FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Automatisiertes White-Box Testen für regelbasierte Modelltransformationen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Business Informatics

eingereicht von

Thomas Franz, BSc
Matrikelnummer 0627583
Sabine Wolny, BSc
Matrikelnummer 0725659

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: O.Univ.Prof. Dipl.-Ing. Mag. Dr.techn. Gertrude Kappel
Mitwirkung: Univ.Ass. Mag.rer.soc.oec. Dr.rer.soc.oec. Manuel Wimmer

Wien, 27.08.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

Erklärung zur Verfassung der Arbeit

Thomas Franz, BSc
Am Eichgraben 1, 2122 Ulrichskirchen

Sabine Wolny, BSc
Treffzgasse 17, 1130 Wien

Hiermit erklären wir, dass wir diese Arbeit selbständig verfasst haben, dass wir die verwendeten Quellen und Hilfsmittel vollständig angegeben haben und dass wir die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht haben.

(Ort, Datum) (Unterschrift Verfasser)

Danksagung

Zu Beginn dieser Diplomarbeit möchte ich mich bei all jenen bedanken, die diese Arbeit ermöglicht haben, beginnend bei meiner Familie, die mich in allen Phasen des Studiums, sowohl leichte als auch herausfordernde, moralisch unterstützt und somit das Studium zu einem spannenden Lebensabschnitt gemacht hat. Mein Dank gilt meinen Eltern, die mir nicht nur das Studium finanziell ermöglicht haben, sondern mir durch konstruktive Kritik gezeigt haben, dass ich mehr bewerkstelligen kann, als ich am Anfang für möglich gehalten habe. Weiters möchte ich mich bei meinen Schwestern Brigitte und Martina bedanken, die mir mit Rat und Tat vor allem bei der Diplomarbeit zur Seite gestanden sind und durch ihre Anmerkungen so manche neue Idee entstehen ließen.

Für die gute Betreuung und fachliche Beratung bei der Erstellung der Diplomarbeit möchte ich mich herzlich bei Herrn Univ.Ass. Mag. Dr. Manuel Wimmer und Frau O.Univ.Prof. Dipl.-Ing. Mag. Dr. Gertrude Kappel bedanken.

In diesem Zusammenhang sei auch Herrn Ao.Univ.Prof. Mag Dr. Tompits und vor allem Herrn Dipl.-Ing. Christoph Redl gedankt, die mich fachkundig unterstützt haben.

Ein besonderer Dank gilt meinem Kollegen Thomas Franz, der mit mir diese Arbeit verfasst hat. Seine großartige Unterstützung und vor allem seine amüsanten Anmerkungen machten die Arbeit an der Diplomarbeit zu einer lehrreichen aber auch unterhaltsamen Zeit.

Ich möchte mich auch bei meinem Freund Florian bedanken, der mir liebevoll zur Seite stand und doch für den rechten Ausgleich zwischen konstruktiver Arbeit und Entspannung sorgte.

Meinen Freunden und Studienkollegen sei für eine schöne, lehrreiche und auch lustige gemeinsame Studienzeit gedankt. Abschließend möchte ich mich bei all jenen bedanken, die mein Vorankommen förderten und noch nicht namentlich erwähnt wurden.

Danksagung

Zuallererst möchte ich mich bei meinen Eltern, Adelheid und Robert, dafür bedanken, dass sie mich bei meinem Studium sowohl finanziell als auch moralisch unterstützt haben. Vor allem ihre Geduld und Motivation, wenn es mal nicht so ideal gelaufen ist, haben mir sehr geholfen. Und natürlich auch bei meiner Schwester Claudia, die es immer wieder geschafft hat meinen Ehrgeiz, das Studium möglichst rasch und erfolgreich zu beenden, aufs Neue anzufachen.

Bedanken möchte ich mich an dieser Stelle auch bei meinen Betreuern, Frau O.Univ. Prof. Dipl.-Ing. Mag. Dr. Gertrude Kappel und Herrn Univ. Ass. Mag. Dr. Manuel Wimmer, für ihre Anregungen und konstruktive Kritik. Dadurch taten sich nicht selten neue Blickwinkel auf und ich konnte viele neue Erfahrungen und neues Wissen sammeln.

Ein besonderer Dank geht an Sabine Wolny für die ertragreiche und unkomplizierte Zusammenarbeit beim Schreiben dieser Diplomarbeit. Ihre Unterstützung und Anregungen haben bei der Lösung so manchen Problems den entscheidenden Durchbruch ermöglicht.

Weiters möchte ich mich bei meinen Studienkollegen für eine schöne und unterhaltsame Zeit während, aber auch abseits des Studiums bedanken, an die ich mich immer gerne und mit Freude erinnern werde.

Abstract

For several years, *Model Driven Software Development (MDSD)* is on its rise and software models are not only used for design purposes but became a major element in the software engineering process. Modeling languages are available to describe models which are the basis for further development. In addition to the modeling languages, model transformations are an important ingredient of model driven software development. Basically, the goal of a model transformation is to transform a given source model to a desired target model.
Often the model transformation fails and the troubleshooting is lengthy and time-consuming because the mistakes can be in the transformation itself or in the metamodels. Another problem is to obtain a good test coverage.
The aim of this work is to find an automated process which can test rule-based transformations. A tool is to be developed in *Java*, which is based on the meta language *Ecore* and model transformations in *Atlas Transformation Language (ATL)*. One of the goals of the thesis is to find an automated *white-box testing* approach for model transformations in ATL. By the automatic generation of test instances a high test coverage should be ensured. In addition, various errors and especially runtime errors in the transformation should be detected.
Beside the necessary basics such as *Answer Set Programming (ASP)* and *testing* in general the concrete implementation of the approach, the *White-Box Testing (WBT) Tool*, is described. Based on a collection of model transformations of the course "Model Engineering" of the Technical University of Vienna from the years 2009 until 2012 an evaluation is carried out and discussed. The method of evaluation is a case study with an objective test data set as the transformations are not created by ourselves. This set is significant with about 160 transformations.

The results of the case study show that the developed WBT-Tool finds different errors. For example, failures during the validation of the models are detected and the tool finds runtime errors in the model transformation. An essential advantage is that there is no interaction except the initial setting and therefore more complex test data can be tested automatically.

Kurzfassung

Seit mehreren Jahren ist die *Modellgetriebene Softwareentwicklung* (Englisch: *Model Driven Software Development (MDSD)*) auf dem Vormarsch und Softwaremodelle werden nicht mehr nur für Entwurfszwecke genutzt, sondern sind ein Hauptelement im Software Engineering Prozess geworden. Modellierungssprachen werden genutzt um Modelle darzustellen, die eine Basis für die weitere Entwicklung bilden. Neben den Modellierungssprachen spielen in der modellgetriebenen Softwareentwicklung auch Modelltransformationen eine entscheidende Rolle. Das Ziel einer Modelltransformation ist ein bestimmtes Quellmodell in ein gewünschtes Zielmodell zu transformieren.
Oftmals schlagen Modelltransformationen fehl und die Fehlersuche ist mühsam und langwierig, da diese in der Transformation selbst oder in den Metamodellen liegen können. Eine weitere Herausforderung ist es eine hohe Testabdeckung zu erreichen.
Ziel dieser Arbeit ist das Auffinden eines automatisierten Ablaufs, der regelbasierte Transformationen testen kann. Dabei soll ein Tool in *Java* implementiert werden, welches auf der Metamodellierungssprache *Ecore* und regelbasierten Modelltransformationen in der *Atlas Transformation Language (ATL)* aufbaut. Es wird ein automatisierter White-Box Testing Ansatz entwickelt. Außerdem sollen mit Hilfe einer automatischen Generierung von Testinstanzen eine hohe Testabdeckung sichergestellt und verschiedene Fehler, vor allem Laufzeitfehler in der Transformation aufgezeigt werden.
Neben den nötigen Grundlagen, wie zum Beispiel *Answer Set Programming (ASP)* und *Testen* im Allgemeinen, wird auf die konkrete Implementierung eines möglichen Ansatzes, das sogenannte *White-Box Testing (WBT) Tool*, eingegangen. Basierend auf den Modelltransformationen der Übungsaufgaben aus der Lehrveranstaltung "Model Engineering" der Technischen Universität Wien der Jahre 2009 bis 2012 wird eine Evaluierung des entwickelten Tools durchgeführt. Die Methode der Evaluierung ist eine Fallstudie mit einem objektiven Testdatenset. Es werden rund 160 unabhängige Transformationen getestet.

Die Ergebnisse der Fallstudie zeigen, dass das entwickelte WBT-Tool viele verschiedene Fehler, sowohl bei der Validierung der Modelle, als auch in der Transformation selbst aufzeigt. Das Tool birgt den Vorteil auch komplexere Testdatensets automatisch testen zu können, da bis auf die Anfangseinstellungen keine weitere Interaktion notwendig ist.

Inhaltsverzeichnis

KAPITEL 1

Einführung[1]

1.1 Motivation

Die modellgetriebene Softwareentwicklung *Model Driven Software Engineering (MDSE)* [13] gewinnt immer mehr an Bedeutung. Modelle und ihre zugehörigen Metamodelle sind nicht mehr nur für Entwurfszwecke wichtig, sondern entwickelten sich zu Kernelementen im Software Engineering Prozess. Um diese entwickeln zu können, benötigt man Modellierungssprachen. Eine der Bekanntesten ist die *Unified Modeling Language (UML)* [43], welche ein Standard der *Object Management Group (OMG)* [72] ist. Neben UML gewinnt auch die Modellierungssprache *Ecore* immer mehr an Bedeutung, da diese ein Hauptbestandteil des *Eclipse Modeling Framework (EMF)* ist und daher für Anwender in der Entwicklungsumgebung *Eclipse* als integrierter Bestandteil zur Verfügung steht.
Neben den Modellierungssprachen und den modellierten Metamodellen spielen in der modellgetriebenen Softwareentwicklung auch Modelltransformationen eine entscheidende Rolle. Modelltransformationen dienen dazu ein Quellmodell in ein gewünschtes Zielmodell zu transformieren. Auch hier gibt es viele verschiedene Modelltransformationssprachen, wobei die regelbasierte *Atlas Transformation Language (ATL)* [52] zunehmende Verbreitung findet.
Da diese Werkzeuge im Software Engineering Prozess immer mehr an Bedeutung finden und angewandt werden, ist es wichtig, dass eine gewisse Softwarequalität sichergestellt wird. Um diese zu erreichen, müssen auch Modelle und ihre Transformationen getestet werden um etwaige Fehler bereits früh erkennen und beheben zu können.

1.2 Problemstellung

In der modellgetriebenen Softwareentwicklung werden mit Hilfe von Modelltransformationen Quellmodelle in Zielmodelle übergeführt. Bei dieser Überführung müssen zumin-

[1] Verfasser: Sabine Wolny, BSc

dest folgende Bedingungen gelten:

- das Quellmodell muss konform zum Quellmetamodell sein (Metamodelle repräsentieren die Modellierungssprache, in der die Modelle formuliert werden)
- das Zielmodell muss konform zum Zielmetamodell sein
- die Transformation muss die Korrespondenzen zwischen Quell- und Zielmetamodell korrekt abbilden

Für eine wiederholte Anwendung von Transformationen muss der Benutzer wissen, wie gut diese tatsächlich funktionieren. Dazu gibt es bereits mehrere Ansätze in Form von Black-Box Tests, die anhand verschiedenster manuell erstellter Quellmodelle prüfen, ob die Transformation durchführbar ist, das heißt, ob sie ohne Laufzeitfehler terminiert. Außerdem wird geprüft, ob das Ergebnis (das Zielmodell) eine valide Instanz des Zielmetamodells ist. Allerdings ermöglicht dieser Ansatz keine genauen Aussagen darüber, welche Regeln ausgeführt beziehungsweise nicht ausgeführt wurden. Dadurch ist momentan keine tiefergehende Fehleranalyse möglich. Das Testen von Modelltransformationen gewinnt aber immer mehr an Bedeutung, da Fehler in der Transformation selbst liegen können, die nicht mit Hilfe von Black-Box Tests, sondern erst mit White-Box Tests entdeckt werden können. Daher werden zunehmend auch White-Box Ansätze zum Testen von Modelltransformationen entwickelt. Viele dieser Ansätze (ob Black- oder White-Box) erfordern Benutzerinteraktionen. Das manuelle Testen ist aber oft langwierig und schwierig, daher wird zunehmend nach Lösungen mit einem möglichst hohen Grad an Automatisierung gesucht.

1.3 Zielsetzung

Oftmals schlagen Transformationen fehl und die Fehlersuche gestaltet sich schwierig, da die Fehler entweder in der Transformation selbst oder in den (Meta-)Modellen liegen können. Diese Arbeit beschäftigt sich mit dem Testen von Modellen und ihren Modelltransformationen, die auf *Ecore* Metamodellen und der Modelltransformationssprache ATL basieren. Hierfür wird ein automatisierter Ablauf gesucht, der regelbasierte Transformationen testen kann. Neben dem Festlegen von theoretischen Bedingungen soll ein Tool in der Programmiersprache *Java* entwickelt werden. Um den Aufwand des Testens zu verringern, ist es zusätzlich erforderlich, die Anzahl an Testmodellen sinnvoll zu reduzieren, da im Allgemeinen die Menge an gültigen Modellen zu einem Metamodell zu groß ist, um alle beim Testen berücksichtigen zu können. Daher wird nur eine gewisse Anzahl an Modellen, welche wesentliche Bestandteile des Metamodells und der Modelltransformation enthalten, getestet. Durch das Definieren von Vorbedingungen, die vor der Ausführung der Transformation erfüllt sein müssen, kann die Anzahl noch verringert werden.

Aus diesen Anforderungen ergeben sich folgende Fragen und somit Ziele der Arbeit:

1. Ist es möglich einen White-Box Testansatz für ATL Transformationen zu finden?

2. Wie (effizient) können Testmodelle automatisch aus einem Metamodell generiert werden?

3. Wie hoch ist die Fehlererkennungsrate?

4. Welche Arten von Fehlern können festgestellt werden?

1.4 Methodisches Vorgehen

Die methodische Vorgehensweise kann in folgende Arbeitsschritte unterteilt werden:

1. Literaturrecherche
   Zunächst wird eine umfassende Analyse der bereits existierenden Ansätze und der damit verbundenen Arbeiten durchgeführt. Außerdem wird recherchiert, ob ein bereits existierender Ansatz für die neuen Anforderungen erweitert und adaptiert werden kann.

2. Implementierung und Testphase 1
   Nach erfolgreicher Analyse der bereits existierenden Ansätze besteht der nächste Teil der Arbeit in der Entwicklung und Implementierung eines Tools, welches die geforderten Ziele erfüllt. Für die Implementierung wird die Einschränkung getroffen, dass nur die Metamodellierungssprache Ecore und die regelbasierten Modelltransformationen in der ATL verwendet werden und das Tool in Java entwickelt wird. Während der Forschungsarbeit wird nach mehreren Tests der erste Ansatz einer Adaption und Erweiterung eines existierenden Programms *EMFtoCSP* nicht weiterverfolgt, da hier aufgrund der fehlenden Dokumentation des Sourcecodes keine Möglichkeit zur Adaption in einem adäquaten Zeitrahmen gefunden wird. Der zweite, nämlich ein rein Java-basierter Ansatz, zur Generierung von Modellen wird nach ein paar Tests ebenfalls verworfen. Der Grund dafür ist, dass er einige Einschränkungen aufweist, die durch die Komplexität der Aufgabe vorgegeben sind, wodurch auch der Java-Code eine nicht mehr vernünftig bewältigbare Komplexität erreicht.

3. Implementierung des White-Box Testing (WBT) Tools
   Nach dieser ersten Implementierungsphase wird ein Programm, welches aus drei eigenständigen Modulen aufgebaut ist, entwickelt.

   a) Modellgenerierung mit Answer Set Programming (ASP)
      Für die Modellgenerierung wird ein logisch orientierter Ansatz, nämlich ASP, genutzt um möglichst viele verschiedene Modelle zu erhalten.

   b) Bedingungen mit Object Constraint Language (OCL)
      Für die Modelle können mit Hilfe von OCL Bedingungen festgesetzt werden.

   c) Modelltransformation mit Atlas Transformation Language (ATL)
      Mit Hilfe von ATL werden die Modelltransformationen durchgeführt und gleichzeitig die ausgeführten Regeln aufgezeichnet.

Durch die Entwicklung einzelner Module kann das Tool flexibler eingesetzt werden, da die einzelnen Teile allein oder in beliebigen Kombination verwendet werden können. Abschließend wird für die transformierten Modelle eine Überprüfung umgesetzt. So werden diese Modelle auf ihre Konformität zum zugehörigen Metamodell überprüft und es können OCL Constraints für diese angegeben werden.

4. Evaluierung
   Zum Schluss wird mit Transformationen aus der Lehrveranstaltung "Model Engineering" der Technischen Universität Wien aus den Jahren 2009 bis 2012 getestet, ob das Tool die gestellten Anforderungen auch erfüllt. Es wird festgestellt, welche Fehler erkannt werden, wo und wie oft diese auftreten. Die Analyse soll zeigen, wie effizient das Programm arbeitet und welche Arten von Fehlern in Modelltransformationen festgestellt werden können.

1.5 Aufbau der Arbeit

Die Arbeit gliedert sich in 9 Kapitel, welche wiederum Unterkapitel umfassen.
Im Kapitel 2 wird ein Überblick zur modellgetriebenen Softwareentwicklung gegeben. Hier wird vor allem auf die unterschiedlichen Abstraktionsebenen und Metaebenen eingegangen, die existieren. Insbesondere werden zwei Standards der OMG, nämlich UML und Meta-Object Facility (MOF), näher erläutert und der Zusammenhang zur Metamodellierungssprache *Ecore* hergestellt. Außerdem wird im Abschnitt Transformationen der Terminus Modelltransformation erklärt und auf die unterschiedlichen Arten und Unterteilungen näher eingegangen.
Das nächste Kapitel 3 befasst sich mit dem Begriff des "Testens" in der Softwareentwicklung. Zuerst wird auf allgemeine Grundlagen, den Testprozess und verschiedene Testmethoden eingegangen. Anschließend wird der Fokus konkret auf Black-Box und White-Box Tests gelegt. Außerdem werden die für diese Arbeit relevanten Bereiche, wie Testautomatisierung, Testen im Model Engineering und Testen von Modelltransformationen, verdeutlicht.
Im Zuge dieser Arbeit wurde ein Tool entwickelt, welches im Kapitel 4 vorgestellt wird. Der Fokus dieses Kapitels liegt darauf einen Überblick über die Funktionsweise und die Zusammenhänge der einzelnen Module des Programms zu geben, wobei auf die technischen Aspekte der Umsetzung erst im Kapitel 6 näher eingegangen wird.
Kapitel 5 gibt einen Überblick zu Answer Set Programming, das bei der Entwicklung von Testinstanzen zum Einsatz kam. Zuerst werden die Architektur und Methodik näher erklärt und dann wird auf die Syntax und Semantik von Answer Set Solvern näher eingegangen. Abschließend werden kurze Beispiele für den Einsatz genannt.
Im Kapitel 6 wird das entwickelte Programm detailliert erklärt. Zuerst werden die eingesetzten Technologien aufgezeigt und dann werden die einzelnen Module, aus denen das Tool besteht, beschrieben. Hierbei wird auf die Erzeugung der Testinstanzen mit Hilfe von Datalog with disjunction (DLV), dann auf die Überprüfung von Constraints mit OCL und auf die Durchführung der Modelltransformation mit ATL eingegangen.

Außerdem wird das Tool als Ganzes genauer hinsichtlich seiner technischen Aspekte und Funktionalität erklärt und ein konkretes Beispiel für die Funktionsweise gegeben. Abschließend werden alternative Technologien vorgestellt.
Kapitel 7 legt den Fokus auf die Evaluierung des entwickelten Programms und die Analyse der Ergebnisse. Es wurde auf zwei verschiedenen Testumgebungen das Tool mit den Daten der Lehrveranstaltung "Model Engineering" aus den Jahren 2009 bis 2012 von der Technischen Universität Wien getestet und schließlich ausgewertet.
Natürlich gibt es neben dieser Arbeit schon einige andere wissenschaftliche Beiträge, die sich mit ähnlichen Problemen beschäftigen. Einige hiervon werden im Kapitel 8 näher betrachtet. Einerseits wird auf den Bereich der Modellgenerierung eingegangen und andererseits werden Ansätze zu Testverfahren im Bezug auf Black-Box und White-Box Tests vorgestellt.
Abschließend wird im Kapitel 9 eine Schlussbetrachtung und Ausblick auf zukünftige Arbeit gegeben.
Des Weiteren verfügt diese Arbeit über ein Abkürzungsverzeichnis, in dem die wichtigsten Abkürzungen, die in dieser Arbeit verwendet werden, mit ihren Bedeutungen aufgelistet sind. Ein Begriff wird bei seiner Erstnennung sowohl in Langform als auch mit der Abkürzung angegeben. Im weiteren Verlauf wird nur noch die Abkürzung verwendet.

1.6 Aufteilung der Arbeit

Die Arbeit wurde von zwei Autoren verfasst, wobei die einzelnen Unterkapitel von jeweils einer Person geschrieben wurden. Wenn ein gesamtes Kapitel nur von einem Autor verfasst wurde, ist die Autorschaft in einer Fußnote beim Titel vermerkt. Ansonsten wird bei den jeweiligen Unterkapiteln der Verfasser als Fußnote angegeben. Die Aufteilung sieht wie folgt aus:

- **1 "Einführung"**
  Verfasser: Sabine Wolny, BSc
- **2 "Modellgetriebene Softwareentwicklung"**
  Verfasser: Sabine Wolny, BSc
- **3 "Testen von Software"**
  Verfasser: Thomas Franz, BSc
- **4 "Überblick über das *White-Box Test Tool*"**
  Verfasser: Sabine Wolny, BSc
- **5 "Answer Set Programming"**
  Verfasser: Sabine Wolny, BSc
- **6 "*White-Box Testing Tool*"**
  Verfasser: Thomas Franz, BSc & Sabine Wolny, BSc

- **6.1 "Eingesetzte Technologien"**
  Verfasser: Sabine Wolny, BSc
- **6.2 "DLV - Erzeugung der Testinstanzen"**
  Verfasser: Sabine Wolny, BSc
- **6.3 "OCL - Überprüfung von Constraints"**
  Verfasser: Thomas Franz, BSc
- **6.4 "ATL - Durchführung der Modelltransformation"**
  Verfasser: Thomas Franz, BSc
- **6.5 "White-Box Testing Tool"**
  Verfasser: Thomas Franz, BSc
- **6.6 "Alternative Technologien"**
  Verfasser: Sabine Wolny, BSc

- **7 "Evaluierung"**
  Verfasser: Thomas Franz, BSc & Sabine Wolny, BSc
  - **7.1 "Zielsetzung"**
    Verfasser: Sabine Wolny, BSc
  - **7.2 "Aufbau der Fallstudie"**
    Verfasser: Sabine Wolny, BSc
  - **7.3 "Durchführung der Fallstudie"**
    Verfasser: Thomas Franz, BSc & Sabine Wolny, BSc
    - **7.3.1 "Modellgenerierung"**
      Verfasser: Sabine Wolny, BSc
    - **7.3.2 "Testumgebung - 1"**
      Verfasser: Thomas Franz, BSc
    - **7.3.3 "Testumgebung - 2"**
      Verfasser: Sabine Wolny, BSc
  - **7.4 "Diskussion der Ergebnisse"**
    Verfasser: Thomas Franz, BSc
  - **7.5 "Erkenntnisse"**
    Verfasser: Sabine Wolny, BSc

- **8 "Verwandte Arbeiten"**
  Verfasser: Thomas Franz, BSc & Sabine Wolny, BSc
  - **8.1 "Modellgenerierung"**
    Verfasser: Sabine Wolny, BSc
  - **8.2 "Testen von Modelltransformationen"**
    Verfasser: Thomas Franz, BSc

- **8.3 "Abgrenzung zum WBT-Tool"**
  Verfasser: Thomas Franz, BSc

- **9 "Schlussbetrachtung und Ausblick"**
  Verfasser: Thomas Franz, BSc & Sabine Wolny, BSc
  - **9.1 "Zusammenfassung"**
    Verfasser: Sabine Wolny, BSc
  - **9.2 "Ausblick"**
    Verfasser: Thomas Franz, BSc

KAPITEL 2

Modelgetriebene Softwareentwicklung[1]

Model Driven Software Engineering (MDSE) ist ein sehr breit gefächerter Bereich in der Softwareentwicklung, der immer mehr an Bedeutung gewinnt. In diesem Kapitel soll ein kurzer Überblick über die Möglichkeiten der modellgetriebenen Softwareentwicklung gegeben werden. Zuerst wird im Abschnitt 2.1 auf die Grundlagen wie Definition und Ziele eingegangen. Außerdem wird die Bedeutung von der OMG dargelegt. Danach wird im Abschnitt 2.2 die *Model Driven Architecture (MDA)* näher erläutert, wobei auch auf die Unterschiede zur MDSD eingegangen wird. Abschnitt 2.3 gibt einen Einblick in die Metamodellierung und ihre einzelnen Ebenen. In diesem Zusammenhang werden auch wichtige Begriffe wie Domäne, Syntax und Semantik näher erklärt. Der Abschnitt 2.4 beschäftigt sich mit der Definition und den verschiedenen Arten von Transformationen, sowie der Einteilung von Transformationssprachen.

2.1 Grundlagen der modellgetriebenen Softwareentwicklung

> *"Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen."* [94, S. 11]

Die *modellgetriebene Softwareentwicklung* (Englisch: MDSD) wird also nach Stahl et al. [94] mit drei Elementen beschrieben:

1. *Formale Modelle*
   Ein formales Modell beschreibt einen Aspekt der Software, wobei alle relevanten

[1] Verfasser: Sabine Wolny, BSc

Elemente wie zum Beispiel die Struktur, das Verhalten und die Funktion angegeben werden. Außerdem müssen diese formalen Modelle für jeden Benutzer immer dieselbe Auswertung haben [43]. Eine genauere Betrachtung findet sich im Abschnitt 2.3.

2. *Lauffähige Software*
   Das Ziel von MDSD ist am Ende ein lauffähiges Programm zu erhalten. Um eine ausführbare Software aus Modellen zu erzeugen, gibt es einerseits Generatoren und andererseits Interpreter [94]. Generatoren erzeugen aus dem Modell Quellcode in einer Programmiersprache wie Java. Interpreter hingegen lesen Modelle zur Laufzeit ein und führen je nach Inhalt bestimmte Aktionen durch [94].

3. *Automation*
   Aus einem Modell soll automatisch eine ausführbare Software entstehen. Das bedeutet, dass Modelle nicht mehr nur als Designwerkzeug dienen, sondern für die Implementierung von entscheidender Wichtigkeit sind. Sollte es in der Software eine Änderung geben, so wird diese nicht im Quellcode allein durchgeführt (dann hätte das Modell keinen Einfluss), sondern sie wird direkt im Modell vorgenommen. Diese Automation bedeutet aber nicht, dass die Software nur noch automatisch erzeugt wird, da die Modelle sehr wohl "händisch" erstellt und auch gewartet werden müssen [94]. Außerdem können bestimmte Teile der Implementierung nicht im Modell abgebildet und müssen nachher im generierten Code sehr wohl händisch hinzugefügt werden. So ist es zum Beispiel in einem Modell zwar möglich einen Methodenkopf zu definieren, jedoch nicht die Funktion dieser Methode.

Im Zusammenhang mit modellgetriebener Softwareentwicklung fallen oft auch Begriffe wie *Model Driven Development (MDD)* [89] oder *Model Driven Engineering (MDE)* [32].

Gründe für MDSD

Es gibt verschiedene Gründe warum MDSD im Softwareentwicklungsprozess immer mehr an Bedeutung gewinnt und eingesetzt wird.

- Abstraktion
  Die Möglichkeit auf einer höheren Ebene zu programmieren ist einer der wichtigsten Gründe für MDSD [94]. Durch die Modellierung von Modellen kann ein System unabhängig von der endgültigen technischen Implementierung beschrieben und dessen Komplexität abgebildet werden [94].

- Einheitliche Architektur
  Die automatisierte Überführung von formalen Modellen in ausführbare Software passiert einheitlich [94]. Es werden also alle Komponenten, die in einem Modell existieren, immer auf dieselbe Weise übersetzt und durch das Modell wird ein Rahmen festgelegt, welcher die Dokumentation der Architektur vereinfacht [94]. Natürlich ist es aber möglich, dass sich die Architektur im Laufe der Entwicklungsarbeit verändert und erweitert werden kann, wobei dies im Modell und somit an einer zentralen Stelle angepasst wird [94].

- Wiederverwendung
  Einer der wichtigsten Gründe für MDSD ist die Wiederverwendbarkeit von Architektur, Modellierungssprachen und Generatoren [94]. Wenn also Systeme entwickelt werden, die einander sehr ähnlich sind, kann es möglich sein, dass durch die Wiederverwendbarkeit von bestimmten Teilen Zeit und Kosten gespart werden.

- Interoperabilität und Portabilität
  Ein weiterer Grund für MDSD ist die Interoperabilität und Portabilität von Systemen. So verfolgt die modellgetriebene Softwareentwicklung unter anderem auch das Ziel unabhängig von Hersteller und Plattform zu sein. Dieses Ziel wird jedoch nie vollständig erfüllt werden können, da die Struktur von Modellen oft sehr wohl abhängig von der Plattform ist und daher die Überführung in eine neue Plattform sehr mühsam und langwierig sein kann [94]. Außerdem existieren in der lauffähigen Software auch händisch programmierte Teile, die nicht ohne weiteren Aufwand in eine andere Programmiersprache übergeführt werden können (z.B. Java Code in C++ Code) [94].

Das Ziel, das mit MDSD somit verfolgt wird, ist die Vermeidung von Redundanzen in der Softwareentwicklung und dadurch eine bessere Softwarequalität sicher zu stellen.

Die größte Herausforderung in diesem Zusammenhang an die modellgetriebene Softwareentwicklung ist die Erstellung des Modells. Dieses muss einerseits ausreichend detailliert modelliert werden, um zum Beispiel eine zufriedenstellende Codegenerierung zu ermöglichen, andererseits aber auch ausreichend generell modelliert werden, um mehrmals wiederverwendet zu werden. Außerdem müssen Modelle konform zu ihren Metamodellen sein (siehe Abschnitt 2.3). Dementsprechend ist ein Nachteil der MDSD der hohe Anfangsaufwand zur Erstellung eines aussagekräftigen Modells und auch zur Entwicklung eines Generators oder Interpreters.

Einer der Vorteile ergibt sich aufgrund der erhöhten Abstraktion, wodurch es möglich ist das Problem klarer zu beschreiben. Ein weiterer langfristiger Vorteil von MDSD ist auch der reduzierte Aufwand zur Softwareerstellung in Bezug auf Zeit und Kosten durch die automatisierte Erstellung von Code. Modelle stehen in der modellgetriebenen Softwareentwicklung also im Mittelpunkt des Prozesses, dementsprechend haben sie auch einen eignen Lebenszyklus (siehe Abbildung 2.1). Softwareentwicklungen bauen daher von Anfang an auf Modellen auf, bei Änderungen werden diese immer wieder mit adaptiert und der Prozess wiederholt sich von vorne.

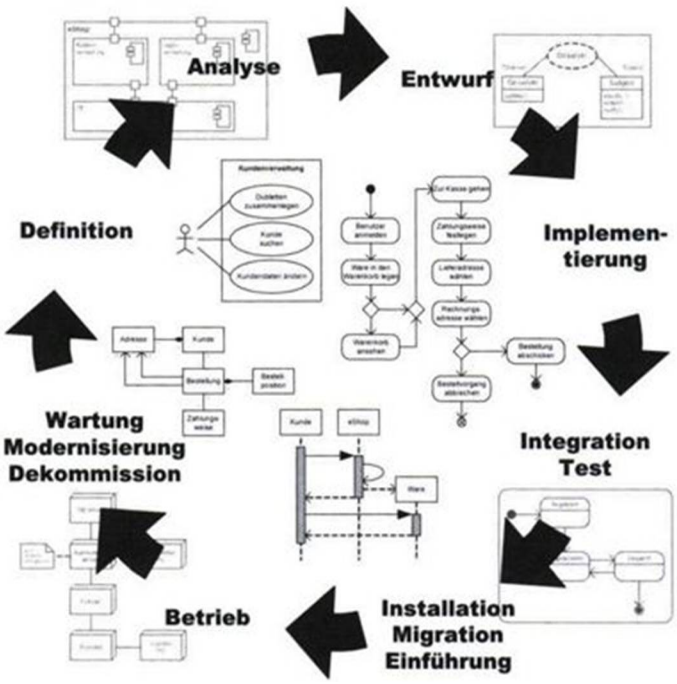

Abbildung 2.1: "Modelle im Mittelpunkt des Software-Lebenszyklus" [43, S. 20]

Object Management Group (OMG)

Die *Object Management Group (OMG)*[2] ist ein internationales, offenes, gemeinnütziges Industriekonsortium, welches 1989 gegründet wurde [72]. Das Hauptziel der OMG ist die Spezifikation von unabhängigen Standards, welche die Interoperabilität und Portabilität von Softwaresystemen erhöhen soll [94]. Standards des Konsortiums sind unter anderem *Business Process Modeling Notation (BMPN)*, *Common Object Request Broker Architecture (CORBA)* und *Data Distribution Service (DDS)* [73]. Im Bereich der Modellierung und Metadaten Spezifikation wurden zum Beispiel die Standards MDA, MOF, UML entwickelt [73]. Eine genaue Liste aller momentan existierenden Standards der OMG mit dem Bereich, dem sie zugeordnet sind, ist auf der Webseite [73] verfügbar.

2.2 MDA - Model Driven Architecture

Die *Model Driven Architecture (MDA)*[3] ist ein Standard der OMG, wobei vor allem das Ziel der Interoperabilität und Portabilität der MDSD verfolgt wird [94]. MDA ist sehr ähnlich zu MDSD, wobei es aber im Detail verschiedene Unterschiede gibt, wie zum Beispiel die Fokussierung auf UML [94]. In der modellgetriebenen Softwareentwicklung wird versucht Modellelemente zur Nutzung für den Softwareentwicklungsprozess

[2] `http://www.omg.org/`, Zuletzt besucht:: 30-07-2013
[3] `http://www.omg.org/mda/`, Zuletzt besucht:: 30-07-2013

zur Verfügung zu stellen und zwar unabhängig von der konkreten Werkzeugwahl. Nach Stahl et al. [94] ist MDA die Standardisierungsinitiative zur modellgetriebenen Softwareentwicklung. Die Grundidee der MDA ist es Softwarekomponenten unabhängig von der technischen Umsetzung zu spezifizieren und die dafür notwendige Infrastruktur/Architektur zur Verfügung zu stellen [43].

2.2.1 Ziele der MDA

Die Ziele der MDA sind sehr ähnlich zu denen von MDSD. Einige von diesen sollen hier nun genannt werden.

- Höhere Wartbarkeit
  Durch die klare Trennung der Systemfunktionalität von der tatsächlichen Implementierung ist auch eine Trennung von den Verantwortlichkeiten möglich und dadurch soll eine bessere Wartbarkeit ermöglicht werden [94].

- Portabilität
  Mit Hilfe der MDA sollen das System und technische Aspekte abstrakter beschrieben werden. Dadurch lässt sich das System auch später besser auf neue Versionen oder andere Zielumgebungen übertragen [43].

- Interoperabilität
  Mit Hilfe des Standards soll eine Herstellerunabhängigkeit ermöglicht werden. Dies wird vor allem durch die klare Trennung der Konzepte und der tatsächlichen Repräsentation bewerkstelligt. Dadurch lassen sich bestehende Komponenten auch zu einem späteren Zeitpunkt in einem Softwareentwicklungsprozess wiederverwenden [43].

Natürlich versucht die MDA die Softwareentwicklung mit den standardisierten Vorgaben effizienter zu gestalten und daher Zeit und Kosten zu sparen.

2.2.2 MDA Modelle

Nachdem die MDA das Ziel verfolgt, dass die Spezifikation eines Systems klar getrennt wird von der technischen Realisierung, wird dies auch in den Modellen beachtet. So werden zuerst Modelle unabhängig von technischen Details entwickelt. Diese werden erst im Zuge des Entwicklungsprozesses hinzugefügt. So ändert sich auch der Abstraktionslevel, auf dem sich die Modelle befinden.
Die wichtigsten Modelle der MDA werden nun kurz vorgestellt.

- *Computational Independent Model (CIM)*

  > "Das *berechnungs-unabhängige Modell* liefert eine Sicht auf das Gesamtsystem, die völlig unabhängig davon ist, wie es implementiert wird." [43, S. 27]

Das CIM beschreibt also die Anforderungen und die Umwelt eines Systems ohne konkret auf bestimmte Implementierungstechnologien einzugehen. In einem Domänenmodell beziehungsweise Geschäftsmodell wird mit Hilfe der in einer Modellierungssprache vorgegebenen Notationen das Softwaresystem abgebildet. In einem dazugehörenden Anforderungsmodell werden dann die Aufgaben des Systems näher beschrieben, jedoch ohne auf die konkrete Umsetzung derer einzugehen [43].

- *Platform Independent Model (PIM)*
  Im PIM wird die gesamte Funktionalität und Struktur des zu entwickelnden Systems dargestellt, wobei auch bei diesem keine plattformspezifischen Informationen abgebildet werden. Dadurch haben Änderungen bezüglich der zu nutzenden Plattform keinen Einfluss auf diese Modelle.

- *Platform Specific Model (PSM)*
  Wenn zu einem PIM noch plattformspezifische Informationen modelliert werden, so entsteht ein PSM. Wie der Name schon sagt, haben diese Modelle spezifische Informationen zu der Implementierungsplattform. Auf den PSMs basierend wird schließlich auch ausführbarer Quellcode erzeugt [43]. Es gibt verschiedene Codegeneratoren, die den Benutzer bei diesem Vorgang unterstützen.
  In diesem Zusammenhang sind Modelltransformationen ein wesentlicher Bestandteil. Eine genauere Betrachtung, welche Arten von Transformationen es gibt, findet sich in Abschnitt 2.4.

2.3 Metamodellierung

Der zentrale Begriff in der modellgetriebenen Softwareentwicklung ist das Modell. Modelle in verschiedenen Abstraktionsebenen sind die Basis des gesamten Entwicklungsprozesses. Zusammen mit weiteren wichtigen Begriffen, die in der Metamodellierung von Bedeutung sind, werden diese nun näher definiert.

Domäne

Im Zusammenhang mit MDSD fällt immer wieder auch der Begriff *Domäne*. Hierbei handelt es sich um ein abgegrenztes, zusammenhängendes Wissensgebiet [43]. Wichtig ist, dass die Komplexität einer Domäne nicht zu groß ist, damit unterschiedliche Domänen auch klar voneinander abgegrenzt werden können [94]. Es ist aber erlaubt Domänen in Subdomänen zu unterteilen.

Domänenspezifische Sprache

Die *Domain Specific Language (DSL)* ist, wie der Name schon verrät, die Programmiersprache zu einer Domäne [94]. Die DSL stellt also genau das Wissensgebiet einer Domäne mit Hilfe von Abstraktionen und Notationen dar [43]. Um dies bewerkstelligen zu können benötigt die Sprache sowohl eine eindeutige abstrakte als auch konkrete Syntax und eine eindeutige Semantik, damit alle Konzepte genau definiert werden können [94]. Eine domänenspezifische Sprache kann einerseits grafisch, wie zum Beispiel ein

Entity Relationship (ER) Modell, oder textuell, wie in der *Data Definition Language (DDL)*, dargestellt werden [43].

Abstrakte und konkrete Syntax

Ein weiterer wichtiger Begriff im Zusammenhang mit der Metamodellierung ist die Syntax. Hierbei muss klar zwischen abstrakter und konkreter Syntax unterschieden werden, wobei aber die Syntax für zum Beispiel einzelne Domänen immer eindeutig sein muss. Im Allgemeinen beschreibt eine Syntax, welche Elemente und Kombinationen dieser in zum Beispiel einem Modell vorkommen dürfen [43]. Die Syntax einer Sprache beschreibt also die Vorschriften und Regeln, welche Komponenten zur Sprache gehören oder eben nicht.

Abstrakte Syntax

> "Die *abstrakte Syntax* einer Sprache ist unabhängig von einer konkreten Darstellung und beschreibt, wie die Konzepte einer Sprache in struktureller Hinsicht in Beziehung zueinander stehen." [43, S. 68]

So könnte man zum Beispiel für die Programmiersprache Java folgende Aussage bezüglich ihrer abstrakten Syntax treffen: In Java gibt es unter anderem Klassen, die einen Namen besitzen, abstrakt sein können, von einer anderen Klasse erben, Methoden enthalten können und so weiter [94].

Konkrete Syntax

Die konkrete Syntax, im Gegensatz zur abstrakten, beschreibt, wie die einzelnen Elemente tatsächlich dargestellt werden [43]. Für die Programmiersprache Java wird mit Hilfe der konkreten Syntax zum Beispiel beschrieben, dass Klassen das Schlüsselwort *class* haben müssen [94].

Semantik

Im Allgemeinen kann man unter der Semantik die Bedeutung der jeweiligen syntaktisch festgelegten Elemente verstehen. Die Semantik einer Sprache beschreibt also deren Interpretation, das heißt zum Beispiel die Bedeutung von einzelnen Wörtern. Bei der Semantik handelt es sich also um den Sinn der Elemente. Damit ein Programm die Semantik hinter der Syntax erkennt, benötigt es festgelegte Spezifikationen, die diese beschreiben [43].

Modell

Wie bereits erwähnt, handelt es sich bei den Modellen in der modellgetriebenen Softwareentwicklung nicht um einfache Grafiken, die eine Struktur wiedergeben, sondern um formale Modelle, aus denen ausführbare Software generiert werden kann. Diese können zum Beispiel UML Modelle oder auch textuelle Modelle sein [94]. Ein Modell spezifiziert

ein System, indem es Funktionen, Verhalten und Struktur beschreibt [43]. Die Modelle müssen in einer DSL geschrieben sein, damit sie wirklich als Modelle in der MDSD gelten [94]. Es wird also immer eine eindeutige Syntax und Semantik benötigt.

Metamodell

Oft wird der Begriff Metamodell als Modell von Modellen erklärt. Dies ist aber nicht korrekt, da Metamodelle Modellierungssprachen beschreiben und daher Modelle von diesen sind [43]. Mit Hilfe von Metamodellen werden die abstrakte Syntax und die Beziehungen zwischen Elementen einer Modellierungssprache beschrieben [94]. Das Metamodell definiert somit, welche Konzepte alle möglichen Instanzen enthalten können und welche sie nicht enthalten dürfen. Es werden also alle erlaubten Konstrukte beschrieben. Modelle, die mit Hilfe der Modellierungssprache dann gebildet werden, sind Instanzen von dem Metamodell [43]. Ein Metamodell, auf dem sehr häufig in verschiedensten Softwareentwicklungen aufgebaut wird, ist das der Modellierungssprache UML (siehe Unterabschnitt 2.3.2).

Metametamodell

Der Begriff Metametamodell ist nun das Metamodell des Metamodells und beschreibt wie Metamodelle aussehen dürften [94]. Auf dieser Ebene wird sozusagen die Metasprache definiert, auf der Metamodelle aufbauen. Die Beschreibung erfolgt abstrakter, da der Metalevel höher ist [43]. Eines der bekanntesten Metametamodelle ist MOF. Dieses wird in Unterabschnitt 2.3.1 näher erklärt.

2.3.1 MOF

Meta-Object Facility (MOF)[4] ist ein Standard der OMG. Die genaue Spezifikation von MOF 2.4.1 kann unter [70] nachgelesen werden. MOF ist das Kernelement von MDA, da sie das Metametamodell für alle zu MDA konformen Programme ist [94]. Sie ist das Metamodell der Modellierungssprache UML und dient zur Erstellung und Erweiterung von den Metamodellen [43]. MOF besteht unter anderem aus folgenden zwei Hauptbestandteilen:

- *Essential Meta-Object Facility (EMOF) Modell*
  Dieses Paket ist eine Untermenge von MOF und enthält die grundlegende Funktionalität [43]. EMOF bietet ein Framework für einfache Metamodelle um Modelle in Implementierungen, wie zum Beispiel XML Metadata Interchange (XMI), abbilden zu können [70]. Ein Ziel von EMOF ist es, einfache Metamodelle mit einfachen Konzepten zu definieren und Erweiterungen für anspruchsvollere Metamodellierung mit CMOF zu unterstützen [70].

- *Complete Meta-Object Facility (CMOF) Modell*
  CMOF umfasst die gesamte Funktionalität von MOF und wird aus EMOF und dem *Core Paket* von UML gebildet [43].

[4] `http://www.omg.org/mof/`, Zuletzt besucht:: 30-07-2013

2.3.2 UML

Die Modellierungssprache *Unified Modeling Language (UML)*[5], welche eine Instanz des Metametamodells MOF ist, ist ein weiterer Standard der OMG. Sie ist eine der verbreitetsten Sprachen, die im Bereich der Modellierung eingesetzt wird, und soll hier kurz erwähnt werden. Der Standard UML wurde von der OMG schon mehrmals angepasst und erweitert um Fehler und ungenaue Spezifikationen zu beseitigen. Zum Beispiel bei der Änderung auf UML 2.0 wurden die UML Sprachkonstrukte klar mit Hilfe von MOF Modellen in der Superstructure definiert [94]. Davor gab es wichtige Unterschiede zwischen UML und MOF. Die aktuelle Version ist momentan UML 2.4.1, wobei die genaue Spezifikation unter [75] nachgelesen werden kann. UML ist eine Modellierungssprache zur Spezifikation von Systemen, wobei der Nutzer frei wählen kann, welche Diagramme er im Entwicklungsprozess der Software verwenden will [43].

Einer der Vorteile von UML ist, dass die Modellierungssprache unabhängig von einer Plattform ist und vom Benutzer an eigene Anforderungen angepasst beziehungsweise erweitert werden kann. Andererseits ist UML recht groß und schlecht unterteilt, was die Modellierung von manchen Funktionalitäten erschwert [94].

Allerdings bietet UML eine Vielzahl an verschiedenen Arten von Diagrammen, wodurch der Benutzer in vielen Bereichen der Konstruktion von Software Systemen unterstützt wird. Einige der möglichen Diagramme seien hier kurz aufgelistet:

- Strukturdiagramme
  - Klassendiagramme
  - Objektdiagramme
  - Paketdiagramme
  - Komponentendiagramme
- Verhaltensdiagramme
  - Use Case Diagramme
  - Aktivitätsdiagramme
  - Zustandsdiagramme
  - Sequenzdiagramme

Ein weiterer großer Vorteil von UML ist, dass mit Hilfe von XMI[6], einem weiteren Standard von der OMG, ein Austausch von Modellen zwischen Modellierungswerkzeugen oder die Codegenerierung möglich ist [43]. XMI beruht dabei auf MOF.

[5] `http://www.uml.org/`, Zuletzt besucht:: 30-07-2013
[6] `http://www.omg.org/spec/XMI/`, Zuletzt besucht:: 30-07-2013

2.3.3 4-layer Metamodell Hierarchie

Von der OMG werden in der MOF insgesamt vier Metalevel von M0 bis M3 definiert. Diese sind in der Abbildung 2.2 im Zusammenhang mit ihren Sprachen abgebildet. Auf der höchsten Ebene M3 befinden sich die Metametamodelle, wie zum Beispiel MOF selbst. Prinzipiell könnte man das Prinzip der Metamodellierung ewig weiterführen, sodass nicht auf der Metametaebene aufgehört wird, jedoch wird auf zusätzliche Level verzichtet, da Elemente der Metametaebene typischerweise mit sich selbst beschrieben werden [43]. So beschreibt MOF sich durch alle definierten Konstrukte selbst. Mit Hilfe der Metametamodelle werden die Metasprachen definiert, die von den zugehörigen Metamodellen der Ebene M2 verwendet werden. Auf dieser Ebene findet sich zum Beispiel die Modellierungssprache UML, die durch ihr zugehöriges Metamodell definiert wird. Die Sprache wird dann eine Ebene tiefer auf dem M1 Level verwendet um Modelle zu beschreiben. Diese Modelle stellen nur einen Ausschnitt aus den möglichen erlaubten Notationen dar [43]. Ein Beispiel hierfür wäre unter anderem ein Klassendiagramm in UML. Auf der letzten Ebene M0 befinden sich konkrete Objekte beziehungsweise Systeme. Hier wäre nun eine konkrete Instanz des Klassendiagramms zu finden.

Im Prinzip kann die 4-layer Metamodell Hierarchie auch für andere Metametamodelle und deren Ebenen außer MOF angewandt werden. Ein weiteres Beispiel ist das im nächsten Abschnitt vorgestellte Metametamodell Ecore.

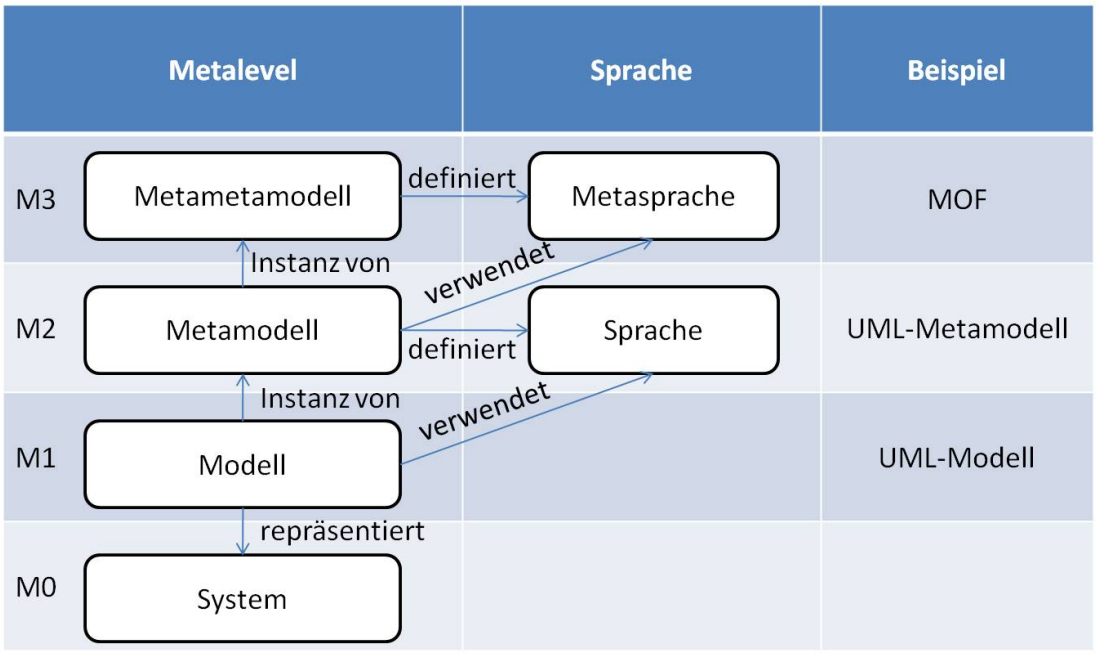

Abbildung 2.2: 4-layer Metamodell Hierarchie [43, 54]

2.3.4 Ecore

Im EMF[7] bildet *Ecore* das zugrunde liegende Metametamodell. Ecore ist eine Java-basierte Implementierung von den wichtigsten Komponenten von MOF und ist sehr ähnlich zu EMOF [94]. Eine vereinfachte Darstellung des Metametamodells ist in Abbildung 2.3 zu sehen, wobei Attribute und Referenzen zwischen den Elementen nicht angegeben sind.

In den vorher beschriebenen Metaebenen von der OMG kann man Ecore auf dem Level M3 einordnen, da Ecore sich selbst beschreibt und definiert. Metamodelle, die mit Hilfe von Ecore modelliert werden, befinden sich somit auf Ebene M2 und deren Instanzen, die Modelle werden in der Ebene M1 eingeordnet. Diese Einteilung unterscheidet sich ein wenig zu dem oben gegebenen Beispiel der Zuordnungen mit UML, jedoch wird Ecore in der Metamodellierung in EMF konkret so eingesetzt.

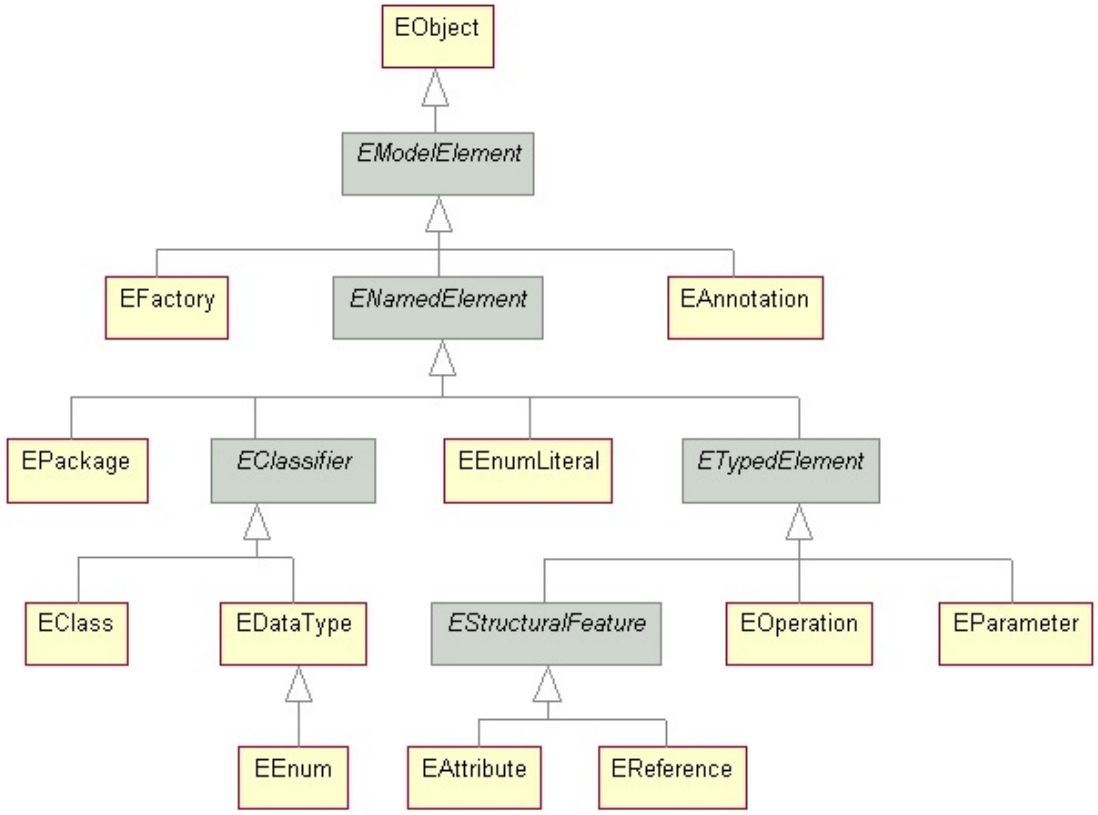

Abbildung 2.3: Ecore Metametamodell
(Quelle: `http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html`, Zuletzt besucht: 27-07-2013)

[7] `http://eclipse.org/modeling/emf/`, Zuletzt besucht: 18-03-2013

2.4 Transformationen

2.4.1 Grundlagen

Im Allgemeinen ist das Ziel einer Transformation eine gegebene Eingabe (Input) in eine gewünschte Ausgabe (Output) zu transformieren. In der modellgetriebenen Softwareentwicklung handelt es sich bei der Eingabe meistens um ein Modell und in diesem Zusammenhang um Modelltransformationen. Es gibt verschiedenste Gründe, warum Transformationen durchgeführt werden. Gruhn et al. [43] beschreiben verschiedene Anwendungsfälle für Transformationen, von denen einige nun näher vorgestellt werden.

- Abstraktion
  Ein Grund für eine Transformation kann sein, dass aus einem gegebenen Quellcode zum Beispiel Schnittstellen gewonnen werden sollen, die mittels des erzeugten Modells direkt angesprochen werden können [43]. Es wird von einer tieferen Ebene wieder auf eine abstraktere Ebene fokussiert um die neu gewonnenen Informationen breiter einsetzen zu können.

- Verfeinerung
  Das genaue Gegenteil von Abstraktion. Hier werden Modelle mit Hilfe der Transformation detailreicher und somit verfeinert. Ein mögliches Beispiel wäre die Transformation von einem PIM zu einem PSM in der MDA.

- Migration
  In Softwareprozessen ist es nicht selten, dass Systeme aktualisiert und neuere Versionen von Technologien eingesetzt werden. Somit müssen auch die Softwaresysteme angepasst werden, damit sie für den Benutzer funktionsfähig bleiben. Hierfür ist es notwendig, dass Systeme migriert und die Bestandteile richtig transformiert werden um weiterhin genutzt werden zu können. Ein Beispiel für eine komplexe Migration ist unter anderem ein Wechsel von COBOL zu Java [43].

- Refaktorisierung und Optimierung
  In einer Software kann es auch Änderungen zum Beispiel in Bezug auf die innere Struktur geben, die unter anderem die Verständlichkeit erhöhen. Die Funktionalität ist aber im Gegensatz zur Migration nicht betroffen. Auch im Bereich der Refaktorisierung können Transformationen im Prozess hilfreich sein. Sehr ähnlich ist es im Bereich der Optimierungen. Auch hier bleibt die Funktionalität von der Quelle und dem Ziel dieselbe, jedoch werden zum Beispiel im Bereich der Rechenzeit Optimierungen durchgeführt [43].

Transformationen können auf verschiedene Arten definiert und implementiert werden. Drei der möglichen Ansätze werden im Folgenden vorgestellt.

- Direkte Modellmanipulation [90]
  Im Fall der direkten Modellmanipulation wird meistens eine universelle Programmiersprache, wie zum Beispiel Java, eingesetzt. Die gewünschten Transformationen

werden in einer Methode direkt implementiert und später ausgeführt [43]. Nachteile dieses Ansatzes sind, dass Änderungen direkt im Quellcode vorgenommen werden müssen, oft auf einen passenden Abstraktionslevel der Transformation verzichtet wird und manche Transformationen aufgrund der Einschränkungen der Application Programming Interface (API)s nicht möglich sind [90].

- Zwischenrepräsentation [90]
  Eine weitere Möglichkeit ist es, Modelle in ein gängiges Format zu überführen und mit Hilfe von existierenden Tools die Transformation durchzuführen. Ein mögliches Standardformat wäre die Extensible Markup Language (XML). Jedoch bleibt auch bei diesem Ansatz das Problem bestehen, dass Änderungen im Modell erst wieder in das Standardformat übergeführt werden müssen, bevor die Transformation beeinflusst wird [90].

- Transformationssprachen
  Transformationssprachen haben explizite Konstrukte zum Durchführen von Transformationen und geben die Möglichkeit Änderungen durchzuführen, ohne das gesamte Programm neu zu übersetzen [43]. Eine genauere Beschreibung und mögliche Sprachen finden sich in Unterabschnitt 2.4.2.

2.4.2 Transformationssprachen

Transformationssprachen dienen explizit zur Beschreibung der Transformationen selbst und können zu diesem Zweck auch angepasst werden. Der Vorteil ist, dass diese Art der Implementierung für Transformationen die Anforderung einer DSL erfüllt. Eine Transformationssprache benötigt nur Wissen zur Beschreibung der Transformation und nicht Kenntnisse über andere Abläufe im Programm [43]. Es gibt viele verschiedene entwickelte Sprachen, die für Transformationen verwendet werden, wobei manche auch visuelle Konstrukte anbieten. Einige mögliche Transformationssprachen im Bereich von Model to Model (M2M) Transformationen werden nun in ihren Grundzügen näher vorgestellt.

2.4.2.1 Query View Transformation - QVT

Query View Transformation (QVT)[8] ist ein weiterer Standard der OMG, welcher 2005 zum ersten Mal veröffentlicht wurde. Der Standard QVT umfasst nicht eine, sondern drei Transformationssprachen [74]:

- QVT Core
  Die Core-Sprache ist deklarativ und ist eine minimale Erweiterung von EMOF und OCL. In dieser Sprache müssen alle Transformationsregeln explizit als MOF Metamodelle spezifiziert werden. Da es keinen Mechanismus zur Identifikation von Zielmodellelementen gibt, muss dies mit expliziten Regeln erfolgen [94]. Diese Sprache ist recht einfach, doch durch diesen Umstand eher selten alleine in der Praxis genutzt.

[8] `http://www.omg.org/spec/QVT/`, Zuletzt besucht:: 30-07-2013

- QVT Relations
  Die Relations-Sprache ist wie die Core-Sprache deklarativ. Sie benötigt Relationen zwischen Elementen des Quell- und Zielmetamodells in Form von Objektmustern [94]. Es werden die *Trace* Klassen und deren Instanzen automatisch erzeugt, damit erfasst wird, was während einer Transformation stattgefunden hat [74]. Sie bietet im Gegensatz zur Core-Sprache Mechanismen zur Identifikation von Zielmodellelementen und ist daher benutzerfreundlicher. Die Semantik der Sprache ist allerdings über die Core-Sprache definiert.

- QVT Operational Mapping
  Im Gegensatz zu den Sprachen Core und Relations ist die Operational Mapping Sprache imperativ. Es wird OCL verwendet, allerdings mit imperativen Teilen erweitert, um unter anderem Berechnungen durchzuführen [94]. Es ist möglich eine Transformation vollständig mit Hilfe der Sprache auszudrücken.

Oft werden die einzelnen Sprachen von QVT nicht alleine, sondern in einem hybriden Ansatz zusammen verwendet. So werden Teile der Transformation mit Hilfe der Relationsbeziehungsweise Core-Sprache beschrieben und zusätzlich individuelle Regeln als *Black-Box-Mappings* mit der Operational Mapping Sprache definiert [94].

2.4.2.2 Atlas Transformation Language - ATL

Die *Atlas Transformation Language (ATL)*[9] ist eine Programmiersprache um gegebene Quellmodelle in Zielmodelle zu transformieren. Sie wurde von *ATLAS INRIA & LINA* entwickelt als Antwort zu der Transformationssprache QVT der OMG. Das Quellmodell bei einer ATL Transformation wird nur gelesen und das Zielmodell nur geschrieben, wobei der Code hierfür deklarativ in Regeln definiert wird. In bestimmten Regeln kann aber auch imperativer Code ausgeführt werden.

Da das im Zuge dieser Arbeit entwickelte Tool selbst auf ATL Transformationen aufbaut, findet sich die genaue Struktur und Syntax in Abschnitt 6.4.

2.4.2.3 Triskell Metamodeling Kernel - Kermeta

Eine weitere Möglichkeit, mit der Transformationen definiert werden können, ist die Programmiersprache *Kermeta*[10]. Kermeta wurde als open-source Projekt von *INRIA* entwickelt. Die Sprache ist eine Modellierungssprache, welche nicht nur für Modelltransformationen, sondern auch zur Definition von Metamodellen, zum Hinzufügen von Constraints und zur Spezifikation von Verhalten der Modelle genutzt werden kann [23]. Die Sprache ist konform zu EMOF und Ecore und wird als Eclipse Plugin zur Verfügung gestellt. Kermeta basiert auf den zwei Sprachen *Xion* und *Model Transformation Language (MTL)*, wobei Xion zur plattformunabhängigen Implementierung von Operationen und Methoden von UML Klassendiagrammen und MTL als objektorientierte Sprache APIs zur

[9] `http://wiki.eclipse.org/MMT/Atlas_Transformation_Language_(ATL)`, Zuletzt besucht:: 06-02-2013

[10] `http://www.kermeta.org/`, Zuletzt besucht:: 06-08-2013

Manipulation von Modellen anbietet [67]. Kermeta erweitert diese Konzepte unter anderem durch eine Modellnavigation mit Hilfe von OCL-Constraints. Im Zusammenhang mit Modelltransformationen ist ein weiterer wichtiger Punkt, dass Kermeta eine imperative Sprache ist. Im Gegensatz zu Regeln verwendet die Sprache Operationen, die mit Methoden, wie zum Beispiel in Java, verglichen werden können. Ein Vorteil von Kermeta ist, dass durch die verschiedenen Features, wie zum Beispiel Exception Handling oder Generizität, die andere Transformationssprachen nicht besitzen, viele Transformationen besser beschrieben werden können [23]. Allerdings hat die Sprache in diesem Bezug auch Nachteile. Es müssen im Code die Modelle immer explizit geladen und gespeichert werden, und es gibt keine automatische Tracing Unterstützung [47].

Zusammengefasst wird die Sprache Kermeta unter anderem mit folgenden Begriffen beschrieben [23]:

- Modellorientiert
  Der Hauptzweck der Sprache ist es eine Möglichkeit zur Manipulation von Modellelementen zu bieten.
- Imperativ
- Aspektorientiert
  Es können Elemente von verschiedenen Quellen eingebunden und existierende Metamodelle erweitert werden.
- Objektorientiert
  Wie in Java werden objektorientierte Features (Klassen, Vererbung,...) zur Verfügung gestellt.

2.4.2.4 Tripel-Graph-Grammatik - TGG

Eine andere Transformationssprache ist *Triple Graph Grammar (TGG)*. Um die Funktionsweise von der Tripel-Graph-Grammatik zu verstehen, ist das Konzept von Graphtransformationen wichtig. Bei Graphtransformationen werden Modelle aus graphentheoretischer Sicht interpretiert. Die Elemente eines Modells und die Beziehungen untereinander werden als Knoten und Kanten in einem Graphen gesehen [44]. Eine Graphtransformationsregel hat zumindest eine linke Seite (*Left-Hand Side (LHS)*) und eine rechte Seite (*Right-Hand Side (RHS)*, wobei diese jeweils dem Quell- und dem Zielmodell entsprechen [44]. Wenn eine Regel ausgeführt wird, werden die Elemente der LHS in die der RHS transformiert. Bei komplexen Transformationen ist es ab und zu notwendig noch zusätzliche Bedingungen zu definieren um die Anwendung von Regeln einzuschränken. Diese werden mit Hilfe von *Positive Application Condition (PAC)* und *Negative Application Condition (NAC)* festgelegt, wobei eine PAC einen benötigten und NAC einen verbotenen Kontext beschreibt [44].

Die Graphtransformationssprache TGG baut auf diesem Konzept auf, wobei die Regeln um einen dritten Graphen, den *Korrespondenzgraphen*, erweitert werden [87]. In diesem werden die Elemente von Quell- und Zielmodell logisch miteinander verlinkt

und der Graph kann daher als *Tracing*-Information der Transformation gesehen werden. Eine TGG Regel enthält drei Graphgrammatiken, die jeweils den Quellgraphen, den Korrespondenzgraphen oder den Zielgraphen bearbeiten, wobei alle drei gleichzeitig angewandt werden [87].

Vorteile von TGG sind unter anderem, dass sowohl unidirektionale als auch bidirektionale Transformationen spezifiziert werden können und dass die Korrespondenzen zwischen Quell- und Zielmodell explizit modelliert werden, und nicht auf 1-zu-1 Beziehungen beschränkt sind [87].

2.4.3 Kategorisierung von Transformationen

Transformationen können nach verschiedenen Kategorien unterteilt werden. Einige mögliche Gliederungen sollen hier im Folgenden näher vorgestellt werden. Die Kategorien sind nicht disjunkt, sondern es ändern sich die Kriterien, die für die Einteilung herangezogen werden. Daher finden sich auch in verschiedenen Bereichen dieselben Transformationssprachen. Für die Benennung der einzelnen Kategorien wurde unter anderem die Klassifizierung von Weisemöller [99] verwendet. Eine weitere mögliche Einteilung findet sich bei Czarnecki und Helsen [21].

2.4.3.1 Struktur und Abstraktionsebene

Eine mögliche Unterscheidung von Transformationen ist anhand der Struktur und Abstraktionsebenen von den Quell- und Zielmodellen gegeben und wird in Abbildung 2.4 veranschaulicht.

