Modelltransformationen mit BOTL

Stefan Heukamp
252 458

Betreut von Dipl.-Inform. Anne-Thérèse Körtgen

Zusammenfassung

Bei der Software-Entwicklung mit ingenieurwissenschaftlichen Methoden werden vor allem während der Anforderungsanalyse und der Entwurfsphase Modelle von der zu erstellenden Software angefertigt, die z.B. das Außenverhalten oder die Architektur beschreiben. Sie entstehen durch schrittweise Verfeinerung der Modelle der vorhergegangenen Phase. Damit spiegelt ihr Abstraktionsgrad den der jeweiligen Phase wieder und nimmt mit der Zeit ab.
In der vorliegenden Arbeit wird die Bidirektionale Objektorientierte Transformationssprache BOTL vorgestellt, mit der Objektmodelle, eine Klasse von Modellen in graphischer Notation, in andere Objektmodelle transformiert werden können. Durch gewisse Eigenschaften der Sprache und der in ihr formulierten Regeln können diese Transformationen auch automatisiert durchgeführt werden. Dies steigert die Produktivität, da die Transformationen nicht mühsam von Hand durchgeführt werden müssen und senkt somit die Kosten. Weitere Eigenschaften der Sprache sichern dabei die Konsistenz der Umformungen und machen damit eine Weiterverarbeitung der Modelle möglich. Dies unterstützt den ebenfalls in dieser Arbeit beschriebenen Entwicklungsprozess KOGITO, der auf diesen automatischen Modellverfeinerungen basiert.

Inhaltsverzeichnis

1 Einleitung

Die Programmierung in höheren Programmiersprachen wie z.B. Java und C++, wie sie heute in den meisten Softwareentwicklungsprozessen betrieben wird, ist für die Firmen teuer, da sie zeitaufwändig und verhältnismäßig fehleranfällig ist. Durch einen höheren Abstraktionsgrad wird Softwareentwicklung einfacher, da der Entwickler sich auf die wesentliche Programmlogik konzentrieren kann und sich nicht mit Implementierungsdetails beschäftigen muss. Dies kann die Entwicklung beschleunigen, die Anzahl der Fehler reduzieren und damit die Kosten deutlich senken. Um den Abstraktionsgrad auf die gewünschte Stufe zu heben, werden Modelle eingesetzt. Diese Modelle können jedoch nicht unmittelbar von einer CPU ausgeführt werden. Werden diese Modelle von Hand in eine Programmiersprache übersetzt, so ist der Gewinn gering, da bei dieser Übersetzung die oben beschriebenen Nachteile der Programmierung auftreten. Günstiger wäre also eine automatisierte Transformation der Modelle in eine ausführbare Form.

Die automatische Transformation von Modellen ist ein wesentlicher Bestandteil der modellgetriebenen Softwareentwicklung, mit der sich das Seminar „Unterstützung modellgetriebener Softwareentwicklung“, für das diese Arbeit erstellt wurde, beschäftigt. Für automatische Modelltransformationen wird eine Sprache, in der Transformationsregeln definiert werden können und ein Algorithmus, der diese anschließend ausführt, benötigt. Im Folgenden wird die Sprache BOTL als Regelsprache vorgestellt, die die Definition der benötigten Transformationsregeln erlaubt. Des weiteren wird der Algorithmus beschrieben, der die Regeln automatisch anwendet.

1.1 Model Driven Architecture

Seit der Softwarekrise in den 1960ern wird die Softwareentwicklung als Ingenieursdisziplin angesehen [7]. Daher werden auch dabei Modelle eingesetzt um von unwichtigen Details zu abstrahieren und den Überblick zu behalten. Bei dem von der Object Management Group (OMG) [2] vorgestellten Konzept der *Model Driven Architecture* (MDA) [10] werden die Modelle zum zentralen Element. Der Quellcode wird erst im letzten Schritt aus den Modellen erzeugt. Bis zu diesem Zeitpunkt werden die Modelle stetig weiter entwickelt und können mit Hilfe von automatischen Transformationen in detailliertere Modelle überführt werden. Die Idee der Transformation von Modellen in Quellcode ist wie die Verwendung von Modellen nicht neu. Schon in den 1970er Jahren wurden Werkzeuge zum automatischen Generieren von Quellcode aus Modellen in textueller Form erfolgreich verwendet. Beispiele dafür sind der Parser-Generator *yacc* [5] und der Scanner-

Generator *lex* [1]. Das neue an dem MDA Ansatz ist seine Vielseitigkeit. Während bisherige Ansätze wie oben gezeigt nur für bestimmte Software eingesetzt werden konnten, soll MDA die Entwicklung von beliebiger Software unterstützen.

Die konsequente Verwendung von Modellen hat einige wesentliche Vorteile. Ein solcher Vorteil ist, dass Modelle im Gegensatz zu Code sehr gut wiederverwendet werden können. Diese Wiederverwendung spart Zeit und damit Kosten bei der Entwicklung von neuen Anwendungen in der gleichen Domäne. Außerdem können die abstrakten Modelle für die Entwicklung der gleichen Software auf einer anderen Plattform oder auf Basis einer anderen Technologie verwendet werden, da die grundlegenden Konzepte dabei oft gleich sind. Ob z.B. bei einer verteilten Anwendung bei den verteilten Funktionsaufrufen eine Middleware wie CORBA [15] oder Webservices [11] eingesetzt werden, kann so noch sehr spät entschieden werden bzw. eine getroffene Entscheidung kann durch die Angabe geeigneter Modelltransformationen sehr leicht verändert werden. Ein weiterer Vorteil der Modellbildung ist, dass Arbeiten an graphischen Modellen mit einem angemessenen Abstraktionsgrad für das menschliche Gehirn leichter zu bewerkstelligen sind als Arbeiten am textuellen Quellcode. Somit werden dabei weniger Fehler gemacht, was wiederum hilft, die Entwicklungszeit und damit die Kosten zu senken.

Konkret sieht die MDA drei Modellebenen vor: (1) Das *Computation Independent Model* (CIM) dient dem Dialog zwischen fachlichen Experten und Entwicklern und definiert die fachlichen Anforderungen an das System. (2) Das *Platform Independent Model* (PIM) legt die Struktur des Systems in einer plattformunabhängigen Weise fest. (3) Beim *Platform Specific Model* (PSM) können bei der Modellierung auch die speziellen Eigenschaften der Plattform berücksichtigt werden. Genauere Beschreibungen der Ebenen finden sich auch in [9].

Nach den Autoren von [14] war die MDA die logische Konsequenz aus den bisherigen Entwicklungen. Seit dem Beginn der Programmierung von Computern benutzen die Entwickler Abstraktion, die mit der Zeit einen immer höheren Grad erreichte. Der erste Schritt abstrahierte von den Prozessorbefehlen als Binärwörtern auf leichter zu merkende Assemblerbefehle. Es folgten höhere Sprachen wie C, die von vielen Eigenschaften des Prozessors abstrahierten und komplexe Kontrollstrukturen einführten. Der nächste Schritt waren „Garbage Collection“ und „Middleware“, die den Abstraktionsgrad erneut erhöhten. Somit sei die logische Konsequenz ein nächster Schritt, bei dem von der Programmiersprache selbst mit visuellen Modellen abstrahiert wird und dies, anders als bei den bisherigen Ansätzen zur Codegenerierung, auf alle Probleme übertragbar.

