Universität Leipzig
Institut für Informatik
Abteilung Datenbanken

UNIVERSITÄT LEIPZIG

Seminar: Schema Evolution

Model Management für Schema-Evolution

Wintersemester 2005 / 2006

Autor: Daniel Trabold (Mat.-Nr. 9180264)
Studiengang: Diplominformatik (5. Semester)

Betreuerin: Dipl.-Inf. Sabine Maßmann
Dozent: Prof. Dr. Erhard Rahm

Eingereicht am: 27.01.2006

Inhaltsverzeichnis

1 Einführung

Häufig müssen konzeptionelle Schemata einer Datenbank, eXtended Markup Language Document Type Definitions oder Ontologien im Laufe ihrer Nutzung geändert werden, um veränderten Anforderungen Rechnung zu tragen. Die Änderung eines Schemas von einer Version zu einer darauf aufbauenden Folgeversion wird als Schemaevolution bezeichnet. Die Anpassung erfolgt häufig manuell durch Programmierer, welche die notwendigen Änderungen vornehmen müssen [ME05], da bisher geeignete Software fehlt, diesen Vorgang generisch zu unterstützen. Bei komplexen Änderungen ist dieser Ansatz nicht nur sehr zeitaufwendig, sondern auch fehleranfällig, so dass zwangsweise umfangreiche Tests durchgeführt werden müssen, um die korrekte Funktionalität eines auf diese Weise geänderten Schemas zu prüfen. Je komplexer die Änderungen, desto größer wird der Aufwand und die Gefahr, Fehler zu übersehen.

Der Programmieraufwand wird durch den Einsatz geeigneter Software verringert. Das automatische Durchführen von Änderungen reduziert zudem die Zahl der möglichen Fehler. Der Ansatz des Model-Managements versucht den Anteil manueller Arbeit bei derartigen Änderungen so weit wie möglich zu reduzieren.

Um Änderungen an allen Formen von Schemata zu unterstützen, ist eine gewisse Abstraktion notwendig. Diese für eine allgemeine Lösung notwendige Abstraktion leisten Modelle. Aber auch mit Modellen lassen sich nicht alle Aspekte automatisieren. Insbesondere bei Entscheidungen über Ähnlichkeiten von Attributen, einer sehr zentralen Frage in diesem Rahmen, können die Algorithmen keine menschlichen Entscheidungen ersetzen. Einige Operationen laufen daher zwangsweise nur semiautomatisch.

Modelle lassen sich durch Bäume realisieren, um Schemata in einer von der Anwendungsdomäne unabhängigen Weise zu repräsentieren. Wir werden sehen in wie weit eine Automatisierung der notwendigen Anpassungen möglich ist. Dazu wird zunächst an einem Beispiel in Kapitel 2 das Konzept der Schema-Evolution dargestellt. Anschließend werden Modelle und weitere notwendige Strukturen erläutert. Dies erfolgt in Kapitel 3. In Kapitel 4 folgt eine Beschreibung der notwendigen Operatoren. In Kapitel 5 geht es um Einsatzmöglichkeiten von Modelmanagementlösungen, nicht nur im Bezug auf Schema-Evolution. In Kapitel 6 wird ein Prototyp vorgestellt, an dem der derzeitige Stand der Forschung und auch die heutigen Grenzen auf dem Gebiet des Model-Managements deutlich werden. Es folgt eine Zusammenfassung in Kapitel 7.

2 Schema-Evolution

Schemata werden je nach Kontext sehr unterschiedlich repräsentiert. So gibt es DTDs, Ontologien und Datenbankschemata zur Beschreibung von Datenstrukturen. Im Folgenden werden wir das Prinzip der Schema-Evolution am Beispiel eines Datenbankschemas erläutern. Die Überlegungen gelten analog für andere Formen von Schemata.

Ein Datenbankschema beschreibt das Verhalten einer Datenbank. Dieses umfasst unter Anderem folgende Teile [TÜ00]:

- Typen, Tabellen und Sichten
- Einschränkungen und Assoziationen
- Funktionen, gespeicherte Funktionen und Trigger

Die SQL-Definitionssprache (SQL-DDL) erlaubt insbesondere das Hinzufügen und Entfernen von Attributen, sowie deren Umbenennung in einer Tabelle. Wie in der Einleitung bereits erwähnt, wird die Änderung eines Schemas als Schema-Evolution bezeichnet. Dabei reicht bereits die Änderung eines einzelnen Attributs, um von Schema-Evolution zu sprechen.
Meist kommt es während der Nutzungsdauer eines Systems zu mehr als einer Änderung. Eine Messung von Änderungshäufigkeiten in einem Gesundheitssystem deckte auf, dass in dem Beobachtungszeitraum von 9 Monaten die Zahl der Relationen in der Datenbank um 139% anwuchs. Im selben Zeitraum vermehrten sich die Attribute um 274% [SJ93]. Leider fehlen vergleichbare Daten in größerem Umfang. Für den Einzelfall wird dennoch deutlich, dass es ein großes Einsparungspotenzial gibt, das durch geeignete Lösungen erschlossen werden kann.

Im Folgenden wird statt des Begriffs Schema-Evolution zur Vereinfachung nur der Begriff Evolution verwendet.

Bevor wir mögliche Lösungen betrachten, soll der Vorgang der Evolution anhand eines Beispiels verdeutlicht werden. In Abb. 1a wird eine mögliche Tabelle für ein Produkt mit seinen Attributen dargestellt. Zu dessen Beschreibung werden eine eindeutige ProduktID, der Name, sowie eine Kategorie gespeichert. Wird zusätzlich die Information benötigt, von welchem Hersteller das Produkt stammt, muss ein weiteres Attribut hinzugefügt werden. In Abb. 1b wurde HerstellerID hinzugenommen, um den Hersteller des Produktes zu identifizieren.

Produkt
ProduktID
Name
Kategorie

Produkt
ProduktID
Name
Kategorie
HerstellerID

Abb1 (a) Schema für Produkt (b) erweiterte Schema für Produkt

3 Strukturen

3.1 Modelle

In Kapitel 2 wurde das Prinzip der Evolution erläutert. In Abb. 2 wird das Schema aus dem Beispiel von Abb. 1a als XML-Schema dargestellt.

```
<schema xmlns="…">
        <complexType name="Produkt">
                <element name="ProduktID" type="xs:int"/>
                <element name="Name" type="xs:string"/>
                <element name="Kategorie" type="xs:string"/>
        </complexType>
</schema>
```

Abb.2 XML-Schema für Produkt wie in Abb. 11 dargestellt

Bei einem Vergleich von Abb. 1a und 2 wird deutlich, dass in der Form Repräsentation große Unterschiede bestehen. Daher ist eine abstrakte Repräsentation für Schemata wünschenswert. Ohne eine allgemeine Darstellungsform müssten Operatoren und Funktionen mehrfach implementiert werden. Bei einer Darstellungsform, die sich aus allen Schemata generieren lässt und aus der sich Schemata wieder erzeugen lassen, reicht eine Implementierung. Zusätzlich müssen entsprechende Konverter bereitgestellt werden (Vgl. 4.2.6). Modelle beschreiben die Attribute eines Schemas in allgemeiner Form und können aus DTDs, Datenbankschemata, Ontologien etc. generiert werden. Melnik [ME05] bezeichnet Datenbankschemata, Ontologien und dergleichen selbst als Modelle. Obgleich es sich zweifelsfrei um Modelle handelt, wollen wir den Begriff hier enger fassen und den Begriff Modell bzw. allgemeines Modell nur für eine Form verwenden, in die sich alle vorgenannten Modelle konvertieren lassen und aus dem wir wieder solche erzeugen können.

Um sprachlich zwischen den beiden Formen zu differenzieren, werden wir Ontologien, DTDs, Datenbankschemata und dergleichen als konkrete Modelle bezeichnen.

Modelle bestehen ganz allgemein aus Objekten. Um aus Objekten komplexere Strukturen erstellen zu können, müssen diese gruppierbar sein. Objektgruppierungen sollen die Mengeneigenschaften erfüllen. Es soll insbesondere immer klar sein, ob ein gegebenes Objekt zu einer Menge von Objekten gehört oder nicht. Durch Assoziation, Aggregation und Generalisierung entstehen aus Objekten schließlich Modelle. Jedes Modell m hat ein Wurzelelement w, welches m identifiziert. Jedes Objekt, das von w aus erreichbar ist, gehört zu m.

Definition: Ein Modell m ist eine Menge von Objekten o mit ausgezeichnetem Wurzelelement w, so dass jedes o ≠ w, von w aus erreichbar ist.

Nicht nur ein Modell soll eindeutig identifizierbar sein, sondern auch jedes Objekt muss über eine Identität verfügen. Dies ist notwendig, um Objekte ändern zu können und um zwischen verschiedenen Objekten eine eindeutige Beziehung (Mapping oder Morphismus Vgl. 3.2) herzustellen. Zudem brauchen Objekte Eigenschaften, in denen sich beispielsweise der Datentyp speichern lässt.

In [BLP00] werden Modelle als „beschriftete gerichtete Graphen“ beschrieben. Andere Darstellungsformen sind möglich. Bernstein [BE03] suggeriert durch die Abbildungen eine Repräsentation als Baum. Wir werden für die weitere Betrachtung Modelle als Bäume modellieren ohne die Eigenschaften mit zu notieren, weil diese Darstellungsform übersichtlicher ist. In Abb. 3 wird das Produkt aus Abb. 1a als Modell wiedergegeben.