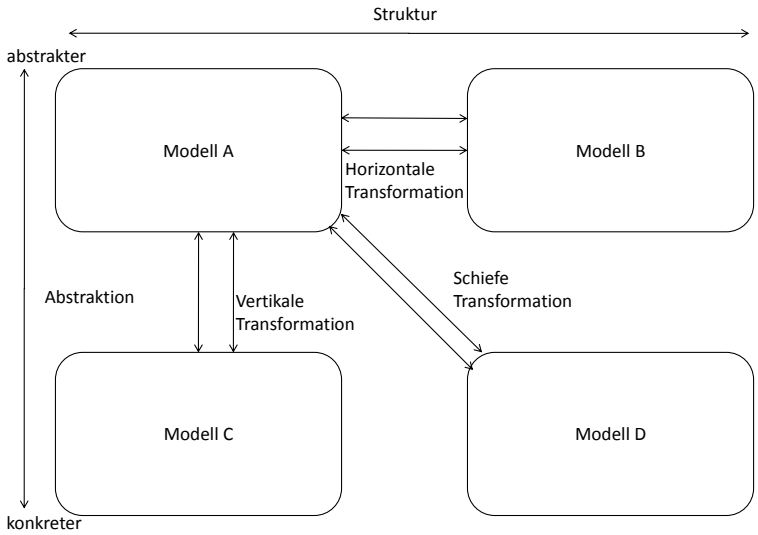

Abbildung 2.4: Klassifizierung von Transformationen anhand der Abstraktionsebene und Struktur von Modellen [43]

Vertikale Transformationen

Vertikale Transformationen ändern die Abstraktionsebene, jedoch wird die Struktur nicht verändert [43]. Ein Beispiel hierfür wäre, wenn ein Modell verfeinert wird um mehr Details für die Implementierung abbilden zu können.

Horizontale Transformationen

Die horizontale Transformation beeinflusst nicht die Abstraktionsebenen, sondern verändert die Struktur des Zielmodells im Bezug auf das Quellmodell. Migration von Modellen ist hierfür ein konkretes Beispiel [62].

Schiefe Transformationen

Die schiefe Transformation ist nichts anderes als eine Kombination aus der horizontalen und vertikalen Transformation [43]. Hier wird bei der Umwandlung von Quell- in Zielmodell nicht nur die Struktur, sondern auch die Abstraktionsebene geändert.

2.4.3.2 Quell- und Zielsprache

Eine weitere Einteilung für Transformationen kann anhand ihrer Quell- und Zielsprache vorgenommen werden.

Endogene Transformationen

Wenn das Quell- und Zielmodell mit derselben Sprache modelliert sind, so spricht man von *endogenen* Transformationen, wobei auch der Begriff *rephrasing* in diesem Zusammenhang benutzt wird [62]. Unter anderem wird bei einer Optimierung mit Hilfe der Transformation ein Teil des Programms verbessert, wobei aber die Semantik sich nicht verändert. So bleibt also auch das Zielmodell in derselben Sprache wie das Quellmodell.

Exogene Transformation

Im Gegensatz zu endogenen Transformationen sind bei *exogenen* die Sprachen des Quell- und Zielmodells verschieden. Solche Transformationen werden auch *translations* genannt. Beispiele hierfür sind unter anderen Migration und Codegenerierung [62]. Diese Beispiele können auch im Zusammenhang zur vorigen Unterteilung anhand der Struktur und Abstraktionsebene gesehen werden. Zum Beispiel wird bei der Migration eines Programms in eine neue Sprache die Transformation auf derselben Abstraktionsebene (horizontal) durchgeführt, während bei der Generierung von Code aus einem Modell, von einem abstrakteren auf ein spezifisches Level (vertikal) geändert wird.

2.4.3.3 Modell-zu-Modell Transformationen

Bei *Model to Model (M2M)* Transformationen handelt es sich konkret um Umwandlungen von Modellen. Sowohl die Quelle, als auch das Ziel ist ein Modell. Diese Modelle müssen immer konform zu ihren zugehörigen Metamodellen sein. Die Transformation legt Regeln fest, wie die Elemente des Quellmetamodells auf Elemente des Zielmetamodells abgebildet werden dürfen. Die eigentliche Transformationsausführung führt dann die

Regeln konkret für die im Modell vorhandenen Elemente aus. Transformationssprachen, mit denen M2M Transformationen implementiert werden können, sind unter anderem ATL und QVT. Eine mögliche Anwendung ist ein abstrakteres Modell in ein spezifischeres umzuwandeln, wie in der MDA zum Beispiel die Transformation von einem PIM zu einem PSM.

In-place Transformation

Modellmodifikationen oder *in-place* Transformationen sind eine Art von M2M Transformationen. Hier wird ein Modell modifiziert, das bedeutet, dass die Änderungen direkt im Modell vorgenommen werden. In dieser Transformation ist das Quell- und Zielmodell also dasselbe. Der Vorteil ist, dass Elemente, die von der Transformation nicht betroffen sind, nicht kopiert werden müssen und bei den Transformationsregeln nicht beachtet werden [13]. Durch die Veränderung von Elementen, beziehungsweise auch durch das Hinzufügen von neuen verändert das Modell sozusagen seinen Zustand [94]. In Abbildung 2.5 ist diese Art von Transformation graphisch veranschaulicht. Für In-place Transformationen eignen sich zum Beispiel Graphtransformationen.

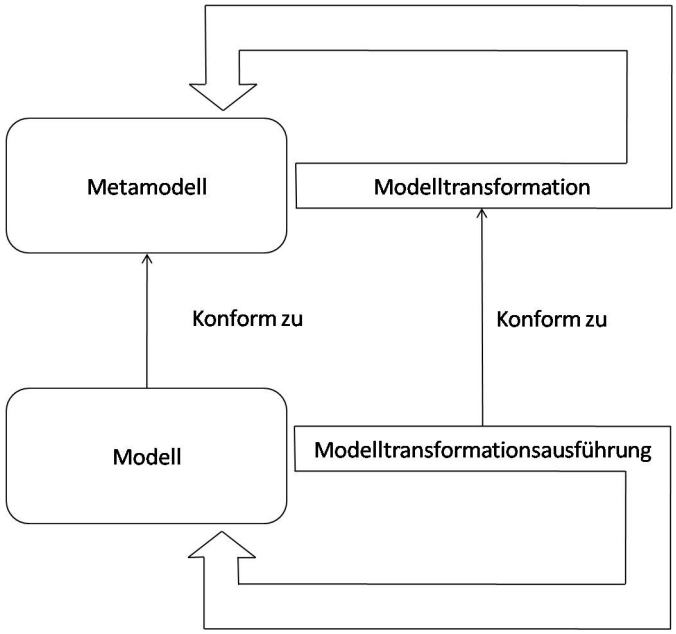

Abbildung 2.5: Modellmodifikation / In-place Transformation [13]

Out-place Transformation

Out-place Transformationen führen im Gegensatz zu in-place Transformationen keine Modellmodifikationen durch, sondern erstellen neue Zielmodelle. In Abbildung 2.6 ist die Funktionsweise dieser Art von Transformation abgebildet. Mit Hilfe der Transformationsausführung, welche konform zu den definierten Transformationsregeln ist, wird ein

neues Zielmodell erstellt, welches zum Zielmetamodell konform sein muss. Das Quellmodell wird nicht verändert, sondern wird für die Transformation nur gelesen. Für die Transformation ist es prinzipiell egal, ob Quell- und Zielmodell dasselbe Metamodell oder verschiedene haben [62]. Ein Beispiel für eine out-place Transformationssprache ist ATL.

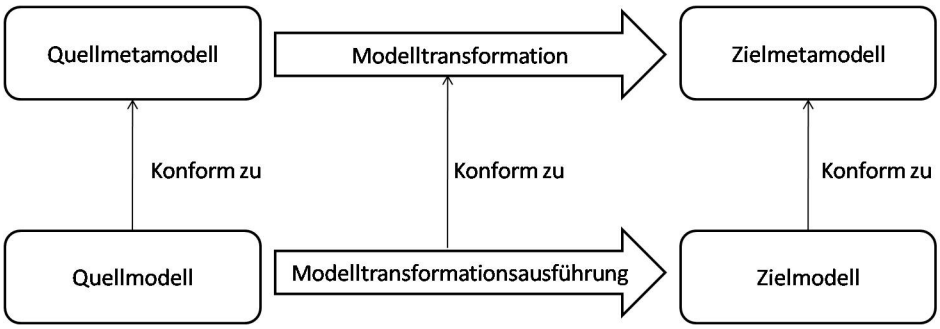

Abbildung 2.6: Out-place Transformation [13]

In diesem Zusammenhang sei noch erwähnt, dass die Abbildung 2.6 für unidirektionale Sprachen korrekt ist. Das bedeutet, dass die Modelltransformation nur in eine Richtung, also vom Quell- zum Zielmodell, ausgeführt werden kann, jedoch wird für die Umkehrung eine eigene Transformation benötigt. Dies ist aber bei bidirektionalen Transformationen möglich. Daher müsste für bidirektionalen Transformationen der Pfeil nicht nur von Quell- zu Zielmodell führen, sondern auch zurück. ATL gehört zu den unidirektionalen Sprachen. Im Gegensatz dazu sind Graphtransformationen mit zum Beispiel TGG bidirektional [20].

In Bezug auf exogene und endogene Transformationen sind exogene meist out-place Transformationen und endogene vor allem in-place Transformationen [62]. Es ist allerdings auch möglich endogene out-place und exogene in-place Transformationen zu implementieren.

In diesem Zusammenhang sei noch das *Modell-Weaving*, beziehungsweise *Linking* erwähnt. Hierfür werden mehrere Quellmodelle, welche entweder dasselbe Metamodell oder verschiedene haben können, nach dem Einlesen miteinander verlinkt. Bei Stahl et al. [94] wird das Modell-Weaving als Spezialfall einer in-place Transformation beschrieben. Es kann jedoch auch als out-place Transformation implementiert werden, wie zum Beispiel mit dem Tool Atlas Model Weaver (AMW)[11]. Hier werden unter anderem *matching Transformationen* definiert, welche automatisch *Weaving-Modelle* erzeugen.

2.4.3.4 Modell-zu-Text Transformationen

Im Gegensatz zu M2M handelt es sich bei *Model to Text (M2T)* Transformationen um Transformationen von Modellen in Text. Gerne wird auch der Begriff *Model to*

[11] `http://www.eclipse.org/gmt/amw/`, Zuletzt besucht: 08-08-2013

Code (M2C) verwendet, wobei nicht nur Quellcode aus einem Modell erzeugt werden muss. Ein Problem, welches im Zusammenhang mit M2T Transformationen steht, ist die Synchronisation von Modellen und Code [43]. Es ist jedoch zu beachten, dass Änderungen in Modell und Code nicht immer parallel umgesetzt werden. Ansonsten kann der aktuelle Entwicklungsstand zwischen beiden variieren und Schwierigkeiten bei der Weiterentwicklung hervorrufen. In [43] werden unter anderem folgende Lösungsvorschläge genannt:

- Forward Engineering Only
  In diesem Fall wird der ganze Code aus dem Modell generiert und somit immer bei Neuerungen überschrieben.

- Partial Round-Trip Engineering
  Bei diesem Ansatz werden nur Teile manuell im Code hinzugefügt, wobei der Entwickler sich daran halten muss, dass der generierte Code nicht verändert werden darf. Es werden immer nur die Änderungen von dem Modell im Code übernommen, wobei die manuell hinzugefügten Teile erhalten bleiben [43].

- Full Round-Trip Engineering
  Hier werden Änderungen sowohl im Code, als auch im Modell immer übernommen. Es gibt keinen Unterschied zwischen manuellem und generiertem Code, sondern Modell und Text sind zwei verschiedene Sichten auf dasselbe System [43].

Bei M2T Transformationen gibt es verschiedene Unterteilungen, wobei zwei kurz vorgestellt werden.

Line-printer

Eine Möglichkeit Text aus einem Modell zu erzeugen ist die Elemente im Objektgraphen durchzugehen und eine textuelle Repräsentation schreiben zu lassen [43]. Da diese Zeilenweise geschrieben wird, wird dieser Ansatz *Line-printer* genannt [43]. Der Nachteil ist, dass die Wartbarkeit meist schwierig ist, da die Struktur des resultierenden Textes nur schwer erkennbar ist.

Templates

Ein anderer Ansatz zum Erzeugen von Text ist die Nutzung von *Templates*. *"A template usually consists of the target text containing splices of metacode to access information from the source and to perform code selection and iterative expansion..." [21, S. 9]* Um den Text zu erzeugen werden sowohl statische Textbausteine als auch Elemente, die von der jeweiligen Template-Sprache abhängen, für dynamische Inhalte verwendet [43]. Ein Vorteil ist, dass Templates unabhängig von der Zielsprache sind. Eine mögliche Sprache zur Erzeugung von Code ist XPand [25], welche auch in dieser Arbeit bei der Implementierung genutzt wurde (siehe Unterabschnitt 6.2.2.1).

2.4.3.5 Ablaufsteuerung

Eine letzte mögliche Einteilung von Transformationen, die noch näher betrachtet werden soll, ist die Unterteilung je nach Ablaufsteuerung.

Imperative Transformationen

Imperative Transformationen fokussieren darauf, **wie** die Transformation selbst abläuft [62]. Es werden die notwendigen Schritte definiert, um von einem Quellmodell zu einem Zielmodell zu gelangen. Ein Vorteil ist, dass mit Hilfe eines imperativen Ansatzes der Kontrollfluss klar definiert ist. Das bedeutet unter anderem, dass die Reihenfolge der Transformationsschritte genau festgelegt und kontrolliert werden kann. Es kann aber dazu führen, dass implementierte Transformationsregeln adaptiert werden müssen, wenn die Quell- und Zielmodelle außerhalb der Transformation stark verändert wurden [62]. Mögliche Sprachen, die einen imperativen Ansatz verfolgen, sind Kermeta und QVT Operational Mapping.

Deklarative Transformationen

Im Gegensatz zu imperativen Transformationen geht es bei den *deklarativen* Ansätzen darum, **was** transformiert werden soll [62]. Es werden Regeln, beziehungsweise Relationen zwischen Elementen des Quell- und Zielmodells definiert. Daher werden diese Ansätze auch *relational* genannt [21]. Viele deklarative Sprachen bieten die Möglichkeit zu einer bidirektionalen Transformation, was bei imperativen Transformationen kaum möglich ist. Außerdem wird die Navigation durch das Quellmodell, die Erzeugung des Zielmodells und die Reihenfolge der Regelausführung implizit in der Transformationssprache ausgeführt, wodurch die Transformationsregeln leichter zu implementieren sind [62]. Ein Nachteil ist jedoch, dass dadurch kein expliziter Kontrollfluss möglich ist und komplexe Transformationen schwieriger zu implementieren sind. Eine Sprache, die diesen Ansatz verfolgt, ist zum Beispiel QVT Relations.

Hybride Transformationen

Hybride Transformationen, beziehungsweise Transformationssprachen verwenden sowohl die Konzepte von deklarativen als auch von imperativen Transformationen. Sie sind also eine Mischung aus zwei Ansätzen. Eine hybride Sprache ist ATL, wobei *matched rules* und *lazy rules* die deklarativen Regeln sind. In *called rules* wird allerdings imperativer Code ausgeführt (siehe Abschnitt 6.4). Durch die Kombination von deklarativ und imperativ kann je nach Situation die bessere Methode ausgewählt werden und somit eine flexiblere Lösung resultieren.

Regelbasierte und Nicht Regelbasierte Transformationen

Im Zusammenhang mit der Einteilung in imperative, deklarative und hybride Transformationen ist es auch möglich nach regelbasierten und nicht regelbasierten Ansätzen zu unterteilen.

Hierbei heißen Transformationssprachen, die auf Anwendung und Definition von Regeln aufbauen, wie das Wort schon andeutet, *regelbasiert*. Bei diesen Sprachen wird der gesamte Ablauf in Regeln beschrieben, die durchgeführt werden, wenn jeweils ihre Vorbedingungen erfüllt sind. Beispiele für diese Sprachen sind unter anderem die Graphtransformationssprache TGG und ATL.

Auf der anderen Seite stehen die *nicht regelbasierten* Sprachen, wo der Ablauf der Transformation in imperativen Anweisungen steht. Unter anderem kann Java als nicht regelbasierte Transformationssprache fungieren, wenn im Programmcode mit Hilfe von Methoden die Transformationen ausgeführt werden. Eine weitere nicht regelbasierte Sprache ist Kermeta.

KAPITEL 3

Testen von Software[1]

Eine der wichtigsten Aufgaben bei der Entwicklung von Software ist das Testen ebendieser. Für viele Programmierparadigmen[2] gibt es gut dokumentierte und anwendbare Ansätze für diese Aufgabe. Zunächst werden in diesem Kapitel im Abschnitt 3.1 allgemeine Begriffe und Prinzipien des Softwaretestens vorgestellt.

Viele Konzepte werden zunehmend für den Bereich des Model Engineering adaptiert. Zwei der wichtigsten davon sind das Black- und White-Box Testen. Diese haben mittlerweile auch beim Testen im Rahmen von Model Engineering einige Bedeutung erlangt. Da sich diese Arbeit mit der Entwicklung eines White-Box Ansatzes zum Testen von Modelltransformationen beschäftigt, werden diese beiden Konzepte im Abschnitt 3.2 näher vorgestellt.

Um das Testen möglichst effizient durchführen zu können, gibt es für viele Aufgaben bereits automatisierte Lösungen. Da die Automatisierung auch ein wichtiges Ziel des in dieser Arbeit entwickelten Ansatzes ist, wird im Abschnitt 3.3 näher beleuchtet, welche Aufgaben auf welche Art und Weise automatisiert werden können. Außerdem muss festgelegt werden, mit welchen Daten die Tests durchgeführt werden sollen. Diese Entscheidung kann eine recht komplexe Aufgabe darstellen, die in Abschnitt 3.4 näher untersucht wird. Für alle diese Probleme werden zunächst die Lösungsansätze vorgestellt, die im Software Engineering entwickelt wurden.

Das Testen im Zusammenhang mit Model Engineering und vor allem im für diese Arbeit relevanten Bereich der Modelltransformationen stellt neue Aufgaben und Herausforderungen. Um diese zu lösen gibt es bereits verschiedenste Ansätze, die im letzten Teil dieses Kapitels (Abschnitt 3.5, Abschnitt 3.6) vorgestellt werden. Es wird auch darauf eingegangen, wie die aus dem Software Engineering bekannten Konzepte für das Testen an sich und dessen (Teil-)Automatisierung, angewandt werden können.

[1] Verfasser: Thomas Franz, BSc

[2] https://de.wikipedia.org/wiki/Programmierparadigma

3.1 Testen in Software Engineering

Durch die steigende Komplexität von Softwareprojekten steigt auch die Anzahl an möglichen Fehlerquellen. Um diese aufzudecken ist eine enge Integration von Tests in den Entwicklungsprozess erforderlich. Zur Anwendung auf verschiedenen Ebenen (Levels/ Teststufen) stehen verschiedenste Methoden zur Verfügung. Im Laufe der Zeit wurden unterschiedliche Werkzeuge entwickelt um den Vorgang des Testens in zunehmendem Maße zu automatisieren.

Das Testen von Software ist ein umfangreiches Themengebiet. Es gibt unzählige Klassifikationen, Testarten und Testmethoden. Im Folgenden werden die am weitesten verbreiteten Konzepte und Einteilungen vorgestellt wie zum Beispiel Teststufen oder Phasen des Testprozesses. Allerdings ist zu beachten, dass viele Themenbereiche eng miteinander verknüpft sind, sodass es teilweise schwierig ist diese genau voneinander abzugrenzen. Es werden daher zu Beginn ausgewählte Begriffe vorgestellt, die in den weiteren Abschnitten des Öfteren auftauchen oder generell von Bedeutung sind.

3.1.1 Grundlagen

3.1.1.1 Softwaretest

Ein Softwaretest dient dazu festzustellen, ob die entwickelte Software den Anforderungen, die an sie gestellt werden, in der spezifizierten Art und Weise entspricht. Es soll also die Qualität des Produktes gemessen werden. Um diese sicher zu stellen ist es im Rahmen immer komplexerer Software-Projekte notwendig, den Vorgang des Testens eng in denjenigen der Entwicklung der Software zu integrieren.

Vorrangiges Ziel bei der Anwendung von Softwaretests ist die Vermeidung von Fehlern bei der Ausführung des Programms. Hier ist auch die Kostenfrage von Bedeutung. Denn je früher ein Fehler bemerkt und die Software dementsprechend verbessert wird, desto weniger Kosten fallen dafür an. Im Gegenzug steigt allerdings auch der Aufwand, denn schon früh muss die Qualitätsmessung mittels Softwaretests hohen Anforderungen genügen [24, 59, 60, 79, 95].

Generell wird geprüft, ob die Software die folgenden Kriterien erfüllt [24]:

- Eine korrekte Eingabe liefert eine korrekte Ausgabe.
- Das Programm erkennt ungültige Eingaben und liefert entsprechende Fehlermeldungen zurück
- Programmabstürze werden weder durch gültige, noch durch ungültige Eingaben ausgelöst
- Das Programm wird nicht früher als erwartet beendet
- Das Programm arbeitet, wie es in der Spezifikation vorgesehen ist

3.1.1.2 Testfall

Ein Testfall dient der Beschreibung eines Softwaretests. Er enthält alle wichtigen Attribute, die benötigt werden um eine möglichst vollständige Dokumentation des Testvorgangs zu ermöglichen. Dessen Ziel ist es letztendlich eine Aussage darüber treffen zu können, ob der Test erfolgreich war oder nicht. Dies wird ebenfalls im Testfall dokumentiert. Jeder Testfall sollte zumindest die folgenden Angaben enthalten[3] [48, 93]:

- Eine eindeutige Identifikation des Testfalls
- Eine Beschreibung des Testfalls (was wird getestet)
- Eine Beschreibung des Objekts und dessen Eigenschaften, die getestet werden sollen
- Beschreibung der Eingabedaten (sowie deren Beziehungen falls erforderlich)
- Beschreibung der erwarteten Ausgabedaten (es werden entweder die exakten Werte oder ein Toleranzbereich angegeben)
- Angabe über den Erfolg/Misserfolg des Tests (evtl. auch Angabe der genauen Ergebnisse)
- Abhängig von der Komplexität des Testfalls kann die Angabe weiterer Attribute erforderlich sein. Dazu zählen unter anderem
- Angabe der Umgebungseinstellungen (Hard- und/oder Softwarekonfiguration)
- Angabe von Vorbedingungen wie z.B. eine vollständige Auflistung aller Identifikationen von Testfällen, die vor dem aktuellen ausgeführt werden müssen.
- Angabe, ob es sich um einen manuellen oder automatisierten Test handelt
- Identifikation des Testers

Testfälle können in Positivtests und Negativtests eingeteilt werden. Erstere prüfen, ob das Programm mit gültigen Eingaben auch korrekt arbeitet (also gültige Ausgaben zurückliefert). Negativtests prüfen hingegen, ob das Programm auch bei falschen/fehlerhaften Eingaben das gewünschte Verhalten aufweist.

3.1.1.3 Testabdeckung

Die Testabdeckung ist ein wichtiges Qualitätskriterium für Software. Genauer gesagt handelt es sich hierbei um eine Metrik, also eine Funktion, die eine oder mehrere Eigenschaften von Software in Zahlen abbildet. Dies soll auch dazu beitragen, dass Bewertungen von und Vergleiche mit anderer Software vorgenommen werden können [59,60,95,98].

[3] `https://de.wikipedia.org/wiki/Testfall`, Zuletzt besucht: 07-02-2013

Die Testabdeckung selbst beschreibt, in welchem Ausmaß ein Programm getestet wurde. Dabei wird die Anzahl an tatsächlich getroffenen Aussagen mit der Anzahl aller möglichen in Beziehung gesetzt [59, 60, 95, 98].

Die Testabdeckung ist eng mit der Anzahl an definierten Testfällen verknüpft. Oftmals kann eine hohe Testabdeckung nur durch eine hohe Anzahl an Testfällen erzielt werden. Daher besteht ein Trade-off zwischen der Testabdeckung und dem Testaufwand. Aus diesem Grund ist auch eine vollständige Testabdeckung oft nicht erzielbar bzw. soll diese nicht immer unter allen Umständen erreicht werden. Dabei ist aber zu beachten, dass der zu erreichende Grad an Testabdeckung auch von der Art des Projekts abhängig ist [59, 60, 95, 98].

Die wichtigsten Messgrößen (siehe Abbildung 3.1) sind [59, 60, 95, 98]:

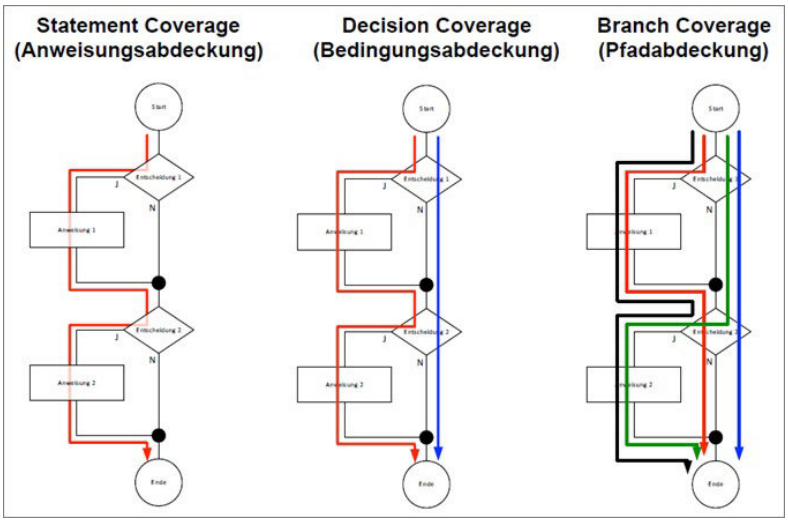

Abbildung 3.1: Darstellung der wichtigsten Überdeckungsparameter

- Die *Anweisungsabdeckung* gibt an, wie viel Prozent der möglichen Entscheidungen bei der Testdurchführung ausgeführt wurden.

- Die *Bedingungsabdeckung* gibt an, wie viel Prozent der möglichen Ergebnisse von Entscheidungen ausgeführt wurden.

- Die *Pfadabdeckung* gibt an, wie viel Prozent der möglichen Pfade ausgeführt wurden. Dieser Parameter impliziert die beiden vorangegangenen.

- Daneben gibt es noch zahlreiche weitere Möglichkeiten das Ausmaß, in dem Tests verschiedenste Code-Konzepte (z.B. Schleifen, Bedingungen) abdecken, zu ermitteln.

3.1.2 Teststufen

Die Teststufen definieren sich nach dem Punkt in der Entwicklung eines Programms in dem sie eingesetzt werden. Diese Vorgehensweise wird durch das V-Modell in Abbildung 3.2 visualisiert.

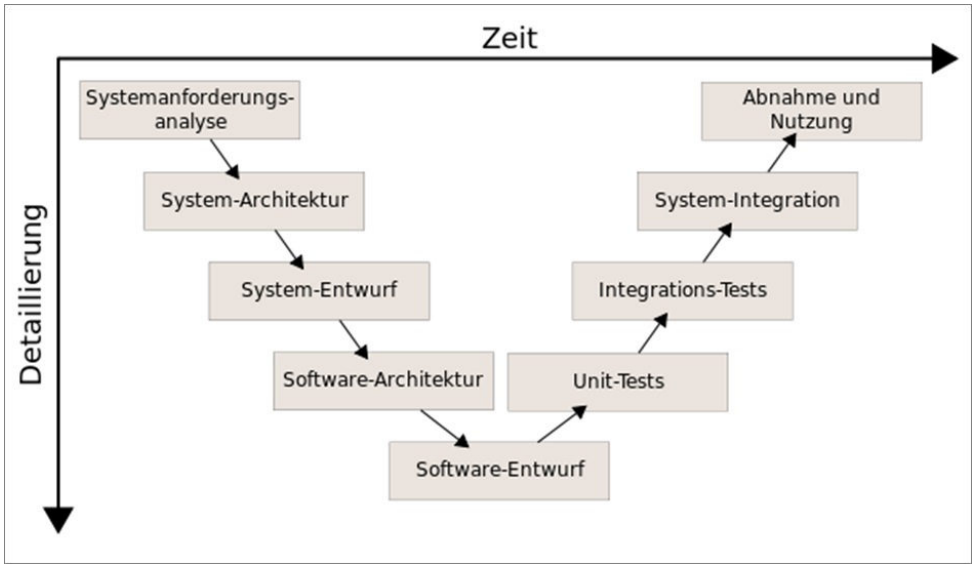

Abbildung 3.2: Darstellung der Teststufen im "V-Modell"
(Quelle: `https://de.wikipedia.org/wiki/V-Modell`, Zuletzt besucht: 07-02-2013)

Im Wesentlichen wird jeder Entwicklungsstufe eine Teststufe zugewiesen. Diese wiederum stellt sicher, dass auf der getesteten Stufe alle Spezifikationen erfüllt werden [42, 93].

Die hier beschriebene Vorgehensweise basiert auf der Annahme, dass das Programm auf traditionelle Weise spezifiziert (vom gesamten System immer detaillierter bis zu einzelnen Komponenten) und in umgekehrter Reihenfolge implementiert wird. Für andere Ansätze der Softwareentwicklung werden andere Modelle benötigt. Auch spielt hier die Komplexität des Projektes eine wichtige Rolle. Dementsprechend werden weitere Tests benötigt und an die Untergliederung der Teststufen angepasst [42, 93].

3.1.2.1 Komponententest

Bei Komponententests werden die kleinsten unabhängigen Bestandteile des Programms getestet. Diese verfügen oft nur über einfache Algorithmen mit geringer Komplexität und genau definierte Schnittstellen. Diese beiden Eigenschaften sorgen dafür, dass diese Komponenten mit wenigen Testfällen weitgehend vollständig und auch einfach getestet werden können [8, 42, 93, 95, 98].

Im Idealfall werden die Komponenten isoliert, das heißt unabhängig von anderen Komponenten getestet. Dies ist aber nicht immer möglich (z.B. wenn eine Komponente

für Datenbankzugriffe entwickelt wird). In diesem Fall werden solche externen Komponenten oder interagierende Komponenten des Programms simuliert - zumindest so weit diese zum Testen der aktuellen Komponente benötigt werden [8, 42, 93, 95, 98].

Diese Teststufe wird bereits sehr früh in der Entwicklungsphase eingesetzt. Dadurch ist es auch möglich Fehler frühzeitig zu erkennen und zu vermeiden. Sie stellt auch sicher, dass alle Bestandteile für sich betrachtet gemäß ihrer Spezifikation arbeiten, bevor sie zu größeren Bestandteilen zusammengefügt bzw. in diese integriert werden [8, 42, 93, 95, 98].

Stehen solche Komponententests zur Verfügung ist es auch möglich zu einem späteren Zeitpunkt in der Entwicklungsphase eine "Restrukturierung" des Codes vorzunehmen ("Code refactoring"). Dabei wird nur die interne Struktur verändert, während das Verhalten nach außen unverändert bleibt. In diesem Fall kann mit Hilfe der bestehenden Testfälle rasch und einfach festgestellt werden, ob das Programm weiterhin wie gewünscht arbeitet [8, 42, 93, 95, 98].

Neben Tests der Funktionalität werden oft auch nicht-funktionale Aspekte von Komponenten getestet (siehe dazu auch Unterabschnitt 3.1.2.3) und strukturelle Tests durchgeführt, um die Testabdeckung der Komponente zu verifizieren. Für dieses Qualitätskriterium soll ein möglichst hoher Wert erzielt werden. Dadurch kann die Anzahl an benötigten Testfällen zum Testen von Subsystemen gering gehalten werden. Die Gesamtzahl entspricht bei der beschriebenen Vorgehensweise der Addition der Testfälle, die benötigt werden um für jede Komponente eine hohe Testabdeckung zu erreichen. Falls dieses Ziel nur für ein Subsystem erreicht werden soll, ohne es vorher für dessen Komponenten erreicht zu haben, entspricht die Anzahl der benötigten Testfälle der Kombination aller Möglichkeiten [42, 60, 93, 95, 98].

3.1.2.2 Integrationstest

Im Integrationstest wird die Zusammenarbeit der einzelnen Bestandteile getestet. Dabei wird der Fokus auf die Schnittstellen zu diesen gelegt. Geprüft wird also, ob das Programm auch komplexere Abläufe korrekt ausführen kann [8, 42, 93].

Integrationstests können auch auf unterschiedlichen Ebenen erfolgen. Bei den integrierten Bestandteilen muss es sich nicht zwangsläufig um Komponenten (wie im vorangegangenen Abschnitt definiert) handeln, sondern es können auch komplette Systeme in ein anderes integriert werden. Im ersten Fall werden Integrationstests nach den Komponententests durchgeführt. Im zweiten Fall werden diese erst nach erfolgreichen Systemtests durchgeführt [8, 42, 93].

Die Integration von Bestandteilen kann inkrementell erfolgen oder es werden alle Bestandteile in einem Schritt integriert ("big bang"). Der Vorteil der ersten Methode ist, dass es hier leichter möglich ist Fehler früher zu erkennen. Dabei kann die Integration entweder "Bottom-up" oder "Top-down" erfolgen. Beide Alternativen sind in Abbildung 3.3 dargestellt. Welche dieser beiden Strategien angewandt wird, ist von der Systemarchitektur, von funktionalen Anforderungen oder diversen Aspekten der zu integrierenden Systeme/Komponenten abhängig [8, 42, 93].

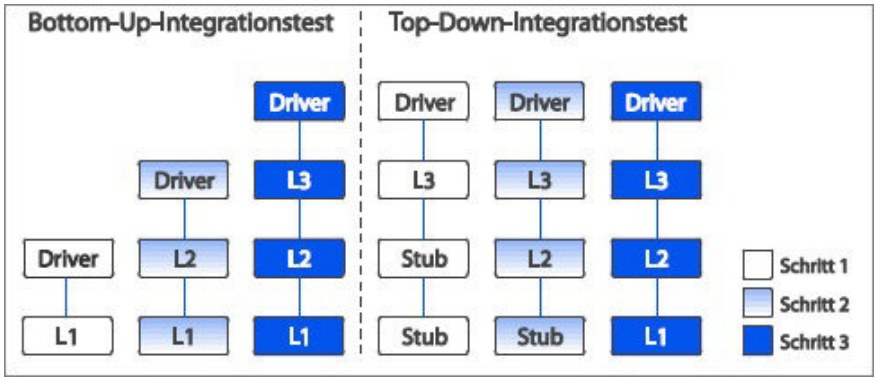

Abbildung 3.3: Alternative Ansätze zur Integration einzelner Komponenten

3.1.2.3 Systemtest

Bei Systemtests wird das gesamte System getestet. Dieser Vorgang erfolgt häufig in einer Testumgebung, in der die zukünftige Produktivumgebung (Hard- und Softwareumgebung), in der das Programm eingesetzt werden soll, simuliert wird. Dadurch soll verhindert werden, dass beim späteren Einsatz Probleme mit der Umgebung auftreten [8, 42, 92, 93].

Bei Systemtests wird nicht nur geprüft, ob das Programm die spezifizierten funktionalen Anforderungen erfüllt, sondern auch ob nicht-funktionale Anforderungen zufriedenstellend erfüllt werden [8, 42, 92, 93].

Funktionale Tests beziehen sich auf die Funktionen und Eigenschaften des Systems (auch Interoperabilität mit anderen Systemen). Nicht-funktionale Tests dagegen beziehen sich auf die Arbeitsweise des Systems. Dabei wird geprüft, wie sich das Programm in verschiedenen Zuständen verhält [8, 42, 92, 93].

Zum Testen nicht-funktionaler Aspekte eines Programms werden unter anderem die folgenden Tests verwendet:

Lasttest: Hier wird geprüft, wie sich das Programm verhält, wenn mehrere Nutzer gleichzeitig darauf zugreifen. Dabei wird getestet, ob das System auch bei hoher Belastung noch zufriedenstellend arbeitet ohne abzustürzen. Ein wichtiger Parameter hierfür sind die Antwortzeiten, also die Zeit, die vergeht, bis das Programm eine Antwort auf eine Eingabe zurückliefert [24, 92].

Mit Hilfe dieses Verfahrens kann auch festgestellt werden, ob es zu Inkonsistenzen bei Daten kommen kann, wenn mehrere Benutzer gleichzeitig Lese- und Schreibzugriff auf die Daten haben und davon auch Gebrauch machen [24, 92].

Im Zuge dieser Tests wird oftmals die Anzahl an simulierten Benutzern, wie auch die Dauer der Zugriffe dieser Benutzer auf das System variiert. Oftmals gibt es eine bestimmte Grenze, mit der diese Parameter eingeschränkt werden - diese entspricht der im Normalfall zu erwartenden maximalen Belastung des Systems. Eine Belastung des Systems über diese Grenze hinweg wird als “Stresstest” bezeichnet [24, 92].

Usability-Test: Fokus dieser Tests liegt auf dem Gebrauch des Systems durch den Benutzer, also ob und, wenn ja, wie einfach das System zu bedienen ist (vor allem, wenn der Benutzer es zum ersten Mal verwendet) [95].

Sicherheitstests: Diese sollen die Informationssicherheit möglichst hoch halten. Das Programm wird hier auf Schwachstellen getestet, die das Ausnutzen von Sicherheitslücken oder das Einschleusen von Code ermöglichen [95].

Daneben gibt es noch eine Reihe weiterer Tests, mit denen unter anderem die Zuverlässigkeit, Wartbarkeit oder Portabilität des Systems getestet werden können.

3.1.2.4 Regressionstest

Regressionstests stellen keine Teststufe an sich dar, sondern können in jeder von ihnen eingesetzt werden. Bei Regressionstests werden einzelne, Teilmengen oder alle Tests wiederholt ausgeführt. Damit sollen Fehler, die bei der Modifikation von bereits getestetem Code auftreten können, ermittelt werden. Die Testfälle selbst werden dabei nicht verändert [24, 59, 95].

3.1.3 Testprozess

Wie bereits erwähnt, ist das Testen von Software in der Regel eng mit dessen Entwicklung verknüpft. Bedingt durch die so gestiegene Komplexität entwickelte sich parallel zum Entwicklungsprozess ein eigener Testprozess. Dieser lässt sich grob in die in Abbildung 3.4 dargestellten Phasen unterteilen [8, 65, 79, 93].

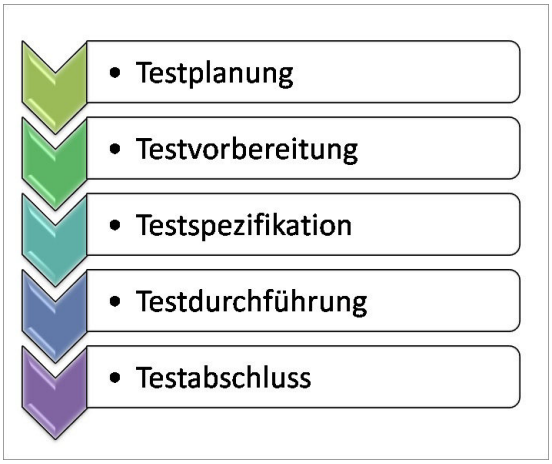

Abbildung 3.4: Darstellung der wichtigsten Phasen im Testprozess

3.1.3.1 Testplanung

In der Phase der Testplanung wird ein Zeitplan erstellt und es werden alle administrativen Tätigkeiten vorbereitet. Sie umfasst die laufende Dokumentation des Testprozesses. Zu diesem Zweck werden in dieser Phase auch einheitliche Vorlagen für wichtige Dokumente (Spezifikation der Testfälle, Fehlerberichte usw.) erstellt. Das wichtigste Dokument in dieser Phase (und auch für den gesamten Prozess) ist der Testplan [8, 65, 79, 93].

Im Testplan werden Ziele für jede Testphase definiert. Basierend auf diesen wird eine Strategie zur Erreichung derer festgelegt. Dabei erfolgt die Beschreibung des Testumfangs, des Risikomanagements (vor allem der Risikoabschätzung) und der Testabdeckung, die erreicht werden soll. Es werden die Kriterien für Beginn, Ende und Abbruch von Tests definiert. Außerdem wird die Testumgebung beschrieben und die Testdaten, welche verwendet werden, werden festgelegt [8, 65, 79, 93].

Ein weiterer wichtiger Punkt, der im Testplan beschrieben wird, ist die Organisation des Testteams. Dabei werden die Verantwortlichkeiten klar abgegrenzt und es wird genau festgelegt, wer welche Rolle im Testprozess zu erfüllen hat. Dieser Punkt trägt nicht unwesentlich zur Effizienz des Testprozesses bei, da hier im Idealfall zum Beispiel Kompetenzüberschneidungen vermieden werden. Außerdem sollte darauf geachtet werden, dass die Mitglieder gemäß ihrer Fähigkeiten und Erfahrungen optimal eingesetzt werden [8, 65, 79, 93].

Außerdem werden im Testplan alle benötigten Ressourcen aufgelistet, welche Werkzeuge und Hilfsmittel eingesetzt werden und welche Testmetriken herangezogen werden [8, 65, 79, 93].

Zunächst erfolgt die Erstellung des Testplans in sehr allgemeiner Form auf globaler Ebene. Aus diesem Testplan für den gesamten Prozess werden Testpläne für verschiedene Phasen und Teststufen abgeleitet, die auch einen höheren Detaillierungsgrad aufweisen.

3.1.3.2 Testvorbereitung

In der Testvorbereitung werden alle Aktivitäten, die im Testplan spezifiziert wurden, für die beiden folgenden Phasen vorbereitet. So werden unter anderem die einzusetzenden Werkzeuge zur Verfügung gestellt und die Testumgebung aufgesetzt. In diese werden dann die Objekte aus der Entwicklungsumgebung übernommen [8, 65, 79, 93].

Es werden Subsysteme definiert und mit Hilfe von Checklisten auf ihre Testbarkeit geprüft. In diesen wiederum werden die einzelnen Module, die getestet werden sollen, definiert. In Abhängigkeit von der Teststrategie wird auch festgelegt, wie diese Module getestet werden sollen [8, 65, 79, 93].

3.1.3.3 Testspezifikation

In der Phase der Testspezifikation werden alle Informationen zur Verfügung gestellt, die zur Durchführung der Tests erforderlich sind. Dazu werden die entsprechenden Testfälle erstellt. Es werden also die Vorbedingungen, die Eingabedaten und die erwarteten Ergebnisse festgelegt. Außerdem wird die Reihenfolge, in der die Tests ausgeführt werden, festgelegt [8, 65, 79, 93].

3.1.3.4 Testdurchführung

Die Testdurchführung ist eine der wichtigsten Phasen im Testprozess. Hier werden die Tests tatsächlich ausgeführt und aufgedeckte Fehler korrigiert. Um die Fehlerkorrektur möglichst optimal auszuführen ist eine effiziente Koordination von Entwicklungs- und Testaktivitäten erforderlich [8, 65, 79, 93].

Wichtig ist hier eine möglichst detaillierte Dokumentation um die gefundenen Fehler nachvollziehen, genau identifizieren und effizient beseitigen zu können. Um dieses Ziel zu erreichen ist es erforderlich, die Tests mehrmals auszuführen und deren Ergebnisse zu dokumentieren und zu bewerten [8, 65, 79, 93].

Dies ist insbesondere wichtig, da es gerade in dieser Phase eine ganze Reihe verschiedenster Fehlerquellen gibt. So muss der Fehler nicht unbedingt im getesteten Objekt selbst liegen. Es kann durchaus der Fall sein, dass bereits die Spezifikation fehlerhaft war. Das Testwerkzeug oder die Testumgebung können ebenfalls für ein falsches Ergebnis verantwortlich sein (so sie falsch eingesetzt werden oder die Einstellungen falsch gewählt wurden) [8, 65, 79, 93].

Ein weiterer wichtiger Punkt bei der Auswertung der Testergebnisse ist die genaue Analyse der Testfälle. Sind mehrere Komponenten voneinander abhängig (insbesondere, wenn die Ausgabedaten einer Komponente die Eingabedaten einer anderen Komponente sind), kann es durchaus vorkommen, dass viele von ihnen falsche Ausgaben liefern, obwohl nur eine von ihnen fehlerhaft ist. In diesem Zusammenhang sind auch Regressionstests wichtig, da die Korrektur von Fehlern in einer Komponente durchaus auch unerwünschte Auswirkungen auf andere Komponenten haben kann [8, 65, 79, 93].

3.1.3.5 Testabschluss

Der Test wird beendet, wenn er bestimmte Bedingungen, wie etwa eine definierte Qualitätsstufe, erreicht. Basierend auf der Dokumentation der Testphase werden Werte für vordefinierte Qualitätskriterien für alle Komponenten ermittelt. Dieser Vorgang wird auch für alle definierten Subsysteme (basierend auf den Ergebnissen der Komponenten, aus denen sich diese zusammensetzen) bis hin zum gesamten System durchgeführt [8, 65, 79, 93].

Diese Phase umfasst auch abschließende Tätigkeiten wie Vervollständigung, Abschluss und Archivierung der laufenden Dokumentation. Auch der Testprozess wird abschließend analysiert und bewertet um Verbesserungsmöglichkeiten für künftige Projekte aufzudecken [8, 65, 79, 93].

Der Testabschluss muss allerdings kein “sauberer” sein, bei dem alle definierten Ziele erreicht werden. Es kann auch vorkommen, dass eine Teststufe aus diversen Gründen abgebrochen werden muss [8, 65, 79, 93].

3.1.4 Testmethoden

Testen von Software ist ein sehr komplexes Aufgabengebiet. Dementsprechend hoch ist die Anzahl an Testmethoden, mit denen verschiedenste Aspekte eines Programms getes-

tet werden können. Für diese gibt es daher auch eine Vielzahl an möglichen Klassifikationen.

Eine wichtige Einteilung der Tests ist die bereits erwähnte in funktionale und nichtfunktionale Tests (siehe dazu auch Unterabschnitt 3.1.2.3). Funktionale Tests lassen sich wiederum in weitere Kategorien unterteilen, die sich auf die Eigenschaften dieser Tests beziehen [8]:

- Abhängig davon, ob sich die Testfälle auf die Spezifikation oder den Code selbst beziehen, kann eine Einteilung in *Black- und White-Box Tests* erfolgen.
- Abhängig davon, ob der Code tatsächlich ausgeführt wird oder nicht, kann eine Einteilung in *statische* und *dynamische* Testmethoden erfolgen.
- Abhängig davon, wie restriktiv die Ableitung von Testfällen erfolgt, kann eine Einteilung in *formale* (detaillierte Vorgaben) und *nicht-formale* Testmethoden (höhere Variabilität der Testfälle) erfolgen.
- Abhängig davon, welche Ziele verfolgt werden sollen, kann eine Einteilung in *Error Finding* (Fehler sollen gefunden werden) und *Function Detection* (Funktionalität soll bewiesen werden) erfolgen.

Die bedeutendsten Einteilungen sind der sogenannte Box Ansatz sowie die Einteilung in statische und dynamische Testverfahren (siehe dazu auch Abbildung 3.5). Daher wird auf diese in den folgenden Abschnitten näher eingegangen.

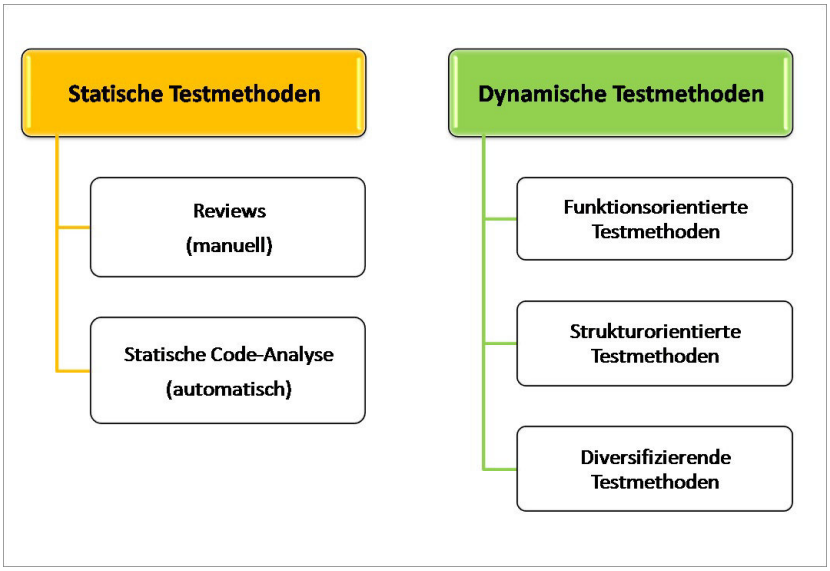

Abbildung 3.5: Wichtigste Vertreter statischer und dynamischer Testmethoden

3.1.4.1 Statisches Testverfahren

Beim statischen Testverfahren handelt es sich um ein analysierendes Verfahren, für das keine Codeausführung erforderlich ist. Zu diesem Verfahren zählen

- Reviews (manuell)
- Statische Code-Analysen (automatisiert)

Reviews werden oft während oder kurz nach der Entwicklung durchgeführt. Dabei wird der Code von einem oder mehreren Gutachtern analysiert (Korrektur gelesen). Dieser Vorgang erfolgt noch bevor ein Test durchgeführt wird. Ein großer Vorteil von Reviews ist, dass Fehler so in einem sehr frühen Stadium gefunden und dementsprechend kostengünstig korrigiert werden können. Diese Technik wird aber nicht nur für den Code sondern auch für Dokumente oder die Systemarchitektur durchgeführt [59, 65, 93].

Es werden 4 verschiedene Arten von Reviews unterschieden [59, 65, 93]:

1. *Technisches Review*: Dokumente/Code werden auf die Erfüllung aller Spezifikationen geprüft.
2. *Informelles Review*: Dieses entspricht dem technischen Review, allerdings ist keine Dokumentation erforderlich, wodurch es weniger Zeit in Anspruch nimmt.
3. *Walkthrough*: Der Autor präsentiert seine Arbeit, die anschließend von einer Gruppe von gleichgestellten Mitarbeitern diskutiert wird (z.B. hinsichtlich Alternativen).
4. *Inspektion*: Sehr formaler Vorgang, der auf Regeln und Checklisten basiert.

Im Wesentlichen werden auch bei der *statischen Code-Analyse*, welche zur Compilezeit stattfindet, die selben Ziele verfolgt. Hier wird eine Reihe formaler Prüfungen ausgeführt um Fehler noch vor der ersten Ausführung des Codes zu finden. Weitere Ziele, die mit dieser Art Code-Analyse verfolgt werden, sind die Verbesserung der Wartbarkeit des Codes und das Aufzeigen von Abhängigkeiten und Inkompatibilitäten in Software-Modellen. Oft ist es so auch möglich Fehler zu finden, die auch mit Hilfe dynamischer Testverfahren nicht gefunden werden können [12, 59, 93].

3.1.4.2 Dynamisches Testverfahren

Dynamische Testverfahren unterscheiden sich von statischen dadurch, dass tatsächlich der Code ausgeführt wird um ihn zu testen. Eine wichtige Aufgabe noch vor der Anwendung solcher Testverfahren ist die genaue Beschreibung der Testfälle [59].

In diesem Bereich gibt es zahlreiche Möglichkeiten unterschiedlichste Aspekte von Code zu testen bzw. auch wie diese Aspekte getestet werden können. Mögliche Kategorien von dynamischen Testverfahren sind :

- *Funktionsorientierte Testmethoden*: Geprüft wird, ob die Software die spezifizierte Funktionalität zur Verfügung stellt (Black-Box Test) [59].
- *Strukturorientierte Testmethoden*: Testfälle bauen auf der Struktur des Codes auf (White-Box Test) [59].
- *Diversifizierende Testmethoden*: Umfassen Techniken, die die Testergebnisse verschiedenster Versionen der selben Software vergleichen [59].

Innerhalb dieser Kategorien können noch weitere Subkategorien unterschieden werden. Die Kategorisierung von Tests in Black- und White-Box Tests ist eine der wichtigsten beim Testen von Software. Daher werden diese beiden Testmethoden im folgenden Abschnitt näher beschrieben.

3.2 Black- und White-Box Testen

Diese Arbeit beschäftigt sich mit der Entwicklung eines Ansatzes zum White-Box Testen. Da dieses eng mit dem des Black-Box Testens verknüpft ist, werden beide Verfahren in diesem Abschnitt detaillierter vorgestellt. Abbildung 3.6 stellt den wesentlichsten Unterschied der beiden Methoden dar: Im Gegensatz zu Black-Box Tests ist bei White-Box Tests das Testobjekt transparent, das heißt das Wissen über dessen inneren Aufbau wird beim Testen genutzt.

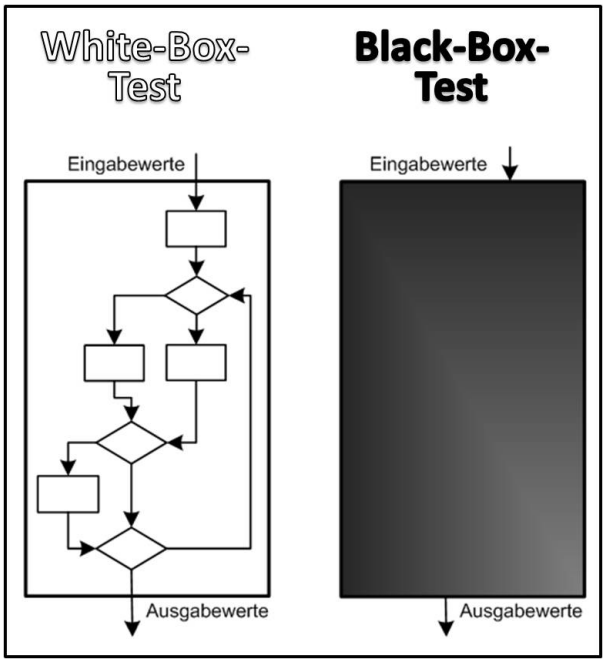

Abbildung 3.6: Vergleichende Darstellung von Black- und White-Box Tests [98]

3.2.1 Black-Box Testen

Diese Art von Tests wird ohne Berücksichtigung der inneren Struktur des zu testenden Objekts entwickelt und durchgeführt. Geprüft wird lediglich, ob das Programm bei gegebenen Eingaben auch die dazu passenden, Ausgaben erzeugt. Wie genau diese erzeugt werden (also welche Teile des Codes ausgeführt werden), ist dabei nicht von Bedeutung [92, 93].

Aufgrund der Tatsache, dass hier nur geprüft wird, ob ein getestetes Programm auch alle Spezifikationen erfüllt, ist es nicht immer möglich alle Fehler in den Komponenten zu finden. Sollten zwei Fehler sich gegenseitig aufheben, wird eine korrekte Ausgabe erzeugt. Dies kann allerdings mit Hilfe von Black-Box Tests nicht festgestellt werden [92, 93].

Aus diesen Eigenschaften leitet sich auch das hauptsächliche Einsatzgebiet von Black-Box Tests ab. Diese werden üblicherweise auf der Stufe der Systemtests eingesetzt [92, 93].

Diese Art von Tests wird verwendet um Systeme auf die folgenden Fehlerarten zu prüfen [24]:

- Fehler, die in Funktionspfaden, die vom System unterstützt werden, auftreten.
- Fehler, die bei Berechnungen im System auftreten.
- Fehler, die den möglichen Wertebereich von Daten, die vom System verarbeitet werden, betreffen.

3.2.1.1 Techniken des Black-Box Testen

Bei der Verwendung von Black-Box Tests werden verschiedenste Techniken angewandt um die Anzahl an Testfällen zu verringern ohne die Testabdeckung zu beeinträchtigen [65, 92, 93]. Diese lassen sich in zwei Gruppen einteilen:

1. Spezifikationsbasierte Techniken

   a) Äquivalenzklassen

   Statt alle möglichen Eingaben zu testen, werden Äquivalenzklassen gebildet. Dazu werden diese Werte nach bestimmten Kriterien in Gruppen zusammengefasst. Die Einteilung erfolgt oft in gültige und ungültige Werte. Getestet wird dann auf Bereichs- und Domänenfehler. Liefert ein Testfall beziehungsweise eine Gruppe von mehreren Testfällen ein korrektes Ergebnis, wird für alle Eingabewerte in dieser Gruppe angenommen, dass sie gültige Ausgaben liefern. Dies gilt auch für ungültige Ein- und Ausgaben [24, 60, 93, 98].

   b) Grenzwertanalyse

   Hier werden die Tests an den Grenzen, die den Wertebereich gültiger Eingaben definieren, ausgeführt. Dabei werden die Grenzwerte selbst sowie Werte, die eine Einheit über beziehungsweise unter diesen liegen, getestet [24, 60, 93, 98].

c) Entscheidungstabellen

Mit Entscheidungstabellen können komplexe Abhängigkeiten im Programm übersichtlich dargestellt werden. Dabei kann es sich um Abhängigkeiten von Bedingungen, aber auch umfangreicheren Abläufen handeln. Durch die Struktur sollen Vollständigkeit und Widerspruchsfreiheit gewährleistet und gleichzeitig Redundanz vermieden werden [49, 93].

d) Modellbasierter Test

Zu dieser Gruppe zählen unter anderem Zustandsbezogene Tests. Für diese werden die Testobjekte als Zustandsdiagramme modelliert. Diese Vorgehensweise findet vor allem in Bereichen Anwendung, in denen nicht nur die Eingabedaten sondern auch die Ergebnisse vorgelagerter Aktivitäten des Systems von Bedeutung sind [93].

e) Anwendungsfallbasierter Test

Getestet wird, ob das System die in den Anwendungsfällen spezifizierten Anforderungen erfüllt. Diese werden für verschiedenste Gruppen von potenziellen Anwendern definiert und können in unterschiedlichem Ausmaß sowohl Überschneidungen als auch Abweichungen aufweisen [93].

2. Erfahrungsbasierte Techniken

a) Testen auf Basis der Erfahrungen des Testers

b) Design, Durchführung und Dokumentation des Testfalls erfolgen in Echtzeit ohne vorherige Planung. Meist wird unmittelbar nach oder während der Implementierung getestet.

c) Ausfallbasierte Tests, wie zum Beispiel *Error Guessing*, bei dem der Tester genau jene Teile des Codes testet, in denen er, basierend auf seinen Erfahrungen und seiner Intuition, Fehlerquellen vermutet [11].

Natürlich gibt es auch in diesem Fall andere mögliche Klassifizierungen. Auch erhebt die Liste an Techniken, die hier beispielhaft für jede der beschriebenen Gruppen angeführt wurden, keinen Anspruch auf Vollständigkeit hinsichtlich der möglichen Techniken die beim Black-Box Testen angewandt werden können.

3.2.1.2 Vor- und Nachteile des Black-Box Testen

Vorteile:

- Es kann besser verifiziert werden, ob das Gesamtsystem auch wirklich alle spezifizierten Anforderungen erfüllt [65, 92, 93].
- Bei geeigneter Spezifikation können die semantischen Eigenschaften des Testobjekts gut getestet werden [65, 92, 93].

- Systematisch erstellte Testsequenzen können für plattformunabhängige Implementierungen übernommen werden [65, 92, 93].

Nachteile:

- Es ist ein größerer organisatorischer Aufwand erforderlich, da beim Black-Box Testen das Testteam im Idealfall unabhängig vom Entwicklungsteam agieren sollte. So soll sichergestellt werden, dass die Testfälle nicht durch Wissen um den inneren Aufbau des Programms verfälscht werden [65, 92, 93].

- Werden zusätzliche Funktionen implementiert, aber schlecht dokumentiert, ist es möglich, dass diese gar nicht oder nur durch Zufall getestet werden [65, 92, 93].

- Ist die Spezifikation bereits unzulänglich, sind auch die darauf aufbauenden Testsequenzen unbrauchbar [65, 92, 93].

3.2.2 White-Box Testen

Im Gegensatz zu Black-Box Tests bauen White-Box Tests auf der inneren Struktur des Testobjekts auf. Ziel dieser Art von Tests ist es, logische Fehler im Programm zu finden und die Testabdeckung zu prüfen. Der letztere Aspekt weist auch darauf hin, dass das wesentliche Einsatzgebiet von White-Box Tests in der Stufe der Komponententests liegt [19, 92, 93].

Dies zeigt auch eine Limitierung im Zusammenhang mit dieser Art von Tests: Arbeiten zwei Komponenten für sich genommen korrekt, ist noch nicht gewährleistet, dass sie auch dann korrekte Ergebnisse liefern, wenn sie interagieren. Solche Fehler sind mit White-Box Tests aufgrund ihrer Methodik nicht immer einfach feststellbar [19, 92, 93].

Da es sich hier um ablaufbezogene Tests handelt, können sie bei optimaler Anwendung gewisse Bedingungen erfüllen [24]:

- Es kann sichergestellt werden, dass jeder Pfad in einer Komponente zumindest einmal ausgeführt wird.

- Es können alle möglichen Entscheidungsausgänge (wahr und falsch) erprobt werden.

- Es können alle Schleifen ausgeführt werden (sowohl an ihren Grenzen als auch innerhalb ihrer Betriebsgrenzen).

- Es können alle internen Datenstrukturen auf ihre Gültigkeit geprüft werden.

Natürlich sind hier in der Praxis gewisse Einschränkungen zu beachten, da alle diese Parameter auch wesentlichen Einfluss auf den Testaufwand haben (siehe Unterabschnitt 3.1.1.3).

3.2.2.1 Techniken des White-Box Testen

Auch innerhalb des White-Box Testens gibt es eine Reihe von Techniken, die angewandt werden können:

1. Einfügen von Fehlern

   Hier soll das Verhalten des Testobjekts bei ungültigen Eingaben untersucht werden. Dadurch kann festgestellt werden, ob die gewünschten Reaktionen (meist Return-Codes oder anderweitige Anzeigen von Fehlern) erfolgen. Dies lässt Rückschlüsse auf die Stabilität des Programms zu [24].

2. Testen durch Fehlerbehandlung

   Ziel dieser Art von Tests ist es sicherzustellen, dass das Programm richtig auf Fehler reagiert. Es wird eine Reihe von ungültigen Werten an das Programm übergeben um die Effizienz der Fehlerbehandlungsroutinen zu prüfen. So kann festgestellt werden, ob ein Programmabsturz eintritt oder ob sinnvolle Fehlermeldungen zurückgeliefert werden. Am wirkungsvollsten sind solche Fehlerbehandlungsroutinen auf Komponentenebene, weshalb solche Tests dort zur Anwendung kommen. Da aber nicht alle ungültigen Werte getestet werden können, muss an dieser Stelle eine möglichst sinnvolle Auswahl getroffen werden [24].

3. Speicherproblemtests

   Mit Hilfe dieser Tests soll festgestellt werden, ob Speicherprobleme bei einzelnen Komponenten oder auch bei der Anwendung des Gesamtsystems auftreten. Bei einzelnen Anwendungen kann es durchaus zu Leistungsproblemen kommen wenn der zugewiesene Speicher nicht freigegeben wird [24].

4. Kettentests

   Kettentests untersuchen die Zusammenarbeit von Funktionen des Systems. Diese können in einer Komponente enthalten sein oder sich auch auf Funktionen mehrerer Komponenten zusammensetzen. Dabei werden auch die Parameter, die von einer Funktion an eine andere übergeben werden, auf ihre Gültigkeit - was Datentyp und Wert betrifft - geprüft. Diese Form von Tests setzt die - zumindest einmalige - Ausführung aller möglichen Anweisungen voraus [24].

5. Tests zur Ermittlung des Überdeckungsgrades

   Diese Tests sollen bestimmte Mindeststandards hinsichtlich verschiedenster Parameter sicherstellen (siehe Unterabschnitt 3.1.1.3) [24].

Aufgrund ihrer Spezifikation zeigt sich, dass White-Box Tests sehr schnell sehr umfangreich werden können. Für vollständige White-Box Tests müssten sehr viele unterschiedliche Werte getestet werden. Daher werden häufig nur Grenzwerte getestet (plus/minus eine Einheit). Liefert der Test für den gültigen Wert ein korrektes Ergebnis wird häufig davon ausgegangen, dass die Funktion für alle gültigen Werte solche Ergebnisse liefert.

Gleiches gilt im umgekehrten Fall für ungültige Werte.
Die Komplexität der Tests ist auch von der Überdeckungsart (Anweisungs-, Pfadüberdeckung usw.) und dem Grad der Überdeckung, der erreicht werden soll, abhängig. Wird auf Anweisungsabdeckung wert gelegt, werden weniger Tests benötigt, als im Falle der Pfadabdeckung notwendig wären. Allerdings gibt es auch Einschränkungen: Ist eine Programmanweisung so geschrieben, dass für die meisten Werte - gültig oder ungültig - dieselbe Bedingung erfüllt ist, ist es nur mit sehr viel Aufwand möglich an dieser Stelle einen hohen Abdeckungsgrad zu erreichen, also sicherzustellen, dass auch eine andere Bedingung zumindest einmal erfüllt ist.

3.2.2.2 Vor- und Nachteile des White-Box Testen

Vorteile:

- White-Box Tests eignen sich vor allem zum Testen interner Funktionen. Werden diese auf Komponentenstufe angewandt, ermöglichen sie das frühe Auffinden und kostengünstige Beheben von Fehlern [19, 92, 93].
- Der organisatorische Aufwand zur Durchführung solcher Tests ist geringer als beispielsweise beim Black-Box Testen. Da die Tester über genaue Kenntnisse der internen Funktionen verfügen müssen, werden die Tests üblicherweise von den Entwicklern selbst geschrieben [19, 92, 93].
- Der Aufwand für die Durchführung von White-Box Tests kann sehr hoch sein. Da aber auch viele Testfälle ähnlich sind, gibt es eine Anzahl von Tools zur Testautomatisierung [19, 92, 93].

Nachteile:

- Es wird nur geprüft, ob einzelne Funktionen auch korrekte Ergebnisse liefern. Die Spezifikation selbst wird dabei nicht berücksichtigt [19, 92, 93].
- Es können gewisse Fehler unentdeckt bleiben. Eine Ursache dafür kann auch der Umstand sein, dass "um den Fehler herum" getestet wird. Da die Tests oft von den Entwicklern geschrieben werden, besteht das Problem der fehlenden Objektivität: Gewisse Dinge werden nicht getestet, weil vom Entwickler ohnehin keine Fehler gemacht werden [19, 92, 93].

3.3 Testautomatisierung

Unter Testautomatisierung versteht man die Unterstützung des Testprozesses durch Tools. Dadurch soll vor allem der Testaufwand verringert und Zeit gespart werden. Allerdings sind die Erwartungen, die an die Testautomatisierung gestellt werden, oft überzogen. Diese können nicht immer erfüllt werden, vor allem weil auch die Testwerkzeuge verschiedensten Einschränkungen unterliegen.

Daher sollte die Einführung von derartigen Testwerkzeugen genauestens geplant werden. Nur wenn Klarheit darüber herrscht, wo im Testprozess und noch genauer für welche Aufgaben es eingesetzt werden kann, kann Testautomatisierung eine sinnvolle Unterstützung/Alternative zu manuellen Tests sein.

3.3.0.3 Entscheidungsfaktoren

Es gibt einige Faktoren, von denen es abhängt, ob der Einsatz von Testwerkzeugen in einem Projekt sinnvoll ist. Diese spielen auch eine Rolle bei der Bestimmung des Ausmaßes, in dem Testautomatisierung durchgeführt werden kann.

Ein Faktor ist etwa die *Größe des Projektes*. Je kleiner das Projekt, desto weniger Vorteile bringt eine Testautomatisierung. Das ist vor allem dann der Fall, wenn diese zum ersten Mal eingesetzt werden soll. Hier kann der Aufwand zur Einführung den Nutzen leicht übertreffen.

Eng mit der Größe des Projektes ist auch der *Umfang der Ressourcen* (finanzieller und personeller Natur, Zeit,...), die dafür zur Verfügung stehen, verbunden. Abhängig vom zeitlichen, finanziellen und personellen Spielraum wird auch eine Entscheidung für oder gegen das Automatisieren von Tests getroffen.

Weniger von der Größe, als vielmehr von der *Komplexität des Projektes* ist der Aufwand zur Erstellung automatisierter Testfälle abhängig. Wenn zu viele Ressourcen in die Automatisierung der Testfälle investiert werden müssen, sodass mit vergleichbarem Aufwand mehrere manuelle Ausführungen realisiert werden könnten, wird eine solche eher nicht in Betracht gezogen.