1.2 Aufbau der Arbeit

In der folgenden Arbeit wird die Bidirektionale Objektorientierte Transformationssprache BOTL vorgestellt. Dazu werden zunächst in Kapitel 2 die Grundlagen gelegt, indem die für einen modellbasierten Entwicklungsprozess essentiellen Begriffe wie *Modell*, *Metamodell*, *Prozessmodell* und *konzeptuelles Modell* erklärt werden. Danach wird in Kapitel 3 die Sprache BOTL vorgestellt. Anhand eines Beispiels werden BOTL-Regeln und deren Anwendung in informaler Weise beschrieben und ein UML-Profil für BOTL-Regeln vorgestellt, um zu zeigen, dass die Erstellung der Regeln in UML-Editoren möglich ist. Anschließend wird in Kapitel 3.3 exemplarisch die Formalisierung des Begriffs „Klasse“ vorgestellt. Um in automatischen Transformationsprozessen eingesetzt zu werden, müssen einige Eigenschaften für die Regeln gelten. Auf diese Eigenschaften wird in Kapitel 3.4 näher eingegangen. In Kapitel 4 wird die Verwendung von BOTL in einem tatsächlichen Entwicklungsprozess beschrieben. Abschließend findet in Kapitel 5 eine Diskussion über BOTL statt.

2 Modellbasierte Entwicklungsprozesse

Um modellbasierte Entwicklungsprozesse im Allgemeinen, MDA und die Sprache BOTL im Speziellen zu verstehen, werden zunächst in Kapitel 2.1 die Begriffe Modell und Metamodell definiert. Danach wird in Kapitel 2.2 beschrieben, wie Modelle eingesetzt werden können, um einen Entwicklungsprozess zu beschreiben. In Kapitel 2.3 wird schließlich darauf eingegangen, wie ein zu entwickelndes System mit Hilfe der dort beschriebenen konzeptuellen Modelle abgebildet werden kann.

2.1 Modelle und Metamodelle

Nach Herbert Stachowiak zeichnet sich ein *Modell* im Wesentlichen durch drei Eigenschaften aus [16]:

- Modelle beschreiben einen *Teil der Realität*.
- Modelle erfüllen eine *Funktion*.
- Modelle *abstrahieren* von unwesentlichen Details.

Dass die ersten beiden Punkte sinnvoll sind, ist unmittelbar einsichtig. Die Abstraktion von unwesentlichen Details bringt den Vorteil der Übersichtlichkeit. Dadurch, dass ein Sachverhalt aus einem bestimmten Blickwinkel betrachtet wird, können alle Details, die nicht diesen Blickwinkel betreffen, außer Acht gelassen werden und der Betrachter kann sich auf die wesentlichen Merkmale konzentrieren. Werden Modelle im Entwicklungsprozess eingesetzt, wird dieser dadurch erheblich vereinfacht.

Eine Klasse von Modellen mit einer bestimmten Gemeinsamkeit kann durch ein *Metamodell* beschrieben werden. Metamodelle sind Mengen, ihre Elemente sind die von ihnen beschriebenen Modelle. Die Modelle eines Metamodells werden auch als *Instanzen* bezeichnet. Die Beziehung zwischen Metamodellen und Modellen ist vergleichbar mit der zwischen einer Grammatik und der von ihr erzeugten Sprache. Dabei entspricht das Metamodell der Grammatik und die Instanzen den Wörtern in der erzeugten Sprache. Da Metamodelle ihrerseits wieder Modelle sind, ist die Menge aller Metamodelle eine Teilmenge der Menge aller Modelle.

Im Allgemeinen unterteilt man Modelle des Weiteren in *semiformale Modelle* und *formale Modelle*. Für semiformale Modelle ist die Syntax festgelegt. Ihre Metamodelle legen Kriterien für ihren Aufbau fest. Für formale Modelle ist darüber hinaus auch die Semantik festgelegt. Diese Festlegung erfolgt anhand einer Sprache mit formaler Semantik, wie z.B. einer Programmiersprache oder einer Sprache, die mathematisch fundiert ist. Dadurch wird die Interpretation eines Modells eindeutig, was z.B. bei natürlicher Sprache nicht gegeben ist. Die UML-Modelle sind nach dieser Definition nur semiformal, da die Syntax zwar eindeutig definiert ist, die Semantik jedoch nur textuell und oftmals unpräzise definiert ist (vgl. [6]).

2.2 Prozessmodelle

Um den Softwareentwicklungsprozess ingenieurwissenschaftlichen Methoden anzunähern, kann dieser auch mit Hilfe eines *Prozessmodells* beschrieben werden. In einem definierten Entwicklungsprozess ist die von einem Entwickler als nächstes auszuführende Tätigkeit zu jedem Zeitpunkt genau bestimmt. Dies ist zum einen eine Arbeitserleichterung, da sich der Entwickler auf die Entwicklung konzentrieren kann. Zum anderen lassen sich, wenn der Entwicklungsprozess zum Beispiel firmenweit vereinheitlicht ist, leichter Mitarbeiter neuen Projekten zuteilen, da sich diese nicht mehr in den Entwicklungsprozess als solchen einarbeiten müssen. Entspricht der Entwicklungsprozess einem formalen Modell, ist also auch dessen Semantik formal festgelegt, besteht daneben die Möglichkeit die Durchführbarkeit und die Korrektheit des Prozesses zu verifizieren.

Um Metamodelle für Entwicklungsprozesse anzugeben, müssen zunächst einige Begriffe erklärt werden. Unter einem *Artefakt* versteht man ein Dokument in einem konkreten Entwicklungsprojekt. Beispiele für Artefakte sind Quelltexte, Architekturdiagramme oder Projektdokumentationen. *Artefakttypen* beschreiben im Wesentlichen die Syntax eines Artefakts. Die Menge aller für einen Entwicklungsprozess benötigten Artefakttypen und ihre möglichen Beziehungen werden durch das *Produktmodell* beschrieben. Damit ist ein Produktmodell ein Metamodell und seine Instanzen sind die möglichen Ausprägungen der Artefakte in einem konkreten Entwicklungsprojekt. Ein *Prozessmodell* als Metamodell für einen Entwicklungsprozess besteht aus dem Produktmodell, einer Menge von Aktivitätsbeschreibungen und einer Festlegung der möglichen Reihenfolge der Aktivitäten. Mit *Aktivitätsbeschreibung* wird in diesem Zusammenhang eine Abbildung bezeichnet, die Artefakte in andere Artefakte überführt. Zu der formalen Abbildung gehört eine textuelle Dokumentation des Vorgehens zur Durchführung der Aktivität.

Mit den oben definierten Begriffen ist es möglich einen Entwicklungsprozess als Instanz eines Prozessmodells formal zu definieren und Attribute wie Durchführbarkeit und Korrektheit zu verifizieren. Als ein Beispiel für ein solches Prozessmodell wird in Kapitel 4 die Kogito-Methodik vorgestellt. Die Kogito-Methodik geht nach den Methoden der MDA vor und setzt dabei automatische Modelltransformationen mit Hilfe der Sprache BOTL ein.

2.3 Konzeptuelle Modelle und Transformationen

Im Gegensatz zum Prozessmodell bildet das *konzeptuelle Modell* nicht den Entwicklungsprozess, sondern das zu entwickelnde System und relevante Teile seiner Umwelt ab. Die Bestandteile des konzeptuellen Modells sind alle während des Entwicklungsprozesses entstandenen Artefakte.