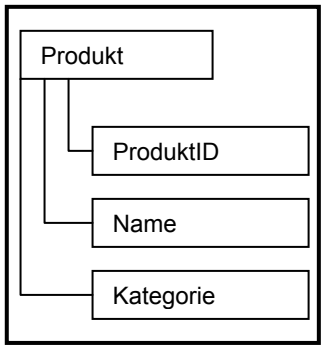


Abb. 3 Modell für Produkt, wie in Abb. 1a

Modelle sind keineswegs statisch und müssen änderbar sein. Dazu ist es notwendig, dass sich einerseits Objekte direkt manipulieren lassen, um Attribute zu ändern, einzelne Objekte zu entfernen oder neue hinzuzufügen. Andererseits müssen sich Modelle als Ganzes manipulieren lassen, sonst erleichtern sie die Arbeit der Anwender nicht. Operationen auf Objektebene sind bereits heute an der Tagesordnung bei der Anpassung von Datenmodellen in Antwort auf evolutionäre Anforderungen.

Für diese Manipulationen sind verschiedene Operatoren notwendig, die in Kapitel 4 vorgestellt werden. Ein Großteil der Operatoren ist darauf angewiesen, Gemeinsamkeiten verschiedener Modelle zu kennen. Solche werden durch Mappings oder Morphismen abgebildet.

3.2 Morphismen und Mappings

Ein Morphismus ist eine binäre Relation zwischen zwei Objekten unterschiedlicher Modelle. Für einen Morphismus ist es nicht notwendig, dass die beiden Modelle der engen Form der Definition genügen. Ein Morphismus kann auch zwischen unterschiedlichen Modellen, beispielsweise einem Datenbankschema und einer DTD, hergestellt werden [MRB03].

Ein Mapping setzt zwei Modelle m_1, m_2 zueinander in Beziehung. Es besteht aus zwei Morphismen und einem Modell map. Die Modelle m_1 und m_2 müssen vom

selben Typ sein, da sonst nicht deutlich wird, von welchem Typ das Mapping ist. Die meisten Objekte in map sind durch mindestens einen der beiden Morphismen mo_1, mo_2 mit einem Objekt in einem der Modelle m_1 oder m_2 verbunden. Für das Mapping zwischen den Objekten aus m_1 und map sorgt mo_1, mo_2 verbindet die Objekte von map mit m_2. Abb. 4 veranschaulicht das Konzept schematisch. Nicht jedes Objekt aus map muss zu Objekten in beiden Modellen einen Morphismus haben. Es gibt Objekte, die weder mit m_1 noch mit m_2 verbunden sind. Solche Objekte dienen der Erhaltung der Ausdrucksstärke und werden nur dann in das Mapping aufgenommen, wenn auf dem Pfad eines Objektes o, das zu dem Mapping gehört, und dem Wurzelelement w liegen.

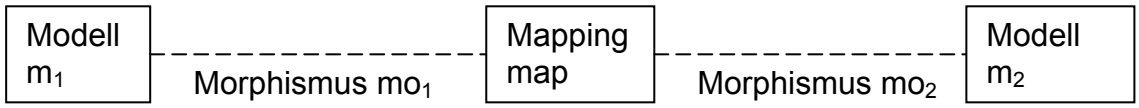


Abb. 4 Schematisch Darstellung eines Mapping zwischen zwei Modellen

In Abb. 5 ist ein beispielhaftes Mapping zu sehen. In $Produkt_1$ gibt es ein Attribut Kategorie, welches in $Produkt_2$ durch Haupt- und Unterkategorie ersetzt wurde. Die Beziehung wird in dem Mapping map_{12} deutlich. Eine Zusammensetzung des Attributes Kategorie aus Haupt- und Unterkategorie kann nur durch Mappings ausgedrückt werden. Man kann sich leicht vorstellen, dass diese Ausdrucksstärke insbesondere bei komplexen Modellen notwendig ist. Diese Vorstellung deckt sich mit den Erfahrungen von Bernstein [BE03]. Morphismen fehlt die notwendige Ausdrucksstärke um solche Situationen zu beschreiben. Allerdings haben letztere den Vorteil, dass sie sich leichter implementieren lassen und immer invertierbar sind [MRB03].

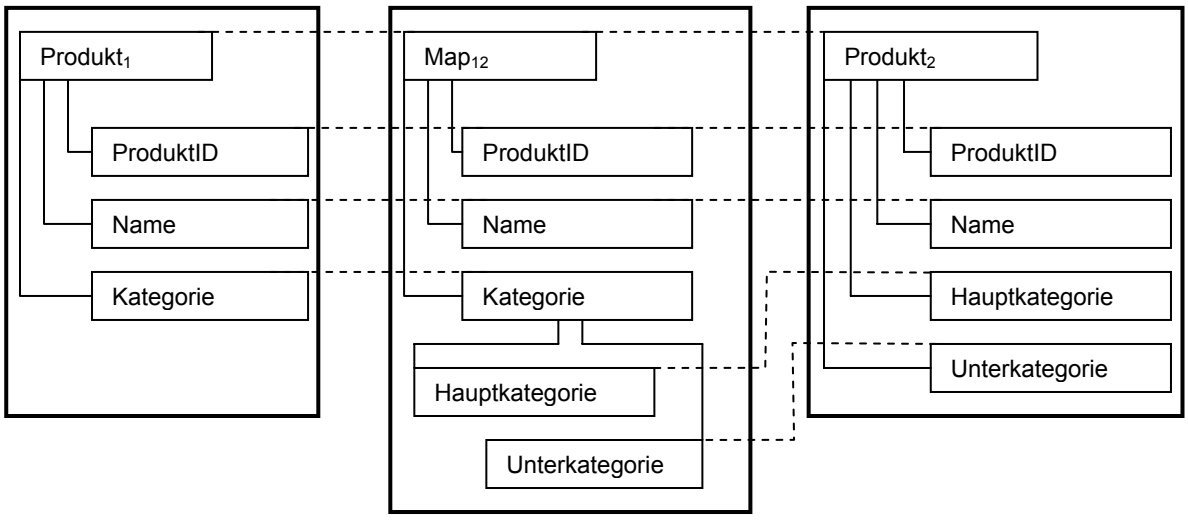


Abb. 5 Mapping Map_{12} zwischen zwei Modellen $Produkt_1$ und $Produkt_2$

4 Operatoren

Die Bezeichnung der Operatoren in der Literatur ist nicht immer identisch. Insbesondere findet sich für den Operator `Select` auch die Bezeichnung `Extract`. Da sich die Beschreibung stark an [BE03] orientiert, wird hier der Begriff `Select` verwendet.

4.1 Einfache Operatoren

4.1.1 Apply

Der Operator `Apply` wendet eine gegebene Funktion f auf alle Objekte o eines Modells m an.

4.1.2 Copy

Für den Operator `Copy` gibt es in der Literatur verschiedene Vorschläge. Vgl. [BE03] und [BLP00]. Die in [BE03] vorgestellte Variante erstellt eine Kopie eines Modells m inklusive aller Attribute a und Relationen r. Dabei werden auch solche Relationen kopiert, die Objekte o aus m mit Objekten außerhalb des Modells verbinden. In [BLP00] wird über ein Flag gesprochen, welches anzeigt, ob das referenzierte Attribut kopiert werden soll. Dies erlaubt ein feineres Steuern des Operators, verursacht auf der anderen Seite jedoch mehr Aufwand durch Setzen des Flags. Zudem wird eine Funktion namens `DeepCopy` vorgestellt, diese kopiert ein Modell und ein zu diesem gehörendes Mapping. Ein Mapping gehört zu einem Modell, wenn es dessen Objekte referenziert.

4.1.3 Delete

Der Operator `Delete` löscht alle Objekte eines Modells, bis auf jene, die Teil eines anderen Modells sind. Man beachte, dass auch Mappings Modelle sind.

4.1.4 Domain

Der Operator `Domain` arbeitet auf Mappings und liefert ein Modell, in dem nur die Elemente des „linken“ Eingabe-Modells vorkommen, auf die in dem Mapping referenziert wird. Möglicherweise sind zusätzlich Supportobjekte s notwendig, falls s auf einem Pfad zwischen Wurzelelement w und einem Objekt o der Ergebnismenge liegt, damit das Modell nichts von seiner Aussagekraft verliert.

Diese zusätzlichen Objekte, die eigentlich nicht zum Ergebnis zählen, müssen auf eine Art gekennzeichnet werden. Wie dies erfolgen kann, ist exemplarisch anhand des Operators `Diff` weiter unten in diesem Text beschrieben. In Abb. 6 ist links ein Modell $Person_A$ und ein Mapping Map_{AB}, zwischen diesem und einem weiteren, in der Grafik nicht gezeigten $Modell_B$ dargestellt. Auf der rechten Seite der Abbildung ist das Ergebnis $Model_{A'}$ der Funktion `Domain`(Map_{AB}) zu sehen. Das Element Adresse in A’ gehört nicht zum Ergebnis, wird aber gebraucht, um die Ausdrucksstärke des Modells zu erhalten. Auf die Kennzeichnung wurde an dieser Stelle verzichtet.

