransformationen

Um die Konsistenz zwischen den Schemata zu erhalten, müssen die Änderungen des relationalen Schemas in das objektorientierte Schema übertragen werden. Der Konsistenzbegriff wird in dieser Arbeit folgendermaßen verstanden:

Konsistenz:
Das objektorientierte Schema kann mittels der initialen Abbildung und den manuellen Restrukturierungstransformationen aus dem relationalen Schema erzeugt werden.

Eine Konsistenz in Bezug auf die Informationskapazität wird in dieser Arbeit unter dem Begriff Äquivalenz verstanden. Ist Äquivalenz erwünscht, so muß das relationale Schema angepaßt werden, wenn im objektorientierten Schema Informationskapazitäts-verändernde Restrukturierungen vorgenommen wurden. Auch dies soll von VARLET unterstützt werden.

Äquivalenz:
Die Informationskapazität der beiden Schemata stimmt überein.

Um diesen beiden Ansprüchen gerecht zu werden, werden einerseits das Abbildungsschema und die Speicherung der Restrukturierungstransformationen mit ihren Parametern gebraucht. Andererseits wird eine Verbindung (Verkettung) zwischen diesen beiden Strukturen benötigt. Die Äquivalenz schließt die Konsistenz ein, aber oftmals wird auch nur die Konsistenz gefordert. In diesen Fällen soll das ursprüngliche relationale Schema unverändert bleiben.

5.1 Erhaltung der Äquivalenz

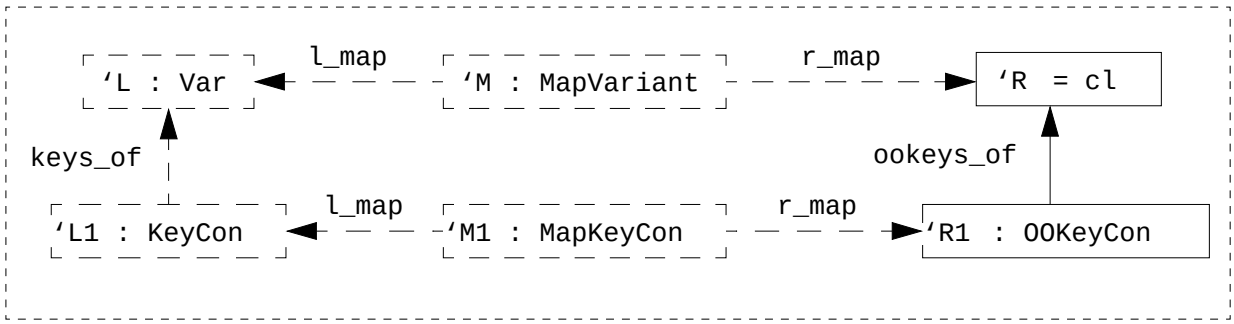
Die Erhaltung der Äquivalenz muß vom Benutzer angestoßen werden, der Grund liegt in den zwei gewünschten Funktionalitäten von VARLET. Einerseits sollen die beiden Schemata äquivalent zueinander sein. Jede relationale Schemakomponente ist mit einem korrespondierenden objektorientierten Konstrukt, mittels des Abbildungsschemas, verbunden. Andererseits soll nur die Konsistenz gewahrt werden.

Die Äquivalenz der Schemata kann durch die Erweiterung der initialen Abbildung nicht gewährleistet werden. Wenn z.B. eine neue Klasse erzeugt wird, gibt es keine relationale Schemakomponente, der die neue Klasse zugeordnet werden kann. Dies ist aber der Fall bei der Aufspaltung einer Klasse, die neu entstehende Klasse wird der ursprünglichen Relation zugeordnet. Im Beispiel-Szenario (siehe Abb. 3.5) sind *`Person`* und `Adresse` der Relation `Person` zugeordnet. Die *Rückwärtsabbildung* erhält die Äquivalenz.

Rückwärtsabbildung

Wenn neue Konstrukte im objektorientierten Schema erzeugt werden, müssen die entsprechenden relationalen Schemakomponenten hinzufügt werden. Dafür werden wieder die Tripelgraphgrammatiken eingesetzt. Die Rückwärtsabbildung benutzt die `backward rules` des Tripelregelsatzes. Dabei gestaltet sich das Abbilden von Klassen wieder am schwierigsten. In einer Vererbungshierarchie muß entschieden werden, ob eine Klasse auf eine neue Relation oder auf eine neue Variante abgebildet werden soll.

```
backward rule Map_Class_toVariant( cl : Class ; out var : Var ;
                    out keycon : KeyCon ; out ookeycon : OOKeyCon ;
                    out mapvar : MapVariant ; out mapkeycon : MapKeyCon) =
```




```
    return var := 'L;
           mapvar := 'M;
           keycon := 'L1;
           mapkeycon := 'M1;
           ookeycon := 'R1;
end;
```

Abb. 5.1: Die Tripelregel `Map_Class_toVariant`

Die Abb. 5.1 zeigt die Abbildung einer Klasse auf eine Variante mit den entsprechenden Schlüsselkonstrukten. Die Variante wird der Relation zugeordnet, die der Vererbungshierarchie der Klasse `'R` entspricht.

Ein anderer Aspekt ist die Behandlung von Klasseneigenschaften, die auf der objektorientierten Seite verschoben werden (z.B. `MoveClassPropsInHierarchy`). Folglich müssen die korrespondierenden relationalen Attribute und Fremdschlüssel in den Relationen oder Varianten ebenfalls verschoben (bzw. hinzugefügt und gelöscht) werden.

5.2 Auswirkungen von Restrukturierungstransformationen auf das Abbildungsschema

Nicht alle Restrukturierungstransformationen bewirken Änderungen im objektorientierten Schema, die in das relationale übertragen werden müssen. Nachdem die Tripelgraphgrammatiken angewandt wurden (siehe Abschnitt 4.4), kann die initiale Abbildung auf verschiedene Weisen erweitert werden.

Zusätzliche `r_map`-Kanten werden von einem `MapList`-Knoten (Oberklasse aller Abbildungsknoten) zu neuen Objekten gezogen, wie bei `SplitClass` in Abb. 5.2 (die Kanten des objektorientierten Schemas wurden aus Gründen der Übersichtlichkeit weggelassen).

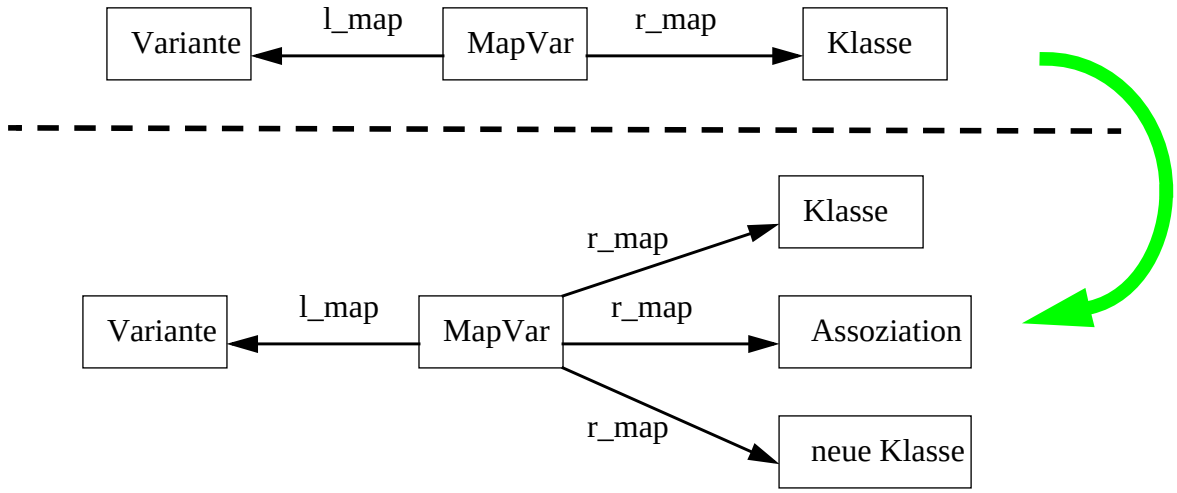


Abb. 5.2: Auffächern der Abbildung nach der Anwendung von `SplitClass`

Andererseits werden die `r_map`-Kanten von nicht mehr „vorhandenen“ Objekten auf die übrig gebliebenen gezogen, wie zum Beispiel bei `MergeClass` in Abb. 5.3 (die Kanten des objektorientierten Schemas wurden wieder weggelassen).

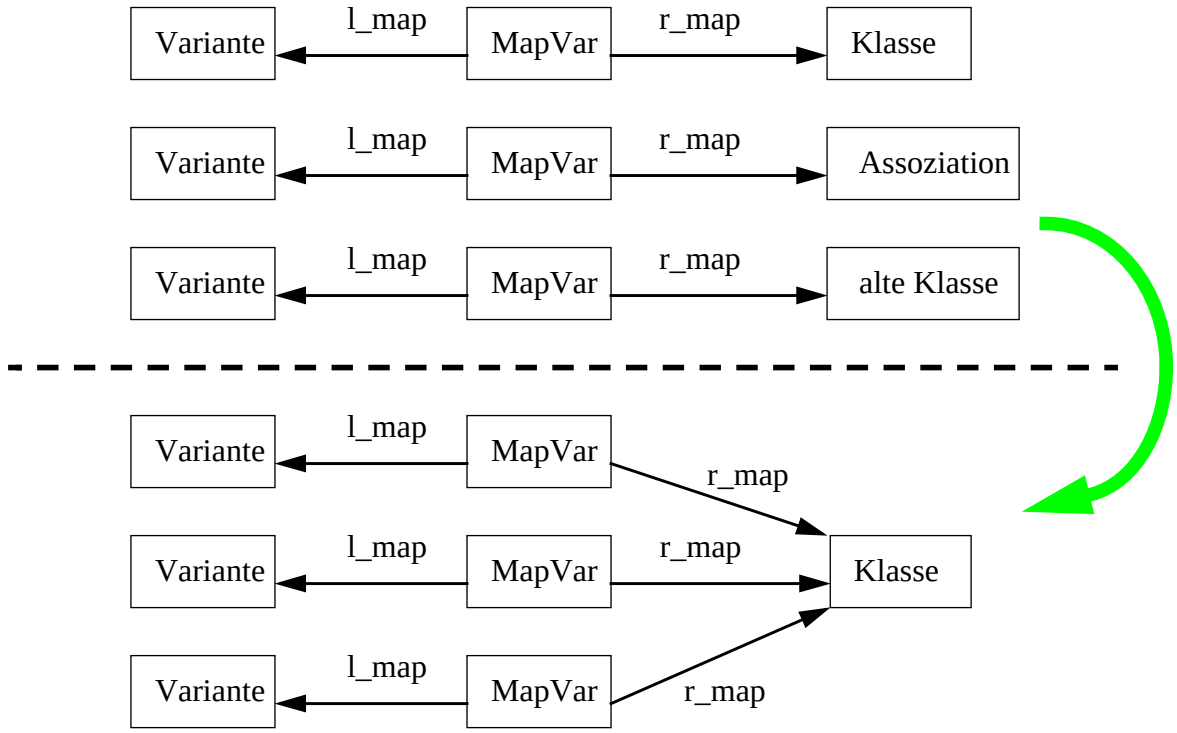


Abb. 5.3: Konzentration der Abbildung nach der Anwendung von `MergeClass`

Bei den anderen Informationskapazitäts-erhaltenden Transformationen verhält sich das Erweitern der Abbildung weitgehend ähnlich wie bei `SplitClass` und `MergeClass`, für `RelationshipToClass` und `ClassToRelationship` sogar genauso.
Bei `Aggregate` und `DisAggregate` wird nur eine neue `r_map`-Kante von dem Abbildungsknoten zur Aggregation bzw. Assoziation gezogen.

Bei den kardinalitätsverändernden Transformationen braucht die Abbildung gar nicht erweitert werden, genauso wenig ist es für die zwei Transformationen `Rename` und `ChangeType` nötig.

Bei vielen Informationskapazitäts-erweiternden Transformationen kann eine Erweiterung der Abbildung nicht durchgeführt werden. Dafür muß die Rückwärtsabbildung angestoßen werden. Danach hat jede objektorientierte Schemakomponente eine Verbindung zu einer relationalen Schemakomponente, die Äquivalenz der Schemata ist wiederhergestellt.

5.3 Erhaltung der Konsistenz

Die Konsistenzerhaltung ist gewährleistet, solange sich im relationalen Schema nichts ändert. Ist dies der Fall, kommt die *Nachtransformation* zum Einsatz. Alle Transformationen, die von veränderten relationalen Schemakomponenten abhängen, müssen während der Konsistenzerhaltung neu ausprobiert werden.
Datenbank Schemata sind häufig groß. Die Änderungen betreffen jedoch nur einen kleinen Teil der relationalen Schemakomponenten, daher ist eine *inkrementelle Konsistenzerhaltung* sinnvoll. Mit *inkrementell* ist gemeint, daß nur die Abbildungskonstrukte, Transformationen, relationalen Schemakomponenten und objektorientierten Schemakomponenten neu untersucht bzw. ausgeführt werden müssen, auf die sich die Änderungen des relationalen Schemas auswirken.

5.3.1 Verwaltung der Abhängigkeiten von Restrukturierungstransformationen

Um die Abhängigkeit aller Konstrukte und Transformationen festzustellen, müssen die initiale Abbildung und die Transformationen mit ihren Parametern gespeichert und verbunden werden. Dafür wird ein *Abhängigkeitsgraph* aufgebaut.

Transformationen, die aufeinander aufbauen, sind im Beispiel-Szenario das Abspalten und das Aggregieren von der Klasse `Adresse` in Abb. 3.5. Die Generalisierung von `Abschluss-spruefung` und `Schein`, die Spezialisierung der Assoziation zwischen `Pruefung` und `Student` und die Kardinalitätsänderung hängen ebenfalls voneinander ab.

Als nächstes wird die interne Verkettung der initialen Abbildung mit den darauf aufbauenden Restrukturierungstransformationen dargelegt.

Die jeweils ersten drei Knoten der Abb. 5.4, von der linken Seite aus gesehen (die RMO-Tupel), stammen von der initialen Abbildung. Alle weiteren Knoten und die objektorientierten Schemakomponenten, die aus der initialen Abbildung resultieren, gehören dann zur internen

Speicherung der Transformationen.

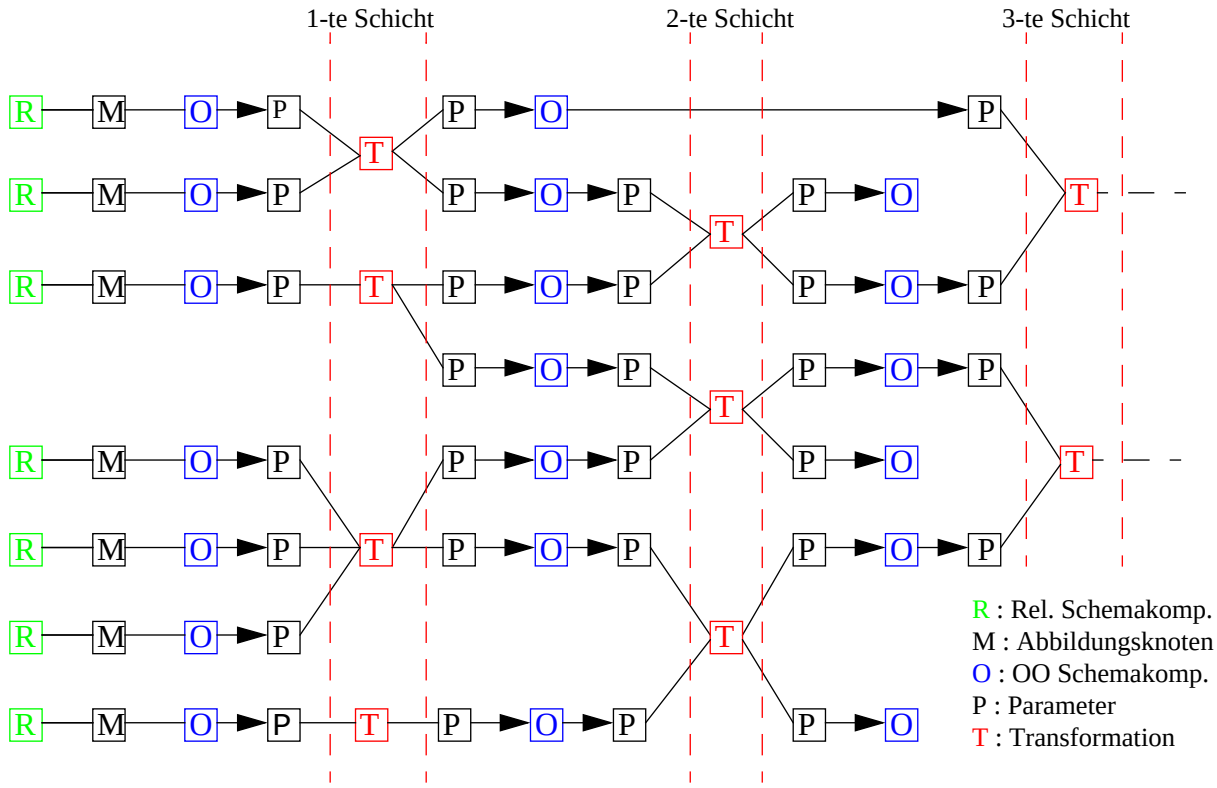


Abb. 5.4: Der Abhängigkeitsgraph

Transformationen haben Eingabeparameter, die aus der initialen Abbildung oder aus anderen Tranformationen stammen. Zur Beschreibung der Tiefensuche im Abhängigkeitsgraph während der Nachtransformation wird ein Begriff von Transformations-Schichten gebraucht.

Transformations-Schichten:
Die erste Schicht bilden die Transformationen, für die alle Eingabeparameter aus objektorientierten Schemakomponenten bestehen, die von der initialen Abbildung erzeugt werden. In der n-ten Schicht sind alle Transformationen die mindestens einen Eingabeparameter haben, der Ausgabeparameter einer Transformation der (n-1)-ten Schicht ist.
Für die n-te Schicht ist die (n-1)-te Schicht die *darüberliegende* und die (n+1)-te Schicht die *darunterliegende*.

Eine kurze Beschreibung der Nachtransformation folgt, um den Gebrauch des Abhängigkeitsgraphen zu erläutern.

Im Fall einer Veränderung des relationalen Schemas werden die veränderten Schemakomponenten markiert. Alle davon abgebildeten objektorientierten Schemakomponenten und die Transformationen, die diese objektorientierten Schemakomponenten als Eingabeparameter haben, werden ebenfalls gekennzeichnet. Alle nachfolgenden Transformationen sind von den Änderungen ebenfalls betroffen und werden dementsprechend markiert. Danach sind im Abhängigkeitsgraphen alle Knoten gekennzeichnet, die während der Nachtransformation neu untersucht werden müssen.

Als erstes muß die initiale Abbildung für die veränderten relationalen Schemakomponenten wiederholt werden. Daraus werden neue objektorientierte Schemakomponenten erzeugt, mit denen die zu wiederholenden Transformationen ausprobiert werden können. Für alle Eingabe- und Ausgabeschemakomponenten werden *Parameter-Knoten* angelegt, um die Reihenfolge zu erhalten. Mit Hilfe dieser Parameter-Knoten wird gewährleistet, daß die neuen objektorientierten Konstrukte, die durch die erneute initiale Abbildung oder eine Nachtransformation verändert wurden, an dieselben Transformationen, an den selben Stelle übergeben werden. Im Abhängigkeitsgraphen werden die „alten“ Eingabeparameter durch die „aktuellen“ ersetzt.

Außerdem müssen die resultierenden Konstrukte als Eingabeparameter einer Tranformation richtig zugeordnet werden, wenn eine vorangegangene Tranformation überflüssig wird. Eine Tranformation wird überflüssig, wenn durch die Rückwärtsabbildung die Restrukturierungen in das relationale Schema übernommen worden sind. Die initiale Abbildung erzeugt dann schon den Zustand, der erst nach der überflüssig gewordenen Tranformation entstand.

Ausgehend von den Transformationen der ersten Schicht, werden die nachfolgenden Transformationen ausprobiert. Der Abhängigkeitsgraph wird benutzt, um zu verhindern, daß eine Transformation T der i-ten Schicht ausprobiert wird, deren vorangegangene Transformationen noch nicht wiederholt wurden. Das heißt: alle Transformationen der ersten bis zur (i-1)-ten Schicht, deren Ausgabeparameter als Eingabe für die Transformation T dienen, müssen erfolgreich wiederholt sein.

Zuletzt werden alle noch markierten Knoten im Abhängigkeitsgraph gelöscht. Transformationen, die nicht erfolgreich wiederholt wurden, und ihre Parameter sind noch markiert. Zudem wird ein Logbuch von diesen Transformationen erstellt, damit der Benutzer nachvollziehen kann, welche Restrukturierungen nicht wiederholt wurden.

Um dies zu verdeutlichen, wird im nächsten Abschnitt auf den internen Aufbau der Transformationen eingegangen.

5.3.2 Aufbau der Restrukturierungstransformationen

Es ist wichtig, alle Transformationen mit ihren Parametern zu speichern, um sie während der *Nachtransformation*, falls möglich, wieder auszuführen. Die objektorientierten Eingabeparameter werden als *Klone* gespeichert.

Ein *Klon*, wie er hier verstanden wird, ist nicht eine vollständige Kopie der objektorientierten Schemakomponente. Vor allem ist der Klon nicht mehr in die Abbildungs-, Listen-, Repräsentations- und logischen Strukturen eingebunden. Der Klon bekommt nur Informationen durch Attributbelegungen des Knotens, die für die Nachtransformation benutzt werden. Dies hat den Grund, redundante Speicherung zu vermeiden, und dadurch den Graphen nicht unnötig zu verkomplizieren und möglichst klein zu halten.

Transformationen können nur ausgeführt werden, wenn bestimmte Vorbedingungen erfüllt sind. Eine Klasse darf beispielsweise nur in eine Assoziation umgewandelt werden (`ClassToAssoc`), wenn sie genau zwei Beziehungen hat. Diese zwei Assoziationen müssen für die umzuwandelnde Klasse total und Eins ([1-1], siehe `StudVorl` Abb. 3.5) sein.

Es gibt zwei Arten von Vorbedingungen:

- **Dynamische Vorbedingungen:**
  Eine Transformation, im einem zu bearbeitenden Graphen, hängt von der Existenz ihres linken Teilgraphen ab. Das heißt: der zu ersetzende Teilgraph (linke Seite) muß in dem aufgebauten Graphen gefunden werden, damit die Transformation angewendet werden kann.
  Die dynamischen Vorbedingungen betreffen die Knoten der linken Seite und deren *Eins-Kontext*. Mit dem Eins-Kontext ist das logische Umfeld von Schemakomponenten gemeint. Zum logischen Umfeld einer Klasse gehören alle Klasseneigenschaften (Attribute, Traversierungspfade) und Schlüssel.
- **Statische Vorbedingungen:**
  Statische Vorbedingungen betreffen eine Menge von Schemakomponenten. Diese müssen keinen direkten Zusammenhang haben. Sie können einmal für alle davon betroffenen Transformationen formuliert werden. Deshalb werden sie als *Invarianten* implementiert, wie z.B. zwei Klassen dürfen nicht den gleichen Namen haben.

Eine Transformation wird mit ihren objektorientierten Parametern und entsprechenden Parameter-Knoten aufgebaut. Abb. 5.5 ist wieder in OMT-Notation.

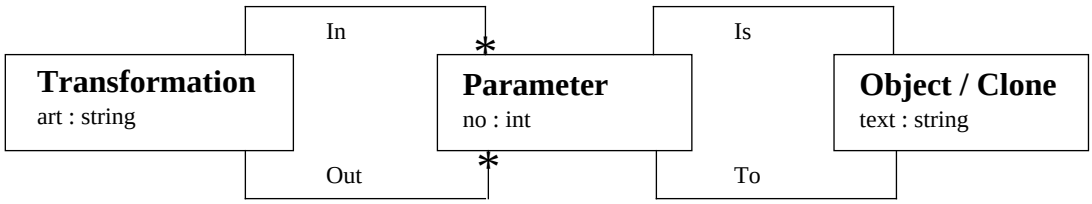


Abb. 5.5: Die Transformation

Bei der Ausführung wird die Transformation wie in Abb. 5.6 gespeichert. Es sei wieder das Beispiel von `SplitClass` gewählt. Auf die Parameter-Knoten wird während der Ablauf-Beschreibung nochmals eingegangen. Die Abbildung ist an die PROGRES-Notation angelehnt.

Für die Transformation wird ein Knoten angelegt, der den Namen der Transformation speichert. Der Knoten `n_Name` ist der bereits im Abschnitt 4.2 gesehene „STRING“-Knoten. Dieser Knoten wird für die Nachtransformation gebraucht. Wenn `SplitClass` wiederholt werden soll, muß der Name der neu zu erzeugenden Klasse übergeben werden. Die doppelten Kästen bei den `cps`- und `param2`-Knoten bedeuten, daß sie eine Menge von Klasseneigenschaften bzw. Parametern enthalten.

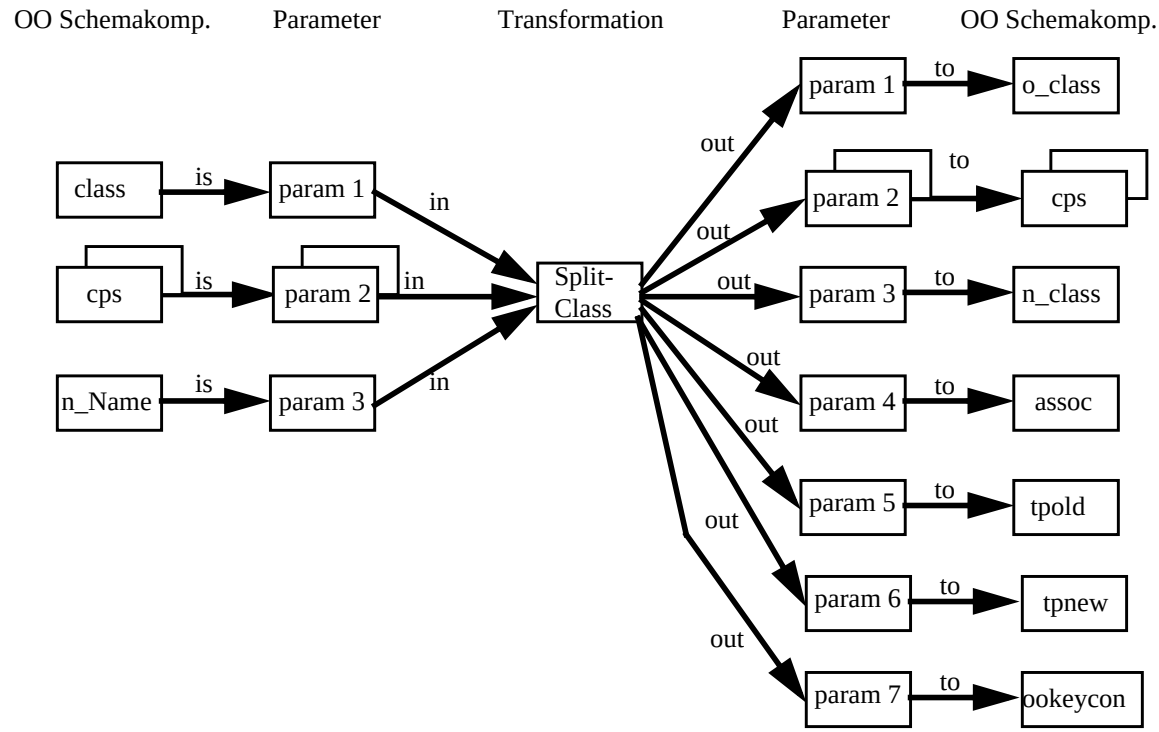


Abb. 5.6: Aufbau einer Transformation

In einer Restrukturierungstransformation darf keine *Iteration* von Pfaden (z.B. `(-is-> & -in-> & -out-> & -to->)*`) auftreten. Die Iteration eines Pfades mit dem *-Operator liefert alle Knoten, die erreicht werden, wenn der Pfad kein oder mehrmals durchlaufen wird. Ausgehend von `class` werden mit `(-is-> & -in-> & -out-> & -to->)*`, `class` selbst, alle Ausgabeknoten von `SplitClass` und alle Ausgabeknoten aller nachfolgenden Transformationen erreicht.

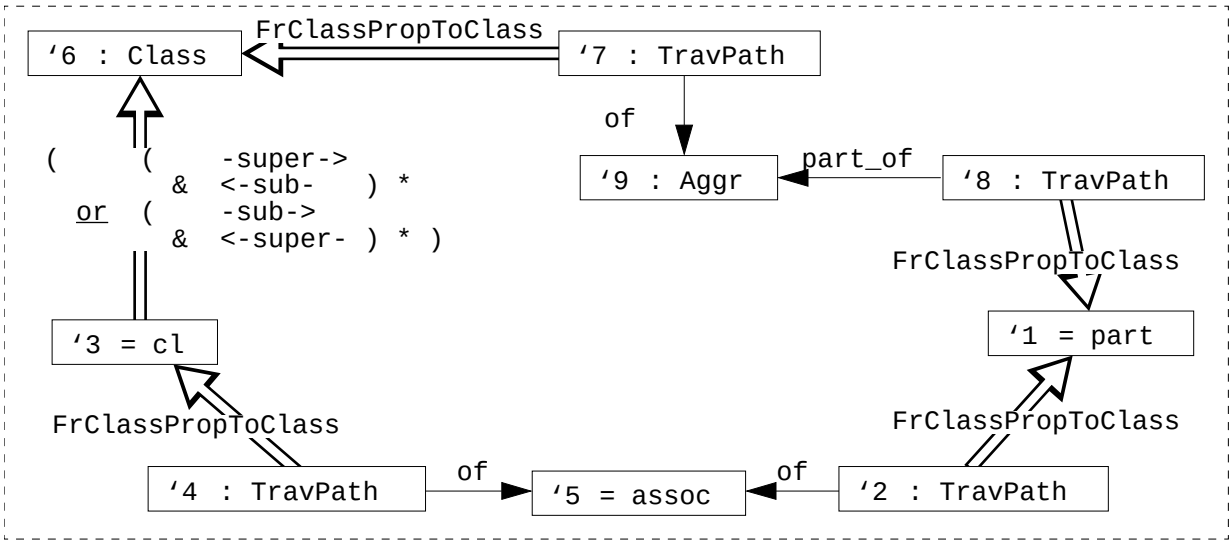
Wäre die Iteration von Pfaden in Restrukturierungstransformationen erlaubt, ließen sich die dynamischen Vorbedingungen nicht mehr ausschließlich im Eins-Kontext formulieren. Außerdem würde der transitive Abschluß während der Nachtransformation die Überprüfung eines großen Teilgraphen bewirken. Viele Transformationen würden markiert werden, weil über einen rekursiven Pfad ihre Eingabeparameter erreicht werden können. Im schlechtesten Fall wäre der ganze Abhängigkeitsgraph markiert. Der Mechanismus für die Inkrementalität wäre dann überflüssig. Die Nachtransformation müßte alle Transformationen erneut ausführen.

Doch das ist keine (wirkliche) Einschränkung. Transformationen, die eine Iteration eines Pfades brauchen, können unterteilt und in einem Makro zusammengefaßt werden. Bei `MoveClassPropsInHierarchy` beispielsweise wird das Verschieben von Klasseneigenschaften nur über eine Vererbungsebene erlaubt. Ein Makro könnte diese Transformation mehrmals aufrufen, um das Verschieben über mehrere Vererbungsebenen zu ermöglichen. Vorbedingungen, für die eine Iteration von Pfaden benötigt würde, können einfach in Invarianten umformuliert werden.

Abb. 5.7 zeigt die Iteration eines Pfades in einer Vorbedingung. Die Klasse `part` soll an die Klasse `cl` aggregiert werden. Vorher muß überprüft werden, ob die Klasse nicht schon an eine Klasse in der Vererbungshierarchie von `cl` aggregiert worden ist. Es macht keinen Sinn, eine

Klasse an zwei Klassen in einer Vererbungshierarchie zu aggregieren. Die Pfade `-super-> & <-sub-` und `-sub-> & <-super-` werden iteriert, um alle Klassen in der Vererbungshierarchie von `cl` zu erreichen. All diese Klassen müßten bei der Nachtransformation berücksichtigt werden. Abb 5.7 zeigt einen `test`, der - mit `not` aufgerufen - diese Vorbedingung ergibt.

```
test ExistanceOfAnAggragtionFromPartToAClassInHierarchy( part : Class ;
                                                  assoc : Assoc ; cl : Class)=
```

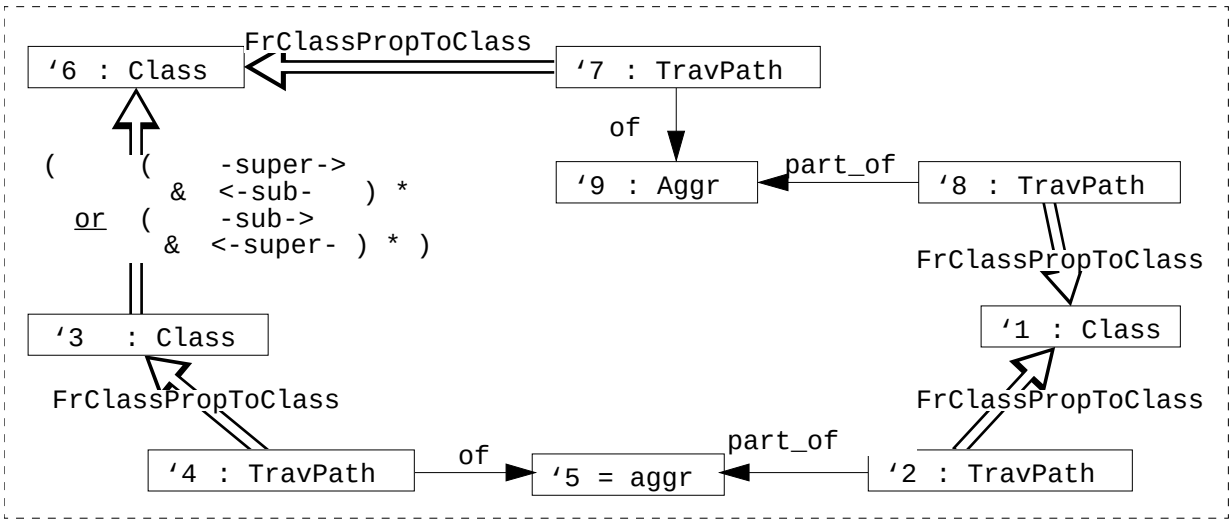



```
    condition ‘2.many = false;
end;
```

Abb. 5.7: Vorbedingung für Aggregate mit der Iteration von Pfaden

Abb. 5.8 zeigt die Invariante, die aus der Vorbedingung entstanden ist. Der `test` muß wieder mit `not` aufgerufen werden, um die Invariante zu erhalten.

```
test ExistanceOfAnAggragtionFromTwoClassesInAnHierarchyToTheSameClass(
                                                          aggr : Aggr) =
```




```
end;
```

Abb. 5.8: Die Invariante, die aus der Vorbedingung umformuliert wurde

Die Invariante unterscheidet sich von der Vorbedingung dadurch, daß sie die Bedingung für alle Aggregationen überprüfen kann. Die Klassen `cl` und `part` werden nicht übergeben. Bei der Nachtransformation werden alle Invariante am Ende überprüft. Ist eine Invariante nicht erfüllt, bekommt der Benutzer eine Fehlermeldung (siehe Abschnitt 5.4).

5.3.3 Aufbau des Abhängigkeitsgraphen

Für den Aufbau des Abhängigkeitsgraphen ist der Ablauf einer Restrukturierungstransformation wichtig. Solange keine Nachtransformation stattfindet, wird die im Abschnitt 5.3.1 beschriebene „Verkettung“ von Schemakomponenten, Parameter-Knoten und Transformations-Knoten aufgebaut.