Bei funktionalen Tests ist eine genaue *Kenntnis der Ausgabewerte* bei gegebenem Input erforderlich, da nur so ein echtes automatisches Testen möglich ist. Auch sollte eine gewisse Stabilität der Ausgabewerte gegeben sein, das bedeutet, dass sich die Anzahl möglicher Ergebnisse in einem genau definierten Bereich bewegen sollte und die Ergebnisse selbst sollten bei gleichem Input auch gleich oder sehr ähnlich sein sollten. Ist dies nicht der Fall, so ist auch die Prüfung der Ergebnisse auf deren Korrektheit mit hohem Aufwand verbunden.

Entscheidend ist auch das *Kompetenzniveau des Teams*. Hier ist die Entscheidung stark abhängig vom bereits vorhandenen Wissen des Teams über grundsätzliche Aspekte der Testautomatisierung und der Tools, die zu diesem Zweck eingesetzt werden sollen, aber auch wie viel Aufwand investiert werden müsste, um die Tester diesbezüglich einzuschulen.

Natürlich ist die Entscheidung auch abhängig von der *Qualität des Test-Frameworks*, das eingesetzt werden kann/soll. Wie im vorhergehenden Punkt bereits erwähnt, spielen hier vor allem die Schwierigkeit des Umgangs mit diesem Tool sowie dessen Qualität (Bugs im Tool; enorm lange Lade- und Verarbeitungszeiten, etc.) eine wesentliche Rolle.

Wichtig für eine sinnvolle Entscheidung ist auch eine sinnvolle Abschätzung der *Anzahl der Testausführungen/-regressionen*. Dieser Aspekt ist auch wieder stark vom Projekt an sich abhängig und davon, wie viele Modifikationen es bis zu seiner Fertigstellung beziehungsweise nach seiner vorübergehenden Fertigstellung geben wird.

Wesentlichen Einfluss hat auch die *Art des Projektes*. Handelt es sich um ein Projekt, das zu Beginn eher grundsätzliche Funktionalität zur Verfügung stellt und für das es in der Zukunft viele Erweiterungen beziehungsweise Plugins geben wird, so wird auch eine Testautomatisierung sinnvoll sein, da ja immer wieder die korrekte Funktion der bestehenden Programmteile (z.B. nach Einführung eines neuen Plugins) gewährleistet sein muss. Da der Vorgang hierfür keine wesentlichen Änderungen erfahren wird, kann eine Testautomatisierung hier Vorteile bringen.

Einfluss auf die Entscheidung können auch die *zukünftigen Projekte* haben. Wenn es wahrscheinlich ist, dass einige künftige Projekte dem aktuellen sehr ähnlich sein werden, wird man wahrscheinlich bestimmte Codeteile wiederverwenden. Wenn für diese bereits automatisierte Tests bestehen, können auch diese mit leichten Adaptionen schnell und einfach eingesetzt werden.

Alle diese Faktoren müssen bei der Entscheidung darüber, ob Testautomatisierung eingesetzt werden soll, beachtet werden. Dazu können noch weitere Entscheidungskriterien kommen, die sich aus dem Projekt selbst, der späteren Produktivumgebung des Systems oder dem Unternehmen (Struktur, Philosophie, usw.) ableiten. Vor allem aber müssen diese Dinge bereits früh in der Planungsphase beachtet werden, da sie starken Einfluss auf den weiteren Testprozess haben können.

3.3.0.4 Anforderungen an Testautomatisierung

Die Komplexität der Entscheidung über den Einsatz von Testautomatisierung leitet sich auch aus den Anforderungen ab, die an diese gestellt werden. Abgesehen von den offensichtlichen Erwartungen wie Kosten- und Zeitersparnis gibt es noch andere [60]:

- Tests sollten in der selben Sprache geschrieben werden wie das Programm selbst. Dadurch können sowohl der Sourcecode als auch die Tests innerhalb derselben Entwicklungsumgebung verwaltet und ausgeführt werden. Im Falle von White-Box Tests kann so auch der zusätzliche Aufwand (z.B. mehr Personal oder Schulungen, wenn die Entwickler die Programmiersprache zur Testentwicklung nicht beherrschen) zur Entwicklung der Tests gering gehalten werden [60].

- Trennbarkeit von Anwendungs- und Testcode
  Dadurch soll verhindert werden, dass der Test Auswirkungen auf das Programm hat [60].

- Unabhängige Ausführung einzelner Testfälle
  Dadurch soll verhindert werden, dass sich einzelne Tests gegenseitig beeinflussen. Vor allem wenn mehrere Tests ausgeführt werden, kann so verhindert werden, dass sich Fehler einschleichen oder fortpflanzen [60].

- Beliebige Zusammenstellungen von Testfällen
  Es werden nicht immer die selben Tests ausgeführt. Daher sollte es auch möglich sein, die einzelnen Testfälle je nach Bedarf zusammenzufassen [60].

- Ergebnisse müssen sofort feststellbar sein

Neben der Erfüllung dieser Grundanforderungen sind viele Testwerkzeuge in der Lage weitere Aufgaben zu übernehmen. Dadurch können sie in vielen Bereichen innerhalb des Testprozesses eingesetzt werden.

3.3.0.5 Einsatzbereiche und Testwerkzeuge

Testwerkzeuge können für verschiedene Aufgaben im Testprozess herangezogen werden.

- *Planung und Verwaltung*
  In diesen Bereich fallen Werkzeuge, die zur Fehlerverfolgung ("Bug Tracking Tools") herangezogen werden. Sie dienen der Dokumentation und Überwachung aller gefundenen Fehler bis zu deren Behebung. Daneben gibt es auch solche, die die Verwaltung der Testfälle, die automatische Berichterstellung, die Fortschrittsüberwachung und das Konfigurationsmanagement unterstützen [8, 24].

- *Testfallerstellung*
  Testfallgeneratoren werden zur Ableitung von Testfällen aus einem formalen Modell herangezogen. Bei diesem Modell kann es sich um einen Graphen, Pseudocode oder Sourcecode handeln. Solche Werkzeuge werden häufig eingesetzt, um bereits bei der Erzeugung der Testfälle eine bestimmte Testabdeckung zu garantieren. Meist werden so allerdings nur logische Testfälle erzeugt, die dadurch gekennzeichnet sind, dass sie lediglich aus einer Abfolge von Aktionen oder der Angabe von Anzahl und Typ der Eingabedaten bestehen. Daher müssen diese vom Tester manuell angepasst werden - vor allem was die Festlegung konkreter Eingabedaten betrifft (siehe dazu auch Abschnitt 3.4 und Unterabschnitt 3.5.1) [8, 24].

- *Testdurchführung*
  In diesem Bereich gibt es die größte Vielfalt an Testwerkzeugen. Dazu zählen etwa *funktionale Testwerkzeuge*, die verschiedenste eingebaute und/oder benutzerdefinierte Prüfungen durchführen. Dazu werden Aktionen aufgezeichnet, automatisch erneut ausgeführt und anschließend die Ergebnisse des aufgezeichneten und des tatsächlichen Verhaltens verglichen [8, 24].
  Daneben fallen in diesen Bereich auch *Werkzeuge für Leistungstests* (Belastungs- bzw. Stresstests), aber auch solche, die eine statische Analyse ermöglichen. In diesem Fall wird das Programm nicht ausgeführt. Diese Testwerkzeuge ermöglichen z.B. das Testen von Qualitätskriterien wie Wartbarkeit, Änderbarkeit des Codes, aber auch das Prüfen der Datenbankkonsistenz oder der Einhaltung von Programmierrichtlinien [8, 24].
  Aber nicht nur die Testdurchführung selbst, sondern auch die Ergebnisanalyse kann mit Hilfe von *Komperatoren*, die Unterschiede zwischen den tatsächlichen und erwarteten Ergebnissen aufzeigen, automatisiert werden [8, 24].
  Bei der Programmausführung selbst können *dynamische Analysewerkzeuge* Informationen über die Testabdeckung liefern oder auch Speicherlecks, die zu Instabilitäten führen können, liefern [8, 24].

Um mehr Informationen über das Verbrauchsverhalten eines Systems zu gewinnen können *Werkzeuge zur Systemüberwachung* eingesetzt werden. So kann der Verbrauch von Ressourcen (CPU-Last, Speicherausnutzung, Netzwerklast usw.) aufgezeichnet und analysiert werden [8, 24].

Der Einsatz solcher Testwerkzeuge sollte bereits sehr früh in der Projektplanung beschlossen werden. Nicht nur allgemeine Überlegungen zur Testautomatisierung spielen hier eine Rolle. Da viele dieser Werkzeuge nur einen sehr eng abgesteckten Einsatzbereich haben, ist es wichtig sich über die Ziele, die man verfolgt im Klaren zu sein. Auch der Aufwand, der mit deren Einsatz verbunden ist, sollte nicht unterschätzt werden. Dieser ist eng mit dem Projekt selbst und den Erfahrungen des Testteams verknüpft [8, 24].

3.3.0.6 Vor- und Nachteile einer Testautomatisierung

Vorteile:

- Eine billige und effiziente Durchführung von Tests ist möglich - vor allem im Falle von Regressionstests, aber auch von Kompatibilitäts- und Konfigurationstests [8, 24].
- Eine Durchführung von Tests, die manuell nicht ausgeführt werden können, wird ermöglicht, wie zum Beispiel Tests die zur Aufdeckung von Speicherproblemen herangezogen werden. Durch den Einsatz von Testwerkzeugen ist es auch einfacher möglich bestimmte Metriken zu ermitteln (z.B. Testabdeckung) [8, 24].
- Im Zusammenhang mit datengetriebenem Testen (wesentliches Merkmal dieser Art von Tests ist die strikte Trennung von Testdaten und -ausführung) können recht einfach zahlreiche mögliche Eingabewerte getestet werden. Auch der Aufwand für die Erstellung und Verwaltung von Testdaten kann so gering gehalten werden [8, 24].
- Ein weiterer Vorteil ist die Stabilität der Eingangsdaten. Es kann recht einfach sicher gestellt werden, dass bei wiederholten Tests die selben Eingangswerte verwendet werden [8, 24].
- Eng mit dem oberen Vorteil verwandt ist die Möglichkeit der Reproduktion von Softwarefehlern. Wird ein Fehler gefunden, ist es einfach möglich die Schritte, die zu diesem geführt haben, zu wiederholen [8, 24].
- Langfristig können durch die Wiederverwendung Kosten gespart werden [8, 24].
- Die Fehleranfälligkeit ist im Vergleich zu manuellen Tests geringer (z.B. Unkonzentriertheit der Tester - vor allem bei gleichförmigen Tests) [8, 24].
- Es steht eine große Vielfalt an Testwerkzeugen, die eingesetzt werden können, zur Verfügung [8, 24].

Nachteile:

- Nicht immer können (alle) Testfälle automatisiert werden [8].
- Der Investitionsaufwand ist hoch. Viele Testwerkzeuge sind teuer und/oder nur entgeltlich in vollem Umfang nutzbar. Einsparungen sind daher erst nach langer Zeit realisierbar. Dadurch besteht auch das Problem einer ungewollt langen Bindung an ein bestimmtes Testwerkzeug, da oftmaliger Wechsel sehr kostenintensiv ist [8].
- Ist die Testautomatisierung erfolgreich, kann es zu Mitarbeitereinsparungen kommen, die sich erst in weiterer Folge negativ auswirken. Vor allem wenn zu einem späteren Zeitpunkt testintensive Projekte realisiert werden sollen beziehungsweise Projekte mit hohem Anteil an manuellen Tests [8].
- Wenn sich das Programm leicht ändert, werden bei zuvor erstellten Test-Scripts Änderungen notwendig, was einen hohen Zeit- und Arbeitsaufwand erfordert => je mehr Scripts, desto höher die Wartungskosten [8].
- Die hohe Anfangsinvestition in die Automatisierung der Tests führt dazu, dass auch Fehler erst viel später gefunden werden und es wegen der verzögerten Behebung zu höheren Kosten kommt. Dieses Problem tritt speziell dann auf, wenn der Einsatz von Testwerkzeugen bereits in der Planungsphase schlecht vorbereitet wurde [8].
- Komplexe automatisierte Tests sind aufwändig in der Erstellung (z.B. schwer abzubildende Spezialfälle) und können womöglich nicht (oft) wieder verwendet werden (Code-Wiederverwendbarkeit) oder sind schwer an ein modifiziertes Testobjekt bzw. an eine modifizierte Testumgebung anzupassen [8].
- Die Wartung kann Kosten verursachen oder es kann zum Verlust der Fähigkeit qualifizierter Wartung kommen, wenn automatisierte Tests lange und oft wieder verwendet werden (z.B. natürlicher Abgang verantwortlicher Mitarbeiter ohne Nachbesetzung) [8].

3.4 Testdaten

Wichtiger noch als die Entscheidung darüber, *wie* getestet werden soll (z.B. manuell oder automatisiert), ist die Entscheidung, *was* getestet werden soll, vor allem auch weil von letzterer Frage die erste in direkter Folge abhängt. Wie bereits in den vorangegangenen Abschnitten gezeigt, können viele Aspekte eines Programms getestet werden.

Damit verbunden ist auch die Frage, mit welchen Daten diese Tests durchgeführt werden sollen. Diese Daten können genau festgelegt sein, aber auch eine gewisse Variabilität aufweisen. Dies hängt nicht zuletzt von der Art des Tests (z.B. funktionaler Test vs. Belastungstest) ab.

Grundsätzlich werden für Tests drei Kategorien von Daten verwendet[4]:

[4] `https://de.wikipedia.org/wiki/Testumgebung`, Zuletzt besucht: 24-02-2013

- Eingabedaten, die vom Testobjekt auch verarbeitet werden
- Daten, auf die das Testobjekt zugreift, ohne diese zu ändern
- Ausgabedaten des Testobjekts (die mit den erwarteten Werten verglichen werden)

In der Regel wird der gültige Wertebereich für Eingabedaten im Programm eingeschränkt. Daher werden für Tests Äquivalenzklassen und/oder Grenzwerte herangezogen (siehe Unterabschnitt 3.2.1). Dadurch soll sichergestellt werden, dass das Programm bei gültigen Werten erwünschtes Verhalten und bei ungültigen Werten kein unerwünschtes Verhalten zeigt. Daraus ergeben sich aber auch gewisse Anforderungen an die Testdaten selbst sowie an den Prozess, mit dem diese erstellt/definiert werden.

3.4.1 Anforderungen an Testdaten[5]

- Testfälle und Testdaten sind eng miteinander verbunden. Um zu verhindern, dass der Test unbeabsichtigte Änderungen durchführt, sollten sie daher möglichst exakt aufeinander abgestimmt werden. Wichtiges Hilfsmittel ist eine genaue Dokumentation über diese Verbindung.
- Die Testdaten müssen so gewählt werden, dass sie nicht nur den Anforderungen der Testfallspezifikation genügen, sondern auch untereinander datenlogisch konsistent sind (z.B. für ein Versandsystem sollten die Bestell- und Rechnungsdaten übereinstimmen). Diese Konsistenz muss ggf. auch über mehrere Plattformen hinweg gegeben sein.
- Die Anzahl an Testdaten muss so gewählt werden, dass ausreichend viele vorhanden sind, aber nicht mehr als nötig. Dies erleichtert auch die Kontrolle der Tests, vor allem wenn diese manuell erfolgt.
- Sollen die Tests mehrmals ausgeführt werden, muss es möglich sein die Testdaten wieder in den Zustand zurückzuversetzen, den sie zu Beginn des Testvorgangs hatten.
- Ändern sich die Datenstrukturen im Programm, müssen auch die Testdaten an diese neuen Bedingungen angepasst werden (z.B. bei Regressionstests).

Abhängig vom Projekt kann der Umfang, den die Sammlung an Testdaten annimmt, sehr schnell sehr groß werden. Daher sollte auch die Verwaltung dieser Daten berücksichtigt und entsprechend geplant werden. Dies ist umso wichtiger, falls datengetriebenes Testen (strikte Trennung von Testdaten und Testausführung) durchgeführt werden soll. Auch bei der Durchführung von Leistungstests kann es vorkommen, dass sehr viele oder sogar alle Testdaten benötigt werden.

Um praxisnäher zu testen können auch Daten aus dem realen Einsatzgebiet eines ähnlichen Systems oder der bereits im Produktivbetrieb befindlichen Vorgängerversion

[5] https://de.wikipedia.org/wiki/Testumgebung, Zuletzt besucht: 24-02-2013

des selben Programms verwendet werden. Dieser Ansatz ist jedoch mit einer Reihe von Problemen behaftet:

- Die Gewinnung solcher Daten kann mitunter sehr aufwändig sein, vor allem wenn sie weder Kunden noch Entwicklern selbst zur Verfügung stehen, sondern von Dritten bezogen werden müssen.
- Es gibt keine Garantie, dass alle möglichen Testfälle mit solchen Daten abgedeckt werden können.
- Der Aufwand, der für die Ergebniskontrolle erforderlich ist, ist in diesem Fall höher. Dies ist vor allem dem Umstand geschuldet, dass bereits die Eingabedaten stark vom Zufall abhängen und daher die Ergebnisse schwerer vorherbestimmbar sind.
- Da es sich um reale Daten handelt, müssen sie oft anonymisiert werden. Auch dieser Vorgang kann sehr aufwändig sein.

3.4.2 Testdaten für White-Box Tests

Die Testdaten, die für White-Box Tests herangezogen werden, müssen weitere Anforderungen erfüllen. Soll etwa eine bestimmte Testabdeckung erreicht werden, muss die Auswahl der Testdaten sicherstellen, dass ausreichend Anweisungen, Bedingungen oder Pfade in den Tests ausgeführt werden [24].

Eine wichtige Quelle zur Festlegung der Beispieldaten für die Tests kann die Dokumentation des Projekts sein. Vor allem die Designdokumente eignen sich, denn diese legen oft die Namen, Definitionen und Strukturen der Datenelemente fest. Der Fokus auf solche Dokumente ergibt sich aus der Tatsache, dass diese, abgesehen vom Sourcecode selbst, relativ genaue Informationen über die interne Funktionsweise des Programms enthalten. Mögliche Quellen, die sich zur Ableitung von Testdaten eignen, werden in Dustin et al. [24, S. 321f] aufgelistet:

- Flussdiagramme (zyklomatische Komplexität[6])
- Datenmodelle
- Werkzeuge zur Programmanalyse
- Designdokumente wie z.B. Strukturdiagramme, Entscheidungstabellen und Aktionsdiagramme
- Detaillierte Funktions- und Systemspezifikationen
- Datenflussdiagramme

[6] `https://de.wikipedia.org/wiki/Zyklomatische_Komplexität`, Zuletzt besucht: 24-02-2013

- Data Dictionaries[7], die Datenstrukturen, Datenmodelle, Bearbeitungskriterien, Bereiche und Domänen enthalten

- Detaillierte Designs, die Datenfelder, Netzwerk, Speicherzuweisung, Daten- bzw. Programmstruktur und Entscheidungsendpunkte angeben

3.4.3 Testdatengeneratoren

Testdaten können einfach (z.B. Zahlenwerte), aber auch sehr komplex sein (z.B. Objekte) und auch die Auswahl einer großen Anzahl an Daten könnte sich als Problem darstellen. Daher entwickelt sich der Prozess der Auswahl geeigneter Werte für sinnvolle Test als sehr aufwändig. Um diesen zu erleichtern wurden eigene Testdatengeneratoren entwickelt. Diese führen eine automatische Datenauswahl durch oder erstellen selbst Testdaten nach dem Zufallsprinzip. Dieser Ansatz wird vor allem für Last- und Stresstests verwendet [8].

Die Regeln, die dazu verwendet werden, werden aus der Spezifikation des Programms oder der Datenbank bezogen. Meist ist der Generator selbst in der Lage diesen Vorgang automatisiert durchzuführen, allerdings kann es trotzdem erforderlich sein, dass manuelle Anpassungen an konkrete Gegebenheiten vorgenommen werden müssen [24].

Häufig werden sie in Kombination mit Testfallgeneratoren eingesetzt. Die von diesen erzeugten logischen Testfälle werden dadurch mit Daten angereichert und in konkrete Testfälle überführt [8].

3.5 Testen und Model Engineering

Dieser Abschnitt beschreibt die Schnittstellen zwischen Model Engineering und dem Testen von Software. Modelle dienen nicht nur zur Visualisierung des zu entwickelnden Systems und zur Unterstützung der Dokumentation des Projekts, sondern werden in den Entwicklungsprozess miteinbezogen. Dadurch dass einige Modellierungssprachen (wie z.B. UML oder Ecore) sehr detaillierte Abbildungen von Software ermöglichen, können sie auch zur Ableitung von Sourcecode herangezogen werden. Ebenso können sie dazu dienen, Testfälle oder Testdaten zu generieren. Diese Konzepte werden in der Folge näher vorgestellt.

Die Qualität von Zielmodellen zur (teil-)automatisierten Ableitung von Code bzw. Tests ist auch von der Qualität der Modelle, die dazu herangezogen werden, abhängig. Dazu ist es erforderlich, auch diese selbst einer Prüfung zu unterziehen. Welche dies in der Regel sind und welche Techniken dazu angewendet werden, wird am Ende dieses Abschnitts erwähnt.

3.5.1 Modellbasiertes Testen

Die Grundkonzepte zum Bereich *Model Engineering* werden im Kapitel 2 näher erläutert. Prinzipiell ist es möglich aus Modellen Sourcecode erzeugen zu lassen. Dieser kann für

[7] `https://de.wikipedia.org/wiki/Data-Dictionary`, Zuletzt besucht: 24-02-2013

gut modellierte, einfache Komponenten bereits vollständig sein, sodass er ohne weitere manuelle Änderungen im System verwendet werden. Werden relativ viele derartige Komponenten erzeugt, kann sich auch der Testaufwand reduzieren. Tests müssen dann nur mehr exemplarisch ausgeführt werden. Auch der Aufwand zur Fehlerbehebung kann - sofern es sich um einen Spezifikationsfehler im Modell handelt, der sich auf den erzeugten Code auswirkt - reduziert werden.

Allerdings kann sich die Fehlerbehebung auch als schwierig erweisen, wenn ein komplexer Generator verwendet wird und dieser für auftretende Fehler verantwortlich ist. Dies zeigt auch, dass sich Fehlerquellen durchaus außerhalb des Sourcecodes befinden können. Die weitere Vorgehensweise zum Testen von Software (wie z.B. Integrations- und Systemtests) sowie die Analyse diverser Metriken (Testabdeckung) stellt sich, wie in den vorangegangenen Abschnitten beschrieben, dar. Die Verbindung von Softwareentwicklung und -tests ist in diesem Bereich daher kaum von der eines herkömmlichen Projekts zu unterscheiden.

Eine engere Verknüpfung zwischen *Model Engineering* und *Testen* besteht beim modellbasierten Testen (*Model Based Testing (MBT)*). Hier werden Modelle genutzt um eine (Teil-) Automatisierung der Testerstellung und der Testdurchführung zu erreichen. Dadurch sollen die folgenden Ziele verfolgt werden [59, 82, 85, 91, 96, 101]:

- Die Testqualität soll unabhängiger von einzelnen Personen werden.
- Die Wirtschaftlichkeit des Testprozesses soll erhöht werden.
- Der Entstehungsprozess von Testfällen soll transparenter werden und leichter zu steuern sein.

Oft enthält die Dokumentation eines Software-Projekts bereits bestimmte Modelle, die in weiterer Folge zur Durchführung von modellbasiertem Testen herangezogen werden können. Auch kann die Beziehung zwischen Modellen und dem Testprozess für unterschiedliche Projekte unterschiedlich eng sein.

3.5.1.1 Verwendete Modelle & MBT-Ausprägungen

Aus der Vielzahl an Modellen, die im Rahmen eines Software-Projektes entstehen (können), sind besonders die folgenden für MBT von Interesse:

- *Umgebungsmodelle*
  Ihr Fokus liegt auf der Darstellung des Systems und dessen direkter Umgebung, sowie der Beschreibung der Anforderungen, die das System in dieser Umgebung erfüllen muss, und unter welchen Grenzbedingungen es noch einsatzfähig sein muss [82].
- *Systemmodelle*
  Sie beschreiben die Anforderungen an das System selbst, also insbesondere auf dessen statische und dynamische Eigenschaften [82].

- *Testmodelle*
  Sie werden aus den Umgebungs- und Systemmodellen abgeleitet. Ihr Fokus liegt auf den Tests, die später aus diesen Modellen generiert beziehungsweise die basierend auf diesen Modellen durchgeführt werden sollen.. Dazu werden sie oft mit testspezifischen Informationen angereichert [82].

Abhängig davon, welche dieser Modellkategorien eingesetzt werden, und wie Modelle im Testprozess integriert werden, kann modellbasiertes Testen in unterschiedlichen Ausprägungen durchgeführt werden.

- *Modellorientiertes Testen*
  Modelle dienen als Grundlage für das Testdesign. In diesem Fall müssen allerdings nicht zwingend Generatoren zur Anwendung kommen [82].
- *Modellgetriebenes Testen*
  In diesem Fall werden Generatoren angewandt um Testfälle/Tests aus dem Modell zu erzeugen. Hier besteht auch die Möglichkeit, die Ergebnisse der Testdurchführung im Modell zu ergänzen, auch wenn diese kaum genützt wird [82].
- *Modellzentrisches Testen*
  Das Modell enthält alle relevanten Testinformationen und steht im Zentrum des Testprozesses. Alle eingesetzten Werkzeuge werden zu einem Ring verbunden und auf Basis dieses Modells eingesetzt, auf das bzw. dessen Informationen sie immer wieder zugreifen [82].

3.5.1.2 Einsatzszenarien von MBT

Abhängig vom Projekt können Modelle zu einer Verbesserung der Qualität und der Effizienz des Testprozesses beitragen. Dabei spielen auch deren Art und Anzahl eine wichtige Rolle. Im folgenden werden drei mögliche Einsatzszenarien mit zunehmendem Grad an Automatisierung vorgestellt.

- Modellierung von Testfällen
  Testfälle beschreiben die konkret durchzuführenden Tests auf einer eher abstrakten Ebene. Oft liegen diese Testfälle allerdings nur in textueller Beschreibung vor, wodurch nicht immer klar ist, wie diese zu interpretieren sind. Durch deren Visualisierung in Form eines Modells verspricht man sich in dieser Hinsicht mehr Klarheit, aber auch in Bezug auf die Zusammenhänge im System. Auch Widersprüche sollen so einfacher aufgedeckt werden können. Im Idealfall enthalten die so gewonnen Modelle bereits Informationen über die zu verwendenden Testdaten. Alternativ können die Testfälle auch direkt (ohne den Zwischenschritt der textuellen Beschreibung) modelliert werden (siehe Abbildung 3.7) [41, 81, 93, 98].
- Generierung von Testfällen aus Modellen
  Aus Modellen können auch konkrete Testfälle generiert werden. Bei diesen Modellen kann es sich um Umgebungs-, System- oder Testfallmodelle einer höheren

Abstraktionsebene handeln. Diese müssen allerdings gewisse Informationen über das gewünschte Verhalten sowie detaillierte Regeln für die Ableitung der Testfälle enthalten. Nur so ist eine Automatisierung dieses Vorgangs möglich (siehe Abbildung 3.8) [40, 81].

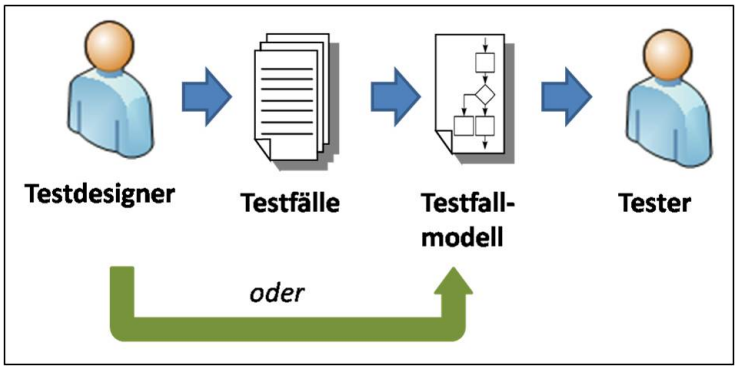

Abbildung 3.7: Modellierung von Testfällen

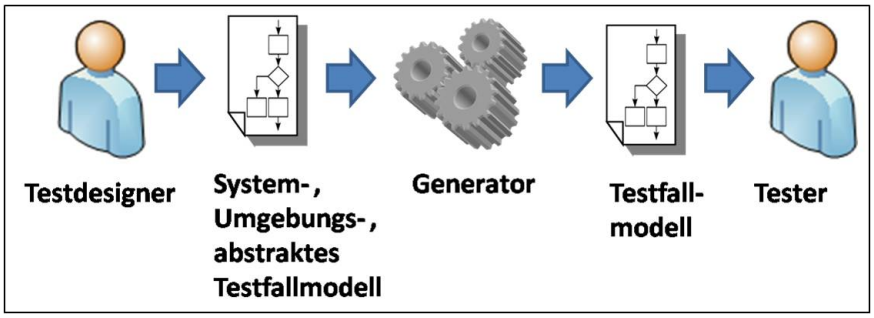

Abbildung 3.8: Generierung von Testfällen aus Modellen

- Testautomatisierung mit modellbasiertem Testen
  Werden Testfälle nicht nur erzeugt (egal ob manuell oder automatisiert) sondern auch *ausgeführt*, spricht man von Testautomatisierung. Für die Testdurchführung wird hier ein sogenannter "Testroboter" verwendet, der mit dem System, das getestet werden soll, verbunden ist (siehe Abbildung 3.9) [81, 93].

Für diese Einsatzszenarien stehen unterschiedliche Werkzeuge zur Verfügung. Neben *Testfällen* können bestimmte Werkzeuge auch *Testdaten* aus Modellen erzeugen. Soll beim Testen ein White-Box Ansatz verfolgt werden, sollten diese Werkzeuge in der Lage sein, bei der Erzeugung der Testfälle noch ein weiteres Kriterium zu berücksichtigen: die Testabdeckung (siehe dazu auch Unterabschnitt 3.1.1.3) [59, 82, 85, 91, 96, 101].

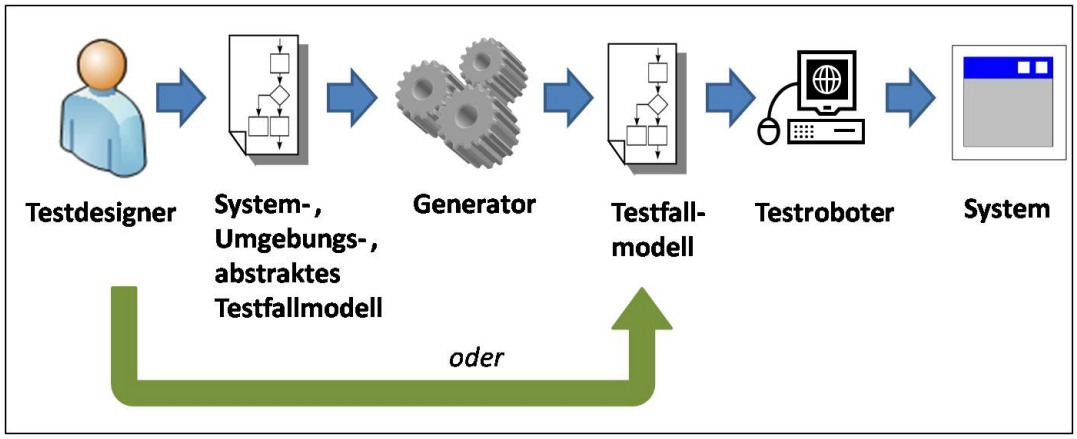

Abbildung 3.9: Testautomatisierung mit modellbasiertem Testen

3.5.2 Techniken zur Ableitung von Testfällen

Die Effektivität von modellbasiertem Testen MBT beruht hauptsächlich auf dem Ausmaß an Automatisierung das es ermöglicht. Kann das Modell automatisiert verarbeitet werden, gibt es diverse Strategien, wie daraus Testfälle abgeleitet werden können. Eine weitere Voraussetzung dafür ist, dass das Modell auch das Verhalten in korrekter Weise abbildet [14, 96, 101].

Das System kann als *endlicher Zustandsautomat* dargestellt werden. Testfälle können dann aus den möglichen Pfaden vom Start- zum Endzustand abgeleitet werden. Diese Methode kann aber nur dann angewandt werden, wenn ein deterministisches Modell zugrunde liegt [14, 84, 96, 101].

Eine weitere Technik, mit der Testfälle erzeugt werden können, ist die des *Theorem proving*. Dazu wird das Verhalten des Systems mittels logischer Ausdrücke abgebildet. Diese werden dann in Äquivalenzklassen eingeteilt, wobei jede Klasse ein bestimmtes Systemverhalten beschreibt und daher als Testfall aufgefasst werden kann [14, 15, 96, 101].

Zur Auswahl von Testfällen kann auch *Constraintprogrammierung* [46] verwendet werden. Hier wird das System in Form von Bedingungen (Constraints) dargestellt. Diese Constraints werden anschließend mit Hilfe bestimmter Programme gelöst. Die Ergebnisse dienen dann als Testfälle [14, 22, 96, 101].

Eine andere Möglichkeit Testfälle aus Modellen abzuleiten nutzt die Technik des *Model checking* [18]. Grundsätzlich wird bei diesem Ansatz geprüft, ob eine bestimmte Eigenschaft einer Spezifikation im Modell erfüllt ist. Das Programm kann während des Prüfvorgangs sogenannte “Zeugen” (Pfade in der Modellausführung, die die Eigenschaft erfüllen) und “Gegenbeispiele” (Pfade, die die Eigenschaft verletzen) identifizieren. Diese dienen dann als Testfälle [14, 30, 96, 101].

Testmodelle können auch mit Hilfe von *Markow-Ketten* [69] realisiert werden. In diesem Fall spricht man auch von *Usage/Statistical Model Based Testing*. Die Markow-Ketten werden dabei aus zwei Bestandteilen erzeugt: Einem endlichen Zustandsautoma-

ten, der alle möglichen Verwendungsszenarien des Systems darstellt, und “Operational Profiles”, die den Zustandsautomaten auf die - statistisch betrachtet - wahrscheinlichsten Möglichkeiten einschränkt. Die erste Komponente beschreibt also, was getestet werden kann, während die zweite zur Ableitung der operativen Testfälle herangezogen wird. Dieser Ansatz wird dazu verwendet, die Zuverlässigkeit des Systems bzgl. der am häufigsten zu erwartenden Verwendungen zu erhöhen [14, 56, 96, 101].

3.5.2.1 Vor- und Nachteile von MBT

Vorteile:

- Ein Ziel von MBT ist, dass die Testqualität unabhängig von einzelnen Personen ist. Da zur Ableitung von Testfällen/Tests oft genau definierte Prozesse herangezogen werden - die meist auch automatisiert ausgeführt werden - wird eine gleichbleibende Qualität gewährleistet. Die Qualität der Tests ist hier nicht der Schwankungsbreite unterworfen, die manuell geschriebene Tests aufweisen können, da sie nicht von den Erfahrungen und Fähigkeiten der Personen abhängig sind, die sie schreiben.

- Die (Teil-) Automatisierung kann auch wesentliche Einsparungen mit sich bringen. Diese machen sich vor allem durch eine Reduktion des Personal- und Zeitaufwands bemerkbar, was die Wirtschaftlichkeit des Testprozesses erhöht.
  Das trifft vor allem bei Änderungen in den Spezifikationen oder des Designs des zu entwickelnden Programms zu. Diese müssen im Modell nur entsprechend berücksichtigt werden um daraus wiederum Testfälle/Tests generieren zu können. So werden Änderungen nur an einer zentralen Stelle vorgenommen und es müssen nicht viele Tests manuell angepasst werden.

Nachteile:

- Der Aufwand zur Erstellung der Modelle kann recht hoch sein, vor allem weil ein gewisser Ausgleich zwischen Abstraktion (zur besseren Darstellung) und Detaillierung (ausreichend Informationen zur Erstellung der Testfälle) gefunden werden muss.

- Werden Testfälle aus Modellen erzeugt, kann deren Anzahl bei einem sehr komplexen Modell sehr schnell sehr hoch werden. Wird dieser Vorgang automatisiert ausgeführt, ist nicht auszuschließen, dass viele ähnliche Testfälle erzeugt werden und die hier erzielten Einsparungen an Ressourcen (vor allem Zeit) im weiteren Verlauf des Testprozesses wieder verloren gehen.

- Es können nicht immer alle Aspekte eines Systems modelliert werden. Das kann im schlimmsten Fall dazu führen, dass wichtige funktionale oder nicht-funktionale Anforderungen nicht (ausreichend) getestet werden. Dies ist umso wahrscheinlicher, je höher der Grad an Automatisierung ist.

3.5.3 Testen im Model Engineering

Zwei wichtige Begriffe im Zusammenhang mit dem Testen von Modellen sind *Verifikation* (auch *Verifizierung*) und *Validierung*.

Verifikation bezeichnet den *"Nachweis, dass ein behaupteter oder vermuteter Sachverhalt wahr ist"*[8]. Um diesen Nachweis zu erbringen gibt es formale Mechanismen (z.B. mathematische Beweisführung), die zum Teil auch automatisch ausgeführt werden können.

Validierung steht für die Möglichkeit, anhand genauer Vorgaben ein bestimmtes Ergebnis immer wieder erzielen zu können [3]. Je detaillierter dabei die Vorgehensweise beschrieben ist, umso wahrscheinlicher ist es wiederholt zu denselben Resultaten zu gelangen.

Diese beiden Begriffe sind auch im Software Engineering im Rahmen der Softwarequalitätssicherung von Bedeutung: Mittels Verifikation soll sichergestellt werden, dass ein System zu seiner Spezifikation konform ist, wogegen Validierung[9] sicherstellen soll, dass ein System die Anforderungen in der Praxis erfüllt. Bei beiden handelt es sich um unabhängige Prozesse, die aber oft gemeinsam eingesetzt werden. Das führt zum Beispiel im Model Engineering dazu, dass es nicht immer einfach ist, sie voneinander abzugrenzen.

Das Testen von Modellen kann, abhängig davon, ob ein Metamodell definiert wird oder nicht, auf zwei verschiedenen Ebenen erfolgen. Zunächst wird oft ein Metamodell in einer bestimmten Modellierungssprache (z.B. UML, Ecore) entwickelt. Wie alle Sprachen haben auch diese bestimmte syntaktische Regeln zu befolgen. Diese können in der Regel von eigenen Programmen geprüft werden. Somit kann sichergestellt werden, dass diese Metamodelle syntaktisch korrekt sind. Oft wird dies von Modellierungswerkzeugen bereits bei der Erstellung geprüft, sodass es zum Beispiel gar nicht möglich ist, Metamodelle zu erstellen, die die Syntax der Modellierungssprache verletzen beziehungsweise wird dieser Umstand sofort angezeigt.

Im nächsten Schritt werden Modelle erstellt, die auf diesen Metamodellen basieren. Mit Hilfe von verschiedensten Werkzeugen kann nun überprüft werden, ob diese Modelle syntaktisch korrekt sind, das heißt ob sie die im Metamodell definierten Anforderungen erfüllen und Einschränkungen nicht verletzen. Ist ein Modell eine gültige Instanz, spricht man auch davon, dass es "valide" ist.

Eine wichtige Grundregel bei der Erstellung von Metamodellen ist, dass sie nicht zu viele Einschränkungen enthalten sollen. Da sie darüber hinaus auf einer abstrakten Ebene erstellt werden, ist es nicht immer möglich alle Einschränkungen, die die später daraus abgeleiteten Modelle aus verschiedensten Gründen erfüllen müssen, zu formulieren beziehungsweise modellieren. Daher werden vor allem im Model Engineering die Modellierungssprachen durch Spezifikationssprachen (wie z.B. OCL) ergänzt. Diese ermöglichen die Definition von zusätzlichen Bedingungen, die das Modell erfüllen muss, die aber so nicht mit der Modellierungssprache definiert werden können.

[8] `https://de.wikipedia.org/wiki/Verifizierung`, Zuletzt besucht: 18-03-2013

[9] `https://de.wikipedia.org/wiki/Validierung_(Informatik)`, Zuletzt besucht: 18-03-2013

Im Wesentlichen handelt es sich bei der Überprüfung, ob ein Modell eine gültige Instanz eines Metamodells ist, um einen Prozess der Validierung. Hier wird die Validierung als Plausibilitätstest eingesetzt. Dabei soll allgemein sichergestellt werden, dass ein Wert auch in seinem gültigen Wertebereich liegt beziehungsweise dass er zu einem bestimmten Datentyp gehört. Vor allem ersteres wird bei der Modellerstellung durch zusätzliche Bedingungen sichergestellt, da dies oftmals nicht direkt modellierbar ist.

Es gibt aber Möglichkeiten zur Verifikation von (Meta)modellen. Dabei werden zum Beispiel die - bereits im vorhergehenden Abschnitt erwähnten - Techniken des *Model Checking* und der *Constraintprogrammierung* eingesetzt. Ein Werkzeug, das die zweite Technik einsetzt um eine gültige Instanz eines Metamodells zu erzeugen, ist das Tool *EMFtoCSP*[10] (siehe auch Unterabschnitt 8.1.2.1). Hier wird ausgehend von einem Metamodell geprüft, ob es möglich ist eine gültige Instanz zu finden oder nicht. So kann festgestellt werden, ob das Metamodell Fehler enthält, die es unmöglich machen ein gültiges Modell daraus abzuleiten. Außerdem werden bei diesem Ansatz auch zusätzliche Bedingungen geprüft, wodurch in einem gewissen Ausmaß auch eine Validierung der erzeugten Instanz durchgeführt wird.

3.6 Testen von Modelltransformationen

Neben dem Generieren von Text (Code) aus Modellen ist die Transformation von Modellen ein weiterer großer Themenbereich im Model Engineering. Diese Modelltransformationen sind von ihrer Konzeption her mit dem Black-Box Ansatz aus dem Software Engineering zu vergleichen. Es ist das Quellmodell sowie das Ausgabemodell bekannt, allerdings wenig bis gar nichts darüber, was im Transformationstool passiert. Es gibt bereits einige Ansätze, die sich damit beschäftigen, diese Black-Box transparenter zu machen. Darüber hinaus gibt es beim Testen von Modelltransformationen eigene Herausforderungen und Probleme zu beachten.

Wie in allen anderen Bereichen des Model Engineering können auch bei der Entwicklung von Modelltransformationen Fehler auftreten. Laut Küster [55] gibt es sechs verschiedene Arten von Fehlern, die bei der Programmierung von Transformationsregeln auftreten können:

1. Die Regeln werden programmiert, ohne dass das gesamte Metamodell berücksichtigt wird. Dadurch kann es vorkommen, dass bestimmte Elemente des Quellmodells nicht verarbeitet werden können.

2. Es werden Modelle mit syntaktischen Fehlern erzeugt (z.B. sie sind nicht konform zum Zielmetamodell oder sie verletzen Constraints).

3. Wird die Transformationsregel für ein Modell aufgerufen, für das sie nicht entwickelt wurde, kann es vorkommen, dass das Ergebnis zwar syntaktisch, aber nicht semantisch korrekt ist.

[10] `https://code.google.com/a/eclipselabs.org/p/emftocsp/`, Zuletzt besucht: 12-03-2013

4. Die Transformation liefert verschiedene Ausgaben für dasselbe Modell. Der Grund dafür ist fehlende Konfluenz. Das kann auch dazu führen, dass Zwischenmodelle erzeugt, aber nicht weiter verarbeitet werden können.

5. Die Transformation erfüllt nicht alle Anforderungen, die an sie gestellt werden (z.B. es besteht die Möglichkeit eines Deadlock).

6. Wie überall, wo Code erzeugt wird, kann es auch bei der Erzeugung von Modelltransformationen zu klassischen Programmierfehlern kommen.

Um zu verhindern, dass diese Fehler sich auf die Zielmodelle - und darüber hinaus - auswirken, ist es notwendig, die Transformationen zu testen. Dies ist umso wichtiger, als Modelltransformationen nicht nur einmal sondern wiederholt ausgeführt werden sollen. Da sich die Quellmodelle nicht immer gleichen, kann es durchaus auch vorkommen, dass eine Transformation für ein Modell ein korrektes Ergebnis liefert, für ein anderes Modell aber eine fehlerhafte Ausgabe erzeugt [5].

Bei Baudry et al. [5] werden drei wichtige Aufgaben im Testprozess vorgestellt:

1. Erzeugen von Testdaten
   Ein wichtiger Punkt im Testprozess ist die Bereitstellung von Testdaten. Im konkreten Fall ist es erforderlich, Quellmodelle (auch Testmodelle genannt) zur Verfügung zu stellen. Eine Mindestanforderung ist die Konformität zum Metamodell.

2. Definition von Eignungskriterien
   Nicht alle möglichen Testmodelle können getestet werden. Daher müssen bestimmte Kriterien definiert werden, die die Eignung von Quellmodellen hinsichtlich bestimmter Eigenschaften überprüfen. Dadurch soll die Anzahl an zu testenden Modellen verringert werden ohne die Qualität des Testvorgangs zu beeinträchtigen.

3. Konstruieren eines *Orakels*

   Bei einem *Orakel* handelt es sich um einen Mechanismus aus dem Software Engineering, der das Ergebnis von Tests auf ihre Korrektheit prüft. Ist das Ergebnis eines Tests bekannt, wird das Orakel dazu verwendet, das erwartete mit dem tatsächlichen Ergebnis zu vergleichen [10]. Oft liegt aber kein bestimmtes Zielmodell vor. In diesem Fall wird das Orakel eingesetzt um anhand der vorhandenen Eingabedaten alle möglichen Ergebnisse zu ermitteln. Diese werden dann mit dem tatsächlichen Ergebnis verglichen.

Allerdings gibt es beim Testen von Modelltransformationen auch einige spezifische Herausforderungen zu beachten, um Fehler zu identifizieren und sie beheben zu können. Diese leiten sich aus den Eigenschaften von Modelltransformationen ab [5].

Die Eingabe- und Ausgabedaten sind beim Testen von Modelltransformationen die Modelle. Diese müssen zu ihren Metamodellen konform sein. Allerdings können sowohl die Modelle als auch die Metamodelle sehr groß und komplex werden. Darüber hinaus

wird oft gefordert, dass die Modelle zusätzliche Bedingungen erfüllen. Um diese zu formulieren werden oft eigene Sprachen (z.B. OCL) verwendet. Diese Bedingungen können selbst sehr hohe Komplexität aufweisen und über eine größere Anzahl an Elementen aus dem Metamodell definiert sein [5].

Nicht immer stehen bereits (ausgereifte) Frameworks zum Testen von Modelltransformationen oder zum Erzeugen und Ändern, sowie zur Visualisierung und Analyse von Modellen zur Verfügung. Das hat auch Auswirkungen auf den Testprozess, denn die fehlende Tool-Unterstützung begünstigt auch Fehler bei der Erzeugung von Testmodellen unabhängig davon, ob diese nun manuell oder automatisch erzeugt werden [5].

Die fehlende Unterstützung durch geeignete Werkzeuge kann auch eine andere wichtige Herausforderung, die sich aus dem Testen von Software ableitet, mit sich bringen: Die spätere Produktivumgebung der Transformation kann nur unzureichend oder gar nicht simuliert werden [55].

Eine weitere Herausforderung stellt die Vielzahl an Möglichkeiten an Sprachen und Techniken zur Realisierung von Modelltransformationen dar. Transformationen können in Programmiersprachen wie zum Beispiel Java oder in eigens zu diesem Zweck entwickelten Sprachen wie QVT oder ATL entwickelt werden. Dies hat zur Folge, dass es für bestimmte Modelltransformationen verschiedenste - oder auch keine - Testtechniken geben kann. Eine weitere Folge ist, dass es im Falle von White-Box Tests schwierig bis unmöglich ist, allgemeine Eignungskriterien zu definieren [5].

Mittlerweile gibt es verschiedenste Ansätze, die sich mit dem Testen von Modelltransformationen beschäftigen und dabei auch die hier erwähnten Herausforderungen beachten.

3.6.1 Ansätze

Bei Schönböck [86] wird eine Klassifizierung von verschiedenen Techniken zum Testen von Transformationen vorgestellt, die auch in Abbildung 3.10 in leicht vereinfachter Form dargestellt wird. Die wesentlichste Unterscheidung erfolgt dabei, ob der Fokus auf der Erzeugung von Quellmodellen oder auf der Vorhersage von Ergebnissen liegt.

3.6.1.1 Erzeugen von Quellmodellen

Liegt der Fokus auf der Generierung von Quellmodellen, so können diese aus dem Metamodell abgeleitet werden. Aufgrund der Tatsache, dass Metamodelle selbst nur sehr grundlegende Einschränkungen enthalten, kann eine große Anzahl an möglichen Testinstanzen erzeugt werden. So kann - indirekt - ein hoher Grad an Testabdeckung erreicht werden [86].

Alternativ kann auch die Transformation selbst zur Generierung der Testinstanzen herangezogen werden. Dieser Ansatz ist wesentlich komplexer, hat aber auch den Vorteil, dass bestimmte Aspekte der Transformation gezielt getestet werden können [86].

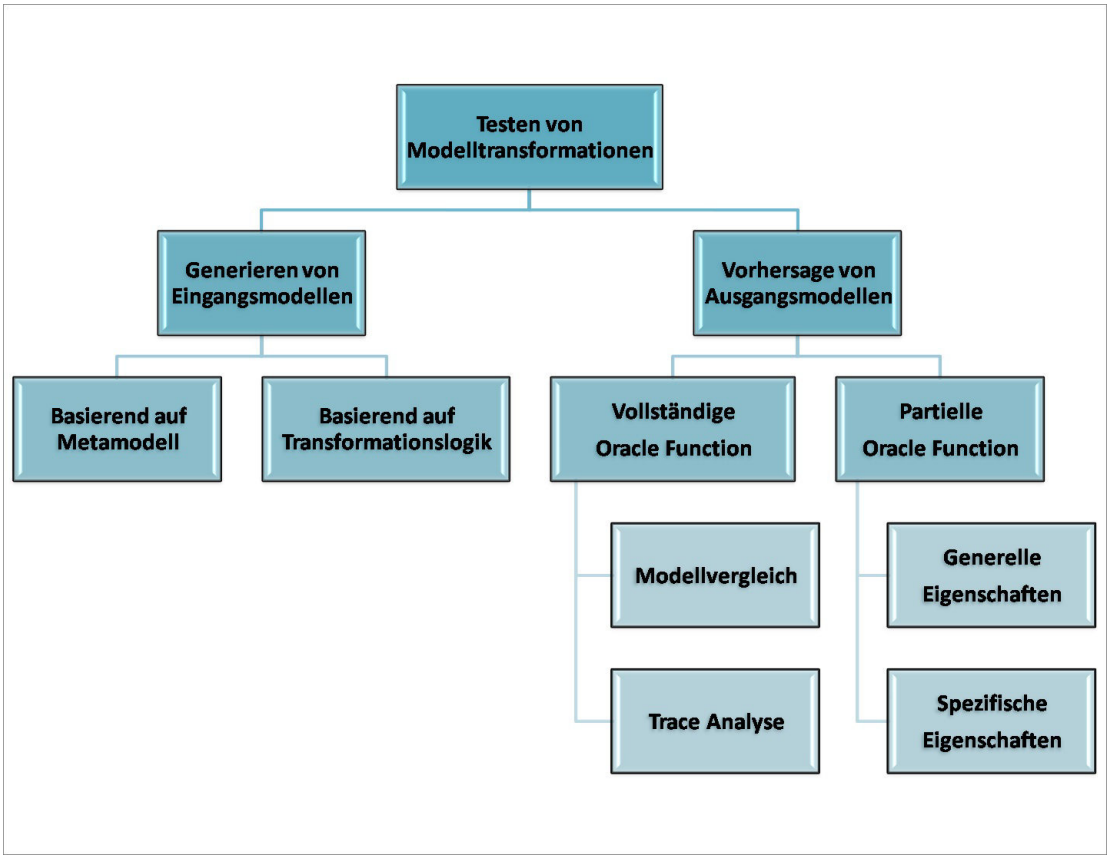

Abbildung 3.10: Einteilung der Ansätze, die zum Testen von Modelltransformationen entwickelt werden [86]

3.6.1.2 Vollständige Orakelfunktionen

Die zweite große Gruppe von Ansätzen baut auf dem Einsatz von *Orakeln* auf. Hier liegt der Fokus auf der Prognose der Zielmodelle. Die Ergebnisse der Modelltransformation können ganz oder in Teilen vorhergesagt werden. Erstere Möglichkeit wird häufig für kleinere oder Modelle von geringer Komplexität angewandt [86].

Steht ein Referenzergebnis zur Verfügung, kann dieses mittels Modellvergleich mit dem Ergebnis der Transformationsausführung verglichen werden. Stimmen sie nicht überein, liegt ein Fehler vor. Anhand des resultierenden Differenzmodells soll nun festgestellt werden, welche Teile der Transformation für diese Unterschiede verantwortlich sind [86].

Eine zweite Möglichkeit aus dieser Gruppe von Ansätzen ist die *Trace Analysis*. Dabei werden die Quell- und Zielmodelle per *Trace Links* miteinander verknüpft. Diese werden in einem eigenen Modell dargestellt, das für eine bestimmte Transformation mit einem existierenden, beispielhaften Modell verglichen werden. Diese *Trace Links* können direkt

auf potenzielle Fehlerquellen verweisen [86].

Modellvergleich und Trace Analyse erfordern die Ausführung der Transformation, was einige Vorteile bietet [86]:

- der Testprozess ist einfach durchzuführen
- die Transformation wird in der Umgebung getestet, in der sie später eingesetzt werden soll
- der Testprozess kann automatisiert werden

Diese beiden Ansätze haben allerdings das Problem, dass es mit zusätzlichem Aufwand verbunden ist, Referenzmodelle zur Verfügung zu stellen, oder dass die Komplexität der Modelle und Transformationen es schwierig macht, Fehlerquellen zu finden. Um diese Schwierigkeiten zu umgehen gibt es Ansätze, die statt des gesamten, nur Teile des Ergebnisses vorhersagen [86].

3.6.1.3 Partielle Orakelfunktionen

Ansätze, die dieser Gruppe zuzuordnen sind, fokussieren auf der Prüfung bestimmter Eigenschaften der Eingabe- und Ausgabemodelle sowie der Transformationen. Dabei kann weiter unterschieden werden, ob eher allgemeine Eigenschaften oder spezifische Eigenschaften geprüft werden sollen [86].

Unter allgemeinen Eigenschaften versteht man solche, die alle Modelltransformationen erfüllen sollen, wie etwa die Vermeidung von Deadlocks (Endlosschleifen) beziehungsweise die Sicherstellung, dass eine Transformation immer beendet wird (ob nun ein Fehler auftritt oder nicht). Im Gegensatz dazu soll mit dem Testen spezifischer Eigenschaften sichergestellt werden, dass die Transformation korrekt arbeitet [86].

Die Prüfung spezifischer Eigenschaften berücksichtigt auch sogenannte *Contracts*. Ursprünglich wurden solche *Contracts* im Software Engineering eingesetzt um objektorientierte Programme zu verifizieren [4, 86]. Dabei sollen diese "Verträge" sicherstellen, dass einzelne Programmmodule problemlos zusammenarbeiten. Um dieses Ziel zu erreichen, werden Vorbedingungen (die der Aufrufer einhalten muss), Nachbedingungen (die der Aufgerufene einhalten muss) und Invarianten (beweisen die Korrektheit des Algorithmus) definiert [64]. Diese Ideen wurden auch für Modelltransformationen übernommen [4, 86].

Die Contracts können auf verschiedene Arten eingesetzt werden. Sie können Vor- und Nachbedingungen für die Transformationsausführung definieren. Erstere dienen dazu die Quellmodelle einzuschränken und können daher auch zur Generierung von Testmodellen herangezogen werden. Nachbedingungen definieren bestimmte Eigenschaften, die im Zielmodell erfüllt sein müssen. Sie können auch Eigenschaften beschreiben, die das Quell- und Zielmodell miteinander verlinken. Somit können sie einen wichtigen Beitrag zur Evaluierung des Ergebnisses leisten [4].

Eine weitere Möglichkeit besteht darin, die Contracts innerhalb der Transformation einzusetzen wenn diese aus mehreren Sub-Transformationen besteht. So kann auch genau

definiert werden, wann eine bestimmte Sub-Transformation ausgeführt wird, und es kann getestet werden, ob diese auch die gewünschten Ergebnisse liefert. Dieses Einsatzgebiet ist dem ursprünglichen aus der Softwareentwicklung am ähnlichsten [4].

Werden Contracts auf diese beiden Arten eingesetzt, berücksichtigen (bzw. betreffen) sie sowohl die Quell- als auch die Ausgangsdaten. Es kann aber auch der Fall sein, dass nur Eigenschaften des Zielmodells von den Contracts erfasst werden. In diesem Fall sind die wichtigsten Überprüfungen, ob die Konformität des Modells zum Metamodell gegeben ist und ob das Modell auch die zusätzlich definierten Bedingungen (Constraints) erfüllt [4].

3.6.2 Testdaten/Testmodelle

Die Testdaten sind beim Testen von Modelltransformationen die Quellmodelle, mit denen die Transformation gestartet wird. Diese müssen in der Regel zwei Bedingungen erfüllen: Sie müssen zum Metamodell konform sein und zusätzlich definierte Bedingungen (Constraints) erfüllen [4, 5].

Die Testmodelle können entweder manuell erstellt oder automatisch generiert werden. Der erste Ansatz birgt das Risiko in sich, dass die erstellten Modelle viele Fehler enthalten. Das kann vor allem dann der Fall sein, wenn das zugehörige Metamodell sehr komplex ist, aber auch wenn ein komplexeres Modell (in dem viele Elemente miteinander verbunden sind oder das viele Attribute mit unterschiedlichen Werten enthält) erzeugt werden muss um bestimmte Aspekte der Transformation testen zu können [5].

Automatische Modellgeneratoren hingegen können selbst recht schnell eine unüberschaubare Komplexität entwickeln. Werden die Testmodelle auf diese Weise erzeugt, kann es für den Tester schwierig werden, diese zu interpretieren. Dieses Problem kann durch fehlende Semantik im Modell (automatisch generierte Attribute sind nicht immer sinnvoll) verschärft werden. Weitere Schwierigkeiten ergeben sich, wenn ein Ansatz zur Vorhersage des Zielmodells verfolgt wird und der Tester das korrekte Ergebnis der Transformation anhand des Quellmodells ermitteln muss (z.B. für Modellvergleiche). Aber auch wenn die Transformation scheitert, muss der Tester anhand des Quellmodells die Fehlerursache ermitteln können [4].

Um Modelltransformationen effizient testen zu können müssen die Testmodelle eine gewisse Variabilität aufweisen. Dabei sind neben der Existenz unterschiedlicher Konzepte vor allem die Multiplizitäten der Referenzen und Attribute zu berücksichtigen. Um Fehler in der Transformation aufdecken zu können, kann es durchaus sinnvoll sein Teilmodelle neben einem möglichst vollständigen Modell zu testen. Eine Möglichkeit zur automatisierten Generierung solcher Testmodelle bietet der Einsatz von *Answer Set Programming (ASP)*.

KAPITEL 4

Überblick über das *White-Box Testing Tool*[1]

In diesem Kapitel wird ein kurzer Einblick in das entwickelte WBT-Tool gegeben. Zuerst wird darauf eingegangen, wofür es eingesetzt werden kann und welche Ziele bei der Entwicklung verfolgt wurden. Danach wird ein kleiner Überblick in die Architektur des Tools gegeben. Die genaue Implementierung und technische Umsetzung findet sich dann im Kapitel 6.

4.1 Einsatzbereich

Im Software Engineering Prozess gewinnt die modellgetriebene Softwareentwicklung immer mehr an Bedeutung. Modelle und ihre Metamodelle werden in den Prozess miteinbezogen und entwickeln sich zu Kernelementen. Neben den Modellen werden auch Modelltransformationen in Prozessen benötigt, da sie für verschiedene Anwendungsfälle, wie Verfeinerungen, Abstraktionen und Optimierungen, eingesetzt werden [43]. Die Transformationen sollen im Softwareprozess fehlerfrei funktionieren um eine hohe Softwarequalität sicherstellen zu können. Deshalb ist es notwendig diese ausreichend zu testen. Eine Möglichkeit ist Testfälle manuell zu erstellen und diese durchlaufen zu lassen. Allerdings steigt der Aufwand je nach Komplexität des Systems, welches getestet werden soll und es kann passieren, dass Fehler übersehen oder erst sehr spät erkannt werden. Mit Hilfe einer Testautomatisierung kann der Aufwand für einen einzelnen Testfall reduziert und gleichzeitig die Testabdeckung erhöht werden.

Das Ziel des WBT-Tools ist es einen automatisierten Ansatz zum Testen von Modelltransformationen zur Verfügung zu stellen. In einem ersten Schritt ist es notwendig genügend Testinstanzen für die Transformation zu erstellen. Im Programm wird zur Generierung von Testmodellen aus dem Quellmetamodell ein Black-Box Ansatz verwendet.

[1] Verfasser: Sabine Wolny, BSc

Zur Erzeugung solcher Testdaten gibt es verschiedene Ansätze, wobei im WBT-Tool ein deklarativer eingesetzt wird. Dieser verwendet ein Programm aus der Familie der *Answer Set Programming (ASP) Tools*. Die Grundstruktur, Syntax und Semantik von ASP kann im Kapitel 5 nachgelesen werden. So kann eine relativ breite Palette an Testinstanzen erzeugt werden. Im weiteren Verlauf werden bei der Transformationsausführung nur die Modelle akzeptiert, die - falls gegeben - auch zusätzliche Bedingungen erfüllen. Danach werden für diese Modelle die Aktionen aufgezeichnet, welche während der Transformation ausgeführt werden um die Zielmodelle zu erzeugen. Dadurch wird der gesamte Transformationsprozess transparenter. Aus den so gewonnenen Informationen können Schlüsse über die Testabdeckung gezogen und mögliche Fehler im Prozess gefunden werden.

Da es sich bei der im WBT-Tool gewählten Transformationssprache um die regelbasierte Sprache ATL (siehe Abschnitt 6.4) handelt, kann anhand der Anzahl und Aufzeichnung der ausgeführten Regeln geprüft werden, ob alle Klassen des Quellmodells auch korrekt in Klassen des Zielmodells transformiert wurden. Dadurch wird in gewissem Maße auch das Konzept der *Trace Analysis* verwirklicht, indem überprüfbar wird, ob für alle Klassen die richtige Regel ausgeführt wurde, beziehungsweise ob im Zielmodell eine entsprechende Anzahl an korrekt generierten Klassen vorhanden ist. Da die Modellgenerierung bereits sicher stellt, dass nur gültige Modelle erzeugt werden, kann der Fehler in diesem Fall nur in der Transformation zu finden sein.

Statt einzelne Eigenschaften zu testen, wie dies beim *Partiellen Oracle-Testing* (siehe Unterabschnitt 3.6.1.3) der Fall ist, wird beim vorliegenden Programm das gesamte Modell getestet. Da allerdings eine Vielzahl an Modellen automatisch generiert wird, werden auch "fragmentierte Modelle" (Modelle, die nicht aus allen möglichen Bestandteilen bestehen) erzeugt. Durch diesen Ansatz entfällt der Aufwand zu entscheiden, welche Teilmodelle erzeugt und getestet werden sollen. Ebenso entfällt die Verschmelzung der Ergebnisse solcher Tests um ein Gesamtbild für ein vollständiges Modell zu erhalten. Statt dessen wird der dafür erforderliche Aufwand hier in die Generierung der Testmodelle investiert.

4.2 Architektur des WBT-Tools

Das WBT-Tool ist ein Programm zum Testen von *M2M Transformationen*. Eine vereinfachte Darstellung der Architektur ist in der Abbildung 4.1 zu sehen.

Das Programm ist neben der Modelltransformationssprache ATL auf die Benutzung von *Ecore* Metamodellen ausgelegt. Wenn ein Benutzer ein Quellmetamodell und die zugehörige Modelltransformation testen will, so benötigt das Tool folgende Elemente, die der Benutzer angeben muss:

- Quellmetamodell (.ecore)
- Zielmetamodell (.ecore)
- Modelltransformation (.atl)

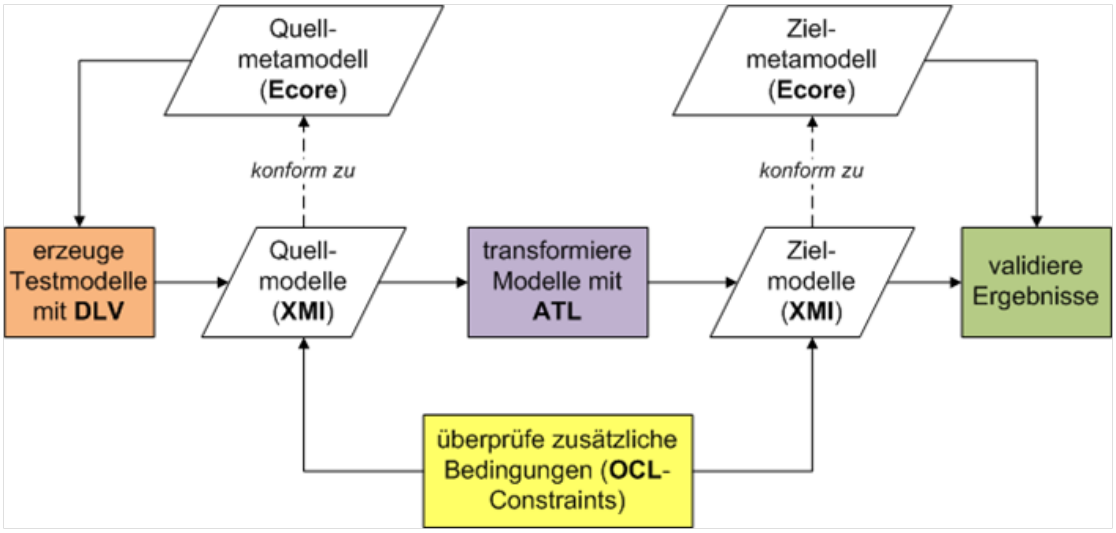

Abbildung 4.1: Grundarchitektur des WBT-Tools

Das Tool entwickelt nun aufgrund der Informationen, die im Metamodell enthalten sind, eigene Testinstanzen ohne weitere Interaktion des Benutzers. Dies geschieht mit einer Subroutine, in der das Programm DLV ausgeführt wird. DLV gehört zu den Answer Set Programming (ASP) Tools und erstellt mit Hilfe von Regeln einzelne Testfälle in Form von Quellmodellen, die konform zum gegebenen Metamodell sind.

Nachdem die Modelle erzeugt sind, wird die Modelltransformation durchgeführt. Für korrekte Transformationen werden Zielmodelle erzeugt, andernfalls werden die Fehlermeldungen aufgezeichnet. Für alle Zielmodelle wird zusätzlich überprüft, ob diese konform zu ihrem Zielmetamodell sind. Schließlich werden die validen Instanzen zurückgeliefert.

Alle für den Benutzer relevanten Informationen, wie Fehlermeldungen und ungültige Instanzen, werden von dem WBT-Tool in eine Logdatei geschrieben, sodass der Benutzer nach der Ausführung eine genauere Analyse der Modelltransformation durchführen kann.

Neben dem reinen Testen der Modelltransformation und der Überprüfung der Konformität zum Metamodell ist es möglich für die Quellmodelle zusätzliche Bedingungen zu definieren, die sie erfüllen müssen. Die Bedingungen müssen in Form von OCL-Constraints in einer eigenen Datei angegeben werden und zu Beginn in das Tool geladen werden. Die Bedingungen werden nach der Erzeugung der Quellmodelle überprüft und haben keinen Einfluss auf die Testdatengenerierung mit DLV. Modelle, die die Bedingungen nicht erfüllen, werden gelöscht und im weiteren Prozess nicht mehr beachtet. In der Logdatei wird allerdings protokolliert, welcher Constraint fehlgeschlagen ist. Auch für die Zielmodelle ist die Angabe solcher zusätzlichen Bedingungen möglich. Das Prinzip ist dasselbe wie für die Quellmodelle.
Ein Vorteil des WBT-Tools ist, dass das Programm nicht immer in vollem Umfang verwendet werden muss, sondern auch nur einzelne Komponenten genutzt werden können:

- Generierung von Modellinstanzen mit DLV
  Es kann nur die Option zu Generierung von Testdaten verwendet werden um zu einem Metamodell valide Modelle zu erstellen.