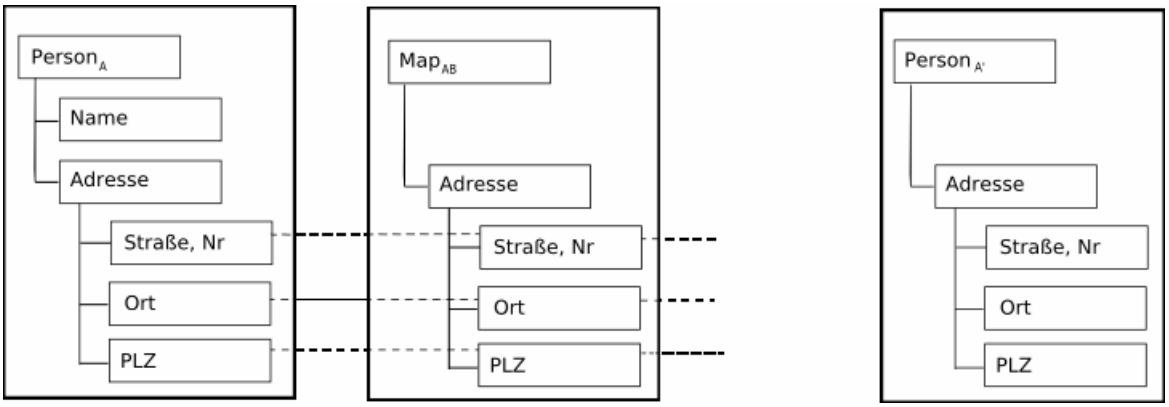


Abb. 6 Links sind Person und Map_{AB} zu sehen, rechts das Ergebnis der Funktion `Domain`(Map_{AB}), ohne Kennzeichnung. Man beachte, dass Adresse selbst nicht zum Ergebnis zählt.

4.1.5 Enumerate

Der Operator `Enumerate` liefert einen Cursor, der mit der Operation next das jeweils nächste Objekt des Modells liefert oder null, falls das Ende des Cursors erreicht wurde. Dieser Operator ist für Fälle gedacht, in denen viele Objekte eines Modells geändert werden müssen.

4.1.6 Range

Analog zu `Domain` liefert `Range` alle Objekte o des „rechten“ Modells eines Mappings map, die durch map referenziert werden.

4.1.7 Weitere

Neben den bisher vorgestellten Operatoren gibt es eine Reihe weiterer Funktionen, die weniger bedeutend sind, da sie sich entweder als Spezialfälle der oben beschriebenen Operatoren ergeben oder seltener gebraucht werden. Sie sollen hier kurz vorgestellt werden. Einen guten Überblick über die Operatoren biete [MRB03].

`RestrictDomain`, `RestrictRange` und `Subgraph` können als Spezialfälle des weiter unten beschriebenen Operators `Select` betrachtet werden. `RestrictDomain`(Map, OM) liefert eine Kopie von Map, in der nur Objekte vorkommen, deren `Domain`, in der übergebenen Objektmenge OM vorkommt. `RestrictRange` ist analog für die Beschränkung einer Map über deren `Range` definiert. `Subgraph` ist ein `Select` für den Fall, dass Modelle als Graphen repräsentiert werden.

`Restrict`(Map, M_1, M_2) wendet sowohl `RestrictDomain` als auch `RestrictRange` an. Dies führt dazu, dass nur Objekte des Mappings map gewählt werden, die sowohl in M_1 als auch in M_2 referenziert werden.

`Invert` vertauscht `Domain` und `Range` eines Mappings.

`Transitive Closure` berechnet die transitive Hülle eines Mappings.

`Id`(Objekte) erzeugt ein Mapping, dessen `Domain` und `Range` identisch sind.

`Traverse`(Map, Objekte) ermöglicht das Traversieren eines Mappings oder einzelner Teile eines Mappings. Dazu werden mit `RestrictDomain`(Map, Objekte) die in Betracht kommenden Objekte des Mappings identifiziert. Anschließend wird deren `Range` bestimmt. Man beachte, dass dies keinesfalls trivial ist und bei Vererbungen etliche Probleme in sich birgt.

4.2 Komplexe Operatoren

4.2.1 Match

Der Operator `Match` erzeugt ein Mapping zwischen zwei Modellen. Für diese Arbeit braucht er eine Definition von Ähnlichkeit. Die Definition der Ähnlichkeit muss extern erfolgen, um hinreichend flexibel zu sein. Es werden zwei Formen des Matches unterschieden, das elementare und das komplexe `Match`.

Das elementare `Match` kann immer dann eingesetzt werden, wenn Objekte einfachen Ähnlichkeitskriterien genügen. Dies ist z.B. dann der Fall, wenn sie gleiche Namen haben. Sehr wahrscheinlich sind gleiche Namen, wenn ein Schema weiter entwickelt wurde. Es kann aber auch sein, dass Namen bewusst geändert wurden. Wenn das Wissen über die Gleichheit extern gespeichert wird, kann dort die Definition angepasst werden und der Algorithmus auch in diesem Fall verwendet werden.

Komplexes `Match` geht weiter und muss dann eingesetzt werden, wenn keine einfache Ähnlichkeit gegeben ist. Dies ist bei Schema-Integration sehr wahrscheinlich, kann aber auch bei Schema-Evolution auftreten, wenn Objekte in Teilobjekte zerlegt, oder aus diesen zusammengesetzt werden. Komplexes `Match` soll unterscheiden können zwischen Objekten, die gleich sind und solchen, die sich nur ähnlich sind. In Abb. 4 sind sich Kategorie einerseits und Hauptkategorie, Unterkategorie ähnlich. Alle anderen Objekte sind gleich. Die Beziehungen zwischen ähnlichen Objekten können nicht automatisch geklärt werden. An dieser Stelle wird nach wie vor ein Anwender eingreifen müssen.

Analog zu Join Operationen in Relationalen Datenbanksystemen gibt es einen `Left-` bzw. `RightOuterMatch`, der Objekte eines Modells M_1 bzw. M_2 auch dann in map_{12} aufnimmt, wenn die Objekte nur einseitig referenziert werden und es in dem anderen Modell kein entsprechendes Objekt gibt.

4.2.2 Merge

Der Operator `Merge` arbeitet mit zwei Modellen und einem Mapping als Eingabe und gibt eine Kopie beider Modelle zurück, in der identische Objekte nur einfach vorkommen. In Abb. 7 ist das Ergebnis $Produkt_3$ eines Merges zwischen $Produkt_1$ und $Produkt_2$ zu sehen.

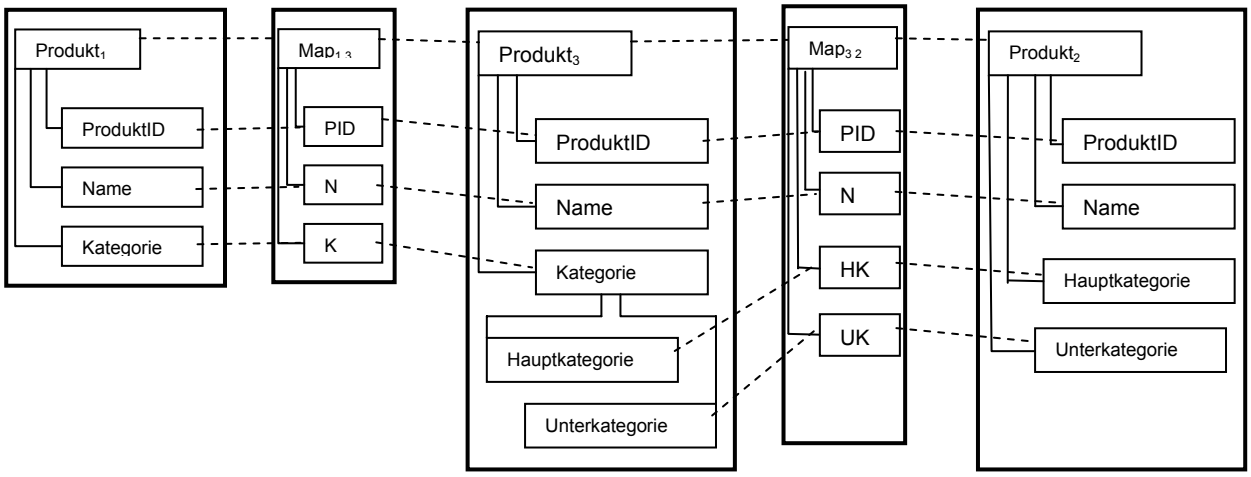


Abb. 7 Ergebnis von `Merge`(Produkt$_1$, Produkt$_2$, Map$_{1\,2}$)

Die allgemeine Form von `Merge` lautet `Merge`(m_1, m_2, map) und ergibt m_3, map_{13}, map_{32} als Ergebnis. Der Operator lässt sich nur anwenden, wenn die Wurzelelemente von m_1 und m_2 gleich sind.

In m_3 kommen alle Objekte aus m_1 und m_2 vor. Objekte aus m_1 und m_2, die durch map als identisch ausgewiesen werden, existieren nur einmal in m_3. Alle Relationen aus m_1 und m_2 sind vollständig in m_3 enthalten. Hat ein Objekt o in m_1 und m_2 unterschiedliche Attribute und Relationen, so weist o alle Attribute und Relationen in m_3 auf. Map_{13} referenziert alle Objekte in m_1 und jene in m_3, die aus m_1 kopiert wurden. Analog ist map_{32} definiert. Für eine Diskussion eventuell auftretender Integritätsprobleme sei dem interessierten Leser [LNE89] empfohlen.