Eine *Sicht* ist eine Teilmenge eines konzeptuellen Modells. Die enthaltenen Artefakte werden vom *Aspekt* der Sicht festgelegt. In der Softwareentwicklung wird häufig die Sicht der Systemstruktur als Aspekt betrachtet. Enthalten in dieser Sicht sind statische Architekturdiagramme, wie z.B. Komponentendiagramme oder, auf einem detaillierteren Level, Klassendiagramme. In dieser Sicht wird von allen Artefakten abstrahiert, die das dynamische Verhalten des Systems beschreiben.

Ziel der modellbasierten Entwicklung ist die automatische Transformation der konzeptuellen Modelle in andere konzeptuelle Modelle. Dazu lassen sich die Transformationen in drei Gruppen einteilen:

1. Artefakte können in andere Artefakte überführt werden. Diese Transformation erfolgt anhand der Aktivitätsbeschreibungen und ist im Kapitel 2.2 näher beschrieben.

2. In einer Bottom-Up-Entwicklung können Artefakte mehrerer Sichten zu einem konzeptuellen Modell integriert werden.

3. Konzeptuelle Modelle können in verfeinerte Modelle transformiert werden. In die entstandenen Modelle können anschließend wieder Sichten bzw. deren Artefakte integriert werden.

Transformationssprachen dienen dazu die Transformationen zwischen Metamodellen zu spezifizieren. Bei einer *Objektorientierten Transformationssprache* sind die Metamodelle Klassenmodelle im Sinne der UML und deren Instanzen entsprechend Objektmodelle. Klassenmodelle bestehen aus Klassen und Assoziationen zwischen ihnen und Objektmodelle aus Objekten und Objektassoziationen.

Es gibt viele verschiedene Anforderungen an Transformationssprachen, die sich auch teilweise gegenseitig ausschließen. Sie beziehen sich auf die Mächtigkeit, Effizienz, Anwenderfreundlichkeit, Ausführbarkeit der Transformationsregeln und viele andere Eigenschaften der jeweiligen Sprache.

3 BOTL

Im vorliegenden Kapitel wird die Bidirektionale Objektorientierte Transformationssprache BOTL vorgestellt. Zunächst wird sie dazu anhand von Beispielen in informeller Weise beschrieben. In Kapitel 3.2 wird dann ein UML-Profil für BOTL-Regeln vorgestellt, um zu zeigen, dass sie sich leicht mit schon vorhandenen Werkzeugen erstellen lassen. In [13] kann eine mathematisch fundierte Einführung in die Sprache gefunden werden. In Kapitel 3.3 werden davon nur einige wichtige Begriffe exemplarisch durch mathematische Definitionen vorgestellt, um einen Einblick zu geben, wie sich Klassen, Objekte und ähnliche Dinge formalisieren lassen. Abschließend erfolgt in Kapitel 3.4 eine Diskussion einiger wichtiger Eigenschaften von BOTL.

3.1 BOTL-Regeln und ihre Anwendung

BOTL transformiert als eine objektorientierte Transformationssprache im Allgemeinen eine Menge von Objektmodellen $m_0, \ldots, m_n$ als Instanzen der entspre-

chenden Klassenmodelle $mm_0, \ldots, mm_n$ in ein Objektmodell m_t mit zugehörigem Meta- bzw. Klassenmodell mm_t. Diese Eigenschaft unterstützt einen modellbasierten Entwicklungsprozess insofern, als dass dadurch beispielsweise ein Modell m_0 mit hohem Abstraktionsgrad und ein Modell m_1, das einen Detailaspekt näher definiert, zu einem Zielmodell m_t verschmolzen werden können. Im folgenden Beispiel wird jedoch, um die Komplexität nicht zu erhöhen, ein einziges Quellmodell in ein Zielmodell transformiert.

Die Abbildungen 1 und 2 beschreiben zwei Klassenmodelle zur Verwaltung von Büros und Angestellten in einer Firma. Es ist unmittelbar klar, dass beide Modelle bis auf den Namen der Firma den gleichen Informationsgehalt haben. Ein Personobjekt enthält die gleiche Information wie ein Angestelltenobjekt mit seiner Assoziation zu einem Büroobjekt. Ein Raumobjekt enthält die gleiche Information wie ein Büroobjekt mit einem assoziierten Telefonobjekt. Die Unterschiede bestehen also lediglich in der Darstellung der Information. Im Klassenmodell mm_1 ist Telefon eine eigene Entität, im Klassenmodell mm_2 wird es durch ein Attribut des Raumes beschrieben, in dem es steht. Mit Hilfe geeigneter Regeln sollten also Instanzen der Metamodelle mm_1 und mm_2 ineinander transformierbar sein.

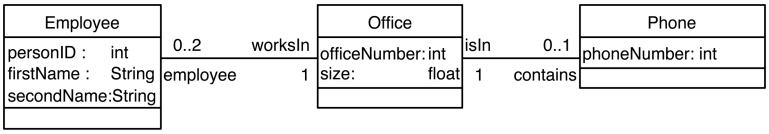

Abbildung 1: Das Klassenmodell mm_1 zur Verwaltung von Büros, Angestellten und Telefonen. (Aus [8])

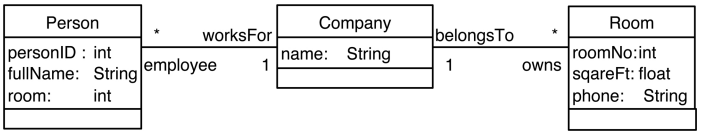

Abbildung 2: Das Klassenmodell mm_2 zur Verwaltung von Personen und Räumen mit Telefonnummern in einer Firma. (Aus [8])

Abbildung 3 zeigt ein Beispiel für ein Objektmodell, das konform zu dem Klassenmodell mm_1 in Abbildung 1 ist. Es enthält die beiden Angestelltenobjekte 17 und 18, die sich das Büro (Objekt 1517) teilen. In der obersten Zeile jedes Objektes wird vor dem Klassennamen ein *Objektidentifikator* notiert, der eindeutig unter den Instanzen einer Klasse ist. Zwei Objekte unterschiedlicher Klassen können somit den gleichen Objektidentifikator haben.

Annähernd die gleiche Information wie in Abbildung 3 wird durch das Objektmodell m_2 gemäß des Klassenmodells mm_2 in Abbildung 4 modelliert. Dabei

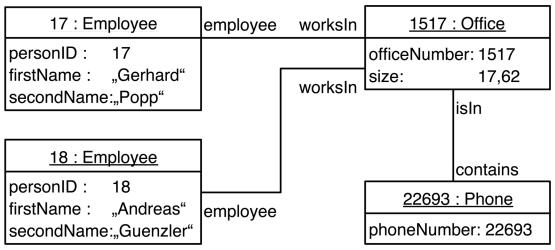

Abbildung 3: Beispiel für eine Instanz m_1 des Klassenmodells mm_1 mit Büroobjekt, zugehörigem Telefon- und zwei Angestelltenobjekten. (Aus [8])