- Überprüfung von OCL-Constraints
  Es können für ein Metamodell und ein zugehöriges Modell Bedingungen überprüft werden.

- Durchführung und Test der ATL Transformation
  Die ATL Transformation kann auch nur konkret für ein bestimmtes Quellmodell durchgeführt und getestet werden.

Alle diese Komponenten können beliebig kombiniert werden, solange der Benutzer am Beginn die benötigten Dateien angibt.

Ein weiterer Vorteil für den Benutzer ist, dass das WBT-Tool nach Angabe der gewünschten Funktionen die weiteren Schritte automatisiert durchführt und daher keine Interaktion des Benutzers während des Testdurchlaufs erforderlich ist. Dadurch können auch große Test Suiten erzeugt und automatisch ausgeführt werden.

KAPITEL 5

Answer Set Programming[1]

In diesem Kapitel wird ein Einblick in Answer Set Programming (ASP) gegeben, worauf der DLV Solver aufbaut. Dieser wurde bei der Entwicklung des Testmodellgenerators (Abschnitt 6.2) verwendet. Zuerst wird allgemein auf ASP eingegangen. Dann werden die Architektur und die Methodik, sowie die wichtigsten Elemente der Syntax und Semantik erklärt. Zum Schluss wird auf verschiedene existierende Ansätze bezüglich der Anwendung von ASP eingegangen.

5.1 Einführung in ASP

> *"Answer set programming (ASP) is a form of declarative programming oriented towards difficult, primarily NP-hard, search problems. As an outgrowth of research on the use of nonmonotonic reasoning in knowledge representation, it is particularly useful in knowledge-intensive applications." [58, S. 1]*

Lifschitz beschreibt klar, worum es in ASP geht. Bei ASP handelt es sich um einen deklarativen Programmieransatz. Im Gegensatz zu imperativen Programmiersprachen wird bei deklarativen nicht das *"wie"* und der genaue Ablauf beschrieben, sondern nur was gegeben ist. Aus den Regeln wird die Lösung dann hergeleitet. Ein weiterer wichtiger Punkt, den Lifschitz aufzeigt, ist, dass ASP an NP-schweren Suchalgorithmen orientiert ist. NP steht für "non-deterministic polynomial-time". Ein Problem heißt NP-Schwer, wenn ein Algorithmus, der dieses löst, benutzt werden kann, um alle Probleme in NP zu lösen[2]. Die genauere Beschreibung von NP Problemen würde allerdings den Rahmen dieser Arbeit sprengen und kann unter anderem bei Schwinger und Koch [88] nachgelesen werden.

[1] Verfasser: Sabine Wolny, BSc

[2] `http://de.wikipedia.org/wiki/NP-Schwere`, Acessed:2013-04-10

5.2 Architektur und Methodik von ASP

Es gibt mittlerweile eine Vielzahl von verschiedenen ASP Solvern. Einer der bekanntesten ist sicherlich DLV[3], welches von der Universität von Kalabrien in Kooperation mit der Technischen Universität Wien entwickelt wurde. Außerdem muss in diesem Zusammenhang Potassco (Potsdam Answer Set Solving Collection) [33] genannt werden, welche mehrere Tools, die von der Universität von Potsdam entwickelt wurden, umfasst. Das bekannteste Tool in dieser Sammlung ist clasp[4]. Dieses Tool zusammen mit DLV sind die momentan effizientesten Solvers. Weitere Solver wären unter anderem Smodels[5], ASSAT[6] und SUP[7] (kombiniert die Ideen der Solver Smodels und Cmodels) um einige zu nennen. Die ASP Solver unterscheiden sich unter anderem dadurch, für welches Betriebssystem sie entwickelt wurden, als auch in der Syntax und in den umgesetzten Möglichkeiten. Allerdings ist die Grundarchitektur von den Solvern immer gleich. So kann diese nach Eiter et al. [27] allgemein als 2-Layer Architektur mit folgenden zwei Schritten beschrieben werden:

1. *Grounding Step*
2. *Model Search*

Der gesamte Prozess von ASP, der diese zwei Hauptschritte umfasst, wird in Gebser et al. [33] noch näher beschrieben und wird in der nachfolgenden Abbildung 5.1 ähnlich veranschaulicht.

Zuerst wird das zu lösende Problem in Regeln umgesetzt. Diese müssen von den verwendeten Solvern verstanden werden und werden daher in der Syntax des jeweiligen Programms geschrieben. Da es sich bei allen Solvern immer um deklarative Ansätze handelt, müssen die Regeln nicht in einer bestimmten Reihenfolge angegeben werden. Das macht die gesamte Umsetzung flexibler als bei imperativen Programmiersprachen. In den Problemstellungen werden fast immer Variablen und nicht nur Konstanten zum Beschreiben der gewünschten Lösung verwendet und so muss als nächster Schritt das gegebene Problem P vom Solver in den neuen Programmcode P' übergeführt werden, wo alle Regeln keine Variablen mehr haben, sondern durch alle Möglichkeiten ersetzt wurden. Zu bedenken ist, dass dieses neue Programm P' nun bis zu exponentiell viele Regeln haben kann. Damit P' trotzdem möglichst klein und leicht zu evaluieren bleibt, wurden von den verschiedenen Solvern bestimmte Bedingungen für die Regelsyntax festgelegt [27]. Bei DLV ist diese Einschränkung die Voraussetzung von *rule saftey*. "rule safety*(DLV): every variable in a rule must occur in some positive body literal (i.e., not prefixed with not) whose predicate is not "=" or any another built-in comparison predicate. This is a standard condition in the area of deductive databases." [27, S. 89]*

[3] `http://www.dlvsystem.com/`, Zuletzt besucht: 18-03-2013
[4] `http://potassco.sourceforge.net/index.html`, Zuletzt besucht: 18-03-2013
[5] `http://www.tcs.hut.fi/Software/smodels/`, Zuletzt besucht: 18-03-2013
[6] `http://assat.cs.ust.hk/`, Zuletzt besucht: 18-03-2013
[7] `http://www.cs.utexas.edu/~tag/sup/`, Zuletzt besucht: 18-03-2013

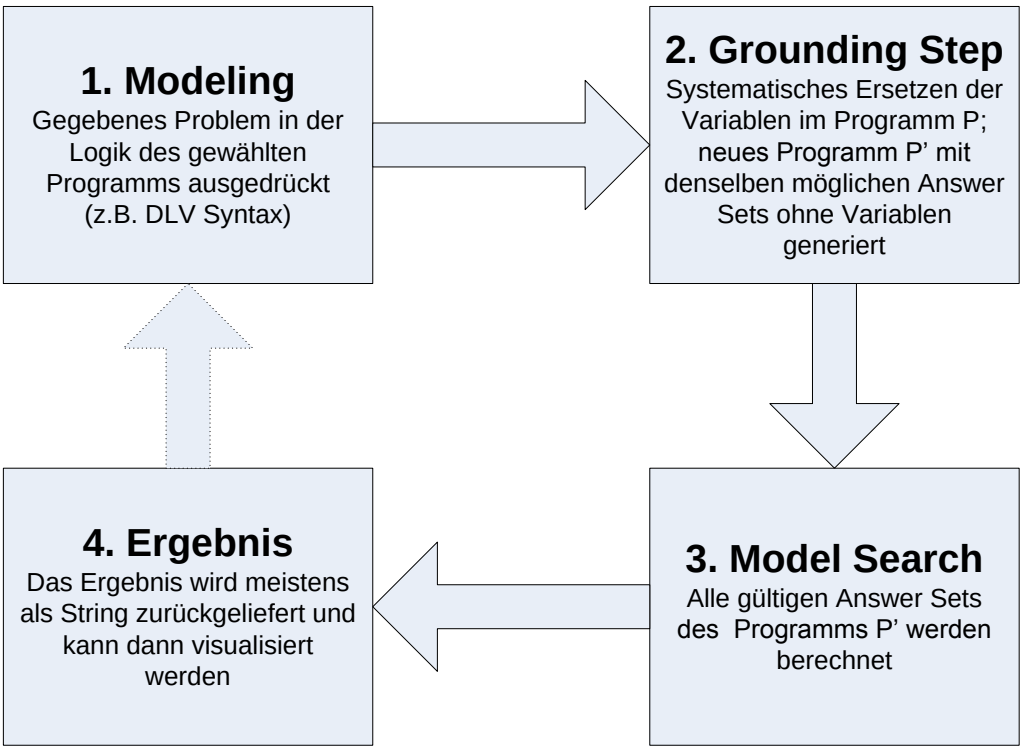

Abbildung 5.1: Prozess von ASP

Wie diese Einschränkungen für den *Grounding Step* aussehen, ist von dem jeweilig verwendeten Solver abhängig und unterscheidet sich daher bei den verschiedenen Umsetzungen.

Nach diesem Schritt wird die Modellsuche durchgeführt. Die Methodik, die bei Eiter et al. [27] angewandt wird, kann als *Guess and Check* Methode deklariert werden. Zuerst wird ein mögliches Modell generiert (*Guess*) und danach wird überprüft, ob die Stabilitätsbedingung erfüllt ist. Wann ein Modell als stabil gilt, kann bei der Semantik in Abschnitt 5.4 nachgelesen werden. Auch Lifschitz [97] verfährt nach einem ähnlichen Prinzip, wobei er die Regeln, die im Programmcode vorkommen, in bestimmte Blöcke *Generate, Define, Test* einteilt. Im Block *Generate* sind die Regeln definiert, die eine große Anzahl an möglichen Modellen liefern können. Im *Define* Block werden Regeln definiert, die als Hilfe benötigt werden, und im letzten Block *Test* werden die Einschränkungen festgelegt, die erfüllt sein müssen. Daher muss auch bei der Modellsuche jedes aus dem Teil *Generate* generierte Modell bezüglich dem Teil *Test* geprüft werden. Ein einfaches Beispiel ist in der Abbildung 5.2 veranschaulicht, wobei die durchgezogene Linie einer deterministischen Ausbreitung entspricht. Konkret bei DLV wird die Methodik *Guess/Check/Optimize (GCO)* [57] angewandt. Im ersten Teil *Guess* wird der Suchraum

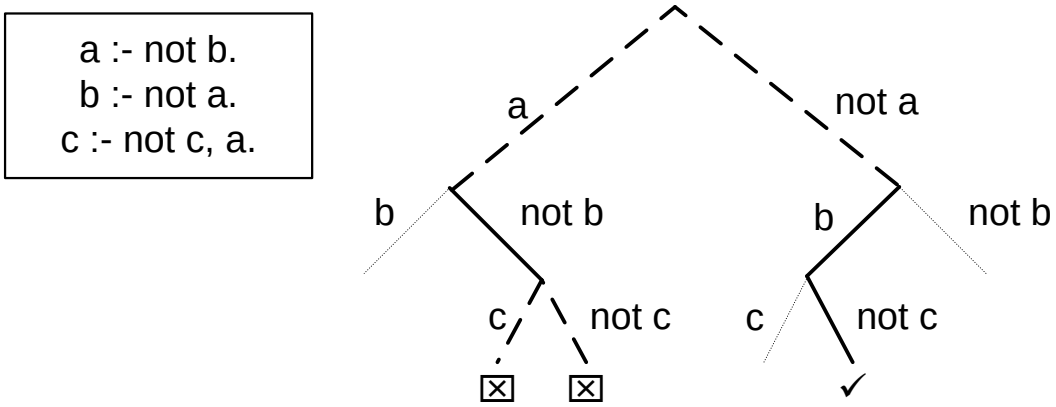

Abbildung 5.2: Modellsuche anhand eines einfachen Programms [27, S. 90]

festgelegt, in dem alle möglichen Modellinstanzen liegen können. *Check* filtert alle Modelle, die nicht erlaubt sind, und liefert nur noch die Möglichkeiten, die die Bedingungen erfüllen. Bis zu diesem Teil ist der Ansatz mit dem Ansatz *Guess and Check* vergleichbar. Jedoch kann nun als dritter Schritt *Optimize* durchgeführt werden. Dieser Schritt ist abhängig davon, ob in dem Programm *weak constraints* definiert wurden. Näheres zu *weak constraints* kann im Unterabschnitt 6.2.1 Syntax von DLV nachgelesen werden. Mit Hilfe dieser Constraints ist es möglich konkrete Kosten für Lösungen festzulegen. Es werden also nur noch die Modelle ausgegeben, die einen minimalen Kostenwert haben.

Das Ergebnis sind die berechneten Modelle, beziehungsweise auch *Answer Sets* genannt. Welche Modelle zurückgeliefert werden, ist abhängig von dem gewählten *reasoning mode*, wobei die der Anfrage entsprechenden *Answer Sets* textuell zurückgeliefert werden [33]. Je nachdem, ob das Ergebnis zufriedenstellend ist oder nicht, kann das Ausgangsproblem verfeinert, Regeln hinzugefügt und so der Prozess von neuem gestartet werden.

5.3 Syntax von ASP

Ein Programm, welches mit Hilfe von ASP gelöst werden soll, besteht aus Regeln. Diese Regeln sind aber durch einige weitere Elemente definiert, die jetzt erläutert werden sollen.

Konstanten, Variablen, Prädikate und Funktionen

ASP baut im Grunde auf dem theoretischen Konstrukt einer Modellstruktur auf, welche aus folgenden vier Bestandteilen besteht [28]:

1. Gegenstandsbereich (Domäne)

2. Menge von Funktionen

3. Menge von Prädikaten

4. Menge von Konstanten

Die dazugehörende Signatur besteht aus den drei Komponenten

1. Menge von Funktionssymbolen

2. Menge von Prädikatensymbolen

3. Menge von Konstantensymbolen

Die Domäne ist der Bereich, aus dem die Werte allgemein gewählt werden dürfen. Die Menge von Konstantensymbolen enthält für jede Konstante ein Konstantensymbol. Konstanten sind festgelegte gegebene Werte, die nicht mehr verändert werden können. Zwei Konstanten seien hier erwähnt, nämlich verum $\top$ (immer wahr) und falsum $\bot$ (immer falsch). Variablen sind freie unbekannte Individuen und dürfen prinzipiell jeden Wert aus der Domäne annehmen. (In ASP sind Variablen Platzhalter für gegebene Elemente.) Auch Funktionen werden konkrete Funktionssymbole zugeordnet, wobei Funktionen immer einen Wert aus der Domäne zurückliefern. Im Gegensatz dazu liefern Prädikate bei der Auswertung immer einen Wahrheitswert zurück. Sowohl Funktionen als auch Prädikate können einstellig oder mehrstellig sein, also mehrere Elemente umfassen. Terme können Variablen, Konstanten und/oder Funktionen umfassen. Ein kurzes Beispiel soll die verschiedenen Elemente und ihre Auswertung veranschaulichen.

Es wird folgende vereinfachte Familienstruktur als gegeben gesehen: [28, S. 86]
FamHuber = <Personen,{Vater,Mutter},{Geschwister,weiblich,männlich},{Max, Susi, Thomas, Berta}>
Die Domäne umfasst *Personen*, die Funktionen lauten *Vater* und *Mutter* und sind beide einstellige Funktionen. Die Prädikate sind *Geschwister* (zweistelliges Prädikat), *weiblich* und *männlich* (jeweils einstellig). *Max, Susi, Thomas, Berta* sind die bekannten Konstanten, wobei wir voraussetzen, dass Max und Susi Geschwister und Thomas und Berta ihre Eltern sind. *Vater(Susi)* würde also, da es sich hierbei um eine Funktion handelt, *Thomas* als Antwort zurückliefern. Während *Geschwister(Susi,Max)* den Wahrheitswert wahr als Ergebnis zurückgeben würde. *männlich(Berta)* würde das Ergebnis falsch liefern.

Dieselben Konstrukte werden in ASP verwendet, wobei bei den meisten Solvern auf die Verwendung von Funktionen verzichtet wird. *"To facilitate finite computations (and answer sets), many systems do not support function symbols (that is, they handle the so called Datalog fragment of logic programming), or only in a very limited form; this is because as already mentioned, function symbols are a well-known source of undecidability," [27, S. 89]*

Atome und Literale

Ein Atom ist ein Ausdruck mit folgender Form $p(t_1,t_2,...,t_n)$, wobei p ein Prädikatensymbol ist und t_1 bis t_n Terme sind. Ein Literal l wiederum kann ein positives Atom $p(t)$ oder negiertes Atom $\neg p(t)$ sein [34]. In Bezug auf positive und negative Literale muss beachtet werden, dass diese im Kontext der Wissensbasis bekannt sind. Das bedeutet, dass zum Beispiel wirklich das negierte Atom $\neg p(t)$ existiert. Allerdings wird in manchen Situationen eine Möglichkeit benötigt, die sicherstellt, dass für das Erfüllen einer Regel ein Literal nicht existiert, obwohl dies nicht explizit in der Wissensbasis stehen muss. Dies wird mit Hilfe von *negation as failure* bewerkstelligt und wird in der Form *not* l geschrieben [34]. Ein anderer Name für diese Art der Negation wäre *default negation*. Literale, die ein *not* vorangestellt haben, werden *"extended literals"* [34, S. 3] genannt. *Negation as failure* passt also solange l nicht in das Set der bekannten Literale aufgenommen wird. Sobald l in das Set hinzukommt, ist *not* l nicht mehr erfüllt.

Regeln

Regeln sind der Hauptbestandteil von ASP. Allgemein kann zwischen verschiedenen Arten von "logic programs" unterschieden werden und entsprechend unterscheidet sich die Form der Regeln, wobei die Grundstruktur immer dieselbe ist. Eine Regel besteht aus einem Kopf H (*Head* genannt), einem Ableitungszeichen $\leftarrow$ und einem Körper B (*Body*).

$$\mathrm{H} \leftarrow \mathrm{B}$$

Abbildung 5.3: Grundstruktur einer Regel

Auf einige Formen von Regeln wird nun näher eingegangen. Die Syntax von Regeln in einem *"normal logic program (nlp)"* [34, S. 3] ist konkret vorgegeben. Der Head besteht aus einem einzigen Atom A und im Body kommen beliebig viele Literale L vor.

$$A \leftarrow L_1, \ldots, L_m$$

Abbildung 5.4: Syntax einer Regel von nlp [35, S. 1]

Diese Regel drückt eigentlich aus, dass A durch L_1 und L_2 und ... bis L_m impliziert wird und so könnte auch die mathematische Formel angegeben werden:

$$(L_1 \wedge \ldots \wedge L_m) \supset A$$

Abbildung 5.5: Mathematische Formel einer Regel von nlp [35, S. 1]

Wenn der *Body* von normal logic program durch die Möglichkeit von *negation as failure* erweitert wird, spricht man von einem erweiterten logischen Programm. Eine weitere Möglichkeit ist das *"normal disjunctive logic program (ndlp)"* [63, S. 3]. Der Head der Regel besteht hierbei aus einer Disjunktion von Atomen und der Body aus Atomen und Atomen, die mit Hilfe von *negation as failure* negiert wurden.

$$a_1 \vee a_2 \vee ... \vee a_k \leftarrow b_1, ... , b_m, \text{not } c_1, ... , \text{not } c_n$$
$$k, m, n \geq 0$$

Abbildung 5.6: Syntax einer Regel in ndlp [63]

Da in dieser Arbeit für die Generierung der Testinstanzen (6.2) ebenfalls ein disjunktiver Ansatz nämlich *"extended disjunctive logic program (edlp)"* [63, S. 1] benutzt wurde, wird auf diesen noch näher eingegangen. In diesem Fall können im Gegensatz zu dem ndlp Literale im Head (h) und Body (l) stehen.

$$h_1 \vee h_2 \vee ... \vee h_k \leftarrow l_1, ... , l_m, \text{not } l_{m+1}, ... , \text{not } l_n$$
$$k \geq 0, n \geq m \geq 0$$

Abbildung 5.7: Regel in edlp [34]

In dieser Form der Regeln kann nun sowohl die echte Negation als auch *negation as failure* verwendet werden. Außerdem besteht der Head nicht mehr nur aus positiven Atomen, sondern kann auch negierte Atome umfassen. Durch diese Erweiterung ist es möglich mehr Informationen im Programm zu verarbeiten.

Die Logik einer Regel ist ähnlich zu einem "if then"-Befehl in z.B. Java. Der Body ist der Teil der vorhanden sein muss (if) und der Head ist die Folgerung (then). Es gibt aber nie einen "else" Teil.

Fakten

Fakten sind Regeln, die keinen Body haben. Ein leerer Body wird gehandhabt wie $\top$, also als wahr. Daher ist die Regel für den Head immer erfüllt. Im Allgemeinen wird daher bei den Fakten der Body und das Ableitungszeichen weggelassen. So ist zum Beispiel *p(a)* ein Faktum. Die gegebenen Fakten sind immer in der Wissensbasis und jede Regel, die das Gegenteil dieser Fakten im Body hat, kann nie wahr werden. Ein Beispiel hierfür wäre folgende Anfrage:
Faktum: *p(a)*
Regel: $q(a) \leftarrow \neg\ p(a)$
Das einzige mögliche Answer Set ist *p(a)*, da *q(a)* nur existieren könnte, wenn *p(a)* negiert vorkommen würde.

Constraints

Im Gegensatz zu Fakten sind Constraints Regeln, die nie erfüllt sein dürfen. Bei diesen Regeln ist der Head leer [34]. Jeder Constraint kann als Regel gesehen werden, wo ein Element positiv im Head und negiert im Body hinzukommt. So kann z.B. $\leftarrow c$, auch in der Form $a \leftarrow \neg\ a, c$ ausgedrückt werden. Diese Regel kann nie wahr werden. Diese Art der Constraints werden auch *integrity constraints* genannt. Mit Hilfe dieser Constraints ist es möglich Einschränkungen für die Answer Sets festzulegen.

5.4 Semantik von ASP

ASP basiert auf der Semantik von stabilen Modellen [58]. In diesem Bereich werden daher einige grundlegende Definitionen und Erklärungen für stabile Modelle gegeben. Die Sprache von einem logischen Programm wird durch die Variablen, Konstanten, Funktionen und Prädikate definiert. Auf dieser Grunddefinition bauen nun die weiteren Definitionen auf.

Herbranduniversum

> *"Given a logic program P, the Herbrand universe of P, HU(P), is the set of all terms which can be formed from constants and functions symbols in P (resp. the vocabulary of L, if explicitly known)." [27, S. 48]*

Das bedeutet, dass das *Herbranduniversum* die Menge der Grundterme über der Signatur $\sum$ des Programms P ist, welche alle Funktions- und Prädikatssymbole umfasst.

Herbrandbasis

Die *Herbrandbasis* von einem Programm P, HB(P), ist das Set aller *Grundatome* auch *ground atoms* genannt, welche durch die Prädikate in P und allen Termen in *HU(P)* gebildet werden [27]. Die *Grundatome* sind alle Atome, in denen keine Variablen vorkommen. Anders ausgedrückt ist die *Herbrandbasis* die Menge aller *Grundatome* über der Signatur $\sum$ von P.

Herbrandinterpretation

> *"A (Herbrand)interpretation is an interpretation I over HU(P), that is, I as subset of HB(P)." [27, S. 48]*

Jede Teilmenge von der *Herbrandbasis* ist also eine *Herbrandinterpretation* [7].

Herbrandmodelle

Eine *Herbrandinterpretation* M ist ein *Herbrandmodell*, falls gilt, dass P eine logische Konsequenz von M über der Signatur $\sum$ ist. [7]

Geschlossener Zustand

> *"Sei P ein erweitertes logisches Programm, in dessen Regeln keine Default-Negation auftritt, und sei S ein Zustand. S heißt geschlossen unter P, wenn für jede Regel r aus P gilt: Ist pos(r)⊆S,so ist head(r)∩S=∅." [7, S. 288]*

Gelfond-Lifschitz Redukt

Im Zusammenhang mit der Definition eines *geschlossenen Zustands* ist auch die Definition des Redukts, welches nach seinen Erfindern Gelfond und Lifschitz benannt ist, relevant. Sei P ein erweitertes logisches Programm, so gilt:

> *Für einen Zustand S definieren wir das Redukt P^S von P bzgl. S wie folgt:*
> *$P^S := \{H \leftarrow A_1,...,A_n \mid H \leftarrow A_1,...,A_n$, not $B_1,...$, not $B_m \in P$,*
> *$\{B_1,...,B_m\} \cap S = \emptyset\}$ [7, S. 288]*

Da bei dieser Reduktion alle Regeln mit *not* B im Body entfernt werden, wenn B $\in$ S ist und ansonsten die negativen Literale aus dem Body entfernt werden, handelt es sich bei dem *Gelfond-Lifschitz Redukt* um ein logisches Programm ohne *negation as failure*.

Stabile Modelle

Durch die vorhergehenden Definitionen ist es nun möglich eine Aussage zu treffen, wann ein Modell *stabiles Modell* genannt wird. Je nachdem, um welche Art von logischem Programm es sich handelt, unterscheidet sich die Definition hierfür ein wenig.

> *"Sei P ein klassisches logisches Programm. Das stabile Modell von P ist die kleinste Menge S von Atomen der Herbrandbasis H(P), die unter P geschlossen ist." [7, S. 289]*
> *"Sei P ein normales logisches Programm, und sei S eine Menge von Atomen. Ist S das stabile Modell von P^S, so heißt S stabiles Modell von P." [7, S. 289]*
> *"Seien P ein erweitertes logisches Programm ohne Default-Negation und S ein Zustand. S heißt Antwortmenge (answer set) von P, wenn S eine minimale, unter P geschlossene Menge ist (wobei Minimalität bzgl. Mengeninklusion gemeint ist)." [7, S. 293]*
> *"Seien P ein erweitertes logisches Programm und S ein Zustand. S heißt Antwortmenge (answer set) von P, wenn S Antwortmenge des Reduktes P^S ist." [7, S. 294]*

Für ASP relevant sind nun die letzten beiden Definitionen, da diese im Bezug auf die Struktur von ASP definiert worden sind. Allerdings könnten diese Definitionen nicht gemacht werden ohne den vorigen theoretischen Hintergrund. Dies ist nur ein kleiner Einblick in die Semantik von *Answer Sets*. Weitere Definitionen und Überlegungen können unter anderem bei Gelfond und Lifschitz [35] beziehungsweise Minker und Ruiz [63] nachgelesen werden.

5.5 Anwendungsgebiete

Abschließend sollen einige Anwendungsgebiete kurz erläutert werden.
ASP bietet generell Lösungen für verschiedene Entscheidungsprobleme, wenn diese durch logische Programme ausgedrückt werden können. Unter anderem wurde ASP bei dem Entscheidungsunterstützungssystem einer deutschen Versicherungsfirma in der Überprüfung der medizinischen Abrechnungen verwendet [6]. Die verwendete Implementierung

war der DLV Solver.
Answer Set Programming wird auch in anderen Anwendungsgebieten der Wissenschaft und Technik verwendet. So wurde zum Beispiel ein System entwickelt, welches Entscheidungshilfe bei Planungs- und Diagnoseaufgaben übernehmen kann und erfolgreich einige Funktionen eines Space Shuttles steuern kann [68]. Auch wurde bei einer Realisierung einer Web-basierten Produktkonfiguration das Wissen von ASP eingesetzt [58].
Ebenso wurde das Potenzial von logischen Programmieransätzen im Bereich der modellgetriebenen Softwareentwicklung erkannt und auch ausgenutzt. Eine Umsetzung hierzu findet sich im Unterabschnitt 8.1.1.

KAPITEL 6

White-Box Testing Tool[1]

In diesem Kapitel werden zunächst die einzelnen, unabhängigen Hauptbestandteile des entwickelten *"White-Box Testing Tools"* oder kurz *WBT-Tools* erläutert. Dabei handelt es sich um einzelne Projekte, die jeweils eine Hauptaufgabe übernehmen:

- Generieren von Modellinstanzen basierend auf einem Metamodell
- Überprüfen zusätzlicher Bedingungen, die im Metamodell nicht modelliert werden können
- Ausführen der Modelltransformation

Jede dieser Komponenten kann für sich allein eingesetzt werden.
Im Abschnitt 6.1 werden die Technologien, welche bei der Entwicklung des Tools verwendet wurden, näher erklärt. Danach wird im Abschnitt 6.2 ein Überblick über das Programm *DLV* gegeben. Außerdem wird der Einsatz dieses Programms und die Generierung der Modellinstanzen in der WBT-Tool Komponente näher erläutert. Im Abschnitt 6.3 wird zuerst ein Überblick zu *OCL* gegeben. Danach wird die Komponente des WBT-Tools, welche zur Überprüfung der zusätzlichen Bedingungen, also der OCL-Constraints, genutzt wird, als eigenständiges Programm betrachtet. Der Abschnitt 6.4 beschäftigt sich mit der Transformation von Modellen mit Hilfe von *ATL*. Nach einer kurzen Einleitung über diese Modelltransformationssprache folgt eine nähere Beschreibung des automatisierten Ansatzes zur Modelltransformation. Abschließend wird im Abschnitt 6.5 das WBT-Tool vorgestellt. Dieses Programm vereint alle Komponenten um ein durchgängiges, automatisiertes Testen von ATL-Transformationen zu ermöglichen.
Der Sourcecode des Programms kann als Eclipse Projekt unter `https://subversion.assembla.com/svn/wbt-tool/` ausgecheckt werden.

[1] Verfasser: Thomas Franz, BSc & Sabine Wolny, BSc

6.1 Eingesetzte Technologien[2]

Die Entwicklung des WBT-Tools, sowie der einzelnen Teilkomponenten, erfolgte in der objektorientierten Programmiersprache **Java**[3]. Die Gründe dafür waren

- Plattformunabhängigkeit um das Tool auf möglichst breiter Basis einsetzen zu können
- Tatsache, dass die Java Runtime Environment mittlerweile sehr weit verbreitet ist und daher keine besonderen Installationen erforderlich sind um das Tool ausführen zu können
- ausgewählte Entwicklungsumgebung

Als Entwicklungsumgebung wurde **Eclipse**[4] gewählt. Sie unterstützt eine Vielzahl an Erweiterungen, darunter auch solche, die die Erzeugung, Verwaltung und Verarbeitung von Modellen unterstützen.

Eng mit Eclipse verbunden ist das **Eclipse Modeling Project**[5], das die Unterstützung und Weiterentwicklung von Modell-basierten Technologien im Rahmen von Eclipse zum Ziel hat. Um dieses zu erreichen stellt es vereinheitlichte Modellierungsumgebungen, Werkzeuge und Standard-Implementierungen zur Verfügung.

Die Subprojekte des Eclipse Modeling Project, die im Rahmen dieser Arbeit von Bedeutung sind, seien hier aufgezählt:

- **Eclipse Modeling Framework (EMF)**
- **Eclipse Modeling Framework Technologies (EMFT)**
- **Model Development Tools (MDT)**
- **Model to Model (M2M)**
- **Model to Text (M2T)**

Beim Eclipse Modeling Framework handelt es sich um eine Eclipse-basierte Modellierungsumgebung, die die Erzeugung, Verwaltung und Verarbeitung von Modellen sowie weitere Aufgaben unterstützt. Einer der Hauptbestandteile von EMF ist das Metamodell **Ecore**, das auf dem EMOF Standard basiert. Außerdem unterstützt EMF die Serialisierung von Modellen in das Austauschformat **XMI**.

Die ursprünglich als eigenes Projekt entwickelte Plattform openArchitectureWare (oAW) fokussierte die Unterstützung von modellgetriebener Softwareentwicklung und modellgetriebenem Testen. Die oAW[6] besteht unter anderem aus einer Sprachfamilie

[2] Verfasser: Sabine Wolny, BSc
[3] `https://de.wikipedia.org/wiki/Java_(Programmiersprache)`, Zuletzt besucht: 18-03-2013
[4] `https://de.wikipedia.org/wiki/Eclipse_(IDE)`, Zuletzt besucht: 18-03-2013
[5] `http://www.eclipse.org/modeling/`, Zuletzt besucht: 18-03-2013
[6] `https://de.wikipedia.org/wiki/OpenArchitectureWare`, Zuletzt besucht: 18-03-2013

(u.a. **Xpand**) und einem Generator, der **Modeling Workflow Engine (MWE)**, die nun Bestandteile verschiedener Eclipse-Projekte sind.

So ist etwa die MWE ein inaktiver Bestandteil des EMFT-Projekts[7], welches neue Komponenten zur Erweiterung oder Ergänzung des EMF zur Verfügung stellt. Die Templatesprache Xpand[8] dient zur Konvertierung von Modellen in Text (in den meisten Fällen in Code). Daher ist sie auch Bestandteil des M2T-Projektes[9], das genau diese Konvertierungen als eine seiner Hauptaufgaben definiert. Beide Technologien wurden verwendet, um aus dem Metamodell Code zur Erzeugung von Modellen zu generieren.

Für die Erzeugung von Modellen aus einem Metamodell wurde der DLV Solver von **DLVSystem** verwendet, ein System, das zur Lösung von Problemen mit Hilfe von ASP eingesetzt wird. Im Rahmen dieser Arbeit wurde der **DLVWrapper** eingesetzt, eine in Java implementierte Bibliothek, die den Einsatz von DLV in eigenständigen Programmen ermöglicht (siehe Abschnitt 6.2).

Das MDT-Projekt beschäftigt sich mit der Erzeugung von Metamodellen, die auf Industriestandards basieren, sowie der Entwicklung von Tools, die die Ableitung von Modellen aus diesen Metamodellen unterstützen. Ein Bestandteil dieses Projektes ist die **Object Constraint Language (OCL)**, die zur Beschreibung von Einschränkungen, die das Modell erfüllen muss, welche aber nicht direkt modelliert werden können, verwendet wird (siehe Abschnitt 6.3).

M2M (auch Model to Model Transformation (MMT)) beschäftigt sich mit der Transformation von Modellen. Eine Möglichkeit dies zu bewerkstelligen baut auf der **Atlas Transformation Language (ATL)** auf (siehe Abschnitt 6.4).

Da alle diese Komponenten zum Teil selbst in Java programmiert sind (z.B. ATL) oder Schnittstellen zur Integration in Java-Code zur Verfügung stellen, erwies sich die Wahl dieser Umgebung, im Hinblick auf die Entwicklung eines möglichst vollständig automatisierten Test-Tools, als vorteilhaft. In Tabelle 6.1 sind alle Technologien mit der Version, in der sie verwendet wurden, aufgeschlüsselt.

[7] `http://wiki.eclipse.org/Eclipse_Modeling_Framework_Technologies`, Zuletzt besucht: 18-03-2013

[8] `http://www.openarchitectureware.org/`, Zuletzt besucht: 18-03-2013

[9] `http://www.eclipse.org/modeling/m2t/`, Zuletzt besucht: 18-03-2013

Tabelle 6.1: Technologie Versionen

Technologie	Version
Java	1.6.
Eclipse	3.7. (Projektname “Indigo”)
EMF	Eclipse Plug-in Version: 2.7.2.
Ecore Tools	Eclipse Plug-in Version: 1.0.0.
DLV	DLV-Wrapper Version: 4.2.
MWE	Eclipse Plug-in Version: 1.1.1.
XPand	Eclipse Plug-in Version: 1.1.1.
OCL	Eclipse Plug-in Version: 3.1.1.
ATL	Eclipse Plug-in Version: 3.2.1.

6.2 DLV - Erzeugung der Testinstanzen[10]

Im Bereich ASP gibt es verschiedene programmiertechnische Umsetzungen. Eine davon nennt sich Datalog with disjunction (DLV). Dieses Programm ist frei verfügbar und ist ein deduktives Datenbanksystem. DLV wird von DLVSystem[11] zur Verfügung gestellt. Neben dem Programm gibt es auch noch weitere Anwendungen, von denen einige genannt werden sollen:

- DLV^{DB}: eine Erweiterung von DLV um verteilte Eingabe- und Ausgabedaten von Datenbanken bearbeiten zu können
- Aspide: eine Integrated Development Environment (IDE) um den Entwicklungsprozess von ASP zu unterstützen
- JDLV: neues Framework um DLV mit Java Programmierung zu verbinden

6.2.1 Syntax von DLV

Datalog ist eine deklarative Programmiersprache, das heißt, im Gegensatz zu den imperativen Programmiersprachen[12] wird die Lösung beschrieben und nicht, wie der Prozess ablaufen muss. Es wird also festgelegt, wie die Lösung aussehen soll [53]. Die “Datalog Inferenzmaschine”, oder auch deduktives Datenbanksystem genannt, versucht dann einen Weg zu finden das Problem zu lösen bzw. die gewünschte Lösung zu erzielen [53]. Dies geschieht auf Basis von Regeln und Fakten, wie es in ASP üblich ist. Hierzu gibt das Kapitel 5 nähere Erklärungen ab. Das besondere an DLV ist, dass Datalog um die Disjunktion erweitert wurde, sodass Regeln mit logischem Ausdruck ODER (Zeichen v) erlaubt sind. Dies ist besonders wichtig um bewerkstelligen zu können, dass ein Ausdruck in positiver oder negierter Form zur Lösung hinzukommt, wenn die Regel dafür

[10] Verfasser: Sabine Wolny, BSc
[11] `http://www.dlvsystem.com/`, Zuletzt besucht: 18-03-2013
[12] `http://de.wikipedia.org/wiki/Deklarative_Programmierung`, Zuletzt besucht: 15-03-2013

erfüllt ist. Ein einfaches Beispiel hierfür wäre: *instanz1* gehört zu *haus* oder nicht, wobei *instanz1* und *haus* gegebene Konstanten sind.

```
class(haus).
alphabet(instanz1).
of(X,Y) v -of(X,Y) :- alphabet(X),class(Y).
```

Wenn dieses Programm mit DLV gelöst wird, sind in der Lösung folgende zwei Modelle, auch *Answer Sets* genannt:

```
{class(haus), alphabet(instanz1), of(instanz1,haus)},
{class(haus), alphabet(instanz1), -of(instanz1,haus)}
```

Anhand dieses Beispiels lässt sich auch die Syntax von DLV erkennen. So sind die gegebenen *Atome* `class(haus)` und `alphabet(instanz1)`. Da diese bekannt sind, werden sie auch *Fakten* genannt. `of(X,Y) v -of(X,Y) :- ...` ist eine Regel, welche wiederum aus Atomen besteht und durch die Fakten bestimmt wird. Im Programm wird allgemein zwischen Konstanten und Variablen unterschieden. Konstanten (haus,instanz1) müssen mit einem Kleinbuchstaben beginnen oder sie werden unter Anführungszeichen gesetzt. Variablen (X,Y) beginnen mit einem Großbuchstaben und sind Platzhalter für Konstanten. Es gibt auch noch eine besondere Form der Variable, die sogenannte anonyme Variable. Diese ist ein "_" und beschreibt den Platz, wo eine Variable steht. Jeder _ beschreibt in der Regel, in der sie vorkommt, eine neue eigene Variable, die an keiner anderen Position der Regel existiert [9]. Atome haben die Form $\texttt{a(t}_1\texttt{,t}_2\texttt{,...,t}_n\texttt{)}$, wobei a der Name des Atoms ist und t_1 bis t_n wiederum Variablen oder Konstanten sind [17]. Atome müssen mit einem Buchstaben beginnen. Jedes Faktum bzw. jede Regel muss mit einem Punkt enden und das Ableitungszeichen :- ist bei Regeln relevant. Ein anderer wichtiger Punkt bezüglich der Syntax von DLV, der noch erwähnt sein soll, ist die Möglichkeit der Verwendung von *Aggregatfunktionen*. An sich erlaubt DLV keine Funktionen [17], allerdings werden fünf Aggregatfunktionen zusätzlich zur Verfügung gestellt. Die Funktionen lauten *#count, #sum, #times, #min* und *#max* [9]. Der Unterschied zur restlichen Programmlogik ist, dass Funktionen keinen Wahrheitswert, sondern wieder einen konkreten Wert zurückliefern. In diesem Fall wird immer eine Zahl zurückgeliefert.

- #count: zählt die Vorkommnisse einer Variable in einem Set
- #sum: liefert die Summe einer Variable in einem Set
- #times: liefert im Gegensatz zu #sum das Produkt
- #min: liefert den kleinsten Wert einer Variable in einem Set
- #max: liefert den größten Wert einer Variable in einem Set

Ein einfaches Beispiel soll nun einige Aggregatfunktionen veranschaulichen. Die Umsetzung erfolgt bei den anderen Funktionen in ähnlicher Weise und kann im "User Manual" [9] nachgelesen werden.

Es gibt verschiedene Arten von Wohnungen und zu diesen wird neben dem Typ auch die Größe (in m^2) gespeichert in der Form Wohnung(typ,größe).

```
Wohnung(einfamilienhaus,99).Wohnung(einfamilienhaus,120).
Wohnung(einfamilienhaus,220).Wohnung(einfamilienhaus,50).
Wohnung(einfamilienhaus,180).Wohnung(einfamilienhaushaelfte,50).
Wohnung(einfamilienhaushaelfte,25).Wohnung(einfamilienhaushaelfte,110).
Wohnung(eigentumswohnung,70).Wohnung(eigentumswohnung,100).
Wohnung(eigentumswohnung,150).Wohnung(eigentumswohnung,65).
Wohnung(mietwohnung,45).Wohnung(mietwohnung,65).
Wohnung(mietwohnung,80).Wohnung(mietwohnung,120).
ueber100(X) :- #count{T : Wohnung(T,G),G>100} = X.
kleinsteWohnung(X) :- #min{G : Wohnung(T,G)} = X.
groessteWohnung(X) :- #max{G : Wohnung(T,G)} = X.
```

Es wird berechnet, wie viele Wohnungen größer als 100 m^2 sind ($\rightarrow$ vier Wohnungen), wie klein die kleinste Wohnung ($\rightarrow$ 25 m^2) und wie groß die größte Wohnung ($\rightarrow$ 220 m^2) ist. Das Ergebnis, wobei die Fakten nicht angegeben wurden, sieht folgendermaßen aus:

```
{ueber100(4), kleinsteWohnung(25), groessteWohnung(220)}
```

Es muss also für jede Aggregatfunktion immer ein Set angegeben werden, in dem die Funktion rechnen soll. Bei der *count* Funktion soll die Anzahl an Typen ausgegeben werden, wo die Größe 100 übersteigt. Die Aggregatfunktionen *min* und *max* können nur auf Integer-Werte angewandt werden und daher wird hier die Größe als Variable ausgewählt. Außerdem ist es möglich in DLV *weak constraints* zu definieren [9]. Im Gegensatz zu den *integrity constraints*, die immer erfüllt sein müssen, ist eine Verletzung von *weak constraints* noch kein sofortiger Grund, dass ein Modell aus der möglichen Lösung gelöscht wird. *Weak constraints* werden eingesetzt um Optimierungen durchführen zu können. Diese *weak constraints* sind gewichtet und/oder haben Prioritäten, sodass eine Rangordnung zwischen den einzelnen entsteht. Sollte kein Gewicht oder Level explizit angegeben werden, wird standardmäßig 1 verwendet. Die Syntax ist im "User Manual" [9] klar vorgegeben, wobei Gewicht und Level optional sind:

```
:~ Constraint [Gewicht:Level]
```

Für die möglichen *Answer Sets* wird nun immer gezählt, wie viele *weak constraints* verletzt werden. In die Lösung werden nur die aufgenommen, die bezüglich der Anzahl an Verletzungen minimal sind. Außerdem wird beachtet, ob ein *weak constraint* eine höhere Priorität hat und/oder stärker gewichtet ist, als ein anderer. Wenn das der Fall ist, werden Lösungen, wo diese *weak constraints* verletzt werden, zuerst eliminiert. Gewichte und Prioritäten werden intuitiv nach der Wichtigkeit der Bedingung vergeben.

Ein kleines Beispiel soll die Funktion nun veranschaulichen: Angenommen Tom ist bei Susi eingeladen und weiß, Susi mag gerne Schokolade oder Tee. Wenn er Schokolade mitbringt, schenkt er ihr auch Blumen. Allerdings sagt Susi immer, er müsse gar nichts mitbringen. Was soll er denn jetzt mitbringen?

```
schokolade v tee.
blumen :- schokolade.
:~ blumen.
:~ tee.
:~ schokolade.
```

Die Antwort ist `tee`, da das Modell `blumen,schokolade` im Gegensatz dazu nicht nur einen, sondern zwei Constraints verletzt. Natürlich können auch mehrere *Answer Sets* Lösung sein, wenn diese gleich minimal in der Anzahl an Verletzungen von *weak constraints* sind.
Ein letzter Punkt bezüglich DLV sei noch erwähnt, nämlich die Möglichkeit Anfragen (sogenannte *Queries*) zu stellen. Dann werden anstatt der Ausgabe aller Modelle, die die Regeln erfüllen, entweder nur die Modelle ausgegeben, die die konkrete Anfrage erfüllen, und/oder es wird eine Aussage bezüglich des Erfüllens oder nicht Erfüllens zurückgeliefert. Diese Möglichkeiten seien hier nur kurz erwähnt, da für die Entwicklung des Tools alle *Answer Sets* von Bedeutung waren. Der interessierte Leser kann diese Möglichkeiten im "User Manual" [9] nachlesen:

- *ground queries*: Ergebnis der Anfrage: alle Answer Sets, die die Query erfüllen
- *cautious reasoning*: Ist die Anfrage in allen Answer Sets erfüllt?
- *brave reasoning*: Gibt es ein Answer Set, welches die Query erfüllt?

6.2.2 DLV-Tool

Das Programm, welches zur automatischen Erzeugung von Modellinstanzen aus einem Metamodell dient, wurde in Java entwickelt. Um das Programm erfolgreich zu starten müssen mehrere Parameter in der richtigen Reihenfolge der *main-Methode* in der Klasse `DLVRunner.java` übergeben werden:

1. Der Pfad zum Input Metamodell (muss ein Ecore-File sein)
2. Der Pfad zum Ordner, in dem das generierte DLV-File gespeichert werden soll
3. Der Pfad zum Ordner, in dem die generierten Modellinstanzen gespeichert werden sollen
4. Der Pfad zum ausführbaren Programm von DLV (entweder eine exe-Datei für Windows oder eine bin-Datei für Linux)
5. Optional: Die Anzahl der maximal generierten Instanzen (default: 2)

Da das Programm nicht nur als eigenständiges Programm, sondern auch in dem automatisierten WBT-Tool funktionieren soll, müssen auch die Pfade, die im Punkt 2 und 3 beschrieben werden, angegeben werden. Optional kann die maximale Anzahl an Instanzen für das Metamodell bestimmt werden. Im Allgemeinen ist diese auf 2 gesetzt

um die Anzahl der generierten Möglichkeiten in einem kleinen Rahmen zu halten, da DLV immer alle möglichen Sets generiert. Der Wert muss gesetzt werden (daher standardmäßig auf 2), sonst kann das Programm nicht nach endlicher Zeit aufhören. Wenn keine obere Grenze gesetzt wäre, würde DLV alle möglichen Ausgänge mit immer mehr Instanzen generieren. Wenn der Benutzer die Parameter richtig angegeben hat, wird das Programm ohne weitere Benutzeraktion durchgeführt. Das Programm ist in zwei Arbeitsschritte aufgeteilt. Zuerst wird der MWE workflow ausgeführt, welcher das im Metamodell modellierte Szenario in die Syntax von DLV überführt und in einem eigenen File speichert. Im zweiten Schritt werden dann die gefundenen Lösungen wieder in XMI Files übergeführt um die Modelle für weitere Anwendungen nutzbar zu machen. Nach der Durchführung liefert das Programm die Information, wie viele Modelle mit Hilfe von DLV generiert und wie viele als XMI erzeugt wurden. Die Werte unterscheiden sich in der Regel genau um 1, weil der DLV Solver meist auch die leere Lösung als Lösung zurückliefert, diese ist jedoch nicht von Bedeutung. Sollte ein Fehler auftreten und eine Exception geworfen werden, wird diese gefangen und ausgegeben.

In der Abbildung 6.1 ist die Architektur des DLV-Tools vereinfacht dargestellt, wobei die genaue Funktionsweise der einzelnen Teile in den folgenden Abschnitten näher erklärt wird. Im Unterabschnitt 6.2.2.1 wird die Erzeugung des Programmcodes für den DLV Solver und im Unterabschnitt 6.2.2.2 die Speicherung der Modelle beschrieben.

6.2.2.1 Überführung eines Metamodells in Programmcode für den DLV Solver

Die Lösungen bzw. "Answer Sets" für das gegebene Metamodell werden direkt im DLV Solver gesucht, wobei der DLV-Wrapper den externen Aufruf des Programms in Java ermöglicht. Somit müssen die Informationen des Metamodells zuerst in eine für den Solver verständliche Sprache übergeführt werden. Um dies zu bewerkstelligen wird mit Hilfe des MWE workflows eine M2T Transformation durchgeführt. Der MWE workflow benötigt folgende Parameter, die gesetzt werden müssen:

1. Der Name des Metamodells mit Dateiendung
2. Der Pfad zum Ordner für das generierte DLV File
3. Der Pfad zum Ordner des Metamodells
4. Der Pfad zum Metamodell
5. Die Anzahl der maximalen Instanzen

Da das Metamodell nicht notwendigerweise im "workspace" von Eclipse liegen muss, musste für den MWE workflow das `org.eclipse.emf.mwe.utils.StandaloneSetup` erweitert werden. Die Methode `addRegisterEcoreFile(String file)` wird konkret mit dem richtigen Dateipfad aufgerufen und nicht nur mit dem Namen der Datei. Genauso wird bei der Methode `setUri(String file)` vom `org.eclipse.emf.mwe.utils.Reader` verfahren.

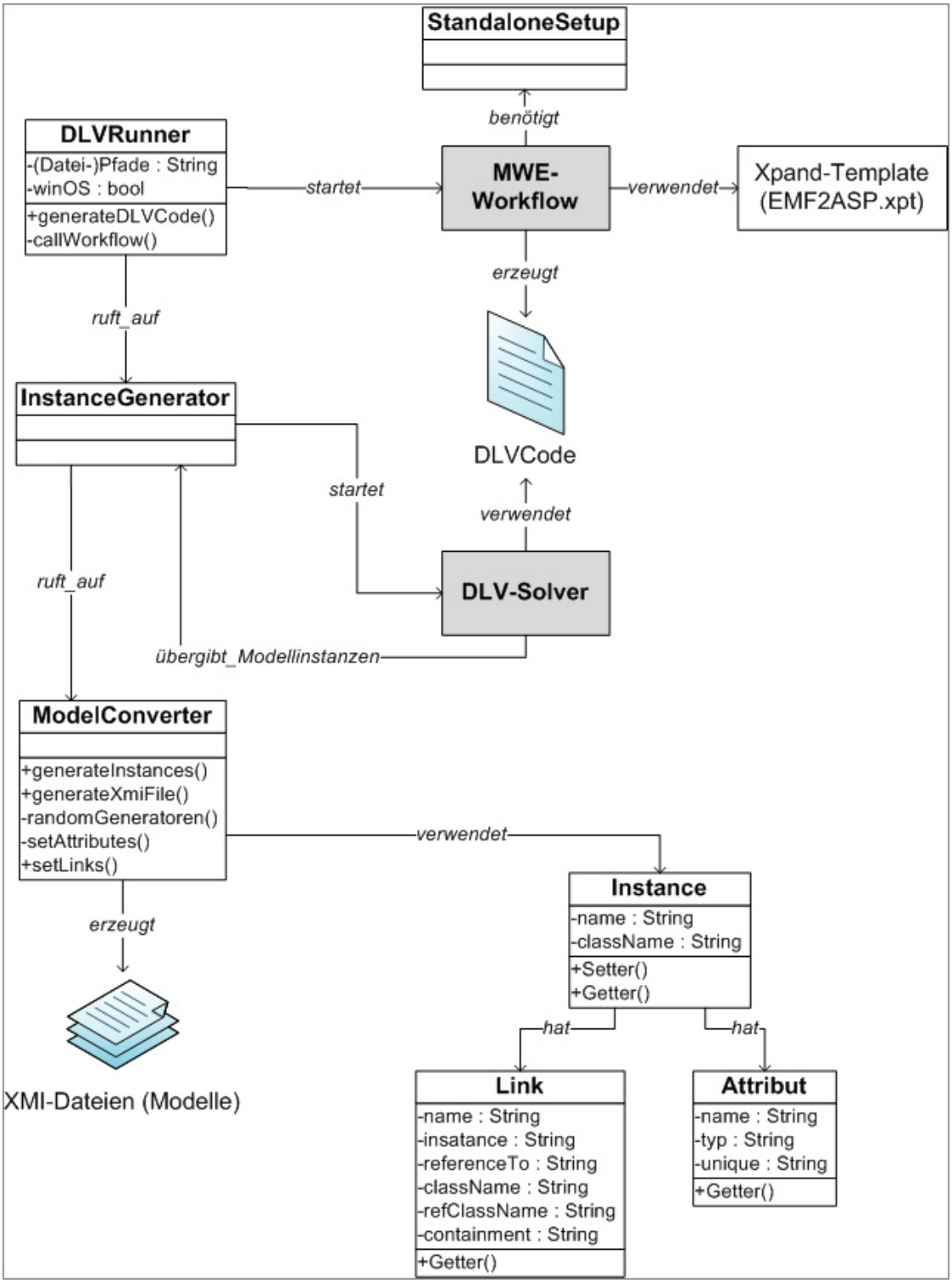

Abbildung 6.1: Architektur des DLV-Tools

So ist es nun möglich Metamodelle aufzurufen, die in einem beliebigen Ordner liegen. Nach der Registrierung des Metamodells wird das *Xpand File* für dieses aufgerufen. Xpand[13] ist eine Template-Sprache im Bereich M2T, wobei die Generierung von Text aus Modellen im Vordergrund steht. Ursprünglich war Xpand als Teil von oAW konzipiert, ist aber mittlerweile eine Komponente von Eclipse.

Tabelle 6.2: Elemente des Metamodells, die unterstützt bzw. nicht unterstützt werden

Objekte bzw. Verbindungen	Unterstützt
EPackage	✓
Subpackage	
EClass	✓
EAttribute single valued	✓
EAttribute multi valued	
EOperation	
EEnum	✓
EnumLiteral	✓
EDataType	
EAnnotation	
Details Entry	
EAnnotationLink	
EReference	✓
Inheritance	✓

In Tabelle 6.2 und Tabelle 6.3 wird veranschaulicht, welche Elemente von einem Ecore-Metamodell im Xpand File und dann im DLV Programm umgesetzt wurden. Zusammenfassend ergeben sich bestimmte Bedingungen, die für die Elemente aus dem Metamodell gelten müssen:

- Es kann nur ein EPackage verarbeitet werden; das bedeutet, dass weder Unterpackages noch weitere Packages neben dem Hauptpackage existieren dürfen.
- Namen von Klassen müssen eindeutig sein, da sie sonst vom DLV Solver als dieselbe Klasse angesehen werden. (Konstanten, die denselben Namen haben, können nicht unterschieden werden.)
- Bei Attributen werden keine Collections beachtet. Zusätzlich bedeutet das, dass nur Multiplizitäten bzgl. mindestens 0 oder 1 und maximal 1 beachtet werden.
- Der Typ von den Attributen ist beschränkt auf die Standardtypen (int, double, String, boolean) oder Enumeration.

[13] `www.eclipse.org/modeling/m2t/?project=xpand`, Zuletzt besucht: 23-03-2013

Tabelle 6.3: Unterstützte bzw. nicht unterstützte Eigenschaften der umgesetzten Elemente

Objekte bzw. Verbindungen	Properties	
	unterstützt	nicht unterstützt
EPackage	Name NS Prefix NS URI	
EClass	Name Abstract ESuperType	Default Value Interface Instance Type Name
EAttribute	Name EType Unique Lower Bound Upper Bound	Changeable Default Value Literal Derived ID Ordered Transient Unsettable Volatile
EEnum	Default Value Name	Instance Type Name Serializable
EnumLiteral	Literal Name	Value
EReference	Name EType EOpposite Containment Lower Bound Upper Bound	Changeable Container Default Value Literal Derived EKeys Ordered Resolve Proxies Transient Unique Unsettable Volatile

Die Auswahl von Elementen und Eigenschaften, die umgesetzt wurden, erfolgte anhand von Überlegungen, welche Elemente bei einer Modelltransformation mit ATL getestet werden sollen. Mehrere Eigenschaften, wie zum Beispiel derived, wurden daher nicht beachtet, da diese vor allem für die Java Code Generierung und nicht konkret für Modelltransformationen relevant sind.

Im folgenden werden nun einige Kernpunkte der Xpand Datei "EMF2ASP" (siehe Abbildung 6.1) näher erläutert. Prinzipiell wird im workflow eine definierte Regel (Syntax: «DEFINE name FOR element») aus dem Xpand File aufgerufen und diese wird dann abgearbeitet. Im Fall des DLV-Tools wird die definierte Regel für das root Package aufgerufen und zusätzlich ist dieser noch ein Parameter übergeben:

```
«DEFINE main(int maxInt) FOR ecore::EPackage»
```

Der übergebene Parameter ist die vom User eingegebene Anzahl an Instanzen oder standardmäßig "2" und legt damit fest, wie viele maximale Instanzen pro Klasse erzeugt werden. Für Klassen und Enumerations werden die eigens definierten Teile ausgeführt, wobei die gesetzten Werte in einer bestimmten Struktur gespeichert werden. Der übergebene Parameter in der main Regel wird daher erst im Teil für die Klassen relevant. So wird abhängig von dessen Wert das Element `alphabet(«this.name.toFirstLower()»«i»)` für jede Klasse erzeugt, wobei `«this.name.toFirstLower()»` der Name der Klasse und `«i»` eine Laufvariable ist, die bei 1 startet und bis zum maximalen Wert, der übergeben wurde, geht. Außerdem wird noch eine Regel erzeugt, die aussagt, dass diese Instanz der Klasse existieren kann oder nicht.

```
of(«this.name.toFirstLower()»«i»,"«this.name»") v
-of(«this.name.toFirstLower()»«i»,"«this.name»").
```

Für die Referenzen, die in einer Klasse existieren, werden folgende Informationen gespeichert:

```
reference("«this.name»","«e.eType.name»","«e.name»",«e.lowerBound»,
«IF e.upperBound==-1»100«ELSE»«e.upperBound»«ENDIF»,«e.containment»).
```

Diese speichern den Namen der Klasse, den Namen der referenzierten Klasse, den Namen der Referenz, den minimalen und den maximalen Wert, sowie den boolean Wert von containment. Sollte bei einer Referenz kein maximaler Wert gesetzt sein (also unendlich), wird bei Ecore-Metamodellen -1 gespeichert. DLV benötigt jedoch für den Constraint zur Überprüfung des maximalen Werts eine positive Zahl und so wird der Wert auf 100 gesetzt (was natürlich eine Einschränkung darstellt und voraussetzt, dass der minimale Wert kleiner 100 ist). Analog werden auch die anderen unterstützten Elemente in DLV Code umgesetzt. Wichtig in diesem Zusammenhang ist, dass mit `reference(...)` nur allgemein gespeichert wird, welche Referenzen eine Klasse hat. Das bedeutet, dass noch keine Links zwischen verschiedenen Instanzen existieren. Diese werden konkret mit folgender Regel festgelegt.

```
link(U,V,Z,T,T1) v -link(U,V,Z,T,T1) :-
class(T),class(T1),reference(T,T1,Z,_,_,_),of(U,T),of(V,T1).
```

Hier werden nur noch Variablen verwendet, da dies allgemein für alle Links aller Klassen gelten muss. Bei reference werden außerdem manche Elemente als anonyme Variable (_) angegeben, da diese an keiner anderen Position der Regel vorkommen und für die konkrete Regel keine Bedeutung haben und daher auf diese Elemente nicht zugegriffen werden muss. Außerdem werden noch einige Einschränkungen definiert, damit auch der semantische Kontext von z.B. einem maximalen Wert gegeben ist.

```
%Anzahl der Verlinkungen von einer Instanz zu anderen ohne Vererbung
anzahl(K,A,T,B,C) :- alphabet(K), of(K,B), free(C), not vererbung(C),
not isAbstract(B), not isAbstract(C), reference(B,C,T,_,_,_),
A1=#count{ X : link(K,X,T,B,C) }, A2=#count{ Y : link(Y,K,T,B,C) },
B!=C, A=A1+A2, #int(A1), #int(A2), #int(A).
%Constraint: Anzahl darf nicht über den maximalen Wert gehen
:- anzahl(K,A,Z,U,V), reference(U,V,Z,B,C,_), of(K,U),A>C.
```

Die Regel `anzahl(K,A,T,B,C)` ist eine Hilfsregel, um im weiteren Verlauf einen Constraint definieren zu können. In dieser Regel wird mit Hilfe der Aggregatfunktion *count* die Anzahl an Links von einer Instanz bezüglich einer Referenz gezählt, wobei die Instanz nicht in einer Vererbungsbeziehung vorkommt. Danach wird die Bedingung festgelegt, dass die Anzahl nicht größer sein darf als der maximale Wert. Constraints dürfen nie erfüllt sein: Daher muss der Fall modelliert werden, der ausgeschlossen werden soll. Wenn die maximale Anzahl kleiner gleich einem Wert sein soll, dann muss der Constraint genau die Aussage maximaler Wert über dem vorgegebenen Wert aussagen. Es wird also immer die Aussage modelliert, die nicht eintreten soll. Nach demselben Prinzip wird bei der Überprüfung des minimalen Werts und auch bei den Multiplizitäten der Attribute verfahren.

Zusätzlich zu dieser eher einfachen Überprüfung ist auch die Überprüfung innerhalb von Vererbungsbeziehungen wichtig. Das Prinzip der Vererbung korrekt umzusetzen ist einer der Knackpunkte im Bereich der Modellgenerierung. Soll eine Referenz von einer Klasse zu einer Klasse innerhalb einer Vererbungsbeziehung existieren, bedeutet das, dass die Multiplizitäten für den gesamten Vererbungsbaum eingehalten werden müssen. Um diese Anforderung zu erfüllen, dürfen Referenzen nicht denselben Namen haben. Ein vereinfachtes Beispiel ist in Abbildung 6.2 zu sehen. Die Klasse *A* hat eine Referenz *ref1* zu Klasse *B*, die die Superklasse der Klassen *C* und *D* ist. In diesem Beispiel bedeutet die Vererbung, dass die *ref1* nicht nur zu *B*, sondern auch zu *C* oder *D* zeigen darf. Die Referenz *ref2* gilt nur noch für *C* und *D* und die Referenz *ref3* darf nur auf *D* zeigen, da die Superklassen keinen Einfluss auf zusätzliche Funktionen der Subklassen haben. Durch diese Vererbungsbeziehung gestaltet sich daher auch die Überprüfung der Multiplizitäten als aufwendiger, da nun für den gesamten Vererbungsbaum die Einschränkungen gelten müssen. Zum Beispiel darf eine Instanz von *A* nur einen oder keinen Link *ref1* zu einer Instanz von *B* oder *C* oder *D* haben.

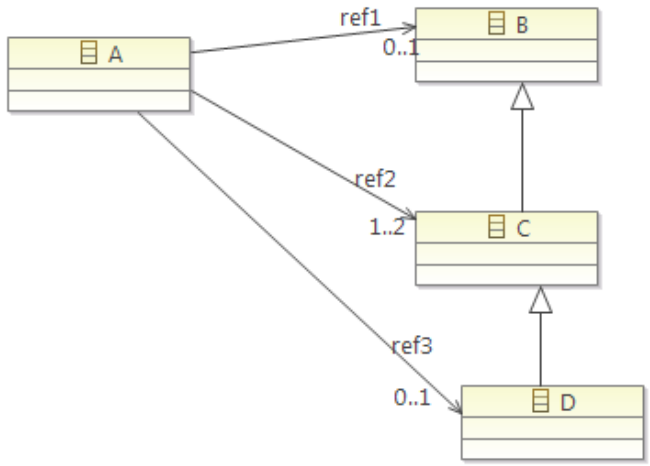

Abbildung 6.2: Referenzen von einer Klasse zu Klassen in einer Vererbungsbeziehung

Analog gilt dieses Prinzip auch für die anderen Referenzen. Außerdem muss für die Vererbung beachtet werden, dass der Vererbungsbaum unendlich viele Subklassen haben kann und auch zwei oder mehr Klassen von derselben Klasse erben können, und dass Vererbungsbeziehungen auch von abstrakten Klassen ausgehen können. Alle diese Möglichkeiten müssen daher korrekt umgesetzt werden.

Ein kleiner Ausschnitt des hierfür relevanten Codes folgt nun:

```
%Anzahl Referenzen mit Vererbung
anzahlVerb(K,A,T,B,C) :- alphabet(K), of(K,B), -free(C),
not isAbstract(B), not isAbstract(C), reference(B,C,T,_,_,_),
A1=#count{ X : link(K,X,T,B,C) }, A2=#count{ Y : link(Y,K,T,B,C) }, B!=C,
A=A1+A2, #int(A1), #int(A2), #int(A).
...
anzahlVerb(K,A,T,B,C) :- alphabet(K), of(K,B), free(C), vererbung(C),
not isAbstract(B), not isAbstract(C), reference(B,C,T,_,_,_),
A=#count{ X : link(K,X,T,B,C) }, B=C, #int(A).

%Anzahl von Links, wenn nur zu einer einzigen anderen Klasse
anzahlRef(K,A,T,B,C):- anzahlVerb(K,A,T,B,C),
#count{ I,D : anzahlVerb(K,I,T,B,D) } = 1.

%Anzahl Link in der Vererbung zwischen zwei Klassen
anzahlRef(K,Y,T,B,D) :- anzahlVerb(K,A1,T,B,C), anzahlVerb(K,A2,T,B,D),
 generic(C,D), Y=A1+A2, #int(Y).
...
```

```
%Anzahl der Links, wenn zwei Klassen von derselben Klasse erben
anzahlRef(K,Y,T,B,X) :- anzahlVerb(K,A1,T,B,C), anzahlVerb(K,A2,T,B,D),
C!=D, generic(C,X), generic(D,X), Y=A1+A2, #int(Y).

%Ueberpruefung der Anzahl bzgl. min und max bei Vererbung
maxReference(K,X,T,B,C) :- referenceUrsprung(W,C,T,_,_), of(K,B),
B=W, #max{S : anzahlRef(K,S,T,B,C)} = X, #int(X).

maxReference(K,X,T,B,C) :- referenceUrsprung(W,C,T,_,_), of(K,B),
generic(B,W), #max{S : anzahlRef(K,S,T,B,C)} = X, #int(X).

:- maxReference(K,A,T,B,C), referenceUrsprung(_,C,T,M,N),of(K,B),A>N.
...
```

Es ist notwendig über den gesamten Vererbungsbaum die Vorkommnisse der Links zu zählen und zu überprüfen. Die Zählung der Links erfolgt mit Hilfe von *anzahlVerb*, wobei die verschiedenen Möglichkeiten berücksichtigt werden. Bei *anzahlRef* werden nun alle Anzahlen der Links mit demselben Namen addiert, die von einer Instanz zu anderen Instanzen gehen. Für jede Klasse, die in einer Vererbungsbeziehung vorkommt, wird explizit noch in *generic(X,Y)* gespeichert von welcher Klasse sie erbt (X erbt von Y). So werden für das Beispiel von Abbildung 6.2 folgende Werte gespeichert:

```
generic(C,B).
generic(D,C).
generic(D,B).
```

Klar erkennbar ist, dass für D auch abgespeichert wird, dass diese Klasse von B erbt, obwohl sie nicht in direkter Beziehung steht. Da *generic(X,Y)* ein 2-Tupel ist, werden in *anzahlRef* auch nur jeweils 2-Tupel Beziehungen gespeichert. Die Überprüfung muss aber für den gesamten Vererbungsbaum erfolgen und so wird bei *maxReference* das Maximum aller 2-Tupel abgespeichert. Für dieses wird dann überprüft, ob es größer als der Minimalwert und kleiner als der Maximalwert ist. Die Überprüfung findet konkret nur einmal statt nämlich mit Hilfe der in *referenceUrsprung* gespeicherten Werte. Diese Referenz wird immer für die Klasse gespeichert, zu der die Referenz ursprünglich zeigt. Im Falle des Beispiels werden für die Referenz *ref1* alle Werte bezogen auf Klasse B in *referenceUrsprung* gespeichert.

Neben der sehr komplexen Lösung zur Vererbung werden auch die semantischen Bedeutungen von bidirektionalen Links und Containment Beziehungen umgesetzt.