4.2.3 Diff

`Diff` bestimmt die sich voneinander unterscheidenden Objekte zweier Modelle. Man könnte auf die Idee kommen, `Diff` daher auf zwei Modelle anzuwenden. Um festzustellen, welche Objekte sich voneinander unterscheiden, kann man aber auch all jene Objekte nehmen, die nicht gleich sind. Da `Match` die Gleichheit von Objekten feststellt, wird `Diff` so definiert, dass er mit dem Ergebnis von `Match` arbeiten kann. Daher wird `Diff` auf ein Modell und ein Mapping angewendet und liefert als Ergebnis ein Modell und ein Mapping. Es ist aber nicht zwingend erforderlich, dass im Ergebnis auch ein Mapping geliefert wird.

`Diff`(m, map) liefert als Ergebnis all jene Objekte aus m, die nicht durch einen Morphismus mo, mit map verbunden sind. Dies liefert allerdings kein gültiges Modell. Deswegen werden zusätzliche Objekte im Ergebnis benötigt, um immer ein Modell zu haben. Diese zusätzlichen Objekte, die eigentlich nicht zum Ergebnis zählen müssen, auf eine Art gekennzeichnet werden. Denkbar wäre, jedes Objekt durch einen boolschen Wert dahingehend zu kennzeichnen, ob es ein Objekt des Modells ist, oder nur ein Hilfsobjekt, das nur für die Struktur gebraucht wird. Um dies zu vermeiden, wird ein Mapping verwendet.

Um ein valides Modell zu erhalten, müssen alle Objekte o hinzugenommen werden, die auf einem Pfad von dem Wurzelelement w zu einem Objekt der Ergebnismenge oe liegen. Auch w muss in dem von `Diff` erzeugten Modell vorkommen.

Objekte der Ergebnismenge oe werden durch das Mapping des Operators `Diff` mit den korrespondierenden Objekten in m verbunden und sind auf diese Weise markiert. Hilfsobjekte werden nicht von dem Mapping erfasst und verfügen folglich über keine Markierung. Abb. 8 verdeutlicht das Ergebnis <$Produkt_3$', $Map_{33'}$> der Operation `Diff`($Produkt_3$, Map_{13}). *Hauptkategorie* und *Unterkategorie* in $Produkt_3$' sind direktes Ergebnis der Operation und werden deswegen durch $Produkt_{map}$' referenziert. *Kategorie* ist ein Hilfsobjekt und wird deswegen nicht durch das Mapping $Map_{33'}$ referenziert.

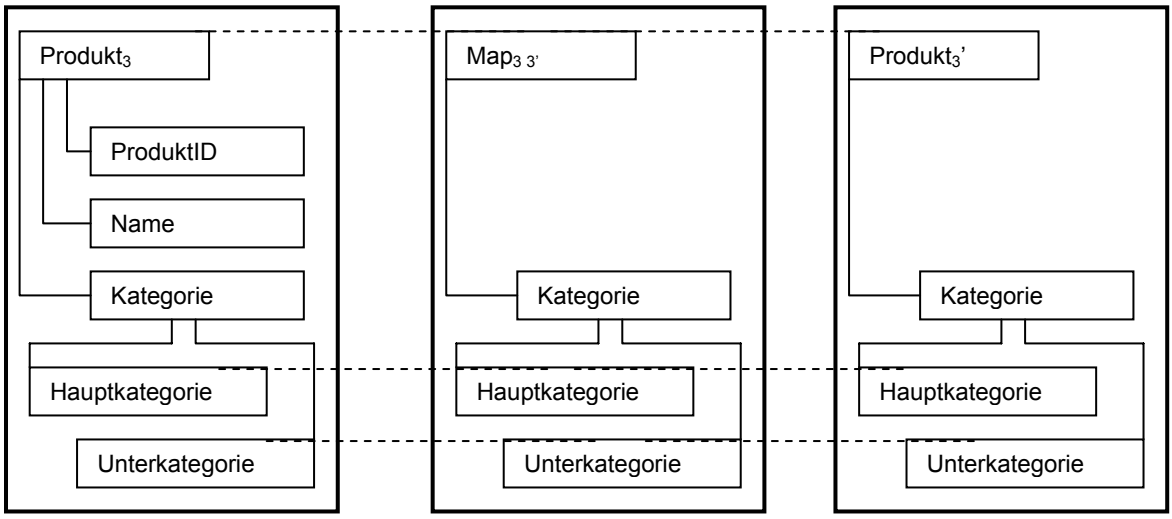


Abb. 8 `Diff`($Produkt_3$, $Map_{1\,3}$) liefert das Ergebnis <$Produkt_3$', $Map_{3\,3'}$>.

4.2.4 Select

Der Operator `Select` erzeugt als Ergebnis eine Teilkopie eines Modells, in dem alle Objekte o vorkommen, die einem gegebenen Kriterium genügen. `Select`(M, Anfrage) liefert wie `Diff` ein Modell und ein Mapping zurück. Durch das Mapping werden, wie bei `Diff`, Objekte der Ergebnismenge und solche, die nur zur strukturellen Ergänzung hinzugenommen wurden, unterschieden.

4.2.5 Compose

`Compose` erzeugt aus einem Mapping zwischen Modell M_1 und M_2 und einem Mapping zwischen Modell M_2 und M_3 ein Mapping zwischen den Modellen M_1 und M_3.

Um die Funktion von `Comopse` zu erläutern, führen wir zunächst noch zwei Begriffe ein. Für jedes Objekt m_1 aus map_1 ist `domain`(m_1) die Menge der Objekt

auf die in M_1 referenziert wird, range(m_1) sind die in M_2 referenzierten Objekte. Beispielsweise ist in Abb. 9 domain(4) = {1} und range(4) = {7,8}.

Eine Komposition zweier Mappings lässt sich auf zwei unterschiedliche Arten durchführen. Diese werden „linke“ beziehungsweise „rechte Komposition“ genannt. Der Name gibt an, welche Map der beiden Maps als Vorlage für die Ergebnismap dient. Wir werden die rechte Komposition betrachten. D.h. die Objekte der Ergebnismap map_3 ergeben sich als Kopie von map_2. Bei der linken Komposition würden sich die Objekte von m_3 aus einer Kopie von m_1 ergeben.

Bei der rechten Komposition wird zunächst map_2 kopiert. Diese Kopie nennen wir map_3. Domain und range jedes Objektes in map_3 stimmen mit denen in map_2 überein. Da wir die rechte Komposition betrachten, stimmt range aller Objekte in map_3 bereits mit dem gesuchten Ergebnis überein. Die Domain der Objekte ist hingegen noch zu ändern.

Für jedes Objekt m_3 in map_3 sei Input(m_3), die Menge aller Objekte m_1 aus map_1, so dass range(m_1) ∩ domain(m_3) ≠ Ø. In Abb. 9 ist Input(13) = {4,5,6}, denn domain(13) = domain(10) = {7,9}, range(4) = {7,8}, range(5) = {8,9} und range(6) = {9}.

Domain(m_3) wird folgendermaßen definiert.

$$domain(m_3) = \begin{cases} \bigcup_{m_{1i} \in Input(m_3)} domain(m_{1i}), \text{ if } Input(m_3) \neq \emptyset \\ \emptyset \end{cases}$$

In dem Beispiel ist domain(13) nach diesem Schritt ∪ {domain(4), domain(5), domain(6)} = {1,2,3}. Für alle Objekte m_3 mit domain(m_3) = Ø ist zu prüfen, ob es ein Objekt o zwischen m_3 und root von map_3 gibt, so dass domain(o) ≠ Ø. Ist dies nicht der Fall, muss m_3 gelöscht werden.

In der folgenden Abbildung wird die rechte Komposition graphisch veranschaulicht.

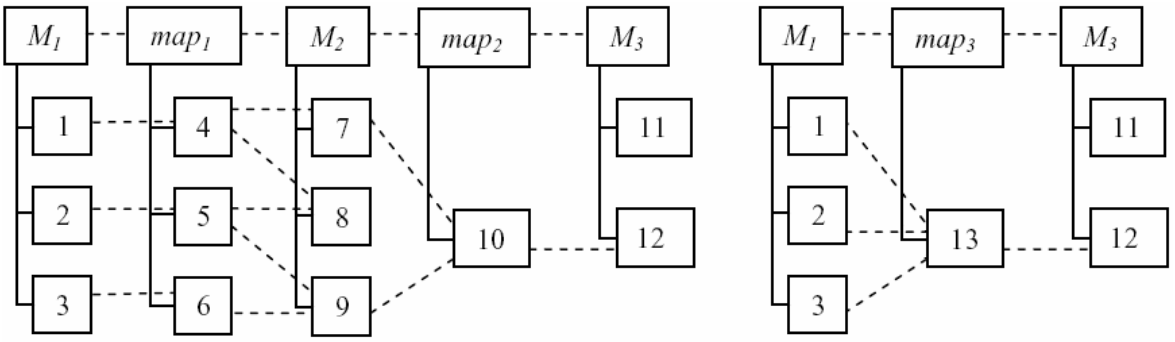


Abb. 9 Komposition zweier Mappings. Adaptiert nach [BE03].

4.2.6 ModelGen

Der Operator `ModelGen` wird in [MRB03] vorgestellt. Er dient der Transformation eines Modells von einer Repräsentationsform in eine andere. Wir hatten den Begriff des Modells in Kapitel 3 sehr eng gefasst und die Frage, wie man von einer Modellform zu einer anderen kommt, zunächst offen gelassen. Die notwendige Transformation leistet `ModelGen`. Der Operator soll nicht nur eine Modellform in eine andere konvertieren können, sondern auch ein Mapping zwischen Ein- und Ausgabemodell liefern. Das Mapping unterstützt die Übertragung späterer durchgeführter Änderungen in einem der Modelle von diesem auf das andere. Der Operator ist als einziger der hier vorgestellten nicht generisch.