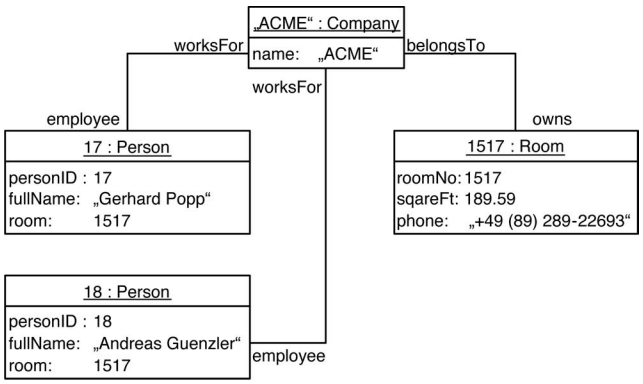

Abbildung 4: Beispiel für eine Instanz m_2 des Klassenmodells mm_2 mit Firma ‚ACME‘, zwei Personobjekten und einem Raumobjekt. (Aus [8])

wurden im Wesentlichen die Namen der Angestellten im Attribut fullName konkateniert, die Verbindung zum Büro nicht als Assoziation worksIn sondern als Attribut room dargestellt und das Telefon ebenfalls nicht als eigene Entität Phone sondern als Attribut phone des Raumes modelliert. Daneben wird die Telefonnummer mit Vorwahl abgespeichert und die Firma wird als Entität Company dargestellt. Da die Modelle die gleichen Informationen beinhalten, ist es prinzipiell möglich, Transformationsregeln anzugeben, anhand derer die Objektmodelle m_1 und m_2 ineinander überführt werden können. Die entsprechenden Regeln dazu werden nach einer kurzen Definition der BOTL-Regeln und Regelwerke und ihrer Anwendung vorgestellt.

BOTL-Regeln bestehen aus einer linken und einer rechten Regelseite. Eine *Regelseite* besteht aus Objekten und Objektassoziationen und muss bis auf die Kardinalität der Objektassoziationen konsistent sein zu dem jeweiligen Klassenmodell. Die linke Regelseite muss demnach konsistent zum Quellmetamodell und die rechte Seite konsistent zum Zielmetamodell sein. Den Attributen eines Objektes werden *Terme* zugeordnet, die aus konstanten Werten, z.B. Zeichenketten oder

Ganzzahlen, Variablen oder dem Symbol $\Diamond$ bestehen. Eine Menge von Regeln wird *Regelwerk* genannt.

Die Anwendung einer BOTL-Regel untergliedert sich im Wesentlichen in die folgenden Schritte:

1. Es wird ein Fragment im Quellmodell gesucht, das die gleiche Struktur hat, wie die linke Regelseite. Dabei werden die Attributterme in der Regelseite mit den Attributwerten des Quellmodellfragments unifiziert und die Variablen in den Termen entsprechend belegt. Enthält ein Term das Symbol $\Diamond$, so wird der zugehörige Attributwert für diese Regelanwendung nicht weiter betrachtet. Dieser Vorgang wird *Matching* genannt.

2. Es wird ein Zielmodellfragment mit der Struktur der rechten Regelseite erzeugt und die Attribute werden gemäß der Attributterme der rechten Regelseite und der Variablenbelegung aus Schritt 1 belegt. Erscheint das Symbol $\Diamond$ auf der rechten Regelseite, so gilt der Attributwert als nicht initialisiert.

3. Das Zielmodellfragment wird in das Zielmodell eingefügt. Existiert dabei schon ein Objekt mit dem gleichen Objektidentifikator im Zielmodell, so werden die Objekte verschmolzen. Beim Verschmelzen werden nicht initialisierte Attribute im Objekt des Zielmodells oder im Objekt des Modellfragments durch die Attributbelegung des jeweils anderen überschrieben. Sind beide Attributwerte nicht initialisiert, so ist auch der Attributwert des Ergebnisses nicht initialisiert. Enthalten die zu verschmelzenden Objekte einen unterschiedlichen Attributwert, der von „nicht initialisiert“ bzw. $\Diamond$ verschieden ist, so bricht die Anwendung der Regel mit einem Fehler ab.

Jede Regel im Regelwerk wird für jedes gefundene Fragment im Quellmodell genau einmal ausgeführt. Die Reihenfolge der Regelanwendung spielt dabei keine Rolle. Enthält das Zielmodell, nachdem alle Regeln angewendet wurden, nicht das Symbol $\Diamond$ und wurde keine Regelanwendung mit einem Fehler abgebrochen, so war die Regelwerksanwendung erfolgreich.

Um mm_1-konforme Modelle in mm_2-konforme Modelle zu transformieren sind die Regeln r_0 und r_1 wie in den Abbildungen 5 und 6 dargestellt nötig. Regel r_0 transformiert ein Employee- und ein Office-Objekt und deren worksIn-Assoziation in die korrelierenden Room- und Person-Objekte und deren Assoziationen. Daneben wird auch das Company-Objekt erzeugt. Regel r_1 definiert in den Room-Objekten die Telefonnummer, die es aus dem zugehörigen Office-Objekt und dem dazu assoziierten Phone-Objekt berechnet.

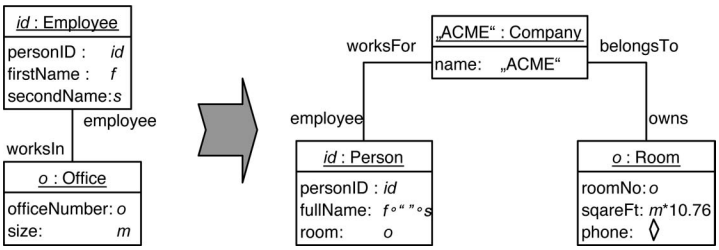

Abbildung 5: Regel r_0 zur Transformation eines Angestellten- und eines Büroobjekts im Quellmetamodell mm_1 in die entsprechenden Konstrukte des Zielmetamodells mm_2. (Aus [8])

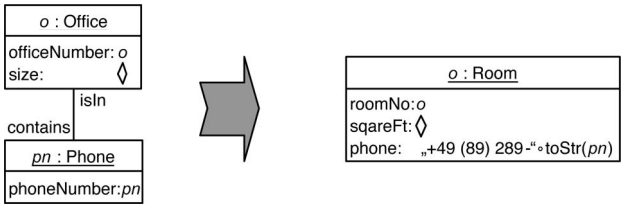

Abbildung 6: Regel r_1 zur Transformation eines Büro- und eines Telefonobjekts im Quellmetamodell mm_1 in ein Raumobjekt des Zielmetamodells mm_2. (Aus [8])

Wendet man die Regeln r_0 und r_1 mit dem oben beschriebene Verfahren auf das Beispielmodell m_1 an, so wird bei der ersten Anwendung von Regel r_0 zunächst das Angestelltenobjekt 17 und das Büroobjekt 1517 im Modell m_1 durch die linke Regelseite gematched und die Variablen wie folgt belegt: id = 17, f = Gerhard, s = Popp, o = 1517, m = 17.62. Dann wird ein Zielmodellfragment mit dem Personobjekt „17“, dem Raumbjekt 1517 und dem Firmenobjekt ACME angelegt. Die Attribute der Objekte werden gemäß der Variablen gesetzt, und die Objekte werden über die entsprechenden Assoziationen miteinander verbunden. Das Attribut Phone ist zu diesem Zeitpunkt noch nicht initialisiert. Im letzten Schritt wird das Modellfragment in das noch leere Zielmodell eingefügt. Das Zielmodell enthält danach genau die Objekte und Assoziationen des erzeugten Zielmodellfragmentes.

Bei der zweiten Anwendung von Regel r_0 wird das Personobjekt 18, das Raumobjekt 1517 und das Firmenobjekt ACME angelegt. Beim Einfügen in das bestehende Zielmodell werden jeweils die Firmen- und die Raumobjekte miteinander verschmolzen, da sie den gleichen Objektidentifikator 1517 bzw. ACME haben. Das Ergebnis ist das Zielmodell wie in Abbildung 7 dargestellt, mit den Objekten für die Firma ACME, den Raum 1517 mit noch nicht initialisiertem Attribut Phone, und die Personen 17 und 18. Daneben enthält das Zielmodell die jeweiligen

Assoziationen zwischen den Objekten für die Firma und den Raum bzw. zwischen der Firma und den Personen.