Bei den Lösungen, die DLV liefert, sind auch symmetrische Lösungen enthalten, welche jedoch nicht eliminiert werden können. Hierzu ist ein einfaches Beispiel gegeben:

```
class(k1).
alphabet(instanz1).
alphabet(instanz2).
of(Y,X) v -of(Y,X) :- class(X), alphabet(Y).
```

```
:- of(Y,X), of(Y,Z), X!=Z.
...
```

In der Lösung gibt es unter anderem folgende Modelle:

```
{class(k1),alphabet(instanz1),alphabet(instanz2),
of(instanz1,k1), -of(instanz2,k1), ...}
{class(k1),alphabet(instanz1),alphabet(instanz2),
-of(instanz1,k1), of(instanz2,k1), ...}
```

Unter der Annahme, dass diese zwei Modelle dieselbe Struktur aufweisen, werden sie trotzdem als zwei verschiedene Modelle gehandhabt, da einmal der Name der existierenden Instanz *instanz1* lautet und einmal *instanz2*. In DLV gibt es keine Schlussfolgerungen über die einzelnen Modelle und daher können symmetrische Lösungen auch nicht eliminiert werden, da hierfür ein Vergleich von den einzelnen Answer Sets notwendig wäre.

Nach dieser Überführung in für den DLV Solver verständlichen Code wird der Solver mit Hilfe des DLV-Wrappers aufgerufen und sucht die Modelle zur erwünschten Lösung. Diese werden in einer Liste als Strings gespeichert.

6.2.2.2 Speicherung der Modelle mit Hilfe von Dynamic EMF

Während der DLV Solver die Modelle generiert, werden diese im `ModelBufferedHandler` des DLV-Wrappers in der Klasse `InstanceGenerator.java` gespeichert. Die einzelnen erzeugten Answer Sets werden gleich weiter an die Klasse `ModelConverter.java` zusammen mit dem Pfad zum Metamodell und dem Pfad zum Ordner, wo die Modelle gespeichert werden sollen (siehe Abbildung 6.1). Dort wird das Modell, welches aus einem einzigen zusammenhängenden String besteht, wieder in die einzelnen Bestandteile geteilt. Dafür werden zuerst für jedes einzelne Modell folgende Schritte durchgeführt, falls der jeweilige String existiert:

1. Die Instanzen werden als eigene Java Objekte erzeugt und in einer *HashMap* gespeichert.

2. Die Referenzen werden in einer HashMap gespeichert.

3. Enumerations werden in einer HashMap gespeichert.

4. Attribute werden als eigenes Java Objekt gespeichert und in einer Liste der zugehörigen Instanz zugeordnet.

5. Links werden als eigenes Java Objekt gespeichert. Zusätzlich wird der Wert der Containment Beziehung von der Referenz abgespeichert. Danach werden die Links in einer Liste bei den zugehörigen Instanzen gespeichert.

6. Enumeration Literale werden bei der richtigen Enumeration gespeichert.

Die Instanzen, Attribute und Links wurden als eigene Java Objekte gespeichert um eine größere Flexibilität in ihrer Nutzung zu haben. Außerdem können diese Elemente dadurch leichter um zusätzliche Eigenschaften erweitert werden. Nachdem nun auf die einzelnen Komponenten zugegriffen werden kann, ist es möglich, daraus wieder ein XMI Modell zu erzeugen, welches dann auch für die Überprüfung von OCL Constraints oder für die Modelltransformation mit Hilfe von ATL genutzt werden kann. Zur Erzeugung wurde Dynamic EMF genutzt. Der Vorteil ist, dass zu den Klassen des Metamodells nicht Java Sourcecode generiert werden muss um Modelle z.B. zu validieren, serialisieren oder zu aktualisieren [25]. EMF bietet ein Application Programming Interface (API), welches ermöglicht mit dynamischen, nicht generierten Modellen zu arbeiten[14]. Als erstes muss das Metamodell geladen und registriert werden, damit auf die einzelnen Objekte des Modells zugegriffen werden kann. Nach der Registrierung des Metamodells wird auf das Hauptpackage zugegriffen. Dort wird für jede einzelne Instanz die zugehörige EClass gesucht und dann wird das EObject dieser Instanz erzeugt.

```
//EFactory Objekt des Hauptpackages
EFactory factory = ePackage.getEFactoryInstance();
...
//name: Der Name der Klasse der Instanz
EClass object = (EClass) ePackage.getEClassifier(name);
EObject instance = factory.create(object);
```

Außerdem werden zu den einzelnen Objekten die zugehörigen Attribute bzw. Links gespeichert. Hierbei ist zu beachten, dass nur single-valued Attribute gespeichert werden können. In den Modellen wurde zuerst nur abgespeichert, ob ein Attribut existiert oder nicht, jedoch kein konkreter Wert. Dieser wird nun "zufällig" erzeugt. Hierzu wird `Random random = new Random();` verwendet und je nach Typ des Attributs wird ein String, Integer, Double oder Boolean Wert zurückgeliefert. Sollte der Typ eines Attributs eine Enumeration sein, wird hier ein zufälliges Literal ausgewählt. Andere Datentypen können nicht verarbeitet werden. Zur Erzeugung der random Werte werden eigene Methoden aufgerufen, daher ist es relativ leicht möglich diese Werterzeugung auch zu adaptieren. Bei den Links wird zwischen *single-valued* und *multi-valued* Links unterschieden, und je nachdem wird der konkrete Wert gesetzt oder der Wert der Liste hinzugefügt.
DLV liefert für die hier gewünschte Lösung auch immer ein Modell mit keiner Instanz zurück. Dieses Modell wird aber nicht erzeugt, da es nur ein leeres Modell wäre. Abschließend wird ein neues `resourceSet` mit der *Uri* des Files für das Modell (Dateiendung: xmi) erzeugt und die einzelnen EObjects werden hinzugefügt. Dadurch wird jedes Modell in einem eigenen XMI File abgespeichert. So kann man die Erzeugung der Modelle mit Hilfe von Dynamic EMF als Text to Model (T2M) Transformation bezeichnen.

[14] `http://www.devx.com/Java/Article/29093/0/page/2`

6.3 OCL - Überprüfung von Constraints[15]

Die Hauptaufgabe der Object Constraint Language (OCL) ist es zusätzliche Bedingungen (Constraints) für Bestandteile (Objects) des Metamodells zu definieren, die nicht modellierbar sind. Die engste Beziehung besteht zwischen OCL und UML, weshalb die volle Funktionalität von OCL auch für UML-Metamodelle nutzbar gemacht werden kann. Da auch MOF eine gemeinsame Basis mit UML hat, können OCL-Ausdrücke auch in diesem Zusammenhang verwendet werden, allerdings in etwas eingeschränkterem Umfang. [71]

Bei der OCL handelt es sich um eine Spezifikationssprache: Das bedeutet, dass die Ausführung/Überprüfung von OCL-Ausdrücken keine direkten Auswirkungen auf das Modell hat. Sie können helfen, den Zustand eines Systems zu spezifizieren, diesen allerdings nicht ändern. Daraus ergeben sich auch die Einsatzgebiete der OCL [71]:

- Als Abfragesprache um bestimmte Informationen aus dem Modell auszulesen
- Zur Spezifikation von Invarianten (Bedingungen, die im Modell jederzeit erfüllt sein müssen)
- Zur Beschreibung von Pre- und Postconditions (Vor- und Nachbedingungen) für Operationen und Methoden (Bedingungen, die vor bzw. nach der Ausführung der Operation/Methode erfüllt sein müssen)
- Zur Beschreibung von Guards (Guards sind Ausdrücke, die mit Zustandsübergängen in State Machines verknüpft sind. Ein OCL-Constraint kann zur Einschränkung solcher Zustandsübergänge verwendet werden.)
- Zur Spezifikation von Bedingungen, die Ausgangswerte ("Initial Values") und abgeleitete Werte ("Derived Values") erfüllen müssen. Erstere werden bei der Erzeugung des Objekts (z.B. eines Attributes) überprüft, während es sich bei letzteren um Invarianten handelt, die angeben, ob ein bestimmter Wert auch zu jeder Zeit demjenigen entspricht, den der OCL-Ausdruck liefert.
- Als Definition: Definitionen werden dazu verwendet um OCL-Ausdrücke zu definieren, die von anderen OCL-Ausdrücken aufgerufen und verwendet werden können.

Diese Einsatzmöglichkeiten machen die Anwendung von OCL-Constraints vor allem im Zusammenhang mit Klassen- und Sequenzdiagrammen besonders interessant [71].

Wichtig beim Einsatz all dieser Konzepte ist die Berücksichtigung der Datentypen, die von den OCL-Ausdrücken zurückgeliefert werden und die im Metamodell spezifiziert sind. So entspricht ein OCL-Ausdruck nur dann den Ansprüchen der Wohlgeformtheit, wenn zum Beispiel im Falle eines Integer-Attributes die zugehörige Invariante auch einen Integer-Wert zurückliefert [71] (ein konkretes Beispiel wäre die Bedingung, dass ein Auto nie weniger als 4 Reifen haben darf; siehe Unterabschnitt 6.3.2).

[15] Verfasser: Thomas Franz, BSc

6.3.1 Syntax von OCL-Ausdrücken

Die Syntax von OCL-Ausdrücken folgt immer einem bestimmten Schema [71]:

1. Zunächst muss der Kontext des Constraints angegeben werden. Dieser beschreibt, für welches Package, welche Klasse, welches Attribut usw. dieser im Metamodell definiert wird, und wird nach dem Schlüsselwort *context* angegeben.

2. Anschließend wird der Typ des Constraints (Invarianten, Precondition, Postcondition usw.) angegeben. Für jede dieser Alternativen gibt es bestimmte Schlüsselwörter. Für die drei oben genannten Beispiele sind das etwa *inv:*, *pre:* und *post:*.

3. Danach folgt der eigentliche OCL-Ausdruck, der beschreibt, was genau bei der Ausführung geprüft wird. Dieser kann von der einfachen Prüfung, ob das zu prüfende Objekt leer ist, bis zu verschachtelten, komplexen Prüfungen reichen, die Daten aus einem oder mehreren anderen Objekten innerhalb des Metamodells abfragen.

Ein weiteres wichtiges Schlüsselwort ist *self*. Dieses stellt, wie der Name bereits andeutet, eine Referenz zum Kontext-Objekt selbst her. Es wird z.B. benötigt um zu prüfen, ob die Attribute der im Kontext definierten Klasse bestimmte Bedingungen erfüllen. Eine vollständige Liste aller Schlüsselwörter findet sich im OMG-Standard der OCL [71, S. 15].

OCL stellt außerdem eigene, vordefinierte Methoden zur Verfügung, wie z.B. *isEmpty()* um zu prüfen, ob das Objekt leer ist. Einige dieser Methoden können für verschiedene Datentypen verwendet werden, andere sind an bestimmte Typen gebunden. So können z.B. die Methode *sum()* zur Summenberechnung von Zahlenwerten oder die Methode *concat()* zur Verknüpfung von Strings verwendet werden. [71]

Andere wichtige vordefinierte Operationen sind *oclIsTypeOf* und *oclIsKindOf*. Diese dienen zur Typprüfung. Mittels ersterer kann überprüft werden, ob der Typ des Rückgabewerts des Constraints dem erwarteten Typ entspricht. *oclIsKindOf* dient zur Überprüfung, ob der Typ des Rückgabewertes des Constraints dem erwarteten Typ oder zumindest einem seiner Sub-Typen entspricht. [80]

Abhängig vom Datentyp können auch bekannte Konzepte für komplexere Abfragen verwendet werden. So können etwa für Zahlenwerte die Grundrechnungsarten (+, -, *, /) und Vergleiche (<, >) ebenso verwendet werden wie logische Operatoren (AND, OR) für Boolean-Werte [80]. Auch if-then-else-Abfragen können in den Constraints zur Anwendung kommen [71].

Neben diesen Operationen, die direkt auf Datentypen angewandt werden, können mit Hilfe von OCL-Ausdrücken auch Operationen auf Attribute der zugehörigen Klasse im Metamodell definiert werden (z.B. *self.typ*; siehe Unterabschnitt 6.3.2). Auch eine Navigation innerhalb des Metamodells, basierend auf Rollennamen, kann mittels OCL umgesetzt werden. Ein wesentlicher Bestandteil von OCL-Constraints ist unter anderem die Einbeziehung von Konstanten. Dabei handelt es sich beispielsweise um bestimmte Zahlenwerte, denen ein bestimmtes Attribut entsprechen muss (bzw. in dessen Wertebereich es liegen muss). [80]

Darüber hinaus können Operationen auf Sets von Objekten in OCL-Ausdrücken angewandt werden. Mit dem Befehl *select()* können einzelne Elemente - in diesem Fall konkrete Instanzen des Metamodell-Objekts - ausgewählt werden. Um ein bestimmtes Sub-Set dieser Instanzen zu erhalten, wird über alle iteriert und die Bedingung geprüft, die im obigen Befehl innerhalb der Klammern angegeben wird. Auch hier kann es sich um eine oder mehrere Bedingungen von beliebiger Komplexität handeln. [71]

Ein im weiteren Verlauf dieses Kapitels wichtiger Aspekt ist der Umstand, dass konkret angegeben werden kann, auf welches Package sich die Constraints beziehen. Dazu werden diese zwischen den Schlüsselwörtern *package* - gefolgt von der genauen Angabe des (Sub-)Packages - und *endpackage* deklariert. [71]

6.3.2 OCL-Beispiel - Teil 1

Das nun folgende Beispiel soll Einblick in die Verwendung von OCL-Constraints bringen. Diese werden anhand eines einfachen Metamodells und eines daraus abgeleiteten Modells veranschaulicht.

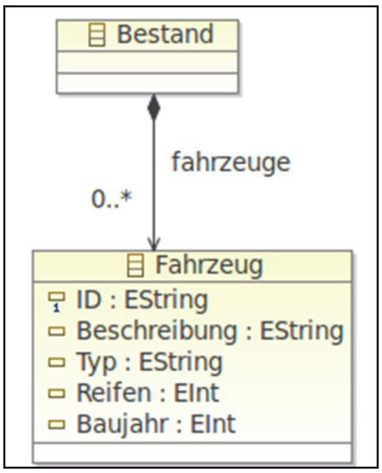

Abbildung 6.3: Ecore-Metamodell

Das Beispiel in Abbildung 6.3 stellt stark vereinfacht einen Fahrzeug-Verleih dar, der eine beliebige Anzahl an Fahrzeugen in seinem Bestand hat. Diese Fahrzeuge werden durch verschiedenste Attribute charakterisiert. Die Darstellung von Metamodellen folgt dem Ansatz möglichst wenige Einschränkungen zu modellieren, um möglichst viele konkrete Anwendungen abzudecken. Das führt allerdings oft dazu, dass für unterschiedliche Anforderungen, die an die konkreten Modelle gestellt werden, andere Einschränkungen gelten sollen. Diese kann man dann mit Hilfe von OCL-Constraints ausdrücken.

Im vorliegenden Beispiel können etwa folgende Constraints formuliert werden (siehe Abbildung 6.4).

```
-- Jedes Fahrzeug muss Reifen haben (aber nicht mehr als 4)
context Fahrzeug
self.Reifen > 0 and self.Reifen <= 4

-- Autos müssen genau 4 Reifen haben, Fahrräder 2 und Motorräder 2 oder 3
context Fahrzeug
inv: if self.Typ = 'Auto' and self.Reifen = 4 then true
        else if self.Typ = 'Fahrrad' and self.Reifen = 2 then true
            else if self.Typ = 'Motorrad' and self.Reifen > 1 and self.Reifen < 4 then true
                else false endif
            endif
        endif

-- Beispiele zur Verwendug von OCL als Abfragesprache
-- Anzahl an Fahrzeugen die aktuell im Bestand sind
context Bestand
self.fahrzeuge -> asSet() -> size()
```

Abbildung 6.4: OCL-Constraints

Der erste Constraint (siehe Abbildung 6.4) prüft, ob ein Fahrzeug überhaupt Reifen hat, und dass kein Fahrzeug mehr als 4 Reifen haben darf (unter der Annahme, dass weder Sattelschlepper noch Radpanzer verliehen werden). Der zweite Constraint bestimmt die Anzahl der Reifen anhand des Fahrzeugtyps. Bei beiden handelt es sich um Invarianten, also um Constraints, die zu jedem Zeitpunkt erfüllt sein müssen. Diese beiden Möglichkeiten sollen auch zeigen, wie einfach oder komplex OCL-Constraints schon in einem "einfachen" Beispiel sein können.

Der letzte Constraint zeigt, wie man OCL-Ausdrücke zu Abfragezwecken verwenden kann. Grundsätzlich sollten OCL-Constraints aber immer einen Boolenwert (also *true*, wenn sie erfüllt, bzw. *false*, wenn sie nicht erfüllt werden) zurückliefern. Dadurch ist es möglich, schnell und einfach Fehler im Modell aufzudecken. Abfragen, wie hier gezeigt, werden oft als Hilfe für die Formulierung von Constraints herangezogen. Das hier angeführte Beispiel liefert die Anzahl an Fahrzeugen, die sich im Bestand befinden. Dazu werden, ausgehend von der Klasse *Bestand*, alle Fahrzeuge (die mit einer Referenz an den Bestand geknüpft sind) in ein Set aufgenommen und anschließend dessen Größe ermittelt. Für beide Beispiele (Abbildung 6.5, Abbildung 6.6) liefert die Abfrage zurück, dass der Bestand jeweils 6 Fahrzeuge umfasst. In letzter Konsequenz würde man hier die Abfrage also mit der Prüfung auf eine fix angegebene Anzahl an Fahrzeugen, die sich im Bestand befinden, zu einer Invariante ausbauen. Man könnte also die Prüfung, ob die Anzahl der Elemente gleich 6 ist, zu einer solchen Invariante ausbauen. Diese Abfrage wird allerdings im weiteren Verlauf nicht weiter beachtet.

Auf Basis des Metamodells ist es jetzt möglich, Modelle zu generieren. Für dieses Beispiel wurden zwei Modelle erzeugt.

Das erste Modell (Abbildung 6.5) erfüllt alle Constraints, wogegen im zweiten Modell (Abbildung 6.6) jeweils ein Fahrzeug eines Typs gegen zumindest einen der zwei Constraints zur Prüfung der Reifenanzahl verstößt.

Zur Modellierung von Metamodellen und zur Generierung von Modellen wird, wie vorher im Abschnitt 6.1 erwähnt, oft die Entwicklungsumgebung *Eclipse* verwendet.

```xml
<?xml version="1.0" encoding="ASCII"?>
<verleih:Bestand xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w
  <fahrzeuge ID="a1" Beschreibung="Toyota" Typ="Auto" Reifen="4" Baujahr="2004"/>
  <fahrzeuge ID="a2" Beschreibung="Opel" Typ="Auto" Reifen="4" Baujahr="2007"/>
  <fahrzeuge ID="m1" Beschreibung="BMW" Typ="Motorrad" Reifen="2" Baujahr="2005"/>
  <fahrzeuge ID="m2" Beschreibung="BMW" Typ="Motorrad" Reifen="3" Baujahr="2003"/>
  <fahrzeuge ID="f1" Beschreibung="KTM" Typ="Fahrrad" Reifen="2"/>
  <fahrzeuge ID="f2" Beschreibung="KTM" Typ="Fahrrad" Reifen="2"/>
</verleih:Bestand>
```

Abbildung 6.5: Modellinstanz 1

```xml
<?xml version="1.0" encoding="ASCII"?>
<verleih:Bestand xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w
  <fahrzeuge ID="a1" Beschreibung="Toyota" Typ="Auto" Reifen="3" Baujahr="2004"/>
  <fahrzeuge ID="a2" Beschreibung="Opel" Typ="Auto" Reifen="4" Baujahr="2007"/>
  <fahrzeuge ID="m1" Beschreibung="BMW" Typ="Motorrad" Reifen="2" Baujahr="2005"/>
  <fahrzeuge ID="m2" Beschreibung="BMW" Typ="Motorrad" Reifen="5" Baujahr="2003"/>
  <fahrzeuge ID="f1" Beschreibung="KTM" Typ="Fahrrad" Reifen="2"/>
  <fahrzeuge ID="f2" Beschreibung="KTM" Typ="Fahrrad"/>
</verleih:Bestand>
```

Abbildung 6.6: Modellinstanz 2

Diese stellt auch eine interaktive Konsole zur Prüfung von OCL-Constraints zur Verfügung. Allerdings muss man hier jedes einzelne Element des Modells manuell durchklicken, um die zugehörigen OCL-Ausdrücke zu prüfen. Ein Beispiel dafür findet sich in Abbildung 6.7.

Da sich dieser Ansatz aber für große Modelle bzw. eine hohe Anzahl an Constraints als unzureichend erweist, stellt sich die Frage, ob und wie man die Prüfung von OCL-Constraints möglichst allgemein automatisieren kann.

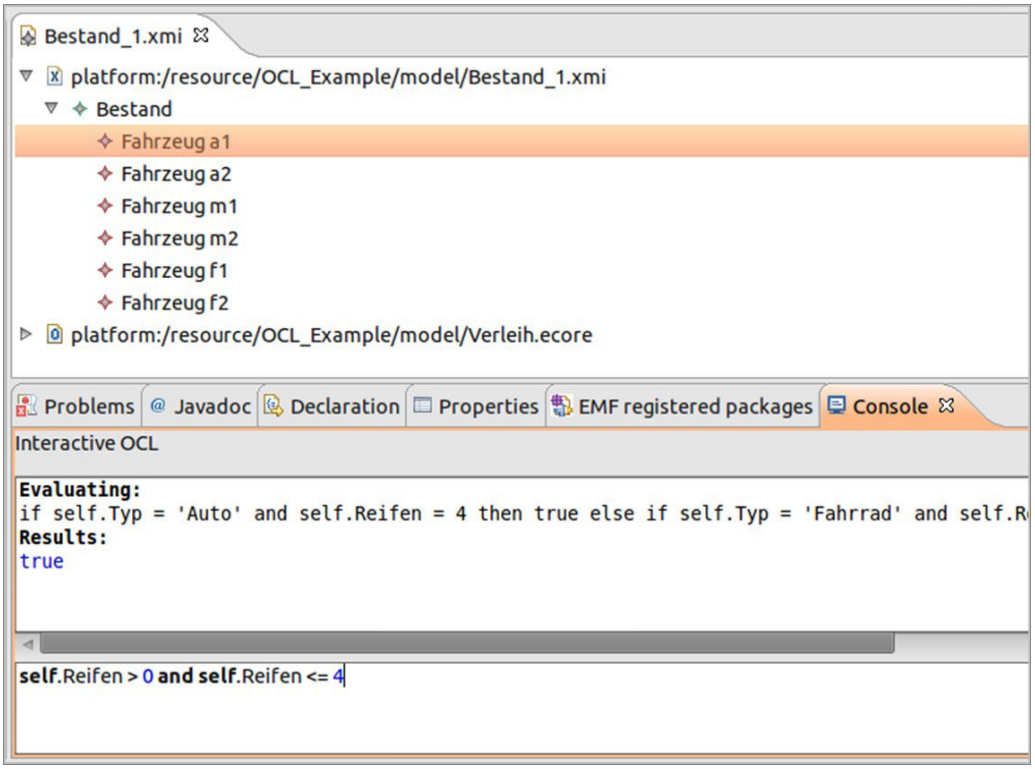

Abbildung 6.7: Verwendung "Interactvie OCL"-Console

6.3.3 OCL-Tool

Das komplette Programm zur automatischen Prüfung von OCL-Constraints wurde in Java entwickelt und basiert im wesentlichen auf dem Ansatz des "Eclipse Community Forums"[16]. Da es aber nicht nur als eigenständiges, ausführbares Programm, sondern auch im Rahmen des WBT-Tools eingesetzt werden soll, muss es bestimmte Anforderungen erfüllen. Daher sind die benötigten Eingabe-Parameter:

1. Dateipfad zum Metamodell
2. Dateipfad zum Modell
3. Dateipfad zu einer Textdatei, in der die OCL-Constraints gespeichert sind
4. Dateipfad zum Zielort, an dem die Datei mit den neuen, bearbeiteten OCL-Constraints gespeichert werden soll

Alle diese 4 Parameter *müssen* angegeben werden um das Programm auch wirklich starten zu können. Die Gründe für die Verwendung der Dateipfade sind einerseits Flexibilität, da das Programm so von jedem beliebigen Ort des PCs aus aufgerufen werden

[16] `http://www.eclipse.org/forums/index.php/t/217359/`, Zuletzt besucht: 29-01-2013

kann und Dateien von jedem beliebigen (anderen) Ort verarbeiten kann. Andererseits kann das Programm so einfacher in das WBT-Tool eingebunden werden (siehe dazu Abschnitt 6.5).

In der ausführbaren .jar-Datei ist die Klasse `OCLEvaluator` enthalten. Diese umfasst den gesamten Code, der für die Prüfung der OCL-Constraints gegen ein Modell erforderlich ist. Die Struktur dieser Klasse stellt sich wie in Abbildung 6.8 dar.

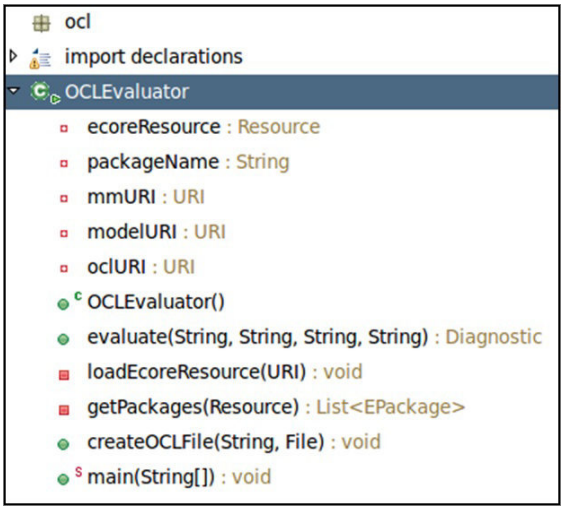

Abbildung 6.8: Struktur `OCLEvaluator.java` in ocl.jar

In der *main-Methode* wird zunächst geprüft, ob auch wirklich alle 4 benötigten Parameter angegeben wurden. Anschließend wird die Methode `evaluate` aufgerufen. In dieser werden die folgenden Schritte ausgeführt:

1. Initialisierung von OCL:

   ```
   CompleteOCLStandaloneSetup.doSetup();
   OCLstdlib.install();
   ```

   Die erste Zeile ermöglicht die Verwendung von OCL als "Standalone", d.h. OCL kann durch Einbinden bestimmter Bibliotheken (.jar-Files) in einem eigenständigen Programm verwendet werden. Die zweite Zeile ermöglicht den Zugriff auf die Standardbibliothek von OCL, sodass gewährleistet wird, dass OCL-Constraints auch wirklich verarbeitet werden können (Interpretation der Syntax).

2. Um von den bestehenden, eingebundenen Programmteilen (.jar-Files) verarbeitet werden zu können, werden die URIs der Dateien erzeugt.

3. Das Metamodell wird in der *ResourceFactoryRegistry* registriert. Dies erfolgt mit dem Dateityp *ecore*, sodass das Programm in der Lage ist Metamodelle mit Dateiendung *.ecore* zu verarbeiten.

4. Aus dem Metamodell wird das oberste Package ausgelesen und dessen Namespace-URI ermittelt. Diese dient der eindeutigen Identifikation des Packages, das in der *PackageRegistry* registriert wird. Das Auslesen aller Packages im Metamodell erfolgt mit Hilfe der Methode `getPackages()`. Dabei ist das oberste Package immer das erste in der Liste.

5. Das oberste Package aus dem Modell auslesen:

```
Resource model = resourceSet.getResource(modelURI,true);
ePackage = model.getContents().get(0).eClass().getEPackage();
```

   Dieser Schritt ist für die beiden folgenden wichtig.

6. Da die Datei, die die OCL-Constraints enthält, gewissen Anforderungen genügen muss um vom Programm verarbeitet werden zu können, wird der Inhalt der angegebenen Datei automatisch überarbeitet und um wichtige Informationen ergänzt. In der ersten Zeile des Dokuments wird das Schlüsselwort *import*, gefolgt vom absoluten Dateipfad zum Ecore-Metamodell, angegeben.
   Danach folgt das Schlüsselwort *package* mit dem Namen des Paketes, das im vorhergehenden Schritt ausgelesen wurde. Wichtig ist, dass nach dem letzten Constraint das Schlüsselwort *endpackage* angegeben wird um anzuzeigen, dass sich alle Constraints auf Bestandteile desselben Packages beziehen. Im konkreten Fall wird davon ausgegangen, dass sich der Package-Kontext auf das implizit im Metamodell enthaltene bezieht.
   Anschließend wird das Originaldokument Zeile für Zeile abgearbeitet. Dabei werden Kommentare verworfen und nur die eigentlichen Constraints in das neu erstellte Dokument übernommen. Wichtig ist hier die Tatsache, dass nach dem Schlüsselwort *inv* der Constraint benannt werden kann. Dies erfolgt in diesem Fall mit dem Wort *Constraint_* gefolgt von durchgehender Nummerierung. Diese Benennung dient in weiterer Folge der genauen Identifikation von fehlgeschlagenen Constraints, also solchen, die *false* zurückliefern.
   Da eines der wichtigsten Schlüsselwörter in der OCL *self* ist und daran auch der Beginn von Constraints erkannt wird, muss auch darauf geachtet werden, dass Wörter in Kommentaren wie “itself”, “himself” oder “herself” nicht missverständlich als Constraints betrachtet werden.

7. Erstellen des URI des neu erstellten Dokuments und Registrieren des in Schritt 6 ausgelesenen Packages. (siehe Abbildung 6.9)

8. Initialisieren des `CompleteOCLEObjectValidator`, der für die Prüfung der Constraints verantwortlich ist. Diesem werden das Package, in dem Constraints geprüft werden sollen, sowie die URI des Dokuments, das diese enthält, übergeben. (siehe Abbildung 6.9)

9. Ähnlich wie zuvor beim Metamodell wird nun die Verarbeitung von Modellen (in diesem Fall .xmi-Modellen) durch einen Eintrag in der Registry ermöglicht. (siehe Abbildung 6.9)

10. Das oberste Element des Modells (das *rootObject*) wird ausgelesen. (siehe Abbildung 6.9)

11. Das in Schritt 6 ausgelesene Package und der `CompleteOCLEObjectValidator` werden im `EValidator` registriert. (siehe Abbildung 6.9)

12. Die eigentliche Prüfung der Constraints wird durchgeführt. Die genauen Informationen über fehlgeschlagene Constraints werden in einem Diagnostic-Objekt als "Kinder" dieses Objekts gespeichert. Diese "Kinder" werden dann auch an die main-Methode zurückgeliefert. (siehe Abbildung 6.9)

```
83        oclURI = URI.createFileURI(save.getAbsolutePath());
84
85        EPackage.Registry.INSTANCE.put(p.getNsURI(), ePackage);
86
87        //initialize OCL-Validator
88        CompleteOCLEObjectValidator myValidator = new CompleteOCLEObjectValidator(ePackage, oclURI);
89
90        Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(
91                Resource.Factory.Registry.DEFAULT_EXTENSION,
92                new XMIResourceFactoryImpl());
93
94        EObject rootObject = model.getContents().get(0);
95
96        EValidator.Registry.INSTANCE.put(ePackage, myValidator);
97
98        Diagnostic diagnostics = Diagnostician.INSTANCE.validate(rootObject);
```

Abbildung 6.9: Auszug aus dem Code von `OCLEvaluator.java` betreffend die Schritte 7-12 zur Prüfung der OCL-Constraints

In der main-Methode folgt die abschließende Prüfung, ob alle Constraints erfüllt wurden (die Liste der "Kinder" des Diagnostic-Objekts ist leer) oder ob es fehlgeschlagene Constraints gab. Abhängig davon wird auch eine entsprechende Ausgabe erzeugt:

- "All Constraints for model are valid!", wenn alle Constraints erfüllt sind
- "ERROR: Constraints are invalid!", wenn zumindest ein Constraint nicht erfüllt ist

Bei beiden handelt es sich um manuell erstellte Ausgaben. Im zweiten Fall wird vor dieser Meldung noch durch die Liste der "Kinder" des Diagnostic-Objekts iteriert und deren textuelle Repräsentation ausgegeben (siehe dazu auch Unterabschnitt 6.3.4).

Wie aus der Beschreibung der Funktionsweise des Programms bereits teilweise ersichtlich, müssen bei dessen Anwendung einige Einschränkungen beachtet werden:

- Es können nur Ecore-Metamodelle geladen werden (siehe Schritt 3).
- Bei der Erstellung der bearbeiteten Datei, die die OCL-Constraints enthält, werden nur Invarianten berücksichtigt (siehe Schritt 6).
- Es können nur Modelle mit zugrunde liegender XMI-Struktur geladen werden (siehe Schritt 9).
- Constraints können nur für ein bestimmtes Package geprüft werden. Es wird davon ausgegangen, dass es abgesehen vom implizit existierenden Package - das alle Elemente des Metamodells enthält - keine explizit im Metamodell enthaltenen Packages gibt. Dies wirkt sich natürlich auch auf die Existenz von Packages im konkreten Modell aus.

Das vorliegende Programm stellt allerdings nicht die einzige Alternative zur Prüfung von OCL-Constraints dar. Vor diesem wurden bereits andere Ansätze auf ihre Tauglichkeit bezüglich einer automatischen Validierung von OCL-Constraints geprüft. Nähere Beschreibungen zu diesen Alternativen finden sich im Kapitel 9.

6.3.4 OCL-Beispiel - Teil 2

Dieser Abschnitt ist die Fortsetzung von Unterabschnitt 6.3.2 OCL-Beispiel - Teil 1. Hier wird auf dasselbe Beispiel zurückgegriffen, allerdings wird nun der Fokus auf die Verwendung des im vorangegangenen Abschnitt vorgestellten Programms gelegt. Abbildung 6.10 und Abbildung 6.11 zeigen die Ergebnisse, die das Tool zurückliefert.

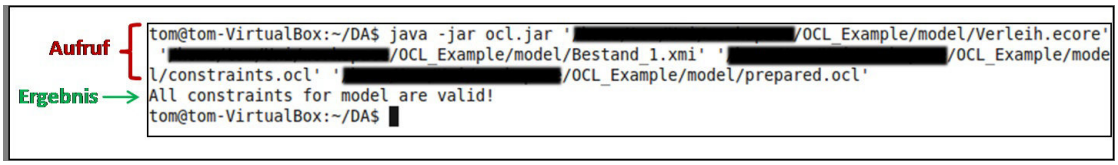

Abbildung 6.10: Verwendung des OCL-Tools: Beispiel mit erfüllten OCL-Constraints

Wie auch aus dem Modell ersichtlich, liefert das Programm für das erste Beispiel die Bestätigung, dass alle Constraints erfüllt sind.

Für das zweite Beispiel liefert das Programm den Hinweis, dass nicht alle Constraints erfüllt sind, sowie eine genaue Auflistung darüber, welcher Constraint an welcher Stelle im Modell fehlgeschlagen ist (siehe Markierung in Abbildung 6.11). Insgesamt existieren fünf Fehlerquellen im Modell. Da alle diese Fehlermeldungen gleich aufgebaut sind, wird anhand der letzten illustriert, wo sich die Information darüber findet, welcher der Constraints fehlgeschlagen ist. Dies verdeutlicht auch den zusätzlichen Nutzen, den eine eindeutige Benennung der Constraints bei der Aufbereitung der Datei mit den ursprünglichen Constraints liefert.
Außerdem wird in diesen Fehlermeldungen angegeben, in welchem Modell (in diesem

Fall “Bestand_2.xmi”) der Constraint nicht erfüllt wurde und für welches Objekt (angegeben durch “@fahrzeuge.5”). Folgt man im Modell also der Referenz “fahrzeuge” zum 6. Eintrag (das erste Objekt hat den Index 0) findet man exakt das Objekt, das den Constraint verletzt.

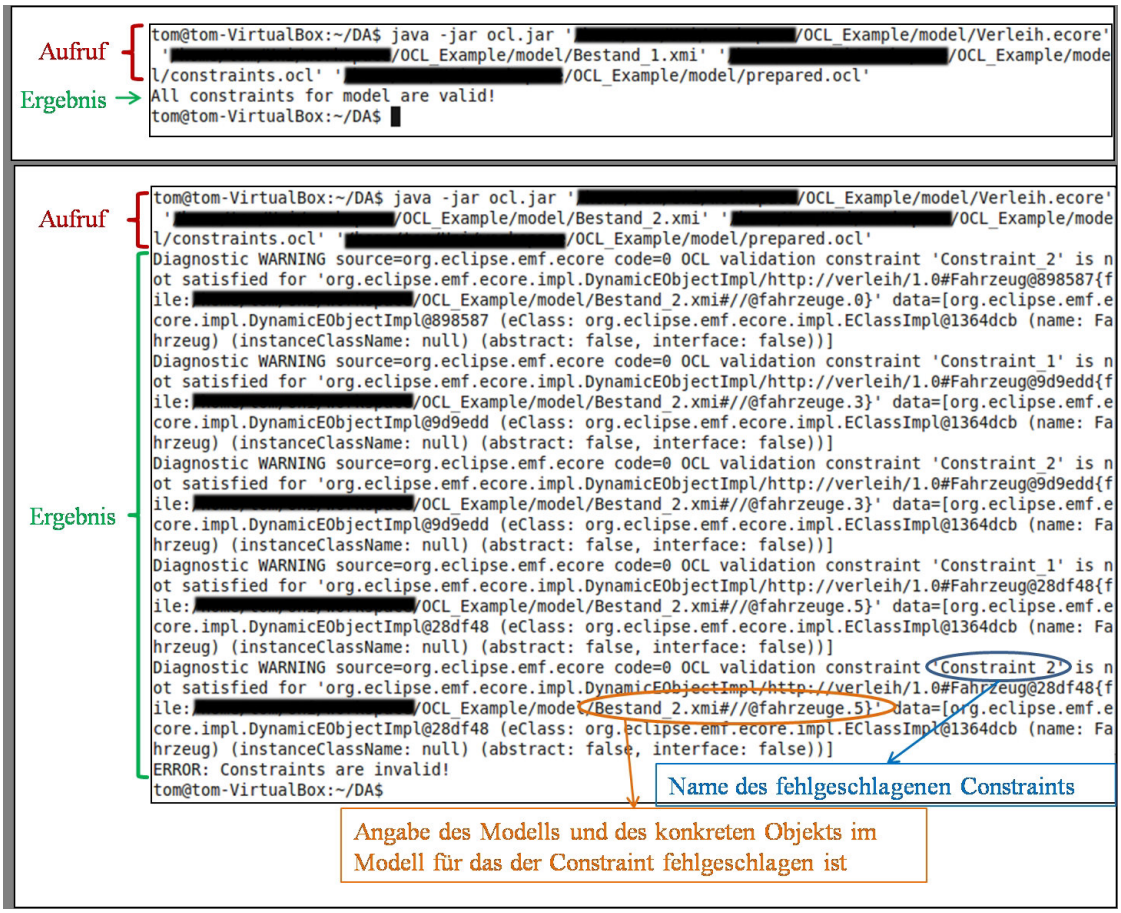

Abbildung 6.11: Verwendung des OCL-Tools: Beispiel mit nicht erfüllten OCL-Constraints

Für das eigentliche Ziel der Arbeit ist nur wesentlich, ob das Modell zumindest einen OCL-Constraint nicht erfüllt. Daher stellt die detaillierte Auflistung der Fehlerquellen nur einen Zusatznutzen für die eigenständige Verwendung zur Constraint-Prüfung dar. Auf eine automatisierte Aufbereitung dieser Fehlermeldungen - vor allem im Hinblick auf eine bessere Auffindbarkeit der konkreten Objekte, die einen oder mehrere Constraints verletzen - wurde daher verzichtet.

6.4 ATL - Durchführung der Modelltransformation[17]

Die Atlas Transformation Language (ATL)[18] ist eine Modelltransformationssprache. Eine solche Transformation wird auch als *Modul* bezeichnet. Daneben besteht die Möglichkeit mit ATL auch Abfragen zu generieren oder die Transformation mit Hilfe von *Bibliotheken* zu fragmentieren.

Hauptaufgabe ist die Erzeugung eines Zielmodells aus einem Quellmodell. Für beide Modelle wird dabei gefordert, dass sie gültige Instanzen ihrer Metamodelle sind. Die Sprachkonzepte der ATL stellen dabei selbst ein Metamodell für die Transformation dar. Diese Konzepte sind anhand des konkreten Beispiels "FamiliesToPersons" von der Eclipse ATL Website[19], das auch im weiteren Verlauf verwendet wird, dargestellt. Für die Transformation sind drei Modelle wichtig: Die Beispiel-Familie als Quellmodell, die Transformation selbst, sowie das Zielmodell ("sample-Persons.xmi"), das erzeugt werden soll. Alle Modelle müssen bestimmte Anforderungen erfüllen, die in ihren Metamodellen definiert sind. Die drei Metamodelle bauen auf der Semantik des Ecore-Metametamodells auf, was die folgende Abbildung 6.12 zeigt. [52]

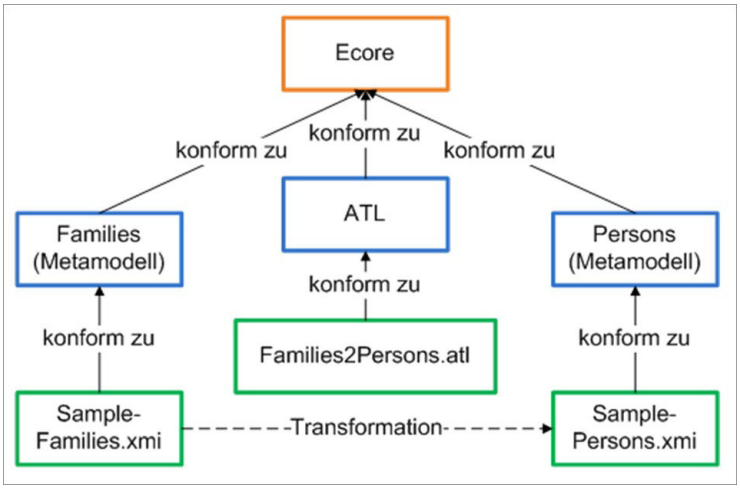

Abbildung 6.12: ATL-Transformation
(Quelle: `http://wiki.eclipse.org/ATL/Concepts`, Zuletzt besucht: 10-03-2013)

[17] Verfasser: Thomas Franz, BSc

[18] `http://wiki.eclipse.org/MMT/Atlas_Transformation_Language_(ATL)`, Zuletzt besucht: 06-02-2013

[19] `http://www.eclipse.org/atl/documentation/basicExamples_Patterns/`, Zuletzt besucht: 06-02-2013

6.4.1 Syntax von ATL-Transformationen

Eine ATL-Transformation besteht aus den folgenden Elementen [20]:

- Einer sogenannten *Header-section*
- Einem Abschnitt, in dem zusätzliche Bibliotheken eingebunden werden können
- Einem Abschnitt, in dem zusätzliche Funktionen (*Helper*) definiert werden
- Einem Abschnitt, in dem die eigentlichen Transformationsregeln definiert werden

In der Header-section wird das aktuelle Modul benannt. Die angegebene Benennung muss mit dem Namen der .atl-Datei, die die Transformation enthält, übereinstimmen. Anschließend werden Quell- und Zielmetamodell definiert. Für die Transformation von "FamiliesToPersons" sieht die Header-section folgendermaßen aus:

```
module Families2Persons;
create OUT : Persons from IN : Families;
```

Durch diesen Befehl wird angegeben, dass eine Instanz von Familie in eine Instanz von Person transformiert werden soll. *OUT* bzw. *IN* stellen an dieser Stelle Platzhalter für die konkreten Modelle dar, die später bei der Ausführung erzeugt bzw. geladen werden. Ist die Anzahl an Modellen, die transformiert werden sollen, größer, so muss eine entsprechende Anzahl an Platzhaltern definiert werden. Zu beachten ist hier, dass die Anzahl an Zielmodellen mit der Anzahl an Quellmodellen übereinstimmen muss, da ATL nicht in der Lage ist, mehr als ein Zielmodell aus einem Quellmodell zu generieren.

In ATL ist es möglich, Transformationen zu fragmentieren. Dadurch können Teile von Modelltransformationen wiederverwendet werden. Um solche Bibliotheken einzubinden wird der Name der entsprechenden Datei nach dem Schlüsselwort *uses* angegeben. Hier ist allerdings zu beachten, dass diese Bibliotheken nur *Helper* enthalten können.

Helper stellen ein Konzept der imperativen Programmierung dar, d.h. sie können mit Methoden, wie sie aus objektorientierten Programmiersprachen, wie etwa Java, bekannt sind, verglichen werden. Das hat den Vorteil, dass auch innerhalb der Transformation oft benötigte Teile nur ein Mal programmiert werden müssen um sie anschließend an verschiedensten Stellen wiederholt aufrufen zu können.

```
helper Kontext Kontext_Typ def : helper_Name(Parameter) :
Rückgabetyp = Ausdruck;
```

Ein Helper wird über das gleichlautende Schlüsselwort definiert. Anschließend kann ein konkreter Kontext angegeben werden (z.B. eine Klasse eines Modells oder ein bestimmter Datentyp). Auf diese Weise kann genau festgelegt werden, für welche Elemente

[20] `http://wiki.eclipse.org/ATL/User_Guide_-_Overview_of_the_Atlas_Transformation_Language`, Zuletzt besucht: 10-03-2013

der Helper aufgerufen wird. Wird der Kontext nicht explizit angegeben, so wird statt dessen das Modul (die gesamte Transformation) als Kontext angenommen. Nach dem Schlüsselwort *def* folgt die Angabe des Namens. An dieser Stelle können in Klammern auch Parameter an den Helper übergeben werden. Diese müssen immer mit einem Parameternamen sowie deren Datentyp - durch *:* getrennt - deklariert werden. Für jeden Helper muss außerdem ein Rückgabetyp definiert werden, da kein Helper ohne Rückgabewert existieren darf. Dieser Rückgabewert ist das Ergebnis eines bestimmten ATL-Ausdrucks. Um komplexere Ausdrücke definieren zu können, können in der ATL auch Konzepte ähnlich denen der OCL eingesetzt werden (um z.B. bestimmte Attribute durch Navigation durch das Quell-Metamodell abfragen zu können). [31] [52]

Daneben können auch Attribute definiert werden, die ähnlich wie Helper definiert werden, allerdings ohne Angabe der Parameter. Diese Attribute können ebenfalls komplexere Aufgaben erfüllen oder lediglich konstante Werte zurückliefern.

Das wichtigste Konzept in der ATL sind die Regeln. Diese sind Konzepte aus der deklarativen Programmierung, die auf der Beschreibung von Problemen basieren. Im Zusammenhang mit Modelltransformationen ist dies die Überführung der Konzepte (häufig Klassen) der Quellmodelle in solche der Zielmodelle. Die ATL kennt drei Arten von Regeln [31] [52]:

1. matched rules
2. lazy rules
3. called rules

Jede Regel wird durch ihren Namen eindeutig identifiziert. Dieser wird nach dem Schlüsselwort *rule* angegeben. In der Regel folgt die Benennung der Konvention, dass angegeben wird, welche Konzepte des Quellmodells in welche Modelle des Zielmodells überführt werden (z.B. *Member2Male* um ein Familienmitglied in eine männliche Person zu transformieren; siehe auch Unterabschnitt 6.4.3).

Innerhalb von *matched rules* werden diese Konzepte durch die Schlüsselwörter *from* - für solche des Quellmodells - und *to* - für solche, die im Zielmodell erzeugt werden sollen - gekennzeichnet. Für beide muss ein Typ definiert werden, der ein Konzept des zugehörigen Metamodells darstellt. Basierend auf diesen Angaben wird während der Ausführung geprüft, ob eine bestimmte Regel ausgeführt wird, oder nicht. Soll z.B. eine Instanz der Klasse *Member* aus dem Metamodell *Families* verarbeitet werden, so wird geprüft, welche Regeln gemäß ihrer Angaben in der *from*-Deklaration dafür in Frage kommen, und diese werden dann ausgeführt. Im konkreten Beispiel wird also aus einem Member eine Person. Die in der *Member*-Instanz gespeicherten Parameter (der Vor- und der Nachname) werden ebenfalls in die neu erzeugte *Male*-Instanz übernommen. [31] [52]

Innerhalb eines Blocks nach dem Schlüsselwort *using* können eigene Variablen definiert werden, die nur innerhalb der Regel sichtbar sind.

Optional können innerhalb der Regeln auch sogenannte *do-Blöcke* mit dem Schlüsselwort *do* definiert werden. Hier können weitere Operationen definiert werden, die dann

während der Modelltransformation ausgeführt werden (wenn auch die Regel ausgeführt wird). Dies wird oft eingesetzt, wenn bestimmte Attribute nicht direkt von der Quell- in die Zielinstanz übernommen werden können, sondern auf eine bestimmte Weise bearbeitet werden müssen. [31]

Lazy rules gleichen *matched rules*, werden aber im Gegensatz zu diesen nur von anderen Regeln aus aufgerufen. Sie werden durch die Angabe der Schlüsselwort-Kombination *lazy rule* deklariert. [31]

Called rules sind Regeln ohne Angabe eines Konzepts aus dem Quellmodell. Sie sind eine Mischung aus Regel und Helper, da für diese Art von Regeln Übergabeparameter definiert werden können. Durch den Aufruf einer solchen Regel wird eine neue Instanz einer Klasse des Zielmodells erzeugt. Die übergebenen Parameter sind häufig Attribute dieser neuen Instanz. [31]

6.4.2 ATL-Beispiel - Teil 1

In diesem Abschnitt wird die Durchführung einer ATL-Transformation in Eclipse illustriert. Dabei wird auf das einfache Beispiel "FamiliesToPersons" von der Eclipse ATL Website[21] zurückgegriffen. Hierbei handelt es sich um ein ATL-Projekt, das direkt in Eclipse geladen werden kann. In diesem befinden sich folgende Dateien:

- Families.ecore (Input-Metamodell)
- Persons.ecore (Output-Metamodell)
- sample-families.xmi (Input-Modell)
- sample-persons.xmi (Output-Modell)
- Families2Persons.atl (ATL-Transformation)
- Families2Persons.asm (kompilierte ATL-Transformation)
- Families2Persons.launch

Die letztgenannte Datei enthält alle Informationen, die zur Durchführung der ATL-Transformation erforderlich sind (die *Launch*-Konfiguration). Dabei handelt es sich hier um eine speziell für diesen Zweck erzeugte Datei. In Eclipse werden üblicherweise die Informationen, die zur Durchführung der ATL-Transformation benötigt werden, in den sogenannten *Run Configurations* angegeben. Die hier angegebenen Informationen werden aber nicht dauerhaft gespeichert. Eine solche längerfristige Speicherung ist mit einer *.launch-Datei* möglich. Existiert diese Datei, werden die darin enthaltenen Informationen direkt in die *Run Configurations* übernommen (siehe Abbildung 6.13).

Sind alle Parameter korrekt angegeben, kann die Transformation gestartet werden. Dieser Vorgang muss allerdings manuell erfolgen. Ändert man im konkreten Fall den

[21] `http://www.eclipse.org/atl/documentation/basicExamples_Patterns/`, Zuletzt besucht: 06-02-2013

Namen des Output-Modells, wird diese Datei automatisch im Projekt erzeugt. Diese hat exakt denselben Inhalt wie die mitgelieferte Datei “sample-Persons.xmi”.

Der Nachteil bei dieser Art der Transformationsausführung ist, dass jeweils nur ein Input-Modell in ein Output-Modell überführt werden kann. Soll eine höhere Anzahl an unterschiedlichen Modellen eines Metamodells transformiert werden, müssen die Parameter manuell angepasst werden. Für ein effizientes und vor allem auch automatisiertes Testen einer ATL-Transformation ist allerdings ein anderer Ansatz erforderlich.

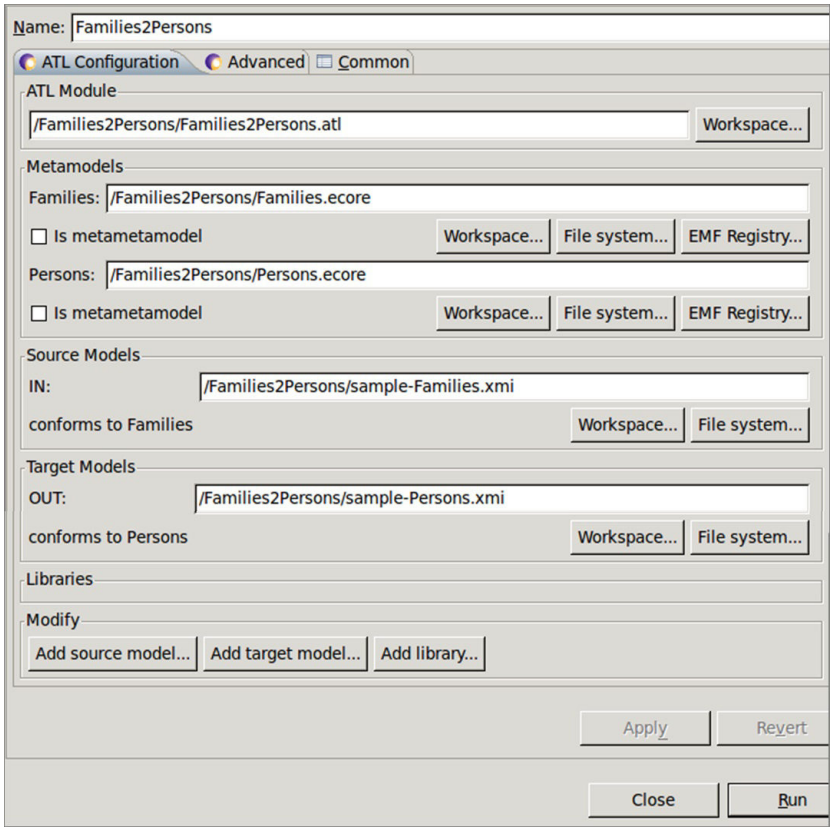

Abbildung 6.13: ATL Run Configuration aus Eclipse

6.4.3 White-Box Testen

Ein wichtiges Unterscheidungsmerkmal zwischen dem Black- und White-Box Testen ist, dass bei letzterem der innere Aufbau des zu testenden Objekts bekannt ist. Dadurch ist es auch möglich, bestimmte Metriken - Überdeckungsgrade für Pfade, Entscheidungen usw. - zu berechnen. Selbiges trifft auch für Modelltransformationen zu [61].

Eine Möglichkeit solche Überdeckungsgrade zu berechnen basiert auf der Ausnutzung der Sprachkonzepte der ATL. Drei Kennzahlen, die so ermittelt werden können, sind [61]:

- Regelabdeckung: Die Anzahl der tatsächlich ausgeführten Regeln verglichen mit der Anzahl aller definierten Regeln.
- Anweisungsabdeckung: Gibt an, wie viel Prozent der möglichen Anweisungen ausgeführt werden.
- Bedingungsabdeckung: Für jede Entscheidung wird *wahr* oder *falsch* als Ergebnis geliefert. Daraus wird abgeleitet, zu welchem Prozentsatz die möglichen Entscheidungsausgänge abgedeckt werden (siehe Unterabschnitt 3.1.1.3).

Diese Parameter sind für die ATL unterschiedlich schwer zu ermitteln. Die vorliegende Arbeit beschäftigt sich mit der Ermittlung der Regelabdeckung. Da der Testprozess automatisiert ablaufen soll, muss auch für das Problem der Aufzeichnung der ausgeführten Transformationsregeln eine adäquate Lösung gefunden werden. Diese sieht die Einführung eines zentralen Parameters vor, in dem die Informationen über

- das Objekt aus dem Quellmodell, das transformiert wird,
- das Objekt aus dem Zielmodell, das erzeugt wird,

gespeichert werden. Dabei werden die Typinformationen dieser beiden Objekte verwendet. Wird zum Beispiel ein Personen-Objekt aus dem Quellmodell transformiert, so wird die Information darüber, dass es sich um ein solches Personen-Objekt handelt, gespeichert; selbiges gilt für das Objekt des Zielmodells, das erzeugt wird. Dieser Parameter wird in die Originaldatei der Transformation eingefügt, bevor diese weiter verarbeitet wird (siehe dazu auch Unterabschnitt 6.4.5.2).

Diese Lösung hat zwei Vorteile:

1. Bei der Benennung von Regeln wird der Konvention gefolgt, dass deren Namen beschreiben sollten, welche Transformation ausgeführt wird.
2. Abseits dieser Konvention wird so durchgängig dokumentiert, welche Konzepte des Quellmodells in welche Konzepte des Zielmodells überführt werden, auch wenn die Regel, die dies beschreibt, anders benannt wird.

Dadurch wird eine durchgängige Dokumentation der Modelltransformation ermöglicht und es kann festgestellt werden, wo Fehler aufgetreten sind, falls das Zielmodell fehlerhaft sein sollte.

Ein wichtiger Messwert ist die *Bedingungsabdeckung*. Viele dieser Bedingungen werden aber in Helper definiert, was deren Aufzeichnung im Vergleich zur Regelabdeckung zu einer größeren Herausforderung macht. Dies ist ein Aspekt, der in der weiteren Entwicklung des vorliegenden Ansatzes verfolgt werden sollte, da hier oft wesentliche Entscheidungen für die Ausführung der Transformation und deren Ergebnis getroffen werden.

Ein Beispiel dafür ist die in diesem Abschnitt betrachtete Transformation “FamiliesToPersons”. Bei dieser werden Familienmitglieder in männliche und weibliche Personen transformiert. Eine wichtige Rolle bei der Überprüfung des Geschlechts spielt hier

ein Helper, der über die auszuführende Transformationsregel entscheidet. Ist dieser fehlerhaft, wirkt sich dies auch auf das Ergebnis der Transformation aus.

6.4.4 ATL-Programmierschnittstelle

ATL stellt eine Programmierschnittstelle (API) zur Verfügung, die *Core API*[22] (siehe auch Abbildung 6.14). Deren Konzepte können dazu verwendet werden, eine ATL-Transformation mit Hilfe von Java-Code auszuführen.

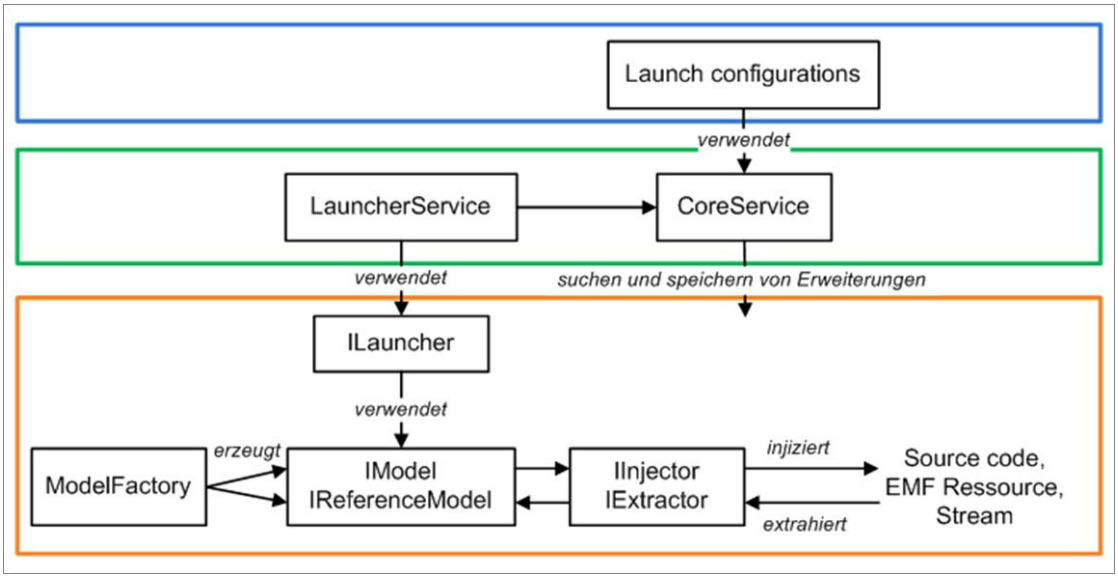

Abbildung 6.14: Grafische Darstellung der ATL Core API
(Quelle: `http://wiki.eclipse.org/ATL/Developer_Guide#Core_API`, Zuletzt besucht: 03-04-2013)

Diese Schnittstelle stellt zwei Services zur Verfügung:

1. Das *CoreService* wird für die Suche nach konkreten Implementierungen in den Eclipse-Erweiterungen oder einem internen Speicher herangezogen. Da die *ATL Core API* nur Schnittstellen (sogenannte Interfaces) zur Verfügung stellt, müssen konkrete Umsetzungen der benötigten Komponenten geladen werden um die Funktionalität auch wirklich nutzen zu können. So gibt es solche Implementierungen z.B. speziell für EMF um (Meta-)Modelle, die in der Modellierungssprache *Ecore* entwickelt wurden, verarbeiten zu können.

2. Das *LauncherService* ermöglicht die Ausführung von Transformationen mit Hilfe der sogenannten *Launch configurations*. Bei diesen handelt es sich um eine Datei, in der die Parameter zur Ausführung angegeben werden (zur Verwendung solcher

[22] `http://wiki.eclipse.org/ATL/Developer_Guide#Core_API`, Zuletzt besucht: 03-04-2013

Konfigurationsdateien siehe z.B. Unterabschnitt 6.4.2). Diese Parameter definieren welche Komponenten (Metamodelle und Quellmodell, evtl. benötigte Bibliotheken) geladen werden sollen, sowie weitere Parameter um eine Feineinstellung der Modelltransformation vornehmen zu können.

Neben diesen beiden Services gibt es noch weitere Komponenten, die zur Durchführung einer Transformation initialisiert und denen konkrete Daten zur Verfügung gestellt werden müssen. Dazu zählt etwa die *ModelFactory*, die für die Erzeugung der (Referenz) Modelle verantwortlich ist. Hier gibt es für verschiedenste Modellierungssprachen eigene Umsetzungen (z.B. für UML oder EMF).

Bei den weiteren Komponenten handelt es sich um *Interfaces*, die lediglich bestimmte Teile einer konkreten Umsetzung vorgeben (z.B. welche Methoden diese enthalten sollten). Interfaces werden häufig zur Beschreibung und vor allem Vereinheitlichung von Schnittstellen innerhalb beziehungsweise zwischen Programmen oder Teilen eines Programms verwendet.

Das `IModel` stellt eine, speziell für die Verwendung in ATL adaptierte, Repräsentation von konkreten Modellen dar. Konkrete Instanzen von `IModel` werden bei der Umsetzung in Java als Speicher für das Quell- und das generierte Zielmodell erzeugt.

Das `IReferenceModel` ist eine spezielle Version von `IModel`, das für die Verarbeitung von Metamodellen verwendet wird.

Der `IInjector` dient zum Laden von Modellen und Metamodellen, während der `IExtractor` am Ende der Transformation für das Auslesen beziehungsweise Erzeugen des Zielmodells verantwortlich ist. Mit Hilfe des `IInjectors` werden etwa das Quellmodell sowie die beiden Metamodelle über die entsprechenden Schnittstellen (`IModel` und `IReferenceModel`) geladen.

Der `ILauncher` ist für die eigentliche Ausführung der Modelltransformation verantwortlich. Auch hier gibt es eine, für die Transformation von EMF-Modellen optimierte, konkrete Umsetzung.

Alle diese Bestandteile spielen bei der Transformationsausführung zusammen. Dabei hängt deren Einsatz auch von der Art der Parameter ab. Einige, wie zum Beispiel die Modelle, werden mit einem `Injector` geladen und stehen zur Verfügung wogegen andere wie solche, die in einer Konfigurationsdatei angegeben werden, mit Hilfe eines Datenstroms zur weiteren Verarbeitung eingelesen werden müssen. Letzteres wird vor allem beim Kompilieren der Transformation eingesetzt. Da ATL also auch in der Lage ist, unterschiedliche Parameterwerte zu verarbeiten, gibt es dementsprechend auch unterschiedliche Möglichkeiten eine Transformation anzustoßen.

Eine Möglichkeit ist die manuelle Ausführung mit Hilfe des Eclipse-Plugins, das ganz oder teilweise durch Verwendung einer Konfigurationsdatei automatisiert werden kann (siehe Unterabschnitt 6.4.2). Ein weiterer Ansatz ist der Einsatz eines Java Programms, das in der Lage ist mehrere Transformationen automatisiert durchzuführen (siehe Unterabschnitt 6.4.6). Für letzteres wurden im Zuge dieser Arbeit - wie bereits angedeutet - die EMF spezifischen Umsetzungen der oben beschriebenen Konzepte verwendet.

6.4.5 ATL-Tool

Dieser Abschnitt beschreibt die Entwicklung und Funktionalität des Programms zur automatischen Durchführung von Modelltransformationen in ATL. Zuerst wird der Ansatz, auf dem die spätere Entwicklung basiert, kurz erläutert. Anschließend wird die Ausführung des Programms Schritt für Schritt erläutert und zwar beginnend bei der Beschreibung der Parameter, die zur korrekten Ausführung erforderlich sind, bis zur Ausgabe der Ergebnisse.

6.4.5.1 Ansatz (ATL-Plugin)

Das Programm wurde in Java geschrieben. Die dazu verwendete Entwicklungsumgebung war *Eclipse*. Daher konnte ein darin enthaltenes Plugin, nämlich das *ATL-Plugin*[23], verwendet werden, das es ermöglicht eine bestimmte Modelltransformation in Java Code zu überführen.

Der aktuelle Abschnitt beschreibt genauer, wie der Ansatz mit Hilfe dieses Plugins verwirklicht wurde. Dadurch soll dem interessierten Leser ermöglicht werden, die Umsetzung nicht nur detaillierter sondern auch praxisnäher, durch eigene Ausführung, nachvollziehen zu können. Für das Verständnis der Funktionsweise des ATL-Tools ist dies allerdings nicht zwingend erforderlich.

Dieses Plugin erzeugt ein neues Projekt, das einen Ordner *src* enthält. In diesem Ordner finden sich zwei Packages. Das erste Package enthält eine Java Klasse `Activator.java`, die zur Kontrolle des Plugin Lebenszyklus dient und nicht weiter benötigt wird, während das zweite Package mit Endung ".files" folgende 4 Dateien enthält:

- Eine Java Klasse, die nach dem der ATL Datei benannt ist,
- Die ATL Datei selbst
- Die kompilierte ATL Datei (mit der Dateiendung ".asm")
- Eine Datei mit Zusatzinformationen (mit der Dateiendung ".properties")

Um diese konkrete Modelltransformation mit Hilfe der Java Klasse ausführen zu können, müssen noch einige Änderungen im Code vorgenommen werden:

1. Löschen der ersten beiden if-else-Abfragen in der Methode `getFileURL()`. Nur der Inhalt des else-Blocks der zweiten Abfrage wird nicht gelöscht.
2. Löschen der if-else-Abfrage in der main-Methode.
3. Ersetzen der beiden *args[]*-Parameter im Aufruf der Methode `loadModels()` in der main-Methode durch Angabe der konkreten Modelle (als Strings).

[23] `http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Tools#ATL_Plugins`, Zuletzt besucht: 06-02-2013

4. Ersetzten des *args[]*-Parameters im Aufruf der Methode `saveModels()` in der main-Methode durch Angabe eines konkreten Modells, das erzeugt werden soll (als String).

Wird das Package in ein anderes Projekt kopiert, sollte beachtet werden, dass auch die Pfade zu den Metamodellen, die sich in der ".properties" Datei befinden, dementsprechend angepasst werden müssen.

Nach der Durchführung dieser Änderungen kann die Java Klasse gestartet werden und die Modelltransformation wird ohne Benutzerinteraktion ausgeführt. Zur Entwicklung des Tools wurde dieser Vorgang zweimal für verschiedene ATL Modelltransformationen[24] durchgeführt:

- Families To Persons
- Tree To List

Der Unterschied zwischen diesen beiden Transformationen ist, dass letzteres auch auf ATL Libraries zurückgreift. Dieser Unterschied macht sich vor allem im ".properties" File bemerkbar.