Für jeden konkreten Modelltyp und das allgemeine Modell muss `ModelGen` separat definiert werden. Jede Definition umfasst zwei Richtungen. Einerseits müssen sich die konkreten, in realen Systemen verwendeten, Modelle in allgemeine Modelle transformieren lassen. Andererseits müssen aus der allgemeinen Form wieder konkrete Modelle werden. Eine direkte Konvertierung, eines konkreten Modells in ein anderes, wie beispielsweise von einer DTD zu einer Ontologie, ist nicht Teil des Operators.

5 Einsatzfelder

Im Folgenden sollen konkrete Beispiele vorgestellt werden, wie die unter 4. vorgestellten Operatoren für Model-Management-Aufgaben eingesetzt werden können. Im Vordergrund steht dabei die Betrachtung der Schema-Evolution.

5.1 Schema-Evolution

5.1.1 Szenario 1

Sind ein Datenbankschema und eine Menge von Sichten auf dieses gegeben, so sind bei Änderungen an dem Datenbankschema in der Regel auch die Sichten anzupassen, damit diese weiterhin verwendet werden können. Das Problem und seine Lösung finden sich in [BE03].

Sei S_1 das ursprüngliche Schema, V_1 die Menge der darauf definierten Sichten und S_2 das geänderte Schema. Nachfolgend wird aufgeführt, wie Management-Operatoren verwendet werden, um die Sichten an S_2 anzupassen. Das Problem und seine Lösung sind darüber hinaus in Abb. 10 dargestellt.

1. map_1 = `Match`(S_1, V_1). Erzeugt ein Mapping zwischen den der Menge der Sichten V_1 und dem ursprünglichen Schema.

2. map_3 = `Match`(S_1, S_2). Erzeugt ein Mapping zwischen ursprünglichem und geändertem Schema. Dieses Mapping dient dazu die nicht geänderten Objekte in S_2 zu identifizieren.

3. $map_4 = map_1 \cdot map_3$. Jedes Objekt m_1 aus map_1 ist in map_4, wenn es zu jedem Objekt s_1 aus S_1, auf das durch m_1 referenziert wird, ein Objekt s_2 in S_2 gibt, so dass $(s_1,s_2) \in map_3$. Diese Komposition beschreibt den Teil der Mappings, die durch die Änderungen in S_2 nicht betroffen sind.

Wurden in S_2 Attribute entfernt, die in V_1 verwendet wurden, finden sich in map_4 keine Referenzen mehr auf diese Attribute. Sie müssen ggf. aus den Sichten gelöscht werden. Statt dies direkt in V_1 zu machen, wird V_1 zunächst kopiert, weil sonst map_1 ungültig würde.

4. $<V_2, map_2>$ = `DeepCopy`(V_1, map_4). Kopiert V_1 und map_4.

5. $<V_2', map_5>$ = `Diff`(V_2, map_4). Identifiziert Objekte in V_2, auf die nicht durch map_4 referenziert wird.

6. Für jedes e in `Enumerate`(map_5), lösche `domain`(e) in V_2. Löscht alle in map_4 nicht mehr referenzierten Objekte aus V_2.

Man beachte, dass die Elemente in V_1 und V_2 ganze Sichten sind. Das Szenario lässt sich analog durchführen, wenn V_1 und V_2 einzelne Sichten sind. Dann sind die Schritte 1 bis 7 für jede Sicht, die auf S_1 definiert ist durchzuführen.

Damit ist das Problem gelöst. Das Hinzufügen neuer Sichten muss separat erfolgen und ist von dem Problem der Erhaltung existierender Sichten bei Änderung der Implementierung getrennt zu betrachten.

Abb. 10 Das Evolutionsproblem und seine Lösung. Aus [BE03].

5.1.2 Szenario 2

Gegeben sei eine Spezifikation eines Produktes $Produkt_A$, die implementiert wurde. Diese soll so geändert werden, dass sie mit der Spezifikation $Produkt_B$ übereinstimmt. Wir setzen voraus, dass es ein 1:1 Mapping map_1 zwischen der Spezifikation $Produkt_A$ und der Implementierung $Produkt_I$ gibt. Da die Implementierung aus $Produkt_A$ erzeugt wird, ist diese Annahme gerechtfertigt.

Ein konkretes Beispiel dieses Problems ist in Abb. 11 dargestellt. Das Attribut Kategorie in $Produkt_A$ soll zu Gruppe in $Produkt_B$ umbenannt werden. Attribut Farbe soll entfernt werden. Zudem soll ein Attribut HerstellerID, das den Hersteller des Produktes referenziert, in das Schema eingefügt werden.

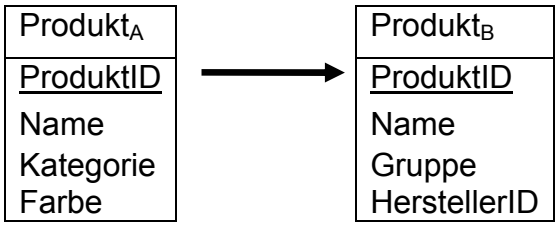


Abb. 11 Änderungsvorhaben

Die notwendigen Schritte um die Änderungen mit Model-Management-Operatoren durchzuführen, werden jetzt behandelt.

1. map_2 = `Match`($Produkt_A$, $Produkt_B$) erzeugt ein Mapping zwischen den beiden Produktmodellen um nicht geänderte Attribute zu identifizieren. Das Ergebnis des Matches map_2 ist in Abb. 12 rechts zu sehen. Da es sich bei Kategorie und Gruppe um eine reine Umbenennung auf Schemaebene handelt, müssen die Daten nicht geändert werden. Das Mapping assoziiert die beiden Objekte deswegen als Ähnlich oder Äquivalent.

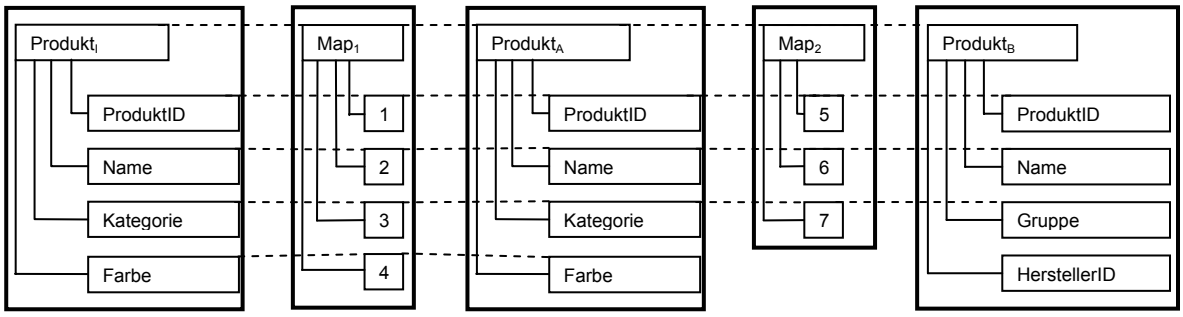


Abb. 12 Die drei Modelle (Implementierung, Spezifikation, geänderte Spezifikation) und die Mappings zwischen ihnen

2. $map_3 = map_1 \cdot map_2$. Map_3 ist die Komposition der Mappings map_1 und map_2. In ihr verbleiben alle Objekte, die sowohl in map_1 und map_2 sind. In map_1 sind alle Objekte, in map_2 nur jene, die nicht geändert wurden. Folglich stehen in map_3 nur die nicht geänderten Objekte. Das Mapping map_3 ist in Abb. 13 veranschaulicht.

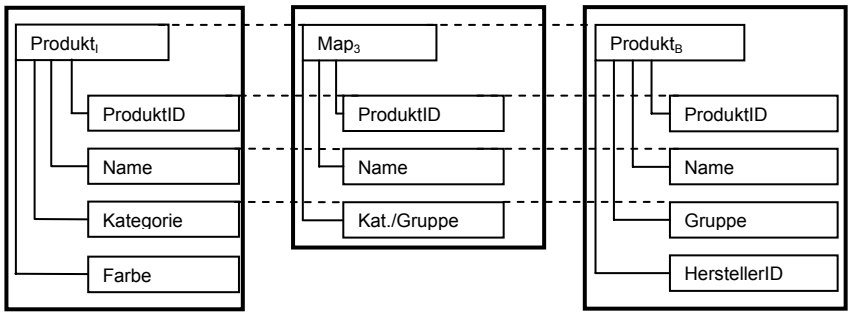


Abb. 13 Komposition map_3 der Mappings aus Abb. 12 und die durch map_3 verbundenen Modelle der Implementierung und der geänderten Spezifikation.

3. <Produkt$_{CI}$, map$_4$> = `DeepCopy`(Produkt$_I$, map$_3$) kopiert die Implementierung Produkt$_I$ und das Mapping map_3 zwischen ihr und der neuen Spezifikation Produkt$_B$. Die Kopie ist notwendig, damit map_1 und map_3 durch Änderungen an der Implementierung nicht ungültig werden.

4. Müssen alle Objekte, deren Namen in Produkt$_B$ geändert wurde, in Produkt$_{CI}$ umbenannt werden. Für diese Operation gibt es keinen Management-Operator. Die Änderung kann durch Vergleich des Namensattributes von `domain`(a_i) und `range`(a_i) für alle $a_i \in$ map3 erfolgen. Unterscheiden sich die Namen, kann das Namensattribut in Produkt$_{CI}$ geändert werden.