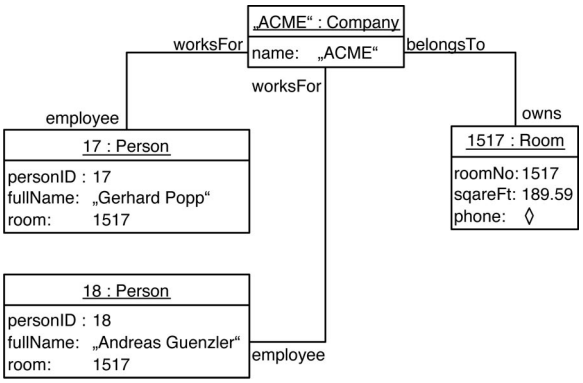

Abbildung 7: Das aus Modell m_1 nach zweimaliger Anwendung von Regel r_0 entstandene Zielmodell. (Aus [8])

Bei der Anwendung der Regel r_1 gibt es nur ein Matching, da es im Objektmodell m_1 nur ein Büroobjekt 1517 und ein Telefonobjekt 22693 gibt. Es werden zunächst die Variable o mit 1517 und pn mit 22693 belegt. Aufgrund des ◊-Symbols auf der linken Regelseite wird die Raumgröße squareFt nicht in einer Variablen abgelegt. Dann wird das Raumobjekt 1517 gemäß der rechten Regelseite erstellt und die Werte der Attribute roomNo und phone berechnet. Beim Einfügen in das Zielmodell wird das Raumobjekt mit dem bestehenden Raumobjekt verschmolzen. Dabei werden die nicht definierten Werte phone im bestehenden Zielmodell und squareFt im Zielmodellfragment mit den definierten Werten des jeweils anderen überschrieben, so dass das Modell m_2, wie in Abbildung 4 dargestellt, als Ergebnis der Transformation zurückgegeben wird.

Die bisher beschriebenen Regeln sind für die Anwendung oftmals zu unflexibel. Soll das oben beschriebene Regelwerk für eine andere Firma angewendet werden, so müssen beide Regeln angepasst werden. In der Regel r_0 müsste der Name und in Regel r_1 die Vorwahl bzw. Telefonnummer der Firma geändert werden. Um diese Anpassungen zentral zu halten, werden in BOTL parametrisierbare Regeln eingeführt. Parametrisierbare Regeln können neben Konstanten, Variablen und ◊ auch Parameter enthalten, die mit dem Zeichen $ beginnen. Parameter müssen vor der Regelwerksanwendung mit Werten belegt werden und werden dann wie Konstanten mit dem Wert des Parameters behandelt. Abbildung 8 zeigt die Regel r_0 in einer parametrisierten Version. Der Name der Firma wird durch einen Parameter $CName spezifiziert, so dass die Regel für Transformationen beliebiger Firmen mit den gleichen Datenmodellen genutzt werden kann.

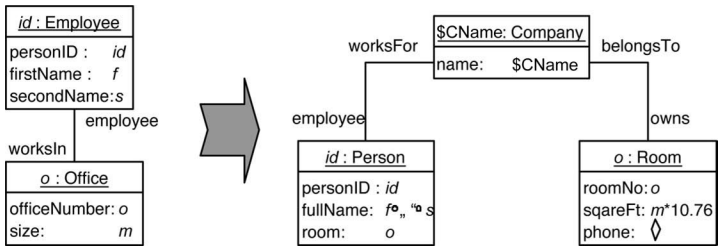

Abbildung 8: Parametrisierte Version der Regel r_0. Macht die Anwendung des Regelwerkes universeller, da es nicht nur für die Firma ACME verwendet werden kann.

3.2 UML-Profil für BOTL-Regeln

Um zu zeigen, dass die Erstellung von BOTL-Regeln in der Praxis einfach möglich ist, wird in [13] ein UML-Profil vorgestellt, das es ermöglicht, die Regeln in jedem UML-Editor zu erstellen. Das Profil enthält vier Stereotype für Pakete:

1. Ein Paket mit dem Stereotyp *Source Model Variable* beschreibt die linke Seite einer BOTL-Regel und enthält ein Objektmodell. Ein Beispiel für eine solche Regelseite findet sich in Abbildung 9. Das Paket mySourceModel-Variable enthält ein Objektdiagramm mit drei Objekten, die untereinander verbunden sind.

2. Das Stereotyp *Target Model Variable* kennzeichnet ein Paket als eine rechte Regelseite. Ein mit diesem Stereotyp versehenes Paket enthält ebenfalls ein Objektmodell, wie Pakete der linken Regelseite.

3. *Rule* ist das Stereotyp, das ein Paket als BOTL-Regel deklariert. Ein solches Paket enthält genau eine linke und eine rechte Regelseite, also zwei Pakete mit den jeweils oben genannten Stereotypen.

4. Ein *Rule Set* ist ein Paket, das beliebig BOTL-Regeln und damit auf Paketebene beliebig viele Pakete mit dem Stereotyp Rule enthält.

Außer den jeweils oben beschriebenen Inhalten dürfen alle Pakete keine weiteren Elemente enthalten. Neben den Stereotypen enthält das UML-Profil für BOTL Eigenschaftswerte. Attribute in den Objekten der Regelseiten können die Eigenschaft *Term* besitzen, deren Wert ein Ausdruck ist. Die so spezifizierten Terme werden wie in Kapitel 3.1 beschrieben für die Attributberechnung eingesetzt. Wird die Eigenschaft Term für ein Attribut einer Regelseite nicht angegeben, so ist der Wert implizit $\Diamond$. In Abbildung 9 ist dies beim Attribut att3 im Objekt BClass der

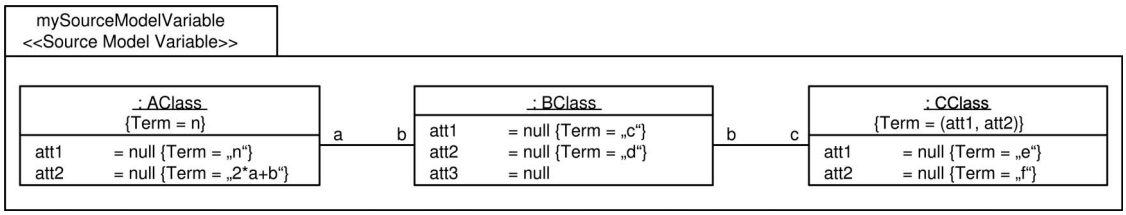

Abbildung 9: Eine linke Regelseite, wie sie im UML-Profil dargestellt wird. (Aus [13])

Fall. Da in UML-Werkzeugen oftmals nur der einfache ASCII Zeichensatz verwendet werden kann, wird statt $\Diamond$ das Zeichen „#“ benutzt. Für Objekte gibt der Wert der Eigenschaft Term den Objektidentifikator an. Objekt AClass hat demnach den Objektidentifikator n, also den Wert des Attributes att2. Bei dem Objekt CClass ist der Objektidentifikator das Tupel $(\mathsf{att1}, \mathsf{att2})$. Für linke Regelseiten, also Pakete mit dem Stereotyp Source Model Variable kann die Eigenschaft Metamodel als Zeichenkette definiert werden. Sie enthält als Wert den Namen des Klassenmodells, von dem das in der Regelseite enthaltene Objektmodell eine Instanz ist.