Um eine weitgehende allgemeine Verwendung dieses Ansatzes zu ermöglichen, wurden die so erhaltenen Ergebnisse in einen gemeinsamen Ansatz aufgenommen. Um dieses Ziel zu erreichen, waren weitere, umfangreichere Änderungen im Code erforderlich. Das Ergebnis wird im nächsten Unterabschnitt 6.4.5.2 beschrieben.

6.4.5.2 Funktionalität

Da auch das ATL-Tool ein integraler Bestandteil des WBT-Tools sein soll, musste die ursprünglich generierte main-Methode in die Methode `runTransformation()` umgewandelt werden. Dies machte auch die Einführung einer neuen main-Methode erforderlich, die in der Lage ist die folgenden Parameter zu verarbeiten:

1. Dateipfad zum Input Modell
2. Angabe des Pfades zum Ort, an dem das Output Modell gespeichert werden soll
3. Dateipfad zum Input Metamodell
4. Dateipfad zum Output Metamodell
5. Pfad zur Datei, die die ATL Transformation enthält
6. Angabe des Pfades zum Ort, an dem die Datei mit der neuen, erweiterten ATL Transformation gespeichert werden soll
7. Angabe des Pfades zum Ort, an dem die Datei, die die Informationen über die ausgeführten Transformationsregeln enthält, gespeichert werden soll

[24] `http://www.eclipse.org/atl/documentation/basicExamples_Patterns/`, Zuletzt besucht: 06-02-2013

8. Pfad zur Datei, die die Bibliothek (ATL Library) enthält (kann optional angegeben werden)

Der gesamte Code zur automatisierten Ausführung von ATL-basierten Modelltransformationen findet sich in der Klasse `ATLRunner.java`. Die Struktur der Klasse stellt sich wie in Abbildung 6.15 dar, wobei aus Gründen der Übersichtlichkeit hier nur deren Methoden angegeben sind.

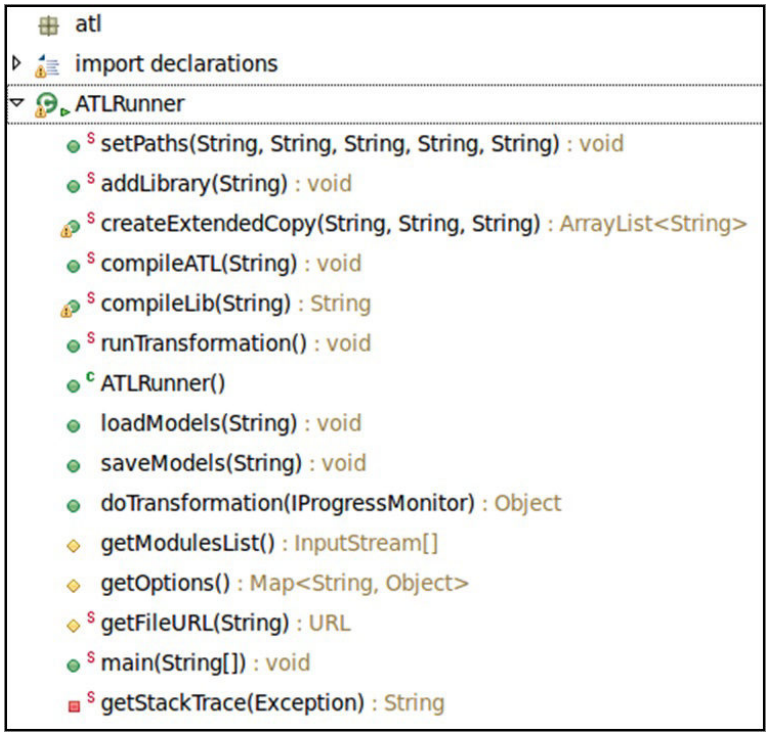

Abbildung 6.15: Struktur `ATLRunner.java` in atl.jar

Zunächst wird überprüft, ob alle benötigten Parameter (die ersten sieben der oben angeführten Parameterliste) auch wirklich angegeben wurden. Anschließend werden die folgenden Schritte ausgeführt:

1. Prüfung, ob bereits eine kompilierte ATL Datei vorhanden ist (falls dies so sein sollte, gehe weiter bei Schritt 5).

2. Sollte dies nicht der Fall sein, wird eine erweiterte Kopie der ursprünglichen ATL Datei erzeugt.
   Zuerst wird die Original-Datei Zeile für Zeile eingelesen und bis zum Auffinden des Schlüsselwortes *rule* in die Kopie übernommen.

   Wird besagtes Schlüsselwort gefunden, so wird aus der entsprechenden Zeile der Name der Regel ausgelesen. Die Namen aller *definierten* Regeln werden in einer Liste gespeichert.

Für die erweiterte Kopie der Transformationsdaten wird vor jeder Regel nun ein zusätzlicher Helper definiert, wie in Abbildung 6.16 zu sehen ist. Bei diesem handelt es sich um einen String, der in weiterer Folge die Namen aller *ausgeführten* Regeln aufnimmt.

```
helper def : output : String = '';
```

Abbildung 6.16: Neu definierter Helper *output* in der ".atl" Datei

Diese werden allerdings nicht direkt übernommen, sondern es werden die Namen der Typen sowohl des Eingangsparameters als auch des Ausgangsparameters der Regel in den String geschrieben (Unterabschnitt 6.4.3).
Abhängig davon, ob eine Regel bereits einen do-Block besitzt oder nicht, wird ein solcher erzeugt beziehungsweise erweitert. Dabei werden zwei Zeilen in jeden davon eingefügt, wie Abbildung 6.17 zeigt.

```
thisModule.output <- thisModule.output + (s.oclType().name + 'To' + t.oclType().name) + '\n';
thisModule.output.writeTo('ATL/output1.txt');
```

Abbildung 6.17: Auslesen der Transformationsregel

Die erste Zeile ist für die Beschreibung der *ausgeführten* Regel verantwortlich. Sie *erstellt* den *Namen* der Regel (wie bereits im vorangegangenen Absatz beschrieben). Die zweite Zeile ruft die in ATL vordefinierte Methode `writeTo()` auf, die die Informationen über die *ausgeführten* Regeln in eine Textdatei schreibt. Da diese Datei bei jedem Aufruf besagter Methode neu erstellt wird und somit der darin enthaltene Text verloren geht, ist es wichtig im ersten Schritt die jeweils neue Information an den bestehenden String anzuhängen. So erhält man nach dem letzten Aufruf eine vollständige Liste aller durchgeführten Transformationen.

3. Ausgabe der Liste der *definierten* Regeln in der main-Methode.
4. Kompilieren der neuen, erweiterten ATL Datei.
   Die ATL stellt zu diesem Zweck mehrere Compiler zur Verfügung. An dieser Stelle wird der Default-Compiler verwendet (siehe Abbildung 6.18). Dieser erstellt die kompilierte Datei an derselben Stelle, an der sich auch die ".atl" Datei befindet.
5. Kompilieren der Bibliothek (ATL Library), falls eine solche für die Transformation benötigt wird.
6. Durchführung der eigentlichen Modelltransformation durch Aufruf der Methode `runTransformation()`.

```java
/*
 * compiles atl-file (i.e. extended copy of original atl-file)
 */
@SuppressWarnings("deprecation")
public static void compileATL(String fileURL) throws IOException{

    File file = new File(fileURL);
    URL url = file.toURL();
    InputStream is = url.openStream();

    CompileTimeError[] error = AtlCompiler.getCompiler(AtlCompiler.DEFAULT_COMPILER_NAME)
            .compile(is, file.getAbsolutePath().replace(".atl", ".asm"));
    for(int i = 0; i < error.length; i++){
        System.out.println("CompileTimeError: " + error[i].getLocation() + " "
                + error[i].getDescription());
    }
}
```

Abbildung 6.18: Aufruf des ATL-Compilers

a) Einlesen und verarbeiten der Informationen, die in der Datei "CommonATL.-properties" enthalten sind.
Diese Datei enthält normalerweise wichtige Properties, die auf die ".atl" Datei oder die Metamodelle verweisen (oder auch auf zusätzliche Bibliotheken, falls solche benötigt werden). Im konkreten Fall werden diese allerdings nicht benötigt, da die wichtigsten Parameter bereits im Programm verarbeitet werden (siehe dazu den folgenden Schritt b) und daher die Erzeugung entsprechender Einträge in den Properties nicht mehr erforderlich ist.
Außerdem enthält diese Datei noch sogenannte *ATL Launching options*. Dabei handelt es sich um Optionen zur Feineinstellung der Transformationsausführung. Sie werden allerdings erst zu einem späteren Zeitpunkt verarbeitet (siehe Schritt c)ii.).
Durch einen entsprechenden Registrierungseintrag wird hier auch die Verarbeitung von Ecore-Metamodellen ermöglicht.

b) Laden der Metamodelle durch Aufruf von `loadModels()`. Da Ecore-Metamodelle verarbeitet werden sollen, wird zunächst eine Instanz der `EMFModelFactory` erzeugt. Anschließend werden Referenz-Modelle erzeugt, in die die beiden Metamodelle dann mit Hilfe des `EMFInjectors` geladen werden.
Das Quellmodell wird an dieser Stelle ebenfalls durch Verwendung des `EMFInjectors` geladen.

c) Aufruf der eigentlichen Modelltransformation in der Methode `doTransformation`.

   i. Eine Instanz des `EMFVMLaunchers` wird erzeugt. Dieser wurde speziell für die Verarbeitung von EMF Modellen entwickelt und ist in weiterer Folge für die Durchführung der Transformation verantwortlich.

   ii. Alle benötigten *launcherOptions* werden initialisiert, wie in Abbildung 6.19

zu sehen ist. Diese sind weitere Parameter für die Transformationsausführung, die aus der Datei “CommonATL.properties” geladen werden. Die Basis dafür bilden die beiden “.properties” Dateien, die bei der Ausführung des Plugins anhand der beiden im vorherigen Unterabschnitt 6.4.5.1 erwähnten Transformationen (“FamiliesToPersons”, “TreeToList”) erzeugt wurden.

```
# ================================================
# Common properties for ATL-Transformations
# ================================================

# Libraries paths

# ATL Launching options
CommonATL.options.supportUML2Stereotypes = false
CommonATL.options.printExecutionTime = false
CommonATL.options.OPTION_CONTENT_TYPE = false
CommonATL.options.allowInterModelReferences = false
CommonATL.options.step = false
```

Abbildung 6.19: ATL Launcher Options

iii. Die Modelle werden in den `EMFVMLauncher` geladen.

iv. Falls eine Bibliothek (eine zusätzliche “.atl” Datei mit Helpern) benötigt wird, wird diese in den Launcher geladen.

v. Die “.atl” Datei wird geladen.

vi. Die Modelltransformation wird mit dem `EMFVMLauncher` ausgeführt. Deren Ergebnis wird als *Object* zurückgeliefert.

d) Das im vorigen Schritt generierte Output-Modell wird unter Verwendung des `EMFExtractors` an der Stelle, auf die der entsprechende Pfad verweist, gespeichert.

7. Ist während der Transformation ein Fehler aufgetreten, wird die so ausgelöste *Exception* am Ende der main-Methode gefangen. Mit Hilfe einer eigenen Methode werden die Informationen aus den Exceptions ausgelesen und als String zurückgeliefert. Dieser wird dann an die Standardausgabe weitergeleitet. Diese Vorgehensweise mag redundant erscheinen, da die Ausgabe der Exceptions im Normalfall automatisch in der Konsole erfolgt, ist aber für die Verwendung der atl.jar als Bestandteil des WBT-Tools von Bedeutung.

Bei der Verwendung des ATL-Tools sind allerdings auch einige Einschränkungen zu beachten. Diese sind eher praktischer Natur und beruhen auf den Erfahrungen, die aus den durchgeführten Tests des Programms gewonnen wurden. So wäre es - betrachtet man den Code abseits der main-Methode - unter anderem möglich mehrere Library Dateien

zu verarbeiten. Dies wird aber dadurch eingeschränkt, dass die main-Methode selbst nur eine einzige derartige Datei akzeptiert. Solche Einschränkungen werden in der folgenden Liste durch das Kürzel *nC* für eine nicht Code-bedingte Einschränkung gekennzeichnet.

- Es können nur Ecore-Metamodelle verarbeitet werden.
- Es kann nur eine Bibliothek (Library Datei) verarbeitet werden (nC).
- Es wird nur eine ".atl" Datei verarbeitet. Theoretisch könnten mehrere als *.modules Properties* in der ".properties" Datei angegeben werden (nC).
- Es werden nur die Transformationen (also die Namen der Quell- und Zielobjekte), nicht aber die manuell definierten Namen der Regeln selbst aufgezeichnet.

6.4.6 ATL-Beispiel - Teil 2

Dieser Abschnitt beschäftigt sich mit der Ausführung der ATL-Transformation aus Unterabschnitt 6.4.2 ATL-Beispiel - Teil 1 mit Hilfe der atl.jar. Die folgende Abbildung 6.20 zeigt den Aufruf und das Ergebnis der Modell-Transformation.

```
tom@tom-VirtualBox:~/DA$ java -jar atl.jar
 '[redacted]/Families2Persons/sample-Families.xmi'
 '[redacted]/Families2Persons/sample-Persons.xmi'
 '[redacted]/Families2Persons/Families.ecore'
 '[redacted]/Families2Persons/Persons.ecore'
 '[redacted]/Families2Persons/Families2Persons.atl'
 '[redacted]/Families2Persons/F2P_prepared.atl'
 '[redacted]/Families2Persons/atl_output.txt'
----- DEFINED RULES -----
Member2Male
Member2Female
-------------------------
tom@tom-VirtualBox:~/DA$
```

Abbildung 6.20: Aufruf der Transformation "Families2Persons.atl" mit der atl.jar

Im vorliegenden Beispiel befinden sich alle benötigten Dateien im Ordner *Families2Persons*, in dem auch die vom Programm erzeugten Dateien (die kompilierte ATL Datei, das Output Modell und die Textdatei, die die Informationen über die aufgerufenen Regeln enthält) gespeichert werden. Es ist aber auch möglich Dateien von beliebigen Orten des PC aus aufzurufen, beziehungsweise an beliebigen Orten zu speichern.

Die Konsole liefert bei korrekter Ausführung der Transformation eine vollständige Liste aller *definierten* Regeln. Die vollständige Liste aller *ausgeführten* Regeln findet sich - wie im Aufruf definiert - in der Datei "atl_output.txt". Diese hat für das aktuelle Beispiel den Inhalt, der in Abbildung 6.21 zu sehen ist.

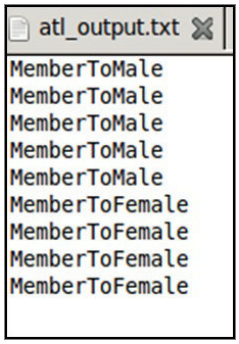

Abbildung 6.21: Liste aller ausgeführten Regeln der Transformation "FamiliesToPersons"

Im vorliegenden Beispiel "FamiliesToPersons" wurde die Regel *MemberToMale* fünf Mal aufgerufen, während die Regel *MemberToFemale* vier Mal aufgerufen wurde. Insgesamt weist die Datei "sample-Persons" also neun Personen-Einträge auf.

Werden alle Parameter in der korrekten Reihenfolge angegeben, ist es mit diesem Programm auch möglich, mehrere Transformationen hintereinander auszuführen und die Ergebnisse aufzuzeichnen.

6.5 White-Box Testing Tool[25]

Das *White-Box Testing Tool (WBT-Tool)* vereint alle in den oberen Abschnitten beschriebenen Sub-Tools (Abschnitt 6.2,Abschnitt 6.3,Abschnitt 6.4) in einem Gesamtprogramm. Die ursprüngliche Idee war die eines einzigen Programms, das alle Bestandteile (Erzeugen von Modellen aus dem Metamodell, Prüfung zugehöriger OCL-Constraints und Transformation in andere Modelle mittels ATL) enthält und linear aufruft. Dies war allerdings aufgrund von Inkompatibilitäten wichtiger einzubindender Zusatzpakete (.jar-Files) nicht möglich.

Daher wurde der Ansatz einer weitgehend unabhängigen Entwicklung der einzelnen Bestandteile verfolgt. Dies führte einerseits dazu, dass ein weiteres Programm erstellt werden musste, das auf die einzelnen Teile zugreift, andererseits dadurch aber die Einsatzmöglichkeiten des Tools in seiner Gesamtheit erhöht. Somit können die einzelnen Bestandteile des Tools flexibler eingesetzt und miteinander kombiniert werden.

6.5.1 Architektur

Aufgebaut ist das WBT-Tool aus vier wesentlichen Komponenten:

1. der Hauptkomponente: `WBTTool`
2. einer grafischen Benutzerschnittstelle: `StartScreen`

[25] Verfasser: Thomas Franz, BSc

3. einem Thread um Laufzeitinformationen aufzuzeichnen: `InfoThread`

4. einer Klasse zur Validierung der Zielmodelle: `Validator`

Diese sind in Abbildung 6.22 zur besseren Übersicht und Lesbarkeit in einem vereinfachten Klassendiagramm dargestellt.

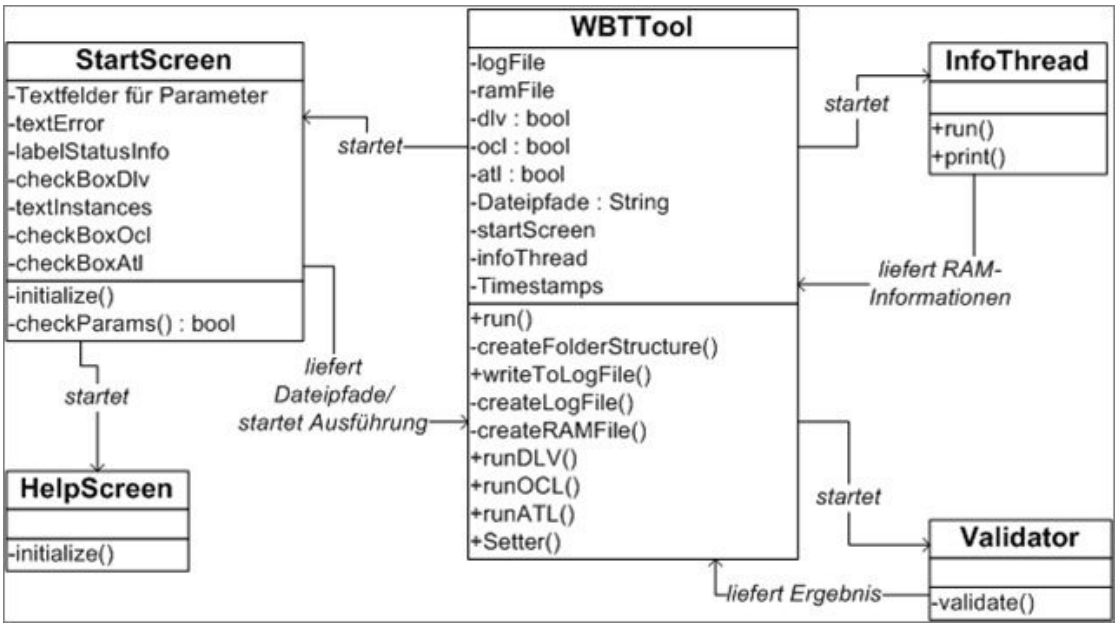

Abbildung 6.22: Vereinfachtes Klassendiagramm des WBT-Tools

Wird das Programm aufgerufen, so wird zunächst die Klasse `WBTTool` gestartet, welche die grafische Benutzerschnittstelle, das Graphical User Interface (GUI), aufruft (siehe Unterabschnitt 6.5.1.1). Diese besteht aus einem Hauptfenster, in der der Benutzer die gewünschte(n) Operation(en) auswählen kann und die entsprechenden Parameter definieren muss, sowie einem weiteren Fenster, das hilfreiche Informationen zur Verwendung des Programms liefert. Die beiden Komponenten werden aufgrund ihrer Bedeutung im Programm und der Komplexität der Aufgaben, die sie erfüllen, in der Folge näher erläutert.

Eine weitere Klasse enthält einen Thread, der immer dann gestartet wird, wenn das Programm tatsächlich eine der möglichen Operationen ausführt. Bei einem Thread handelt es sich um ein Konzept der *nebenläufigen Programmierung*. Dadurch kann die gleichzeitige Ausführung unterschiedlicher Aufgaben simuliert werden. Dies geschieht durch sehr schnellen Wechsel zwischen den einzelnen Aufgaben/Threads.

Im Rahmen des Programms wird der Thread dazu verwendet, während der Ausführung des WBT-Tools Informationen über das System zu sammeln. Dabei soll vor allem die Auslastung des Random Access Memory (RAM)[26] gemessen werden. Im Arbeitsspei-

[26] `https://de.wikipedia.org/wiki/Random-Access_Memory`, Zuletzt besucht: 19-04-2013

cher sind alle Programme, die gerade ausgeführt werden, und alle dazu erforderlichen Daten enthalten[27]. Durch die Messung der Auslastung soll ermittelt werden, welche Rechenleistung bei der Ausführung des WBT-Tools in Anspruch genommen wird. Dieser Messwert wird immer in einem bestimmten Zeitintervall (200 Millisekunden) berechnet und gespeichert. Ist die Ausführung aller Operationen beendet, wird auch der Thread beendet.

Die letzte Komponente ist ein Teilprogramm, das überprüft, ob die durch die Modelltransformation erzeugten Zielmodelle auch gültige Instanzen ihres Metamodells darstellen. Die Klasse `Validator` ist eine reduzierte Ausgabe der Klasse `OCLEvaluator`, die im Rahmen des OCL-Tools (siehe Abschnitt 6.3) entwickelt wurde. Der Aufbau sowie die verwendeten Konzepte sind daher ident, bis auf die Tatsache, dass die speziell für die Überprüfung von OCL-Constraints verantwortlichen Codeteile nicht benötigt wurden (siehe Unterabschnitt 6.3.3).

6.5.1.1 Benutzeroberfläche (GUI)

Das Programm stellt auch eine grafische Benutzerschnittstelle zur leichteren Bedienung zur Verfügung. Im Hauptfenster (`StartScreen`) kann der Benutzer definieren, welche Dateien verarbeitet werden sollen. Abhängig von der gewählten Operation müssen bestimmte Dateien ausgewählt werden. Diese können mit Hilfe eines eigenen Auswahlfensters festgelegt werden. Die Pfade, die zu diesen Dateien führen, werden nach der Auswahl in den entsprechenden Textfeldern angezeigt und können alternativ auch direkt eingegeben werden.

Im Hauptfenster wird auch eine Überprüfung der Parameter durchgeführt. Dabei wird zunächst überprüft, ob auch alle benötigten Parameter für die gewünschte Operation angegeben wurden (siehe Abbildung 6.23).

OCL-Constraints können auch für eine Überprüfung der Ergebnisse der ATL-Transformation übergeben werden. Hierfür muss keine explizite Operationsauswahl vorgenommen werden. Ebenso wie zusätzliche ATL-Bibliotheken werden sie nur dann vom Programm verarbeitet, wenn ihr Dateipfad angegeben ist.

Wurden alle benötigten Parameter angegeben, wird mit einer Überprüfung der Dateiformate fortgefahren (siehe Tabelle 6.4). Fast alle Parameter akzeptieren nur einen bestimmten Dokumenttyp. Das liegt an den Einschränkungen, dass nur Ecore-Metamodelle akzeptiert werden. Modelle, die daraus abgeleitet werden, liegen oft in einem speziellen XML Austauschformat vor: dem XML Metadata Interchange (XMI) Format. Alle OCL-Constraints, die überprüft werden sollen, müssen in einer Textdatei enthalten sein. Bei dieser kann es sich um eine herkömmliche Textdatei oder eine OCL Datei handeln.

Ergibt eine dieser Überprüfungen eine Verletzung der Anforderungen, so wird dies direkt im Hauptfenster angezeigt. Wurden alle Parameter korrekt gesetzt, werden diese an die Klasse `WBTTool` übergeben und dort verarbeitet.

[27] `https://de.wikipedia.org/wiki/Arbeitsspeicher`, Zuletzt besucht: 19-04-2013

Parameter	Operationen						
	DLV	OCL	ATL	DLV + OCL	DLV + ATL	OCL + ATL	DLV + OCL + ATL
Quellmetamodell	X	X	X	X	X	X	X
Quellmodell	-	X	X	-	-	X	-
OCL-Constraints für Quellmodell(e)	-	X	-	X	-	X	X
Zielmetamodell	-	-	X	-	X	X	X
ATL-Transformation	-	-	X	-	X	X	X
ATL-Bibliothek	-	-	O	-	O	O	O
OCL-Constraints für Zielmodell(e)	-	-	O	-	O	O	O

Legende:
Parameter wird nicht benötigt ... „-“
Parameter *muss* angegeben werden ... „X“
Parameter *kann* angegeben werden ... „O“

Abbildung 6.23: Übersicht über alle Kombinationen von Optionen und dafür benötigten Parameter

Tabelle 6.4: Akzeptierte Dateiformate

Parameter	Dateiformat
Quellmetamodell	.ecore
Quellmodell	.xmi
OCL-Constraints für Quellmodell(e)	.ocl/.txt
Zielmetamodell	.ecore
ATL-Transformation	.atl
ATL-Bibliothek	.atl
OCL-Constrains für Zielmodell(e)	.ocl/.txt

6.5.1.2 WBTTool

Die Klasse `WBTTool` ist die Hauptkomponente des Programms und für die Verarbeitung der Parameter, die der Benutzer in die GUI eingegeben hat, verantwortlich. Diese Klasse wird auch mit dem Programmaufruf gestartet.

Zunächst wird eine Ordnerstruktur zur Unterstützung des Testprozesses sowie zur Speicherung der Ergebnisse angelegt. Das ist notwendig, da die Ergebnisse von Modellgenerierung und Modelltransformation, sowie bestimmte Dateien gespeichert werden müssen. Durch die Verwendung einer genau definierten Ordnerstruktur können diese effizient und einfach an einer gemeinsamen Stelle abgelegt werden (z.B. werden alle Zielmodelle in einem gemeinsamen Ordner gespeichert). Im weiteren Verlauf können sie dann automatisch an dieser zentral vorgegebenen Stelle gefunden, eingelesen und weiterverarbeitet werden. Das ist vor allem beim Einsatz von DLV von Bedeutung. Die so generierten Modelle müssen zu einem späteren Zeitpunkt gefunden werden können um weiter Operationen durchzuführen (z.B. um OCL-Constraints zu prüfen oder sie mit ATL zu transformieren).

Anschließend wird die GUI gestartet. Fallen deren Überprüfungen positiv aus, werden die ausgewählten Operationen in der `WBTTool` Klasse gestartet. Dies erfolgt in der `run()` Methode. Hier werden dann die einzelnen Teilkomponenten aufgerufen. Vor beziehungsweise nach jedem Aufruf wird ein Zeitstempel gesetzt um die Dauer der Ausführung exakt messen zu können. Es wird die Zeiteinheit Millisekunden verwendet.

Alle wichtigen Informationen, die während der Ausführung des Programms (genauer der Operationen) gewonnen werden, werden in einer eigenen Textdatei gespeichert, der "log" Datei. In diese werden exakt die Informationen geschrieben, die von den Teil-Komponenten zurückgeliefert werden (z.B. die definierten und ausgeführten Regeln der ATL-Transformation; nähere Informationen in den Beschreibungen der Teilkomponenten DLV, OCL und ATL).

Wurden alle Operationen ausgeführt, werden die Informationen aus den Zeitstempeln angehängt. Eine solche Log-Datei wird für jeden Aufruf des Programms erstellt. Werden mehrere Operationen (unabhängig davon, ob sich Parameter oder Einstellungen ändern) ausgeführt, wird für jede davon eine solche Datei erstellt. Wird das Programm zwischen diesen Aufrufen beendet, ist allerdings zu beachten, dass bereits vorhandene Log-Dateien bei einem neuerlichen Programmstart überschrieben werden. Das gleiche gilt für die aufgezeichneten Informationen des `InfoThread`, die in einer *Excel-Datei* gespeichert werden.

6.5.2 Funktionalität

Während die vorhergehenden Abschnitte die Funktionen und Aufgaben der einzelnen Klassen innerhalb des WBT-Tools beschreiben, wird in diesem Abschnitt näher auf den Programmablauf (siehe Abbildung 6.24) eingegangen.

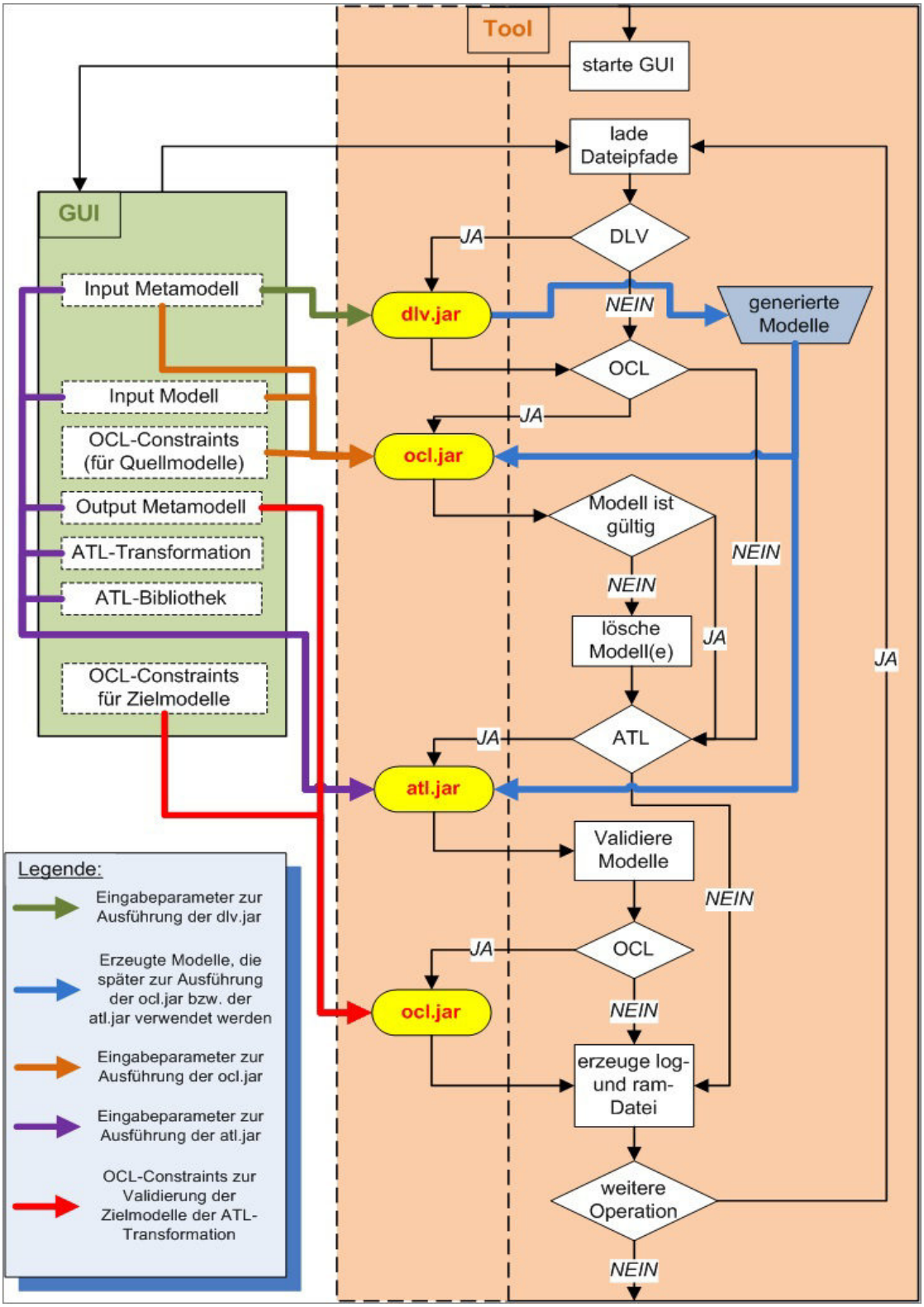

Abbildung 6.24: Grafische Darstellung des Programmablaufs

Wie bereits weiter oben erwähnt, wird zunächst die Hauptkomponente des WBT-Tools gestartet, worauf diese die Benutzeroberfläche aufruft. Die hier angegebenen Parameter werden dann an die `WBTTool` Klasse übergeben. Anschließend wird für jede mögliche Operation geprüft, ob diese ausgewählt wurde.

Möchte der Benutzer z.B. mehrere Modelle mit DLV generieren lassen, wird die entsprechende .jar-Datei (ausführbares Programm als Archiv-Datei) aufgerufen. Hierbei wird der Konsolenaufruf - wie im entsprechenden Unterabschnitt 6.2.2 beschrieben - vom Programm erstellt und ausgeführt. Die so erzeugten Modelle werden in einem Ordner, der bei der automatischen Erstellung der benötigten Ordnerstruktur (siehe Unterabschnitt 6.5.1.2) erstellt wurde, gespeichert.

Ebenso wird für die OCL-Komponente überprüft, ob sie ausgeführt werden soll oder nicht. Wurde ein bestimmtes Modell ausgewählt, wird dieses aufgerufen. Wurde zuvor allerdings die DLV-Komponente ausgeführt, werden die so erzeugten Modelle der Reihe nach aufgerufen. Wird nun festgestellt, dass ein Modell einen der angegebenen Constraints verletzt, so wird dieses automatisch aus dem Modell-Pool gelöscht. Dabei ist zu beachten, dass Modelle nur dann gelöscht werden, wenn sie vom Programm selbst generiert wurden.

Dieselbe Vorgehensweise wird auch bei der Ausführung der ATL-Modelltransformationen verfolgt. Auch hier kann ein vorgegebenes Modell oder eine Sammlung von generierten Modellen transformiert werden. Wurde diese ohne Fehler ausgeführt, wird das Ergebnis anschließend validiert. Hierfür wird die Klasse `Validator` aufgerufen, die prüft, ob das Zielmodell eine valide Instanz des Zielmetamodells ist.

Wurde eine weitere Datei mit OCL-Constraints zur weiteren Validierung an das Programm übergeben (siehe Unterabschnitt 6.5.1.1), werden diese Bedingungen mit Hilfe der ocl.jar überprüft. Die Vorgehensweise entspricht exakt derjenigen, die auch bei explizitem Einsatz der OCL-Komponente eingesetzt wird. An dieser Stelle wird allerdings nur das Ergebnis der Überprüfung in der Log-Datei festgehalten. Es wird keines der Zielmodelle explizit gelöscht, um eine weitergehende, manuelle Analyse der Ergebnisse zu ermöglichen. So können Erkenntnisse gewonnen und Rückschlüsse gezogen werden, die über das Vermögen des vorliegenden Programms hinausgehen.

Wurden alle Operationen ausgeführt, werden die Log-Datei und die Datei, die die Informationen über die RAM Auslastung enthält, in der Ordnerstruktur gespeichert. Anschließend kann der Benutzer über die GUI eine oder mehrere weitere Operationen starten.

6.5.3 WBT-Tool in Aktion

Um die Arbeitsweise des Programms zu veranschaulichen wird in diesem Abschnitt ein vollständiger Aufruf aller möglichen Operationen anhand eines Beispiels dokumentiert. Bei diesem handelt es sich um eine erweiterte Version von *Families To Persons* (die Metamodelle, ein beispielhaft ausgewähltes Quell- mit zugehörigem Zielmodell sowie die ATL-Transformation finden sich in Anhang B).

Diese Erweiterungen beziehen sich im Wesentlichen auf neu eingeführte Attribute, wie zum Beispiel die Körpergröße von Personen. Auch die Transformation wurde auf-

```
1  library Lib4F2P;
2
3  -- defines if person is male or female
4
5  helper context Families!Member def: isFemale() : Boolean =
6    if not self.familyMother.oclIsUndefined() then
7      true
8    else
9      if not self.familyDaughter.oclIsUndefined() then
10       true
11     else
12       false
13     endif
14   endif;
15
16
17 -- calculates value of bmi for male persons
18
19 helper context Families!Member def: calculateBMI() : Real =
20   (self.weight / self.height);
21
22
23 -- defines category of bmi for female persons
24
25 helper context Families!Member def: bmiCategory() : String =
26   if ((self.weight / self.height) < 0.5) then
27     'under'
28   else
29     if ((self.weight / self.height) > 1.0) then
30       'over'
31     else
32       'normal'
33     endif
34   endif;
```

Listing 6.1: Bibliothek für die Transformation “Families2Persons.atl”

geteilt. Die *Helper* befinden sich nun in einer eigenen Bibliothek. Neben dem bereits existierenden, der das Geschlecht des Familienmitglieds bestimmt, wurden zwei weitere *Helper* definiert. Einer davon bestimmt den exakten “Body Mass Index” (BMI) für männliche Personen. Der zweite ermittelt für weibliche Personen, in welche Gewichtskategorie (unter-, über-, oder normalgewichtig) sie fallen (siehe Listing 6.1).

Die Testinstanzen werden mit Hilfe von DLV erzeugt. Die Attribute erhalten hier zufällige Werte, die auch negativ sein können. Mittels OCL-Constraints wird sichergestellt, dass es keine Familienmitglieder mit Körpergröße < 0 geben kann (siehe Listing 6.2).

Nach der Transformation werden für die erzeugten Zielmodelle ebenfalls zusätzliche Bedingungen überprüft. So darf am Ende des Prozesses in keinem Modell eine Person mit negativem Alter enthalten sein (diesen Umstand könnte/sollte man bereits bei der

Erzeugung der Quellmodelle prüfen; hier wird er aber verwendet um zu zeigen, dass auch die Prüfung von Constraints nach der Durchführung der Transformation mit dem WBT-Tool möglich ist). Diese Bedingungen sind in Listing 6.3 enthalten.

```
1 context Member
2 self.height > 0
```

Listing 6.2: OCL-Constraint aus "Families_constraints.ocl", der für Quellmodelle überprüft wird

```
1 -- Frauen
2 context Female
3 self.age > 0
4
5 --Maenner
6 context Male
7 self.age > 0
```

Listing 6.3: OCL-Constraints aus "Persons_constraints.ocl", die für Zielmodelle überprüft werden

Das Programm wird mit den in Abbildung 6.25 dargestellten Einstellungen gestartet. Hier werden alle möglichen Funktionen ausgeführt. Der Parameter "maxInstanzen" gibt an, dass jede Klasse des Metamodells in einem Testmodell keine oder genau eine Instanz haben kann.

Alle wichtigen Informationen während der Ausführung des Tools werden in eine Log-Datei geschrieben. Zwecks Übersichtlichkeit werden in der Folge nur Auszüge aus dieser Datei präsentiert, die sich in der Formatierung, aber nicht im Inhalt von der Original-Datei unterscheiden. Zu Beginn wird aufgezeichnet, wie viele Modelle von DLV erzeugt wurden (siehe Listing 6.4).

```
1 ========== DLV ==========
2 Perform ASP for meta model '...\Families2Persons\Families.ecore'
3
4 Number of generated models: 6
5 5 model(s) are created
```

Listing 6.4: Informationen über Modellgenerierung mit DLV (Auszug aus der Log-Datei "log_1.txt")

DLV erzeugt automatisch 6 Modelle. Allerdings wird in diesem Fall auch ein leeres Modell erzeugt, das für die weitere Verarbeitung aber nicht geeignet ist und daher ignoriert werden kann. Somit wurden für das aktuelle Beispiel in Summe 5 Testinstanzen erzeugt.

Im konkreten Beispiel sind alle OCL-Constraints für drei der fünf Modelle erfüllt. Zwei erfüllen die Bedingung, dass die Körpergröße nicht negativ sein darf, nicht.

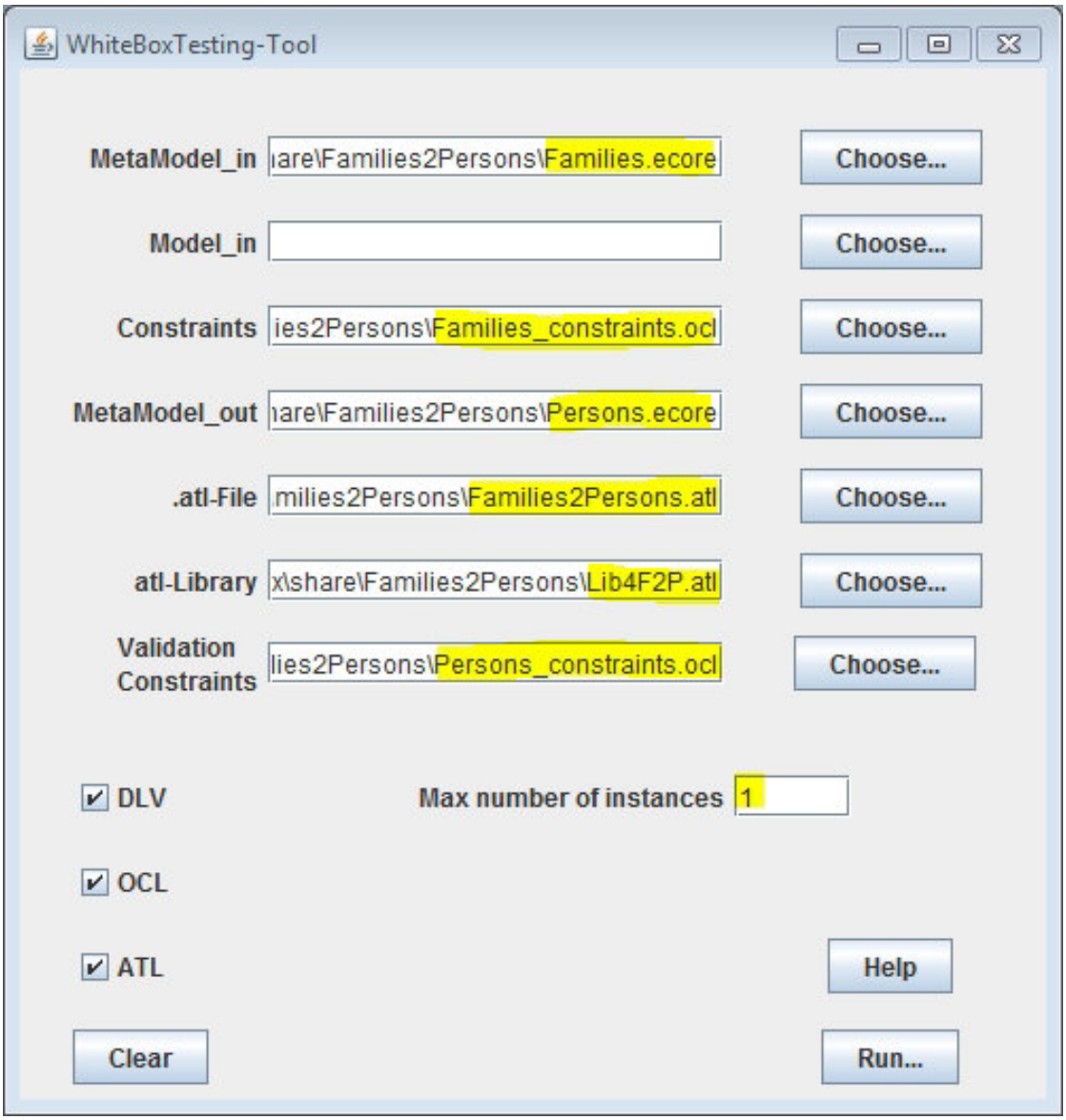

Abbildung 6.25: Grafische Oberfläche mit allen Einstellungen zum Start des Programms

```
7  ============================ OCL ============================
8  Validation results for instance 'Families1.xmi' :
9
10 All constraints for model are valid!
11
12 -------------------------------------------------------------
13 Validation results for instance 'Families2.xmi' :
14
15 Diagnostic WARNING source=org.eclipse.emf.ecore code=0 OCL validation ↩
       constraint 'Constraint_1' is not satisfied for 'org.eclipse.emf.ecore.↩
       impl.DynamicEObjectImpl/http://families/1.0#Member@71f68411{file://../↩
       Tool/OUTPUT/models/Families2.xmi#//@mother}' data=[org.eclipse.emf.↩
       ecore.impl.DynamicEObjectImpl@71f68411 (eClass: org.eclipse.emf.ecore.↩
       impl.EClassImpl@43e8c82f (name: Member) (instanceClassName: null) (↩
       abstract: false, interface: false))]
16 ERROR: Constraints are invalid!
```

Listing 6.5: Informationen über die Prüfung der OCL-Constraints (Auszug aus der Log-Datei "log_1.txt")

In der Log-Datei wird genau ausgegeben, welcher Constraint (in diesem Fall der "Constraint_1") verletzt wurde (siehe Listing 6.5).

Wie bereits weiter oben erwähnt, stehen die automatisch vom Programm erstellten, bearbeiteten Dateien, die die nummerierten Constraints enthalten, auch nach der Ausführung des WBT-Tools zur Verfügung. So kann man rasch feststellen, welche Bedingungen verletzt wurden.

Außerdem wird ausgegeben, wo im Modell der Fehler genau aufgetreten ist. Alle Modelle, die zumindest einen Constraint verletzen, werden bei der weiteren Verarbeitung nicht mehr berücksichtigt und daher werden nur drei Modelle für die Ausführung der Transformation herangezogen.

Vor der ersten Ausführung der ATL-Transformation wird die Originaltransformation vom Programm verarbeitet und erweitert. Dabei werden auch die definierten Regeln ausgegeben. Anschließend wird die Transformation kompiliert und ausgeführt (siehe Listing 6.6).

Im Zuge dieses Vorgangs werden bestimmte Parameter bezüglich der Ausführungszeit gemessen. Es wird ermittelt, wie viel Zeit Vorgänge, wie zum Beispiel das Laden der Metamodelle sowie des Quellmodells in ATL oder die Ausführung der Transformation, benötigen. Im Zuge der Auswertungen dieser Daten wurde festgestellt, dass das Kompilieren der Transformation die zeitintensivste Aufgabe ist und daher nur ein Mal zu Beginn des Transformationsprozesses ausgeführt wird. Allerdings betrifft diese Optimierung nur die Ausführungen des WBT-Tools, bei denen mehrere Modelle mit DLV erzeugt werden. Wird dieselbe Transformation (unabhängig davon, ob mit dem gleichen oder unterschiedlichen Quellmodellen) zwei Mal hintereinander manuell über die grafische Oberfläche gestartet, wird die Transformation für jede Ausführung kompiliert.

```
35 ═══════════════════════════ ATL ═══════════════════════════
36 Perform ATL-Transformation for instance 'Families1.xmi'
37
38 ------ DEFINED RULES ------
39 Member2Male
40 Member2Female
41 -------------------------------
42 ----------TIMESTAMP INFO----------
43 Create ATL = 31 ms
44 Compile ATL = 3183 ms
45 Load Models = 109 ms
46 Perform Trafo = 171 ms
47 -------------------------------
48
49
50 MemberToFemale
51
52 ---------------------------------------------------------------
53 Perform ATL-Transformation for instance 'Families4.xmi'
54
55 ----------TIMESTAMP INFO----------
56 Load Models = 873 ms
57 Perform Trafo = 390 ms
58 -------------------------------
59
60
61 MemberToFemale
62
63 ---------------------------------------------------------------
64 Perform ATL-Transformation for instance 'Families5.xmi'
65
66 org.eclipse.m2m.atl.engine.emfvm.VMException: Unable to access lastName ↩
       on OclUndefined
67   at __initfamilyName#17(Families2Persons.atl[19:5-19:33])
68     local variables: self=IN!<unnamed>
69   ...
70     at main#33(Families2Persons.atl)
71     local variables: self=Families2Persons : ASMModule
72
73 ----------TIMESTAMP INFO----------
74 Load Models = 827 ms
75 -------------------------------
76
77
78 MemberToFemale
```

Listing 6.6: Informationen über die ATL-Transformation (Auszug aus der Log-Datei "log_1.txt")

```
81 ========== Validate ATL-Output ==========
82 Validate model 'Families1_AtlOut.xmi'
83
84 Model is valid!
85
86 ----------
87 Validation results for instance 'Families1_AtlOut.xmi' :
88
89 Diagnostic WARNING source=org.eclipse.emf.ecore code=0 OCL validation ↩
       constraint 'Constraint_1' is not satisfied for 'org.eclipse.emf.ecore.↩
       impl.DynamicEObjectImpl/http://persons/1.0#Female@1ffa87c5{file://../↩
       Tool/OUTPUT/atlOutput/Families1_AtlOut.xmi#/}' data=[org.eclipse.emf.↩
       ecore.impl.DynamicEObjectImpl@1ffa87c5 (eClass: org.eclipse.emf.ecore.↩
       impl.EClassImpl@af04c53 (name: Female) (instanceClassName: null) (↩
       abstract: false, interface: false))]
90 ERROR: Constraints are invalid!
91
92 ------------------------------------------------------------
93 Validate model 'Families4_AtlOut.xmi'
94
95 Model is valid!
96
97 ----------
98 Validation results for instance 'Families4_AtlOut.xmi' :
99
100 All constraints for model are valid!
101
102 ------------------------------------------------------------
103 Validate model 'Families5_AtlOut.xmi'
104
105 File is empty!
```

Listing 6.7: Informationen über Validierung der Zielmodelle (Auszug aus der Log-Datei "log_1.txt")

Im vorliegenden Beispiel wurden die Modelle "Families1.xmi" und "Families4.xmi" erfolgreich transformiert. Das dritte Modell ("Families5.xmi") konnte nicht erfolgreich verarbeitet werden, da die Transformation hier auf ein Attribut (den Nachnamen der Familie) zugreifen möchte, das allerdings nicht existiert.

Neben dieser Fehlermeldung enthält die Log-Datei noch weitere, detailliertere Informationen über die Fehler, die bei der Ausführung auftreten. Aus Gründen der Übersichtlichkeit wurden diese in den Listings allerdings nicht abgebildet.

Nach jeder Transformation wird überprüft, ob die Ergebnisse (Zielmodelle) selbst valide Instanzen ihres Metamodells, sind beziehungsweise ob sie auch alle definierten OCL-Constraints erfüllen, wie in Listing 6.7 zu sehen ist.

Im vorliegenden Beispiel sind die beiden Zielmodelle "Families1_AtlOut.xmi" und "Families4_AtlOut.xmi" gültige Instanzen des zugehörigen Metamodells. Allerdings er-

gibt die Prüfung der OCL-Constraints, dass im ersten Modell eine weibliche Person existiert, deren Alter < 0 ist (und somit die entsprechende Bedingung verletzt ist).

Einzig für die Testinstanz Nummer 4 konnten alle Operationen fehlerfrei ausgeführt werden.

Für das letzte Modell ("Families5.xmi") konnte kein Zielmodell erzeugt werden, da die Transformation nicht gültig durchgeführt wurde. Da das WBT-Tool jedoch für jedes Quellmodell eine Datei erzeugt, in die zu einem späteren Zeitpunkt das Ergebnis der Transformation geschrieben wird, existiert eine solche im entsprechenden Ordner. Für diese Datei wird in der Log-Datei aufgezeichnet, dass sie leer ist und somit keine Validierung möglich ist.

Am Ende der Log-Datei findet sich noch eine Aufschlüsselung über die Ausführungszeiten der einzelnen Teilkomponenten (DLV, OCL, ATL) und der Validierung. Diese werden auch addiert um zu bestimmen, wie lange der Prozess mit den gewählten Aufgaben gedauert hat (siehe Listing 6.8).

```
109 Execution time elapsed:
110 DLV: 4899 ms
111 OCL: 38600 ms
112 ATL: 9812 ms
113 VAL: 16367 ms
114 Total: 69678 ms
```

Listing 6.8: Informationen über Validierung der Zielmodelle (Auszug aus der Log-Datei "log_1.txt")

Im vorliegenden Beispiel nahm die Prüfung der OCL-Constraints die meiste Zeit in Anspruch, während die Erstellung der Testinstanzen mit DLV trotz der Komplexität der Aufgabe die geringste Ausführungszeit benötigt hat.
Diese Werte wurden auch für komplexere Ausführungen ermittelt und ausgewertet. Die Durchführung und Ergebnisse dieser Evaluierung werden im Kapitel 7 näher vorgestellt.

6.6 Alternative Technologien[28]

Oftmals ist bei der Entwicklung von Tools der beschriebene Weg nicht der einzige getestete und zielführende. Es soll daher ein kurzer Einblick in alternative Ansätze zur Umsetzung gegeben werden und eine Begründung, warum diese nicht weiter verfolgt worden sind. Grundsätzlich bedeutet dies jedoch nicht, dass die Realisierung mit einem der beschriebenen Ansätze vollkommen unmöglich ist.

6.6.1 Modellgenerierung

Eine wichtige Voraussetzung beim Testen von Software beziehungsweise Modelltransformationen ist es einen ausreichend großen Pool an Testinstanzen zur Verfügung zu stellen.

[28] Verfasser: Sabine Wolny, BSc

Im Falle von Modelltransformationen gibt es einige theoretische Konzepte, auf deren Basis Tools Modelle erzeugen. Allerdings handelt es sich hierbei um ein relativ komplexes Problem, sodass die Mehrzahl der Ansätze nur in gewissem Umfang einsetzbar ist. Dieser Abschnitt beschäftigt sich mit den Ansätzen, die für diese Arbeit untersucht, beziehungsweise umgesetzt wurden, jedoch aus bestimmten Gründen nicht im entwickelten WBT Tool Verwendung gefunden haben.

6.6.1.1 EMFtoCSP

Ein möglicher Ansatz bestand darin, das Programm *EMFtoCSP* (Unterabschnitt 8.1.2.1) als Grundlage zu einer automatisierten Erzeugung von Modellinstanzen, die dann als Testinstanzen fungieren, zu nutzen. Grundsätzlich hatte dies auch den Vorteil, dass das Programm direkt OCL-Constraints verarbeiten konnte. Es gab jedoch Schwierigkeiten in verschiedenen Bereichen.

Zum einen nutzt *EMFtoCSP* die zwei weiteren, unabhängigen Programme *ECLiPSe*, das zur Lösung des Constraints Satisfaction Problem (CSP) Code herangezogen wird, sowie *GraphViz*, das die grafische Ausgabe der Instanz erzeugt (im konkreten Fall als .png). Vor allem im Falle von *GraphViz* waren zum Zeitpunkt der Evaluierung des Ansatzes massive Probleme unter Windows bekannt.

Zum anderen war zwar der Sourcecode frei verfügbar, allerdings kaum kommentiert. Dadurch war es zum Teil nicht möglich die Erzeugung des CSP nachzuvollziehen, beziehungsweise es zu adaptieren, sodass mehr als eine Instanz erzeugt wird. Auch war nicht feststellbar, wie aus dem Ergebnis, welches das *ECLiPSe*-Tool lieferte, die Ausgabe von *GraphViz* erzeugt wurde, beziehungsweise wie diese Ausgabe in ein anderes Format (z.B. .xmi) konvertiert werden könnte.

6.6.1.2 Java

Ein weiterer Ansatz, der zur Generierung von Modellen auf Basis eines Metamodells verfolgt wurde, war ein rein Java basiertes Programm. Dazu wurde auf Teile von Code zurückgegriffen, die auch vom Tool *EMFtoCSP* verwendet werden. Dabei wurden insbesondere die Klasse `EAnnotation.java` übernommen und die Klasse `EMFModelReader.java` adaptiert.

Zu Beginn wird in der Hauptklasse das Metamodell eingelesen und mit Hilfe einer weiteren Klasse (basierend auf dem `EMFModelReader`) in seine Bestandteile zerlegt. Mit diesen Daten werden dann zwei weitere Klassen aufgerufen, die zwei Modelle erzeugen:

- ein minimales (besteht nur aus unbedingt benötigten Bestandteilen)
- ein vollständiges (enthält zumindest eine Instanz jedes Bestandteils)

Beide Ansätze arbeiten ähnlich. Zunächst wird ausgehend vom obersten Package durch dessen Bestandteile iteriert. Dieser lineare Ansatz wurde bewusst gewählt um als Ergebnis eine gültige .xmi-Datei zu erhalten.

Begonnen bei der Wurzelklasse wird eine Instanz dieser Klasse erzeugt (mit allen benötigten Attributen). Anschließend werden alle ausgehenden Referenzen durchlaufen. Dabei wird unterschieden, ob es sich um eine Containment-Beziehung oder eine einfache Referenz handelt. Im ersten Fall wird ein entsprechender Eintrag in die Liste benötigter Instanzen eingefügt. Eine Referenz wird als weiteres Attribut im aktuellen xml-Tag eingefügt.

Anschließend werden die Containment-Beziehungen weiter verfolgt und für die Klassen, auf die sie verweisen, konkrete Instanzen in das Modell eingefügt. Ist die letzte Instanz in dieser Kette von Beziehungen erreicht, wird der entsprechende xml-Tag geschlossen. Es wird in der xmi-Datei so lange wieder auf höhere Ebenen zurückgegangen, bis das Programm auf einen entsprechenden Eintrag stößt, dass eine weitere Instanz auf der aktuellen Ebene eingefügt werden muss.

Im Wesentlichen wird bei beiden Ansätzen entlang der Beziehungen (Containment, Reference) durch die Klassen im Metamodell navigiert. Diese enthalten das Attribut *lowerBound*, das angibt, wie viele Instanzen der referenzierten Klasse im Modell benötigt werden. Im minimalen Modell können also alle Referenzen, die als untere Grenze den Wert *0* aufweisen, ignoriert werden. Im vollständigen Modell müssen aber auch in diesem Fall Instanzen der Klassen erzeugt werden. Daher wird auch hier am Ende der ersten Ausführung des Algorithmus geprüft, ob auch wirklich alle Klassen besucht wurden. Ist dies nicht der Fall, müssen auch für diese noch Instanzen erzeugt werden.

Die Darstellung des Algorithmus erfolgt hier relativ stark vereinfacht. In der Umsetzung allerdings weist bereits der kleinere der beiden Ansätze (derjenige, der ein minimales Modell liefert) eine hohe Komplexität auf. Diese ist auch bedingt durch die Tatsache, dass er auch Enumerationen, abstrakte Klassen und verschiedenste Attributtypen (Integer, String, Boolean und Double) verarbeiten kann.

Berücksichtigt man aber, dass Ecore noch eine hohe Zahl weiterer Datentypen und Konzepte zur Verfügung stellt, zeigt sich bereits die Beschränktheit dieses Ansatzes. Weit bedeutender ist aber das Problem, dass der Algorithmus für eine größere Vielfalt an generierten Modellen sehr schnell unübersichtlich wird, da ein gewisser Zufalls-Parameter, was die Anzahl an generierten Instanzen betrifft, eingeführt werden muss.

Grundsätzlich müssten für diesen Parameter aber gewisse Einschränkungen gelten. Andernfalls besteht auf Grund der Idee hinter dem Algorithmus das Problem, dass das Modell sehr schnell sehr groß werden kann, wenn jede Referenz eine beliebige neue untere Grenze (die wiederum über der im Metamodell definierten liegen muss) ermittelt und diese sehr hoch gewählt wird.

Ein völlig unbeachtetes Problem bei diesem Konzept sind auch Einschränkungen, die nicht im Modell definiert werden. Wird zum Beispiel ein Integer-Attribut benötigt, wird für dieses ein vorbestimmter Wert eingesetzt, bei dem vollständig auf Variabilität verzichtet wurde. Gibt es aber die nicht im Modell definierbare Einschränkung, dass ein solcher Wert nicht negativ sein darf, ist es in gewissem Maße vom Zufall abhängig, ob ein gültiges Modell erzeugt wird oder nicht.

Ein potenzieller Vorteil ist die Flexibilität des Ansatzes, da alle Bestandteile des Metamodells in beliebiger Anzahl instanziiert und einzeln bearbeitet werden können.

Ein weiterer ist, dass das Programm direkt ein (zumindest syntaktisch) gültiges XMI-Modell liefert, ohne dass weitere Zwischenschritte erforderlich sind.

6.6.2 OCL

In diesem Abschnitt werden die Alternativen, die bei der Entwicklung des Programms zur automatischen Prüfung von OCL-Constraints getestet wurden, beschrieben. Dabei wird jede dieser Alternativen zunächst kurz beschrieben, anschließend wird auf die Schwierigkeiten und Probleme bei der Umsetzung eingegangen.

6.6.2.1 OCL in Java

Der entwickelte Ansatz zur automatisierten Prüfung von OCL-Constraints in einem Modell ist ein rein Java-basierter Ansatz. Die wichtigste Informationsquelle für diese Art von Problemstellung ist die "OCL Documentation" in der "Eclipse Documentation". Konkrete Hinweise auf den Einsatz von *OCL in Java* finden sich dort im Abschnitt "Programmers Guide"[29].

Es wird ein recht dynamischer Einsatz von OCL in Java vorgestellt. Auch finden sich hier zahlreiche konkrete Auszüge aus Java-Code zur Umsetzung der vorgestellten Konzepte. Allerdings beschränken sich diese oft auf die konkrete Verwendung von OCL. Eine weitere Schwierigkeit stellt der Umstand dar, dass diese Teile auf den gesamten Abschnitt verteilt sind und erst zusammengesucht werden müssen. Auch finden sich in dieser Dokumentation fast keine Hinweise darauf, wie man die Konzepte auf bestimmte (Meta-)Modelle anwenden und für bestimmte Klassen und Methoden einsetzen kann.

Die vorgestellten Code-Beispiele bauen auf einem Beispiel auf, bei dem allerdings das Beispielmodell erst in Java-Code überführt werden muss. Dieser Vorgang ist in der Dokumentation zwar beschrieben, jedoch im Abschnitt "Tutorials" -> "OCLinEcore tutorial" -> "Generating Java Code". Im Wesentlichen wird für das bestehende Modell ein sogenanntes "genmodel" erstellt und, basierend auf diesem, von Eclipse automatisch Java-Code generiert.

Abgesehen von der Tatsache, dass dies ein sehr mühsames Vorgehen ist, konnten auch keine Hinweise darauf gefunden werden, wie dieser Prozess automatisiert aus Java-Code heraus angestoßen werden könnte. Auch konnten die Code-Beispiele auf Basis dieses generierten Codes nicht in allen Fällen nachvollzogen, beziehungsweise in der Praxis lauffähig ausgeführt werden. Dieses Problem trifft unter anderem auch auf die im Abschnitt "Standalone Configuration" vorgestellten Schritte zu, womit OCL unabhängig von Eclipse benutzt werden soll. So konnte ein Teil dieses Codes aufgrund fehlender Zusatzpakete (.jar-Files) nicht korrekt gestartet werden.

Allerdings würde auch ein Ansatz, der auf die Codegenerierung verzichten könnte, eine komplizierte Navigation im Modell erfordern um die angegebenen Codeteile auch wirklich ausführen zu können. Generell stellt der vorgegebene Code aber eine recht fle-

[29] `http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.ocl.doc%2Fhelp%2FProgrammersGuide.html`, Zuletzt besucht: 30-07-2013

xible, dynamische Einsatzmöglichkeit von OCL zur Java-basierten Constraint-Prüfung dar.

6.6.2.2 OCLinEcore

Mit *OCLinEcore*[30] wird eine weitere Möglichkeit zur Umsetzung von OCL-Constraints in der "OCL Documentation" im Abschnitt "Tutorials" vorgestellt. Im Wesentlichen werden die OCL-Constraints direkt in das Metamodell eingebunden. Hierfür wird ein eigener, in Eclipse integrierter Editor verwendet. Wird nun ein Modell validiert, wird zusätzlich geprüft, ob die OCL-Constraints erfüllt sind.

Dieser Ansatz lief in der praktischen Anwendung sehr stabil. Allerdings konnte nicht herausgefunden werden, wie die Integration der Constraints in das Metamodell automatisiert aus Java-Code ausgeführt werden könnte.

Eine Alternative dazu wäre die Repräsentation dieser Constraints als Annotations im Ecore-Metamodell auszunutzen, jedoch ist dafür mitunter eine komplizierte Navigation im Metamodell erforderlich. Außerdem muss beachtet werden, dass dessen generelle Struktur nicht verändert wird.

Bei beiden Ansätzen bleibt aber das Problem eines automatisierten Aufrufs des Prozesses der Validierung des Modells bestehen, das nicht zufriedenstellend gelöst werden konnte.

[30] `http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.ocl.doc%2Fhelp%2FTutorials.html`, Zuletzt besucht: 30-07-2013

KAPITEL 7

Evaluierung[1]

Es bleibt nun die Frage offen, ob das neu entwickelte Tool den Anforderungen und der Zielsetzung gerecht wird. Dieses Kapitel beschäftigt sich mit der Evaluierung des Programms anhand bestimmter Eingabesets.

Es wird eine Fallstudie gemacht, die sich an der bei Runeson und Höst [83] beschriebenen Vorgehensweise orientiert. Die Studie wird mit einem objektiven Testdatenset (nicht selbst erstellt) durchgeführt.

In diesem Kapitel werden zuerst im Abschnitt 7.1 die Forschungsfragen näher erläutert. Danach wird der Aufbau der Fallstudie, insbesondere die Eingabedaten beschrieben (siehe Abschnitt 7.2). Im Abschnitt 7.3 werden die Testumgebungen beschrieben und die Durchführungen protokolliert. Außerdem wird konkret auf die Laufzeiten und die RAM Auslastung eingegangen. Im Abschnitt 7.4 wird eine Diskussion der Ergebnisse durchgeführt, und abschließend werden im Abschnitt 7.5 die Erkenntnisse kurz zusammengefasst.

7.1 Zielsetzung[2]

Die Ziele der Evaluierung sind klar an den Zielen der Arbeit orientiert und umfassen daher folgende Bereiche:

1. *Skalierbarkeit*: Wie viele Lösungen (Modelle) werden generiert bei einer sich ändernden Anzahl an Klassen? Gibt es einen Grenzwert, ab dem das Programm nicht mehr funktioniert?

2. *Fehlererkennung*: Gibt es Fehler, die mit den automatisierten Tests im Gegensatz zu "realen", selbst erstellten Tests erkannt oder nicht erkannt werden?

3. *Fehlerhäufigkeit*: Welche Fehler kommen vor? Wie häufig sind diese Fehler?

[1] Verfasser: Thomas Franz, BSc & Sabine Wolny, BSc
[2] Verfasser: Sabine Wolny, BSc

4. *Speicherbedarf*: Wie viel Speicher benötigt das Programm während der Laufzeit?

5. *Komplexität der Laufzeit*: Wie lange dauert der Durchlauf des Programms für die Testsets?

7.2 Aufbau der Fallstudie[3]

Die Forschungsfragen sollen anhand bestimmter Eingabedaten geklärt werden. Als Eingabedaten werden die Übungsaufgaben der Lehrveranstaltung "Model Engineering" der Technischen Universität Wien der Wintersemester 2009 bis 2012 gewählt. Hierbei werden einerseits mit Hilfe von DLV Modellinstanzen zu der gegebenen Musterlösung automatisch generiert und andererseits die Lösungen, die von den Studenten im Rahmen dieser Übung erstellt wurden, getestet. Die gesamte Testsuite umfasst daher rund 160 Transformationen.

Übungsaufgabe 2009: SWML 2 SMVC Transformation
Im Jahr 2009 sollte ein Metamodell, welches eine einfache Modellierungssprache (Simple Web Modeling Language (SWML)) beschreibt, mit Hilfe von ATL in eine einfache Webanwendung, die der Model-View-Controller Architektur (Simple Model View Controller Modeling Language (SMVCML)) folgt, transformiert werden. `swml.ecore` ist das Metamodell der Modellierungssprache für Webanwendungen. SWML Modelle beschreiben jeweils die Daten-Schicht und die Hypertext-Schicht von Web-Anwendungen. Die relevanten Informationen bezüglich der Anzahl an Elementen im Quellmetamodell sind in Tabelle 7.1 zu sehen.

Tabelle 7.1: Übungsbeispiel 2009:
Quellmetamodell swml.ecore: Anzahl an Elementen

Element	Anzahl
Enumerations	1
Klassen	24
Vererbungen	12
Referenzen (kein containment)	6
Referenzen (containment)	11
Bidirektionale Referenzen (containment)	2
Maximale Anzahl an Attributen in einer Klasse	3

Übungsaufgabe 2010: Class 2 Relation Transformationen
Die Aufgabe in diesem Jahr war eine ATL Modelltransformation zu finden, die ein UML-Modell, welches konform zum Klassendiagramm-Metamodell ist, in ein relationales Schema übersetzt. Die Zielmodelle müssen konform zum Zielmetamodell (`Relational.ecore`)

[3] Verfasser: Sabine Wolny, BSc

sein. Die verschiedenen Elemente des Quellmetamodells (`ClassDiagram.ecore`) sowie deren Anzahl sind in der Tabelle 7.2 kurz zusammengefasst.