5. `Delete`(Produkt$_{CI}$) löscht aus Produkt$_{CI}$ alle nicht durch andere Modelle referenzierten Objekte. Dies sind alle Objekte, die in dem neuen Modell nicht mehr verwendet werden und deswegen gelöscht werden sollen. In Abb. 15 ist links Produkt$_{CI}$ nach dem Löschen entfernter Objekte zu sehen.

6. <Produkt$_{B'}$, map$_5$> = `Diff`(Produkt$_B$, map$_3$) identifiziert alle Objekte, die in Produkt$_B$ stehen und nicht durch map_3 referenziert werden. Dies sind all jene Objekte, die in Produkt$_B$ neu eingefügt wurden. Map_5 stellt die

Verbindung zwischen $Produkt_B$ und $Produkt_{B'}$ her. Das Ergebnis für das betrachtete Beispiel ist in Abb. 14 gezeigt.

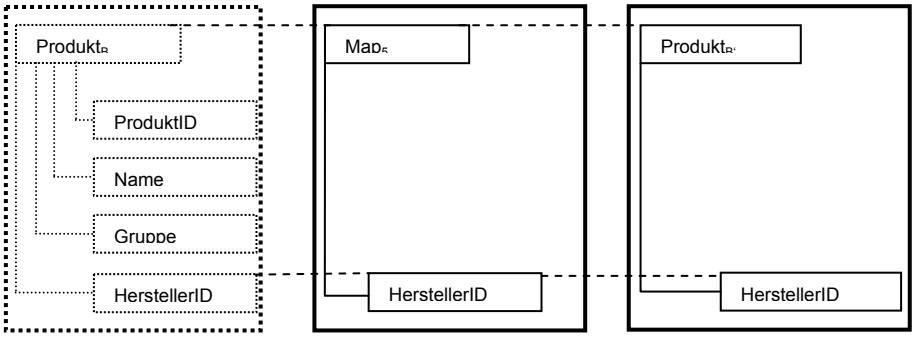


Abb. 14 Ergebnis von `Diff`($Produkt_B$, map_3) = <$Produkt_{B'}$, map_5>. $Produkt_B$ wurde zur besseren Lesbarkeit mit abgebildet.

7. <$Produkt_{Cl'}$, map_7, map_8> = `Merge`($Produkt_{Cl}$, $Produkt_{B'}$, map_6). Map_6 ist ein Mapping zwischen $Produkt_{Cl}$ und $Produkt_{B'}$. Da in $Produkt_{B'}$ nur die neu eingefügten Objekte stehen, in $Produkt_{Cl}$ hingegen nur die bereits existierenden Objekte, ist map_6 leer, von der Wurzel einmal abgesehen. In $Produkt_{Cl'}$ stehen alle Objekte der beiden Modelle der Eingabe. Jedes Objekt in $Produkt_{Cl'}$, das aus $Produkt_{Cl}$ stammt ist durch map_7 mit diesem verbunden. Analog sind alle Objekte aus $Produkt_{B'}$ in $Produkt_{Cl}$ durch map_8 mit $Produkt_{B'}$ verbunden. Das Ergebnis dieses Schrittes ist in Abb. 15 dargestellt.

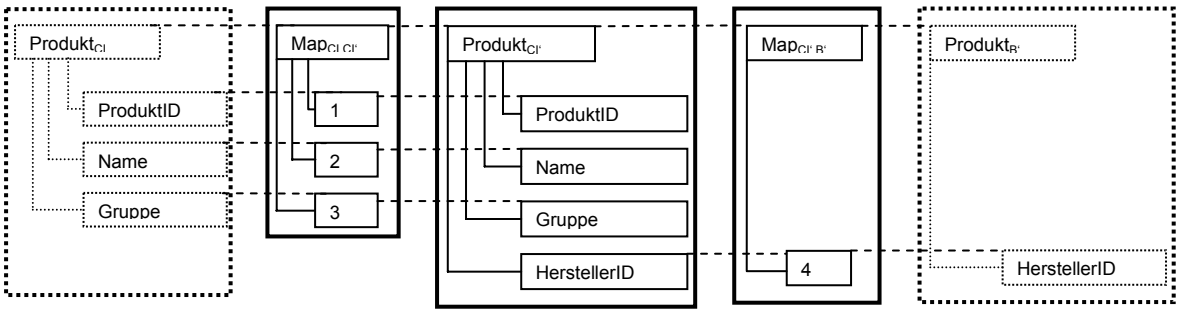


Abb. 15 Das Ergebnis von <$Produkt_{Cl'}$, map_7, map_8> = `Merge`($Produkt_{Cl}$, $Produkt_{B'}$, map_6). Die Modelle am Rand sind Teil der Eingabe und wurden nur zur besseren Lesbarkeit mit dargestellt.

Damit ist die Aufgabe erfolgreich gelöst worden. $Produkt_{Cl'}$ ist das Modell der angepassten Implementierung. Aus diesem lassen sich beispielsweise neue Tabellen erstellen, in welche die Daten migriert werden können. Bei der Migration muss das Attribut HerstellerID mit einem Defaultwert oder sinnvollen Werten belegt werden.

5.2 Weitere Einsatzfelder

Die Einsatzmöglichkeiten der Operatoren beschränken sich nicht auf die Schema-Evolution, sondern lassen sich auch für weitere Szenarien verwenden. Deren Einsatz für Schema-Integration und Round-Trip Engineering werden in diesem Abschnitt kurz beleuchtet.

5.2.1 Schema Integration

Die Integration eines Schemas kann auf ganz unterschiedlicher Grundlage und verschiedene Weise erfolgen. Im einfachsten Fall sind zwei Schemata zu vereinigen. Es kann aber auch sein, dass eine neue Datenquelle an ein Data Warehouse oder an einen Data Mart angebunden werden soll. Wir betrachten hier die Integration einer Datenquelle in ein Data Warehouse, wie von Rahm et al. in [BR00] behandelt.

Ein Data Warehouse zählt zu den Entscheidungsunterstützungssystemen und wird aus mehreren Datenquellen erstellt. Geben sei eine Datenquelle S_1, ein Data Warehouse W und ein Mapping map_1 zwischen S_1 und W. Zu W soll eine zweite Datenquelle S_2 hinzugefügt werden.

Eine Möglichkeit, S_2 weitestgehend automatisch zu integrieren, besteht darin, Ähnlichkeiten zwischen S_1 und S_2 so weit wie möglich auszunutzen. Bei diesem Ansatz, wird das Mapping map_1 bestmöglich genutzt. Dazu wird wie folgt vorgegangen:

1. map_2 = `LeftOuterMatch` (`domain` (map_1), S_2) begrenzt S_1 auf jene Objekte $s_1 \in S_1$, die in map_1 referenziert werden und liefert für diese alle ähnlichen Objekte aus S_2. Existiert zu einem Objekt s_1 kein ähnliches Objekt in S_2, wird wegen des `LeftOuterMatches` dennoch ein Objekt in map_2 eingefügt. Dessen `range` ist dann Ø.

2. $map_3 = map_1 \cdot map_2$. Die Komposition sorgt dafür, dass alle Objekte s_2 aus S_2 in W integriert werden können, zu denen ein Objekt s_1 in S_1 existiert, so dass $s_1 \in$ `range`(map_1).

Für Objekte deren `range` = Ø ist, lassen sich mit `Apply` Dummyobjekte einführen, deren Wert auf „Null“ gesetzt wird. So wird verhindert, dass Teile von S_2 nicht in W integrierbar sind, weil sie nicht über alle notwendigen Attribute verfügen. Stattdessen wird S_2 so weit möglich in W integriert. Sollen zusätzliche Informationen aus S_2 in W aufgenommen werden, muss dies manuell erfolgen. Alternativ könnte auch direkt ein Mapping zwischen S_2 und W erzeugt werden.

Welcher der beiden Ansätze zu bevorzugen ist, hängt im Wesentlichen von den Ähnlichkeiten zwischen (S_2, W) und (S_1, W) ab. Sind sich S_1 und S_2 sehr ähnlich, der Unterschied zwischen S_2 und W aber sehr groß, liefert der erste Ansatz wahrscheinlich das bessere Ergebnis. Im umgekehrten Fall ist zu erwarten, dass ein direktes `Match` zwischen S_2 und W das bessere Resultat liefert. Ist der Unterschied zwischen S_1, S_2 und W jeweils sehr groß oder sehr klein, reicht wahrscheinlich ein direktes `Match` zwischen S_2 und W. Es kann jedoch sein, dass ein `Match` zwischen S_1 und S_2 dennoch ein besseres Ergebnis liefert, wenn die relativen Unterschiede zwischen S_1 und S_2 im Vergleich zu jenen zwischen W und S_2 gering sind.

5.2.2 Round-Trip Engineering

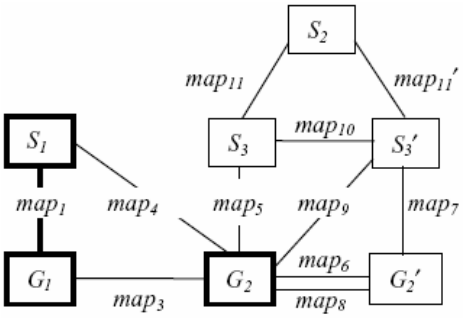
Beim Round-Trip Engineering geht es darum, die Spezifikation einer Implementierung aktuell zu halten. Zunächst wird die Spezifikation implementiert. Nach der

Implementierung ergeben sich mit unter Änderungswünsche seitens der Anwender oder andere Umstände, die ein Anpassen der Implementierung erforderlich machen. Wird diese Anpassung nur auf Ebene der Implementierung durchgeführt, stimmen Implementierung und Spezifikation nicht mehr überein. Das Angleichen der Spezifizierung an die Implementierung wird als Round-Trip Engineering bezeichnet. Das Problem ist anfangs dem der Schema-Evolution sehr verwandt.