3.3 Das mathematische Modell

Der Aufbau und die Anwendung von BOTL-Regeln, die in Kapitel 3.1 in informaler Weise beschrieben wurden, basieren auf einem mathematischen Modell, das von Frank Marshall und Peter Braun entwickelt und in [13] beschrieben wird. Um einen kleinen Einblick zu geben, wie objektorientierte Modelle und Regeln darauf formalisiert werden können, wird in diesem Kapitel exemplarisch der Begriff „Klasse“ formal definiert. Dazu werden zunächst einige grundlegende Mengen definiert, die für die Definition der Klasse benötigt werden.

Definition 1 (Bezeichner, Typ, Typbelegung) *Die unendliche Menge aller Bezeichner wird im Folgenden mit $\mathbb{ID}$ bezeichnet. Die Menge alle Typen $\mathbb{T}$ entspricht der Menge alle Tupel (t, val), wobei $t \in \mathbb{ID}$ und val eine Wertmenge ist. Eine Typbelegung TA ist eine Menge von Typen mit paarweise verschiedenen Bezeichnern.*

Im Folgenden wird der Begriff „Klasse“ als ein Tupel definiert.

Definition 2 (Klasse) *Die Menge aller Klassen wird mit $\mathbb{C}$ bezeichnet. Eine Klasse c ist ein Tupel $(id, Super, A, Keys)$ wobei $id \in \mathbb{ID}$ einen Bezeichner für die*

Klasse festlegt, $Super \subset \mathbb{C} \setminus \{c\}$ *die Menge der Basisklassen,* $A \subset \mathbb{ID} \times TA$ *die Menge der Attribute* (n, t) *mit einem Bezeichner* n *und einem Typ* t *und* $Keys \subset A$ *die Menge der Primärschlüsselattribute festlegt.*

Des Weiteren muss gelten, dass eine Klasse alle Attribute der Basisklassen enthält ($A \supset Super|_A$*) und dass die Bezeichner der Attribute innerhalb der Klasse eindeutig sind, dass also gilt:* $\forall(n_i, t_i), (n_j, t_j) \in A : n_i = n_j \Rightarrow t_i = t_j$*. Darüberhinaus muss die Vererbungsbeziehung azyklisch sein.*

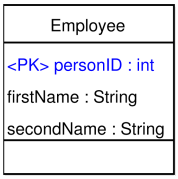

Abbildung 10: Die Klasse Employee. (Aus [13])

Die in Abbildung 10 dargestellte Klasse Employee wird im Formalismus dargestellt als Tupel:

$$(Employee, \emptyset,\\ \{(personID, \text{int}), (fistName, \text{string}), (secondName, \text{string})\},\\ \{(personID, \text{int})\})$$

Der erste Eintrag $Employee$ des Tupels ist der Bezeichner der Klasse. Es folgt die leere Menge der Basisklassen. Der dritte Eintrag ist die Menge der Attribute der Klasse bestehend aus dem Attributbezeichner und dem Typ gefolgt von der Menge der Schlüsselattribute, die im Beispiel nur aus dem Attribut $personID$ besteht.

3.4 Bewertung von BOTL-Regelwerken

Um BOTL für eine automatisierte Transformation von Modellen zu nutzen, muss das eingesetzte Regelwerk zwei wichtige Eigenschaften haben: Regelwerke müssen (1) anwendbar und (2) metamodellkonform sein. Ein Regelwerk ist *anwendbar*, wenn es aus einem beliebigen gültigen Quellmodell deterministisch und ohne Fehler ein Zielmodell erzeugt. Dass ein Regelwerk nicht anwendbar ist, kann drei Ursachen haben:

1. Durch Anwendung von mehreren Regeln oder durch mehrfache Anwendung einer Regel können im Zielmodell zwei Objekte mit gleichem Objektidentifikator entstehen. Im Normalfall werden Objekte mit gleichen Objektidentifikator verschmolzen. Ist dies nicht möglich, da die Objekte für ein Attribut unterschiedliche Werte ungleich $\Diamond$ haben, bricht die Anwendung der Regeln mit einem Fehler ab.

2. Terme innerhalb der Objekte auf der rechten Seite einer Regel können ungebundene, also undefinierte, Variablen besitzen. Da in diesem Fall kein Attribut deterministisch berechnet werden kann, bricht die Transformation ab.

3. Terme können unter einer Belegung der Variablen nicht berechenbar sein. Beispiele dafür sind der Term $\sqrt{x}$ wobei $x = -1$ ist oder $1/x$ mit $x = 0$.

Die beiden letzten Bedingungen definieren die *Anwendbarkeit einer Regel* und können somit pro Regel nachgewiesen werden. Die erste Bedingung kann nur für ein gesamtes Regelwerk nachgewiesen werden.

Ein Regelwerk ist *metamodellkonform*, wenn es nur zu dem Zielmetamodell konforme Modelle erzeugt. Insbesondere müssen dafür die folgenden Eigenschaften erfüllt sein:

- Es dürfen nur im Zielmetamodell erlaubte Objekte erzeugt werden.

- Es dürfen nur im Zielmetamodell erlaubte Objektassoziationen erzeugt werden.

- Die Assoziationen müssen die Multiplizitätsbedingungen erfüllen.

Die beiden ersten Bedingungen können sehr leicht pro Regel überprüft werden. Die dritte Bedingung muss aufwendiger für das ganze Regelwerk gezeigt werden.

In [13] finden sich Methoden, um die Anwendbarkeit und die Metamodellkonformität für ein Regelwerk nachzuweisen. Die dafür benötigten Konzepte sind so formal und deterministisch gehalten, dass die Nachweise automatisiert geschehen können. Dies wurde in der Arbeit auch durch eine tatsächliche Implementierung belegt. Auf eine Vorstellung dieser Methoden wird an dieser Stelle verzichtet, da sie den Rahmen dieser Arbeit sprengen würde.

4 Beispiel für einen Prozess: KOGITO mit BOTL

Um zu zeigen, dass BOTL in einem Entwicklungsprozess sinnvoll eingesetzt werden kann, zeigt Frank Marschall in [13], wie BOTL die KOGITO-Methodik für das Requirements-Engineering erweitert. In Kapitel 4.1 wird zunächst die KOGITO-Methodik vorgestellt. Im Anschluss wird in Kapitel 4.2 der Ansatzpunkt für BOTL darin gezeigt und welche Gemeinsamkeiten diese Kombination mit einem MDA-Prozess hat.

4.1 Die KOGITO-Methodik

Für das Requirements-Engineering werden in der Regel keine modellbasierten Prozesse eingesetzt. Der Grund dafür ist im Wesentlichen, dass ein Metamodell für ein Anforderungsmodell sehr groß und damit unübersichtlich sein müsste, um alle möglichen Anforderungen für alle möglichen Softwareprodukte abzudecken. Die Formalisierung bei einem modellbasierten Requirements-Engineering-Prozess hätte jedoch, im Gegensatz zu einem rein textbasierten Ansatz, wie er ansonsten üblich ist, den Vorteil, dass die erhobenen Anforderungen in weiten Teilen automatisiert auf Widerspruchsfreiheit und Konsistenz überprüft werden können. Die KOGITO-Methodik umgeht den Nachteil des großen Metamodells, indem sie nicht den Prozess des Requirements-Engineering im Allgemeinen, sondern nur für die Entwicklung von Web-basierten B2B[1]-Anwendungen abbildet.