Ablauf einer Transformation:

a) die linke Seite und eventuelle Vorbedingungen werden getestet
(Ist die Transformation anwendbar?)

b) die Eingabeobjekte werden zurückgeliefert und als Klone gespeichert

c) die Klone werden als Eingabeparameter einsortiert

d) die Transformation wird ausgeführt

e) die Ausgabeobjekte werden zurückgeliefert und als Parameter aufgenommen

f) die initiale Abbildung wird bei Bedarf erweitert

g) die Invarianten werden getestet

Die Schritte im einzelnen:

a) Die linke Regelseite und eventuelle Vorbedingungen werden getestet

Als erstes wird getestet, ob alle Knoten der linken Regelseite (siehe `SplitClass` in Abschnitt 4.5) vorhanden sind. Eventuell sind Knoten zu identifizieren, die von dem Benutzer nicht übergeben werden. Der Kontext, in dem sich die Transformation abspielt, muß überprüft werden. Dies ist wichtig, da alle Eingabeparameter der Transformation benötigt werden. Es reicht z.B., wenn der Benutzer bei `MergeClass` die Assoziation zwischen den zwei Klassen übergibt. Für die Ausführung werden die zwei Klassen und die Traversierungspfade benötigt. Wenn ein `string` übergeben wird, muß dafür zuerst ein STRING-Knoten angelegt werden. Zusätzlich wird überprüft, ob die Vorbedingungen erfüllt sind. Eine Klasse darf man nur aufspalten (`SplitClass`), wenn die angegebene Attributmenge entweder den ganzen Schlüssel oder kein Attribut des Schlüssels enthält.

b) Die Eingabe-Objekte werden zurückgeliefert und als Klone gespeichert

Sind alle Vorbedingungen für die Transformation erfüllt und der Test hat alle benötigten Eingabeparameter zurrückgeliefert, werden diese als Klone gespeichert. Wenn eine Vorbedingung nicht erfüllt ist, wird die Transformation nicht weiter ausgeführt und eine entsprechende Fehlermeldung zurückgeliefert. Dasselbe gilt, wenn der Test nicht alle für die Ausführung der Transformation erforderlichen Eingabeparameter identifizieren kann.

c) Die Klone werden als Eingabe-Parameter einsortiert

Anschließend werden für alle Klone Parameter-Knoten erzeugt, die durchnumeriert werden. Diese Numerierung dient nicht zur Erstellung einer Reihenfolge, sondern sie gewährleistet, daß die Parameter bei der Nachtransformation immer eindeutig zugeordnet werden können. Für das obige Beispiel von `SplitClass` (Abb. 5.6) bekommt der Parameter-Knoten der Klasse `class` die Nummer 1. Die Parameter-Knoten, die für die Klasseneigenschaften `cps` erzeugt werden, bekommen alle die gleiche Nummer, hier die `2`. In diesem Schritt wird auch der Knoten für die Transformation erzeugt und die entsprechenden Kanten `is` und `in` werden gezogen, die für den Abhängigkeitsgraphen erforderlich sind.

d) Die Transformation wird ausgeführt

Nun kann die Transformation selbst ausgeführt werden. Die objektorientierten Schemakomponenten, deren Klone als Eingabeparameter dienen, werden von der Graphersetzungsregel transformiert. Da alle Vorbedingungen erfüllt und alle Eingabeparameter vorhanden sind, ist der zu ersetzende Teilgraph eindeutig identifiziert und die Transformation wird erfolgreich abgeschlossen.

e) Die Ausgabe-Objekte werden zurückgeliefert und als Parameter aufgenommen

Alle Knoten der linken Regelseite werden zurückgeliefert, ob sie verändert wurden oder nicht, und als Ausgabeparameter eingeordnet. Dies ist wichtig für eventuell nachfolgende Transformationen. Ausnahmen sind Knoten vom Typ STRING, da sie für weitere Transformationen nicht gebraucht werden. Auch hier werden die Parameter für die Nachtransformation durchnumeriert. Die Ausgabeparameter müssen die „Eingabeklone“ einer nachfolgenden Transformation ersetzen. Dies geschieht analog zu den Eingabeparametern.

f) Die initiale Abbildung wird bei Bedarf erweitert

In manchen Fällen muß nun die initiale Abbildung, wie im Abschnitt 5.2 beschrieben, erweitert werden.

g) Die Invarianten werden getestet

Am Ende einer Transformation werden die Invarianten getestet, für die eine Änderung in Frage kommt. Wenn diese nicht erfüllt sind, werden durch den backtrack-Mechanismus von PROGRES die Schritte a) bis f) rückgängig gemacht und eine Fehlermeldung ausgegeben.

5.3.4 Ablauf der Nachtransformation

Die Voraussetzung für die Nachtransformation ist eine Änderung des relationalen Schemas, sei es direkt durch den Benutzer oder nach einer erneuten Analyse-Phase.

Für die Konsistenzerhaltung ist die Nachtransformation ein wichtiger Schritt, deshalb wird zuerst ein Überblick des Ablaufs aufgezeigt. Anschließend wird mit Hilfe von Beispielen der Ablauf detailliert vorgestellt.

Der Ablauf einer Nachtransformation verläuft, wie folgt:

1) Markierung

- Markierung der von den Änderungen betroffenen Konstrukte im Abbildungsschema
- Vorwärtsmarkierung
- Rückwärtsmarkierung

markieren der abhängigen Transformationen als ungültig

2) Abbildung

- Initiale Abbildung mit Tripelgraphgrammatiken-Mechanismus
- Initiale Aktualisierung

3) Nachtransformation (Rekursiver Aufruf)

- Transformation ausführen, wenn möglich:
  - Test der Eingabeparameter und Vorbedingungen
  - Parameterspeicherung (mit der Erzeugung der Klone)
  - die Transformation wird ausgeführt
  - die Ausgabeparameter werden gespeichert

  ähnlich dem Ablauf einer manuellen Transformation
- Aktualisieren
- Zur nächsten Transformation übergehen

4) Löschen

- Löschen aller alten Knoten (Schemakomponenten, Parameter, Transformationen, ...)

5) Invarianten

- Alle Invarianten werden überprüft.

Nun werden die einzelnen Schritte der inkrementellen Konsistenzerhaltung vorgestellt:

1) Markierung

Alle veränderten relationalen Schemakomponenten sind markiert. Dies wird bei der jeweiligen Änderung durchgeführt.

Als erstes werden die Markierungen der relationalen Schemakomponenten auf das Abbildungsschema und den daraus stammenden objektorientierten Schemakomponenten/Klone erweitert, d.h. diese werden als ungültig gekennzeichnet.

Das Kreuz über der relationalen Schemakomponente in Abb. 5.9 zeigt an, daß sie verändert worden ist. Dementsprechend werden die davon abhängigen Knoten, der Abbildungsknoten (M) und der Klon (O), als ungültig markiert.

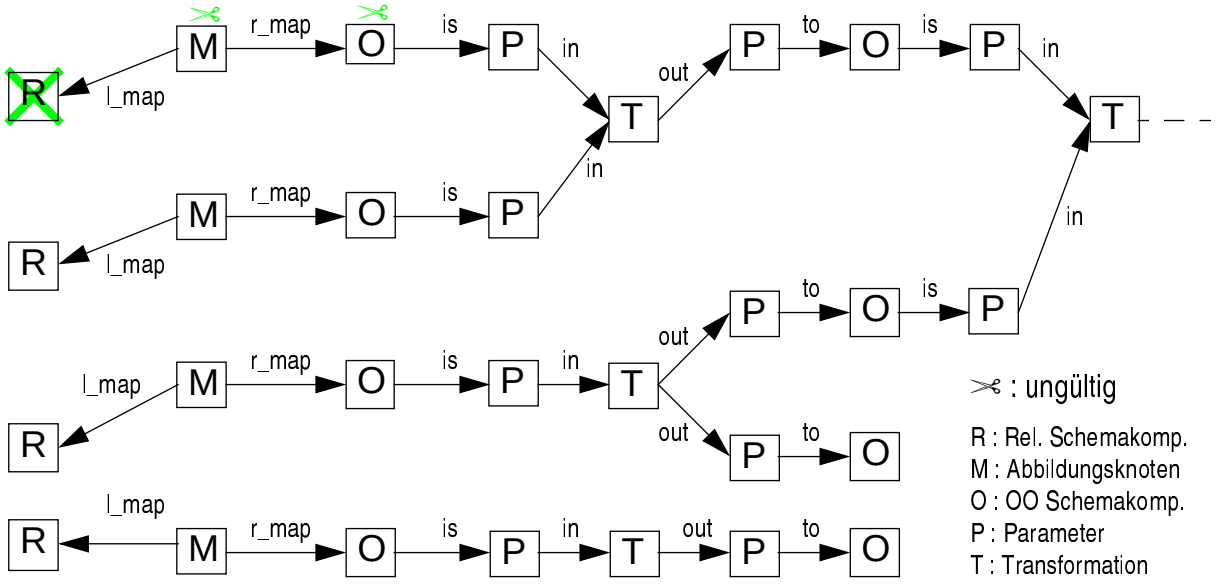


Abb. 5.9: Markierung des Abbildungsschemas

Im Beispiel-Szenario wäre die Relation `Abschlusspruefung` markiert. Zugunsten der Übersichtlichkeit sind in den folgenden Abbildungen nicht alle relationalen Schemakomponenten, Abbildungsknoten und objektorientierten Schemakomponenten aufgeführt. Für die Relation `Abschlusspruefung` müßte auch ihre einzige Variante markiert werden. Hier sind die Relation und ihre Variante zusammengefaßt, wie auch die Abbildungsknoten. Das gilt auch für die Transformationen, die oft mehr als ein oder zwei Ein- und Ausgabeparameter haben. Deshalb ist der Abhängigkeitsgraph in einer abstrakten Form dargestellt. Der Ablauf der Nachtransformation würde unübersichtlich werden, wenn alle betroffenen Knoten dargestellt wären. Ebenso wird auf das Erweitern des Abbildungsschemas verzichtet. Da zudem die Transformationen nicht explizit angegeben werden, wären die Erweiterungen nur fiktiv.

Anschließend werden alle *Transformationskonstrukte* (objektorientierte Schemakomponenten/Klone, Ein-, Ausgabeparameter-Knoten und Transformations-Knoten einer Transformation), die ausgehend von den „ungültigen“ Knoten der ersten Schicht erreicht werden können, als ungültig markiert. Da die Kanten in beiden Richtungen traversierbar sind, kann man den ganzen Graphen erreichen. Gemeint ist hier natürlich, daß den Kanten in der Richtung und Reihenfolge **-Is->, -In->, -Out->, -To->, -Is->, ...** gefolgt wird.

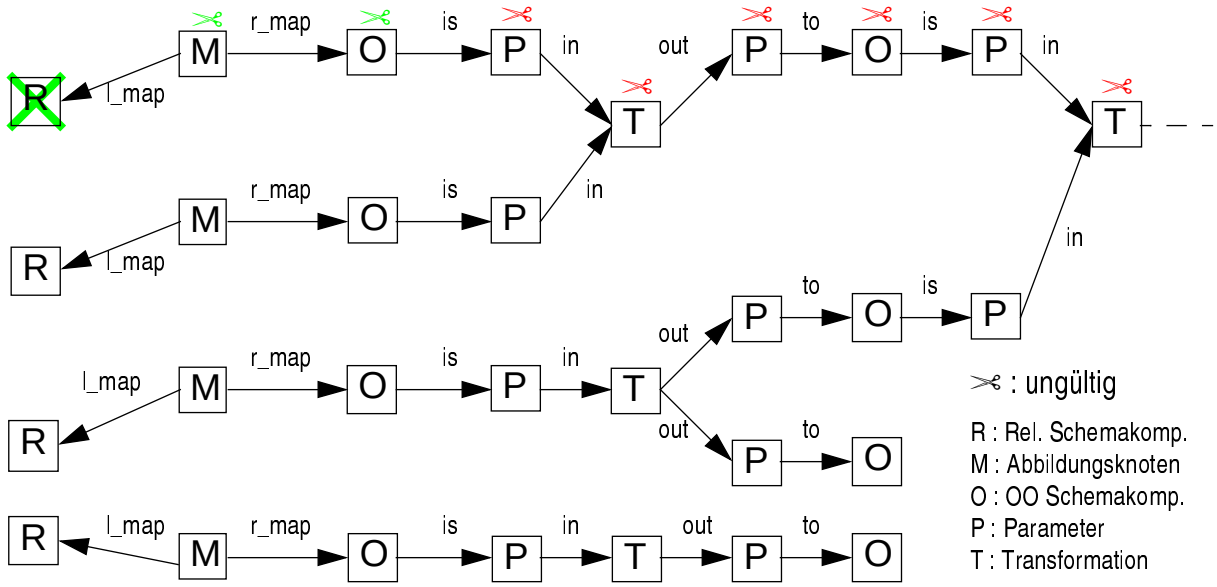


Abb. 5.10: Vorwärtsmarkierung

Als nächstes wird die sogenannte Rückwärtsmarkierung eingeleitet, ausgehend von „ungültigen“ Transformations-Knoten, die als *nicht ungültig* markierte Eingabeparameter-Knoten besitzen, werden rückwärts alle Transformationskonstrukte als ungültig markiert.

Die Rückwärtsmarkierung wird gebraucht, weil die Transformationen als Eingabeparameter „nur“ Klone besitzen. Deshalb müssen die objektorientierten Schemakomponenten von neuem aufgebaut werden. Zum Teil werden also relationale Konstrukte neu abgebildet, die in Wirklichkeit gar nicht verändert wurden.

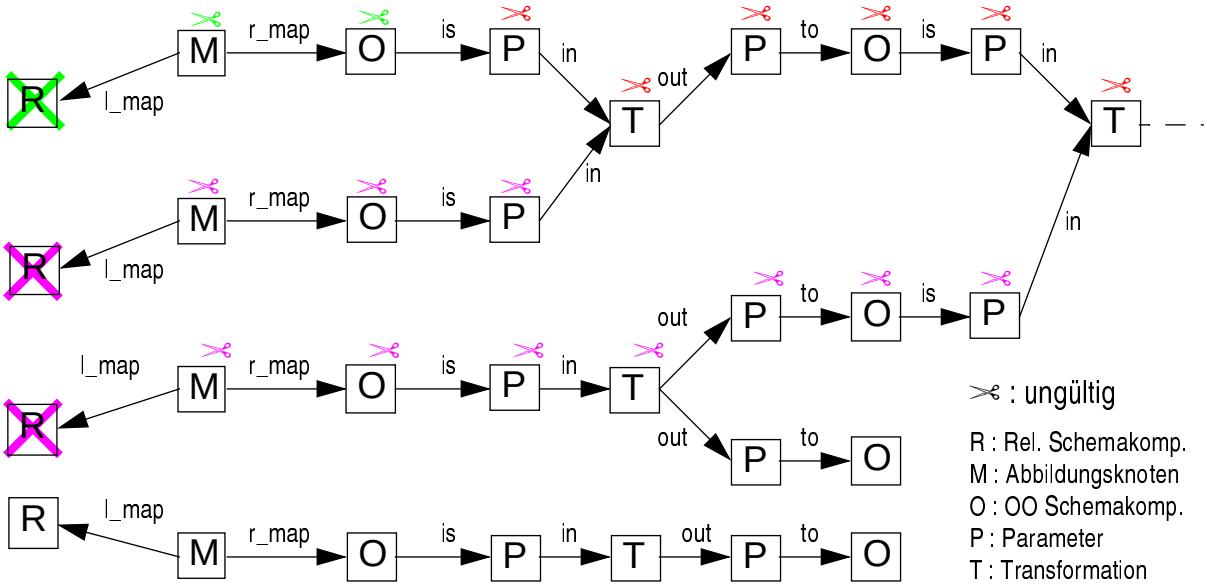


Abb. 5.11: Rückwärtsmarkierung

Die Kanten werden diesmal in der Richtung und Reihenfolge **<-In-, <-Is-, <-To-, <-Out-, <-In-, ...** traversiert. Am Abbildungsschema angekommen, werden die Abbildungsknoten als ungültig und die relationalen Schemakomponenten für die erneute initiale Abbildung als geändert markiert.

2) Abbildung

Danach kann die initiale Abbildung erneut ausgeführt werden.

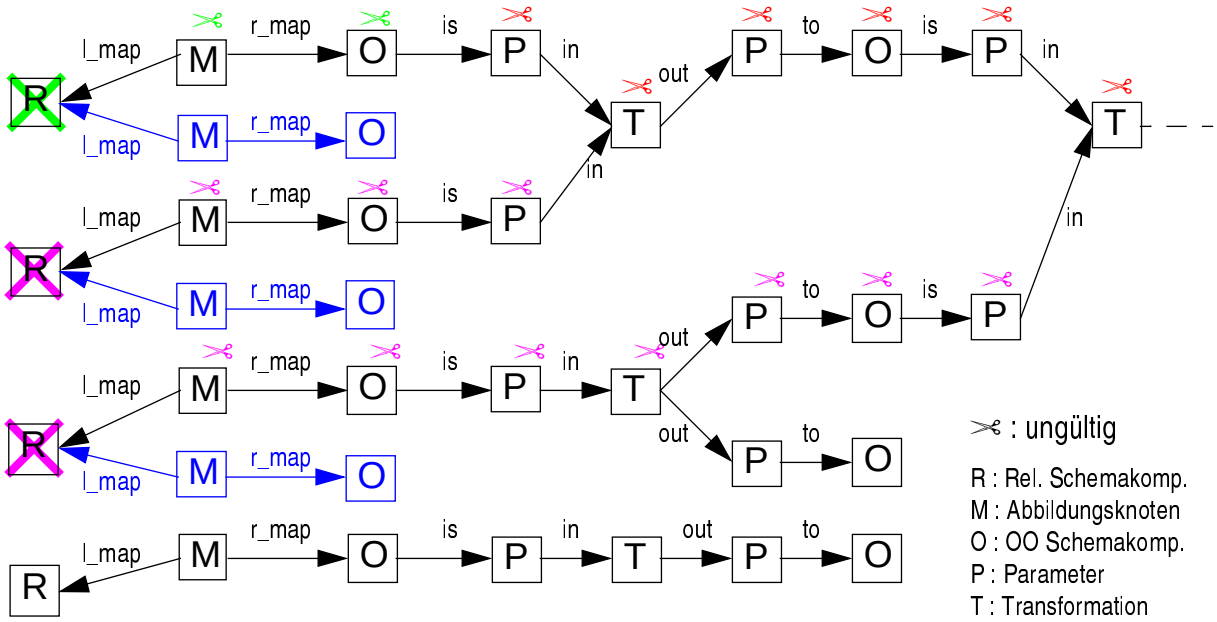


Abb. 5.12: Erneute initiale Abbildung

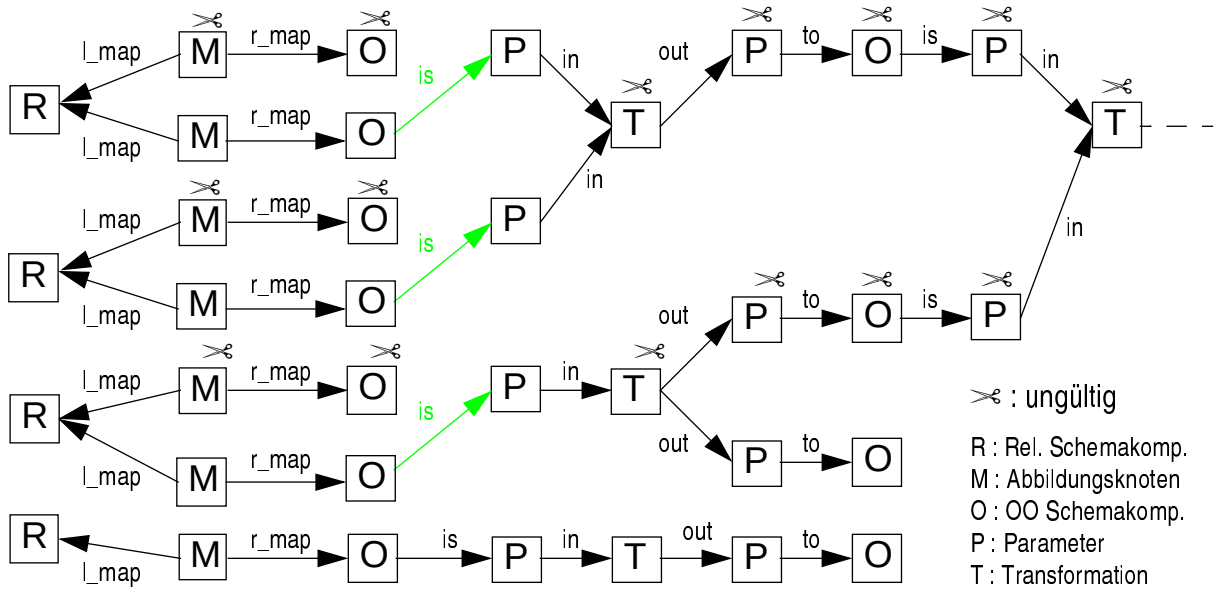


Abb. 5.13: Initiale Aktualisierung

Bei der Aktualisierung werden die objektorientierten Schemakomponenten anstelle der Klone ihren jeweiligen Parameter-Knoten zugeordnet. Die Eindeutigkeit wird über das Abbildungsschema hergestellt.

3) Nachtransformation

Die Nachtransformation ruft rekursiv die ungültig markierten Transformationen auf. Dabei wird versucht, die Transformationen mit der *Tiefensuche* abzuarbeiten. Das heißt: aufeinander aufbauende Transformationen werden nacheinander ausprobiert. Man beginnt mit einer Transformation der ersten Schicht.

Solange alle vorangegangenen Transformationen wieder erfolgreich ausgeführt worden sind, wird versucht eine der nachfolgenden Transformationen auszuführen. Ist dies nicht möglich, wird wiederum bei einer Transformation in der darüberliegenden Schicht begonnen. Ist dort keine *nicht* ausprobierte Transformation mehr vorhanden, wird ein anderer Zweig, ausgehend von einer Transformation der wiederum darüberliegenden Schicht, ausprobiert. Bricht der rekursive Aufruf ab, wird ein neuer Zweig, ausgehend von einer Transformation der ersten Schicht, ausprobiert.

Alle möglichen Transformationen, die erfolgreich wiederholt werden können, werden auch erreicht. Wenn eine Transformation auf mehreren Zweigen aufbaut, wird sie spätestens beim Durchlauf des letzten dieser Zweige erreicht. Die nicht erreichten Transformationen bauen, wegen der veränderten Eingabe, auf nicht erneut ausführbare Transformationen auf und können somit keine aktuellen objektorientierten Schemakomponenten als Eingabeparameter besitzen. Ein Versuch, sie erneut auszuführen, macht keinen Sinn.

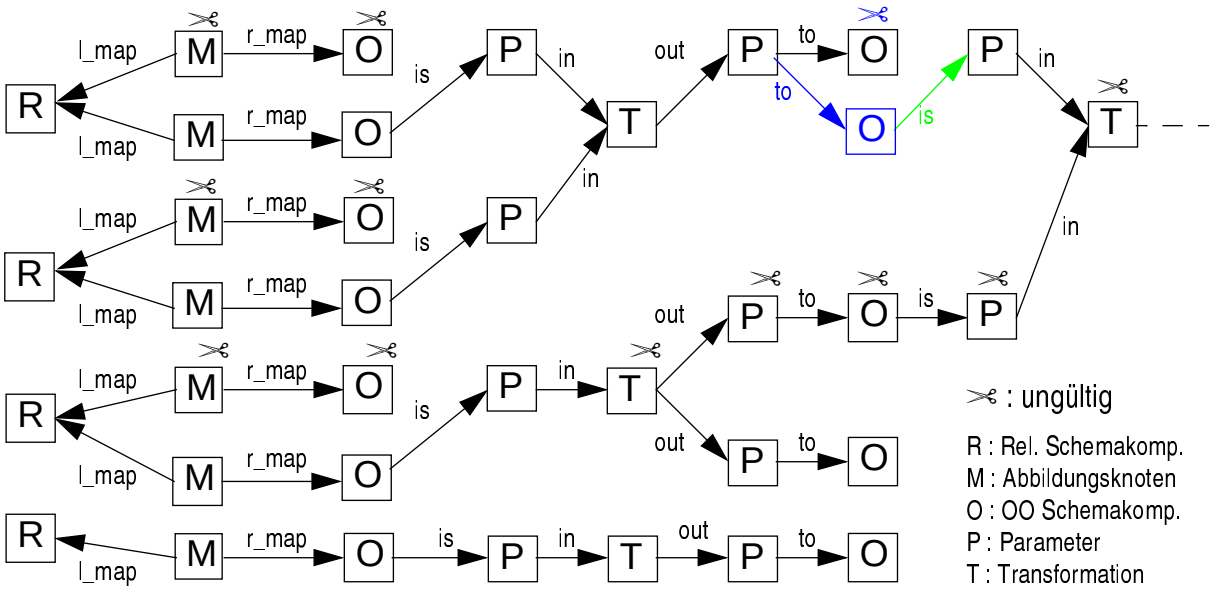


Abb. 5.14: Transformation ausführen

Die Transformation wird nur ausprobiert wenn, alle Parameter aktualisiert und die Vorbedingungen immer noch erfüllt sind. Danach werden die Eingabeparameter geklont und eingeordnet, die Transformation ausgeführt und die Ausgabeparameter einsortiert.

Der Unterschied zu einer erstmalig ausgeführten Transformation besteht darin, daß die Parameter-Knoten erhalten bleiben. Anschließend werden die Ausgabeparameter für eine eventuell nachfolgende Transformation aktualisiert. Auch hier werden die Parameter-Knoten beibehalten, was eine eindeutige Zuordnung gewährleistet.

Danach wird eine Transformation der darunterliegenden Schicht ausprobiert. In Abb. 5.14 würde jetzt versucht, die nachfolgende Transformation auszuführen. Da aber nicht alle Transformationen im Vorbereich als gültig markiert sind, wird wiederum eine aus der ersten Schicht ausgewählt.

Bei der ausgeführten Transformation in Abb. 5.15 wird der „obere“ Ausgabeparameter für die nachfolgende Transformation aktualisiert. Der „untere“ wird durch den neu erzeugten ersetzt, d.h. der alte Ausgabeparameter wird als ungültig markiert.

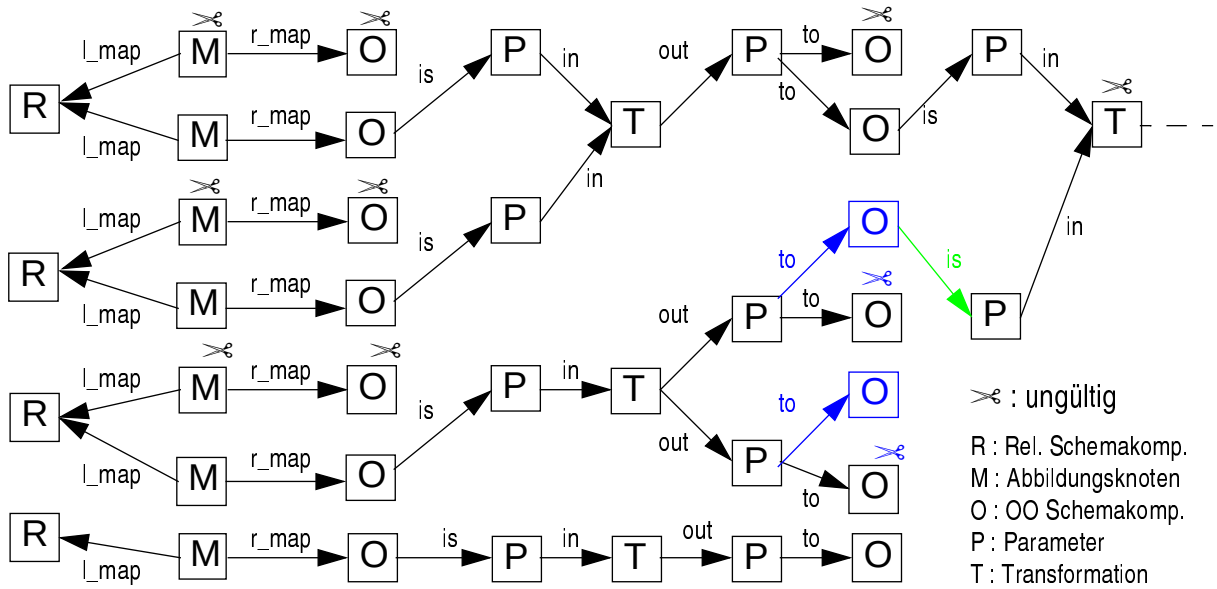


Abb. 5.15: Nächste Transformation ausführen

Danach wird wieder eine (hier gibt es nur eine) nachfolgende Transformation ausprobiert. Durch die Veränderung im relationalen Schema können sich die Voraussetzungen ändern. Hier sei angenommen, daß dadurch die Vorbedingungen nicht mehr erfüllt sind. In diesem Fall werden Eingabeparameter-Knoten als ungültig markiert, um später gelöscht zu werden. Außerdem wird eine Logbuchmeldung konstruiert, dies wird in Abschnitt 5.5 ausführlich beschrieben.

4) Löschen

Im letzten Schritt werden alle nicht erneut ausgeführten Transformationen und die entsprechenden nicht mehr aktuellen objektorientierten Konstrukte, Parameter-Knoten und Abbildungsknoten gelöscht. Dabei wird ein Logbuch angelegt, das diese Transformationen mit ihren entsprechenden Parametern speichert. Dieses Logbuch kann sich der Benutzer anschließend anzeigen lassen. Die Unterschiede zu dem vor der Nachtransformation vorliegenden objektorientierten Schema sind so leichter festzustellen. Gegebenenfalls kann der Benutzer mit Hilfe dieses Logbuchs die Transformationen manuell wiederholen, die das System wegen fehlender Para-

meter oder veränderter Vorbedingungen nicht automatisch wiederholen konnte.

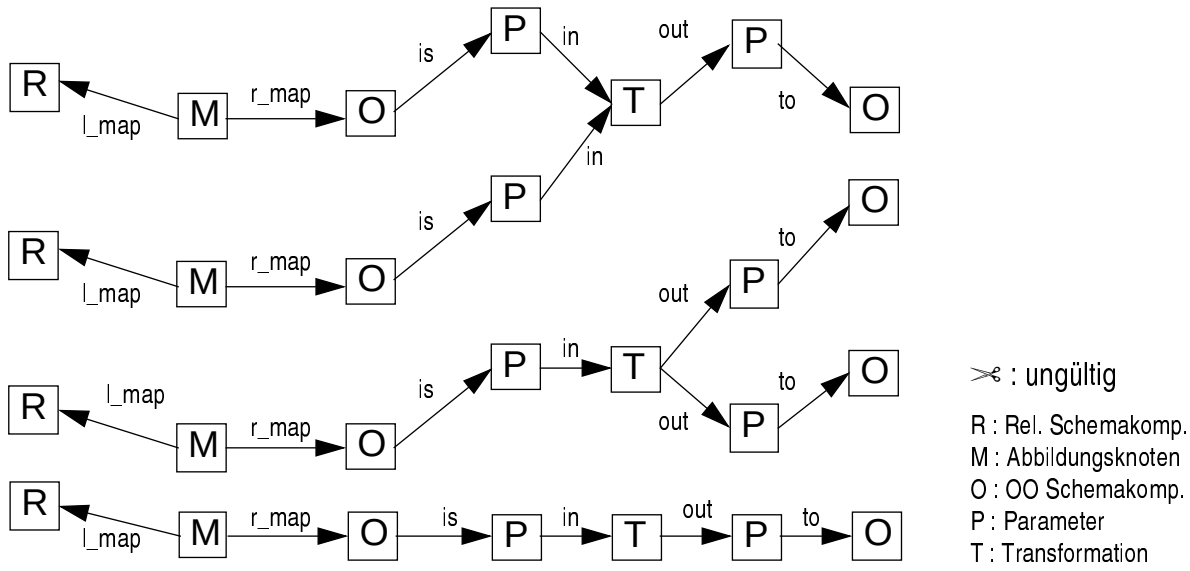


Abb. 5.16: Löschen der ungültigen Knoten

5) Invarianten
Die Invarianten werden überprüft. Falls eine nicht erfüllt, ist wird sie dem Benutzer angezeigt. Dieser muß die Konsistenz dann manuell wiederherstellen. Der Grund hierfür wird in Abschnitt 5.4 erläutert.

Die Konsistenz zwischen dem relationalen und objektorientierten Schema ist somit wieder hergestellt.

5.4 Behandlung von Vorbedingungen

Die Handhabung der dynamischen und statischen Vorbedingungen ist während der Nachtransformation komplizierter. Neben den dynamischen und statischen Vorbedingungen tritt eine dritte Art während der Nachtransformation auf. Alle Eingabeparameter müssen aktuell sein, d.h. *nicht* als ungültig markiert sein. Eine Transformation mit „alten“ objektorientierten Schemakomponenten oder Klonen auszuführen, ist nicht sinnvoll.

Die (der) Eingabeparameter sind nicht aktuell

Mit *nicht aktuell* ist gemeint, daß sie als ungültig markiert sind. Eine Transformation soll während der Nachtransformation ausgeführt werden. Als erstes wird überprüft, ob alle Eingabeparameter aktuell sind. Ist dies der Fall, werden die Eingabeparameter eingelesen, die dynamischen Vorbedingungen überprüft und die entsprechenden Klone angelegt. Andernfalls wird eine Logbuchmeldung konstruiert und der rekursive Aufruf für diese Transformation abgebrochen. Außerdem werden die Parameter-Knoten der Eingabeparameter wieder als ungültig markiert.

Daß Eingabeparameter nicht aktuell sind, kann mehrere Gründe haben. Zum einen kann im relationalen Schema eine Schemakomponente gelöscht worden sein, dessen Abbildung im objektorientierten ein Eingabeparameter war. Oder eine Veränderung im relationale Schema bewirkt, daß die abgebildeten objektorientierten Schemakomponenten sich nicht mehr als Eingabeparameter eignen.

Wenn für alle vorangegangenen Transformationen einer Transformation der Versuch, sie erneut auszuführen, noch nicht unternommen worden ist, können ihre Eingabeparameter nicht aktuell sein. Diese Transformation wird dann nicht ausprobiert. Erst wenn alle Vorgänger-Transformationen nicht mehr als ungültig markiert sind, kann sie ausgeführt werden.

Wenn eine Transformation nicht erneut ausgeführt werden kann, sind die Ausgabeparameter nicht aktuell. Alle nachfolgenden Transformationen haben dann keine aktuellen Eingabeparameter. Dementsprechend wird auch nicht versucht, sie erneut auszuführen. Der rekursive Aufruf wird an dieser Stelle abgebrochen.

Die dynamischen Vorbedingungen haben sich geändert

Die dynamischen Vorbedingungen spielen sich im *Eins-Kontext* ab. Dieser Kontext hängt auch von den vorangegangenen Änderungen und Restrukturierungen ab. Der Eins-Kontext einer Transformation ist die Vereinigung der Eins-Kontexte der betroffenen Schemakomponenten.

Für `MergeClass` beispielsweise sind zwei Klassen und die Assoziation zwischen den beiden betroffen. Der Eins-Kontext von `MergeClass` ist dann die Vereinigung der logischen Umfelder der zwei Klassen und der Assoziation.

Eine Vorbedingung im Eins-Kontext wird spezifisch für einen speziellen Zusammenhang formuliert. Eine Transformation darf nur ausgeführt werden, wenn die Rahmenbedingungen stimmen. Das heißt, spezielle Konditionen müssen gewährleistet sein, hier ein paar Beispiele:

- `SplitClass`: Die Klasseneigenschaften, die abgespaltet werden sollen, müssen zu der Klasse gehören. Außerdem muß die übergebene Menge von Klasseneigenschaften den ganzen Schlüssel oder keine Klasseneigenschaften des Schlüssels enthalten.
- `MergeClass`: Zwei Klassen dürfen nur verschmolzen werden, wenn sie durch eine totale 1:1 (d.h. [1-1]:[1-1]) Beziehung verbunden sind.
- `ClassToRelationship`: Eine Klasse mit Attributen darf nicht in eine Assoziation umgewandelt werden. Falls sie mehr als zwei Beziehungen hat, ist dies ebenfalls nicht erlaubt. Außerdem muß sie mit den zwei anderen Klassen durch totale 1:x Beziehungen verbunden sein.

Für alle Transformationen wird getestet, ob alle Schemakomponenten, die gebraucht werden, vorhanden sind. Der zu restrukturierende Teilgraph muß vollständig existieren. Erst danach kann die Transformation wiederholt werden.

Die statischen Vorbedingungen sind nach der Transformation nicht mehr gültig

Statische Vorbedingungen werden als *Invarianten* implementiert. Diese Invarianten werden nach jeder erstmalig ausgeführten Transformation überprüft.

Würde während der Nachtransformation genauso vorgegangen, könnte dies zu einem unnötigen Abbruch des rekursiven Aufrufs führen. Man betrachte folgendes Szenario:

- Der Benutzer benennt eine Klasse A um in Klasse B.
- Danach benennt er die Klasse B um in C.
- Auf der Klasse C werden Restrukturierungen vorgenommen.
- Zuletzt benennt er eine Klasse X um in B.
- Änderungen im relationalen Schema bewirken die erneute Abbildung der Klasse A.
- Die Nachtransformation wird bei der Ausführung des Umbenennens von A nach B abgebrochen, weil sonst zwei Klassen mit dem Namen B im Schema existieren würden. Alle Restrukturierungen werden damit nicht mehr ausgeführt.

Eine Alternative wäre eine Überprüfung der Invarianten erst am Ende der Nachtransformation. Sind sie nicht erfüllt, wird eine Nachtransformation auf den betroffenen Schemakomponenten durchgeführt. Für das vorhandene Szenario wäre dies eine Lösung. Aber es könnte zu einer Endlosschleife führen. Man betrachte folgendes Szenario:

- Der Benutzer benennt eine Klasse A um in Klasse B.
- Auf der Klasse B werden Restrukturierungen vorgenommen.
- Danach benennt er die Klasse B um in C.
- Zuletzt benennt er eine Klasse X um in B.
- Änderungen im relationalen Schema bewirken die erneute Abbildung der Klasse A.
- Bei der Nachtransformation scheitert eine Restrukturierung, damit wird das Umbenennen von B nach C nicht mehr ausgeführt. Bei der Überprüfung der Invarianten werden zwei Klassen B entdeckt. Die Klasse X ist von der Klasse B (bzw. A) unabhängig. Die Nachtransformation müßte erneut ausgeführt werden. Da aber keine Änderungen während der automatischen Nachtransformation vorgenommen werden können, würden erneut zwei Klassen B entdeckt. Damit hätte man eine Endlos-Schleife.

Deshalb werden die Invarianten zwar am Ende überprüft, aber der inkonsistente Zustand wird dem Benutzer nur gemeldet. Dieser muß die Inkonsistenz selbst beheben.