Die ersten Schritte zur Lösung des Round-Trip-Engineering-Problems entsprechen den ersten Schritten, die bei der Schema-Evolution zum Einsatz kommen. Der ganze Ablauf wird hier kurz geschildert und in Abb. 16 teilweise dargestellt.

Es wird ein Mapping map_1 zwischen Spezifikation S_1 und Implementierung G_1 gebraucht, von der aus ein Mapping map_3 zur geänderten Implementierung G_2 erstellt wird. Aus den beiden Mappings entsteht durch Komposition ein Mapping map_4 zwischen Spezifikation S_1 und neuer Implementierung G_2. S_1 und map_4 werden kopiert. In der Kopie S_3 von S_1, fehlen alle in der Implementierung G_2 hinzugekommenen Objekte. Um diese automatisch zu finden, wird die Differenz zwischen G_2 und map_3 oder map_4 berechnet. Auf diese Weise entsteht ein Modell G_2', in dem alle neuen Objekte vorkommen. Die neuen Objekte sind durch ein Mapping map_6 mit der Implementierung G_2 verbunden. Aus G_2' muss die Spezifikation der neuen Objekte gewonnen werden. Dazu wird auf G_2' `ModelGen` angewendet.

An dieser Stelle gibt es zwei Teilspezifikationen S_3, in der alle Objekte aus S_1 stehen, die auch in G_2 vorkommen und S_3', in der alle neuen Objekte und deren eventuell vorhandenen Supportobjekte stehen. Gebraucht wird ein Mapping zwischen S_3 und S_3'. Statt dies per `Match`(S_3, S_3') zu erstellen, wird durch `Match`(G_2, G_2') ein Mapping map_8 zwischen der Implementierung und den in selbiger hinzugekommenen Objekte inklusive der notwendigen Supportobjekte erstellt. Durch zweifache Komposition entsteht zunächst map_9 als Komposition von map_7 und map_8 und dann map_{10} als Komposition von map_5 und map_9. Map_{10} ist das gesuchte Mapping zwischen den beiden Spezifikationen, in dem nur Objekte stehen können, die in S_3' als Supportobjekte vorkommen. Aus S_3, S_3' und map_{10} lässt sich schließlich durch Anwendung von `Merge`, die komplette Spezifikation S_2 erhalten. Um das Mapping map_2 zwischen G_2 und S_2 herzuleiten, sind weitere vier Schritte notwendig. Die Vereinigung der Kompositionen von map_5 mit map_{11} (map_2') und map_9 mit map_{11}' (map_2") liefert eventuell zwei Mappings für ein Objekt, weswegen auf sie zunächst ein `Match` angewendet werden muss. Schließlich lässt sich ein `Merge` zwischen map_2', map_2" und dem Ergebnis von `Match`(map_2', map_2") durchführen.



S_1 Spezifikation
G_1 Implementierung
G_2 geänderte Implementierung
G_2' neue Objekte aus G_2
S_3 Kopie von S_1
S_3' Aus G_2' erzeugte Spez.
S_2 generierte Spezifikation

Abb. 16 Zwischenergebnis der Lösung des Round-Trip-Engineering-Problems aus [BE03]

6 Rondo, ein Prototyp

Der Prototyp Rondo wurde entwickelt, um die Einsatzfähigkeit der in Kapitel 4 vorgestellten Operatoren für generische Aufgaben im Bereich des Model Managements zu überprüfen. Eine genauere Beschreibung findet sich in [MRB03], Rondo besteht im Wesentlichen aus einem Skriptinterpreter und mehreren Skripts, sowie einem einfachen graphischen Interface, das das Editieren von Morphismen ermöglicht. Die Hauptaufgabe des Interpreters besteht darin, den Datenfluss zwischen den Skripts zu steuern.

Rondo weicht etwas von dem oben vorgestellten Konzept der Modellierung mit Mappings und Morphismen ab. Er unterstützt keine Mappings, sondern lediglich Morphismen. Damit ist eine Transformation von einem Modelltyp zu einem anderen nicht immer möglich, weil sich Informationen zum Transformieren von Instanzen für Mappings nicht sinnvoll definieren lassen. So ist es beispielsweise nicht möglich, eine SQL-Where-Klausel mit Morphismen zu modellieren, um nur bestimmte Instanzen zu wählen, da der Zusammenhang zwischen den Attributen in Morphismen nicht repräsentiert wird. Bei einem Mapping ist dies hingegen möglich, da der Zusammenhang zwischen den Objekten im Mapping deutlich wird.

Zudem gibt es eine weitere, oben nicht vorgestellte Struktur in Rondo, den Selektor. Ein Selektor ist eine Menge von Objektidentitäten eines oder mehrerer Modelle. Er bringt den Vorteil mit sich, dass Operatoren keine wohlgeformten Modelle ausgeben müssen, sondern Teilmengen von Objekten eines Modells als Ausgabe haben können. Wird diese Ausgaben einem Selektor zugewiesen, kann sie als Eingabe eines anderen Operators verwendet werden. Selektoren stellen ein nützliches Konstrukt da, welches die Implementierung erleichtert, aber nicht notwendig ist, weswegen an anderer Stelle auf seine Vorstellung verzichtet wurde.

Die Architektur des Systems wird in Abb. 17 dargestellt. Das System kann um neue Funktionen erweitert werden, in dem ein neuer Operator programmiert wird oder indem ein Skript, bestehend aus mehreren Aufrufen anderen Operatoren, bereitgestellt wird. Den einfachen Operatoren, der graphischen Oberfläche, den Routinen zur Übersetzung von Schemas in Graphen und vice versa, so wie `GrapheMerge` liegt eine direkte Implementierung zu Grunde. Alle anderen Operatoren werden durch Skripts definiert.

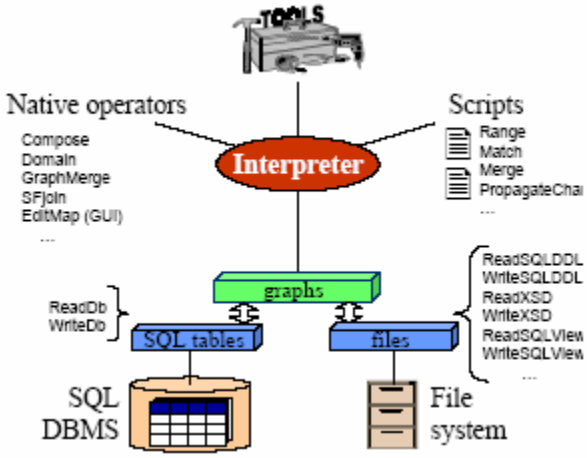


Abb. 17 Rondo-Architektur aus [MRB03]

Rondo unterstützt die Formate SQL DDL, XML-Schemata, RDF-Schemata und SQL-Sichten. Jeder der dafür notwendigen Konverter wird durch eine angepasste nicht generische Implementierung realisiert. Im Folgenden wird auf zwei wichtige Implementierungsdetails eingegangen.

6.1 Similarity Flooding

Der Operator `Match` wird in Rondo mittels des Similarity-Flooding-Algorithmus durchgeführt. Eine ausführliche Beschreibung dieses Algorithmus findet sich in [MGR02]. Hier soll er nur grob skizziert werden. Die beiden zur Übereinstimmung zu bringenden Modelle, mit denen der Algorithmus arbeitet, werden als gerichtete beschriftete Graphen repräsentiert. Weil Rondo selbst gerichtete beschriftete Graphen verwendet, ist die normalerweise vorzusehende Konvertierung in Rondo nicht notwendig. Die weiteren Schritte des Algorithmus sind:

- initialMap = `StringMatch`(G_1, G_2) mit G_1, G_2 den beiden Graphen
- product = SFJoin(G_1, G_2, initialMap)
- result = `SelectThreshold`(product)

Der Operator `StingMatch` berechnet die Ähnlichkeiten zwischen den Knoten der beiden Graphen durch Vergleich üblicher Prä- und Suffixe. Jeder möglichen Kombination der Knoten wird ein Wert zwischen 0 (keine Ähnlichkeit) und 1 (identisch) zugeordnet. Das so erhaltene Mapping wird von SFJoin als Grundlage für die iterative Berechnung der Ähnlichkeiten zwischen Knoten verwendet.

Zur iterativen Ähnlichkeitsberechnung wird zunächst für jede Kante bestimmt, wie gut Ähnlichkeiten über sie propagieren. Anschließend werden die initial berechneten Ähnlichkeiten entlang aller Kanten wiederholt propagiert, bis die eintretenden Änderungen von einer Iteration zur nächsten sehr gering sind, oder bis eine maximale Zahl an Iterationen durchgeführt wurde. Dem liegt die Annahme zugrunde, dass sich Knoten ähnlich sind, wenn es ihre Nachbarn sind.

Für jeden Knoten liefert der Algorithmus eine Vielzahl möglicher Matchkandidaten, weshalb die Ergebnisse anschließend gefiltert werden müssen. `SelectThreshold` ist ein Filter, der in einer Reihe von Tests gute Ergebnisse für unterschiedliche Match-Aufgaben lieferte. Er wählt alle Paare, deren relative Ähnlichkeit einen Schwellwert überschreitet und sichert, dass es zu keinem Knoten einen anderen gibt, der besser zu diesem passen würde. Eine ausführliche Beschreibung des Algorithmus findet sich in [MGR02].