Die KOGITO-Methodik verwendet vier konzeptuelle Metamodelle, die in Abbildung 11 dargestellt sind. Das *Business Requirements Model* (*BRM*) enthält Informationen über die Anwendungsdomäne, also in sehr grober Form eine Beschreibung der vom Softwareprodukt zu lösenden Aufgabe, der beteiligten Aktoren und Güter. Das *Requirements Analysis Model* (*RAM*) beschreibt die Anforderungen an das System und die wesentlichen Teile der Umgebung in einer technologieunabhängigen Art. Das *Detailed Requirements Model* (*DRM*)beschreibt schließlich das zu entwerfende System, wobei die Anforderungen spezifisch für die zu verwendende Technologie sind. Die drei hier beschriebenen Modelle entsprechen in ihrer Funktion den Ebenen CIM, PIM und PSM in der MDA.

Das *Business Reference Model* wird im ersten Schritt erstellt und dient als Glossar für die Entwickler auf der einen Seite und die fachlichen Ansprechpartner auf der anderen Seite. Neben der Klärung fachlicher Begriffe, werden dort auch fachliche Zusammenhänge festgelegt. Das Business Reference Model ist das abstrakte-

[1] Business to Business

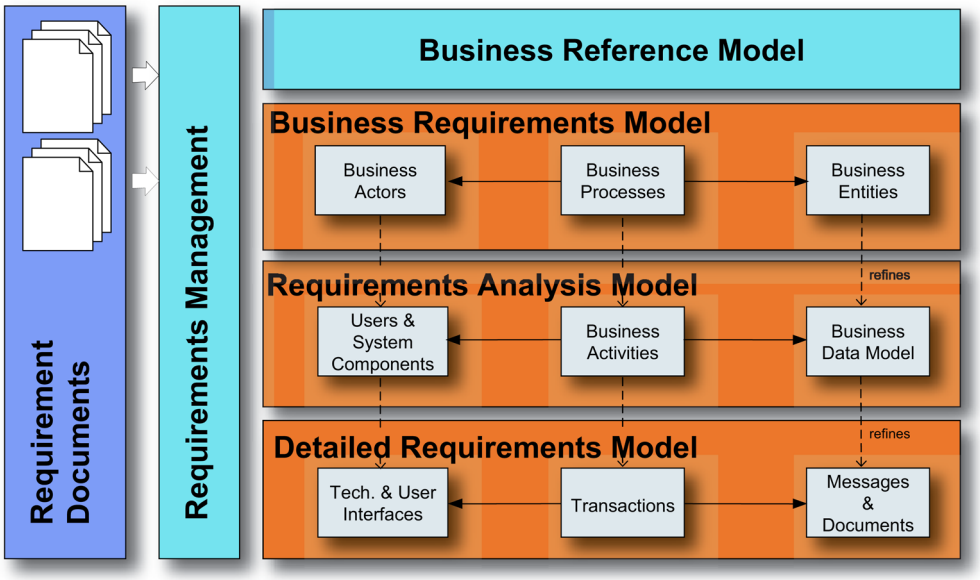

Abbildung 11: Schematische Darstellung der bei KOGITO verwendeten Metamodelle (Aus [13]). Von oben nach unten werden die Modelle verfeinert. Die roten Metamodelle entsprechen im Wesentlichen den Metamodellen der MDA.

ste Modell und damit gut wiederverwendbar, so dass in der Literatur schon eine Vielzahl an Modellen existiert, wie z.B. das Supply Chain Operations Reference Model [3] oder das VICS-Modell [4].

Die oben vorgestellten konzeptuellen Metamodelle sind in der UML in Form von Klassenmodellen definiert. Die Formalisierungen sind in [13] zu finden. Da die Instanzen dieser Metamodelle schon bei einer kleinen Zahl an Anforderungen eine große Anzahl an Objekten haben, sind diese Modelle durch ihre Unübersichtlichkeit sehr schlecht wartbar. Sie werden daher vor dem Anwender verborgen und durch Sichten ersetzt, die durch verschiedene Artefakttypen definiert werden. Aus den vom Anwender erstellten Artefakten werden dann die internen Modelle erzeugt.

Die Verwaltung der Artefakte erfordert zwei weitere Bausteine, die orthogonal zu den beschriebenen Modellen liegen. Zum einen werden *Requirement Documents* als Menge der Artefakte benötigt. Zum anderen wird das *Requirements Management* eingeführt. Letzteres weist die Artefakte den einzelnen Modellen der rechten Seite zu.

Im Folgenden werden für die verschiedenen Modelle jeweils die Typen der Artefakte vorgestellt, aus denen das Modell zusammengesetzt wird. Die Artefakte

werden von den Entwicklern und den fachlichen Experten erstellt. Insgesamt wird dafür als Notation strukturierter Text und in einigen Fällen die UML verwendet, da sich gezeigt hat, dass diese auf der einen Seite eine hohe Akzeptanz unter den Anwendern findet, auf der anderen Seite eine höhere Strukturierung aufweist, als der sonst häufig verwendete nicht strukturierte Text. Die Liste aller Artefakttypen würde den Rahmen dieser Arbeit sprengen. Daher findet sich hier nur ein Auszug aus der vollständigen Liste in [13].

Das Business Reference Model, das meistens von fachlichen Experten erstellt wird, enthält Informationen über die Geschäftsfelder, Prozessfelder, Geschäftsprozesse und Stakeholder[2]. Für jeden der vier Bereiche existiert ein Formular, das Informationen über die entsprechende Entität einsammelt und in das Modell einfügt. Die Felder in den Formularen sind Freitextfelder und entsprechen den Attributen der Klassen im BRM-Metamodell. Ein Beispiel für ein solches Formular findet sich in [13].

Das BRM enthält ebenfalls das Formular zum Einfügen von Geschäftsprozessen. Daneben enthält es Formulare zur Definition von Gütern und Aktoren der Geschäftswelt und mit Anwendungsfalldiagrammen, die einfachste Form von formalen UML-Diagrammen. Die Artefakte des RAM sind schließlich nur noch Diagramme. Hier werden Klassen- und ER-Diagramme verwendet, um das statische Modell darzustellen, und Aktivitätsdiagramme für das dynamische Modell. Beim Einfügen der hier aufgezählten Diagramme in die jeweiligen BRM bzw. RAM, werden diese, um die Metamodellkonformität zu gewährleisten, intern in Objektmodelle übersetzt werden.

4.2 Verfeinerung der konzeptuellen Modelle mit BOTL

Bei der MDA sollen große Teile der Modelltransformationen automatisiert geschehen. Da im KOGITO-Prozess Klassenmodelle als Metamodelle und Objektmodelle als konzeptuelle Modelle verwendet werden, kann BOTL mit Hilfe von geeigneten Regelwerken eingesetzt werden um diese Transformation durchzuführen. In [13] ist ein Regelwerk R_{BRM} bestehend aus 5 Regeln angegeben, das ein BRM in ein RAM transformiert.

Nachdem das Grundgerüst für das RAM aus dem BRM erstellt ist, muss dieses durch die Integration von Artefakten verfeinert werden. Abbildung 12 zeigt diesen Zusammenhang schematisch. Um zu zeigen, dass auch diese Schritte mit BOTL möglich sind, sind die Regelwerke R_{ER} und R_{AD} in [13] ebenfalls formal angegeben.