Am Ende der Migration will der Benutzer eventuell eine ODMG-Text-Ausgabe oder Java generieren. Dies wird nur dann zugelassen, wenn die Invarianten gültig sind, d.h. ein konsistentes Schema vorliegt. Der Benutzer wird „gezwungen“, die Inkonsistenz zu beheben.

5.5 Behandlung der nicht ausgeführten Nachtransformationen

Beim Scheitern einer Transformation wird diese markiert und eine Logbuchmeldung in ein Logbuch gespeichert. Diese Logbuchmeldung entspricht der Art des Fehlers: ein Eingabeparameter ist nicht aktuell, oder eine dynamische Vorbedingung ist nicht erfüllt. Die betroffene Transformation und die Eingabeparameter werden aufgeführt.

Zusätzlich werden alle Transformationen, die im Nachbereich einer solchen Transformation stehen, ebenfalls in dieses Logbuch geschrieben. Vor dem Löschen der ungültig markierten Inkremente (Schemakomponenten und Struktur- bzw. Hilfsknoten) werden diese Transformationen durchlaufen.

Zuerst werden alle Transformationen, die bei der Ausführung gescheitert sind, gesucht. Alle vorangegangenen Transformationen wurden erneut ausgeführt. Für diese Transformationen sind schon Einträge in dem Logbuch vorhanden und sie sind markiert. Ausgehend von diesen Transformationen werden alle nachfolgenden durchlaufen und in das Logbuch geschrieben. Sie werden in Abhängigkeit ihrer jeweils vorangegangenen Transformation(en) aufgezählt. Das heißt, sie werden unter ihrer unmittelbar vorangegangenen Transformation geschrieben und eingerückt. Jede besuchte Transformation wird markiert, um zu vermeiden, daß sie mehrmals aufgeführt wird. Wenn eine Transformation also von mehreren Transformationen abhängt, wird sie nur einmal in das Logbuch geschrieben. Mit Hilfe der Parameter kann der Benutzer jedoch die Abhängigkeiten nachvollziehen.

Kapitel 6

Implementierung und Prototyp

6.1 VARLET-Prototyp

Der VARLET-Prototyp wurde im Rahmen der Projektgruppe VARLET entwickelt und wird in mehreren Diplomarbeiten erweitert. Er läuft auf SUN-Workstations (UNIX) und PROGRES 9.0. Abb. 6.1 zeigt das zugrundeliegende Systemdesign des Prototypen.

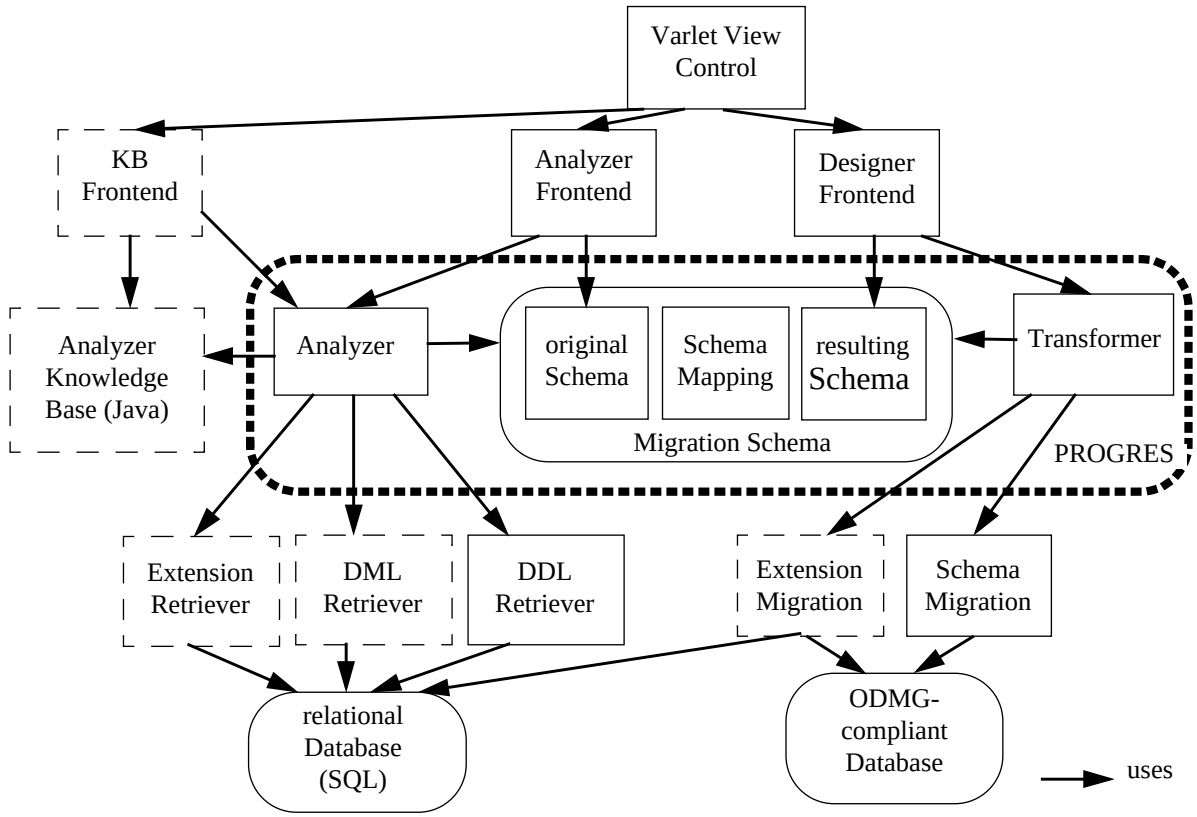


Abb. 6.1: Systemdesign von VARLET

Die gestrichelten Komponenten werden in dieser Arbeit nicht berücksichtigt. Die Benutzeroberflächen sind in Tcl/Tk und über eine C++-Schicht an den PROGRES-Anteil angebunden. Insgesamt wurden ca. 110.000 Zeilen Code in verschiedenen Sprachen in VARLET entwickelt. Davon sind 20.000 Zeilen in PROGRES, daraus werden ca. 200.000 Zeilen C-Code erzeugt. Außerdem wurden ca. 34.000 Zeilen C++, 15.000 Zeilen Tcl/TK, 7500 Zeilen lex&yacc (Retriever) und 26.000 Zeilen Java bis jetzt entwickelt.

Das PROGRES-Rahmenwerk dieser Arbeit besteht aus vier Modulen (`sections`), den Restrukturierungstransformationen, der Abbildung, der Konsistenzerhaltung und den Invarianten. Der PROGRES Anteil der für diese Arbeit entwickelt wurde umfaßt ca. 9.000 Zeilen. Abb. 6.2 zeigt den Umfang der einzelnen Modulen und wie sie sich gegenseitig benutzen.

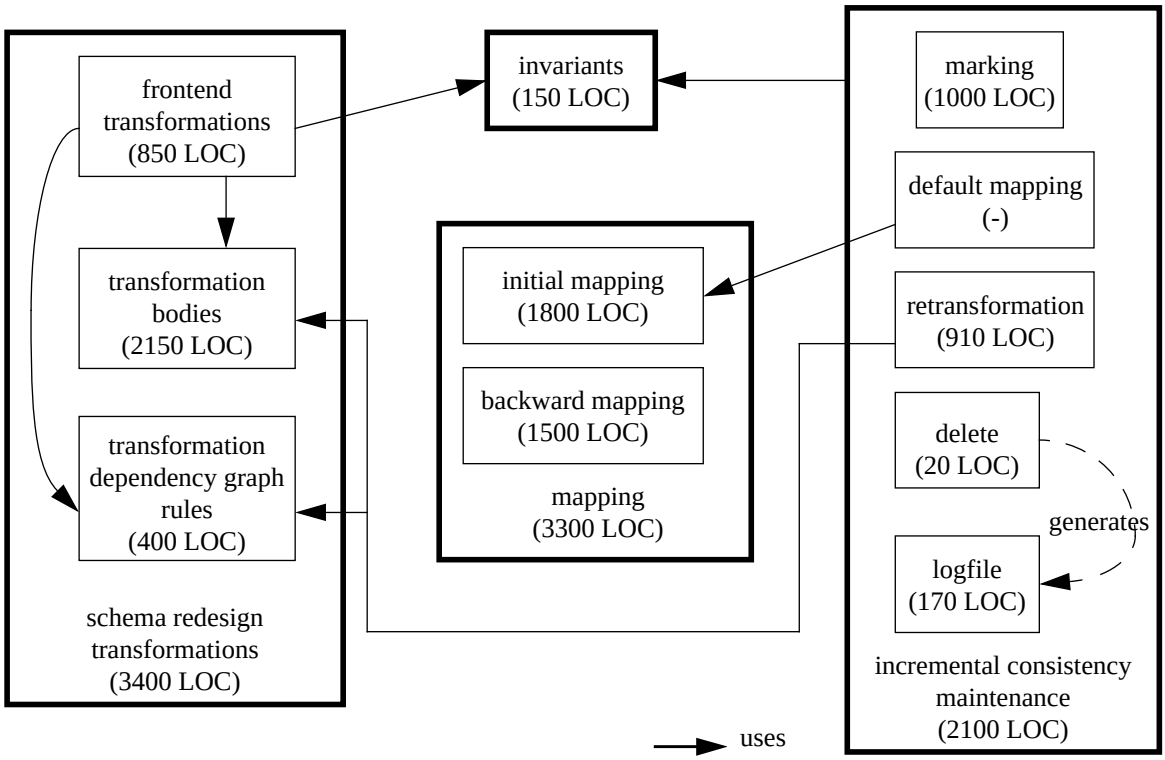


Abb. 6.2: Das PROGRES-Rahmenwerk

Die Operationen, die den Abhängigkeitsgraph erweitern, werden von den Benutzertransformationen (*frontend transformations*) aufgerufen und nicht von den Transformationsrümpfen (*transformation bodies*).

Die generischen Teile sind die Operationen zum Aufbau des Abhängigkeitsgraphen (*transformation dependency graph rules*) und die Transformationsrümpfe, die von allen Benutzer-Transformationen und von der Nachtransformation (*retransformation*) benutzt werden. Die Invarianten (*invariants*) werden von den Benutzer-Transformationen und der Konsistenzerhaltung (*incremental consistency maintenance*) aufgerufen. Die Vorwärtsregeln der initialen Abbildung (*initial mapping*) werden für die Abbildung während der Konsistenzerhaltung (*default mapping*) zum größten Teil wiederverwendet. Bei der Veränderung des Regelsatzes der Transformationen müssen einige dieser Teile angepaßt werden. Das Rahmenwerk wurde so konzipiert, das Erweiterungen bzw. Änderungen einfach durchgeführt werden können.

Anpassungen bei der Erweiterung/Änderung von Restrukturierungstransformationen

Wird eine Restrukturierungstransformation hinzugefügt (gelöscht), muß sie bei den Benutzer-Transformationen zur Verfügung gestellt (gelöscht) werden. Der Rumpf muß geschrieben (gelöscht) und der Aufruf für die Nachtransformation eingefügt (gelöscht) werden. Wenn sie

eine Vorbedingung in Form einer Invariante hat, muß diese bei den Invarianten berücksichtigt (gelöscht) werden.

Falls eine Transformation geändert wird, muß unterschieden werden, ob die Parameter sich ändern, d.h. Veränderung der übergebenen Eingabeparameter, der gebrauchten Eingabeparameter oder der Ausgabeparameter. Sind die Eingabeparameter betroffen, müssen der Test der linken Seite und das Einlesen der Eingabeparameter bei der Nachtransformation angepaßt werden. Außerdem müssen, wie bei der Veränderung der Ausgabeparameter, die Aufrufe für den Aufbau des Abhängigkeitsgraphen bei der Benutzer-Transformation und der Nachtransformation angepaßt werden.

Ändern sich die Vorbedingungen, müssen diese bei der Benutzer-Transformation und der Nachtransformation angepaßt werden. Wird eine Invariante hinzufügt, muß diese außerdem noch bei den Invarianten berücksichtigt werden. Ändert sich eine Invariante lediglich, muß dies nur bei den Invarianten berücksichtigt werden. Ist die Implementierung des Rumpfes betroffen, ändert sich nur etwas bei dem Transformationsrumpf.

Das Beispiel-Szenario in der VARLET-Umgebung

Als nächstes wird der VARLET-Prototyp anhand der beiden Benutzeroberflächen mit dem Beispiel-Szenario vorgestellt.

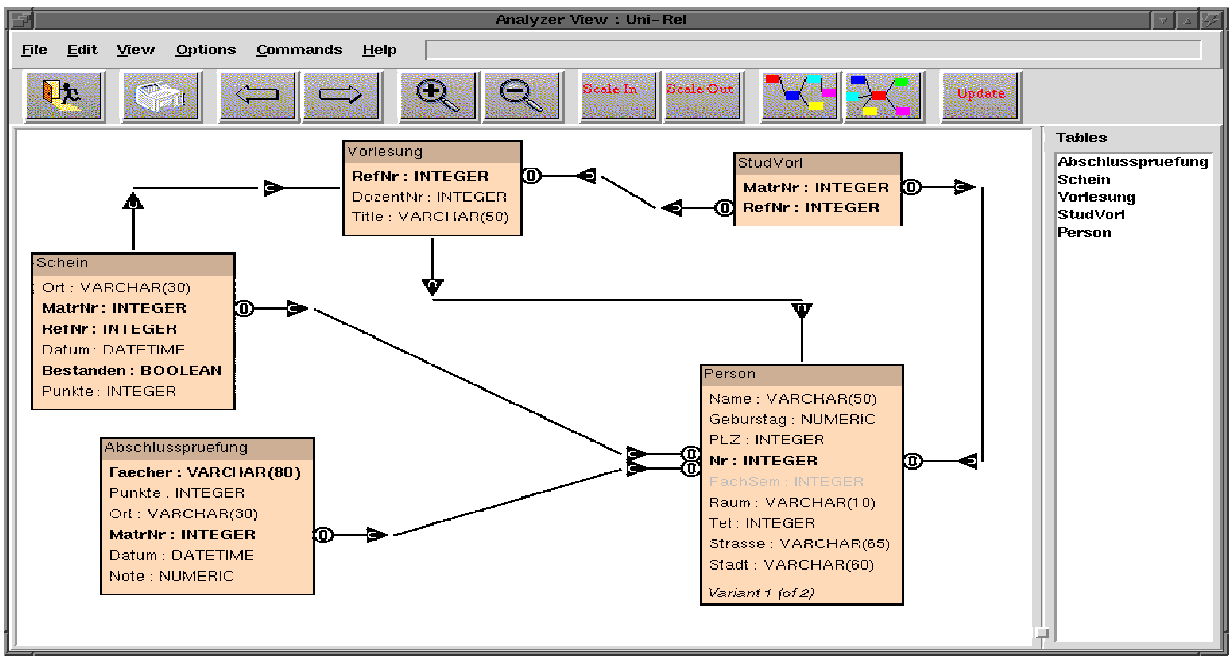


Abb. 6.3: Das relationale Schema nach der Analyse

In Abb. 6.3 wurde für die Tabelle `Abschlusspruefung` ein Schlüssel (Fett hervorgehoben) identifiziert. Die Varianten der Tabelle `Person` sind ebenfalls identifiziert. Die IND kann nur zwischen den Relationen (z.B. `Schein` - `Vorlesung`) oder mit den zugehörigen Attributen (z.B. `Vorlesung` - `StudVorl`) anzeigt werden.

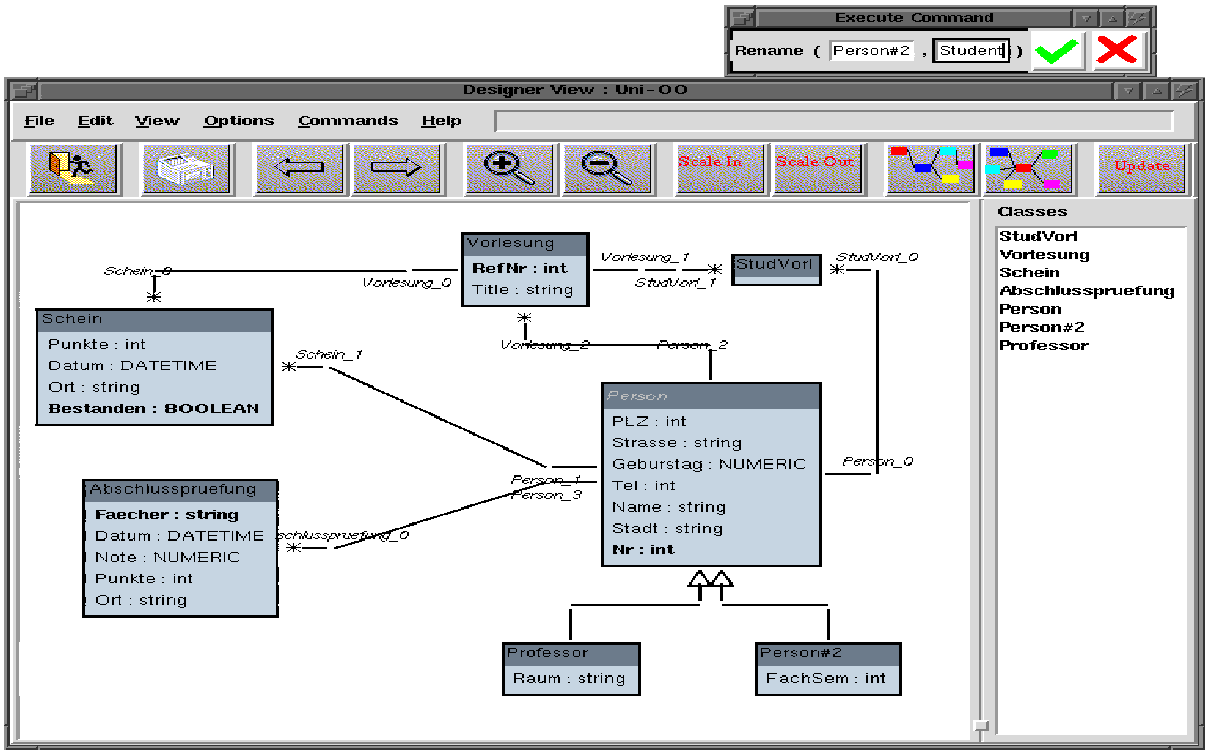


Abb. 6.4: Das objektorientierte Schema - dem relationalen ähnlich

Abb. 6.4 zeigt das objektorientierte Schema nach der initialen Abbildung mit der Umbenennung von `Person#1` zu `Professor` und vor (während) der Umbenennung von `Person#2` zu `Student`. Abb. 6.5 zeigt es nach den Restrukturierungstransformationen aus Abb. 3.5.

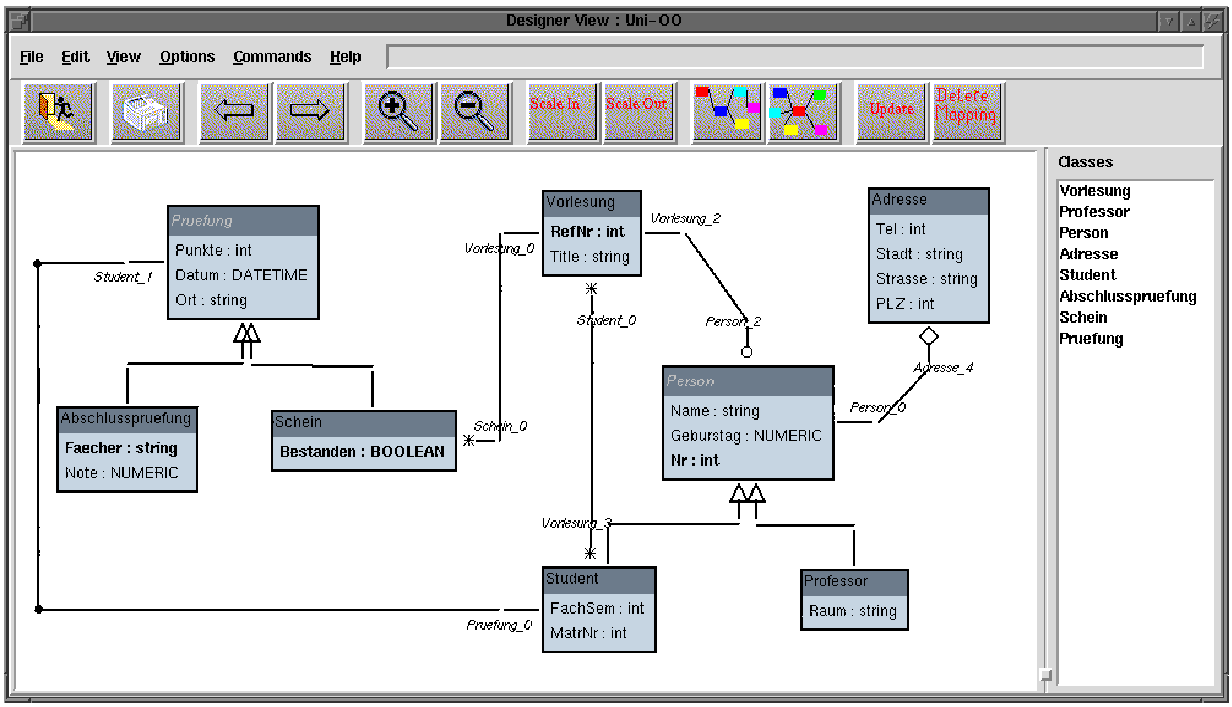


Abb. 6.5: Das objektorientierte Schema

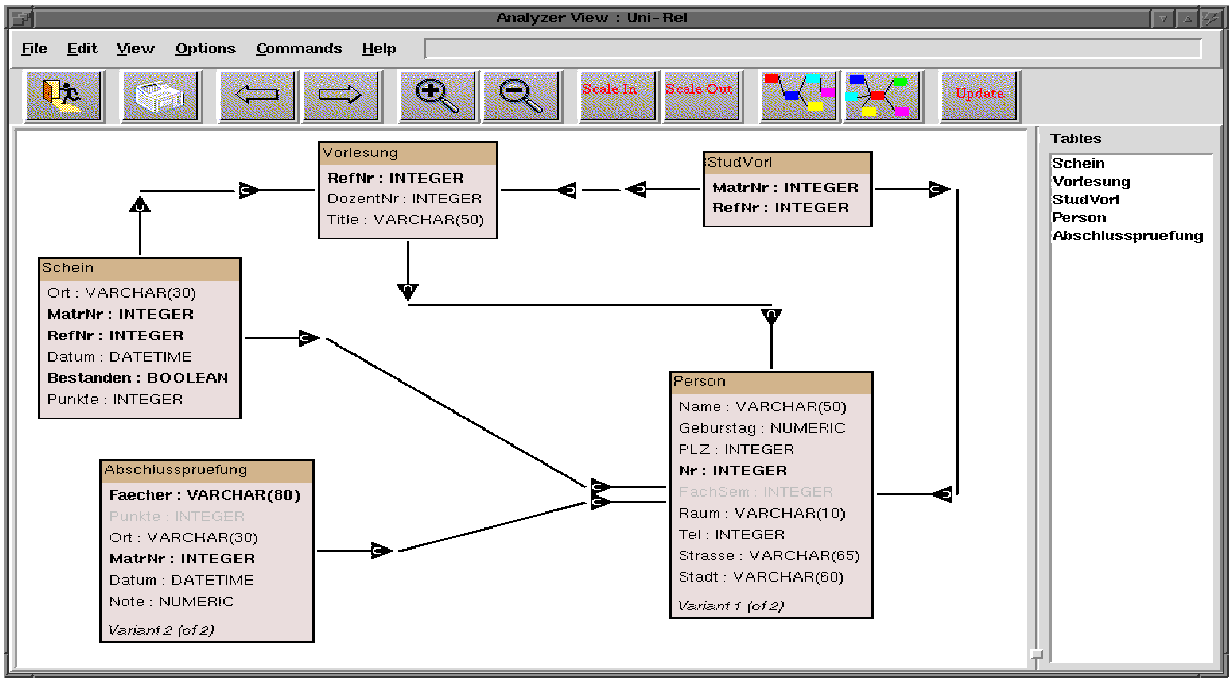


Abb. 6.6: Das relationale Schema nach der erneuten Analyse

Nach der erneuten Analyse besitzt `Abschlusspruefung` zwei Varianten. Das Attribut „Punkte“ (ausgegraut) kommt in der zweiten Variante nicht mehr vor. Abb. 6.7 zeigt dann das endgültige objektorientierte Schema.

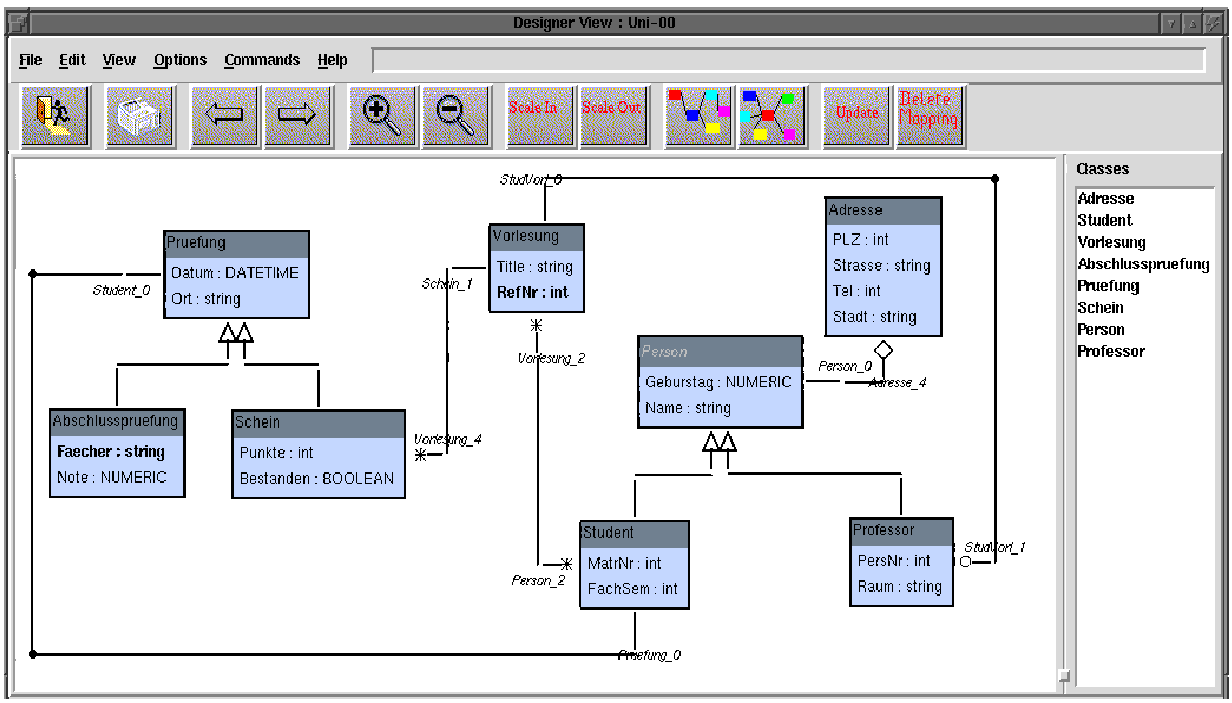


Abb. 6.7: Das endgültige objektorientierte Schema

Nicht alle Transformationen konnten automatisch wiederholt werden. Dafür kann der Benutzer sich das Logbuch anzeigen lassen.

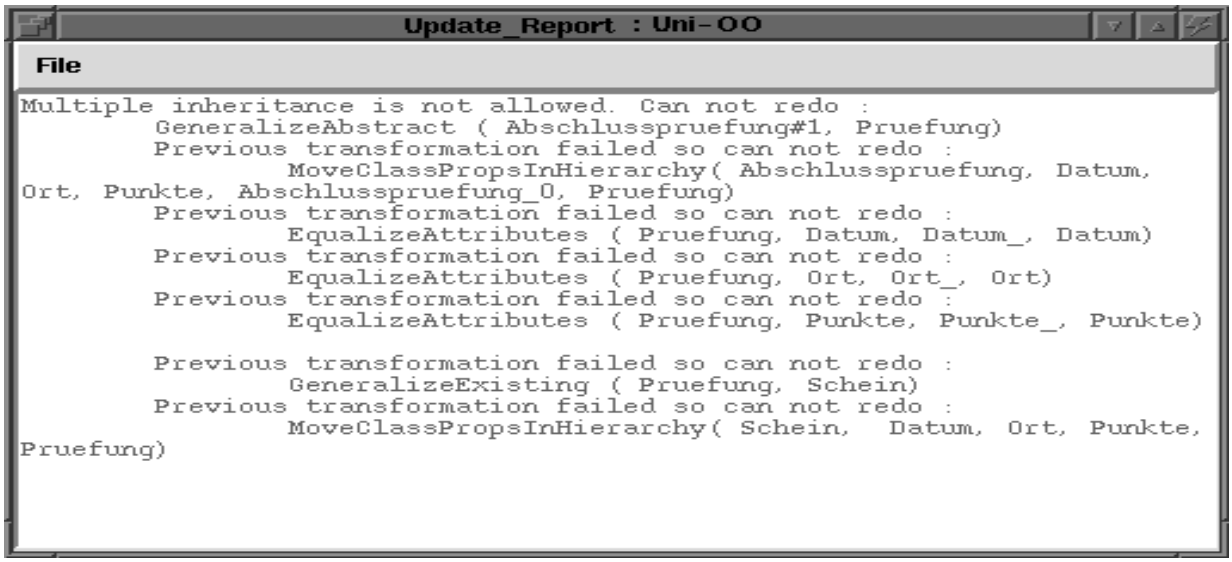


Abb. 6.8: Das Logbuch

Die erneute (initiale) Abbildung erzeugt die Oberlasse `Abschlusspruefung` und die Unterklasse `Abschlusspruefung#1`. Die Nachtransformation probiert `GeneralizeAbstract` auf der Unterklasse `Abschlusspruefung#1` aus. `Abschlusspruefung#1` wird als Eingabe übergeben, weil sie von der Variante, die alle Attribute enthält, abgebildet ist. Die abstrakte Klasse `Pruefung` wird nicht erzeugt, weil mehrfach Vererbung in VARLET nicht erlaubt ist. Im Logbuch erfolgt der Eintrag:

```
Multiple inheritance is not allowed. Can not redo :
      GeneralizeAbstract ( Abschlusspruefung#1, Pruefung)
```

Das Verschieben der Klasseneigenschaften „Datum“, „Ort“, „Punkte“ und „Abschlusspruefung_0“ ist nun nicht mehr möglich. Im Logbuch steht deshalb:

```
Previous transformation failed so can not redo :
     MoveClassPropsInHierarchy( Abschlusspruefung, Datum, Ort,
Punkte, Abschlusspruefung_0, Pruefung)
```

Die nachfolgenden Transformationen können ebenfalls nicht wiederholt werden und sind dementsprechend im Update_Report aufgeführt.

6.2 Implementierung

Die Konsistenzerhaltung besteht aus fünf Schritten: Markierung, Wiederholung der initialen Abbildung, Nachtransformation, Löschen und Überprüfen der Invarianten.

```
transaction Re_Trafo( out Errstr1 : string ; out Errstr2 : string ;
                      out Inv_Errstr1 : string ; out Inv_Errstr2 : string)
    [0:1] =
    use ooschema : OOSchema;
        trafos : Trafo [0:n];
        trafo_update_root : TRAFO_UPDATE_ROOT;
        errstr1, errstr2 : string;
        inv_errstr1, inv_errstr2 : string
    do
        Update_Report ( out trafo_update_root )
      & OldMappings
      & Forward_Pre_Trafo
      & Backward_Pre_Trafo
      & GetOOLogicRoot_Test ( out ooschema )
      & DoDefaultMapping ( ooschema, out errstr1, out errstr2 )
      & ActualiseInitialTrafos
      & IncrGet ( Trafo, out trafos )
      & trafos := ((trafos.valid (self.old = true)) and
                   (trafos.valid (empty ( self.prev ))) )
      & for all trafo := trafos
        do
            Do_Trafo ( trafo, trafo_update_root )
          & Do_Rek_Trafo ( trafo, trafo_update_root )
        end
      & RemoveAllOldStuff : [0:1]
      & Invariants ( out inv_errstr1, out inv_errstr2 )
      & Errstr1 := errstr1
      & Errstr2 := errstr2
      & Inv_Errstr1 := inv_errstr1
      & Inv_Errstr2 := inv_errstr2
    end
end;
```

Abb. 6.9: Die Konsistenzerhaltung

In `Re_Trafo` tritt außerdem noch `Update_Report` (initialisiert das Logbuch) auf. Auf `Update_Report` wird während des Aufbaus des Logbuches eingegangen.

Markierung

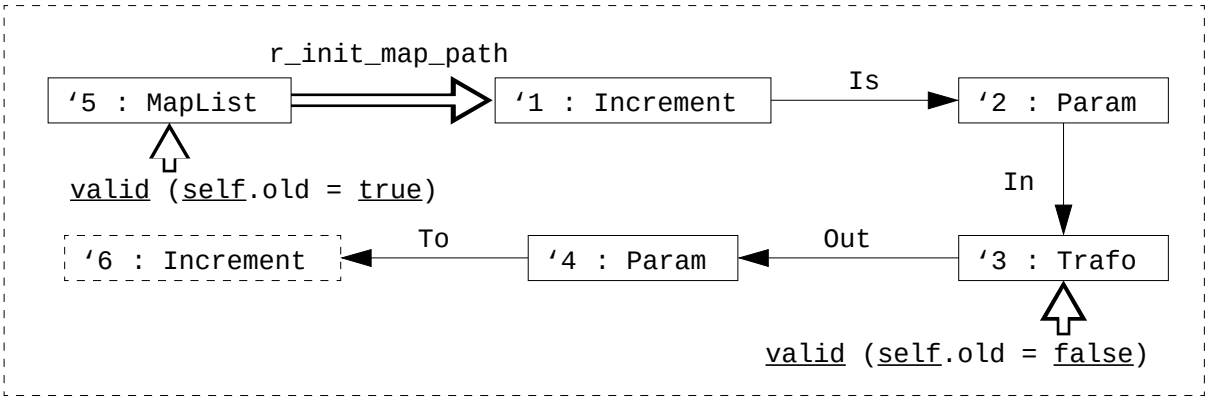
Die Markierung startet mit `OldMappings`, dessen Aufgabe die Markierung des Eins-Kontextes der ungültig markierten relationalen Schemakomponenten ist. Für zwei Klassen, die aus zwei Varianten einer Relation abgebildet sind, macht es keinen Sinn, nur die eine Variante neu abzubilden. Eventuell ist eine neue Variante hinzugekommen oder eine weggefallen. Gegebenenfalls wirkt sich eine Änderung auch auf eine mögliche abstrakte Klasse aus, beispielsweise wenn ein Attribut aus einer Variante gelöscht wird. Deshalb müssen auch die Attribute neu abgebildet werden.

Diese Änderungen wirken sich dann auf die Nachtransformation aus. Deshalb wird `OldMappings` vor der Markierung ausgeführt. Die Implementierung ist im Anhang A zu finden. Als nächstes wird die Vorwärtsmarkierung angestoßen.