Tabelle 7.2: Übungsbeispiel 2010:
Quellmetamodell ClassDiagram.ecore: Anzahl an Elementen

Element	Anzahl
Enumerations	2
Klassen	11
Vererbungen	2
Referenzen (kein containment)	5
Referenzen (containment)	11
Bidirektionale Referenzen (containment)	0
Maximale Anzahl an Attributen in einer Klasse	6

Übungsaufgabe 2011: SOOML 2 SOOPL Transformation
Im Zuge der Übungsaufgabe von 2011 sollte ein Metamodell für einfache objektorierentierte Modellierungssprachen (Simple Object-Oriented Modeling Language (SOOML)) transformiert werden. Das Zielmetamodell war `soopl.ecore`, welches eine einfache objektorientierte Programmiersprache (Simple Object-oriented Programming Language (SOOPL)) beschreibt. Mit Hilfe des Quellmetamodells kann ein Nutzer die Struktur, das Verhalten und die Ausführung eines Softwaresystems spezifizieren. In Tabelle 7.3 werden einige Informationen bezüglich Struktur des Quellmetamodells angegeben.

Tabelle 7.3: Übungsbeispiel 2011:
Quellmetamodell sooml.ecore: Anzahl an Elementen

Element	Anzahl
Enumerations	1
Klassen	23
Vererbungen	10
Referenzen (kein containment)	16
Referenzen (containment)	11
Bidirektionale Referenzen (containment)	2
Maximale Anzahl an Attributen in einer Klasse	2

Übungsaufgabe 2012: GradingSystem 2 FileDefinitionSet
Die Übungsaufgabe aus dem Jahr 2012 befasste sich mit der Transformation eines Benotungssystems (`gradingsystem.ecore`) in ein FileDefinitionSet (`csv.ecore`). Das Metamodell besitzt die folgende Struktur, die in Tabelle 7.4 dargelegt wird.

Tabelle 7.4: Übungsbeispiel 2012:
Quellmetamodell gradingsystem.ecore: Anzahl an Elementen

Element	Anzahl
Enumerations	1
Klassen	9
Vererbungen	2
Referenzen (kein containment)	1
Referenzen (containment)	7
Bidirektionale Referenzen (containment)	0
Maximale Anzahl an Attributen in einer Klasse	2

Skalierbarkeit der Modelle
Die Frage nach der Skalierbarkeit befasst sich vor allem mit der Generierung von Modellinstanzen. Für die Evaluierung der Übungsbeispiele ist die Obergrenze der Anzahl an Modellen, die erzeugt werden, konkret auf den Wert 100 festgelegt. Daher wird für die Frage der Skalierbarkeit, das Programm ohne diese Obergrenze getestet, um eine etwaige Gesetzmäßigkeit zu finden und zu erkennen, ob das Programm ab einer gewissen Anzahl nicht mehr arbeitet. Zuerst werden eigens erstellte Eingabesets, die keine konkreten Metamodelle umfassen, getestet. Diese haben eine unterschiedliche Anzahl an Klassen und Instanzen, welche dann um Klassenelemente, wie Attribute und Referenzen, ergänzt werden. Danach werden die Übungsbeispiele noch getestet.

7.3 Durchführung der Fallstudie[4]

7.3.1 Modellgenerierung[5]

Ein wichtiger Bestandteil des Programms ist die automatische Testdatengenerierung. Da in DLV alle möglichen Testfälle berechnet werden, ist es wichtig zu erkennen, ab wann das Programm die Datenmenge nicht mehr verarbeiten kann. Dafür werden Answer Sets mit variierender Anzahl an Klassen, Attributen und Referenzen berechnet. Die Testläufe werden auf einem PC mit einer Intel(R) Core(TM) i7 CPU (960 @ 3,20 GHz 3,20GHz) mit 16 GB RAM durchgeführt. Werte, die in den nachfolgenden Tabellen nicht mehr angegeben werden, können mit dem Programm nicht mehr verarbeitet werden.

In Tabelle 7.5 ist die jeweilige Anzahl an generierten Answer Sets für unterschiedliche maximale Instanzen und Klassenanzahlen abgebildet, wobei keine Attribute und Referenzen existieren.

Als nächstes werden Tests mit zusätzlichen Attributen durchgeführt. Hier ist zu beachten, dass nur single-valued Attribute mit dem Programm verarbeitet werden können und somit der Maximalwert immer 1 ist. Wenn alle Attributwerte immer vorhanden sein

[4] Verfasser: Thomas Franz, BSc & Sabine Wolny, BSc
[5] Verfasser: Sabine Wolny, BSc

Tabelle 7.5: Anzahl an Answer Sets abhängig von der Anzahl an Elementen und Klassen ohne Attribute und Referenzen

maximale Instanzen \ Klassenanzahl	1	2	3	4	5
1	2	4	8	16	32
2	4	16	64	256	1024
3	8	64	512	4096	32768
4	16	256	4096	65536	

müssen, also genau einmal, dann haben diese keinen Einfluss auf die Anzahl an Modellen. Es werden genau so viele generiert, als wären keine Attribute vorhanden. Ist die Untergrenze jedoch auf 0 gesetzt, erhöht das die Anzahl an möglichen Modellen. Die in der Tabelle 7.6 dargestellten Ergebnisse zeigen die Anzahl der Answer Sets, wenn jede vorhandene Klasse ein Attribut besitzt, welches mindestens null und maximal einmal vorkommt.

Tabelle 7.6: Anzahl an Answer Sets abhängig von der Anzahl an Elementen und Klassen bezogen auf ein Attribut pro Klasse

maximale Instanzen \ Klassenanzahl	1	2	3
1	3	9	27
2	9	81	729
3	27	729	19683
4	81	6561	

Durch diese Tests lässt sich erkennen, dass die Anzahl der Möglichkeiten schon für kleine Anzahlen an Klassen sehr groß wird. Prinzipiell ergibt sich aus diesen Tests, dass die Anzahl der Answer Sets nach einer bestimmten Formel berechnet werden kann, welche in Gleichung 7.1 zu sehen ist. Allerdings ist diese nur gültig, wenn es pro Klasse nur ein Attribut gibt und jedes Attribut mindestens null und maximal einmal vorkommt. Diese Anzahl verändert sich, sobald in einer Klasse mehrere Attribute existieren oder sich die Minimalwerte zwischen den einzelnen Attributen unterscheiden.

$$a = (3^k)^n \tag{7.1}$$

Anzahl der Answer Sets (a) für ein Metamodell mit k Klassen
und n maximale Instanzen mit einem Attribut pro Klasse

Für Referenzen kann keine Gesetzmäßigkeit festgestellt werden, da diese zu stark von den vorgegebenen Multiplizitäten abhängen und zusätzlich noch zwischen beliebig vielen Klassen vorhanden sein können. Einen Einblick in die stark wachsende Anzahl der

Modelle geben die Abbildung 7.1, Abbildung 7.2 und Abbildung 7.3. Hier sind für unterschiedliche maximale Instanzen die Ergebnisse dargestellt, wenn entweder eine Klasse sich selber referenziert (Klassenanzahl 1), eine Klasse eine Referenz auf eine andere Klasse hat (Klassenanzahl 2) oder eine Klasse jeweils eine Referenz auf zwei andere Klassen hat (Klassenanzahl 3). Die Abbildung umfasst drei Graphiken, die die Anzahl der Modelle je nach Multiplizität (0...1, 1...1 und 0...*) darstellen, wobei bei zwei Referenzen beide dieselbe Multiplizität aufweisen.

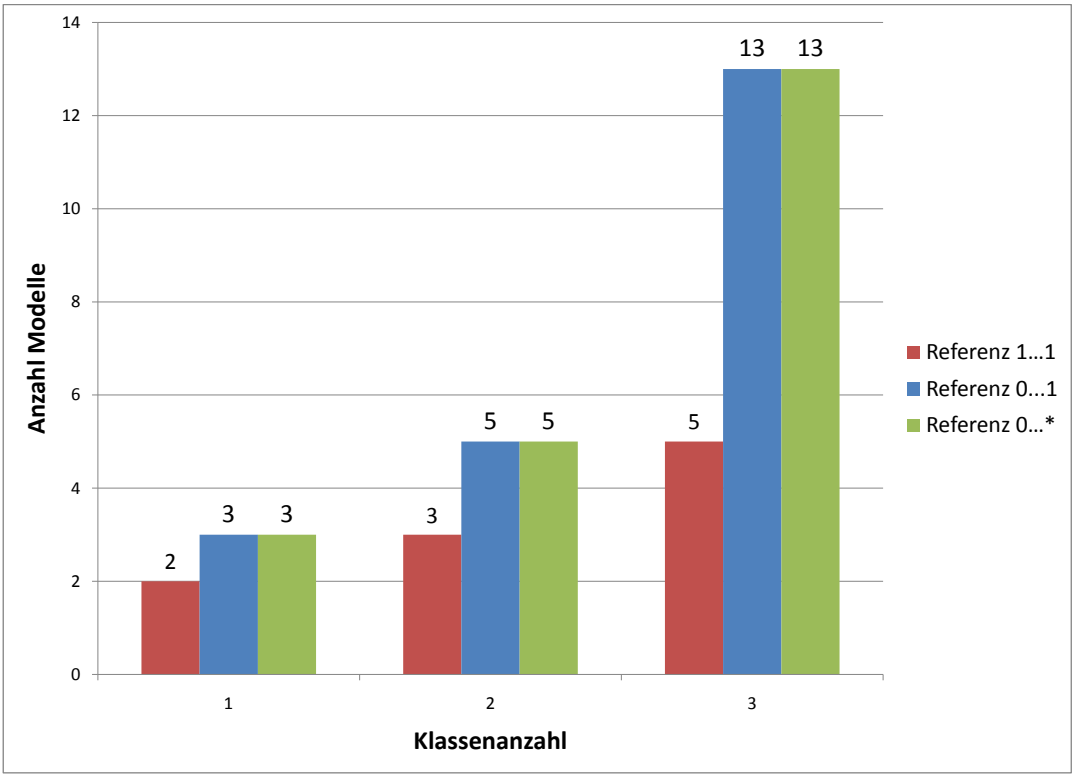

Abbildung 7.1: Anzahl an Answer Sets für Referenzen mit verschiedenen Multiplizitäten für maximal eine Instanz

Bei Abbildung 7.3 gibt es bei 3 Klassen für die Referenzen mit Multiplizität 0...* kein Ergebnis mehr, da dieses nicht generiert werden konnte.

Die dargestellten Ergebnisse der Tests mit Attributen und Referenzen sind nur ein sehr kleiner Ausschnitt aller möglichen Kombinationen, da es für die Elementanzahl in einem Metamodell allgemein keine Grenzen gibt. Außerdem gibt es auch noch Vererbungsbeziehungen, bidirektionale Referenzen und containment Beziehungen, die hier nicht beachtet werden.

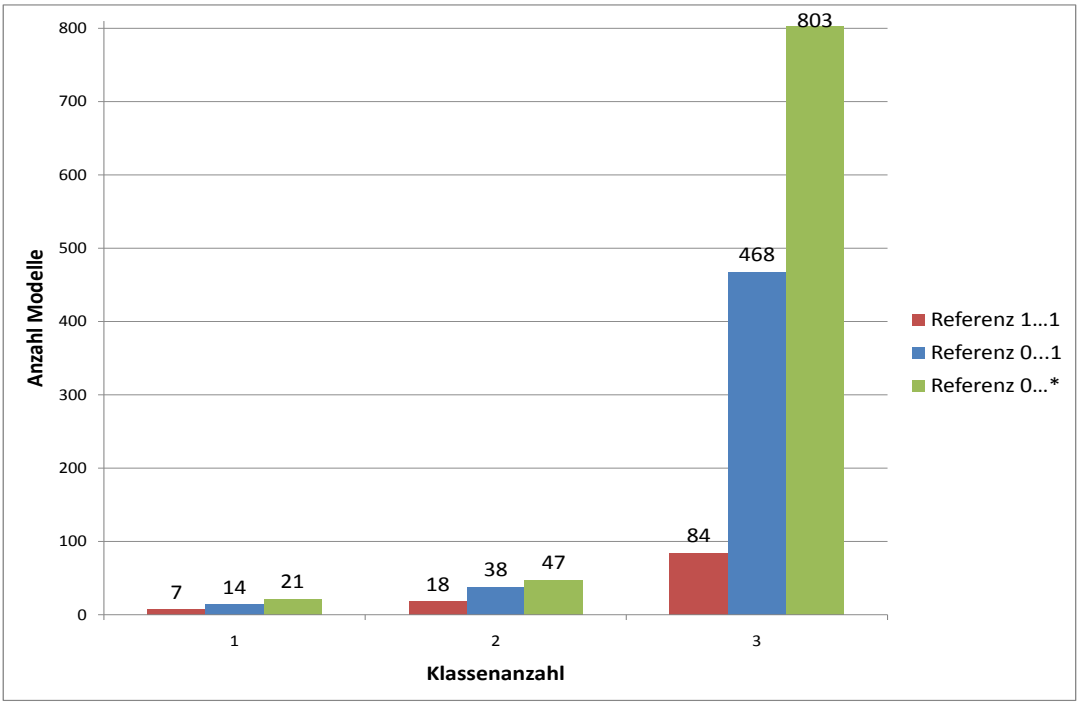

Abbildung 7.2: Anzahl an Answer Sets für Referenzen mit verschiedenen Multiplizitäten für maximal zwei Instanzen

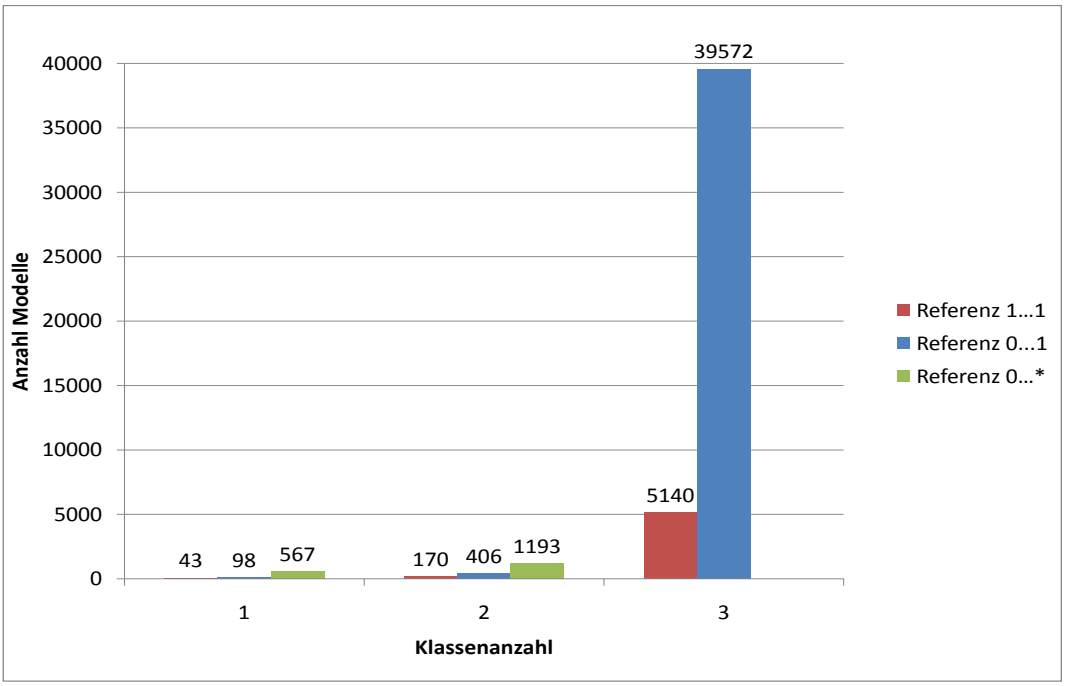

Abbildung 7.3: Anzahl an Answer Sets für Referenzen mit verschiedenen Multiplizitäten für maximal drei Instanzen

Die dargestellten Ergebnisse geben jedoch Aufschluss darüber, wie schnell die Anzahl der generierten Answer Sets wächst.

Einige Eingaben können also schon bei diesen eher einfacheren Tests aufgrund der Datenmenge vom Programm nicht mehr verarbeitet werden. Auch die Tests mit den Übungsbeispielen aus den Jahren 2009 bis 2011 ergeben, dass ohne eine Einschränkung der Anzahl an generierten Answer Sets das Programm nicht erfolgreich beendet werden kann. Die Beispiele sind so komplex, dass eine Vielzahl an Kombinationen möglich ist und verarbeitet werden muss.

Folgende Exceptions werden während der Modellgenerierung geworfen und somit arbeitet das Programm nicht mehr weiter:

- java.lang.OutOfMemoryError: Java heap space
- java.lang.OutOfMemoryError: GC overhead limit exceeded

Der eine Fehler wird verursacht, wenn der "Java heap space"[6] überschritten wird. Der zweite Fehler tritt auf, wenn der Prozess zu viel Zeit für die "Garbage Collection"[7] benötigt und daher wenig bis gar keinen Fortschritt macht. Beide sind abhängig von der Rechnerleistung und treten somit früher oder später auf. In der Tabelle 7.7 sind die Anzahlen an Modellen aufgelistet, die generiert werden, bevor die Fehler auftreten und das Programm abbricht. Die maximale Anzahl für die Instanzen pro Klasse ist bei diesen Tests immer auf 1 gesetzt.

Tabelle 7.7: Anzahl an generierten Modellen bezüglich Übungsbeispielen vor Absturz des Programms

Übungsbeispiel	Anzahl an Modellen
swml2smvcml (2009)	15254
class2relational (2010)	17593
sooml2soopl (2011)	17661

Für das Beispiel aus dem Jahr 2012 "GradingSystem2FileDefinitionSet" kann der Testdurchlauf für maximal eine Instanz pro Klasse erfolgreich durchgeführt werden. Es werden dafür 26119 Modelle in rund 46 Sekunden generiert.

Wenn die Obergrenze für die Anzahl der generierten Answer Sets festgelegt wird, ist es möglich die Fehler, welche aufgrund von zu wenig frei verfügbarem Speicher auftreten, zu verhindern. Bei einem zweiten Rechner Intel(R) Celeron(R) M CPU (440 @ 1,86 GHz 1,86GHz) mit 2 GB RAM treten die Fehler allerdings schon ab 300 Modellen auf. Somit muss beachtet werden, dass je mehr Modelle für Testzwecke erzeugt werden sollen, auch die Rechnerleistung dementsprechend höher sein muss.

[6] `http://docs.oracle.com/cd/E13222_01/wls/docs81/perform/JVMTuning.html`, Zuletzt besucht: 07-08-2013

[7] `https://de.wikipedia.org/wiki/Garbage_Collection`, Zuletzt besucht: 07-08-2013

7.3.2 Testumgebung - 1[8]

Ein Teil der Analyse wird auf einem PC mit einer Intel(R) Core(TM) i3 CPU (550 @ 3,20 GHz 3,20GHz) mit 4 GB RAM durchgeführt.

Zur Erzeugung der Testmodelle mit DLV wird bei allen, in diesem Abschnitt präsentierten Ergebnissen, der Parameter "maxInstanzen" gleich 1 gesetzt. Das bedeutet, dass für jede Klasse aus dem Metamodell nicht mehr als eine Instanz im erzeugten Modell existiert. Insgesamt werden für jeden Durchlauf des Programms 100 Testmodelle erzeugt. Jede Transformation wird mit dieser Einstellung genau 10 Mal durchgeführt.

SWML 2 SMVCML (2009)

Mit diesem Gerät werden für das Beispiel "swml2smvcml" die, in Abbildung 7.4 dargestellten, Ergebnisse erzielt.

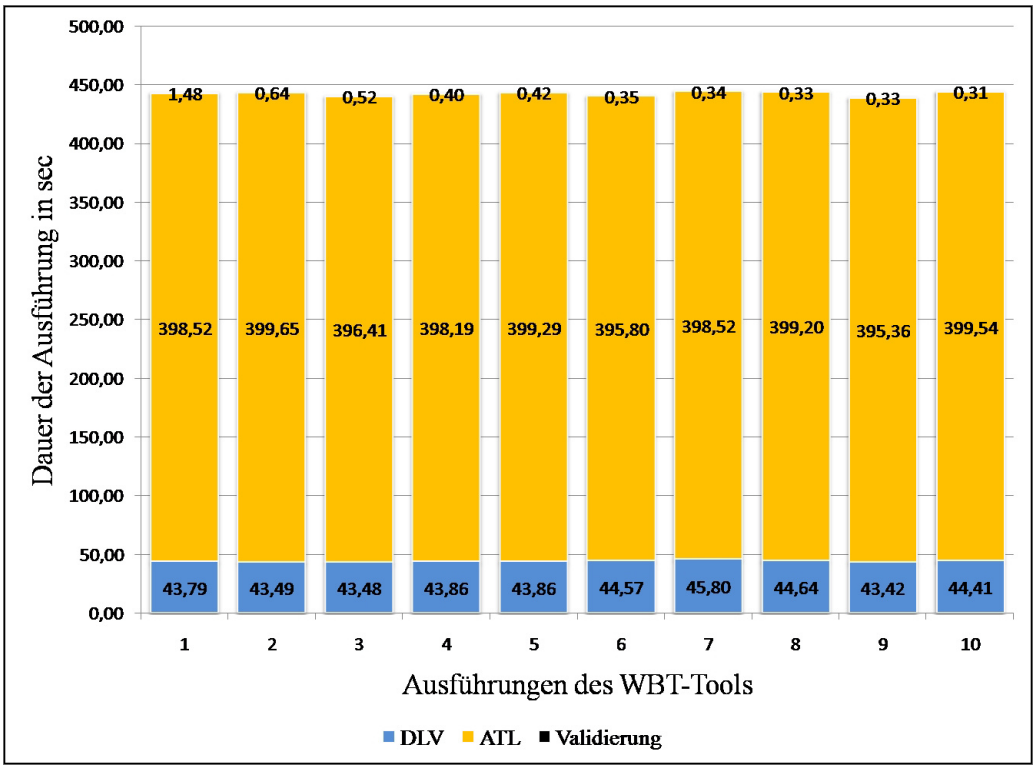

Abbildung 7.4: Ergebnisse der Ausführung der Modelltransformation des Beispiels "swml2smvcml" (mit maxInstanzen = 1)

Die Grafik zeigt, dass die Generierung der Testmodelle sehr wenig Zeit in Anspruch nimmt. Die zeitintensivste Aufgabe ist die Transformation der Modelle. Allerdings sollte bei der Betrachtung der Ergebnisse der Umstand, dass die ATL-Transformation nur

[8] Verfasser: Thomas Franz, BSc

ein einziges Mal (zu Beginn der Ausführung) kompiliert wird, nicht vergessen werden. Dadurch ist es möglich, die Ausführungszeit deutlich zu reduzieren.

Auch für die Validierung der Ergebnisse (die Prüfung, ob die Zielmodelle konform zum Zielmetamodell sind) wird nur sehr wenig Zeit benötigt. Einzig für die erste Ausführung ist der Zeitaufwand höher (im Vergleich zu den übrigen Ausführungen), allerdings mit 1,5 Sekunden kaum erwähnenswert. Insgesamt weisen die einzelnen Aufgaben (Erzeugen der Testmodelle, Transformieren der Modelle sowie deren Validierung) nur sehr geringe Schwankungen auf.
Ein näherer Blick auf die beiden Transformationen mit der längsten (Ausführung 2; siehe Abbildung 7.5) beziehungsweise der kürzesten Laufzeit (Ausführung 9; siehe Abbildung 7.6) für ATL zeigt, dass sich beide nicht sehr stark voneinander unterscheiden. In den beiden Abbildungen werden vor allem die Zeiten, die für das Laden der (Meta-) Modelle und für die Ausführung der eigentlichen Transformation benötigt werden, gegenübergestellt. Dabei zeigt sich, dass die Ausführung der Modelltransformation relativ wenig Zeit in Anspruch nimmt. Dagegen stellt das Laden der (Meta-)Modelle in die entsprechende Programmkomponente eine (zeit)aufwändigere Aufgabe dar.

Alle Linien weisen zu Beginn einen Knick auf. Dieser erklärt sich aus dem Umstand, dass für das jeweils erste Modell auch die Verarbeitung und Kompilierung der ATL-Transformation ausgeführt wird. Hier fallen dann sowohl die Zeiten für das Laden der benötigten Komponenten als auch die Ausführung relativ gering aus.

Die Tatsache, dass die "Java Virtual Machine"[9] zu Beginn der Ausführung von Programmen einige Zeit benötigt, um optimal zu arbeiten, kann hier nicht wirklich festgestellt werden. Einzig der Umstand, dass die Validierung der Zielmodelle nach dem ersten Durchlauf des WBT-Tools mehr Zeit benötigt, weist darauf hin.

[9] `https://de.wikipedia.org/wiki/Java_Virtual_Machine`, Zuletzt besucht: 30-07-2013

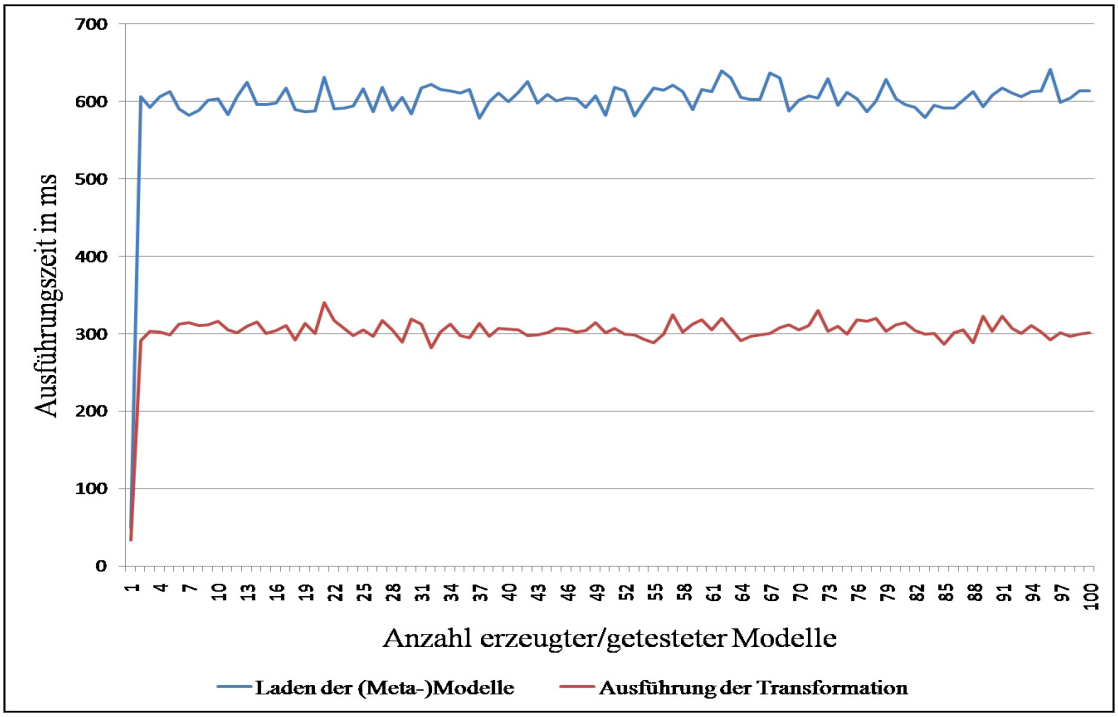

Abbildung 7.5: Analyse des Zeitaufwandes zum Laden der (Meta-)Modelle und Ausführung der Transformation für alle erzeugten Testmodelle (für Ausführung 2)

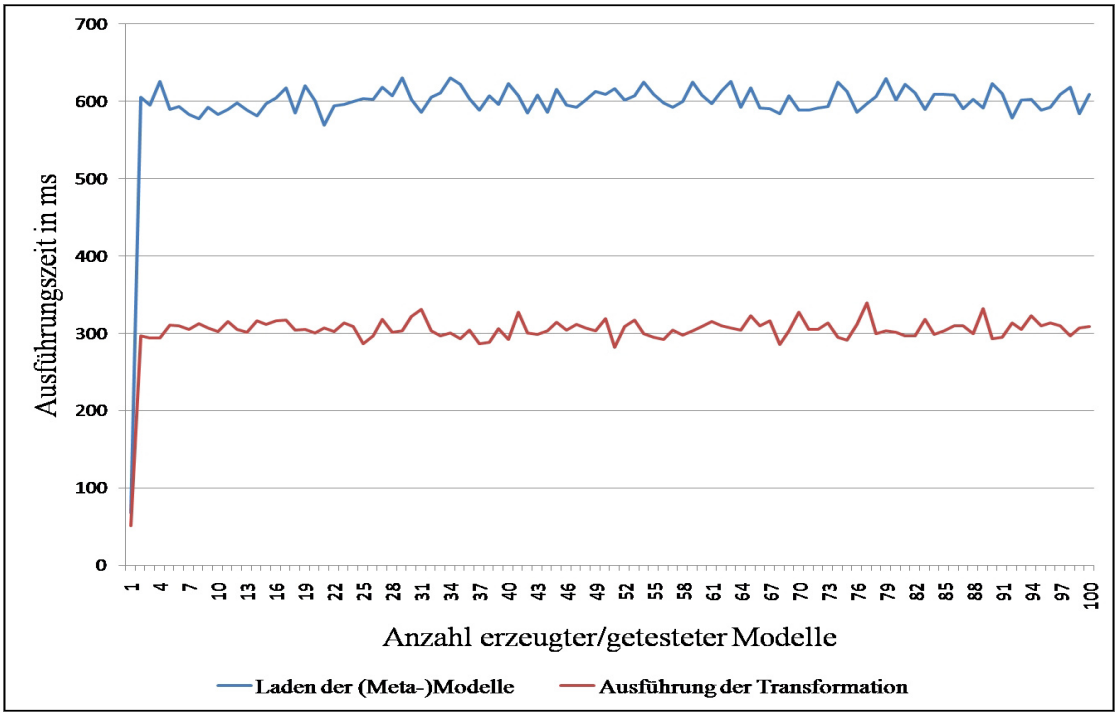

Abbildung 7.6: Analyse des Zeitaufwandes zum Laden der (Meta-)Modelle und Ausführung der Transformation für alle erzeugten Testmodelle (für Ausführung 9)

Dies trifft auch auf die Ergebnisse der weiteren Beispiele zu, die in der Folge vorgestellt werden.

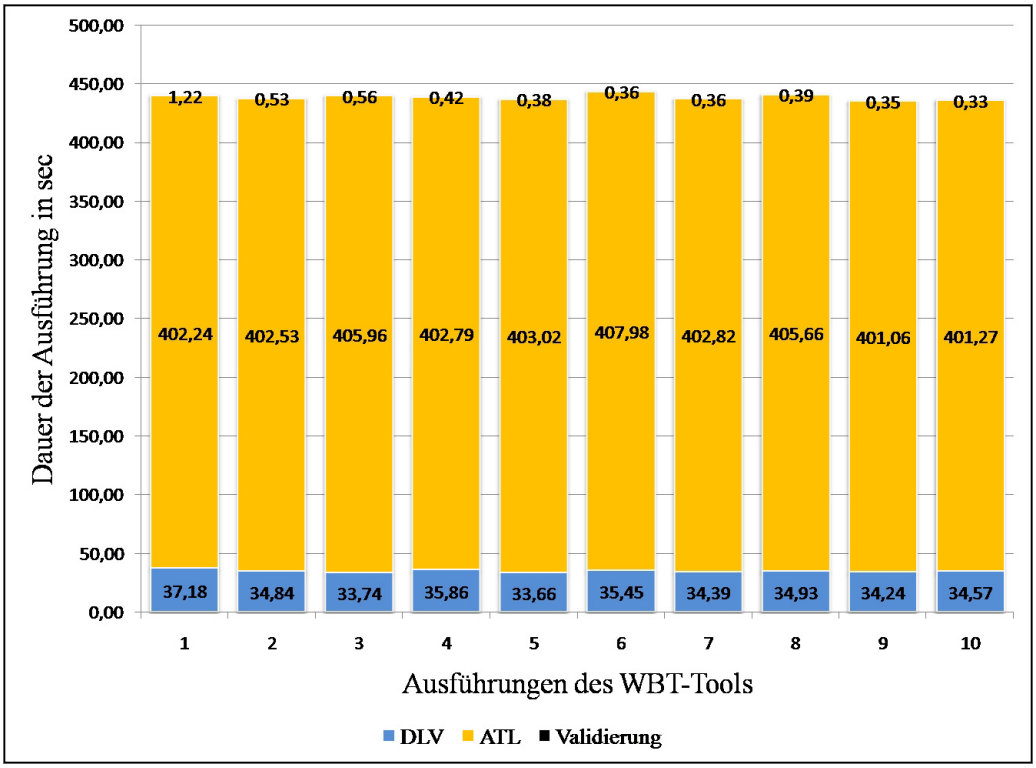

Abbildung 7.7: Ergebnisse der Ausführung der Modelltransformation des Beispiels "class2relational" (mit maxInstanzen = 1)

Class 2 Relational (2010)

In der Abbildung 7.7 sind die Ergebnisse für das Beispiel "class2relational" dargestellt. Im Gegensatz zum vorherigen Beispiel ("swml2smvcml"), bei dem alle Zielmodelle valide Instanzen ihres Metamodells sind, trifft dies im aktuellen Beispiel für kein einziges Ergebnis zu. Hier liefert die Transformation großteils nicht valide Modelle. Ein Teil der Testmodelle kann von der ATL-Transformation gar nicht verarbeitet werden. Der Anteil dieser Modelle liegt für alle 10 Ausführungen konstant bei 28%.

SOOML 2 SOOPL (2011)

Vergleicht man die Ergebnisse für das Beispiel "sooml2soopl" (siehe Abbildung 7.8) mit den beiden vorangegangenen, zeigt sich auch hier ein ähnliches Bild. Allerdings kann man feststellen, dass sowohl für "class2relational" als auch "sooml2soopl" die Generierung der Modelle als auch die Validierung im ersten Durchlauf mehr Zeit benötigen als in den weiteren 9 Durchläufen. Der Unterschied ist allerdings minimal. So macht er für

“sooml2soopl” circa vier Sekunden (ca. 3 Sek. für die Modellgenerierung sowie ca. 1 Sek. für die Validierung) gegenüber dem Durchschnitt der weiteren Ausführungen aus.

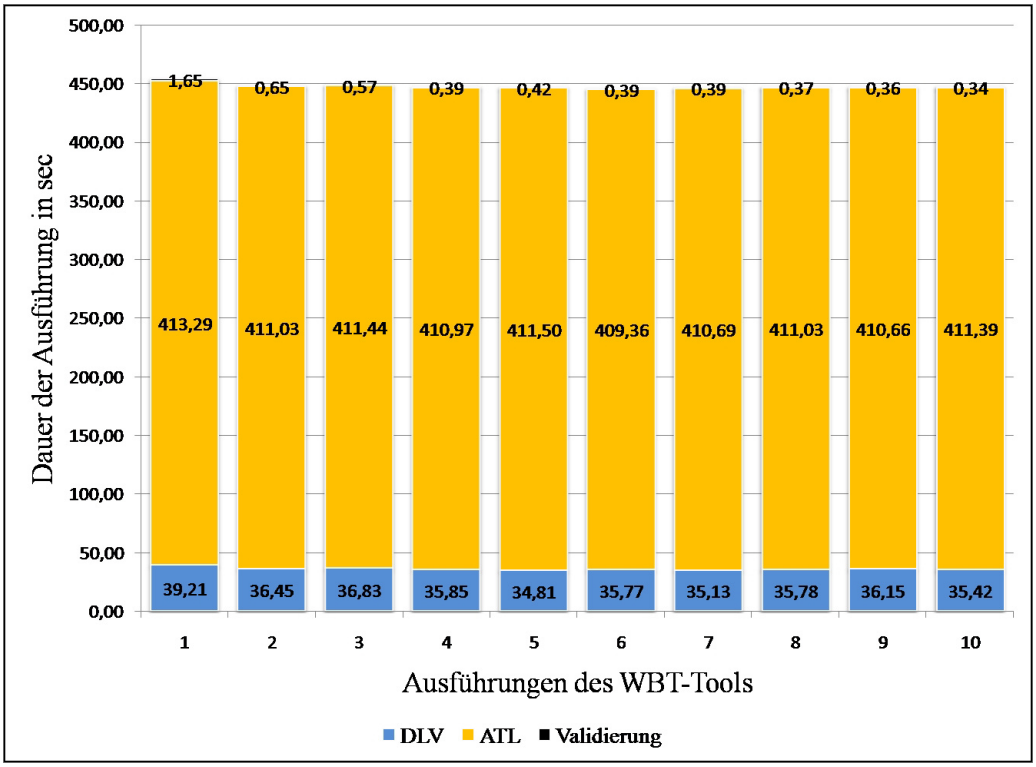

Abbildung 7.8: Ergebnisse der Ausführung der Modelltransformation des Beispiels “sooml2soopl” (mit maxInstanzen = 1)

Auch in diesem Fall können nicht alle Quellmodelle transformiert werden. Wie in Abbildung 7.9 dargestellt, schwankt der Anteil der fehlgeschlagenen Transformationen zwischen den einzelnen Programmläufen. Dies kann als Beleg für die Variabilität der erzeugten Testmodelle angesehen werden. Andernfalls würde sich auch hier ein ähnliches Bild wie im Falle von “class2relational” zeigen, bei dem der Anteil der fehlgeschlagenen Transformationen für alle Programmläufe konstant ist.

Interessant ist, dass die Anzahl der fehlgeschlagenen Transformationen keine gravierenden Auswirkungen auf den Zeitaufwand für diese Aufgabe hat. Ausführungen mit einer hohen Anzahl an Fehlschlägen (wie z.B. Ausführung 5 oder 7) weisen, was die Ausführungszeit von ATL betrifft, keine großen Unterschiede zu solchen mit geringer Fehlerquote (wie z.B. 9) auf.

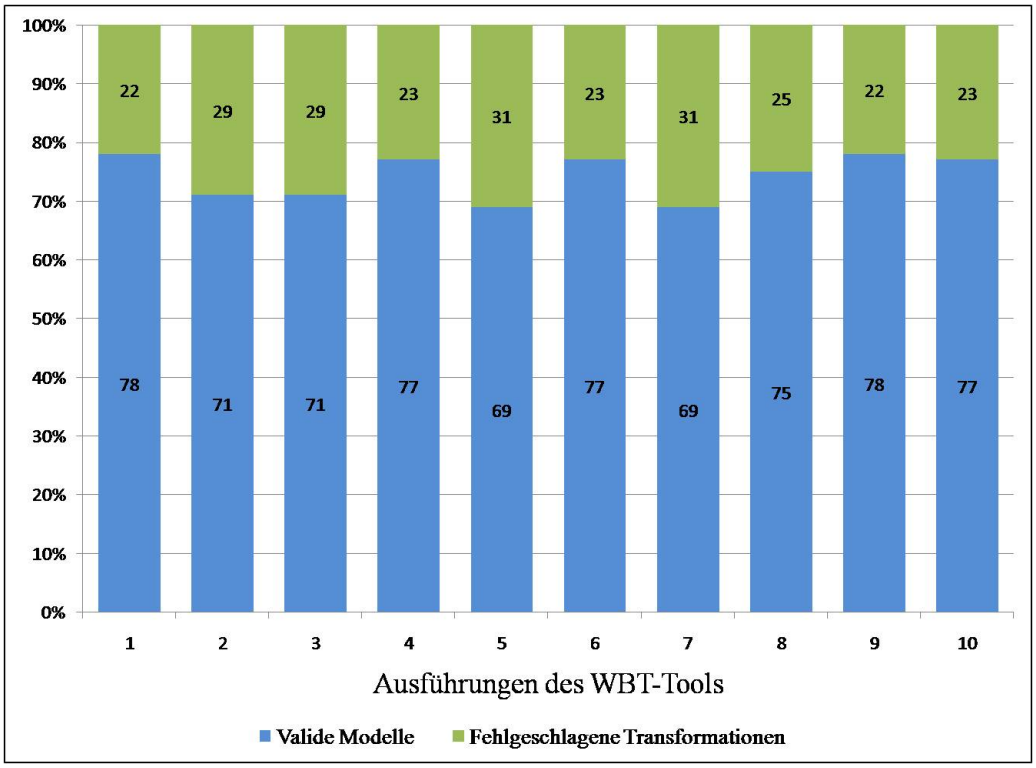

Abbildung 7.9: Erfolgreiche und fehlgeschlagene Transformationen (in % pro Ausführung) der ATL-Transformation für "sooml2soopl"

gradingSystem 2 csv (2012)

Das Beispiel "gradingSystem2csv" kann mit relativ geringem Zeitaufwand (siehe Abbildung 7.10) getestet werden. Das kann allerdings auch daran liegen, dass in jeder Ausführung von 100 Testmodellen nur 9 transformiert werden. In allen anderen Fällen schlägt die Transformation fehl. Außerdem sind diese 9 erzeugten Zielmodelle nicht konform zum entsprechenden Metamodell.

Ein Vergleich der Durchschnittswerte über die jeweils zehn Ausführungen (dargestellt in Abbildung 7.11) zeigt, dass das Beispiel "gradingSystem2csv" mit dem geringsten Aufwand getestet wird. Interessant ist, dass für die anderen drei Beispiele die benötigten Durchschnittszeiten keine gravierenden Unterschiede aufweisen. Einzig für "swml2smvcml" ist die Zeit, die im Mittel für die Erstellung der Testinstanzen benötigt wird, geringfügig größer, während im Durchschnitt geringfügig weniger Zeit für die Transformation benötigt wird.

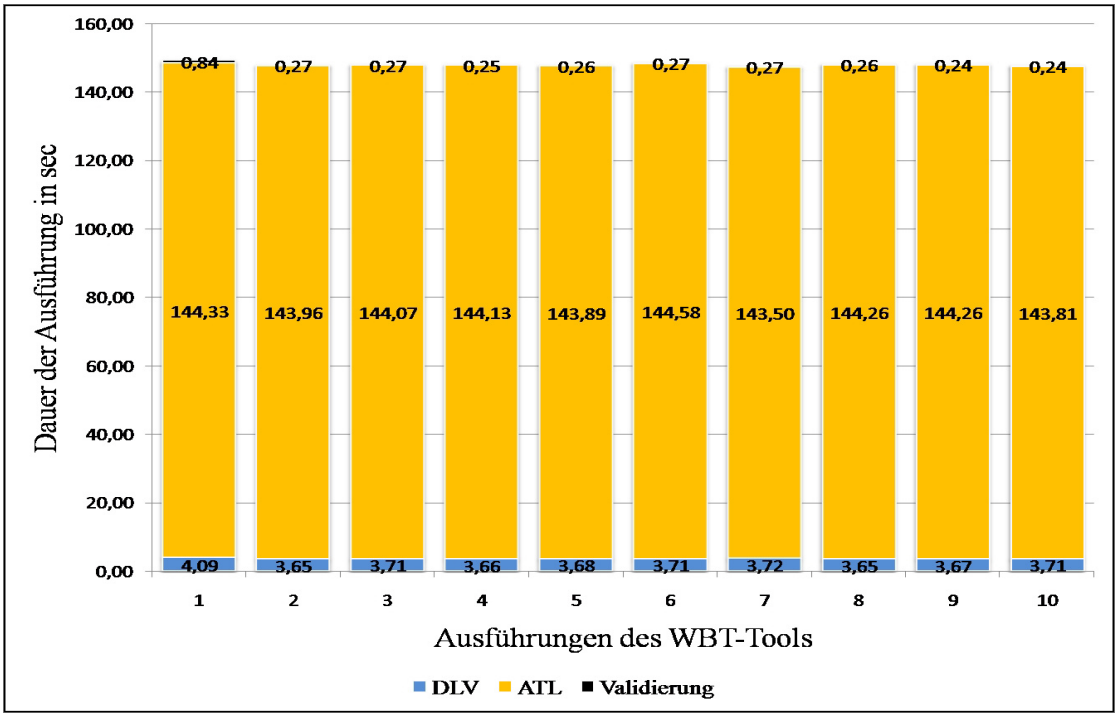

Abbildung 7.10: Ergebnisse der Ausführung der Modelltransformation des Beispiels "gradingSystem2csv" (mit maxInstanzen = 1)

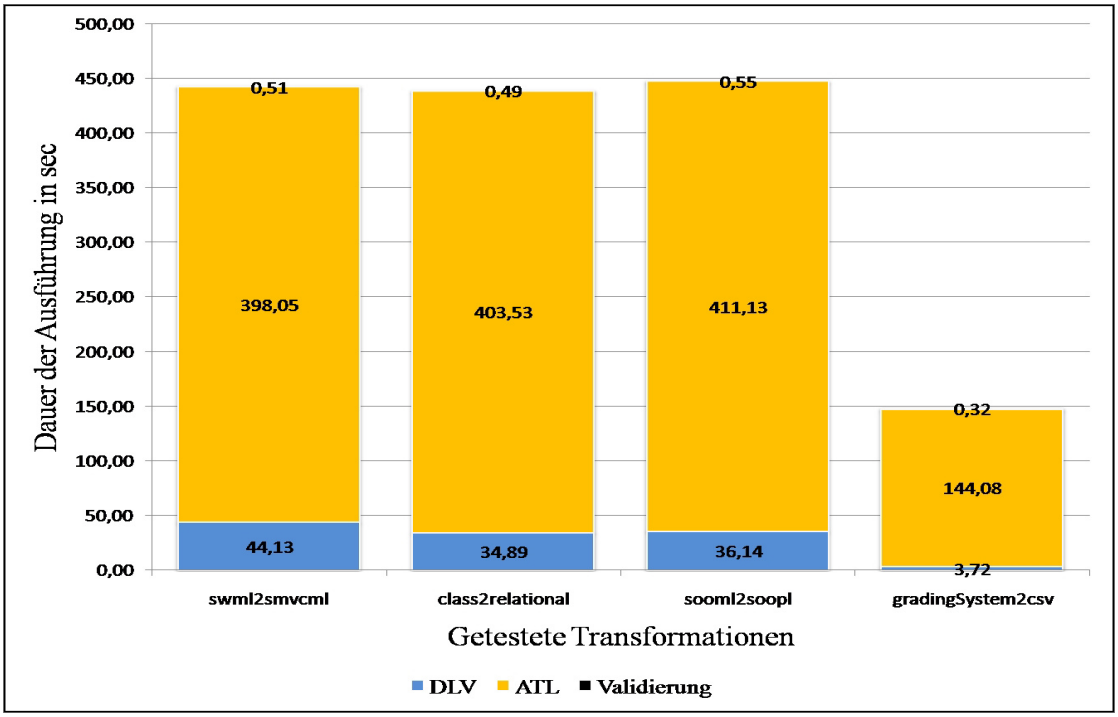

Abbildung 7.11: Vergleich der Durchschnittswerte der Programmdurchläufe der einzelnen Beispiele für Testumgebung 1 (gegliedert nach Teilkomponenten)

RAM Auslastung

Neben der Laufzeit der einzelnen Ausführungen wird auch die Auslastung des RAM Speichers gemessen. Hierfür wird alle 0,2 Sekunden ein entsprechender Messpunkt ermittelt (siehe Unterabschnitt 6.5.1). In Abbildung 7.12 sind die höchsten gemessenen Werte sowie die durchschnittliche Auslastung des RAM-Speichers pro Ausführung für alle Beispiele dargestellt.

Aus diesen Darstellungen können zwei interessante Informationen gewonnen werden:

1. Unabhängig vom Beispiel, fällt die Inanspruchnahme des Arbeitsspeichers jeweils für die erste Programmausführung am geringsten aus. Je öfter das Programm ausgeführt wird, desto größer fällt (im Mittel) die Auslastung des RAM-Speichers aus.

2. Die Speicherauslastung ist, angesichts der zur Verfügung stehenden 4 GB der Testumgebung, in allen Fällen sehr gering.

Diese Beobachtungen werden auch durch die in Abbildung 7.14 ausgewiesenen Werte bestätigt. Auch angesichts der in Listing 7.1 angeführten Parameter der Konfigurationsdatei von Eclipse "eclipse.ini"[10] sind diese Werte sehr gering. Das lässt auch vermuten, dass die Wahl größerer Werte für die angegebenen Parameter keinen zwingenden Einfluss auf die Speicherauslastung oder die Laufzeit des Programms hätte.

```
9  --launcher.XXMaxPermSize
10 256M
11 -showsplash
12 org.eclipse.platform
13 --launcher.XXMaxPermSize
14 256m
15 --launcher.defaultAction
16 openFile
17 -vmargs
18 -Dosgi.requiredJavaVersion=1.5
19 -Xms40m
20 -Xmx2048
```

Listing 7.1: Auszug aus der Konfigurationsdatei "eclipse.ini" der Testumgebung

[10] `http://wiki.eclipse.org/Eclipse.ini`, Zuletzt besucht: 03-08-2013

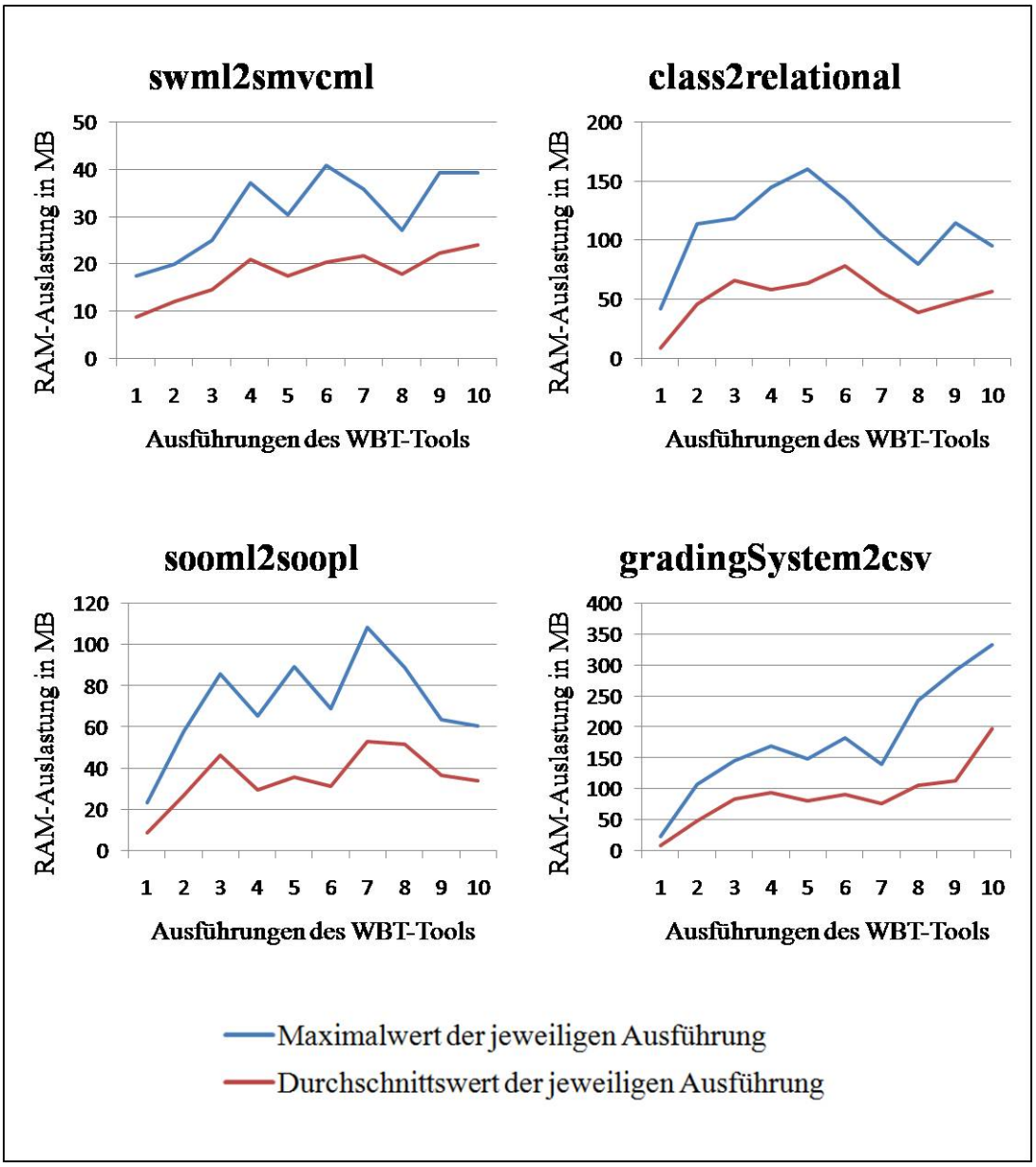

Abbildung 7.12: Durchschnittliche und maximale Auslastung des RAM-Speichers aller Ausführungen der einzelnen Beispiele

Betrachtet man die Auswertungen der RAM-Auslastung genauer, ergibt sich für alle Beispiele im Wesentlichen das in Abbildung 7.13 dargestellte Bild. Die hier ausgewählten Ausführungen zeigen, dass die Schwankungen der Speicherauslastung zu Beginn (für die jeweils 1. Ausführung) sehr gering ausfallen, dafür aber recht häufig auftreten. In späteren Ausführungen gibt es weniger solcher Schwankungen, diese fallen dafür aber umso stärker aus, was auch durch die berechneten Werte für die Standardabweichung in Abbildung 7.14 bestätigt wird. Dieses Bild weisen alle Beispiele mit geringen Unterschieden auf.

Ebenfalls lässt sich aus den aufgezeichneten Daten kein direkter Zusammenhang zwischen dem Ergebnis der Programmausführung und der Speicherauslastung feststellen. So weist das Beispiel “swml2smvcml”, für das in allen Fällen ein valides Zielmodell erzeugt wird, in allen Fällen die geringste Speicherauslastung auf, allerdings schwankt diese zwischen den einzelnen Ausführungen. Auch für das Beispiel “class2relational”, das eine konstante Anzahl an fehlgeschlagenen Transformationen aufweist, schwankt die Speicherauslastung.

Generell lassen die Ergebnisse vermuten, dass die mittlere Auslastung des RAM-Speichers umso höher ausfällt, je mehr Transformationen fehlschlagen (kein Zielmodell liefern). Ein Vergleich der Beispiele deutet darauf hin, auch wenn die Anzahl fehlgeschlagener Modelltransformationen für “sooml2soopl” schwankt (im Mittel aber unter derjenigen von “class2relational” liegt). Die höchste Anzahl an Fehlschlägen weist “gradingSystem2csv” auf, das auch bei der Speicherauslastung die höchsten Werte aufweist.

Die Anzahl an invaliden Zielmodellen spielt anscheinend keine bedeutende Rolle. Diese ist für “class2relational” am höchsten (91). Das einzige weitere Beispiel, für das invalide Modelle erzeugt werden, ist “gradingSystem2csv” (9). Einzig bei einem Vergleich zwischen “swml2smvcml” und “class2relational” lässt sich vermuten, dass die Erzeugung invalider Modelle speicherintensiver ist als die Generierung valider Modelle. Das könnte damit zusammenhängen, dass die Prüfung auf Konformität mit dem Metamodell im ersten Fall aufwändiger ist.

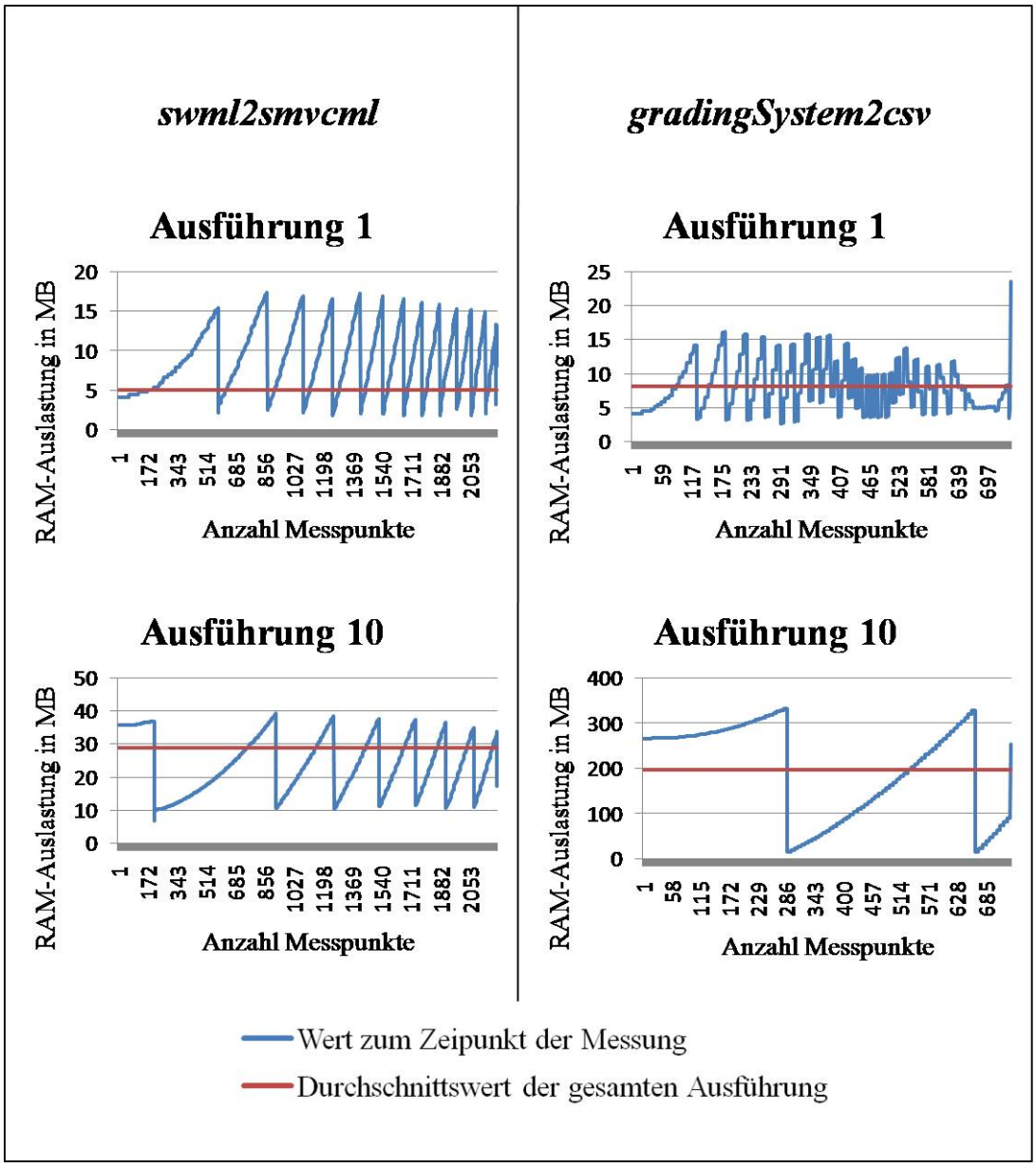

Abbildung 7.13: Durchschnittliche und maximale Auslastung des RAM-Speichers ausgewählter Ausführungen

Beispiel	Kennzahl	Ausführung									
		1	2	3	4	5	6	7	8	9	10
swml 2 smcvml	Mittelwert	8,881	12,223	14,690	21,007	17,511	20,259	21,770	17,881	22,255	23,969
	Standard-abweichung	4,091	4,442	4,780	7,508	6,932	7,749	8,448	5,004	7,729	8,723
class 2 relational	Mittelwert	8,951	45,949	65,685	58,141	63,714	78,358	55,878	39,346	48,530	57,047
	Standard-abweichung	4,181	22,900	28,740	34,888	38,708	35,770	23,389	23,374	26,156	25,583
sooml 2 soopl	Mittelwert	8,843	27,229	46,538	29,621	35,761	31,560	53,236	51,819	36,820	34,038
	Standard-abweichung	3,907	13,076	20,650	16,702	22,426	17,103	29,593	22,279	14,122	12,339
grading-System 2 csv	Mittelwert	8,169	47,520	84,353	94,064	81,139	91,200	76,139	105,899	114,117	198,180
	Standard-abweichung	3,667	22,353	38,422	50,787	43,335	42,016	33,079	53,688	75,901	104,107

Abbildung 7.14: Übersicht über die durchschnittliche Auslastung und Standardabweichung des RAM-Speichers (in MB) für alle Ausführungen der einzelnen Beispiele

7.3.3 Testumgebung - 2[11]

Der zweite Teil der Analyse wird auf einem PC mit einer Intel(R) Core(TM) i7 CPU (960 @ 3,20 GHz 3,20GHz) mit 16 GB RAM durchgeführt.

Es werden die Testläufe mit derselben Konfiguration wie bei der Testumgebung 1 (siehe Unterabschnitt 7.3.2) ausgeführt, um Unterschiede bezüglich Laufzeit zu untersuchen. Der Parameter "maxInstanzen" ist daher auf 1 gesetzt. Für jeden Durchlauf des Programms werden 100 Testmodelle erzeugt, wobei jede Transformation genau 10 Mal durchgeführt wird. Außerdem werden für ausgewählte Übungsbeispiele auch Testläufe durchgeführt, wo der Parameter "maxInstanzen" auf den Wert 2 gesetzt ist.

SWML 2 SMVCML (2009)

In Abbildung 7.15 sind die Ergebnisse für das Übungsbeispiel "swml2smvcml" dargestellt. Hier lässt sich klar erkennen, dass die Laufzeit geringer ist als bei der Testumgebung 1. Im Durchschnitt wird eine Ausführung in 147,96 Sekunden durchgeführt und benötigt somit nur 33,4% der Zeit, die mit dem leistungsschwächeren PC notwendig ist. Wie schon in der Testumgebung festgestellt, braucht die ATL Transformation am längsten, wobei hier vor allem das Laden der (Meta-) Modelle zeitaufwändiger ist als die Transformationsausführung selbst. Interessant ist, dass für diese Transformation nur valide Instanzen erzeugt werden.

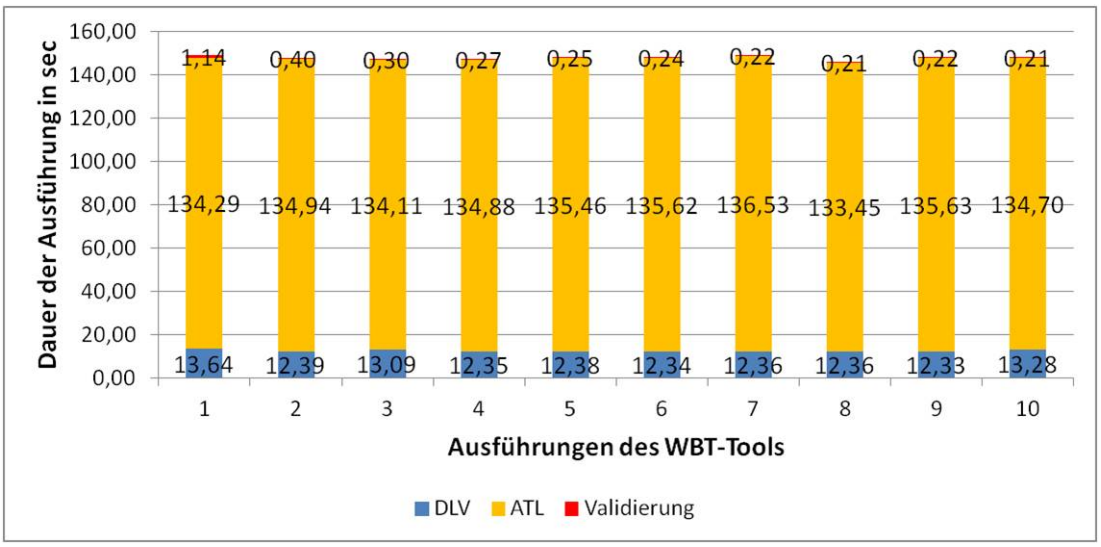

Abbildung 7.15: Ergebnisse der Ausführung der Modelltransformation des Beispiels "swml2smvcml" (mit maxInstanzen = 1)

Da für den Testlauf mit maximal einer Instanz nur valide Modelle erzeugt werden, wird auch ein Testlauf mit maximal zwei Instanzen durchgeführt. Allerdings werden auch

[11] Verfasser: Sabine Wolny, BSc

bei den zehn Ausführungen wieder nur valide Instanzen erzeugt und daher keine Fehler festgestellt. Die Ergebnisse der Ausführungen sind in der Abbildung 7.16 dargestellt. Klar erkennbar ist, dass bei diesen Tests die Modellgenerierung und die ATL Transformation länger brauchen. Die Zeit für die Validierung ist in etwa gleich geblieben. Im Durchschnitt dauert eine Ausführung mit 321,62 Sekunden zweieinhalb Mal so lange wie mit nur einer maximalen Instanz.

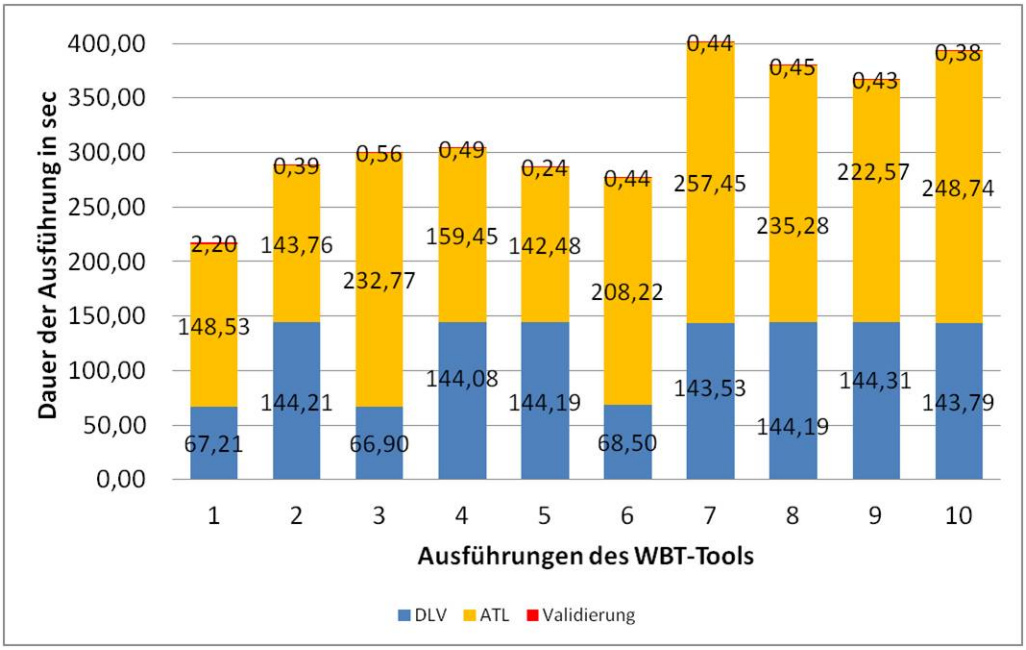

Abbildung 7.16: Ergebnisse der Ausführung der Modelltransformation des Beispiels "swml2smvcml" (mit maxInstanzen = 2)

Class 2 Relational (2010)

Auch bei dem Übungsbeispiel aus dem Jahr 2010 zeigt sich ein ähnliches Bild wie in der ersten Testumgebung. Keines der Zielmodelle ist ein valides Modell und wieder genau für 28% der Modelle pro Ausführung kann die Transformation nicht durchgeführt werden. Auffällig ist, dass die Dauer der Ausführungen stärker schwankt, wie in Abbildung 7.17 zu sehen ist.