[2]Personen, die in unmittelbarem Bezug zur zu entwickelnden Software stehen

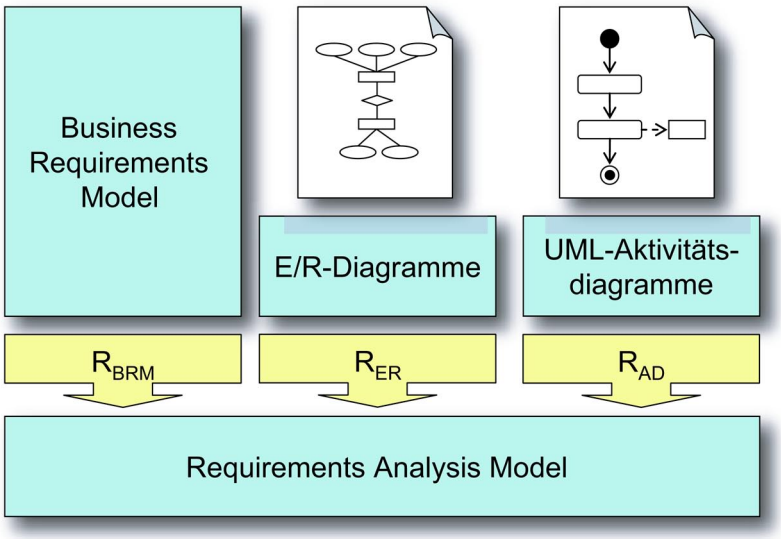

Abbildung 12: Darstellung der Transformation vom BRM zum RAM. Daneben werden weitere Artefakte in das RAM eingefügt. (Aus [13])

5 Zusammenfassung und Diskussion

Die Verwendung von Modellen in der Softwareentwicklung senkt die Kosten, da zum einen die Arbeit an abstrakten, und damit oftmals verhältnismäßig übersichtlichen Modelen leichter und damit weniger fehleranfällig ist als die Arbeit am Quellcode und zum anderen Modelle besser wiederzuverwenden sind. Beim MDA-Ansatz liegt daher der Schwerpunkt auf der Entwicklung von Modellen, die immer weiter verfeinert werden, bis in einem letzten Schritt im Optimalfall Quellcode entsteht. Die Verfeinerung der Modelle kann mit Hilfe von Transformationssprachen automatisiert werden. Frank Marshall hat 2003 zusammen mit Peter Braun mit der Bidirektionalen Objektorientierten Transformationssprache BOTL eine Sprache vorgestellt, mit der solche Modelltransformationen automatisiert durchgeführt werden können und beschrieben, wie diese Sprache in der KOGITO-Methodik für das Requirements Engineering benutzt werden kann. Bei der KOGITO-Methodik werden die Anforderungen sehr sorgfältig erfasst und mit Hilfe von BOTL immer weiter verfeinert, bis schließlich alle Anforderungen an das zu entwickelnde System im Requirements Analysis Model vorliegen.

Bezogen auf die MDA ist dieser Ansatz sicherlich ein Schritt in die richtige Richtung, jedoch ist die Sprache und der KOGITO-Prozess noch weit davon entfernt, ein MDA-Prozess zu sein, bei dem die gesammte Anwendung modelliert wird und die eigentliche Programmierung entfällt. Außerdem ist dieser Entwicklungs-

prozess noch nicht universell einsetzbar, sondern kann nur für Webbasierte B2B-Anwendungen verwendet werden.

Bisher wurde nur gezeigt, wie das Requirements-Enginieering von BOTL profitieren kann. Was bisher noch nicht gezeigt wurde, ist wie bzw. ob BOTL den Anwender bei den folgenden Schritten, der Architekturerstellung und der eigentlichen Implementierung, also der Erzeugung von Code, unterstützt. Da die Anforderungen zumindest in der Theorie vollständig im RAM formal vorliegen, wäre eine automatische Generierung des Codes, zumindest von Teilen der Software, denkbar. Ein Bereich, der in der Arbeit noch gar nicht angesprochen wurde, ist die Modellierung einer (graphischen) Benutzerschnittstelle. Eine Drei-Schichten-Architektur vorausgesetzt, wären also – wenn überhaupt – nur die beiden unteren Schichten automatisch generierbar.

Die Sprache BOTL ermöglicht die Transformation von Modellen in verfeinerte Modelle. Werden die verfeinerten Modelle jedoch verändert während gleichzeitig die Ursprungsmodelle verändert werden, so gibt es keine Möglichkeit diese Änderungen nach einer erneuten Transformation zusammenzuführen, da keine Informationen über die Zusammenhänge zwischen den Ursprungsobjekten und den transformierten Objekten abgespeichert wurden. Dieses Problem versucht Alexander Königs in seiner Arbeit „Model Transformation with Triple Graph“ [12] zu umgehen, indem er die benötigten Informationen in einem weiteren Graph ablegt. Diese Arbeit wird im folgenden Seminarbeitrag vorgestellt.

Literatur

[1] *Lex Homepage*. `http://dinosaur.compilertools.net/lex/index.html`.

[2] *Object Management Group homepage*. `http://www.omg.org/`.

[3] *Supply-chain council homepage*. `http://www.supply-chain.org/`.

[4] *Voluntary interindustry commerce standards association homepage*. `http://www.vics.org/`.

[5] *Yet Another Compiler Compiler homepage*. `http://dinosaur.compilertools.net/yacc/index.html`.

[6] *Unified Modeling Language: Infrastructure version 2.0*. Technischer Bericht, Object Management Group, Mar 2006. `http://www.omg.org/technology/documents/formal/uml.htm`.

[7] BALZERT, HELMUT: *Lehrbuch der Software-Technik: Teil 1: Software-Entwicklung*. Spektrum Akademischer Verlag, Heidelberg, Germany, 1996.

[8] BRAUN, PETER und FRANK MARSCHALL: *Transforming Object Oriented Models with BOTL*. Electronic Notes in Theoretical Computer Science, 72(3):15, 2003. http://www4.in.tum.de/publ/papers/entcs-bm02.pdf.

[9] KEMPA, DR. MARTIN und ZOLTÁN ÁDÁM MANN: *Model Driven Architecture*. Technischer Bericht, sd&m AG software design & management, 2005. http://www.gi-ev.de/service/informatiklexikon/informatiklexikon-detailansicht/meldung/117/.

[10] KLEPPE, ANNEKE G., JOS WARMER und WIM BAST: *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[11] KURT, CHRIS und LUIS FELIPE CABRERA: *Web Services Architecture and Its Specifications*. Microsoft-Press, Mar 2005.

[12] KÖNIGS, ALEXANDER: *Model Transformation with Triple Graph Grammars*. Real-Time Systems Lab, 2005.

[13] MARSCHALL, FRANK: *Modelltransformationen als Mittel der modellbasierten Entwicklung von Software-Systemen*. Doktorarbeit, Technische Universität München, Institut für Informatik, Garching, Germany, Jun 2004. http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2005/marschall.pdf.

[14] SCHÄTZ, BERNHARD, ALEXANDER PRETSCHNER, FRANZ HUBER und JAN PHILIPPS: *Model-Based Development*. Technischer Bericht, Institut für Informatik, Technische Universität München, May 2002. http://wwwbib.informatik.tu-muenchen.de/infberichte/2002/TUM-I0204.ps.

[15] SIEGEL, JON: *CORBA 3, Fundamentals and Programming*. Wiley Computer Publishing, New York, USA, 2000.

[16] STACHOWIAK, HERBERT: *Allgemeine Modelltheorie*. Springer, Wien, Austria, 1973.