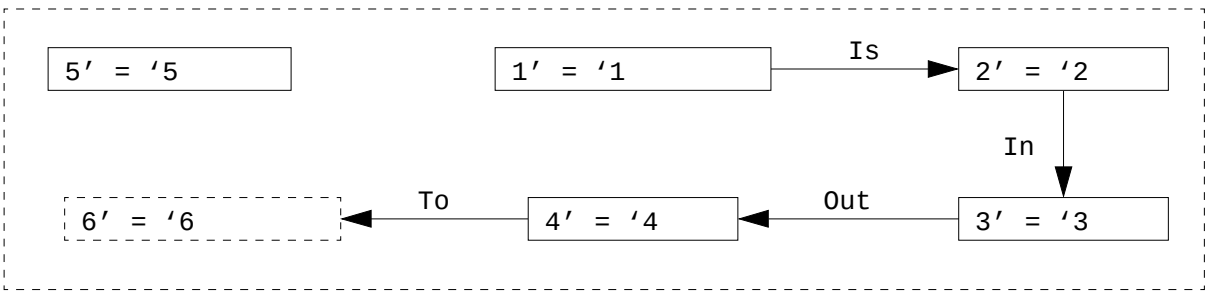
```
transaction Forward_Pre_Trafo [0:1] =
    use trafos : Trafo [0:n];
        trafo : Trafo
    do
        loop
            Forward_Pre_Trafo_Map ( out trafo )
          & trafos := (trafos or trafo)
        end
      & loop
            Forward_Pre_Trafo_Param ( out trafo )
         & trafos := (trafos or trafo)
        end
      & for all t := trafos
        do
            t.old := true
        end
    end
end;
```

Abb. 6.10: Vorwärtsmarkierung

```
production Forward_Pre_Trafo_Map( out trafo : Trafo) [0:1] =
```




```
       ::=
```



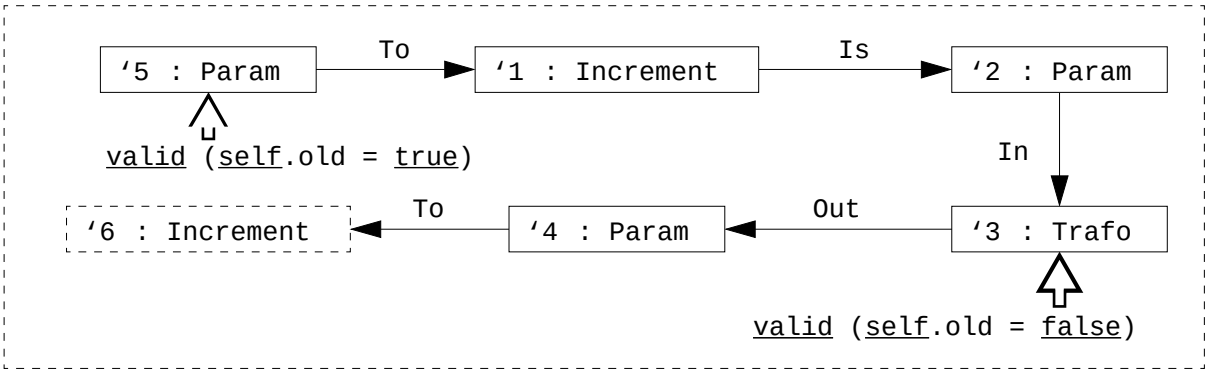

```
    condition ((‘2.old = false) or (‘4.old = false));
    transfer 1’.old := true;
             2’.old := true;
             4’.old := true;
             6’.old := true;
    return trafo := 3’;
end;
```

Abb 6.10.1: Vorwärtsmarkierung der ersten Transformations-Schicht

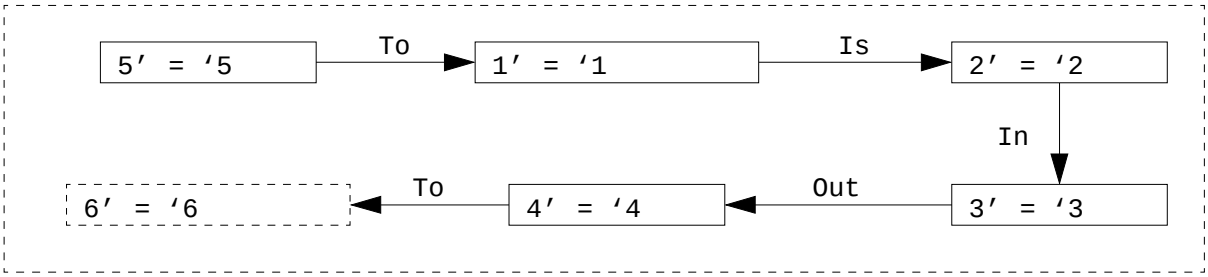
In Abb. 6.10 werden zwei Schleifen (`loop`) durchlaufen. Die erste markiert mit `Forward_Pre_Trafo_Map` alle Transformationen (mit Parameter-Knoten und Objekt/Klone) der ersten Schicht. Die zweite markiert mit `Forward_Pre_Trafo_Param` anschließend alle Transformationen, die auf den markierten Transformationen der ersten Schicht

aufbauen.

```
production Forward_Pre_Trafo_Param( out trafo : Trafo) [0:1] =
```




```
       ::=
```




```
   condition ((‘2.old = false) or (‘4.old = false));
   transfer 1’.old := true;
            2’.old := true;
            4’.old := true;
            6’.old := true;
   return trafo := 3’;
end;
```

Abb. 6.10.2: Vorwärtsmarkierung der restlichen Transformations-Schichten

Die Rückwärtsmarkierung (siehe Abb. 6.11) ist ebenfalls in zwei Schritte unterteilt.

```
transaction Backward_Pre_Trafo [0:1] =
    use rel_incr : Increment;
        trafos : Trafo [0:n]
    do
        loop
            Back_Pre_Trafo
            (* Set old all trafos, params and incrs if there depend from
               a trafo that has to be redone. *)
        end
      & IncrGet ( Trafo, out trafos )
      & trafos := trafos.valid (self.old)
      & for all trafo := trafos
        do
            loop
                PrepareMapping ( trafo, out rel_incr )
              (* Set old all mapping-nodes for the input-params of trafos
                 that have to be redone. *)
            & Extend_PrepareMapping ( rel_incr )
                (* Set old all incrs that depend from the rel_incr. *)
            end
          end
    end
end;
```

Abb. 6.11: Rückwärtsmarkierung

Erstens werden alle Transformationen markiert (`Back_Pre_Trafo`), die zu wiederholen sind, um aktuelle Eingabeparameter und keine Klone zu bekommen. Zweitens werden die relationalen Schemakomponenten markiert (`PrepareMapping`), von denen die Eingaben der Transformationen der ersten Schicht abstammen. Hier muß wieder der Eins-Kontext der relationalen Schemakomponenten berücksichtigt werden (`Extend_PrepareMapping`).

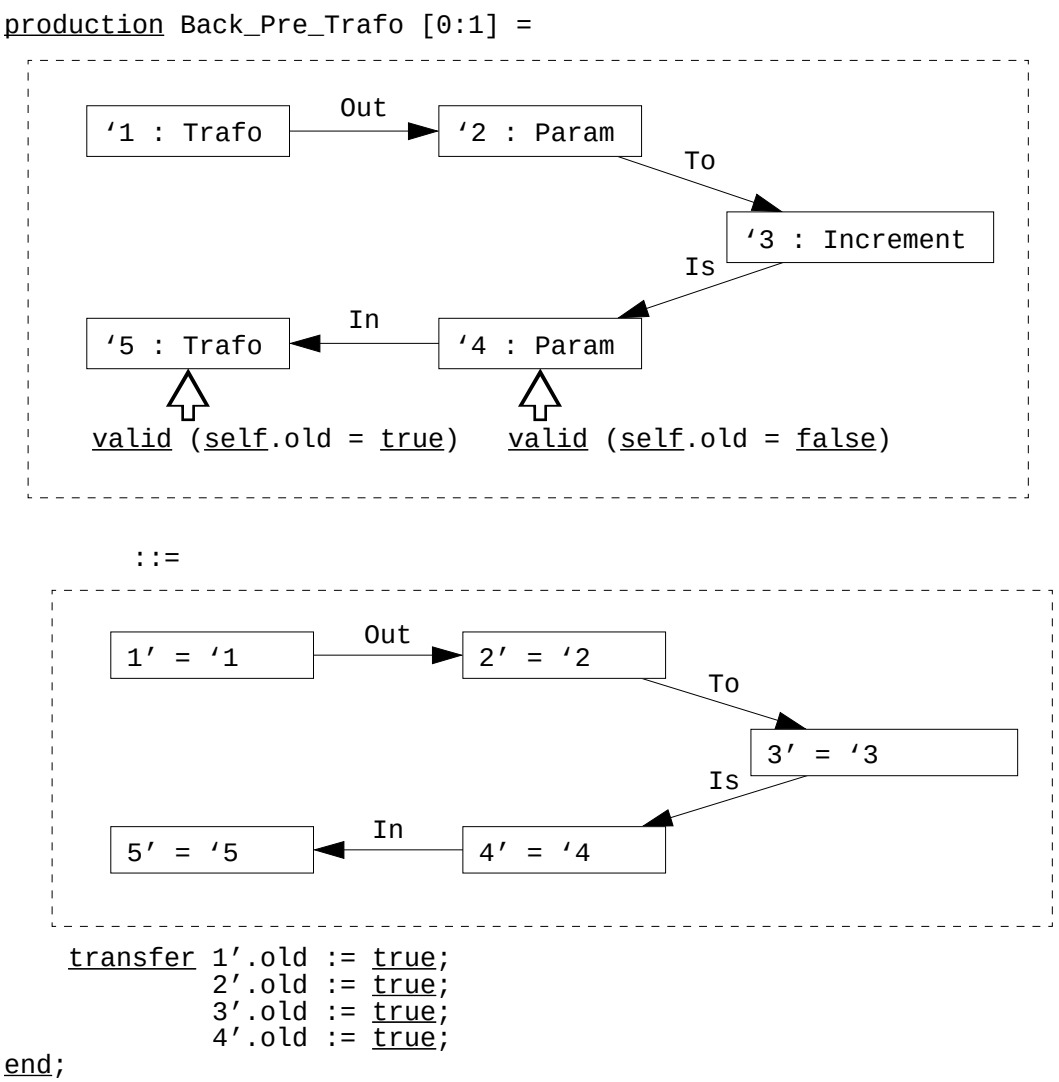
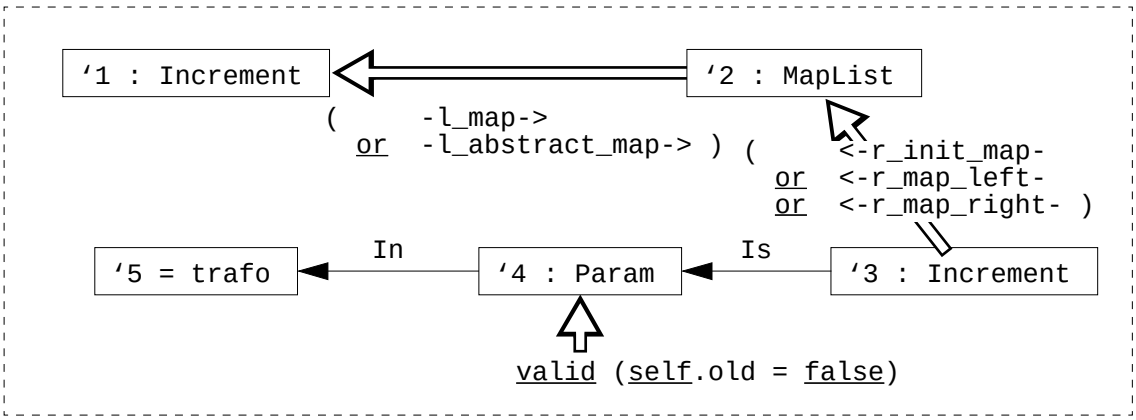


Abb. 6.11.1: Rückwärtsmarkierung der Transformationen

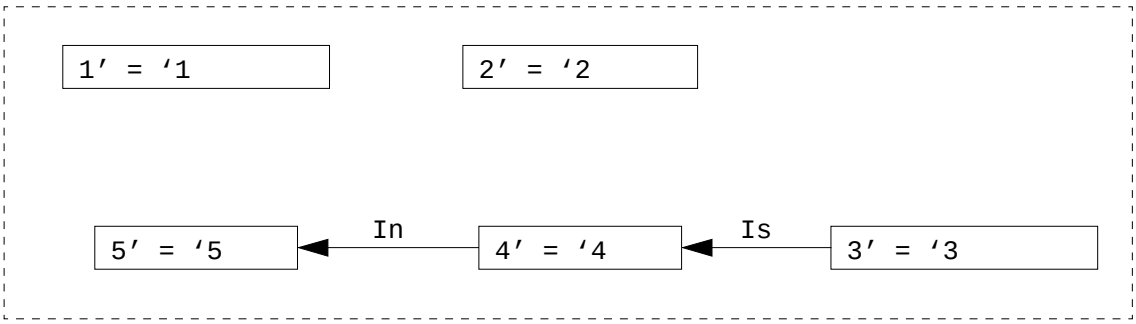
Bisher wurden nur die Kanten `-l_map->` und `-r_map->` erwähnt. Für die Implementierung werden aber mehr benötigt, um eine eindeutige Zuordnung zu gewährleisten.

Neben `-l_map->` gibt es noch eine `-l_abstract_map->` Kante. Diese verläuft von einer Variante zu einem Varianten-Abbildungsknoten. Wenn eine abstrakte Klasse aus mehreren Varianten abgebildet wird, geht von jeder dieser Varianten eine `-l_abstract_map->` Kante zu dem Varianten-Abbildungsknoten. Von diesem geht dann eine `-r_init_map->` Kante zu der abstrakten Klasse.

```
production PrepareMapping( trafo : Trafo ; out rel_incr : Increment)
    [0:1] =
```




```
       ::=
```




```
   transfer 2'.old := true;
            3'.old := true;
            4'.old := true;
   return rel_incr := 1';
end;
```

Abb. 6.11.2: Rückwärtsmarkierung - Abbildung vorbereiten

Die `-r_init_map->` Kante wird von den Abbildungsknoten zu den objektorientierten Schemakomponenten während der initialen Abbildung gezogen. Wenn später die Abbildung erweitert wird, bleibt die `-r_init_map->` Kante auf dem entsprechenden Klon und eine `-r_map->` Kante wird von dem Abbildungsknoten zu den korrespondierenden Ausgangsparametern gezogen. Somit ist gewährleistet, daß bei der Aktualisierung die Eingabeparameter (Objekt oder Klon) der Transformationen der ersten Schicht erreicht werden und nicht die restrukturierten Schemakomponenten.

Zusätzlich gibt es noch die `-r_map_left->` und `-r_map_right->` Kanten, die von einem IND-Abbildungsknoten zu den Traversierungspfaden gehen. Dies hat wieder den Grund, über die `-r_init_map->` Kante die entsprechende Assoziation und nicht einen Traversierungspfad zu erreichen.

```
transaction Extend_PrepareMapping( rel_incr : Increment) [0:1] =
    use cl : Class [0:1];
        attrs : Attr [0:n];
        ookeys : OOKey [0:n];
        tll : TopLevelLogic;
        map_rel : MapRelation;
        maprattr : MapRAttr;
        mapkey : MapKey
    do
        choose
        when not empty ( rel_incr.instance of RAttr )
        then
            tll := (rel_incr.ToEnvList) : Relation [0:1]
          & Prepare_Relation_Mapping ( tll.instance of Relation )
        else
          when not empty ( rel_incr.instance of Var )
          then
              tll :=
                  (rel_incr.ToEnvList.( instance of VarCon ).-vars_of->) :
                    Relation [0:1]
            & Prepare_Relation_Mapping ( tll.instance of Relation )
        else
          when not empty ( rel_incr.instance of IND )
          then
              tll := rel_incr : IND [0:1]
            & Prepare_IND_Mapping ( tll.instance of IND )
        else
            skip
        end
    end
end;
```

Abb. 6.11.3: Abbildung des Eins-Kontextes vorbereiten

In Abb. 6.11.3 wird unterschieden, ob es sich um den Eins-Kontext einer Relation (Attribute und Varianten) - siehe `Prepare_Relation_Mapping` - oder den einer Beziehung - siehe `Prepare_IND_Mapping` - handelt.

```
transaction Prepare_Relation_Mapping( rel : Relation) [0:1] =
    use map_rel : MapRelation [0:1];
        map_vars : MapVariant [0:n];
        classes : Class [0:n];
        map_rattrs : MapRAttr [0:n];
        attrs : Attr [0:n];
        map_keycons : MapKeyCon [0:n];
        ookeycons : OOKeyCon [0:n];
        map_keys : MapKey [0:n];
        ookeys : OOKey [0:n];
        incrs : Increment [0:n]
    do
        map_rel := (rel.<-l_map-) : MapRelation [0:1]
      & map_vars := map_rel.lc_item.instance of MapVariant
      & classes := map_vars.( -r_map->
                              or  -r_init_map-> ).instance of Class
      & map_rattrs := map_vars.lc_item.instance of MapRAttr
      & attrs := map_rattrs.( -r_map->
                              or  -r_init_map-> ).instance of Attr
      & map_keycons := map_vars.lc_item.instance of MapKeyCon
      & ookeycons := map_keycons.( -r_map->
                                   or  -r_init_map-> ).instance of OOKeyCon
      & map_keys := map_keycons.lc_item.instance of MapKey
      & ookeys := map_keys.( -r_map->
                             or  -r_init_map-> ).instance of OOKey
      & incrs := (map_rel or map_vars or classes or map_rattrs or attrs or
                  map_keycons or ookeycons or map_keys or ookeys )
```

```
      & for all incr := incrs
        do
            incr.old := true
        end
      & rel.to_be_mapped := true
    end
end;
```

Abb. 6.11.3.1: Abbildung des Eins-Kontextes einer Relation vorbereiten

```
transaction Prepare_IND_Mapping( ind : IND) [0:1] =
    use map_rind : MapRIND [0:1];
        map_iind : MapIIND [0:1];
        rel_ships : Relationship [0:n];
        left_tp, right_tp : TravPath [0:1];
        incrs : Increment [0:n]
    do
        choose
        when not empty ( ind.<-l_map-.instance of MapIIND )
        then
            map_iind := (ind.<-l_map-) : MapIIND [0:1]
          & rel_ships := (map_iind.( -r_map->
                                    or  -r_init_map-> )) : Inherit [0:n]
          & incrs := (map_iind or rel_ships)
        else
            map_rind := (ind.<-l_map-) : MapRIND [0:1]
          & rel_ships := (map_rind.( -r_map->
                                   or  -r_init_map-> )) : Relationship [0:n]
          & left_tp := (map_rind.-r_map_left->) : TravPath [0:1]
          & right_tp := (map_rind.-r_map_right->) : TravPath [0:1]
          & incrs := (map_rind or rel_ships or left_tp or right_tp)
        end
      & for all incr := incrs
        do
            incr.old := true
        end
      & ind.to_be_mapped := true
    end
end;
```

Abb. 6.11.3.2: Abbildung des Eins-Kontextes einer Beziehung vorbereiten

Wiederholung der initialen Abbildung

Die Wiederholung der Abbildung (`DoDefaultMapping`) verläuft ähnlich wie die initiale Abbildung (siehe Anhang A). Danach werden mit `ActualiseInitialTrafos` die Eingabeparameter der Transformationen der ersten Schicht aktualisiert.

```
transaction ActualiseInitialTrafos [0:1] =
    use trafos : Trafo [0:n]
    do
        IncrGet ( Trafo, out trafos )
      & for all trafo := trafos
        do
          loop
            choose
                ActualiseInitialNotNeededTrafo ( trafo )
              & ActualiseOutParamForNotNeededTrafo ( trafo )
            else
                  ActualiseInitialNotNeededTrafoForAbstractMapping ( trafo )
                & ActualiseOutParamForNotNeededTrafo ( trafo )
            else
                  ActualiseInitialNotNeededTrafoForLeftTravPath ( trafo )
                & ActualiseOutParamForNotNeededTrafo ( trafo )
            else
                  ActualiseInitialNotNeededTrafoForRightTravPath ( trafo )
                & ActualiseOutParamForNotNeededTrafo ( trafo )
```

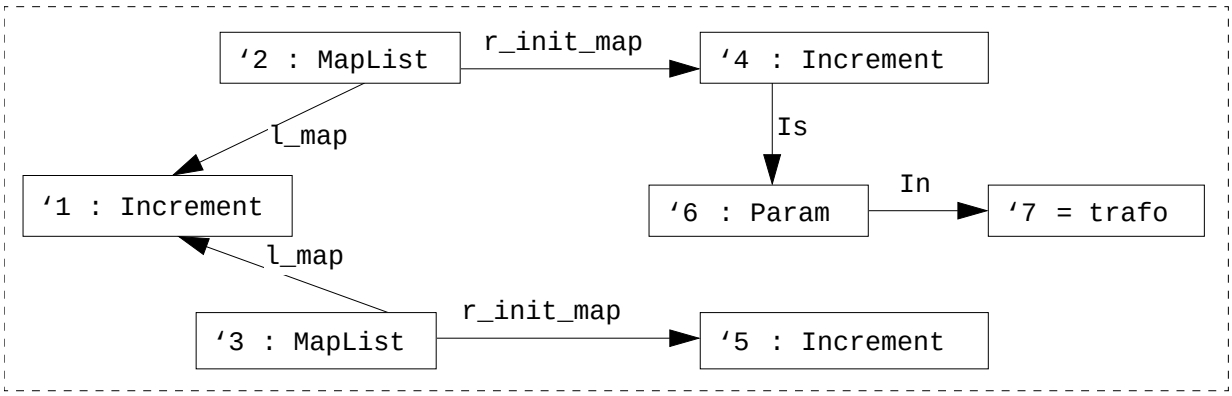
```
            else
                ActualiseInitialTrafo ( trafo )
            else
                ActualiseInitialTrafoForAbstractMapping ( trafo )
            else
                ActualiseInitialTrafoForLeftTravPath ( trafo )
            else
                ActualiseInitialTrafoForRightTravPath ( trafo )
            end
          end
        end
    end
end;
```

Abb. 6.12: Initiale Aktualisierung

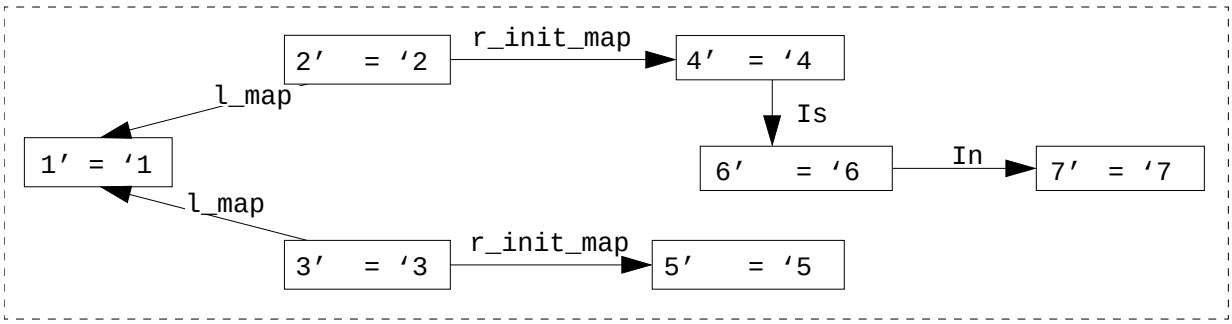
Wegen der verschiedenen Abbildungskanten gibt es jeweils vier Produktionen, die die Aktualisierung ausführen. Um die relationalen und objektorientierten Schemakomponenten eindeutig zuordnen zu können, gibt es folgende Kombinationen:

- `<-l_map-` und `-r_init_map->`
- `<-l_abstract_map-` und `-r_init_map->`
- `<-l_map-` und `-r_map_left->`
- `<-l_map-` und `-r_map_right->`

```
production ActualiseInitialNotNeededTrafo ( trafo : Trafo) [0:1] =
```




```
        ::=
```




```
    condition (‘4.old = true) and (‘7.old = false);
    transfer 2’.old := false;
             4’.old := false;
             6’.old := false;
             3’.old := true;
             5’.old := true;
end;
```

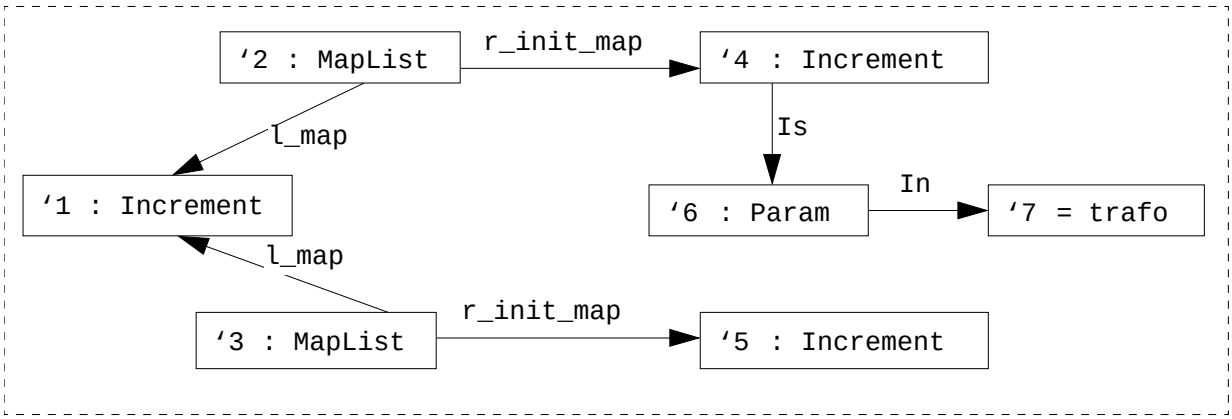
Abb. 6.12.1: Initiale Aktualisierung der nicht gebrauchten Objekte

Für `<-l_map-` und `-r_init_map->` werden `ActualiseInitialNotNeededTrafo` und `ActualiseInitialTrafo` vorgestellt, für die anderen müssen nur die Kanten entsprechend ausgetauscht werden.

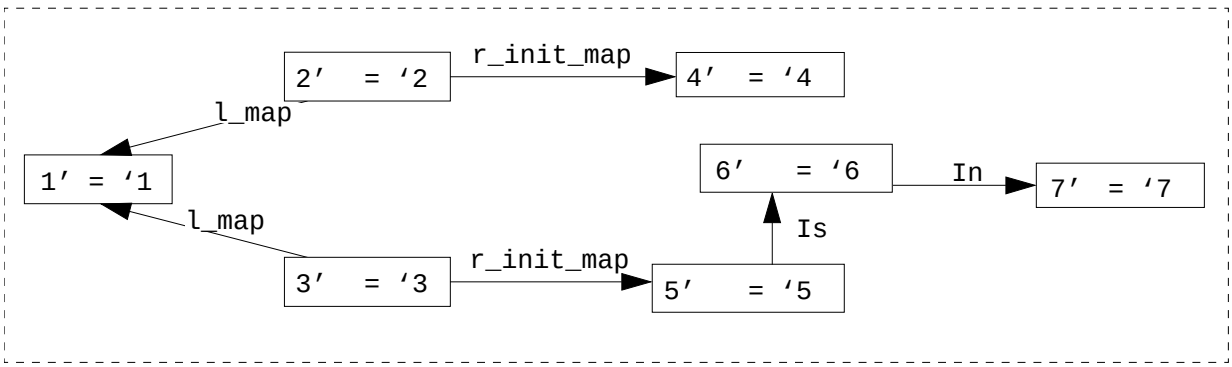
```
transaction ActualiseOutParamForNotNeededTrafo( trafo : Trafo) [0:1] =
    use params : Param [0:n];
        incrs : Increment [0:n]
    do
        params := trafo.-Out->
      & incrs := params.-To->
      & incrs := (params or incrs)
      & for all incr := incrs
        do
            incr.old := false
        end
    end
end;
```

Abb. 6.12.2: Aktualisierung der Ausgabeparameter und Objekte für nicht wiederholte Transformationen

```
production ActualiseInitialTrafo ( trafo : Trafo) [0:1] =
```




```
        ::=
```




```
    condition (‘6.old = true) and (‘7.old = true);
    transfer 2’.old := true;
             4’.old := true;
             6’.old := false;
             3’.old := false;
             5’.old := false;
end;
```

Abb. 6.12.3: Initiale Aktualisierung der Transformationen der ersten Schicht

Die objektorientierten Schemakomponenten, die nicht gebraucht werden, werden als ungültig markiert. Obwohl nur die markierten relationalen Schemakomponenten abgebildet werden, bekommt man einige objektorientierte Schemakomponenten, die nicht gebraucht werden.

Angenommen im Beispiel-Szenario wird „DozentNr“ in „ProfNr“ in der Relation `Vorlesung` umbenannt (siehe Abb. 3.3). Während der Nachtransformation muß die Spezialisierung der Assoziation zwischen `Vorlesung` und *`Person`* in die Assoziation zwischen `Vorlesung` und `Professor` (Abb. 3.5 und 3.7) wiederholt werden. Davor wurde das Attribut „Nr“ von `Person` nach „PersNr“ von `Professor` verschoben. Die initiale Abbildung für `Professor` muß wiederholt werden. Die Varianten `Professor` und `Student` werden während der Rückwärtsmarkierung als geändert markiert. Die Transformation auf der Klasse `Student` ist nicht (als ungültig) markiert. Die Klasse `Student` wird neu erzeugt, obwohl sie nicht gebraucht wird. Die neu erzeugte Klasse `Student` wird als ungültig markiert, da sie als Eingabe nicht gebraucht wird. Alle Klone, Eingabe-, Ausgabeparameter und objektorientierten Schemakomponenten die aus solchen Transformationen resultieren werden als nicht ungültig markiert, da sie sich nicht verändern haben.

Nachtransformation

Während der Nachtransformation wird mit einer Transformation der ersten Schicht begonnen (siehe Abb. 6.9). Dafür wird `Do_Trafo` aufgerufen. Hier werden die Transformationen wiederholt. Als erstes wird hier überprüft, ob alle Eingabeparameter aktuell sind. Danach wird mit dem Konstrukt `choose when ... else when ...` die Art der übergebenen Transformation herausgefunden. Diese wird dann erneut ausprobiert. Danach werden die Ausgabeparameter aktualisiert bzw. wird ein Eintrag ins Logbuch vorgenommen.

```
transaction Do_Trafo( trafo : Trafo ; trafo_update_root : TRAFO_UPDATE_ROOT)
   [0:1] =
    use cls : Class [0:n];
        oldcl, newcl : Class;
        part, cl, cl1, cl2 : Class;
        attr, attr1, attr2 : Attr;
        newattr : Attr [0:1];
        attrs : Attr [0:n];
        relship : Relationship;
        assoc, assoc1, assoc2 : Assoc;
        aggr : Aggr;
        inh, newinh : Inherit;
        oldinh : Inherit [0:1];
        inhs : Inherit [0:n];
        cps : ClassProp [0:n];
        tps : TravPath [0:n];
        tp1, tpold1, tp2, tpold2 : TravPath;
        Card, tpnew, tpold : TravPath;
        ookeycon : OOKeyCon;
        ookeys : OOKey [0:n];
        ook : OOKey;
        inc : Increment;
        namedinc : NamedIncrement;
        incs, incs_ : Increment [0:n];
        STR1, STR2 : STRING;
        in_incrs, out_incrs : Increment [0:n];
        maplists, maplists_abs : MapList [0:n];
        mapincrs, mapincrs_abs : Increment [0:n];
        errStr : string := "";
        incrs : Increment [0:n];
        params : Param [0:n];
        OK, ok : boolean := false
```

```
do
    incrs := trafo.<-In-
  & for all incr := incrs
    do
        OK := (OK or (incr.old))
    end
  & choose
    when not (OK)
    then
        choose
        when (trafo.art = "SplitClass")
        then
            choose
                oldcl := (trafo.GetParam ( 1 ).<-Is-) : Class [0:1]
              & cps := (trafo.GetParams ( 2 ).<-Is-) : ClassProp [0:n]
              & STR1 := (trafo.GetParam ( 3 ).<-Is-) : STRING [0:1]
              & attrs := ((cps.instance of Attr) and
                          (oldcl.lc_item.instance of Attr))
              & ookeys := attrs.ToEnvList.instance of OOKey
              & for all k := ookeys
                do
                    choose
                    when (card ( attrs and (k.lc_item.instance of Attr) )
                              # card ( (k.lc_item.instance of Attr) ) )
                    then
                        ok := true
                    end
                end
              & choose
                when ok
                then
                    CreateTrafoUpdateWithParamUpdate
                    ( "An incomplete key is passed, can not redo : ",
                      trafo, trafo_update_root )
                else
                  when
                    not (card ( cps and
                         (cl1.lc_item.instance of ClassProp) )
                         = card ( cps ) )
                  then
                    CreateTrafoUpdateWithParamUpdate
                    (
                      "All class properties are not in the same class,
                       can not redo : ", trafo, trafo_update_root )
                else
                    in_incrs := (oldcl or cps or STR1)
                  & CloneAndInsertParams ( trafo, in_incrs )
                  & maplists := (oldcl.<-r_map-.instance of MapList)
                  & SplitClass
                    ( oldcl, cps, STR1, out newcl, out assoc, out tpold,
                      out tpnew, out ookeycon )
                  &
                    out_incrs :=
                      (oldcl or cps or newcl or assoc or tpnew or tpold
                       or ookeycon)
                    & Re_Arange_Out_Params ( trafo, out_incrs )
                    & mapincrs := (newcl or assoc)
                    & ActualiseMapping ( maplists, mapincrs )
                end
            else
                CreateTrafoUpdateWithParamUpdate
                ( "Internal error for : ", trafo, trafo_update_root )
            end
        else
          when (trafo.art = "MergeClass")
          ......
```

```
          & loop
                Actualise ( trafo )
            end
        else
            CreateTrafoUpdateWithParamUpdate
             ( "Not every parameter is up to date for : ", trafo,
               trafo_update_root )
        end
    end
end;
```

Abb. 6.13: Nachtransformation

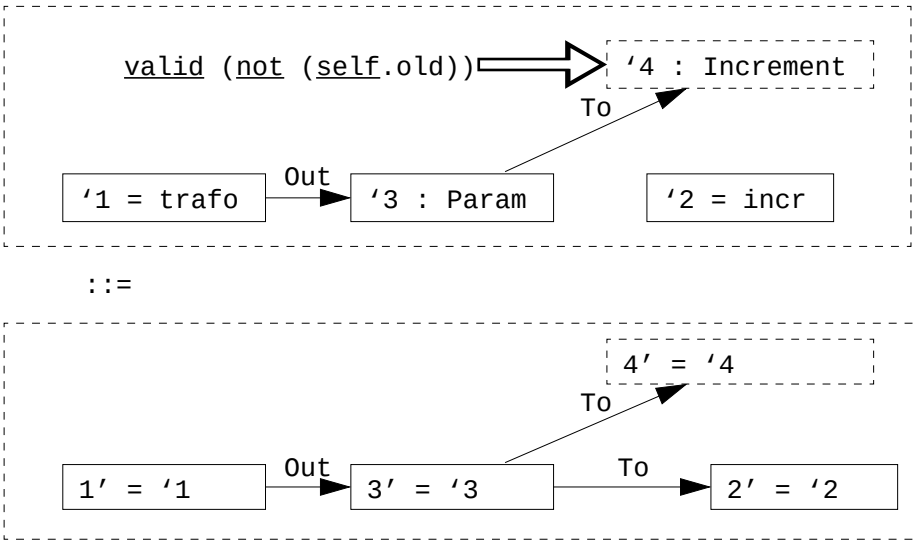
Wenn eine Transformation ausprobiert wird, werden zuerst die Eingabeparameter initialisiert und die dynamischen Vorbedingungen überprüft. Anschließend werden die Eingabeparameter als Klone gespeichert (siehe `CloneAndInsertParams`). Diese ersetzen dann die objektorientierten Schemakomponenten.

```
transaction CloneAndInsertParams( trafo : Trafo ; incrs : Increment [0:n])
    [0:1] =
    use new_incrs : Increment [0:n];
        new_incr : Increment
    do
        for all incr := incrs
        do
          choose
          when not empty ( incr.instance of STRING )
          then
              new_incrs := (new_incrs or incr)
          else
              CopyIncrement ( incr, out new_incr )
            & new_incrs := (new_incrs or new_incr)
          end
        end
    end
end;
```

Abb. 6.13.1: Klone für Parameter anlegen

STRING-Knoten brauchen keine Klone, da sie im weiteren nicht mehr gebraucht werden (siehe Abb. 6.13.1). `CopyIncrement` legt einen Klon an und ersetzt die objektorientierte Schemakomponente (siehe Anhang B).

```
production Re_Arange_Out_Param( trafo : Trafo ; incr : Increment) [0:1] =
```

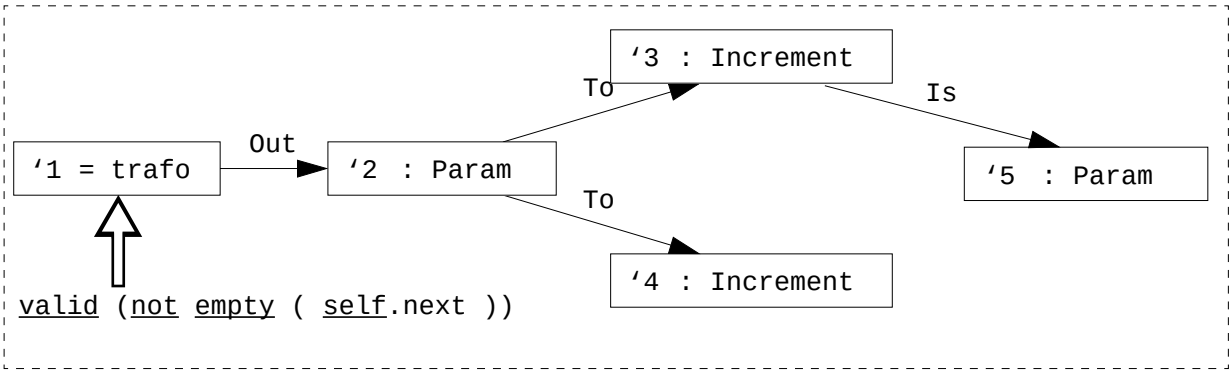



```
    condition (‘3.no = ‘2.no);
    transfer 1’.old := false;
             2’.old := false;
             2’.history := false;
             3’.old := false;
             3’.text := [ not empty ( ‘2.instance of NamedIncrement ) ::
                          (‘2 : NamedIncrement [1:1]).Name | "" ] ;
             4’.old := true;
end;

transaction Re_Arange_Out_Params( trafo : Trafo ; incrs : Increment [0:n])
    [0:1] =
    for all incr := incrs
    do
        Re_Arange_Out_Param ( trafo, incr )
    end
end;
```

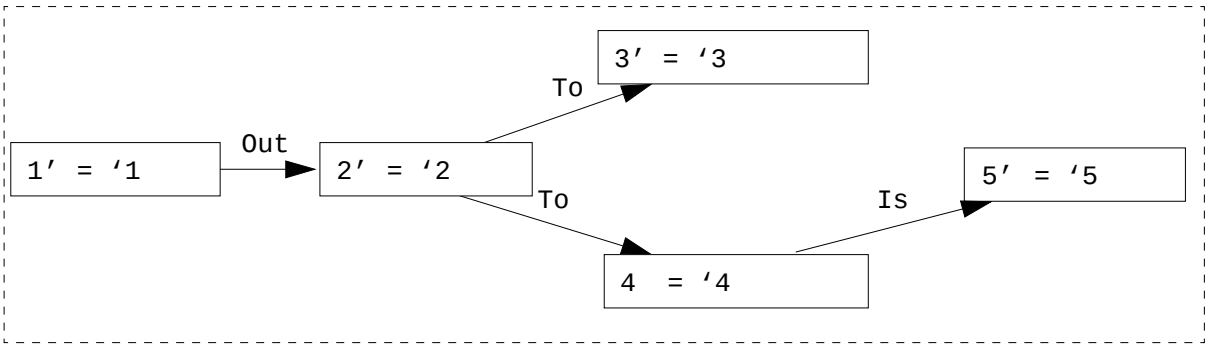
Abb. 6.13.2: Ausgabeparameter aktualisieren

In `Re_Arange_Out_Params` werden für alle Ausgabeparameter die alten objektorientierten Schemakomponenten bzw. die Klone durch die neu erzeugten Schemakomponenten ersetzt bzw. als Ausgabeparameter einsortiert. Die `-To->` Kante wird für die Aktualisierung gebraucht und deshalb nicht gelöscht. Als letztes werden die Parameter für die nachfolgenden Transformationen einsortiert (siehe `Actualise`).

```
production Actualise( trafo : Trafo) [0:1] =
```




```
      ::=
```




```
   condition (‘1.old = false) and (‘5.old = true);
   transfer 5’.old := false;
            4’.old := false;
            3’.old := true;
end;
```

Abb. 6.13.3: Nachfolgende Transformationen aktualisieren

```
transaction Do_Rek_Trafo( trafo : Trafo ;
                          trafo_update_root : TRAFO_UPDATE_ROOT) [0:1] =
    use OK : boolean := false;
        prev_trafos : Trafo [0:n]
    do
        for all next_trafo := trafo.next
        do
            OK := false
          & prev_trafos := next_trafo.prev
          & for all prev_trafo := prev_trafos
            do
                OK := (OK or prev_trafo.old)
            end
          & choose
            when ((not (OK)) and (next_trafo.old))
            then
                Do_Trafo ( next_trafo, trafo_update_root )
              & Do_Rek_Trafo ( next_trafo, trafo_update_root )
            else
                skip
            end
        end
    end
end;
```

Abb. 6.14: Rekursiver Aufruf der nachfolgenden Transformationen

Ausgehend von einer Transformation werden in `Do_Rek_Trafo` alle nachfolgenden Transformationen nacheinander (äußere `for_all` Schleife) ausprobiert. Für die ausgewählte Transformation wird überprüft, ob alle vorangegangenen erfolgreich ausgeführt wurden. Ist dies der Fall, wird sie ausprobiert und der rekursive Aufruf auf ihr gestartet. Andernfalls wird der Aufruf abgebrochen und mit einer Transformation der darüberliegenden Schicht weitergemacht.