Das Ergebnis muss in jedem Fall geprüft und ggf. angepasst werden. Dies ist in Rondo mit einer einfachen graphischen Oberfläche möglich.

6.2 Graph Merge

Auch die Implementierung der Funktion `Merge` verdient besonderer Bedeutung. Sollen Zwei Modelle M_1 und M_2 vereinigt werden, stellt sich die Frage, welchen Namen die Objekte nach der Vereinigung tragen. Dieser Aspekt spielte bei der

Betrachtung des Operators keine Rolle, ist für die Implementierung aber wichtig. Daher werden die Morphismen durch gerichtete Kanten ersetzt. Die Richtung gibt eine Präferenz an, die sich durch den Anwender in einer graphischen Oberfläche vorgeben lässt. Dabei wird das Zielobjekt in das Ergebnis übernommen, während die Quelle verworfen wird. Der Algorithmus arbeitet nach Vorgabe der Präferenzen in drei Schritten, die in Abb. 18 illustriert werden. Im oberen Teil sind zwei Schemata und das Ergebnis deren Vereinigung zu sehen, im unteren Teil, die Schritte des Algorithmus.

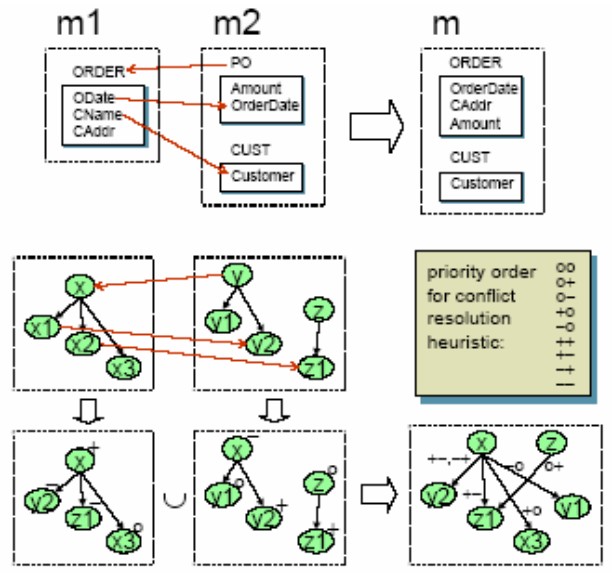


Abb. 18 `Merge` zweier Modelle in Rondo aus [MRB03]

Im ersten Schritt werden alle Knoten an den stumpfen Enden aller gerichteten Morphismen umbenannt. Jeder Knoten bekommen den Namen des Zielknotens, auf den sein Morphismus zeigt. Unten links in Abb. 18 ist das Ergebnis einer solchen Umbenennung für das gegebene Beispiel zu sehen.

Schritt zwei vereinigt die Kanten beider Graphen. Das Ergebnis für unser Beispiel findet sich rechts unten in Abb. 18. Durch die Vereinigung können Graphen entstehen, die keine wohl geformten Modelle sind.

Im dritten Schritt werden Konflikte gelöst. Das Lösen von Konflikten, ist dem Anwender überlassen, der ggf. Kanten entfernen oder hinzufügen muss.

Um Konflikte zumindest teilweise automatisch lösen zu können, wurde eine Heuristik entwickelt, die den Ursprung jeder Kante in dem vereinigten Graph verfolgt. Jedem Knoten wird eine Markierung zugeordnet. Dabei steht (+) dafür, dass der Knoten „Quelle“ eines Morphismus war, (-) zeigt an, dass er Ziel eines Morphismus war. Ist ein Knoten an keinem Morphismus beteiligt, (o) sonst. Jeder Kante in dem vereinigten Graphen werden die Gewichte der durch sie verbundenen Knoten zugeordnet. In dem Beispiel wird der Kante <x, x_3> +o zugeordnet, da x Quelle und x_3 keinen Morphismus hat.

Durch Untersuchung einer Reihe von Problemen konnte eine absolute Ordnung zwischen den verschiedenen Markierungen gewonnen werden. Diese ist mittig rechts in Abb.13 zu sehen. Kanten zwischen nicht geänderten Knoten (oo) haben

die geringste Wahrscheinlichkeit „falsch“ zu sein. Mit dieser Ordnung kann der Graph in Schritt 2 bereits konfliktfrei erstellt werden, indem die Kanten nach der Prioritätsordnung geordnet werden und immer eine Kante mit höchster Priorität aus den verbleibenden Kanten ausgewählt wird. Erzeugt die Kante in dem Graphen einen Konflikt, wird sie verworfen, sonst wird sie in den Graph eingefügt.

Der Algorithmus gewährleistet nicht, dass die Vereinigung mindestens so ausdrucksstark ist, wie die beiden Eingaben sind. Um dies zu gewährleisten wird eine restriktivere Implementierung benötigt.

7 Zusammenfassung

Im Vordergrund der Betrachtung stand die Klasse der Probleme, die bei Schema-Evolution entstehen. Für deren Lösung wurde ein allgemeiner Weg aufgezeigt, um Zeit und Aufwand bei der Anpassung von Systemen an sich wandelnde Anforderungen zu sparen.

Wir haben eine ganze Reihe von Operatoren betrachtet, die für die automatische Lösung der gegebenen Problemstellungen notwendig sind und gezeigt, dass diese auch für eine Reihe anderer Aufgaben verwendet werden können. Dies wurde für drei klassische Managementprobleme mit unterschiedlicher Genauigkeit gezeigt. Im letzten Kapitel wurde ein Prototyp vorgestellt, der die Konzepte zum Teil umsetzt. Komplexe Evolutionsszenarien, wie sie in der Praxis vorkommen, lassen sich mit dem vorgestellten Prototyp nicht bearbeiten. Dafür ist die Oberfläche deutlich zu rudimentär und es fehlt die Möglichkeit, Ergebnisse zwischen zu speichern um die Arbeit zu unterbrechen.

Dennoch sind damit erste Schritte hin zu Modelmanagementlösungen getan. Die notwendigen Operatoren, um Managementaufgaben generisch behandeln zu können, wurden zumindest auf theoretischer Ebene erforscht und hier vorgestellt. Doch müssen die Aufgaben praktisch gelöst werden. Dazu fehlen bisher geeignete Tools. Ähnlichkeiten lassen sich bisher nicht explizit spezifizieren. Dies ist bei Änderungen der Terminologie jedoch sehr wichtig, um den manuellen Aufwand zu minimieren. Außerdem sollten Änderungen an den Schemata automatisch auf die Daten propagiert werden, denn genau an dieser Stelle verbirgt sich das Potenzial, Zeit zu sparen und Fehler zu vermeiden. Sollte es für eine Reihe von Aufgaben gelingen, Änderungen aus Schemata direkt auf die Daten zu übertragen, müssten diese nicht mehr von Hand durchgeführt werden. Wird dies nicht gelingen, ist der Nutzen der Schemaänderungen sehr begrenzt, da der Anwender sich immer für die Daten interessieren wird.

Es ist davon auszugehen, dass die Erforschung und Implementierung von Modelmanagementlösungen viele Forschungsgruppen noch Jahre beschäftigen werden [ME05]. Vermutlich werden sie selbst eine Evolution durchlaufen, an deren Anfang Prototypen stehen, in deren Verlauf externe Tools entstehen, ehe sie am Ende in bestehende Systeme integriert werden.

Bisher existieren nur Lösungsansätze. Der Weg zu einer praktischen Lösung ist noch lang. Ihn zu gehen verspricht aber dauerhaften Nutzen durch schnellere und korrekte Durchführbarkeit von Änderungen und den damit verbundenen Wettbewerbsvorteilen.

8 Literatur

[BE03] Bernstein P.: Applying Model Management to Classical Meta Data Problems, Procceedings of the 2003 CIDR Conference, 2003

[BLP00] Bernstein P., Levy A., Pottinger R.: A Vison for Management of Complex Models, Microsoft, 2000

[BR00] Bernstein P.; Rahm E.:Data Warehouse Scenarios for Model Management, International Conference on Conceptual Modeling, Springer, 2000

[LNE89] Larson J., Navathe S., Elmasri R.: A theory of attribute equivalence in databases with application to schema integration., Transactions on Software Engineering 15(4): 449 – 463, 1989

[ME05] Melnik S.: Model Management: First Steps and Beyond, BTW 2005

[MGR02] Melnik S., Garcia-Molina H., Rahm E.: Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching, 18th international conference on Data Engineering, 2002

[MRB03] Melnik S.; Rahm E.; Bernstein P.: Rondo: A Programming Platform for Generic Model Management, SIGMOD 2003

[SJ93] Sjoberg, D.: Quantifying Schema Evolution. Information and Software Technology Journal, Vol. 35, No. 1, January 1993.

[TÜ00] Türker, C.: Schema Evolution in SQL-99 and Commercial (Object-) Relational DBMS, in: Proc. 9th Int. Workshop on Foundations of Models and Languages for Data and Objects (FoMLaDO 2000), Dagstuhl, Sep. 2000, Springer-Verlag, pp. 1-32.

9 Abbildungsnachweis

Abb. 9 [BE03], Seite 6

Abb. 10 [BE03], Seite 9

Abb. 16 [BE03], Seite 10

Abb. 17 [MRB03], Seite 10

Abb. 18 [MRB03], Seite 8