Löschen

Während des Löschvorgangs (siehe `RemoveAllOldStuff`) wird das Logbuch angelegt. Danach werden die zu löschenden Knoten aus ihren Listenstrukturen und am Ende die Knoten selbst gelöscht.

```
transaction RemoveAllOldStuff =
    use incrs : Increment [0:n];
        trafos : Trafo [0:n]
    do
        IncrGet ( Trafo, out trafos )
      & trafos := trafos.valid (self.old and self.Fail)
      & for all trafo := trafos
        do
            Trafo_Update_Report ( trafo )
        end
      & IncrGet ( Increment, out incrs )
      & incrs := incrs.valid (self.old)
      & for all incr := incrs
        do
          loop
              RmFromList ( incr.ToEnvList.instance of LIST, incr )
          end
        end
      & RemoveIncrement ( incrs )
    end
end;
```

Abb. 6.15: Löschen der als ungültig markierten Knoten

Am Anfang einer Nachtransformation wird das alte Logbuch gelöscht und eine Wurzel für das neue mit `CreateTrafoUpdateRoot` angelegt.

```
transaction Update_Report( out trafo_update_root_out : TRAFO_UPDATE_ROOT)
    [0:1] =
    use trafo_update_roots : TRAFO_UPDATE_ROOT [0:n];
        trafo_updates : TRAFO_UPDATE [0:n];
        param_updates : PARAM_UPDATE [0:n]
    do
        IncrGet ( TRAFO_UPDATE_ROOT, out trafo_update_roots )
      & for all trafo_update_root := trafo_update_roots
        do
            trafo_updates := trafo_update_root.
                             lc_item.instance of TRAFO_UPDATE
          & for all trafo_update := trafo_updates
            do
                param_updates := trafo_update.<-In_Update-.
                                 instance of PARAM_UPDATE
              & for all param_update := param_updates
                do
                    RemoveIncrement ( param_update )
                end
```

```
              & RmFromList ( trafo_update_root, trafo_update )
              & RemoveIncrement ( trafo_update )
            end
          & RemoveIncrement ( trafo_update_root )
        end
      & CreateTrafoUpdateRoot ( out trafo_update_root_out )
    end
end;
```

Abb. 6.16: Logbuch initialisieren

```
transaction Trafo_Update_Report( trafo : Trafo) [0:1] =
    use trafo_update_roots : TRAFO_UPDATE_ROOT [0:n];
        trafos : Trafo [0:n];
        trafo_update_list : TRAFO_UPDATE_LIST_T;
        trafo_update : TRAFO_UPDATE
   do
        IncrGet ( TRAFO_UPDATE_ROOT_T, out trafo_update_roots )
      & for all trafo_update_root := trafo_update_roots
        do
            trafos := trafo.next
          & trafos := trafos.valid (not (self.Fail))
          & choose
            when not empty ( trafos )
            then
                CreateTrafoUpdateList ( out trafo_update_list )
              & trafo_update := (trafo.ToUpdateTrafo) : TRAFO_UPDATE [0:1]
              & AppendLast ( trafo_update, trafo_update_list )
              & for all traf := trafos
                do
                  choose
                  when (traf.Fail)
                  then
                      skip
                  else
                      CreateTrafoUpdateWithParamUpdate
                      ( "\tPrevious transformation failed so can not redo : ",
                        traf, trafo_update_list )
                    & Trafo_Update_Report ( traf )
                  end
                end
            else
                skip
            end
        end
    end
end;
```

Abb. 6.16.1: Das Logbuch aufbauen

Wenn eine Transformation nicht erneut ausgeführt werden kann, wird für sie ein TRAFO_UPDATE Knoten angelegt (siehe Abb. 6.13). Ausgehend von den gescheiterten Transformationen wird für die nachfolgenden eine Liste an den TRAFO_UPDATE Knoten angehängt, in die sie einsortiert werden. Dieser Aufruf geschieht wieder rekursiv analog zur Nachtransformation (siehe Trafo_Update_Report).

```
transaction CreateTrafoUpdateWithParamUpdate( str : string ;
                    trafo : Trafo ; trafo_update_root : TRAFO_UPDATE_ROOT)
    [0:1] =
    use param_updates : PARAM_UPDATE [0:n];
        params : Param [0:n];
        trafo_update : TRAFO_UPDATE;
        STR : STRING
    do
        CreateTrafoUpdate ( trafo, out trafo_update )
      & trafo.Fail := true
      & trafo_update.ART := (str & "\n\t" & trafo.art)
      & params := trafo.<-In-
      & for all param := params
        do
            CreateParamUpdate ( param, trafo_update )
          & param.old := true
          & choose
            when not empty ( param.<-Is-.instance of STRING )
            then
                STR := (param.<-Is-) : STRING [1:1]
               & STR.old := true
            else
                skip
            end
        end
      & AppendLast ( trafo_update_root, trafo_update )
    end
end;
```

Abb. 6.16.2: Logbuchmeldung in das Logbuch eintragen

Die Logbuchmeldung wird an `CreateTrafoUpdateWithParamUpdate` übergeben und dort mit der Art der Transformation und ihren Eingabeparametern vervollständigt. Zusätzlich werden die Parameter-Knoten und die STRING-Knoten als ungültig markiert. Da die STRING-Knoten bei der Vorwärtsmarkierung nicht erreicht werden, müssen sie für *nicht* erneut ausgeführte Transformationen als ungültig markiert werden.

Wenn der Benutzer das Logbuch sehen will, werden die `TRAFO_UPDATE` Knoten durchlaufen und ihr Inhalt in eine Textdatei geschrieben.

Überprüfung der Invarianten

Die Invarianten werden am Ende der Konsistenzerhaltung geprüft. Zur Zeit gibt es nur zwei Invarianten in VARLET. Sie werden überprüft und die entsprechende Fehlermeldung wird zurückgeliefert (siehe `Invariants`).

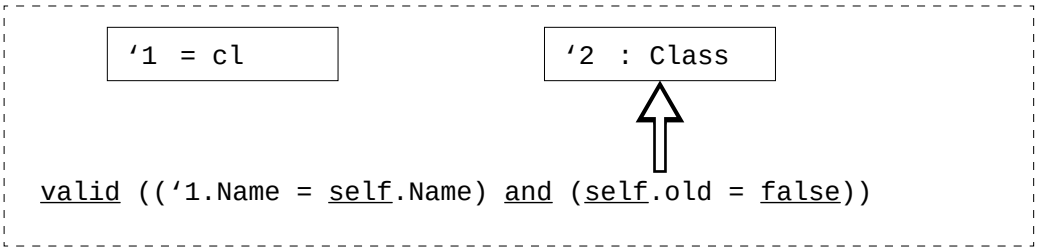
```
transaction Invariants( out ERR_STR1, ERR_STR2 : string) [0:1] =
    use classes, visited_classes : Class [0:n];
        visited_class : Class;
        errstr1, errstr2 : string := ""
    do
        IncrGet ( Class, out classes )
      & classes := classes.valid ((self.old = false) and (self.Name # ""))
      & for all cl := classes
        do
          choose
          when (cl = (cl and visited_classes))
          then
              skip
          else
            choose
            when ExistanceOfTwoClassesWithTheSameName ( cl )
```

```
            then
                errstr1 := (errstr1
                            & "There are two classes with the name : "
                            & cl.Name & "\n")
              & ExistanceOfTwoClassesWithTheSameName_ ( cl,
                                                        out visited_class )
              & visited_classes := (visited_classes or visited_class)
            else
                skip
            end
          end
        & choose
          when ExistanceOfTwoClassPropretiesWithTheSameName ( cl )
          then
          errstr2 := (errstr2
                      & "There are two class propreties with same name
                      in class : " & cl.Name & "\n"                    )
          else
              skip
          end
        end
      & ERR_STR1 := errstr1
      & ERR_STR2 := errstr2
    end
end;
```

Abb. 6.17: Überprüfung der Invarianten

`ExistanceOfTwoClassesWithTheSameName_` ist genauso implementiert wie `ExistanceOfTwoClassesWithTheSameName` mit dem einzigen Unterschied, daß die Klassen, die den gleichen Namen haben, zurückgeliefert werden. Falls beispielsweise zwei Klassen den gleichen Name haben, soll die Logbuchmeldung nicht doppelt angezeigt werden.

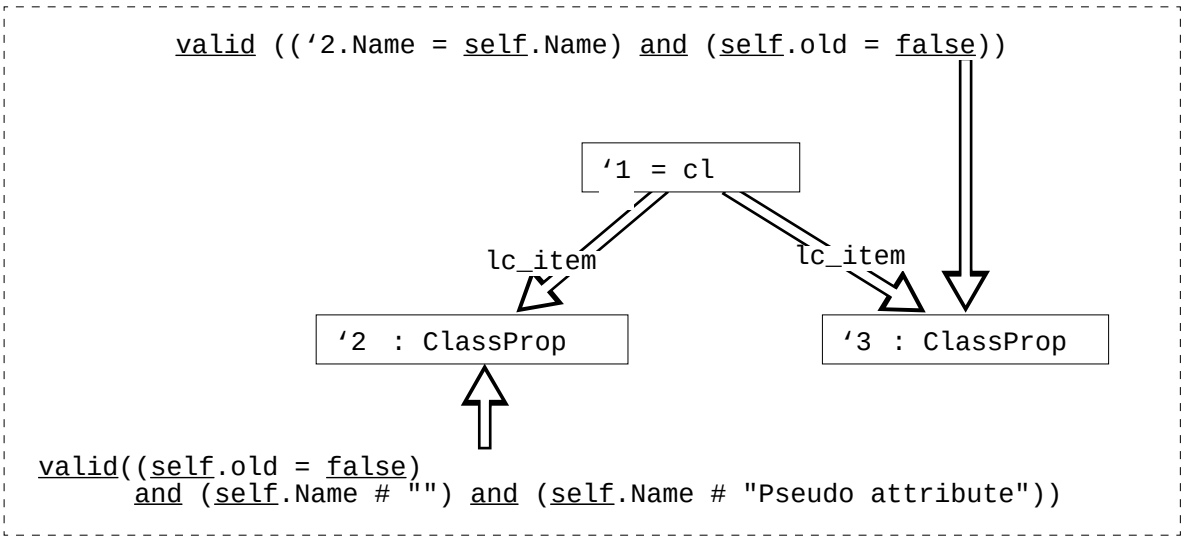
```
test ExistanceOfTwoClassesWithTheSameName( cl : Class)
                                  (* The class is only passed for speed-up *)
        [0:1] =
```




```
      end;
```

Abb. 6.17.1: Invariante für Klassen

```
test ExistanceOfTwoClassPropretiesWithTheSameName( cl : Class)
                                    (* The class is only passed for speed-up *)
        [0:1] =
```




```
    end;
```

Abb. 6.17.1: Invariante für Klasseneigenschaften

Kapitel 7

Zusammenfassung und Ausblick

Neue Anwendungsbereiche stellen neue Anforderungen an die aktuellen Informationssysteme. Die weit verbreiteten relationalen Datenbanksysteme sind ungeeignet zur effizienten Verwaltung komplexer Datenstrukturen. In der Praxis stellt sich die Frage, wie ein schrittweiser, kostensparender Übergang von den vorhandenen relationalen zu den neuen objektorientierten Systemen vollzogen werden kann. Hierzu werden insbesondere Konzepte und Techniken benötigt, mit deren Hilfe relationale Systeme in objektorientierte überführt (migriert) bzw. integriert werden können.

Die erste Phase (Analyse-Phase) in einem solchen Migrationsprozeß ist die semantische Analyse der relationalen Datenbankanwendung, die sogenannte Schemaanalyse. In der zweiten Phase (Migrations-Phase) wird das semantisch angereicherte relationale Schema, durch geeignete Schematransformationen, in ein äquivalentes objektorientierte Schema überführt. Diese Schematransformationen werden teilweise automatisch und teilweise benutzergesteuert ausgeführt. Die automatischen Schematransformationen (initiale Abbildung) übersetzen das relationale Schema in ein objektorientiertes, damit der Benutzer eine objektorientierte Sicht des relationalen Schemas bekommt. Die verwendeten Tripelregeln sichern die Äquivalenz zwischen den Schemata. Der Benutzer kann das nun vorhandene objektorientierte Schema weiter transformieren (Restrukturierung).

Die - durch die Schemamigration erstellte - konzeptionelle Sicht auf das untersuchte Datenbankschema liefert meist neue Hinweise auf weitere Analyseinformationen. Ein iteratives Durchlaufen der Analyse- und Migrations-Phase ist die Regel. Eine Werkzeugunterstützung für die Migrations-Phase muß daher einen verzahnten Ablauf von Schemaanalyse und Schemamigration ermöglichen. Hierbei stellt sich das Problem der Konsistenzerhaltung zwischen dem analysierten relationalen und dem transformierten objektorientierten Schema. In dieser Arbeit wurde ein neues Konzept für die Konsistenzerhaltung vorgestellt.

Die Realisierung des Migrationsprozesses durch miteinander verzahnt ablaufende Restrukturierungs- und Analyseaktivitäten erfordert einen komplexen Mechanismus. Bei jeder Änderung der Analyseinformation des relationalen Schemas müssen alle Migrationsschritte rückgängig gemacht werden, die nun nicht mehr gültig sind (selektives Undo). Das heißt, die Schematransformationen, die auf den veränderten relationalen Schemakomponenten nicht mehr möglich sind, dürfen nicht weiter bestehen. Außerdem müssen die Schematransformationen, die nunmehr möglich geworden sind, durchführbar sein.

Dafür wurden die (initialen) Abbildung und die einzelnen Transformationen im Abhängigkeitsgraphen verbunden. Es hat sich herausgestellt, daß zusätzliche Abbildungsinformationen gespeichert werden müssen. Außerdem müssen die Parameter der Restrukturierungstransformationen für eine spätere eindeutige Zuordnung verwaltet werden. Durch geeignete Einschrän-

kungen bei den Transformationen und ihren Vorbedingungen kann die Konsistenzerhaltung mit einem vertretbaren Aufwand durchgeführt werden. Lösungen wurden vorgeschlagen mit der Zusammenfassung von Transformationen bzw. den Invarianten.

Die Möglichkeit, Transformationen in Makros zusammenzufassen, besteht noch nicht. Die vorhandenen Restrukturierungstransformationen sind noch in der Entwicklung und müssen erweitert bzw. vervollständigt werden. Aber neue Transformationen oder Erweiterungen sind leicht in das vorhandene Rahmenwerk integrierbar. Somit kann das Rahmenwerk noch lange benutzt und für verschiedenen Anwendungen eingesetzt werden. Darüber hinaus läßt sich der vorgestellte Mechanismus zur Konsistenzerhaltung auf ähnliche Probleme übertragen.

Im Rahmen der Projektgruppe VARLET wurde die Werkzeugunterstützung für den Migrationsprozeß implementiert. Der iterative, explorative Aspekt wurde dort noch nicht berücksichtigt. In dieser Diplomarbeit wurden die Konzepte zur Konsistenzerhaltung zwischen den Schemata im VARLET-Prototypen implementiert und getestet. Gegenüber anderen Ansätzen ist, mit der (initialen) Abbildung und den zur Verfügung gestellten Transformationen, ein iterativer Migrationsprozeß möglich.

Weitere Diplomarbeiten befassen sich mit der Schemaanalyse und -anreicherung, der Untersuchung der Restrukturierungstransformationen und der Datenmigration bzw. Datenkonvertierung. Die Ergebnisse dieser Arbeiten sollen im weiteren Verlauf in VARLET zusammengeführt werden. Die Arbeitsgruppe Softwaretechnik plant die Portierung von VARLET auf Linux, die durch die PROGRES-Version 9.0 ermöglicht wird.

Literatur

[ACM94] R. Abu-Hamdeh, J. Cordy and P. Martin. Schema Translation Using Structural Transformation. In *Proc. of CASCON 1994, Toronto,* pages 202-215. November 1994

[And94] M. Andersson. Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering. In *Proc. of the 13th Int. Conference of the Entity Relationship Approach, Manchester*, pages 403–419. Springer, 1994.

[BGD97] A. Behm, A. Geppert and K. R. Dittrich. On the Migration of Relational Schemas and Data to Object-Oriented Database Systems. In *Proc. 5th International Conference on Re-Technologies for Information Systems, Klagenfurt, Austria*. December 1997.

[BP96] M. Blaha and W. Premerlani. A Catalog of Object Model Transformation. Presented at *3rd Working Conference on Reverse Engineering, Monterey, California*. November 1996

[CC90] Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: {A} Taxonomy. In *journal IEEE Software, volume 7, number 1*, pages 13-17. January 1990.

[DA87] K. H. Davis and A. K. Arora. Converting a Relational Database Model into an Entity-Relationship Model. In *Proc. of the 6th Int. Conference of the Entity Relationship Approach, New York,* pages 271–285. North-Holland, November 1987.

[Fong97] J. Fong. Converting Relational to Object-Oriented Databases. In *SIGMOD Record*, Vol. 26, No. 1, March 1997.

[FV95] C. Fahrner and G. Vossen. Transforming Relational Database Schemas into Object-Oriented Schemas according to ODMG-93. In *Proc. of the 4th Int. Conf. of on Deductive and Object-Oriented Databases 1995*, 1995.

[Hol97] J. Holle. Ein Generator für integrierte Werkzeuge am Beispiel der object-relationalen Datenbankschemamigration. Diplomarbeit an der Universität-Gesamthochschule Paderborn, Juli 1997.

[HTJC93] J-L. Hainaut, C. Tonneau, M. Joris and M. Chandelon. Schema transformation techniques for database reverse engineering. In *Proc. of the 12th Int. Conference of the Entity-Relationship Approach, Arligton-Dallas*, December 1993.

[JH98] J.H. Jahnke and M. Heitbreder. Design Recovery of Legacy Database Applications based on Possibilistic Reasoning. In *Proc. of 7th IEEE Int. Conf. of Fuzzy Systems (FUZZ'98). Anchorage, USA.* IEEE Computer Society, May 1998.

[JJ94] M.A. Jeusfeld and U.A. Johnen. An Executable Meta Model for Re-Engineering of Database Schemas. In *Proc. of the 13th Int. Conference of the Entity-Relationship Approach, Manchester*, December 1994.

[JK90] P. Johannesson and K. Kalman. A method for translating relational schemas into conceptual schemas. In F. H. Lochovsky, editor, *Entity-Relationship Approach to Database Design and Querying*. ERI, 1990.

[JSZ96] J. Jahnke, W. Schäfer and A. Zündorf. A Design Environment for Migrating Relational to Object Oriented Database Systems. In *Proc. of the 1996 Int Conference on Software Maintenance (ISCM'96)*. IEE Computer Society, 1996.

[JSZ97] J. Jahnke, W. Schäfer and A. Zündorf. Generic Fuzzy Reasoning Nets as a Basis for Reverse Engineering Relational Database Applications. In *Proc. of European Software Engineering Conference (ESEC/FSE)*, number 1302 in LNCS. Springer, September 1997.

[JZ98] J. Jahnke and A. Zündorf. Using Graph Grammars for Building the Varlet Database Reverse Engineering Environment. Accepted *for 1998 Intl. Workshop on Theory and Applications of Graph Grammars, Paderborn, Germany*. November 1998.

[Lef95] M. Lefering. Integrationswerkzeuge in einer Softwareentwicklungsumgebung. Informatik, Verlag Shaker, 1995.

[MM93] V. M. Markowitz and J. A. Makowsky. Identifying Extended Entity-Relationship Object Structures in Relational Schemas. *IEEE Transactions on Software Engineering*, 16(8):777–790, August 1990.

[ODMG97] R.G.G. Cattell et al. Object Database Standard: ODMG 2.0, Morgan Kaufmann Publishers, Inc. 1997. ISBN 1-55860-463-4.

[OMT] J. Rumbaugh, M. Blaha, W. Premerlani , F. Eddy and W. Lorensen. Objekt-oriented Modeling and Design, Prentice Hall , 1991. ISBN 0-13-629841-9.

[ONTOS] ONTOS, Inc. ONTOS Object Integration Server for Relational Databases 2.0, Schema Mapper User's Guide. OIS - 20-SUN-SMUG-1.0, July 1996.

[PB94] W. J. Premerlani and M. R. Blaha. An Approach for Reverse Engineering of Relational Databases. *Communications of the ACM*, 37(5):42–49, May 1994.

[RH96] S. Ramanathan and J. Hodges. Reverse Engineering Relational Schemas to Object-Oriented Schemas. Techical Report No. MSU-960701, July 1, 1996. Department of Computer Science, Mississippi State University.

[RH97] S. Ramanathan and J. Hodges. Extraction of Object-Oriented Structures from Existing Relational Databases. In *SIGMOD Record*, Vol. 26, No. 1, March 1997.

[Schä96] W. Schäfer. Skriptum zur Vorlesung „Datenbanken und Informationssysteme". Universität-GH Paderborn, Fachbereich Mathematik/Informatik, 1996.

[Schü94] A. Schürr. Specification of Graph Translators with Tripel Graph Grammars. In *G. Tinhofer (ed.): Proc. WG 94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science, Herrsching, Germany*, Juni 1994. LNCS 903, Berlin, Springer Verlag (1994), 151-163.

[SK90] F. N. Springsteel and C. Kou. Reverse Data Engineering of E-R Designed Relational Schemas. In *Proc. of Databases, Parallel Architectures and their Applications,* pages 438–440. Springer, March 1990.

[SLGC94] O. Signore, M. Loffredo, M. Gregori and M. Cima. Reconstruction of ER Schema from Database Applications: a Cognitive Approach. In *Proc. of 13th Int. Conference of ERA, Manchester,* pages 387–402. Springer, 1994.

[Vos94] G. Vossen. Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme. Addison-Wesley, 1994. ISBN 3-89319-566-1.

[Zün95] A. Zündorf. PROgramierte GRaphErsetzungsSysteme (Spezifikation, Implementierung und Anwendung einer integrierten Entwicklungsumgebung), Dissertation RWRH Aachen. Deutscher Universitätsverlag (1995)

Anhang A

Abbildungen mit Tripelgraphgrammatiken - Implementierung

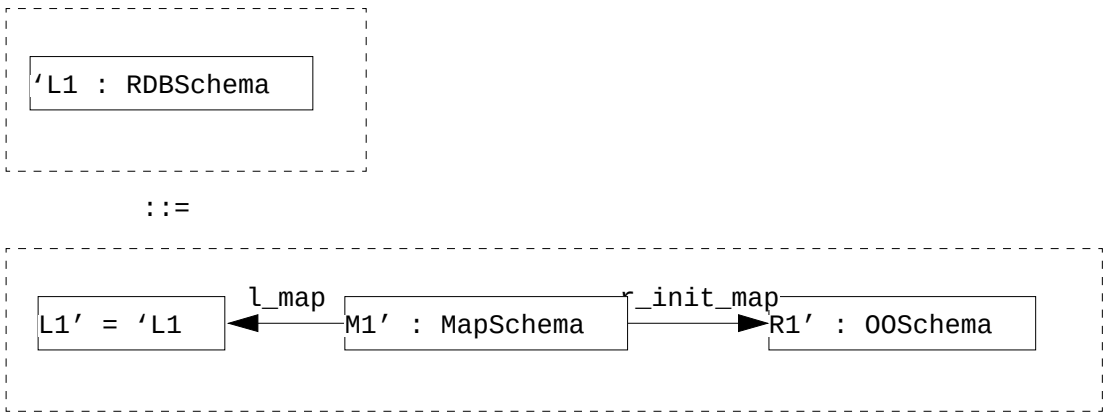
Zuerst wird die Implementierung der initiale Abbildung aufgeführt. Anschließend wird die Wiederholung der initialen Abbildung (`DefaultMapping`) präsentiert.

Initiale Abbildung

```
transaction DoInitialMapping( out ooschema : OOSchema ;
                        out Errstr : string ; out Errstr1 : string) [0:1] =
    use m3schema : OOSchema;
        relschema : RDBSchema;
        mapschema : MapSchema;
        act : integer;
        errstr, errstr1 : string
    do
        TestOnInheritanceForVars ( out errstr )
      & Map_Schema ( out relschema, out m3schema, out mapschema )
      & IncrementOOSchemaActuality ( out act )
      & Map_Relations ( mapschema, m3schema, act, out errstr1 )
      & Map_INDs ( mapschema, act )
      & ooschema := m3schema
      & Errstr := errstr
      & Errstr1 := errstr1
    end
end;
```

`TestOnInheritanceForVars` zeigt eine Fehlermeldung an, falls eine Relation mehr als eine Variante und eine IsA-Beziehung hat. Dies würde zur mehrfach Vererbung bzw. zur Vererbung einer Klasse an Ober- und Unterklasse führen.

```
production Map_Schema( out relschema : RDBSchema ;
              out m3schema : OOSchema ; out map_schema : MapSchema) [0:1] =
```



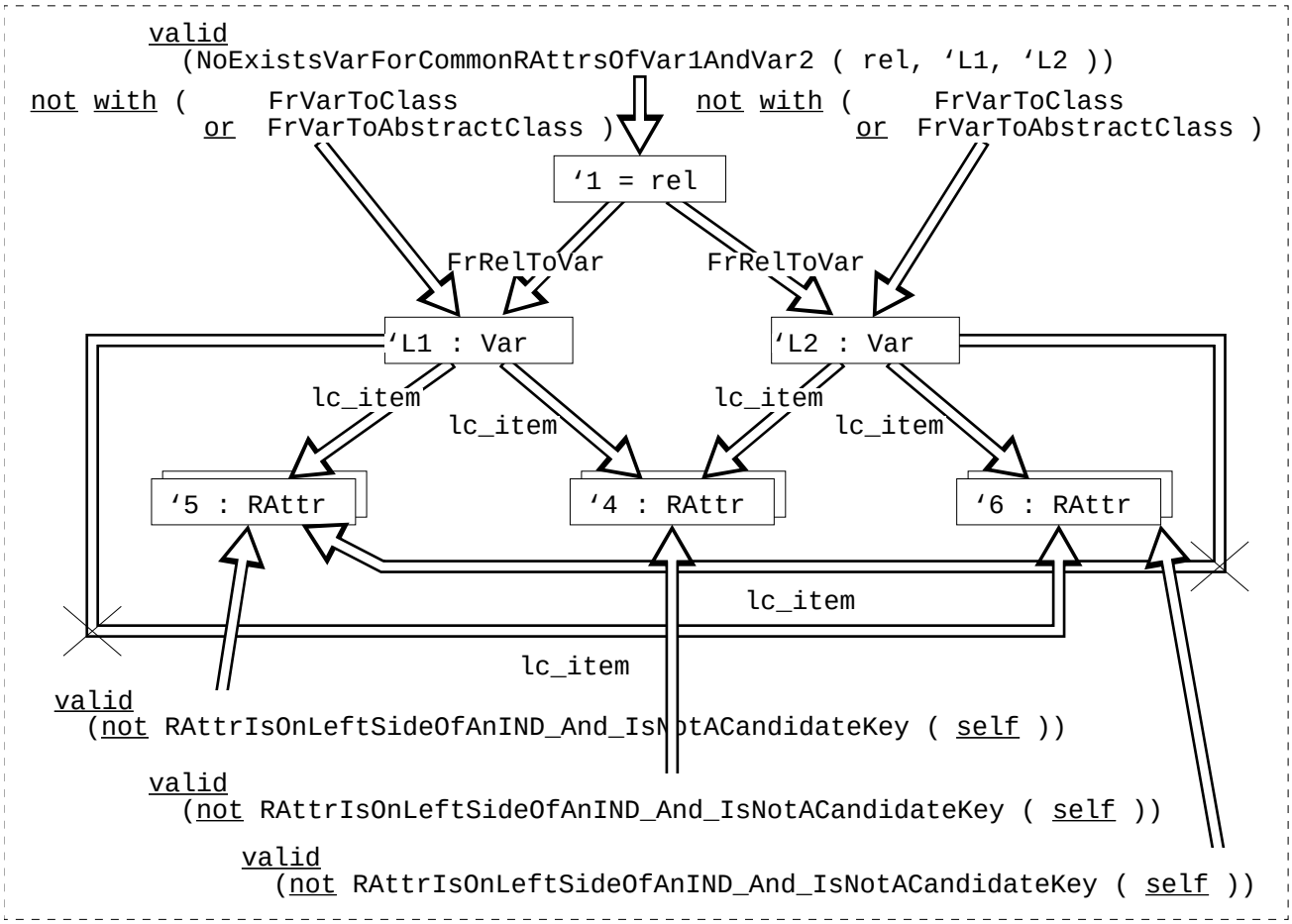

```
    transfer R1'.Name := 'L1.Name; R1'.MYVISIBLE := 1;
    return relschema := L1'; m3schema := R1'; map_schema := M1';
end;
```

`Map_Relations` bildet Relationen bzw. ihre Varianten auf Klassen ab. `Map_Relation` legt ein Abbildungsknoten für die Relation an. Für die Behandlung verschiedener Varianten werden für jede IND der Relation *IND-Attribute* angelegt, um die Abbildung einfach zu halten. `RmRAttrsForINDs` löscht die alle IND-Attribute, `AddRAttrsForINDs` fügt IND-Attribute für die aktuellen INDs hinzu. `Affix` zählt für jede Variante `affix_nr` hoch. `affix_nr` wird mit # an den Namen der Klassen angehängt, wenn es mehr als eine Variante gibt. Die entsprechenden Attribute und Schlüssel werden mit der jeweiligen Variante abgebildet.

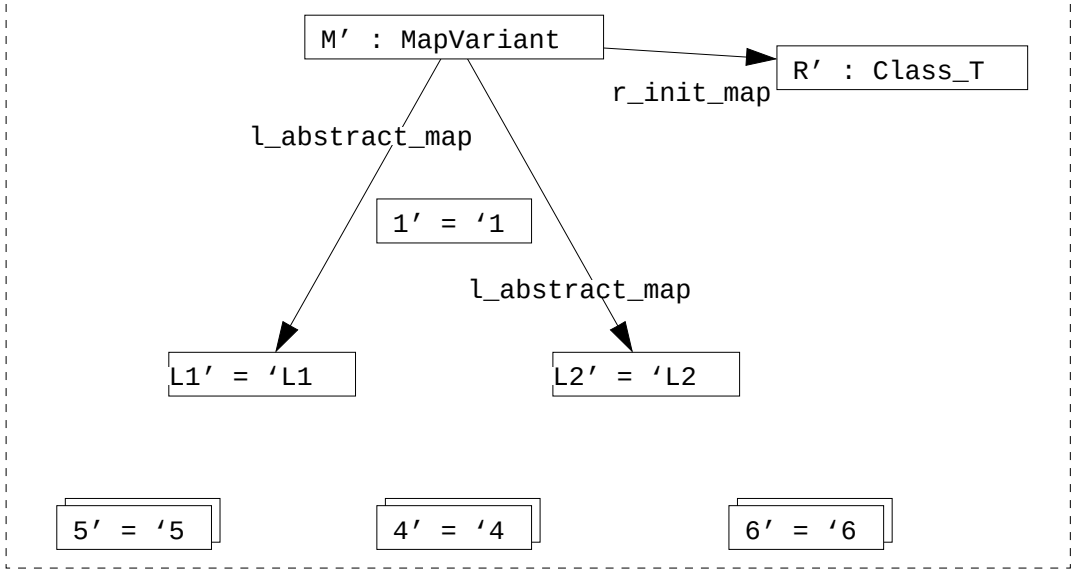
```
transaction Map_Relations( map_schema : MapSchema ; m3schema : OOSchema ;
                           act : integer ; out Errstr : string) [0:1] =
    use relations : Relation [0:n];
        rattrs : RAttr [0:n];
        var1, var2 : Var;
        map_v : MapVariant;
        map_r : MapRelation;
        cl : Class;
        inh : Inherit;
        affix_nr : integer;
        errstr : string := ""
    do
        IncrGet ( Relation, out relations )
      & for all r := relations.valid (self.to_be_mapped)
        do
            affix_nr := (- 1)
          & Map_Relation ( r, out map_r )
          & AppendLast ( map_schema, map_r )
          & RmRAttrsForINDs ( r )
          & AddRAttrsForINDs ( r )
              (* For each IND we add an attribute to keep mapping simple*)
          & loop
                choose
                    Map_VarToClassWhichIsRootOfHierarchy
                    ( r, out map_v, out cl, out rattrs )
                  & affix_nr := (affix_nr + 1)
                  & Affix ( cl, affix_nr )
                  & AppendLast ( map_r, map_v )
                  & AppendLast ( m3schema, cl )
                  & cl.actuality_ := act
                  & Map_RAttrs ( map_v, cl, rattrs )
                  & Map_CurrentKey ( map_v )
                else
                    Map_2VarsToAbstractClassWhichIsRootOfHierarchy
                    ( r, out map_v, out cl, out rattrs, out var1, out var2 )
                  & affix_nr := (affix_nr + 1)
                  & Affix ( cl, affix_nr )
                  & AppendLast ( map_r, map_v )
                  & AppendLast ( m3schema, cl )
                  & cl.actuality_ := act
                  & Map_RAttrs ( map_v, cl, rattrs )
                  & Map_CurrentKey ( map_v )
                  & rattrs := ((var1.lc_item.instance of RAttr)
                               and (var2.lc_item.instance of RAttr))
                  & loop
                        Map_VarToExistingAbstractClass ( r, rattrs )
                      & affix_nr := (affix_nr + 1)
                      & Affix ( cl, affix_nr )
                    end
                else
                    Map_VarToSubClassOfAbstractClass
                    ( r, out map_v, out cl, out rattrs, out inh )
                  & affix_nr := (affix_nr + 1)
                  & Affix ( cl, affix_nr )
                  & AppendLast ( map_r, map_v )
                  & AppendLast ( m3schema, cl )
                  & cl.actuality_ := act
```

```
                & inh.actuality_ := act
                & Map_RAttrs ( map_v, cl, rattrs )
                & Map_CurrentKey ( map_v )
              else
                  Map_VarToSubClassOfClass
                  ( r, out map_v, out cl, out rattrs, out inh )
                & affix_nr := (affix_nr + 1)
                & Affix ( cl, affix_nr )
                & AppendLast ( map_r, map_v )
                & AppendLast ( m3schema, cl )
                & cl.actuality_ := act
                & inh.actuality_ := act
                & Map_RAttrs ( map_v, cl, rattrs )
                & Map_CurrentKey ( map_v )
              else
                  Map_2VarsToAbstractSubClass
                  ( r, out map_v, out cl, out rattrs, out var1, out var2,
                    out inh )
                & affix_nr := (affix_nr + 1)
                & Affix ( cl, affix_nr )
                & AppendLast ( map_r, map_v )
                & AppendLast ( m3schema, cl )
                & cl.actuality_ := act
                & inh.actuality_ := act
                & Map_RAttrs ( map_v, cl, rattrs )
                & Map_CurrentKey ( map_v )
                & rattrs := ((var1.lc_item.instance of RAttr)
                             and (var2.lc_item.instance of RAttr))
                & loop
                      Map_VarToExistingAbstractClass ( r, rattrs )
                    & affix_nr := (affix_nr + 1)
                    & Affix ( cl, affix_nr )
                  end
              end
          end
        & choose
            when not (card ( r.=FrRelToVar=> ) =
                      card ( r.=FrRelToMapVariant=> ) )
            then
                errstr := (errstr & " " & r.Name)
              & RemoveIncrement ( map_r )
              & r.to_be_killed := false
            else
                errstr := errstr
              & r.to_be_mapped := false
              & r.to_be_killed := false
            end
          end
        & Errstr := errstr
    end
end;
```

```
production Map_2VarsToAbstractClassWhichIsRootOfHierarchy(
                  rel : Relation ; out map_var : MapVariant ; out c : Class;
                  out rattrs : RAttr [0:n] ; out var1, var2 : Var) [0:1] =
```

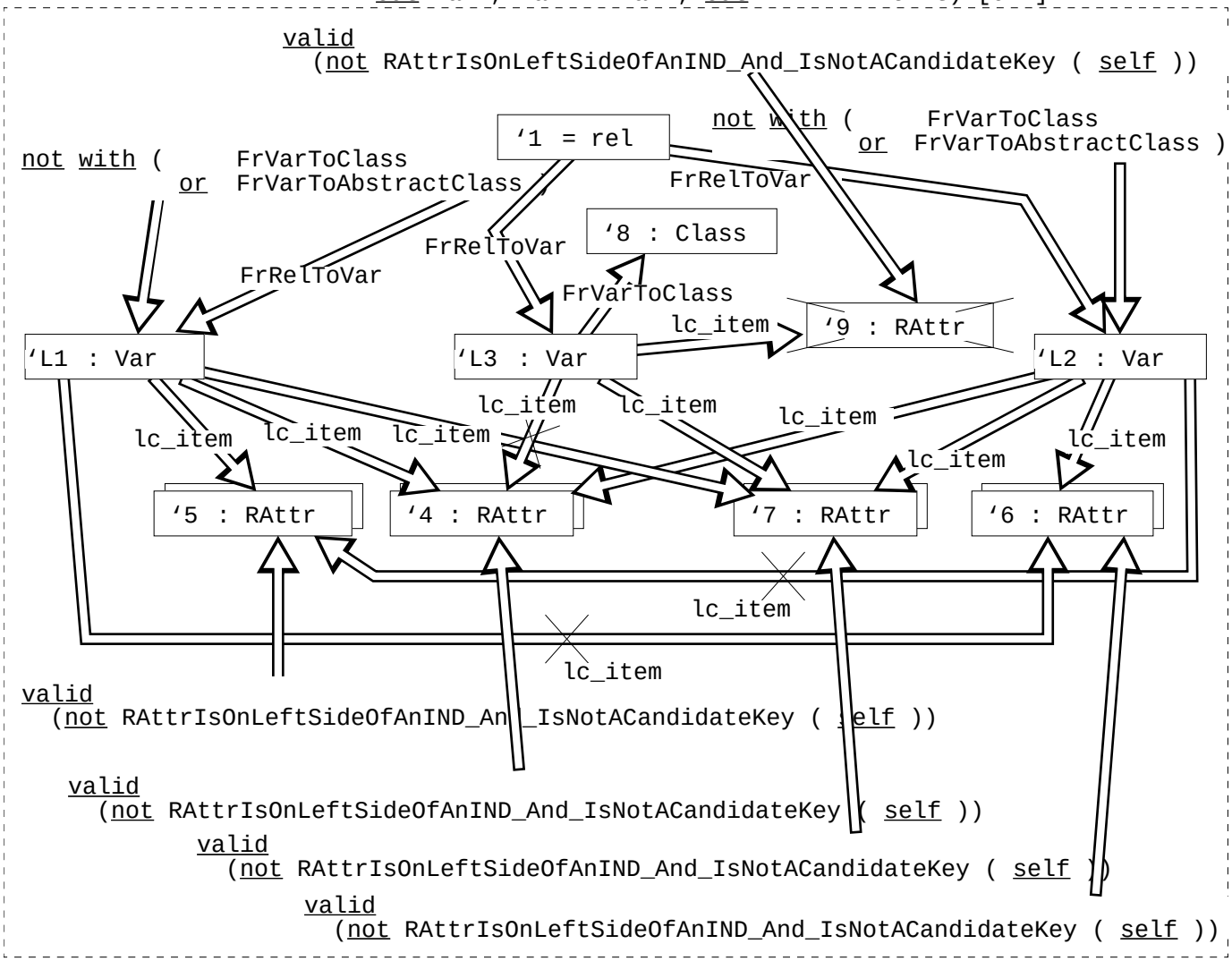


::=



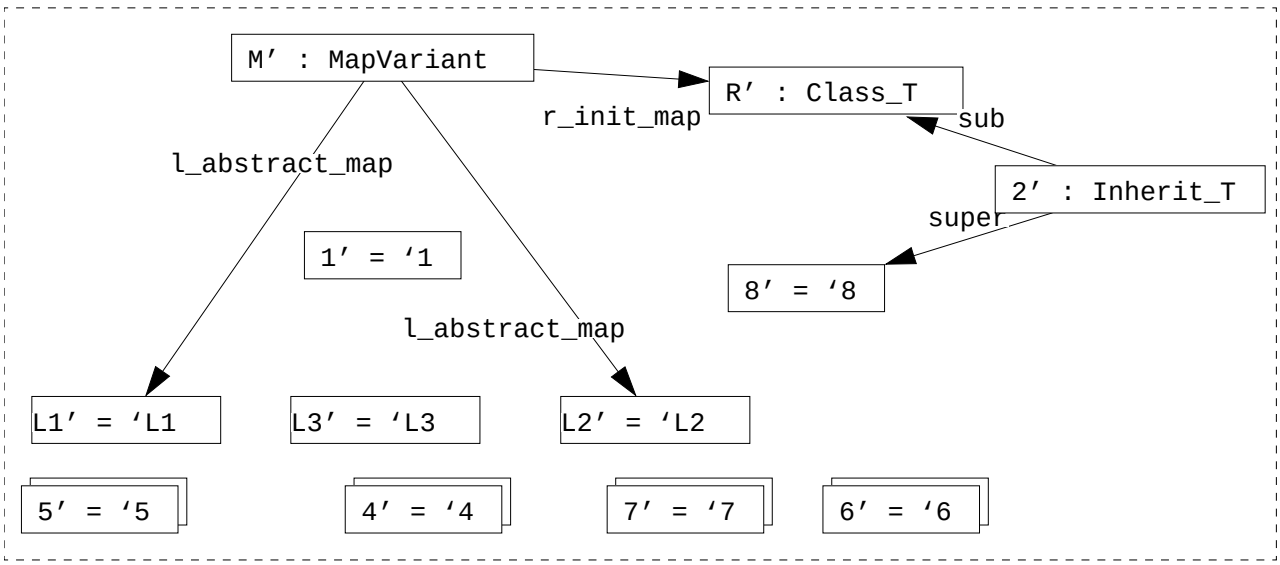

```
   condition (MaxNoOfCommonRAttrsOfVarPairs ( '1 ) =
              NoOfCommonRAttrsFor2Vars ( 'L1, 'L2 ));
   transfer R'.Name := '1.Name; R'.MYVISIBLE := 1;
            R'.abstract := true; R'.old := false; M'.old := false;
   return map_var := M'; c := R'; rattrs := '4; var1 := 'L1; var2 := 'L2;
end;
```

```
production Map_2VarsToAbstractSubClass( rel : Relation ;
      out map_var : MapVariant ; out c : Class ; out rattrs : RAttr [0:n] ;
                    out var1, var2 : Var ; out inh : Inherit) [0:1] =
```




```
valid
  (not RAttrIsOnLeftSideOfAnIND_And_IsNotACandidateKey ( self ))
valid
  (not RAttrIsOnLeftSideOfAnIND_And_IsNotACandidateKey ( self ))
valid
  (not RAttrIsOnLeftSideOfAnIND_And_IsNotACandidateKey ( self ))
valid
  (not RAttrIsOnLeftSideOfAnIND_And_IsNotACandidateKey ( self ))
```

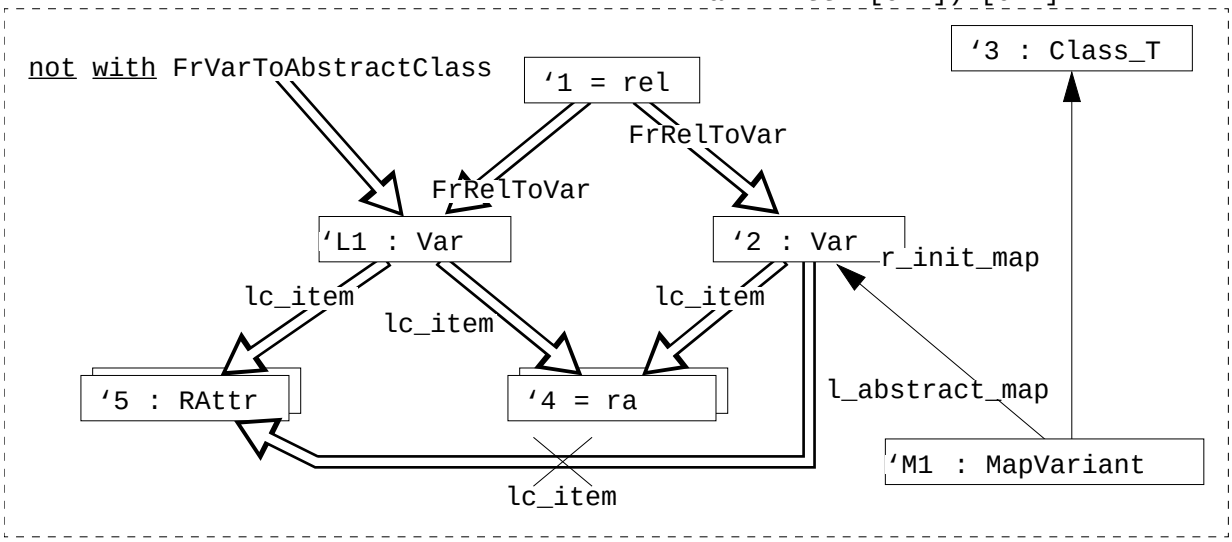
::=



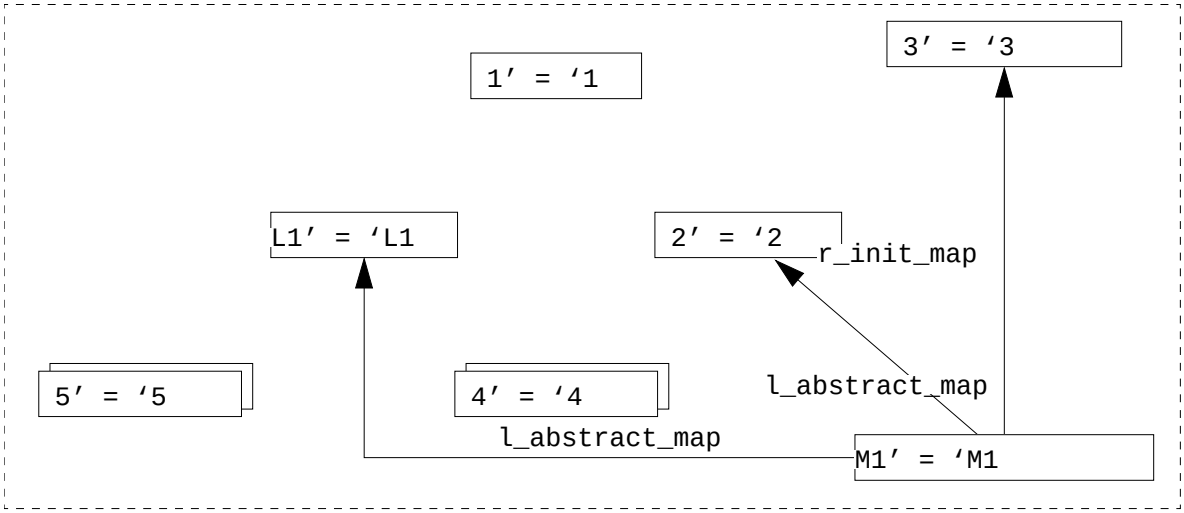

```
condition (MaxNoOfCommonRAttrsOfVarPairs ( '1 )
                = NoOfCommonRAttrsFor2Vars ( 'L1, 'L2 ));
      not (MaxNoOfCommonRAttrsOf2FixedAnd1VariableVar ( '1, 'L1, 'L2 )
           > NoOfCommonRAttrsFor3Vars ( 'L1, 'L2, 'L3 ) );
```

```
    transfer R'.Name := '1.Name; R'.MYVISIBLE := 1; R'.abstract := true;
             R'.old := false; M'.old := false; 2'.old := false;
    return map_var := M'; c := R'; var1 := 'L1; var2 := 'L2;
           rattrs := '4; inh := 2';
end;
```

```
production Map_VarToExistingAbstractClass( rel : Relation ;
                                           ra : RAttr [0:n]) [0:1] =
```

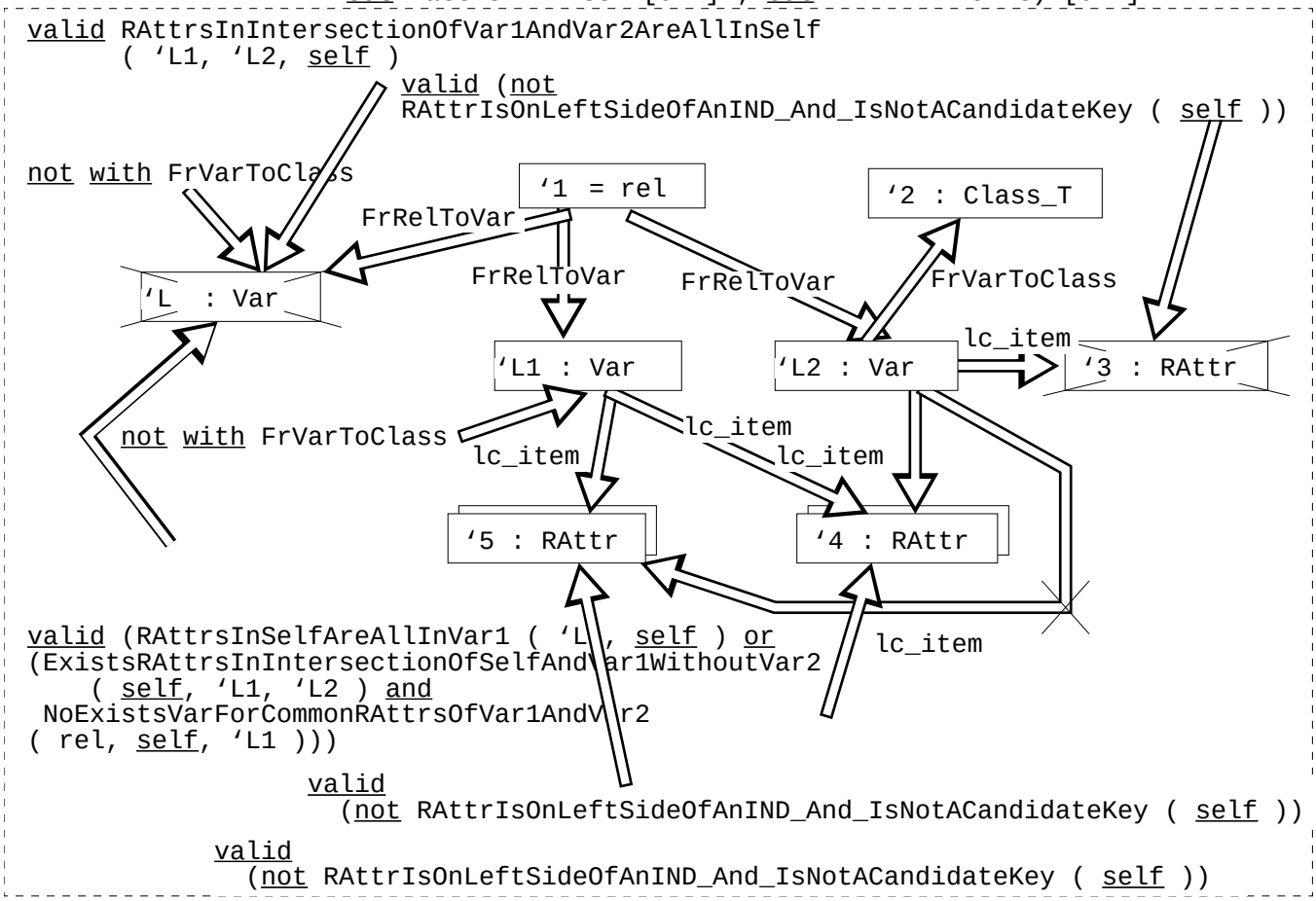



```
        ::=
```

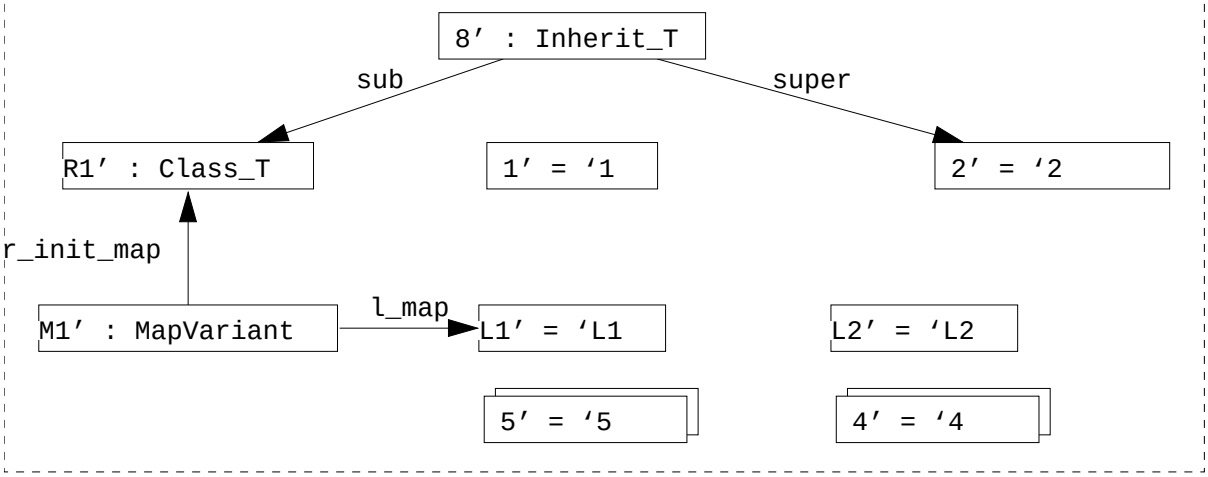



```
    condition (MaxNoOfCommonRAttrsOfAbstractVarPair ( '1 )
              = NoOfCommonRAttrsFor2Vars ( 'L1, '2 )     );
              '3.abstract = true;
              ('M1.old = false) and ('3.old = false);
end;
```

```
production Map_VarToSubClassOfClass( rel : Relation ;
                    out map_var : MapVariant ; out c : Class ;
                    out rattrs : RAttr [0:n] ; out inh : Inherit) [0:1] =
```

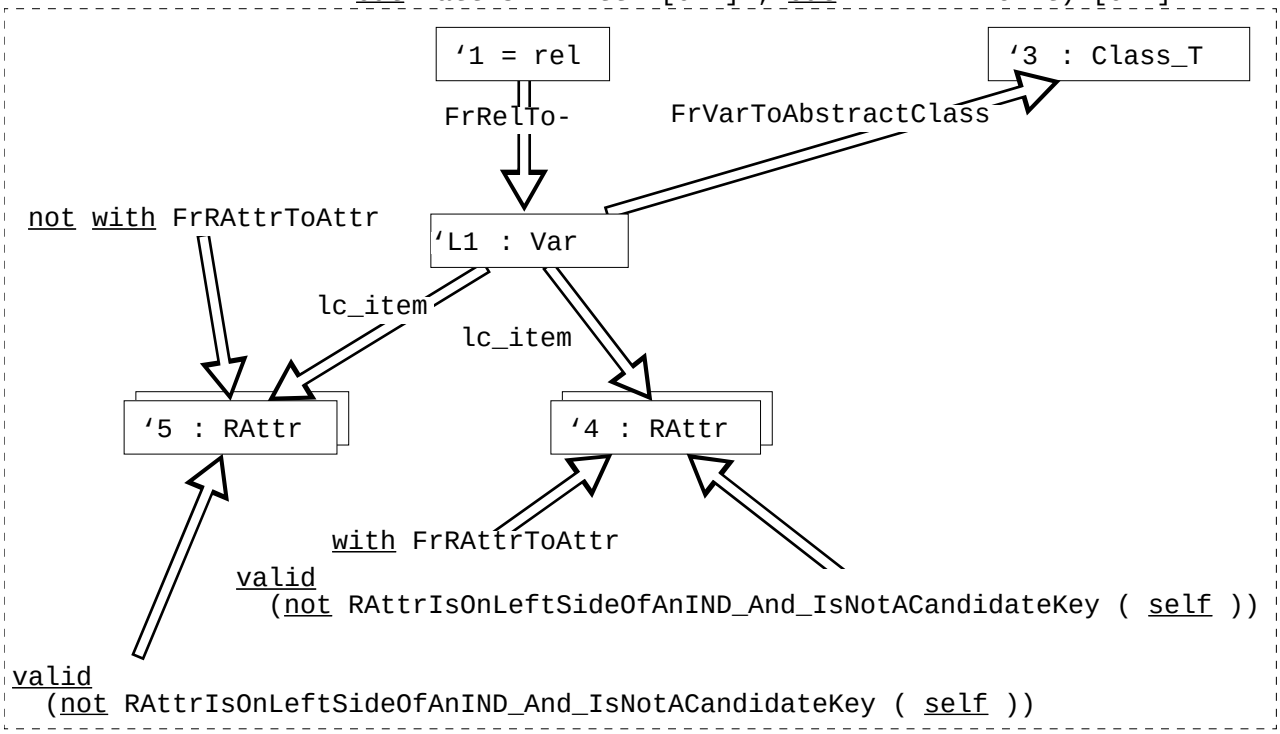



```
        ::=
```

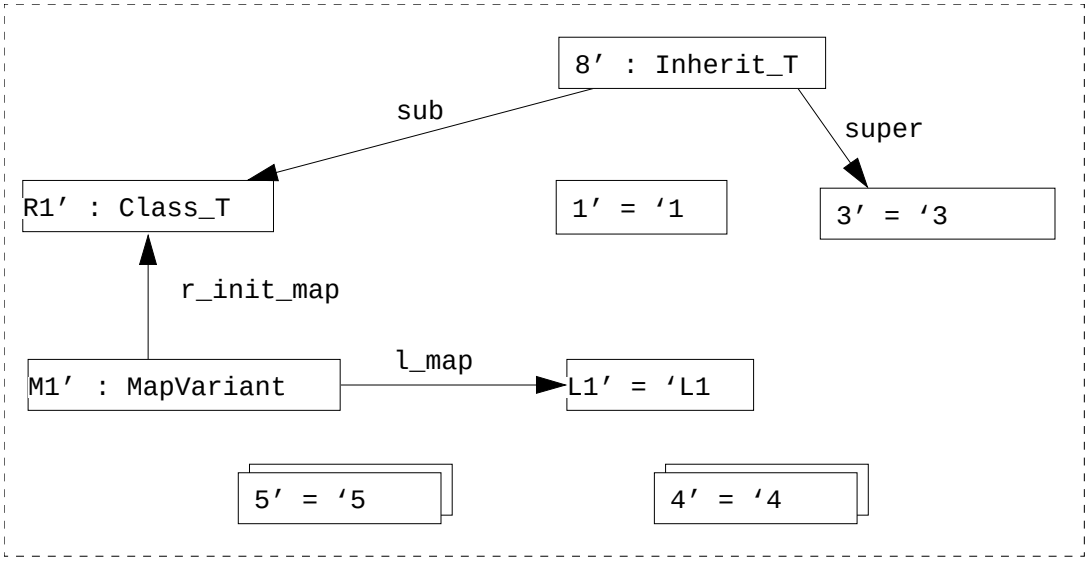



```
    condition (MaxNoOfCommonRAttrsOfVarPair ( '1 )
              = NoOfCommonRAttrsFor2Vars ( 'L1, 'L2 ));
              '2.abstract = false;
    transfer R1'.Name := '1.Name; R1'.MYVISIBLE := 1;
             R1'.old := false; M1'.old := false; 8'.old := false;
    return map_var := M1'; c := R1'; rattrs := 5'; inh := 8';
end;
```

```
production Map_VarToSubClassOfAbstractClass( rel : Relation ;
                      out map_var : MapVariant ; out c : Class ;
                      out rattrs : RAttr [0:n] ; out inh : Inherit) [0:1] =
```

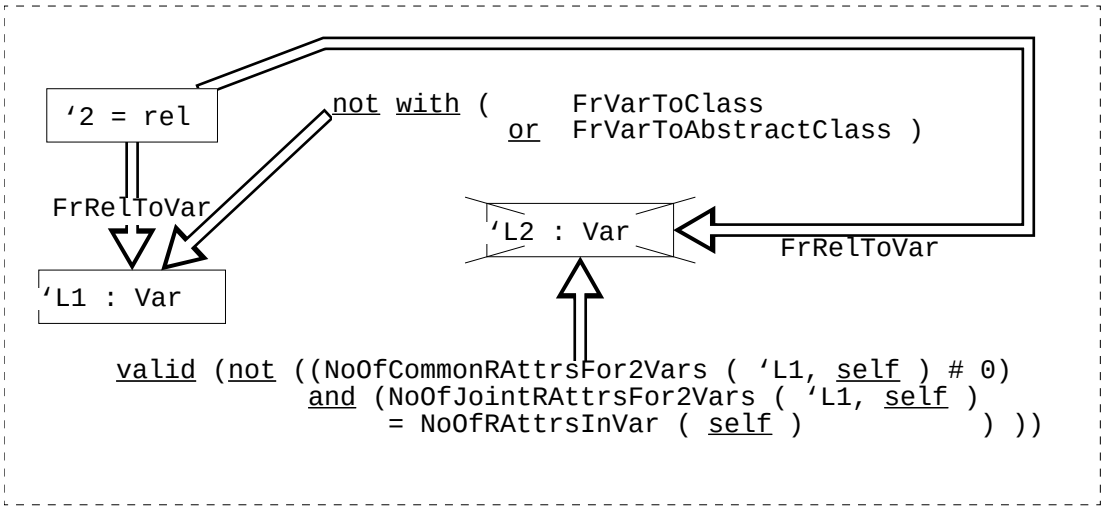



```
        ::=
```

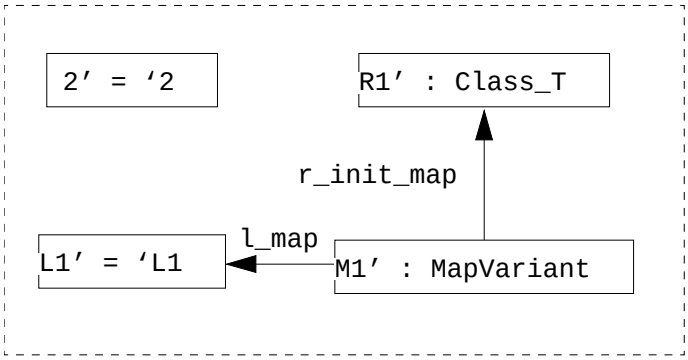



```
    condition ‘3.abstract = true;
    transfer R1’.Name := ‘1.Name; R1’.MYVISIBLE := 1; R1’.old := false;
             M1’.old := false; 8’.old := false;
    return map_var := M1’; c := R1’; rattrs := 5’; inh := 8’;
end;
```

```
production Map_VarToClassWhichIsRootOfHierarchy( rel : Relation ;
                                out map_var : MapVariant ; out c : Class ;
                                out rattrs : RAttr [0:n]) [0:1] =
```

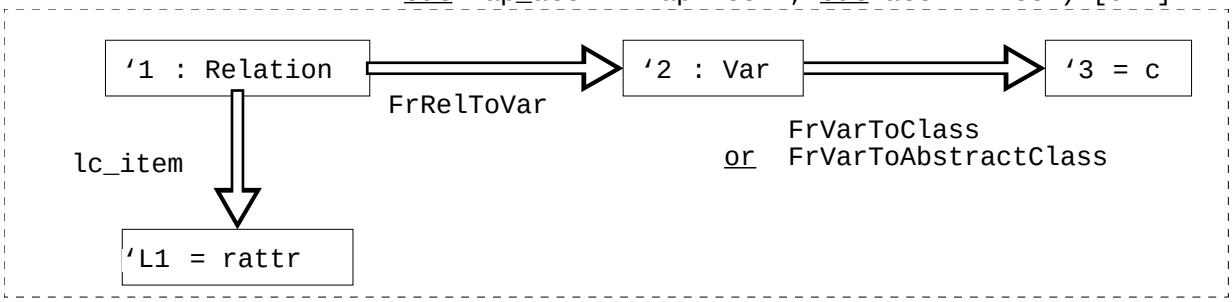


::=

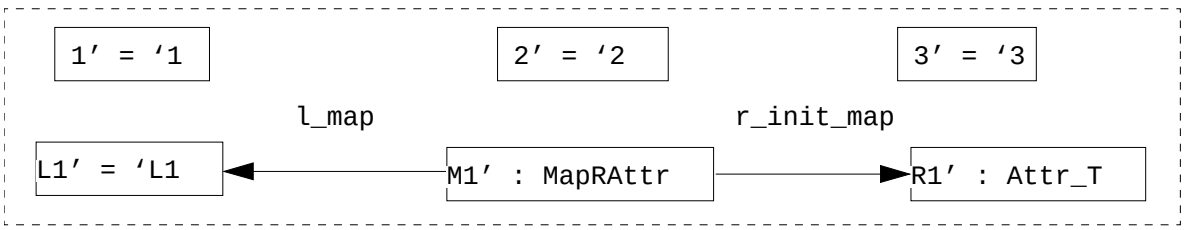



```
    condition VarWithMinRAttrs ( rel ) = NoOfRAttrsInVar ( 'L1 );
    transfer R1'.Name := '2.Name; R1'.MYVISIBLE := 1;
             R1'.old := false; M1'.old := false;
    return map_var := M1'; c := R1'; rattrs := ('L1.=lc_item=>) : RAttr;
end;

transaction Map_RAttrs( map_v : MapVariant ; cl : Class ;
                        rattrs : RAttr [0:n]) [0:1] =
    use a : Attr;
        map_r : MapRAttr
    do
        for all
          ra := rattrs.( not with ( <-item- & ( instance of Corresp ) &
                                    -left-> & ( instance of IND ) ) )
        do
            Map_RAttr ( cl, ra, out map_r, out a )
          & AppendLast ( map_v, map_r )
          & AppendLast ( cl, a )
       end
    end
end;
```

```
production Map_RAttr( c : Class ; rattr : RAttr ;
                      out map_attr : MapRAttr ; out attr : Attr) [0:1] =
```

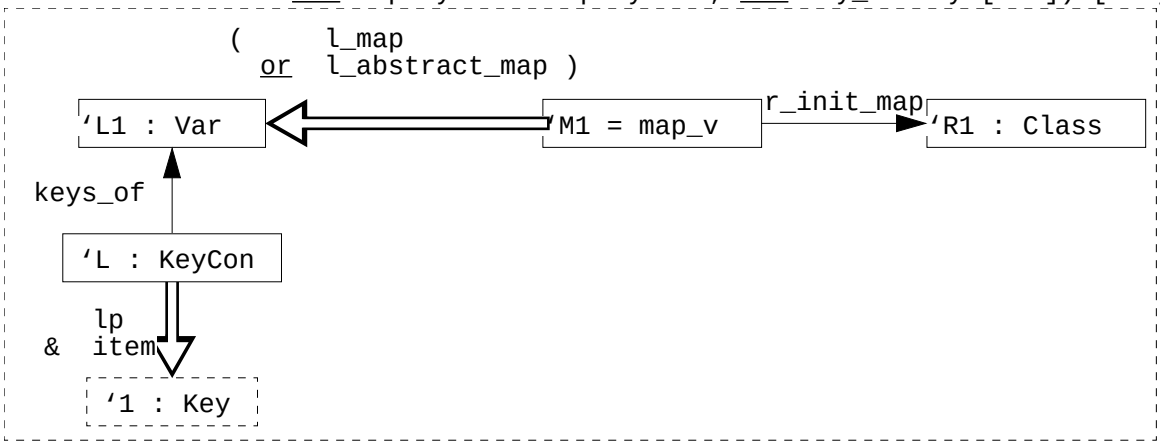



```
        ::=
```

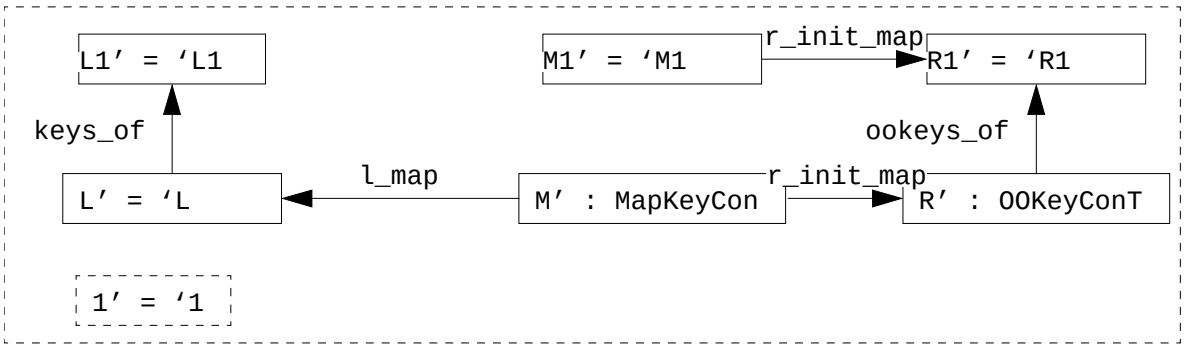



```
    condition ‘1.Name = substr ( ‘3.Name, 0, pos ( "#", ‘3.Name, 0 ) - 1 );
    transfer R1’.Name := ‘L1.Name;
             R1’.Type := [ ((substr ( ‘L1.RType, 0,
                           pos ( "(", ‘L1.RType, 0 ) - 1 ) = "CHAR") or
                           (substr ( ‘L1.RType, 0,
                           pos ( "(", ‘L1.RType, 0 ) - 1 ) = "VARCHAR" ) or
                           (substr ( ‘L1.RType, 0,
                           pos ( "(", ‘L1.RType, 0 ) - 1 ) = "char" ) or
                           (substr ( ‘L1.RType, 0,
                           pos ( "(", ‘L1.RType, 0 ) - 1 ) = "varchar" ))
                           :: "string"
                         | [ ((‘L1.RType = "INTEGER") or
                              (‘L1.RType = "Integer") or
                              (‘L1.RType = "integer"))
                              :: "int"
                           | ‘L1.RType ] ] ;
             R1’.NN := ‘L1.nn; R1’.old := false; M1’.old := false;
    return map_attr := M1’; attr := R1’;
end;

transaction Map_CurrentKey( map_v : MapVariant) [0:1] =
    use k : Key [0:1];
        oo_keycon : OOKeyCon;
        oo_key : OOKey;
        ookey : OOKey [0:n];
        keycons : KeyCon [0:n];
        keycon : KeyCon;
        map_keycon, old_map_keycon : MapKeyCon;
        map_key : MapKey;
        attrs : Attr [0:n];
        map_rattrs : MapRAttr [0:n]
    do
        keycons := (map_v.( -l_map-> or  -l_abstract_map-> ).
                    ( instance of Var ).<-keys_of-).instance of KeyCon
      & for all kc := keycons (* Progres don‘t accept keycon := def_elem
                                 (keycons) in a transaction with [0:1] !!! *)
        do
           keycon := kc
        end
      & choose
        when empty ( keycon.<-l_map- )
        then
            Map_KeyCon ( map_v, out oo_keycon, out map_keycon, out k )
          & choose
            when not empty ( k )
            then
```

```
                Map_Key ( k, out oo_key, out map_key )
              & AppendLast ( oo_keycon, oo_key )
              & AppendLast ( map_keycon, map_key )
              & map_rattrs := k.( lc_item &  instance of RAttr & <-l_map- &
                          instance of MapRAttr & valid (self.old = false) )
              & for all m_a := map_rattrs
                do
                  AppendLast ( map_key, m_a )
                & attrs := (attrs or m_a.-r_init_map->.( instance of Attr ))
                end
              & for all a := attrs
                do
                  AppendLast ( oo_key, a )
                end
            else
                skip
            end
        else
            Arrange_Map_KeyCon ( map_v, out map_keycon, out old_map_keycon )
          & ookey := old_map_keycon.-r_init_map->.( instance of OOKeyCon ).
                                    lc_item.instance of OOKey
          & for all ook := ookey
            do
              AppendLast ( (map_keycon.-r_init_map->) : OOKeyCon [0:1], ook )
            end
        end
      & AppendLast ( map_v, map_keycon )
    end
end;

production Map_KeyCon( map_v : MapVariant ; out ookeycon : OOKeyCon ;
                       out mapkeycon : MapKeyCon ; out key_ : Key [0:1]) [0:1] =
```



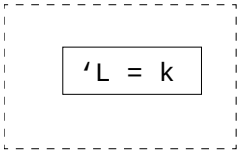

```
          ::=
```



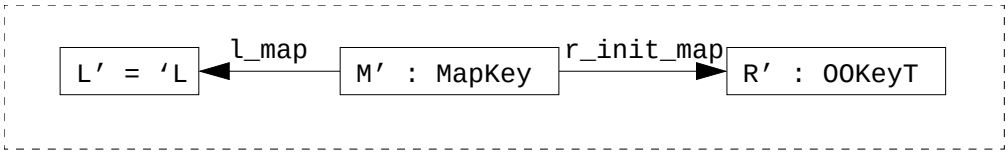

```
    condition 'R1.old = false;
    transfer R'.old := false; M'.old := false;
    return ookeycon := R'; mapkeycon := M'; key_ := 1';
end;
```

`Arrange_Map_KeyCon` verhält sich wie `Map_KeyCon` mit dem Unterschied, daß ein neuer `MapKeyCon`-Knoten und ein neuer `OOKeyCon`-Knoten angelegt werden und die alten als ungültig (`old := true`) markiert werden.

```
production Map_Key( k : Key ; out ookey : OOKey ;
                    out mapkey : MapKey) [0:1] =
```

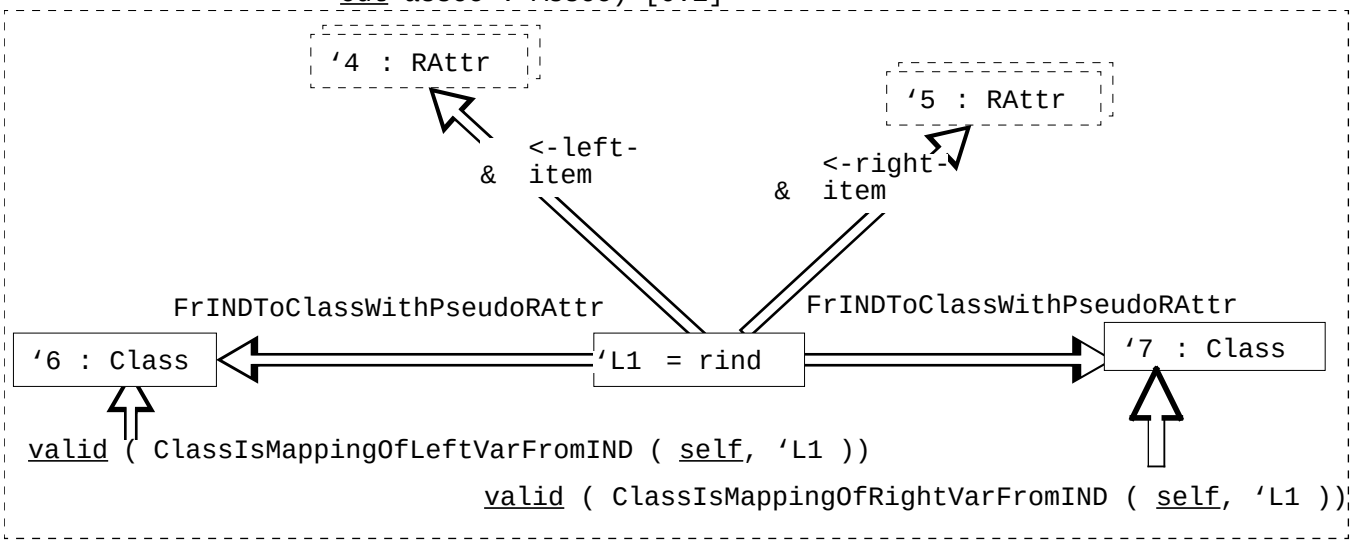


::=

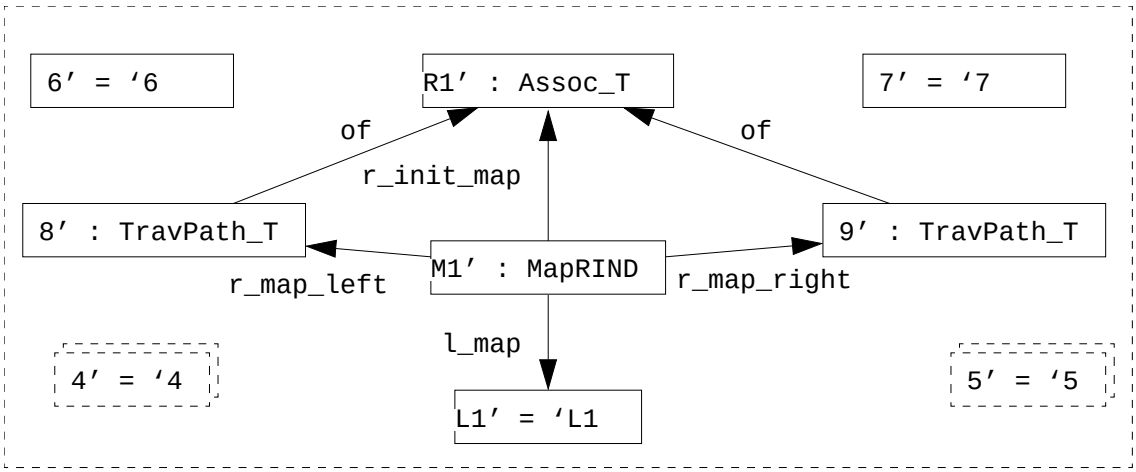



```
    transfer R'.prim := 'L.primary; R'.old := false;  M'.old := false;
    return ookey := R'; mapkey := M';
end;

transaction Map_INDs( map_schema : MapSchema ; act : integer) [0:1] =
    use inds, rinds : IND [0:n];
        cl1, cl2 : Class;
        tp1, tp2 : TravPath;
        assoc : Assoc;
        inh : Inherit;
        maprind : MapRIND;
        mapiind : MapIIND
   do
        IncrGet ( IND, out inds )
      & rinds := inds.valid ((self.keybased) and (self.isa = false))
      & for all i := rinds.valid (self.to_be_mapped)
        do
          choose
              Map_RIND ( i, out maprind, out cl1, out cl2,
                             out tp1, out tp2, out assoc )
            & AppendLast ( map_schema, maprind )
            & AppendLast ( cl1, tp1 )
            & AppendLast ( cl2, tp2 )
            & assoc.actuality_ := act
          else
            skip
          end
        end
      & for all i := inds.valid ((self.isa) and (self.to_be_mapped))
        do
          choose
              Map_IIND ( i, out mapiind, out cl1, out cl2, out inh )
            & AppendLast ( map_schema, mapiind )
            & inh.actuality_ := act
          else
            skip
          end
        end
    end
end;
```

```
production Map_RIND( rind : IND ; out map_rind : MapRIND ;
                     out class1 : Class ; out class2 : Class ;
                     out tpath1 : TravPath ; out tpath2 : TravPath ;
                     out assoc : Assoc) [0:1] =
```

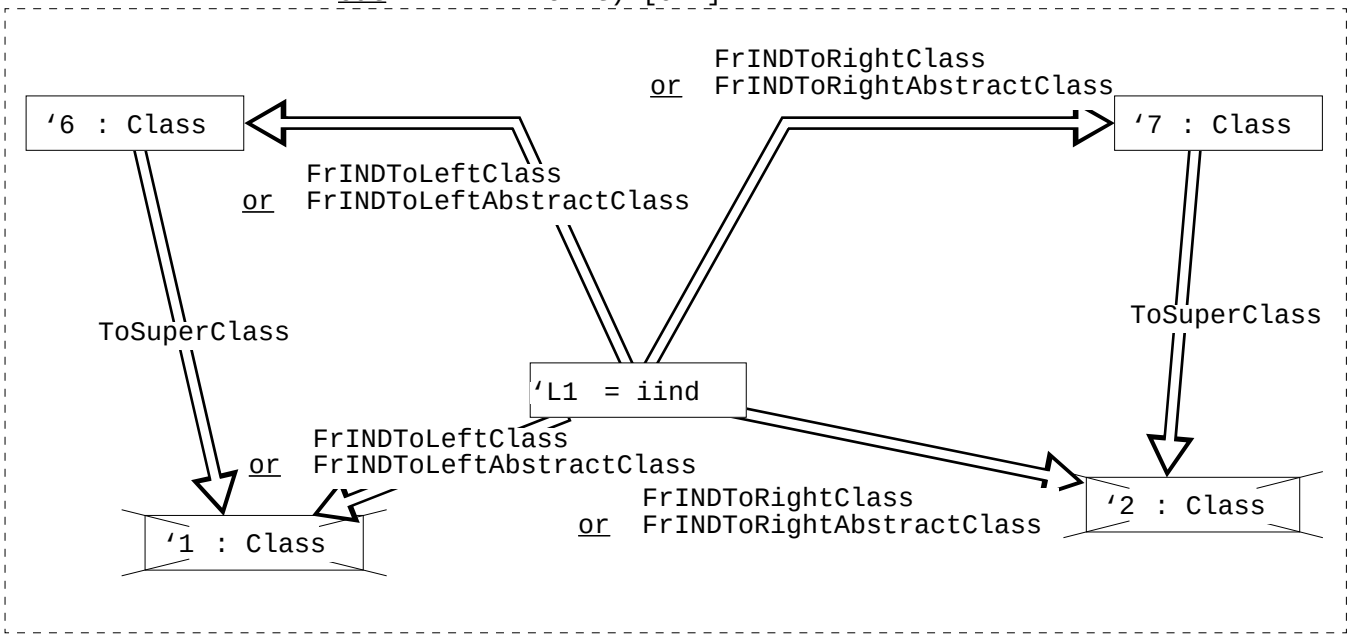


::=

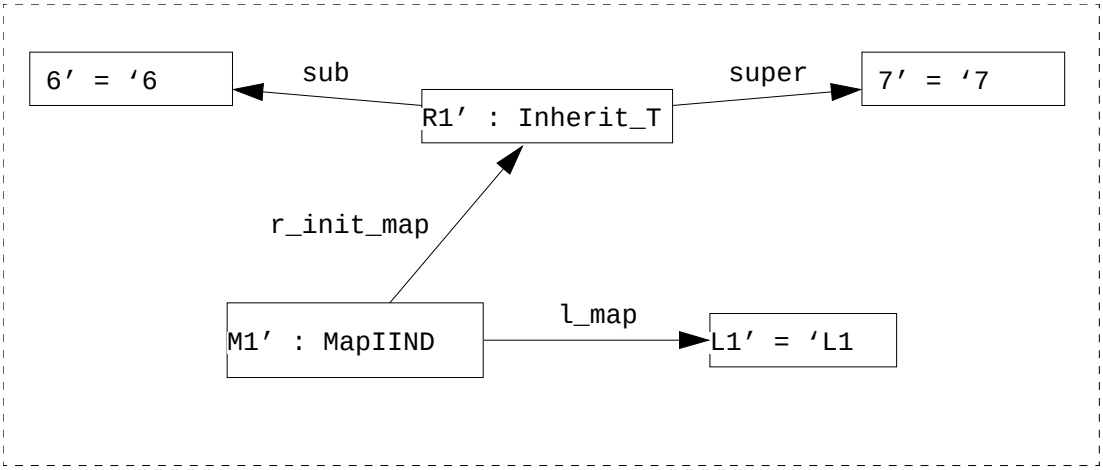



```
    folding { ‘6, ‘7 };
    transfer 9’.NN := ‘L1.inverse; 9’.many := not ‘L1.invkeybased;
             8’.many := false; 8’.NN := true;
             6’.AssocCounter := ‘6.AssocCounter + 1;
             7’.AssocCounter := ‘7.AssocCounter + 1;
             8’.Name := ‘7.Name & "_" & string ( ‘7.AssocCounter );
             9’.Name := ‘6.Name & "_" & string ( ‘6.AssocCounter );
             L1’.to_be_mapped := false; L1’.to_be_killed := false;
             R1’.old := false; M1’.old := false;
             8’.old := false; 9’.old := false;
    return class1 := 6’; class2 := 7’; tpath1 := 8’; tpath2 := 9’;
           map_rind := M1’; assoc := R1’;
end;
```

```
production Map_IIND( iind : IND ; out map_iind : MapIIND ;
                     out class1 : Class ; out class2 : Class ;
                     out inh : Inherit) [0:1] =
```




```
        ::=
```

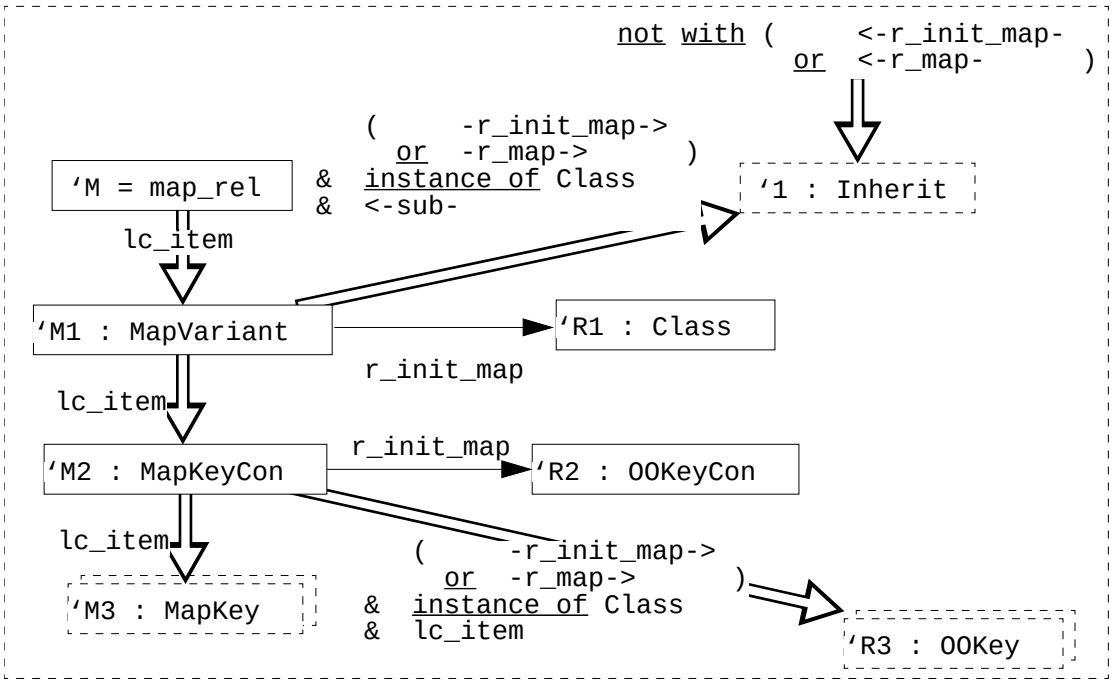



```
    transfer 6’.AssocCounter := ‘6.AssocCounter + 1;
             7’.AssocCounter := ‘7.AssocCounter + 1;
             L1’.to_be_mapped := false; L1’.to_be_killed := false;
             R1’.old := false; M1’.old := false;
    return class1 := 6’; class2 := 7’; map_iind := M1’; inh := R1’;
end;
```

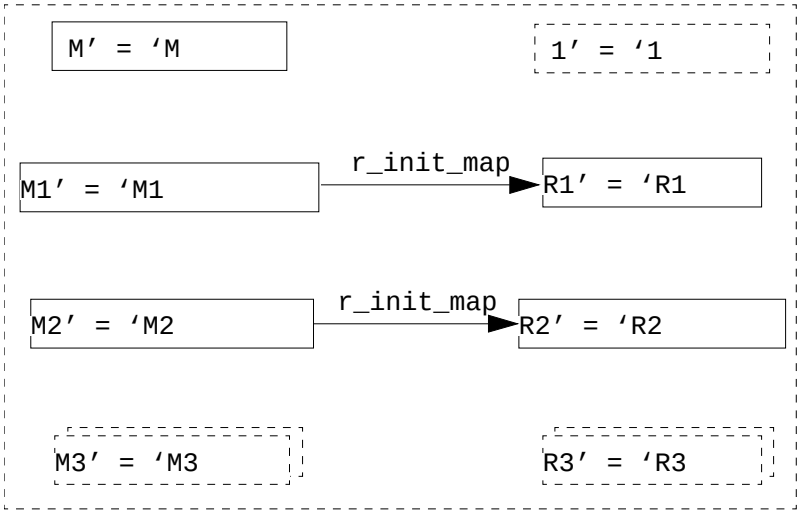
Wiederholung der initialen Abbildung

```
transaction DoDefaultMapping( ooschema : OOSchema ;
                      out Errstr : string ; out Errstr1 : string) [0:1] =
    use
        mapschema : MapSchema
            := (ooschema.<-r_init_map-.instance of MapSchema) : [0:1];
        act : integer;
        errstr, errstr1 : string
    do
        TestOnInheritanceForVars ( out errstr )
      & IncrementOOSchemaActuality ( out act )
      & Map_Relations ( mapschema, ooschema, act, out errstr1 )
      & Map_INDs ( mapschema, act )
      & Errstr := errstr
      & Errstr1 := errstr1
    end
end;

transaction OldMappings [0:1] =
    OldMappingsOfRelations
  & OldMappingsOfINDs
end;

transaction OldMappingsOfRelations [0:1] =
    use relations : Relation [0:n];
        map_rels : MapRelation [0:n]
    do
        IncrGet ( Relation, out relations )
      & for all r := relations
        do
          choose
            when (empty ( r.<-l_map-.( instance of MapRelation ) ))
            then
              r.to_be_killed := false
            else
              when NotEverySubMappingOfRelHasAnl_mapEdge ( r )
              then
                r.to_be_killed := true
            else
              skip
            end
        end
      & for all r := relations.valid ((self.to_be_killed))
        do
             OldMappingOfRelation
                 ( (r.<-l_map-.instance of MapRelation) : [0:1] )
           & r.to_be_mapped := true
        end
      & IncrGet ( MapRelation, out map_rels )
      & for all map_r :=
              map_rels.valid (empty ( self.-l_map->.instance of Relation ))
        do
            OldMappingOfRelation ( map_r )
        end
    end
end;

transaction OldMappingOfRelation( map_rel : MapRelation) [0:1] =
    loop
      OldMappingOfRAttr ( map_rel )
    end
  & loop
      OldMappingOfVariant ( map_rel )
    end
  & map_rel.old := true
end;
```

```
production OldMappingOfVariant( map_rel : MapRelation) [0:1] =
```

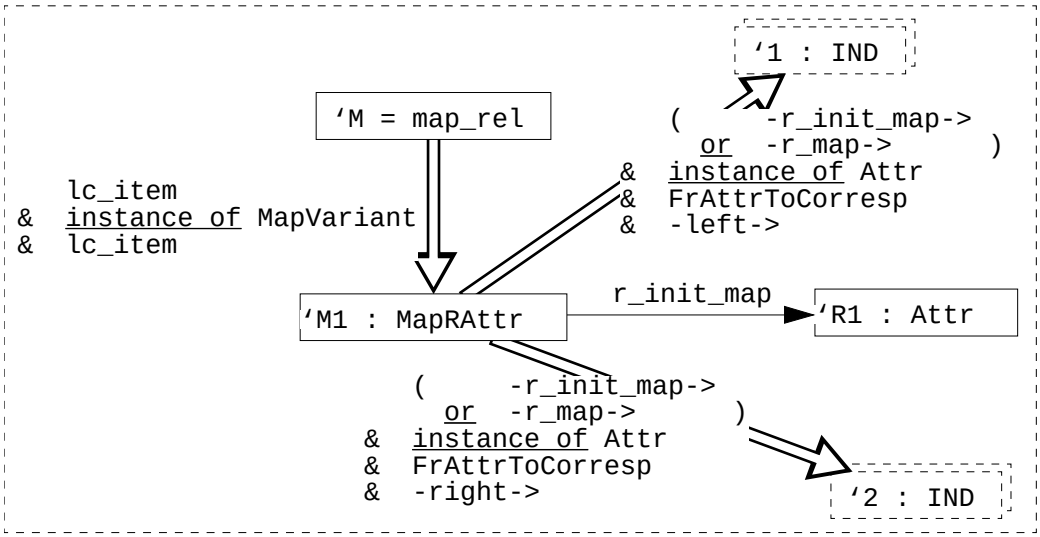


::=

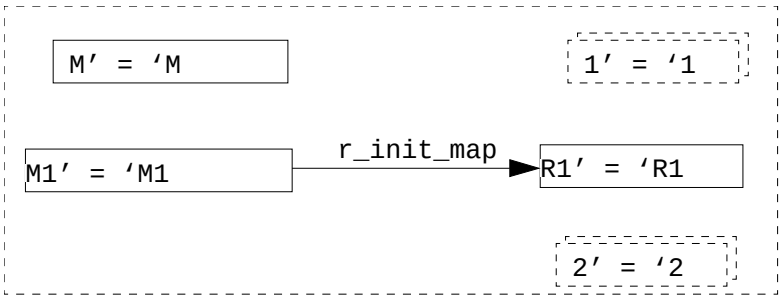



```
    condition (‘M1.old = false) and (‘R1.old = false) and
              (‘M2.old = false) and (‘R2.old = false ;
    transfer M1’.old := true; R1’.old := true; M2’.old := true;
             R2’.old := true; M3’.old := true; R3’.old := true;
             1’.old := true;
end;
```

```
production OldMappingOfRAttr( map_rel : MapRelation) [0:1] =
```

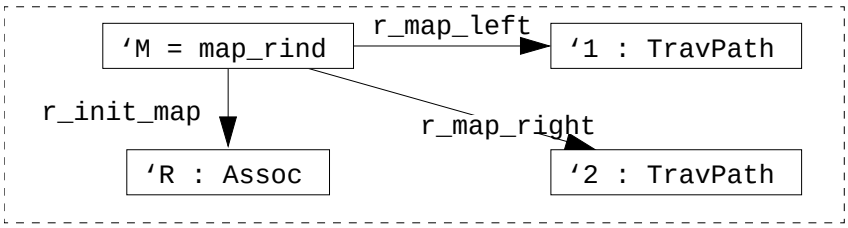



```
        ::=
```

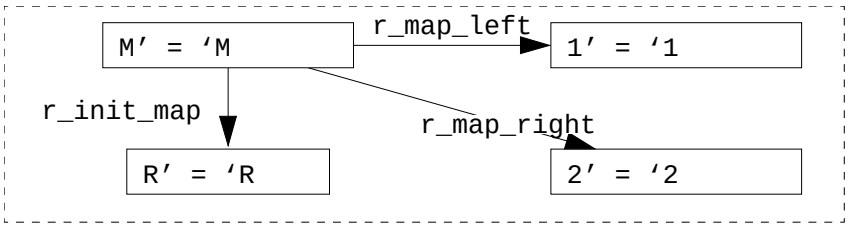



```
    condition (‘M1.old = false) and (‘R1.old = false);
    transfer 1’.to_be_killed := true; 2’.to_be_killed := true;
             M1’.old := true; R1’.old := true;
end;

transaction OldMappingsOfINDs [0:1] =
    use inds : IND [0:n];
        map_rinds : MapRIND [0:n];
        map_iinds : MapIIND [0:n]
    do
        IncrGet ( IND, out inds )
      & for all i := inds.valid( (self.to_be_killed) or
                                 ((self.FrLeftINDToRel).to_be_killed) or
                                 ((self.FrRightINDToRel).to_be_killed) )
        do
          choose
            when not empty ( i.<-l_map- )
            then
              choose
                  OldMappingOfRIND
                        ( (i.<-l_map-.instance of MapRIND) : [0:1] )
              else
                  OldMappingOfIIND
                        ( (i.<-l_map-.instance of MapIIND) : [0:1] )
              else
                  skip
              end
          else
              skip
          end
        & i.to_be_mapped := true
        end
      & IncrGet ( MapRIND, out map_rinds )
     & for all map_rind
               := map_rinds.valid (empty ( self.-l_map->.instance of IND ))
```

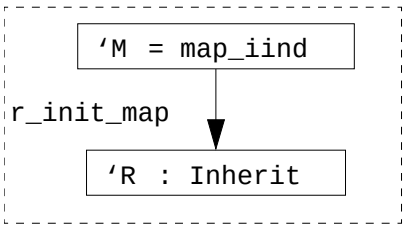
```
        do
          choose
              OldMappingOfRIND ( map_rind )
          else
              skip
          end
        end
      & IncrGet ( MapIIND, out map_iinds )
      & for all map_iind
                := map_iinds.valid (empty ( self.-l_map->.instance of IND ))
        do
          choose
              OldMappingOfIIND ( map_iind )
          else
              skip
          end
        end
    end
end;

production OldMappingOfRIND( map_rind : MapRIND) [0:1] =
```



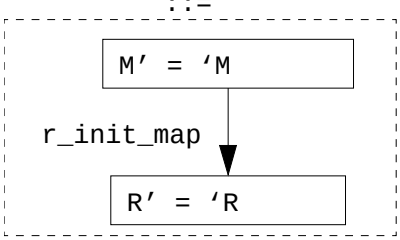
::=




```
    condition (‘M.old = false) and (‘R.old = false);
    transfer M’.old := true; R’.old := true;
             1’.old := true; 2’.old := true;
end;

production OldMappingOfIIND( map_iind : MapIIND) [0:1] =
```



::=




```
    condition (‘M.old = false) and (‘R.old = false);
    transfer M’.old := true; R’.old := true;
end;
```

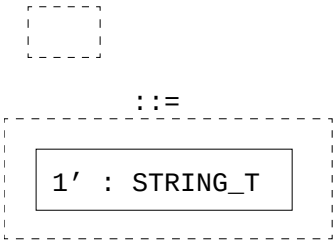
Anhang B

Restrukturierungstransformationen - Implementierung

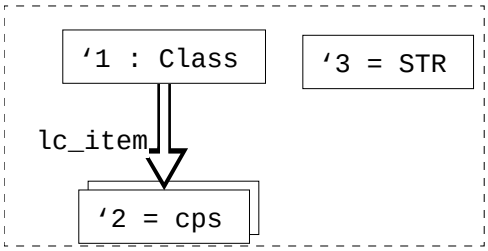
Zuerst wird anhand von SplitClass eine Restrukturierungstransformation komplett gezeigt. Für die restlichen Transformationen werden nur die Rümpfe aufgeführt.

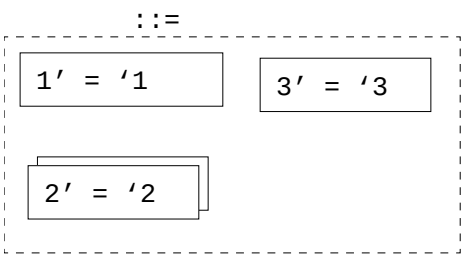
```
transaction DUI_SplitClass( cps : ClassProp [1:n] ; newName : string ;
                            out _errMsg : string) =
                            (* UserInteractionNameIs SplitClass
                               DirectCall *)
    use newclass, oldclass : Class;
        errStr : string := "";
        assoc : Assoc;
        tpnew, tpold : TravPath;
        ookeycon : OOKeyCon;
        ookeys : OOKey [0:n];
        OK : boolean := false;
        attrs : Attr [0:n];
        STR : STRING;
        trafo : Trafo;
        in_incrs, out_incrs : Increment [0:n];
        maplists : MapList [0:n];
        mapincrs : Increment [0:n]
    do
        attrs := cps.instance of Attr
      & ookeys := attrs.ToEnvList.instance of OOKey
      & for all k := ookeys
        do
          choose
          when (card ( attrs and (k.lc_item.instance of Attr) )
                     # card ( (k.lc_item.instance of Attr) )  )
          then
              OK := true
          end
        end
      & choose
        hen (card ( cps.ToEnvList.instance of Class ) > 1)
        then
            errStr := "All selected class properties must belong
                       to one single class."
        else
          choose
          when OK
          then
              errStr := "Only a complete key can participate when
                         a class is splitted."
          else
            choose
                CreateString ( newName, out STR )
              & Test_SplitClass ( cps, STR, out oldclass )
              & in_incrs := (oldclass or cps or STR)
              & InParams ( in_incrs, "SplitClass", out trafo )
              & maplists := (oldclass.<-r_map-.instance of MapList)
              & SplitClass ( oldclass, cps, STR, out newclass, out assoc,
                             out tpold, out tpnew, out ookeycon )
              & out_incrs :=(oldclass or cps or newclass or assoc or
                             tpnew or tpold or ookeycon)
```

```
              & OutParams ( trafo, out_incrs )
              & mapincrs := (newclass or assoc)
              & ActualiseMapping ( maplists, mapincrs )
              & not ExistanceOfTwoClassesWithTheSameName ( newclass )
            else
                errStr := "There is already a class with the choosen name."
            end
          end
      end
    & _errMsg := errStr
    end
end;

production CreateString( str : string ; out STR : STRING) [0:1] =
```



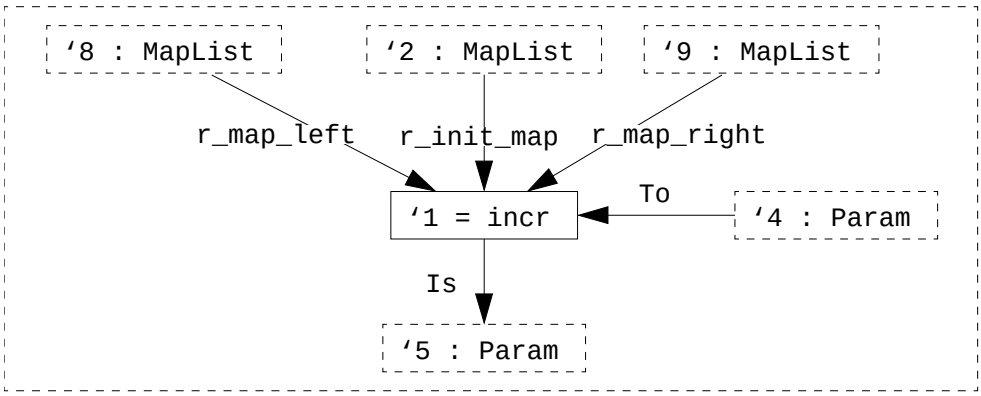

```
    transfer 1'.Name := str;  1'.MYVISIBLE := - 1;
             1'.old := false; 1'.history := false;
    return STR := 1';
end;

production Test_SplitClass( cps : ClassProp [1:n] ; STR : STRING ;
                            out oldClass : Class) [0:1] =
```



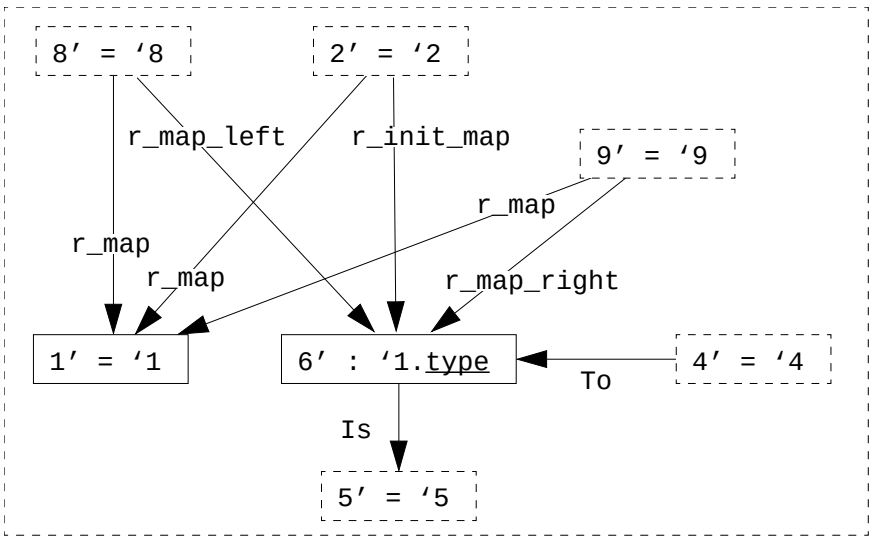



```
    transfer 1'.no := 1; 2'.no := 2; 3'.no := 3;
    return oldClass := 1';
end;

transaction InParams( incrs : Increment [0:n] ; art : string ;
                      out trafo : Trafo) [0:1] =
    use new_incr : Increment;
        new_incrs : Increment [0:n];
        traf : Trafo
    do
      for all incr := incrs
      do
        choose
        when not empty ( incr.instance of STRING )
        then
            new_incrs := (new_incrs or incr)
```

```
        else
            CopyIncrement ( incr, out new_incr )
          & new_incrs := (new_incrs or new_incr)
        end
      end
    & CreateTrafo ( art, out traf )
    & for all n_incr := new_incrs
      do
        InParam ( traf, n_incr )
      end
    & trafo := traf
    end
end;

production CopyIncrement( incr : Increment ; out new_incr : Increment)
           [0:1] =
```

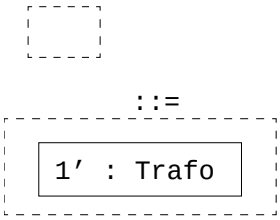



```
        ::=
```

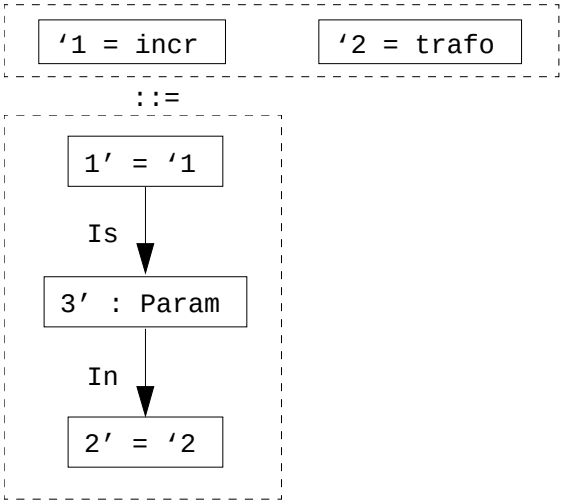



```
    transfer 1’.history := false; 6’.no := ‘1.no; 6’.old := false;
             6’.history := true; 6’.MYVISIBLE := - 1;
             6’.help := [ not empty ( ‘1.instance of NamedIncrement ) ::
                          (‘1 : NamedIncrement [1:1]).Name
                        | not empty ( ‘1.instance of Assoc ) :: "Association"
                        | not empty ( ‘1.instance of Aggr ) :: "Aggregation"
                        | not empty ( ‘1.instance of Inherit ) :: "Inherit"
                        | "" ] ;
    return new_incr := 6’;
end;
```

```
production CreateTrafo( art_ : string ; out trafo : Trafo) [0:1] =
```

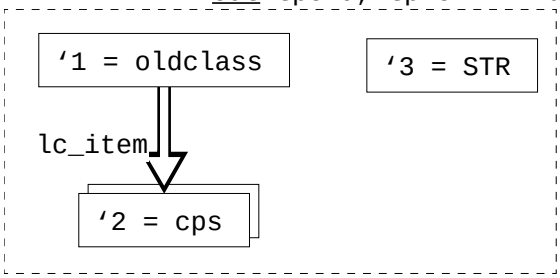



```
    transfer 1’.art := art_;  1’.old := false;
    return trafo := 1’;
end;

production InParam( trafo : Trafo ; incr : Increment) [0:1] =
```

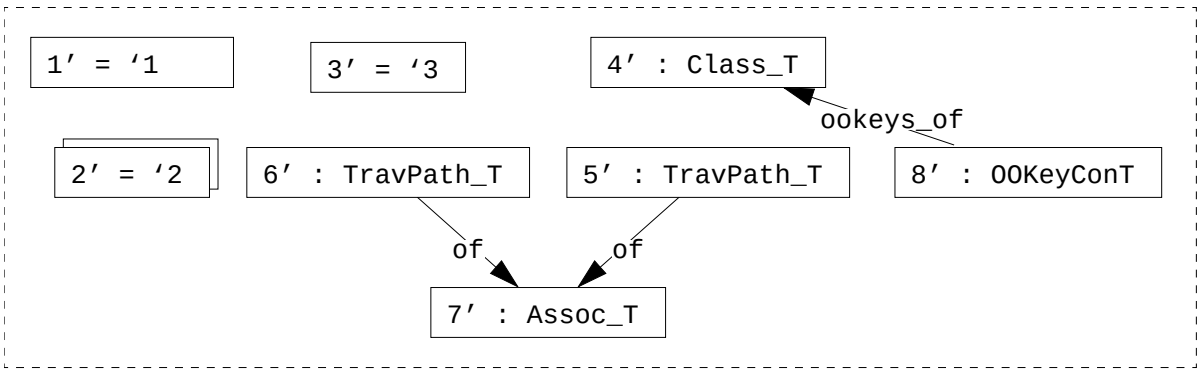



```
    transfer 1’.old := false; 1’.history := true;
             3’.old := false; 3’.no := ‘1.no;
             3’.text := [ (‘1.help = "") ::
                          [ empty ( ‘1.instance of STRING ) :: ""
                            | (‘1 : STRING [1:1]).Name ]
                        | ‘1.help ] ;
end;

transaction SplitClass( oldclass : Class ; cps : ClassProp [1:n] ;
          STR : STRING ; out newclass_ : Class ; out assoc_ : Assoc ;
          out tpold_, tpnew_ : TravPath ; out ookeycon_ : OOKeyCon) [0:1] =
    use act : integer;
        newclass : Class;
       schema : OOSchema;
       assoc : Assoc;
       tpnew, tpold : TravPath;
       ookeycon : OOKeyCon
    do
       DHM_SplitClass ( oldclass, cps, STR, out newclass, out assoc,
                        out tpold, out tpnew, out ookeycon )
     & for all cp := cps
       do
           RmFromList ( oldclass, cp )
         & AppendLast ( newclass, cp )
       end
     & for all keys := oldclass.<-ookeys_of-.lc_item
       do
           RmFromList ( oldclass.<-ookeys_of-, keys )
         & AppendLast ( ookeycon, keys )
       end
     & schema := [ (oldclass.ToEnvList) : OOSchema [0:1] | halt ]
     & InsertFirst ( schema, newclass )
     & IncrementOOSchemaActuality ( out act )
```

```
     & newclass.actuality_ := act
     & oldclass.actuality_ := act
     & assoc.actuality_ := act
     & AppendLast ( oldclass, tpold )
     & AppendLast ( newclass, tpnew )
     & for all relship := cps.( instance of TravPath ).-of->
       do
           relship.actuality_ := act
       end
     & newclass_ := newclass
     & assoc_ := assoc
     & tpold_ := tpold
     & tpnew_ := tpnew
     & ookeycon_ := ookeycon
    end
end;

production DHM_SplitClass( oldclass : Class ; cps : ClassProp [1:n] ;
             STR : STRING ; out newclass : Class ; out assoc : Assoc ;
             out tpold, tpnew : TravPath ; out ookeycon : OOKeyCon) [0:1] =
```

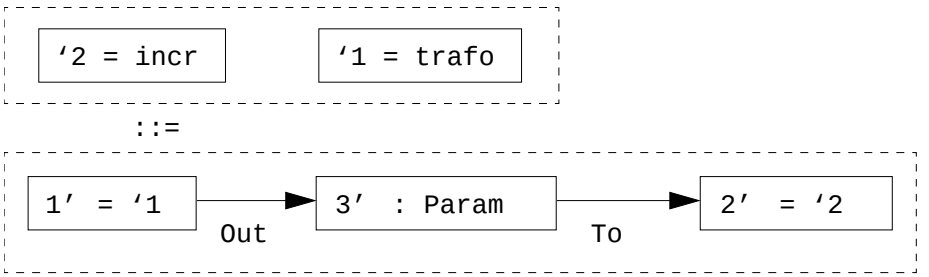


::=

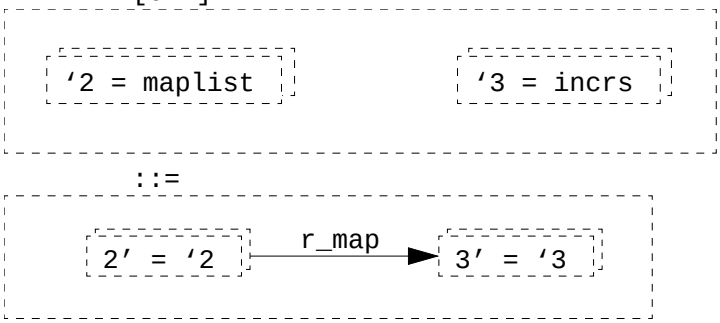



```
    transfer 1’.AssocCounter := ‘1.AssocCounter + 1; 4’.Name := ‘3.Name;
             4’.MYVISIBLE := 1; 4’.abstract := false;
             4’.AssocCounter := 1; 6’.NN := true; 6’.many := false;
             6’.Name := ‘3.Name & "_" & string ( ‘1.AssocCounter );
             5’.NN := true; 5’.many := false; 5’.Name := ‘1.Name & "_0";
             1’.no := 1; 2’.no := 2; 4’.no := 3;
end;

transaction OutParams( trafo : Trafo ; incrs : Increment [0:n]) [0:1] =
    for all incr := incrs
    do
        OutParam ( trafo, incr )
    end
end;
```

```
production OutParam( trafo : Trafo ; incr : Increment) [0:1] =
```



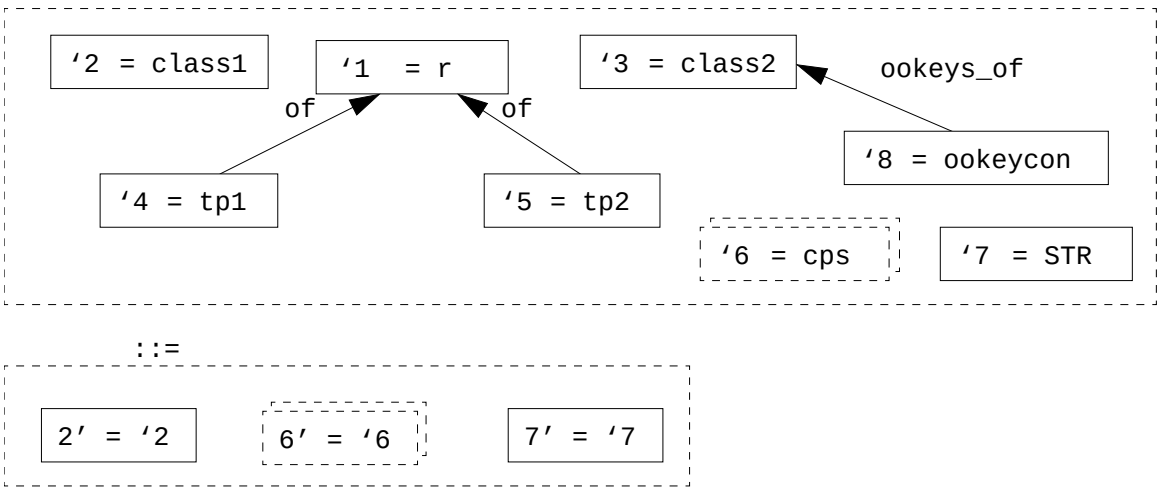

```
    transfer 1’.old := false; 2’.old := false; 2’.history := false;
             3’.old := false; 3’.no := ‘2.no;
             3’.text := [ not empty ( ‘2.instance of NamedIncrement ) ::
                          (‘2 : NamedIncrement [1:1]).Name | "" ] ;
end;

production ActualiseMapping( maplist : MapList [0:n] ;
                             incrs : Increment [0:n])
             (* One of the two sets can be empty, for example after
               GeneralizeNewWithSupeClass. In general it is exactly one. *)
        [0:1] =
```




```
end;
```

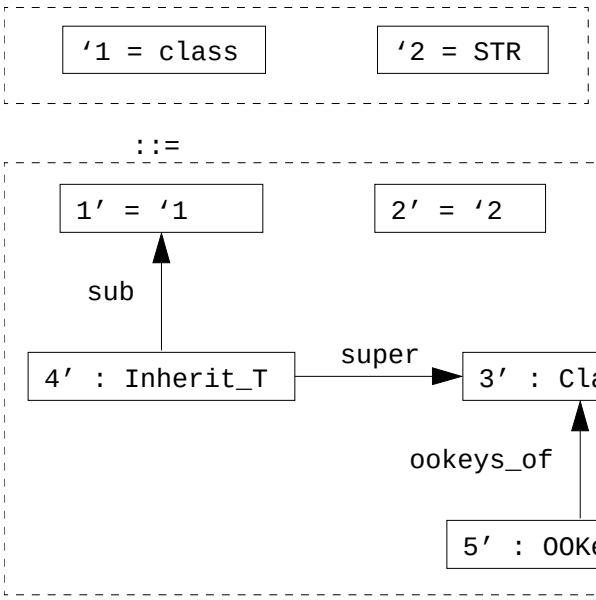
Es folgen die Rümpfe der in VARLET vorhandenen Transformationen.

```
production DHM_MergeClass( r : Relationship ; STR : STRING ;
                           class1, class2 : Class ; cps : ClassProp [0:n] ;
                           tp1, tp2 : TravPath ; ookeycon : OOKeyCon) [0:1] =
```

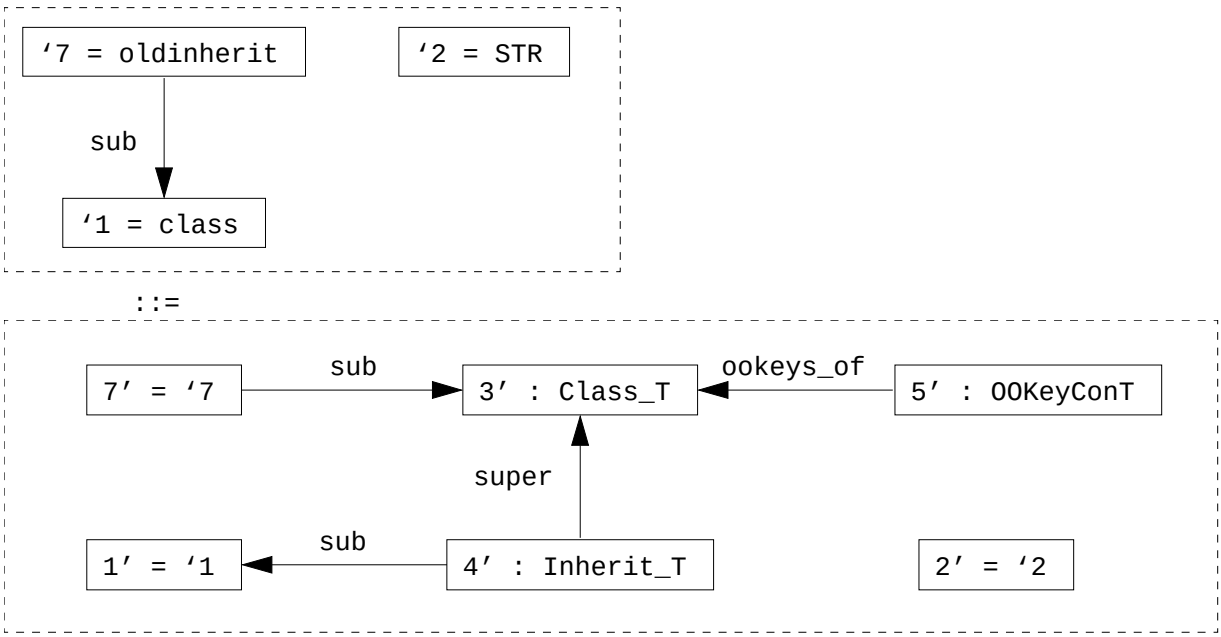



```
    transfer 2’.Name := ‘7.Name; 2’.AssocCounter := ‘2.AssocCounter - 1;
             2’.no := 1; 6’.no := 2;
end;
```

```
production DHM_GeneralizeNew_withoutSuperClass( class : Class ;
                   STR : STRING ; abs : boolean ; out newclass : Class ;
                   out inherit : Inherit ; out ookeycon : OOKeyCon) [0:1] =
```

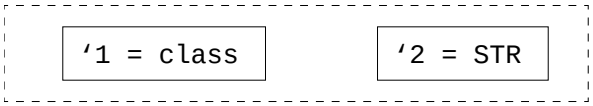



```
    transfer 3’.Name := ‘2.Name; 3’.MYVISIBLE := 1; 3’.abstract := abs;
             3’.to_be_mapped := (not abs); 1’.no := 1;
             3’.no := 2; 4’.no := 3;  5’.no := 4;
    return newclass := 3’; inherit := 4’; ookeycon := 5’;
end;

production DHM_GeneralizeNew_withSuperClass( class : Class ; STR : STRING ;
              oldinherit : Inherit ; abs : boolean ; out newclass : Class ;
              out inherit : Inherit ; out newinherit : Inherit ;
              out ookeycon : OOKeyCon) [0:1] =
```

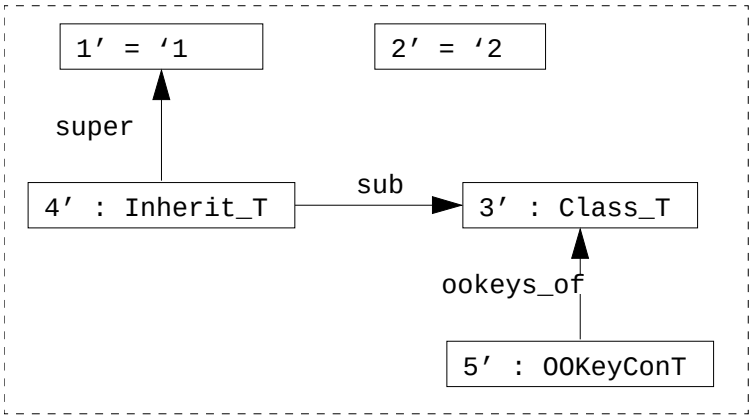



```
    transfer 3’.Name := ‘2.Name; 3’.MYVISIBLE := 1; 3’.abstract := abs;
             3’.to_be_mapped := (not abs); 1’.no := 1; 3’.no := 2;
             4’.no := 3; 7’.no := 4; 5’.no := 5;
    return newclass := 3’; inherit := 4’; newinherit := 7’; ookeycon := 5’;
end;
```

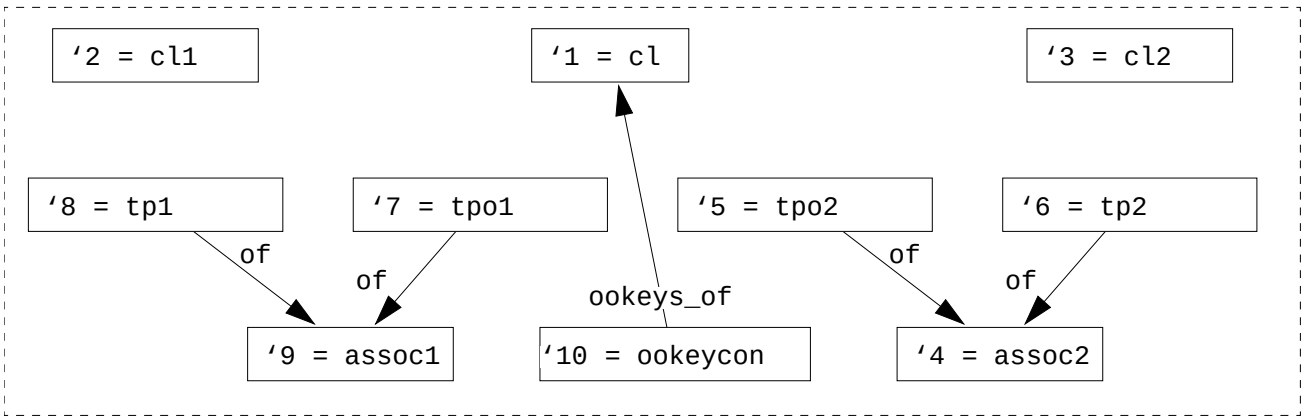
```
production DHM_SpecializeNew( class : Class ; STR : STRING ;
                              out newclass : Class ; out inherit : Inherit ;
                              out ookeycon : OOKeyCon) [0:1] =
```



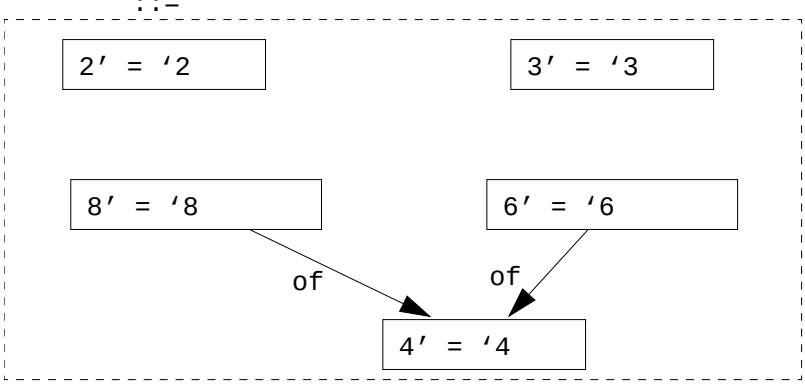
::=




```
    transfer 3’.Name := ‘2.Name; 3’.MYVISIBLE := 1; 3’.abstract := false;
             3’.to_be_mapped := true; 1’.no := 1; 3’.no := 2;
             4’.no := 3; 5’.no := 4;
    return newclass := 3’; inherit := 4’; ookeycon := 5’;
end;

production DHM_ClassToRelationship( cl : Class ; cl1, cl2 : Class ;
                        tp1, tp2, tpo1, tpo2 : TravPath ;
                        assoc1, assoc2 : Assoc ; ookeycon : OOKeyCon) [0:1] =
```

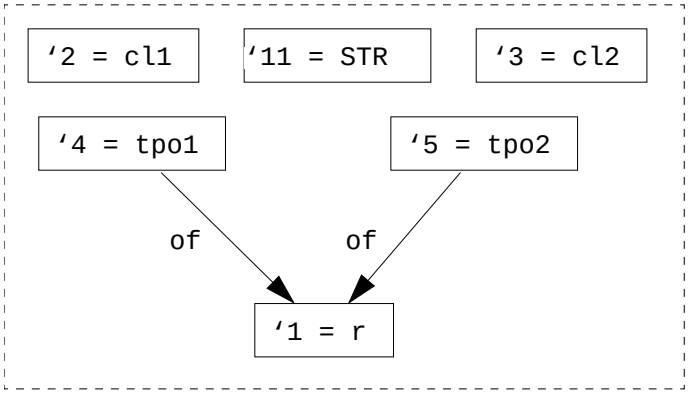


::=

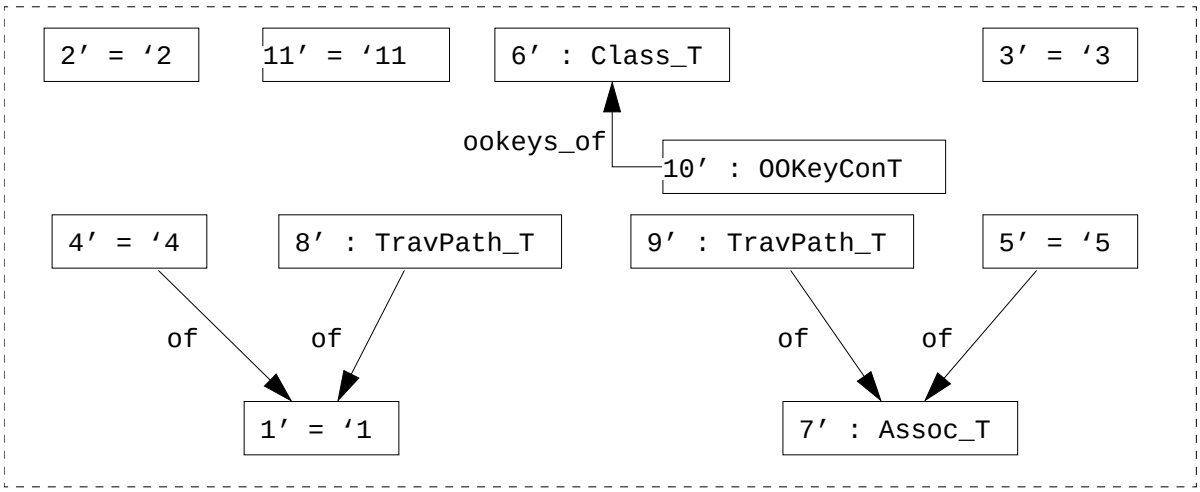



```
    folding { ‘2, ‘3 };
    transfer 2’.no := 1; 3’.no := 2; 4’.no := 3; 8’.no := 4; 6’.no := 5;
end;
```

```
production DHM_RelationshipToClass( r : Relationship ; STR : STRING ;
                cl1, cl2 : Class ; tpo1, tpo2 : TravPath ;
                out newcl : Class ; out assoc : Assoc ;
                out tp1, tp2 : TravPath ;  out ookeycon : OOKeyCon) [0:1] =
```

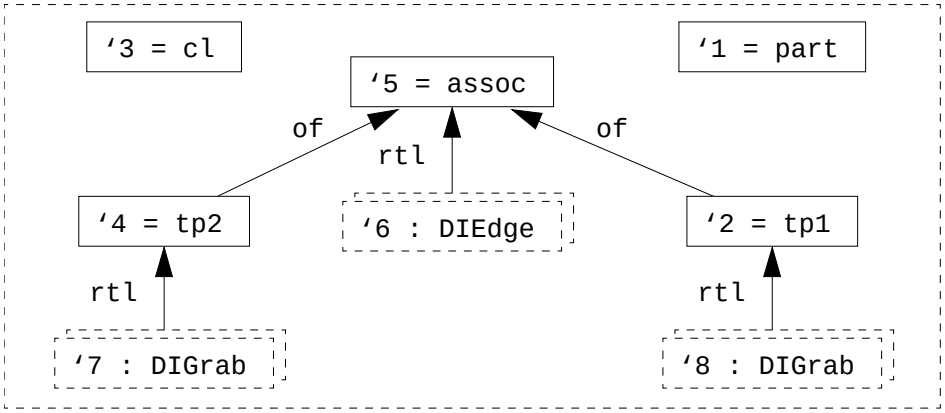


::=

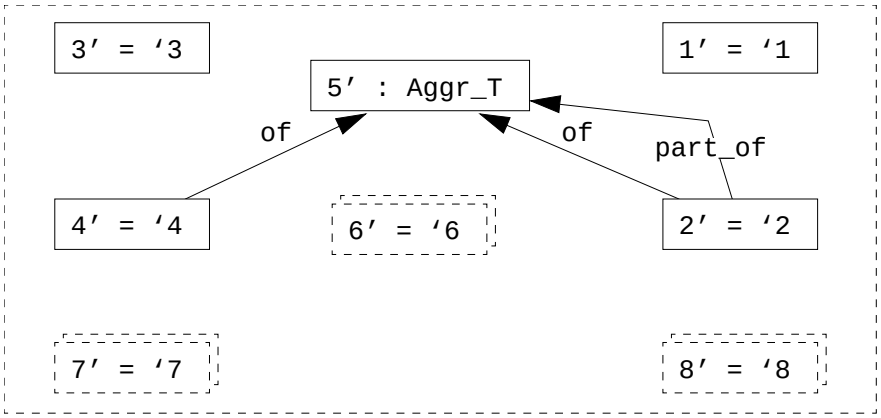



```
    transfer 6’.Name := ‘11.Name; 6’.MYVISIBLE := 1; 6’.abstract := false;
             6’.AssocCounter := 2; 8’.NN := true; 8’.many := false;
             8’.Name := ‘11.Name & "_0"; 9’.NN := true; 9’.many := false;
             9’.Name := ‘11.Name & "_1"; 1’.no := 1; 2’.no := 2;
             3’.no := 3; 4’.no := 4; 5’.no := 5;  6’.no := 6;
             7’.no := 7; 8’.no := 8; 9’.no := 9; 10’.no := 10;
    return newcl := 6’; assoc := 7’; tp1 := 8’; tp2 := 9’; ookeycon := 10’;
end;

transaction EqualizeAttributes( cl : Class ; attr1, attr2 : Attr ;
                                STR : STRING)[0:1] =
    use act : integer
    do
        RmFromList ( cl, attr2 )
      & RemoveIncrement ( attr2 )
      & attr1.Name := STR.Name
      & cl.no := 1
      & attr1.no := 2
      & IncrementOOSchemaActuality ( out act )
      & cl.actuality_ := act
      & cl.to_be_mapped := true
    end
end;
```

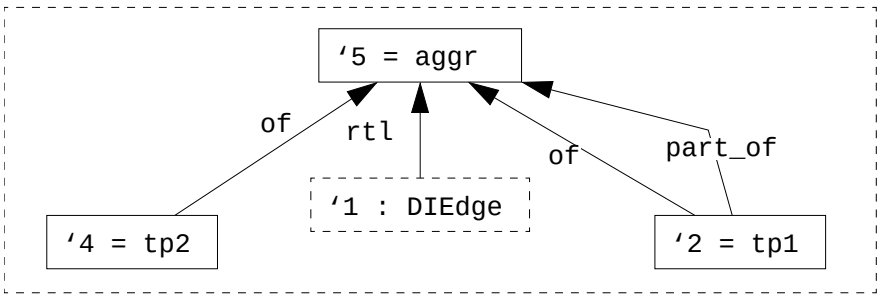
```
production DHM_Aggregate( part : Class ; assoc : Assoc ; cl : Class ;
                          tp1, tp2 : TravPath ; out aggr : Aggr) [0:1] =
```



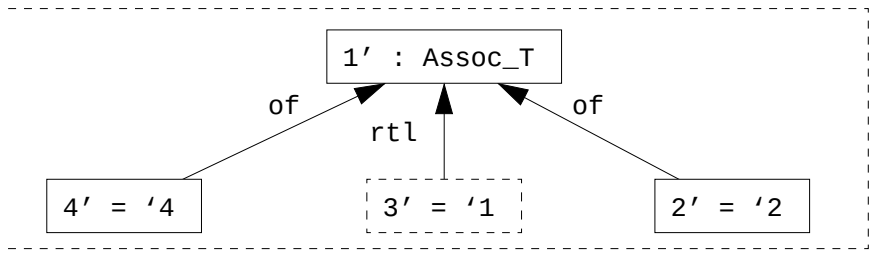
::=




```
    transfer 1’.no := 1; 3’.no := 2; 2’.no := 3; 4’.no := 4; 5’.no := 5;
    return aggr := 5’;
end;

production DHM_DisAggregate( aggr : Aggr ; tp1, tp2 : TravPath ;
                             out assoc : Assoc) [0:1] =
```



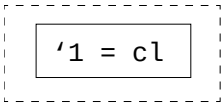
::=



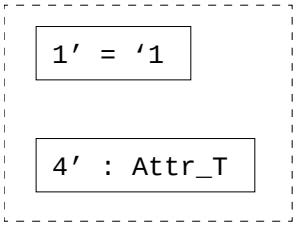

```
    transfer 1’.no := 1; 2’.no := 2; 4’.no := 3;
    return assoc := 1’;
end;
```

```
transaction MoveClassPropsInHierarchy( class : Class ;
                                       cps : ClassProp [1:n] ; goal : Class ;
                                       ookeys : OOKey [0:n]) [0:1] =
    use act : integer;
        tps : TravPath [0:n];
        relship : Relationship
    do
        for all k := ookeys
        do
            RmFromList ( class.<-ookeys_of-, k )
          & AppendLast ( goal.<-ookeys_of-, k )
          & k.no := 4
        end
      & for all cl_prop := cps
        do
          choose
              RmFromList ( class, cl_prop )
          else
              skip
          end
        & AppendLast ( goal, cl_prop )
        & cl_prop.no := 2
        end
      & class.no := 1
      & goal.no := 3
      & IncrementOOSchemaActuality ( out act )
      & class.actuality_ := act
      & goal.actuality_ := act
      & class.to_be_mapped := true
      & goal.to_be_mapped := true
      & tps := cps.instance of TravPath
      & for all tp := tps
        do
            relship := tp.-of->.instance of Relationship
          & relship.actuality_ := act
          & relship.to_be_mapped := true
        end
    end
end;

transaction GeneralizeAttribute( cl, supercl : Class ; attr : Attr ;
                                 out newattr_ : Attr) [0:1] =
    use act : integer;
        newattr : Attr
    do
        DHM_DuplicateAttribute
            ( supercl, attr.Name, attr.Type, attr.NN, out newattr )
      & AppendLast ( supercl, newattr )
      & newattr_ := newattr
      & cl.no := 1
      & supercl.no := 2
      & attr.no := 3
      & IncrementOOSchemaActuality ( out act )
      & supercl.actuality_ := act
      & supercl.to_be_mapped := true
    end
end;
```

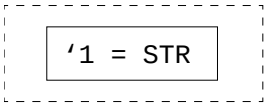
```
production DHM_DuplicateAttribute( cl : Class ; str1, str2 : string ;
                                   nn : boolean ; out attr : Attr) [0:1] =
```



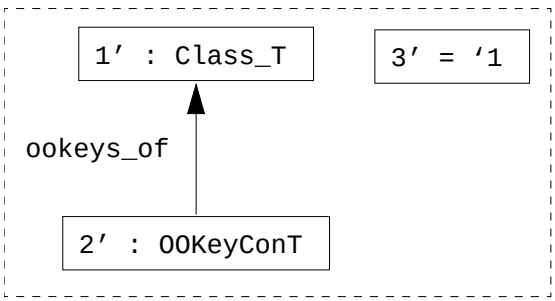
::=




```
    transfer 4’.Name := str1; 4’.Type := str2; 4’.NN := nn; 4’.no := 4;
    return attr := 4’;
end;

transaction SpecializeAttribute( cl, subcl : Class ; attr : Attr ;
                                 out newattr_ : Attr) [0:1] =
    use act : integer;
        newattr : Attr
    do
        DHM_DuplicateAttribute
            ( subcl, attr.Name, attr.Type, attr.NN, out newattr )
      & AppendLast ( subcl, newattr )
      & newattr_ := newattr
      & cl.no := 1
      & subcl.no := 2
      & attr.no := 3
      & IncrementOOSchemaActuality ( out act )
      & subcl.actuality_ := act
      & subcl.to_be_mapped := true
    end
end;

production DHM_CreateClass( STR : STRING ; out cl : Class ;
                            out ookeycon : OOKeyCon) [0:1] =
```

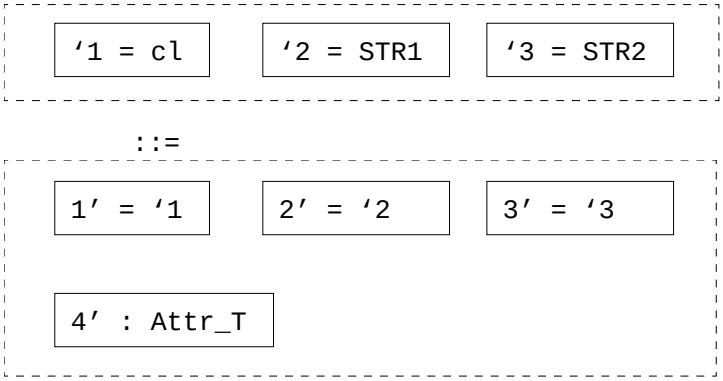


::=




```
    transfer 1’.Name := ‘1.Name; 1’.MYVISIBLE := 1; 1’.abstract := false;
             1’.to_be_mapped := true; 1’.no := 1; 2’.no := 2;
    return cl := 1’; ookeycon := 2’;
end;
```

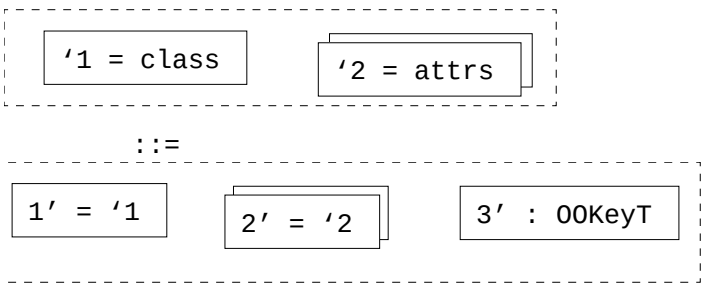
```
production DHM_CreateAttr( cl : Class ; STR1, STR2 : STRING ;
                           out attr : Attr)[0:1] =
```




```
    transfer 4’.Name := STR1.Name; 4’.Type := STR2.Name;
             4’.NN := true; 1’.no := 1; 4’.no := 2;
    return attr := 4’;
end;

transaction SetTotal( Card : TravPath) [0:1] =
    use rel_ship : Relationship;
        act : integer
    do
        rel_ship := Card.-of->
      & Card.NN := true
      & Card.no := 1
      & choose
        when empty ( rel_ship.<-r_map-.instance of MapRIND )
        then
            skip
        else
            rel_ship.to_be_killed := true
        end
      & IncrementOOSchemaActuality ( out act )
      & rel_ship.actuality_ := act
    end
end;
```

Für `SetPartial` wird `Card.NN := false` gesetzt. Bei `SetOne` wird `Card.many := false` bzw. bei `SetMany` `Card.many := true` zugewiesen.

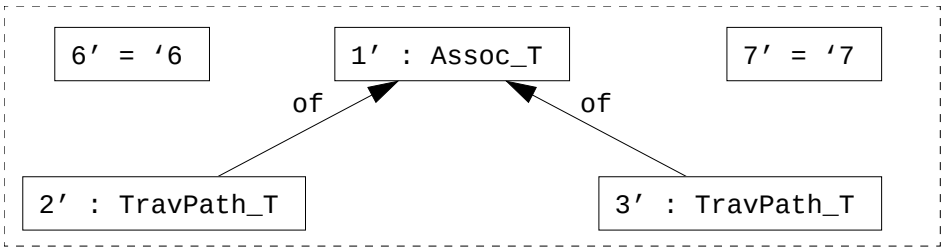
```
production DHM_CreateOOKey( class : Class ; attrs : Attr [1:n] ;
                            primary : boolean ; out k : OOKey) [0:1] =
```



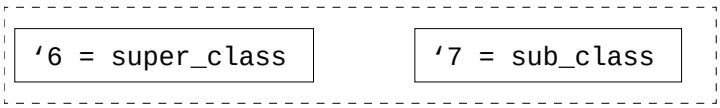

```
    transfer 3’.prim := primary; 1’.no := 1; 2’.no := 2; 3’.no := 3;
    return k := 3’;
end;
```

```
production DHM_CreateAssoc( cl1, cl2 : Class ; out assoc : Assoc ;
                            out tr1, tr2 : TravPath)[0:1] =
```

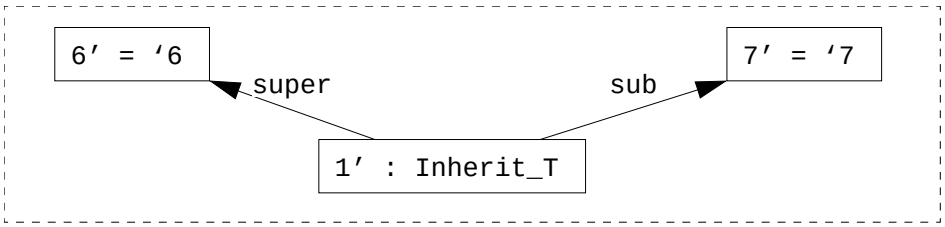
::=




```
    transfer 2’.NN := true; 2’.many := false; 3’.many := false;
             3’.NN := true; 6’.AssocCounter := ‘6.AssocCounter + 1;
             7’.AssocCounter := ‘7.AssocCounter + 1;
             3’.Name := ‘7.Name & "_" & string ( ‘7.AssocCounter );
             2’.Name := ‘6.Name & "_" & string ( ‘6.AssocCounter );
             1’.to_be_mapped := true; 6’.no := 1;
             7’.no := 2; 1’.no := 3; 2’.no := 4; 3’.no := 5;
   return assoc := 1’; tr1 := 2’; tr2 := 3’;
end;

production DHM_GeneralizeExisting( super_class, sub_class : Class ;
                                   out inherit : Inherit) [0:1] =
```

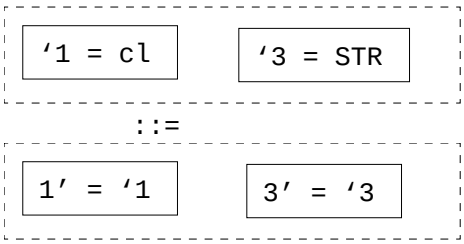


::=

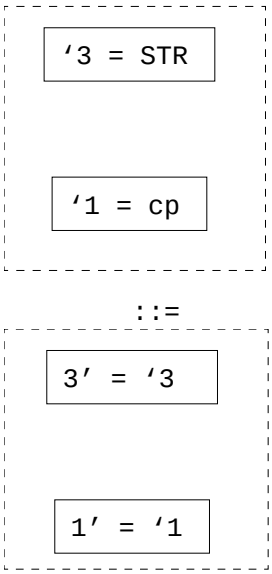



```
    transfer 1’.to_be_mapped := true; 6’.no := 1; 7’.no := 2; 1’.no := 3;
    return inherit := 1’;
end;

transaction ChangeType( attr : Attr ; STR : STRING) [0:1] =
    use cl : Class;
        act : integer
    do
        attr.Type := STR.Name
      & attr.no := 1
      & IncrementOOSchemaActuality ( out act )
      & cl := (attr.ToEnvList) : Class [0:1]
      & cl.actuality_ := act
      & cl.to_be_mapped := true
    end
nd;
```

```
production DHM_RenameClass( cl : Class ; STR : STRING) [0:1]
      =
```

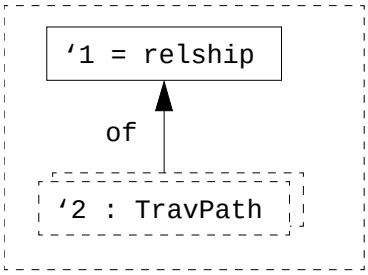



```
    transfer 1’.Name := ‘3.Name; 1’.no := 1;
end;

production DHM_RenameClassProp( cp : ClassProp ; STR : STRING) [0:1] =
```




```
   transfer 1’.Name := ‘3.Name; 1’.no := 1;
end;

transaction DHM_RmClass( cl : Class) [0:1] =
    use sc : Schema;
        ookeycon : OOKeyCon;
        ookeys : OOKey [0:n]
    do
        sc := (cl.<-item-.( instance of LITEMT ).<-lc-) : Schema [0:1]
      & RmFromList ( sc, cl )
      & ookeycon := (cl.<-ookeys_of-) : OOKeyCon [0:1]
      & ookeys := ookeycon.lc_item.( instance of OOKey )
      & for all ookey := ookeys
        do
          for all attr := ookey.lc_item
          do
              RmFromList ( ookey, attr )
          end
        & RmFromList ( ookeycon, ookey )
        & RemoveIncrement ( ookey )        end
      & RemoveIncrement ( ookeycon )
      & for all attr := cl.lc_item
        do
            RmFromList ( cl, attr )
          & RemoveIncrement ( attr )
        end
      & RemoveIncrement ( cl )
    end
end;
```

```
transaction DHM_RmAttr( attr : Attr) [0:1] =
    use cl : Class;
        k : OOKey
    do
        cl := (attr.<-item-.( instance of LITEMT ).<-lc-) : Class [0:1]
      & RmFromList ( cl, attr )
      & choose
            k := (attr.<-item-.( instance of LITEMT ).<-lc-) : OOKey [0:1]
        & RmFromList ( k, attr )
        else
            skip
        end
      & RemoveIncrement ( attr )
    end
end;

production DHM_RmRelationship( relship : Relationship) [0:1] =
```




```
        ::=
```

```
end;
```