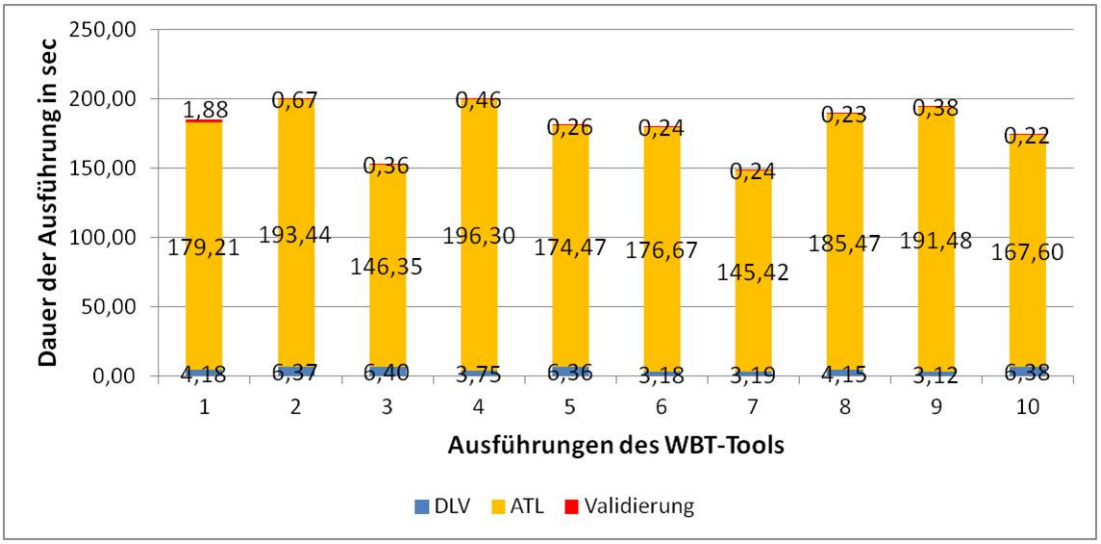

Abbildung 7.17: Ergebnisse der Ausführung der Modelltransformation des Beispiels "class2relational" (mit maxInstanzen = 1)

Diese Schwankungen finden sich auch in der RAM Auslastung wieder. So liegt die durchschnittliche Auslastung bei der zweiten Ausführung, welche am längsten brauchte, bei 79,6 MB, während die kürzeste Ausführung (Ausführung 7) 228,6 MB vom Arbeitsspeicher verwendet. Dies zeigt sich auch im Vergleich der einzelnen gemessenen Werte, wobei die Ausführung 2 mehr Messpunkte hat, da die Durchführung gesamt mehr Zeit benötigte (siehe Abbildung 7.18).

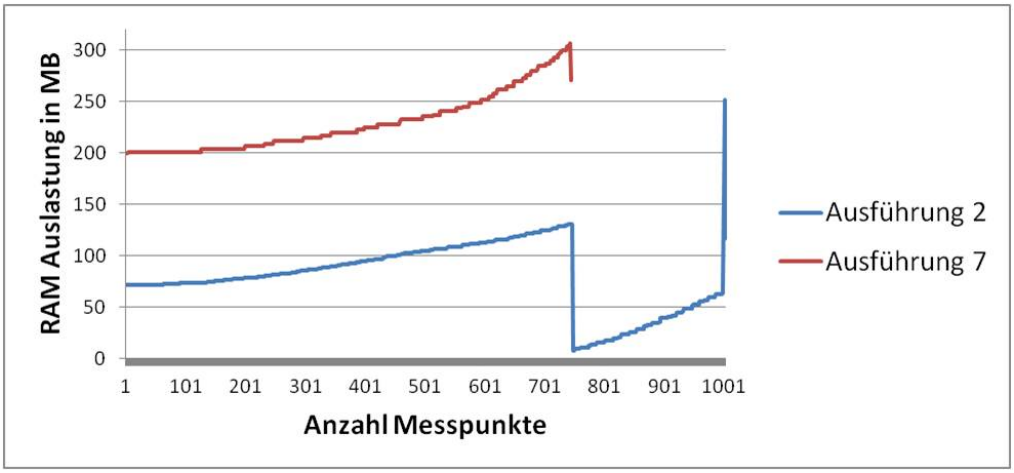

Abbildung 7.18: Vergleich der RAM Auslastung von der Ausführung 2 und 7 der Modelltransformation des Beispiels "class2relational" (mit maxInstanzen = 1)

SOOML 2 SOOPL (2011)

Für das Übungsbeispiel aus dem Jahr 2011 "sooml2soopl" zeigt sich ein ähnliches Bild wie bei dem Testdurchlauf in der ersten Testumgebung. Die Ergebnisse bezüglich der Laufzeit sind in der Abbildung 7.19 dargestellt. Interessant ist, dass entweder die Transformation fehlschlägt oder valide Zielmodelle erzeugt werden, jedoch keine invaliden Instanzen existieren. Die Anzahl von fehlgeschlagenen Transformationen schwankt zwischen 25 (Ausführung 1) und 34 (Ausführung 4 und 7). Dies hat jedoch keine Auswirkungen auf die Laufzeit. Die durchschnittliche Laufzeit beträgt auch hier mit 147,07 Sekunden nur rund 33% der Laufzeit von der ersten Testumgebung.

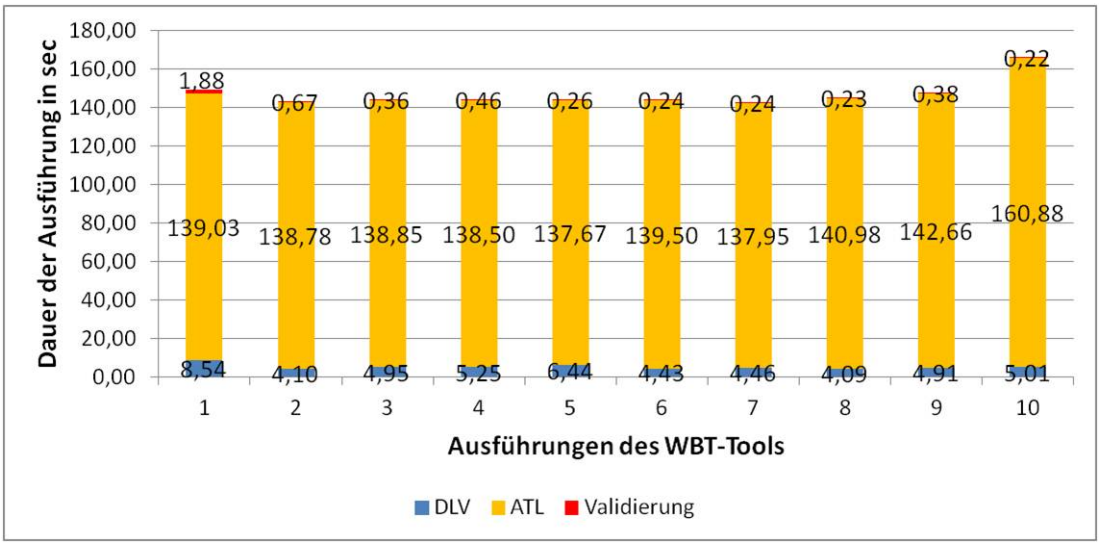

Abbildung 7.19: Ergebnisse der Ausführung der Modelltransformation des Beispiels "sooml2soopl" (mit maxInstanzen = 1)

gradingSystem 2 csv (2012)

Die Ergebnisse des Testdurchlaufes für das "gradingSystem2csv" gleichen jenen des Tests in der ersten Testumgebung. Es werden 91 Modelle nicht transformiert und die restlichen 9 Zielmodelle sind keine validen Instanzen. In der Laufzeit gibt es im Vergleich zu den anderen Übungsbeispielen in dieser Testumgebung keine großen Unterschiede.

Dieses Beispiel wurde außerdem mit dem Wert 2 für den Parameter "maxInstanzen" getestet. Interessant ist hier, dass nun alle Transformationen für alle Ausführungen durchgeführt werden können. 76 Zielmodelle sind valide Instanzen zu ihrem Metamodell. Die restlichen Zielmodelle sind leer, da hier keine Transformationsregel ausgeführt wird. Der Grund hierfür ist, dass die existierenden Elemente des Quellmodells nur innerhalb einer Regel für andere Elemente transformiert werden würden. Außerdem ist im Unterschied zu dem Übungsbeispiel "swml2smcvml" zwischen den Laufzeiten von maximal einer beziehungsweise maximal zwei Instanzen kein großer Unterschied, wie in Abbildung 7.20 zu sehen ist. Der Grund dafür ist wahrscheinlich, dass das Quellmetamodell

weniger komplex als bei dem Beispiel aus dem Jahr 2009 ist. Dadurch müssen trotz der höheren maximalen Anzahl von Instanzen nicht so viele Regeln bei der Modellgenerierung beachtet werden. Einmal dauert die Ausführung mit maximal einer Instanz länger und ein anderes Mal die Ausführung mit maximal zwei Instanzen.

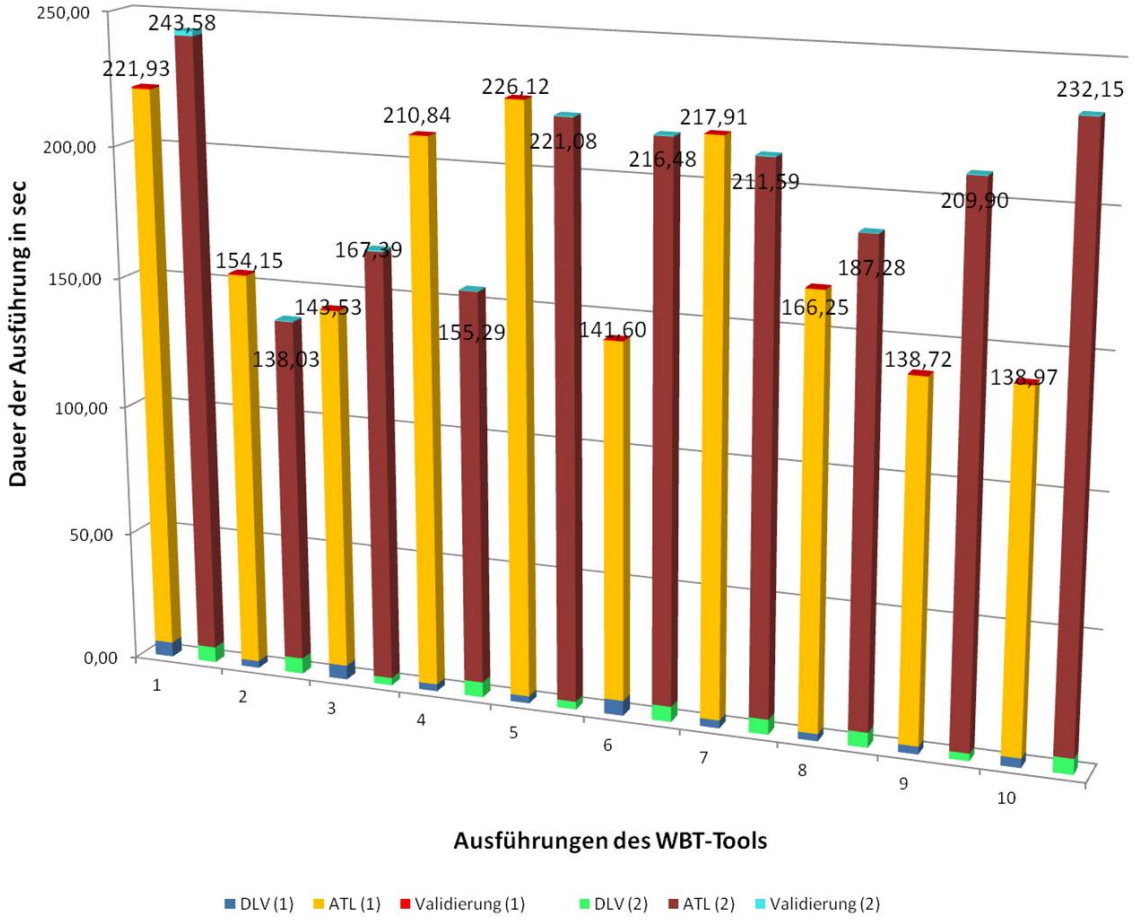

Abbildung 7.20: Vergleich der Laufzeit mit maximal einer Instanz (1) und maximal zwei Instanzen (2) der Ausführung der Modelltransformation des Beispiels "gradingSystem2csv", wobei die Gesamtdauer pro Ausführung in Sekunden angegeben ist

7.4 Diskussion der Ergebnisse[12]

7.4.1 Skalierbarkeit von den generierten Modellen

Aufgrund der Analyse der Anzahl an Modellen mit unterschiedlicher Klassen- beziehungsweise Instanzenanzahl lässt sich erkennen, dass es, solange keine Attribute und Referenzen existieren, eine allgemeine Formel zum Bestimmen der Modellanzahl gibt (siehe Gleichung 7.2). Da die generierten Answer Sets immer auch das leere Modell, welches nicht weiter beachtet wird, enthalten, wird dieses bei der Anzahl abgezogen.

$$m = (2^k)^n - 1 \tag{7.2}$$

Anzahl der Modelle (m) für ein Metamodell mit k Klassen
und n maximale Instanzen ohne Attribute und Referenzen

Diese Berechnung ist ein allgemeiner Richtwert, da in der Regel keine Metamodelle mit nur einer bestimmten Klassen- und Instanzenanzahl ohne Attribute und Referenzen getestet werden. Die Anzahl der möglichen Answer Sets steigt, wenn Attribute erlaubt sind. Das entwickelte Tool kann nur single-valued Attribute verarbeiten, daher ist die Obergrenze immer 1. Allerdings ist als minimaler Wert 0 oder 1 möglich. Sollte der vorgegebene Minimalwert zum Beispiel 0 sein, dann ergibt das für die Answer Sets, dass ein Modell mit dem Attribut und dasselbe Modell noch einmal ohne Attribut existiert. Wenn jetzt mehrere Instanzen ein Attribut haben können oder nicht, werden alle möglichen Kombinationen generiert und als Answer Set zurückgeliefert. Für den Fall, dass alle Attribute auf jeden Fall vorkommen müssen, hat dies auf die Anzahl der Modelle keine Auswirkungen, da keine verschiedenen Kombinationen möglich sind.

Neben Attributen spielen Links zwischen Instanzen eine Rolle, denn diese können im Allgemeinen null Mal bis beliebig oft vorkommen, wobei bei dem entwickelten Tool die festgelegte Obergrenze 100 ist. Es werden prinzipiell alle möglichen Kombinationen generiert.

Die Anzahl der Modelle ist also abhängig von der Anzahl an Kombinationsmöglichkeiten. Es werden alle Kombinationen über die Instanzen (existiert/existiert nicht), Attribute (existiert/existiert nicht) und Links (0 bis 100 Mal) gesucht und ausgegeben. Die einzige Möglichkeit diese exponentiell wachsende Anzahl der Modelle wieder einzudämmen, ist mit Hilfe von gegebenen Bedingungen (Constraints) für Attribute und Links. Wenn in dem Metamodell konkrete Werte für Minimum und Maximum von Attributen und Referenzen festgelegt sind, verringert sich dadurch auch die Anzahl der möglichen Lösungen. Bei einer 1 zu 1 Referenz zum Beispiel, muss der Link genau einmal vorkommen und es gibt keine weiteren Möglichkeiten. So wird die Anzahl der Lösungen abhängig von den Constraints wieder kleiner.

Im entwickelten Programm werden die Modelle in einen `modelBufferedHandler` gespeichert und gleich wieder weiterverarbeitet um eine XMI Repräsentation zu erhalten.

[12] Verfasser: Thomas Franz, BSc

Das Programm stößt hier an seine Grenzen, wenn die “heap size”[13] der Java Maschine, welche vom Arbeitsspeicher abhängig ist, überschritten wird. Zwar liefert der DLV Solver immer noch Lösungen, diese können jedoch nicht mehr von dem Programm verarbeitet werden. Wann die “heap size” überschritten wird, ist vor allem von der Anzahl an Elementen, Vererbungen und Referenzen im Metamodell abhängig.

7.4.2 Komplexität der Laufzeit und Speicherbedarf

Aufgrund der Ergebnisse in den zwei Testumgebungen (siehe Unterabschnitt 7.3.2 und Unterabschnitt 7.3.3) lässt sich erkennen, dass das Programm für die festgesetzten Parameter, wie maximal ein oder zwei Instanzen und 100 generierte Zielmodelle, in einem angemessenen Rahmen läuft. Allerdings verbessern sich die Laufzeiten, wenn ein leistungsstärkerer Computer verwendet wird, wie in der Abbildung 7.21 zu sehen ist. Die Ergebnisse der RAM Auslastung (siehe Unterabschnitt 7.3.2) zeigen, dass diese im Vergleich zum verfügbaren Speicher gering ausfällt. Wenn jedoch mehr Zielmodelle und mehr Instanzen erlaubt werden, steigt die Auslastung stark an und es kann aufgrund zu geringer Speicherkapazität (Java heap size) zu einem Absturz kommen.

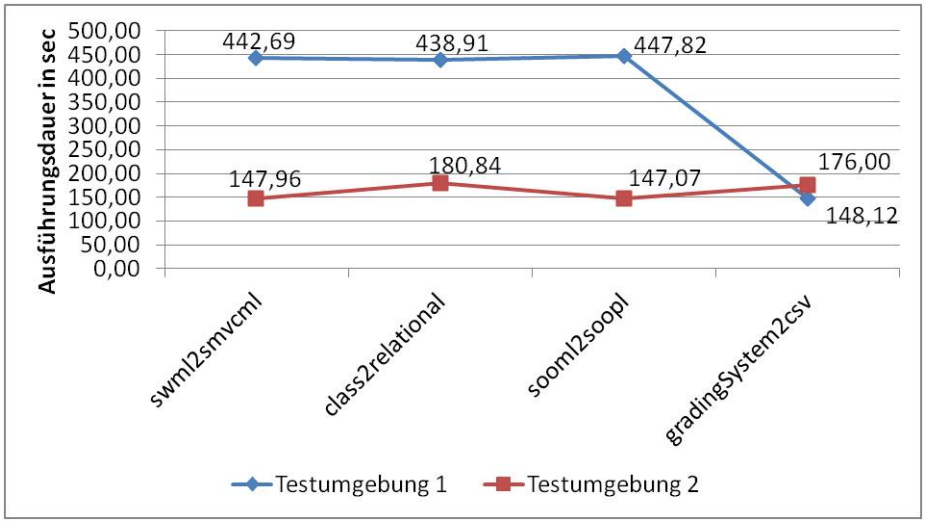

Abbildung 7.21: Vergleich der durchschnittlichen Laufzeiten der Übungsbeispiele in den zwei Testumgebungen (mit maxInstanzen=1)

7.4.3 Fehleranalyse

Die häufigsten Fehler, die bei der Ausführung der Beispiele zur Evaluierung auftreten, betreffen den Zugriff auf nicht existierende/definierte Elemente während der Transformation. Auch die Konformität der Zielmodelle zu ihrem entsprechenden Metamodell ist

[13] `http://docs.oracle.com/cd/E13222_01/wls/docs81/perform/JVMTuning.html`, Zuletzt besucht: 07-08-2013

nicht immer gegeben. Dies liegt in allen Fällen (unabhängig vom Beispiel) daran, dass gewisse Elemente, die laut Metamodell im Modell enthalten sein sollten, nicht von der Transformation erzeugt werden.

Dies zeigt sich auch bei den Tests der Studentenabgaben von den Übungsbeispielen. In der Tabelle 7.8 finden sich die Ergebnisse dieser Auswertungen. Dabei zeigt sich, dass die Zahl der validen Modelle für die einzelnen Beispiele schwankt.

Tabelle 7.8: Übersicht über die Ergebnisse der Studentenabgaben

Beispiel	valide Modelle	nicht valide Modelle	fehlgeschlagene Transformationen
swml2smvcml	18	2	17
class2relational	4	2	20
sooml2soopl	33	2	6
gradingSystem2csv	14	9	31

In diesem Zusammenhang interessant ist ein Blick auf diejenigen Transformationen, die fehlgeschlagen sind bzw. welche Fehler dafür verantwortlich sind. Diese Informationen sind in der Tabelle 7.9 detaillierter dargestellt.

Tabelle 7.9: Übersicht über die aufgetretenen Fehler bei Ausführung der Studentenabgaben

Fehlerart \ Beispiel	swml2 smvcml	class2 relational	sooml2 soopl	grading-System2csv
Fehler bei der Erstellung der erweiterten Kopie (“StringIndexOutOfBoundsException”)	2	4		6
ATL wird nicht kompiliert (“FileNotFoundException”)	8	16	2	23
Laufzeitfehler (“VMException”)				
Operation not found	4			
String index out of range	2		4	
Cannot find reference model	1			
Literal integer does not exist in enumeration FieldType				1
Trying to register several rules as default for element				1
Summe	17	20	6	31

Ein Grund dafür, dass nicht alle ATL-Transformationen erfolgreich ausgeführt werden können, ist, dass die Formatierung der Originaldateien nicht vom Programm verarbeitet werden kann. Das betrifft die Erzeugung der erweiterten Kopie (siehe Ab-

schnitt 6.4). Hier kann zum Beispiel eine “StringIndexOutOfBoundsException” beim Zugriff auf Teilstrings die Ursache für eine fehlgeschlagene Transformation sein.

Außerdem kann die Formatierung auch Ursache für eine andere Art von Fehlern sein: So kann etwa eine fehlerhafte Anzahl an Klammern, die die einzelnen Blöcke (z.B. Regeln; siehe Listing 7.2) definieren, dazu führen, dass die Transformation nicht kompiliert werden kann.

```
1 CompileTimeError: 60:4 mismatched input 'do' expecting RCURLY
2 CompileTimeError: 64:2 mismatched input 'to' expecting EOF
3 java.io.FileNotFoundException:
```

Listing 7.2: Beispiel für Zugriff auf nicht vorhandenes Element (Transformation “class2relational”)

Relativ viele Fehler in den Abgaben sind dadurch verursacht. Ein weiteres Beispiel für eine fehlerhafte Formatierung ist in Listing 7.3 angeführt. Hier wird versucht auf die Variable “model” zuzugreifen, die allerdings nicht definiert ist.

```
1 CompileTimeError: 49:47-49:52 variable 'model' undefined
2 java.io.FileNotFoundException: ..\Tool\OUTPUT\temp\extdATL.asm (Das ↩
    System kann die angegebene Datei nicht finden)
```

Listing 7.3: Beispiel für fehlerhafte Formatierung (Transformation “gradingSystem2csv”)

Die beiden genannten Fehlerquellen betreffen allerdings nur einzelne Spezialfälle der Formatierung der ursprünglichen ATL-Transformation.

Die Fehler werden vom Programm erkannt und auch angezeigt. Dies kann dabei helfen, die entsprechenden Teile der Transformation hinsichtlich der Formatierung anzupassen oder die Programmierung zu verbessern.

Außerdem können mit dem WBT-Tool auch Fehler gefunden werden, die während der Laufzeit auftreten (“VMException”). Ein Beispiel dafür ist der versuchte Zugriff auf Operationen, die in der ATL-Transformation nicht gefunden werden können (siehe Listing 7.4).

Auch während der Laufzeit kann es vorkommen, dass das Programm auf (Teil-) Strings zugreifen möchte. Wenn dieser nicht vorhanden ist und der versuchte Zugriff folglich fehlschlägt, wird dies als “String index out of range”-Fehler ausgegeben. Dieses Problem trat allerdings nur zwei Mal während der Prüfung der Studentenabgaben für das Beispiel “swml2smvcml” auf.

```
1  Beispiel 1:
2
3  org.eclipse.m2m.atl.engine.emfvm.VMException: Operation not found: ↩
       atlkonvertierung : ASMModule.istSuperEntity(org.eclipse.emf.ecore.impl↩
       .DynamicEObjectImpl)
4    at __applyEntityType2Entity#51(atlkonvertierung.atl[190:41-190:72])
5    ...
6
7  ------------------------
8
9  Beispiel 2:
10
11 org.eclipse.m2m.atl.engine.emfvm.VMException: Operation not found: OUT!<↩
       unnamed>.__resolve__(org.eclipse.m2m.atl.engine.emfvm.lib.ASMModule)
12   at CreateSupportedOperation#42(ue2Modeltransformation.atl↩
         [111:6-111:40])
13     local variables: self=ue2Modeltransformation : ASMModule, i=IN!<↩
           unnamed>, s=OUT!<unnamed>
14   at __applyIndexPage2EntityController#110(ue2Modeltransformation.atl↩
         [94:33-94:74])
15   ...
```

Listing 7.4: Beispiele für Fehlermeldungen, die während der Ausführung der Studentenabgaben entdeckt werden

Hier tritt auch ein einzigartiges Problem auf, nämlich der fehlgeschlagene Zugriff der Transformation auf das Zielmetamodell. Angezeigt wird das mit der Fehlermeldung "Cannot find reference model" (siehe Tabelle 7.9).

Während der Durchführung der Studentenabgaben für das Beispiel "gradingSystem2csv" wurden zwei weitere, unterschiedliche Fehler gefunden. Einer davon ist darauf zurückzuführen, dass ein bestimmtes Literal nicht in der entsprechenden Enumeration gefunden werden konnte. Dies war auch der Grund für die fehlgeschlagenen Transformationen für das Beispiel "sooml2soopl" während der Laufzeitmessung (siehe Unterabschnitt 7.3.2 und Unterabschnitt 7.3.3).

Ein weiteres Problem, das auftritt, ist der Versuch von ATL mehrere Regeln als Standardregeln für ein bestimmtes Element zu registrieren. Dieser Fehler dürfte auf unsaubere Regeldefinitionen in der Transformationsdatei zurückzuführen sein.

Wie bereits erwähnt, können auch die Transformationen während der Messungen der Laufzeit, beziehungsweise der Auslastung des RAM-Speichers nicht immer fehlerfrei ausgeführt werden. Teilweise ist dies auf Fehler zurückzuführen, die in diesem Abschnitt bereits erwähnt wurden. So schlugen Ausführungen für das Beispiel "gradingSystem2csv" fehl, weil bestimmte Operationen nicht gefunden werden konnten (siehe als Referenz Listing 7.4).

Für "gradingSystem2csv" konnten einige, für "class2relational" konnten alle Transformationen aufgrund fehlgeschlagenen Zugriffs auf bestimmte Objekte nicht fehlerfrei

```
1 Diagnostic ERROR source=org.eclipse.emf.ecore code=1 The required feature↩
      'parameter' of 'org.eclipse.emf.ecore.impl.DynamicEObjectImpl/http://↩
     www.big.tuwien.ac.at/me/ws11/soopl#ParameterBinding@700498b0{file:/../↩
     Tool/OUTPUT/atlOutput/mowersystem-sooml_AtlOut.xmi#//@classes.7/↩
     @methods.0/@transitions.1/@actions.0/@parameterBinding.0}' must be set
2 ERROR: Model is invalid!
```

Listing 7.6: Beispiel für Fehlermeldung bei nicht validem Zielmodell (Transformation "sooml2soopl")

ausgeführt werden. Die entsprechende Fehlermeldung ist in Listing 7.5 angeführt. Hier soll auf das Element "name" zugegriffen werden, das allerdings nicht definiert ist. Da der Zugriff auf Elemente im Metamodell mit Hilfe von OCL erfolgt, wird hier eine OCL-spezifische Fehlermeldung ausgegeben. Die Ursache für diesen Fehler ist im Quellmetamodell zu finden. Das Attribut "name" hat den Minimalwert 0 und muss somit nicht vorkommen. In der Transformation wird aber immer versucht auf dieses Attribut zuzugreifen. Daher müsste der Minimalwert im Quellmetamodell auf 1 gesetzt werden um diesen Fehler zu verhindern.

```
1 org.eclipse.m2m.atl.engine.emfvm.VMException: Unable to access name on ↩
     OclUndefined
2   at __applyGeneralization_2_FK#50(class2relational.atl[192:11-192:97])
3     local variables: self=class2relational : ASMModule, link=↩
          TransientLink {rule = Generalization_2_FK, ...
4   at __exec__#48(class2relational.atl)
5     local variables: self=class2relational : ASMModule, e=TransientLink {↩
          rule = Generalization_2_FK, ...
6   at main#33(class2relational.atl)
7     local variables: self=class2relational : ASMModule
```

Listing 7.5: Beispiel für eine Fehlermeldung bei versuchtem Zugriff auf ein nicht definiertes Element

Unabhängig vom Beispiel ließ sich der Umstand, dass Modelle nicht valide sind, darauf zurückführen, dass benötigte Elemente im erzeugten Zielmodell fehlten. Der in Listing 7.6 dargestellte Code-Ausschnitt zeigt beispielhaft eine solche Fehlermeldung aus der Durchführung einer Studentenabgabe für das Beispiel "sooml2soopl".

Die in diesem Abschnitt angeführten Beispiele zeigen, dass es mit dem WBT-Tool möglich ist, verschiedenste Fehler zu finden. Eine weitergehende Analyse der Fehlerquellen wird auch dadurch erleichtert, dass in allen Fällen eine detaillierte Fehlermeldung ausgegeben wird. Dies erfolgt sowohl für solche, die aus der Kompilierung, beziehungsweise Ausführung von ATL als auch für solche, die aus dem Java-Code, in dem die Anwendung programmiert wurde, resultieren. All das trifft auch auf die Validierung der Zielmodelle zu.

7.5 Erkenntnisse[14]

In diesem Abschnitt werden noch einmal die wichtigsten Ergebnisse der Evaluierung zusammengefasst. Konkret sollen die Antworten auf die in Abschnitt 7.1 gestellten Forschungsfragen gegeben werden:

1. *Skalierbarkeit*:
   Wie viele Modelle erzeugt werden können, ist stark von der Rechenleistung (vor allem vom RAM-Speicher) abhängig. Die Evaluierung hat gezeigt, dass bei einem Arbeitsspeicher von 16 GB rund 40.000 Modelle erzeugt werden können. Liegen komplexere Metamodelle vor, liegt die Obergrenze bei circa 20.000 Testinstanzen.

2. *Fehlererkennung*:
   Viele verschiedene Fehler aus den einzelnen Bereichen (Modellgenerierung, Constraint-Prüfung, Modelltransformation) werden gefunden. Da vom Programm grundsätzlich alle Fehler gefangen und ausgegeben werden, kann davon ausgegangen werden, dass auch andere, in dieser Arbeit nicht aufgetretene Fehler gefunden werden können. Durch die automatisierte Erzeugung der Testinstanzen nach objektiven Kriterien ist es mit dem WBT-Tool auch möglich Fehler zu finden, die mit manuell erstellten Testinstanzen nicht oder erst sehr spät gefunden werden können.

3. *Fehlerhäufigkeit*:
   Die häufigsten Fehler treten während der Modelltransformation auf, wobei die Verarbeitung der Transformationsdatei die Hauptursache dafür darstellt. Wenn die ATL-Datei kompiliert werden kann, kann sie in den meisten Fällen auch ausgeführt werden. Fehler, die während der Ausführung der Transformation auftreten, haben die unterschiedlichsten Ursachen. Das Programm ist also auch in der Lage eine Vielzahl der möglichen Fehler[15] in ATL zu finden.

4. *Speicherbedarf*:
   Wird von der Erstellung der Testinstanzen abgesehen (wenn mehr als die voreingestellten 100 Testinstanzen erzeugt werden sollen), weist das Programm lediglich eine geringe Speicherauslastung auf. Diese liegt im Durchschnitt bei rund 300 MB, was angesichts der zur Verfügung stehenden 4 GB bzw. 16 GB der Testumgebungen sehr gering ist.

5. *Komplexität der Laufzeit*:
   100 Testinstanzen können innerhalb von 440 Sekunden transformiert werden. Auch hier gilt im Allgemeinen wieder: Je höher die Rechenleistung der CPU ist, desto schneller können die Testläufe ausgeführt werden. Die durchschnittliche Laufzeit beträgt in der besseren Testumgebung circa 160 Sekunden. Wenn allerdings die

[14] Verfasser: Sabine Wolny, BSc

[15] `https://github.com/eclipse/atl/blob/e45255992cc8b8aa7397d27298739e1ef731c4f8/plugins/org.eclipse.m2m.atl.engine.emfvm/src/org/eclipse/m2m/atl/engine/emfvm/messages.properties`, Zuletzt besucht: 22-08-2013

Transformation nicht korrekt ausgeführt wird und keine Zielmodelle erzeugt werden, kann festgestellt werden, dass die Unterschiede in der Laufzeit nicht sehr groß sind.

KAPITEL 8

Verwandte Arbeiten[1]

In diesem Kapitel werden existierende Ansätze im Bereich der Modellgenerierung und Testen von Software näher erläutert. Auch wird ein Vergleich zu dem im Zuge dieser Arbeit entwickelten Ansatz gegeben.

Im Abschnitt 8.1 werden verschiedene Ansätze vorgestellt, die sich mit der Erzeugung von Modellen beschäftigen. Einerseits wird auf Ansätze im Bereich ASP näher eingegangen, andererseits werden auch solche in anderen Programmiersprachen betrachtet.

Im Abschnitt 8.2 werden konkrete Ansätze, die sich mit dem Testen von Modelltransformationen beschäftigen, näher vorgestellt. Dabei wird in Black- und White-Box Ansätzen unterschieden, einer der wichtigsten Einteilungen von Testkonzepten in diesem Zusammenhang.

Abschließend wird das in dieser Arbeit entwickelte WBT-Tool den anderen Ansätzen gegenübergestellt (siehe Abschnitt 8.3).

8.1 Modellgenerierung[2]

8.1.1 ASP-basierter Ansätze

Modellgenerierung ist nicht nur für das Testen von Transformationen relevant, sondern kann auch zum Beispiel zum Überprüfen von den Modellen selbst verwendet werden. Ein bereits existierender Ansatz ist die Modellvalidierung von UML Klassendiagrammen mit Hilfe von ASP [77]. Im Prinzip kann eine Modellvalidierung nur empirisch durchgeführt werden, indem das formale Modell mit den Erwartungen des Benutzers verglichen wird. Bei dem *"Milano Snapshot Generator (MSG)"* [77, S. 1] handelt es sich um ein Tool, welches diesen Prozess unterstützt. Das Tool benötigt zwei Inputs, nämlich das Modell M, welches ein UML Klassendiagramm mit möglichen OCL Constraints ist, und ein Set G

[1] Verfasser: Thomas Franz, BSc & Sabine Wolny, BSc
[2] Verfasser: Sabine Wolny, BSc

mit *generation requests* um die Anzahl der generierten Snapshots endlich zu halten [77]. Bei den generierten Snapshots handelt es sich um UML Objektdiagramme. Das Tool generiert alle möglichen Snapshots, die die Anfragen G erfüllen, wobei keine isomorphen Modelle ausgegeben werden. Mit Hilfe dieses Tools ist es möglich festzustellen, ob gegebene Modelle konsistent sind und ob die Spezifikation den Erwartungen entspricht oder angepasst werden muss.
Dieser Ansatz bezüglich Modellgenerierung ist ähnlich zu dem in dieser Arbeit entwickelten WBT Tool. Es gibt jedoch einige entscheidende Unterschiede, wie in Tabelle 8.1 zu sehen ist.

Tabelle 8.1: Unterschiede in der Modellgenerierung von MSG und dem WBT-Tool

Tools / Features	MSG	WBT
Unterstützte Modelle	XMI Repräsentation von UML Klassendiagrammen	Ecore-Metamodelle
Einschränkungen für das Modell	bestimmte Anfragen (generation requests)	maximale Anzahl an Testinstanzen
Umsetzung des Modells in ASP verständliche Sprache	indirekt: zuerst wird das Modell in die *"intermediate language DLVExi"* übergeführt und dann zusammen mit den Anfragen in DLV-Complex Code übergeführt	direkt: die Elemente werden mit Hilfe von Xpand in DLV verständlichen Code übergeführt
Verwendeter ASP Solver	DLV-Complex (kann zusätzlich noch Funktionen verarbeiten)	DLV
Ziel des Tools	Modellvalidierung	Generierung von Testinstanzen für Transformationen

In einem anderen Ansatz werden die Konzepte von MDSD genutzt um den Entwicklungsprozess von ASP Programmen zu unterstützen. Oetsch et al. [76] stellen hierfür das Tool *VIDEAS* als Prototyp vor, wobei die Abkürzung für "VIsual DEsign support for Answer-Set programming" [76][S. 383] steht. Mit Hilfe dieses Programms ist es möglich in einem Modell die Datengrundstruktur für ein ASP Programm festzulegen und es soll unter anderem verhindern, dass Fakten falsch geschrieben werden. Es kann während der Entwicklung zum Beispiel passieren, dass einmal dasselbe Faktum *classe* und *class* geschrieben wird. Ein ASP Solver erkennt dies nicht als Fehler (sondern als zwei verschiedene Fakten). Daraus können weitere Fehler und Inkonsistenzen entstehen.

Im Programm VIDEAS werden zuerst graphisch die Beziehungen zwischen Elementen in einem *Entity-Relationship (ER)* Diagramm definiert [76]. Die modellierten Bedingungen werden mit Hilfe eines Codegenerators automatisch in Constraints des ASP Codes übernommen. Außerdem kann der Benutzer mit dem *Fact Builder* Daten für Fakten eingeben, wobei der Fact Builder sicherstellt, dass die Fakten zu dem ER-Modell

konsistent sind [76]. Der Programmierer kann zusätzlich Regeln für das ASP Programm definieren. Danach wird der ASP Solver gestartet und die Answer Sets generiert. Das Tool VIDEAS ermöglicht es also durch ein ER-Modell die Fakten und Constraints von einem ASP Programm konsistent zu halten [76].

8.1.2 Weitere Ansätze

Es gibt mehrere verschiedene Ansätze und Programme um Modellinstanzen basierend auf einem Metamodell zu generieren. Der Unterschied zwischen den einzelnen entwickelten Tools ist das Ziel, welches mit der Modellgenerierung verfolgt wird. Einige werden nun etwas näher vorgestellt.

8.1.2.1 EMFtoCSP

Das Tool *EMFtoCSP*[3] [39] ist ein Programm zur teil-automatisierten Verifikation von EMF beziehungsweise UML Modellen. Es ist die Fortführung beziehungsweise Erweiterung des Tools *UMLtoCSP*[4] [16]. Das Programm nutzt den Ansatz der Constraintprogrammierung. Neben dem Modell können OCL-Constraints sowie weitere Eigenschaften, die erfüllt sein müssen, angegeben werden. Aus diesen Eingaben wird ein sogenanntes CSP erzeugt. Dieses wird anschließend in einen Solver geladen, der in der Lage ist dieses Problem zu lösen. Er liefert auch die Aussage darüber, ob ein gültiges oder ein ungültiges Ergebnis vorliegt. Ist das Ergebnis gültig, so wird eine grafische Darstellung einer gültigen Instanz zurückgeliefert. EMFtoCSP nutzt zwei unabhängige Programme. Einerseits wird *ECLiPSe*, das zur Lösung des CSP herangezogen wird, verwendet und andererseits *GraphViz*, das die grafische Ausgabe der Instanz erzeugt (im konkreten Fall als .png). Im Gegensatz zu dem in dieser Arbeit entwickelten Ansatz kann dieses Programm direkt bei der Modellerzeugung OCL-Constraints verarbeiten. Allerdings wird nur eine Instanz dem Benutzer als Grafik zur Verfügung gestellt.

8.1.2.2 USE Tool

Die *UML-based Specification Environment* oder auch kurz *USE*[5] [37] ist ein Programm um Spezifikation von Informationssystemen festzulegen. Es basiert auf einer Teilmenge der UML, wobei das Programm die Spezifikation des Modells in einer textuellen Form benötigt[6]. Somit müssen Modelle erst in für USE verständliche Ausdrücke übergeführt werden. Danach können mit Hilfe von OCL zusätzliche Bedingungen für das Modell festgelegt werden (Invarianten, Pre-, Postconditions). Wenn der Benutzer zu dem Modell ein zugehöriges Objektdiagramm erstellt, wird gleichzeitig geprüft, ob die Bedingungen erfüllt sind. Falls dies nicht der Fall ist, wird genau gezeigt, wo der Fehler verursacht

[3] `https://code.google.com/a/eclipselabs.org/p/emftocsp/`, Zuletzt besucht: 08-03-2013
[4] `http://gres.uoc.edu/UMLtoCSP/`, Zuletzt besucht: 05-03-2013
[5] `http://sourceforge.net/apps/mediawiki/useocl/`, Zuletzt besucht: 10-07-2013
[6] `http://www.db.informatik.uni-bremen.de/projects/use/use-documentation.pdf`, Zuletzt besucht: 10-07-2013

wird. Ein Benutzer kann so verschiedene Zeitpunkte eines Systems, welches durch das Modell beschrieben wird, simulieren und überprüfen, ob unter anderem die in diesem Moment gesetzten Attribute und Links passen und sich somit das System in einem validen Zustand befindet oder nicht. Gogolla et al. [36] beschreiben einen Ansatz, wie das USE Tool erweitert werden kann um gültige "Snapshots" aus einem Modell mit OCL Constraints automatisch zu generieren. Es wird nicht mehr ein Objektdiagramm vom Benutzer explizit erstellt, sondern es werden nur gewünschte Eigenschaften angegeben, die die snapshots erfüllen sollen. Hierfür wird die Sprache *A Snapshot Sequence Language (ASSL)* verwendet, welche aus einem gegebenen Intialsnapshot und gegebenen Parameterwerten eine Sequenz von Snapshots generiert. Diese werden gegenüber den OCL Invarianten geprüft und schließlich wird ein gültiger Snapshot zurückgeliefert, falls er existieren sollte [36].

8.1.2.3 Alloy Tool

Einerseits ist *Alloy*[7] [50] eine Sprache zur Beschreibung von Strukturen und andererseits ist es ein Tool um diese näher zu untersuchen. Das Tool nennt sich *Alloy Analyzer* oder auch *Alcoa (Alloy Constraint Analyzer)* und analysiert Objektdiagramme [51]. Die Sprache Alloy basiert auf der Z Notation[8], ist aber mehr an Objektdiagrammen wie in UML orientiert. Zwei Hauptmerkmale der Sprache Alloy sind nach Jackson et al. [51] folgende:

1. Alloy ist relational.
   Die Sprache baut auf Mengen und Relationen auf. Dadurch ist es oft sehr leicht möglich die Struktur eines Systems zu beschreiben. Alloy ist eine textuelle Modellierungssprache und basiert auf relationaler Logik erster Ordnung.

2. Alloy ist deklarativ.

   > *"the model is built by layering properties using conjunction, in contrast to operational languages in which the model is given by an abstract program. This allows partial models to be built, in which constraints describe how state components are related to one another, without explicit rules for how each component is updated." [51, S. 730]*

Ein Alloy Modell besteht unter anderem aus einer Signatur, Fakten und Prädikaten. Der *Alloy Analyzer* übersetzt dann die im Modell definierten Constraints in eine boolesche Formel, auf die ein *SAT-Solver (satisfiability)*[9] angewandt wird. Der Solver versucht nun ein Modell zu finden, welches die Constraints erfüllt. Sollte er eine Lösung finden, wird

[7] `http://alloy.mit.edu/alloy/`, Zuletzt besucht: 26-07-2013

[8] `https://en.wikipedia.org/wiki/Z_notation`, Zuletzt besucht: 26-07-2013

[9] `https://de.wikipedia.org/wiki/Erfüllbarkeitsproblem_der_Aussagenlogik`, Zuletzt besucht: 26-07-2013
Ein SAT-Solver ist ein algorithmisches Programm, welches versucht möglichst effizient eine Interpretation, die eine gegebene aussagenlogische Formel erfüllt, zu finden. Es gibt mehrere verschiedene SAT-Solver.

diese wieder in einen relationalen Ausdruck umgewandelt und dem Benutzer zurückgeliefert. So kann der Alloy Analyzer Alloy Modelle automatisch simulieren und analysieren und zum Beispiel die Konsistenz von Multiplizitäten in einem Objektdiagramm überprüfen [51]. Mit der Version 4 des Programmes wurde Kodkod[10] als Modellsucher in Alloy integriert und ermöglichte so eine Optimierung.
Im Unterschied zu dem *USE* Tool können nicht alle Elemente von UML und OCL in Alloy leicht umgesetzt werden. Den Grund hierfür sehen Anastasakis et al. [1] in den grundlegenden verschiedenen Designs. Zum Beispiel trennt Alloy nicht klar zwischen Mengen und Relationen, UML dagegen schon [1]. In [1] wurde ein Ansatz entwickelt, Teile des UML Klassendiagramms und von OCL in ein Alloy Modell überzuführen um mit Hilfe des Alloy Analyzers Simulationen und Verifikationen durchführen zu können. Hierfür wurde zuerst aus der gegeben Grammatik von Alloy ein Metamodell, welches MOF konform ist, entwickelt. Danach wurden Transformationsregeln für ein Subset der Metamodelle von UML Klassendiagrammen und OCL festgelegt, welche deren Elemente in Alloy Elemente überführen [1]. Daraus entstand das Tool *UML2Alloy*[11], das ein UML Klassendiagramm, welches zu diesem "Subsetmetamodell" konform ist, automatisch in ein Alloy Modell überführt [1].

8.1.2.4 Modellgenerierung mit Hilfe von Graphgrammatiken

Eine weitere Möglichkeit Modellinstanzen zu einem Metamodell zu generieren wird in [26] vorgestellt. In der Metamodellierung gibt es keine Möglichkeit direkt Instanzen der Sprache zu erzeugen, wie zum Beispiel bei einer String-Grammatik, wo leicht ein Wort aus der Grammatik abgeleitet werden kann. Allerdings ist es bei der Entwicklung von Modelltransformationen notwendig, diese ausreichend zu testen um etwaigen Fehlern vorzubeugen [26]. Ehrig et al. [26] beschreiben einen Ansatz, in dem die Metamodellierung und die Graphgrammatik[12] in einen Kontext gesetzt werden, um mit Hilfe der Grammatik valide Instanzen zum Metamodell zu erzeugen. Mit Hilfe eines Graphen wird der Zustand des Systems beziehungsweise des Programms beschrieben und durch eine Graphtransformation, die auf einer Graphgrammatik basiert, wird das System in einen neuen Zustand übergeführt. In [26] wird zuerst das Metamodell in einen Graphen umgewandelt und dann werden bestimmte Regeln in der Graphgrammatik festgelegt um valide Instanzen zu bekommen. Die definierten Regeln werden in drei Layern beschrieben:

1. Layer 1: Regel zur Erzeugung der Instanzen
   Für jede Klasse im ursprünglichen Graphen, die nicht abstrakt ist, wird eine Instanz erzeugt, und zwar solange, bis der Benutzer abbricht oder ein zeitliches Limit erreicht wird [26].

2. Layer 2: In dieser Schicht werden alle Regeln definiert, um Assoziationen mit einer 1 auf einer Seite abzubilden. Es werden daher drei Regeln definiert für die

[10] `http://alloy.mit.edu/kodkod/`, Zuletzt besucht: 27-07-2013
[11] `http://www.cs.bham.ac.uk/~bxb/UML2Alloy/index.php`, Zuletzt besucht: 27-07-2013
[12] `https://de.wikipedia.org/wiki/Graphersetzungssystem`, Zuletzt besucht: 18-07-2013

Beziehungen 1:N, 1:1 und 1:(0,1) [26]. Diese Regeln definieren daher bestimmte Einschränkungen, die für die Instanzen gelten müssen, da die Multiplizitäten eingehalten werden müssen.

3. Layer 3: Auch hier werden Regeln für Links festgelegt, allerdings sind diese optional. Daher können die Regeln beliebig oft angewandt werden [26]. Die Beziehungen, die hier abgebildet werden, sind (0,1):(0,1), (0,1):N und N:M [26].

Die vorgestellten Regeln gehen von einem vereinfachten Metamodell aus, können aber zum Beispiel für Attributwerte erweitert werden [26]. Ehrig et al. [26] weisen allerdings darauf hin, dass der vorgestellte Ansatz den Nachteil hat, dass OCL-Constraints erst nach der Erzeugung der Instanzen überprüft werden können und so einige nicht benötigte Instanzen erzeugt werden.

8.2 Testen von Modelltransformationen[13]

8.2.1 Black Box Testen

Ein Ansatz zur Unterstützung des Testens von Modelltransformationen, der in Baudry et al. [4] präsentiert wird, propagiert den Einsatz von zusätzlichen Mustern, sogenannten *Pattern*, als Orakel (siehe dazu Abschnitt 3.6). Diese werden als zusätzliche Zwischeninstanz zwischen den Quell- und Zielmodellen und deren Metamodellen eingefügt. Bei den Mustern handelt es sich um Teilmengen von Instanzen des Quell- beziehungsweise Zielmetamodells. Modelle müssen daher nicht nur Instanzen der Metamodelle, sondern auch der Muster sein [4].

Der Ansatz unterstützt den Testprozess durch Validierung der Modelle. Das Quell-Muster kann auch die Entwicklung akkurater Quellmodelle unterstützen. Dies wird durch Spezialisierung des Metamodells und die Einschränkung von möglichen Testinstanzen erreicht. Ähnlich kann das Ziel-Muster für das Design einer effizienten Orakel-Funktion für die Transformation herangezogen werden [4].

Programme zur Modelltransformation teilen viele gemeinsame Eigenschaften und erlauben daher das Design spezifischer Sprachen und Testtechniken für deren Definition und Validierung. Der in Fleury et al. [29] präsentierte Ansatz nutzt die Tatsache, dass Quellmodelle durch deren Metamodelle beschrieben werden um Testkriterien zu definieren. Black-Box Testen hat den Vorteil unabhängig von der Sprache zu sein, in der ein Programm implementiert wird. Trotzdem ist es stark davon abhängig, wie das zu testende Programm spezifiziert ist [29].

Die Idee von diesem Ansatz ist folgende: Die Spezifikation der Modelltransformation (Metamodell und Constraints) soll zur Beschreibung eines Testeignungskriteriums herangezogen werden [29]. Dazu soll die Technik der *Partition Analysis* eingesetzt werden. Der Grundgedanke, auf dem diese basiert, ist es, dass die Wertebereiche der Eingangsdaten

[13] Verfasser: Thomas Franz, BSc

für das Programm (die Funktion), das getestet werden soll, in verschiedene Teilbereiche aufgeteilt wird. Diese dürfen sich nicht überschneiden. Anschließend wird aus jedem dieser Bereiche ein Datenwert ausgewählt und der Test mit diesem Wert durchgeführt. Wird dieser erfolgreich ausgeführt, wird davon ausgegangen, dass derselbe Test auch mit anderen Werten aus demselben Teilbereich erfolgreich ausgeführt werden kann. Eine Herausforderung bei dieser Vorgehensweise ist die Definition der Teilbereiche [29, 78].

Umgelegt auf das Problem der Partitionierung von Metamodellen werden bereits existierende Abdeckungskriterien verwendet, die in Andrews et al. [2] für UML-Modelle definiert wurden. Dabei werden vor allem jene Kriterien, die die Abdeckung von Klassendiagrammen sicherstellen sollen, eingesetzt [29].

Ein Kriterium ist unter anderem die Auswahl repräsentativer Werte für Multiplizitäten von Referenzen. Zum Beispiel werden für die Multiplizität [0..N] die Werte 0, 1 und N gewählt. Bei bidirektionalen Referenzen werden repräsentative Paare mittels kartesischem Produkt der möglichen Multiplizitäten beider Enden ausgewählt. Ein weiteres Kriterium verlangt, dass für Attribute von Klassen alle repräsentativen Werte abgedeckt werden müssen [29].

Um aussagekräftige Werte für die beiden Kriterien zu finden, gibt es zwei Möglichkeiten der Umsetzung: *Standard-* und *Wissensbasierte-Partitionierung*. Bei der *Standard-Partitionierung* handelt es sich um einen reinen Black-Box-Ansatz. Dabei werden die repräsentativen Werte bereits im Vorhinein anhand des Datentyps festgelegt. Für String stellen das Literal “null”[14], der Leerstring (“”) und ein willkürlich gewählter Beispieltext solche vordefinierten Testwerte dar. Für den Datentyp Boolean werden “True” und “False” gewählt [29].

Die zweite Möglichkeit stellt einen White-Box-Ansatz dar, bei dem die repräsentativen Werte aus der Transformation extrahiert werden. Dieser Vorgang wird entweder vom Tester manuell oder automatisiert mittels statischer Typprüfung ausgeführt [29].

Beide Ansätze können auch kombiniert werden. Die *Wissensbasierte-Partitionierung* wird dann eingesetzt, wenn Informationen über ein Attribut oder eine Referenz in der Spezifikation der Modelltransformation zur Verfügung stehen, *Standard-Partitionierung* wird in allen anderen Fällen eingesetzt. Eine weitere Quelle für Testdaten stellen die Vor- und Nachbedingungen der Transformation dar [29].

Mit dieser Vorgehensweise werden Sets von Testdaten für jedes Element des Metamodells gewonnen. Die Idee des Testkriteriums ist nun sicherzustellen, das alle ausgewählten Werte durch Tests abgedeckt werden. Dazu werden Kombinationen dieser Werte für einzelne Testmodelle herangezogen. Dass eine spezifische Kombination von Testdaten auch wirklich durch ein Set an Quellmodellen repräsentiert wird, wird durch eine zusätzliche Bedingung überprüft. Bei der beliebigen Auswahl von Werten kann es auch vorkommen, dass ungültige Kombinationen von Attributwerten in einer Klasse enthalten sind. Diese können einfach eliminiert werden [29].

Modelltransformationen betreffen nicht immer das gesamte Metamodell. Daher wird als weitere Einschränkung vorgeschlagen, nur den Teil des Metamodells zu betrachten,

[14] `http://openbook.galileodesign.de/javainsel5/javainsel03_006.htm#Rxx747java03006040001001F02E101`, Zuletzt besucht: 18-07-2013

dessen Elemente für die Ausführung der Transformation relevant sind. Diese können mit Hilfe der Vor- und Nachbedingungen der Spezifikation der Modelltransformation identifiziert werden [29].

8.2.2 White-Box Testen

Eine in Mottu et al. [66] vorgestellte Möglichkeit zur Unterstützung der automatischen Erzeugung von Testmodellen setzt auf *Statische Analyse*. Dabei soll Wissen darüber, wie die Transformation auf Elemente der Quellmodelle zugreift, gewonnen und ausgenutzt werden [66].

Zunächst wird das partielle Wissen extrahiert, wie die Transformation das Quellmetamodell verwendet. Die statische Analyse der Modelltransformation ermöglicht hier die Identifikation der Klassen und Eigenschaften, die von den Operationen verwendet werden. Diese Information wird in weiterer Folge genutzt um Modellfragmente zu erzeugen [66]. Die Modellfragmente werden, wie auch das Quellmetamodell und zusätzliche Bedingungen, in Prädikate von *Alloy* (siehe Unterabschnitt 8.1.2.3) übersetzt. Diese Prädikate stellen Bedingungen dar, die von den Testmodellen, welche der SAT-Solver von Alloy erstellt, erfüllt werden müssen [66].

Im Kontext von Graphersetzungssystemen[15] werden Modelle als Graphen repräsentiert, wobei jeder Knoten und jede Kante einen bestimmten Typ hat. Diese Typen werden durch ein Metamodell (*Typ-Graph*) definiert. Die wichtigsten Bestandteile einer Graphtransformation sind Regeln, die aus Mustern (*Pattern*) und Operationen zur Modifikation bestehen. Hier wird definiert, welche (Teil-)Graphen aus dem Quellmodell wie verändert und in das Zielmodell eingefügt werden sollen [100].

Bei Wieber und Schürr [100] wird ein strukturbasierter Ansatz für Implementierungen von Graphersetzungsprogrammen vorgestellt. Dieser ist in der Lage Musterabgleiche und Kontrollflussaspekte zu berücksichtigen und zu testen. Zu diesem Zweck werden verschiedenste Strategien der *Mutation Analysis*[16] eingesetzt, um die Muster zu verändern. Bei der *Mutation Analysis* wird das zu testende System laufend modifiziert, wobei jeder auf diese Weise erzeugte Mutant einen bestimmten Fehler enthält [100].

Mit Hilfe dieser mutierten Muster, die aus der Implementierung der Transformation abgeleitet werden, wird die Qualität einer Testsuite bestimmt. Eine Testsuite entfernt einen Mutanten automatisch, wenn sie den bestimmten Fehler, den der Mutant enthält, auch tatsächlich aufdeckt. Je mehr davon gefunden werden, desto besser ist die Eignung einer bestimmten Testsuite. Ziel ist es, unkorrektes Verhalten zu provozieren, das Hinweise auf Fehler im Programm liefert [100].

Die Definition der mutierten Muster wird automatisch über ein Tool bewerkstelligt. Dabei werden mit Hilfe der Original-Transformation bestimmte Abdeckungskriterien ermittelt. Für die Erstellung der Testfälle sind allerdings Benutzerinteraktionen erforderlich [100].

[15] `https://de.wikipedia.org/wiki/Graphersetzungssystem`, Zuletzt besucht: 18-07-2013
[16] `https://en.wikipedia.org/wiki/Mutation_testing`, Zuletzt besucht: 18-07-2013

Der Begriff *TGG*[17] bezeichnet einen wichtigen Ansatz aus dem Bereich der relationalen Modelltransformationen (siehe Unterabschnitt 2.4.2.4). Sie sind ein relationaler Ansatz zur bidirektionalen Modelltransformation und Synchronisation von Modellen. TGG kombiniert drei konventionelle Graphgrammatiken[15] für Quell-, Ziel- und Korrespondenzmodelle [45].

Der in Hildebrandt et al. [45] vorgestellte Ansatz erweitert eine existierende Testumgebung für *TGG* Implementierungen. Diese Testumgebung nutzt grammatikalische Charakteristika von TGG um Testmodelle zu erzeugen. Auf ähnliche Weise werden für diese Modelle auch die entsprechenden Zielmodelle erzeugt, sodass ein komplettes Orakel generiert wird [45].

Die vorgestellte Erweiterung nutzt Abhängigkeiten, die in einer TGG bereits enthalten sind, um minimale Testfälle zu bestimmen, die alle Regeln und Abhängigkeiten abdecken. Dieser Ansatz funktioniert allerdings nur solange die TGG wohlgeformt ist. Die Abhängigkeiten zwischen den Regeln werden dabei als eigene Graphen dargestellt, aus denen die Testfälle abgeleitet werden [45].

Dadurch wird eine höhere Qualität der Testfälle erreicht und somit auch der Testprozess selbst verbessert. Insbesondere kann so eine vollständige Spezifikationsabdeckung, welche sich aus Regelabdeckung und dem Überdeckungsgrad der Abhängigkeiten der Regeln zusammen setzt, erreicht werden. Eine weitere Eigenschaft der Testfälle ist, dass sie im Bezug auf die Regeln, die benötigt werden um eine bestimmte Abhängigkeit zu testen, minimal sind. Wird irgendeine Regel (abgesehen von der letzten) aus der Testfallbeschreibung entfernt, ist der gesamte Testfall nicht mehr einsetzbar [45].

Küster und Abd-El-Razik [55] beschäftigen sich mit dem Testen von Modelltransformationen, die im Rahmen der Geschäftsprozessmodellierung[18], einem weiteren Bereich, für den Ansätze aus dem Model Engineering adaptiert werden, vorkommen. Zu diesem Zweck stellen sie einige sehr unterschiedliche Techniken zur Erzeugung von Testfällen vor.

Testen der Metamodell-Abdeckung: Eine Regel wird in eine Metamodell-Vorlage (Template) transformiert. Aus dieser werden automatisch Instanzen abgeleitet, die sich als Testinstanzen eignen. Bei der Transformation der Regeln muss unter anderem für abstrakte Elemente beachtet werden, dass diese entweder konkret angegeben oder über Parameter definiert werden. Mögliche Werte für diese Parameter werden aus dem Metamodell der Modellierungssprache abgeleitet [55].

Aus jeder Regel kann eine Reihe von Vorlagen abgeleitet werden und gemeinsam können diese einen hohen Abdeckungsgrad (siehe Unterabschnitt 3.1.1.3) für das Metamodell sicherstellen. Aus der Überdeckung für die einzelnen Regeln kann diejenige der Transformation insgesamt abgeleitet werden. Mit diesem Ansatz können Fehler bei der Ermittlung der Metamodell-Abdeckung, aber auch Fehler in der Syntax und der Semantik sowie solche, die aus fehlerhaftem Code resultieren, gefunden werden [55].

[17] `https://de.wikipedia.org/wiki/Tripel-Graph-Grammatik`, Zuletzt besucht: 18-07-2013

[18] `https://de.wikipedia.org/wiki/Geschäftsprozessmodellierung`, Zuletzt besucht: 18-07-2013

Verwendung von Bedingungen: Bei dieser Technik werden zusätzliche Bedingungen (Constraints) zur Erzeugung von Testinstanzen herangezogen. Hauptaufgabe dieser Instanzen ist es, Fehler zu finden, die aus der Verletzung der Bedingungen resultieren. Dazu werden Testfälle entwickelt, die die Korrektheit der Bedingungen nach Ausführung der Transformation sicherstellen. Dies geschieht auf folgende Weise:

- Identifikation von Elementen im Modell, die bei Ausführung der Transformation verändert werden.
- Identifikation von Constraints, die sich auf solche Elemente beziehen.
- Entwicklung eines Testfalls für jeden Constraint. Dieser überprüft, ob die entsprechende Bedingung während der Transformation erfüllt ist.

Gonzales und Cabot [38] stellen einen weiteren Ansatz vor, der sich mit der Erzeugung von Testmodellen aus einer ATL-Transformation beschäftigt. Das Ziel ist die Optimierung des Prozesses durch die Maximierung der Abdeckung der inneren Struktur (z.B. Pfadabdeckung; siehe auch Unterabschnitt 6.4.3) der Transformation.

Dies erfolgt in drei Schritten [38]:

1. Im ersten Schritt wird die ATL-Transformation analysiert und ein abstrakter Graph erzeugt, der die relevanten Informationen enthält. Dieser wird als "Abhängigkeitsgraph" bezeichnet. Der Abhängigkeitsgraph repräsentiert Gruppen von verknüpften Bedingungen, die in Form von OCL-Constraints ausgedrückt werden. Diese müssen (entweder ganz oder teilweise) vom Testmodell erfüllt werden [38].
2. In diesem Schritt wird der Graph mehrmals durchlaufen. Die Anzahl der Durchläufe wird durch ein Überdeckungskriterium bestimmt. Dabei werden Wahrheitswerte für unterschiedliche Bedingungen im Graphen gesetzt. Jeder Durchlauf liefert auf diese Weise ein Set an Bedingungen, das eine Familie von relevanten Testfällen repräsentiert [38].
3. Im letzten Schritt werden die Testfälle erzeugt. Die Modelle müssen dabei konform zum Metamodell sein und die Bedingungen, die für den Testfall definiert wurden, erfüllen. Zur Erzeugung der Testmodelle werden SAT-basierte oder CSP-basierte Solver eingesetzt. In [38] wurde *EMFtoCSP* (siehe Unterabschnitt 8.1.2.1) für diesen Zweck eingesetzt.

   Hier wird auch sichergestellt, dass alle Pfade des Programms (der Transformation) durch Testfälle abgedeckt werden. Um dies sicherzustellen werden zwei Abdeckungskriterien verwendet: Bedingungsabdeckung (*condition coverage*[19]) und Multiple-Bedingungsabdeckung (*multiple-condition coverage*). Der Umstand, dass

[19] `https://en.wikipedia.org/wiki/Modified_condition/decision_coverage`, Zuletzt besucht: 19-07-2013

jeder Knoten des Abhängigkeitsgraphen einen Boolean-Ausdruck[20] enthält und auch als Pfad in der Transformation interpretiert werden kann, ermöglicht den Einsatz dieser beiden Kriterien [38].

Eingesetzt werden die beiden Kriterien, während der Graph durchlaufen wird. Bei jedem Durchlauf wird den OCL-Constraints ein anderer Ausgabewert zugewiesen (Bedingungsabdeckung). Handelt es sich um komplexere OCL-Constraints, werden den einzelnen Komponenten, aus denen sie sich zusammensetzen, unterschiedliche Werte zugewiesen (Multiple-Bedingungsabdeckung) [38].

Eine weitere Besonderheit dieses Ansatzes ist der Umstand, dass ein Durchlauf durch den Abhängigkeitsgraphen abgebrochen wird, wenn die Bedingung eines Knoten zu "False" ausgewertet wird. Der Nachbarknoten wird nur dann besucht, wenn die Bedingung des aktuellen "True" ist [38].

8.3 Abgrenzung zum WBT-Tool[21]

Der in dieser Arbeit vorgestellte Ansatz zeichnet sich durch einen hohen Grad an Flexibilität aus. Der Benutzer hat, sofern er geringfügige Änderungen im Quellcode vornimmt, die Möglichkeit festzulegen, wie viele Testinstanzen automatisch erzeugt werden (einzige Einschränkung hierbei ist die Rechenleistung).

Unabhängig davon, zeichnet sich der Ansatz dadurch aus, dass kaum Benutzerinteraktionen erforderlich sind. Die einzelnen Komponenten (wie z.B. Metamodelle, OCL-Constraints) können so, wie sie üblicherweise vorliegen, vom Programm direkt verarbeitet werden.

Die Testinstanzen werden automatisiert aus dem Quellmetamodell erzeugt. Dabei wird das Metamodell selbst nicht verändert. Im Gegensatz zu den in Abschnitt 8.1 beschriebenen Ansätzen werden Modelle in einem entsprechenden Format (XMI) erzeugt. Das Tool EMF2CSP beispielsweise liefert Modelle in einem Grafikformat als Bild zurück. Auch können viele Konzepte der Metamodelle vom Programm verarbeitet werden. Außerdem ist es möglich mit dem ASP-basierten Ansatz sehr viele unterschiedliche Testinstanzen in kurzer Zeit zu erzeugen.

Ein weiteres Charakteristikum ist, dass das Tool modular aufgebaut ist. Dadurch erhält der Benutzer genaue Informationen über die Ausführung der einzelnen Module. So werden zum Beispiel vom ATL-Modul nur jene Fehler ausgegeben, die während der Kompilierung, beziehungsweise Ausführung der Transformation auftreten.

Ein weiterer Vorteil dieses modularen Aufbaus ist, dass die OCL-Constraints separat überprüft werden. Einige der in Abschnitt 8.2 beschriebenen Ansätze setzen diese zur Modellgenerierung ein. Allerdings ist es so nicht möglich, schlecht definierte Constraints zu identifizieren, was problematisch ist, da diese nicht unerheblichen Einfluss auf die Testinstanzen haben.

[20] `https://de.wikipedia.org/wiki/Boolesche_Algebra`, Zuletzt besucht: 19-07-2013
[21] Verfasser: Thomas Franz, BSc

Ein weiterer wichtiger Unterschied zu den in Abschnitt 8.2 vorgestellten Ansätzen ist die abschließende Validierung der Zielmodelle. Mit Hilfe der durchgängigen Dokumentation kann anhand des Quellmodells und einer detaillierten Fehlermeldung festgestellt werden, warum das Ergebnis der Transformation keine valide Instanz des Zielmetamodells ist.

Insgesamt ist der Testprozess, der mit dem entwickelten Programm realisiert wurde, weniger komplex als einige der in Abschnitt 8.2 vorgestellten, wodurch sich auch der Testaufwand verringert. Durch eine geringe Anzahl an Einstellungen, die entscheidenden Einfluss auf den Testprozess haben, können Änderungen rasch und einfach umgesetzt werden. Durch einige Erweiterungen, wie etwa einer Analyse der Log-Datei und des Quellmetamodells um nicht ausgeführte Regeln zu identifizieren, kann das Modul zur Generierung der Testinstanzen automatisch erweitert werden um eine bessere Testabdeckung mit relativ geringem Aufwand und ohne Verlust des hohen Grades an Automatisierung zu erreichen (siehe dazu Abschnitt 9.2).

KAPITEL 9

Schlussbetrachtung und Ausblick[1]

9.1 Zusammenfassung[2]

Modelle und Modelltransformationen sind im Bereich der Softwareentwicklung immer wichtigere Bestandteile. Dadurch wird es auch notwendig, die Metamodelle und die Transformationen, die auf diesen aufbauen, zu testen um eine gewisse Softwarequalität sicherzustellen.

Diese Arbeit beschäftigte sich damit, ob es möglich ist, automatisiert Metamodelle und Modelltransformationen zu testen. Außerdem sollte der Testansatz nicht, wie bisher vorherrschend in der MDSD, ein Black-Box Ansatz sein, sondern ein White-Box Ansatz. Mit der Entwicklung des WBT-Tools konnte dies für Ecore Metamodelle und ATL Transformationen verwirklicht werden. Durch den modularen Aufbau des Programms ist es möglich, dieses flexibler einzusetzen. Das erste entwickelte Modul beschäftigt sich mit der automatischen Generierung von Modellinstanzen, die konform zum angegebenen Metamodell sind. Hierfür wird der DLV Solver verwendet. Das zweite Modul überprüft Bedingungen in Form von OCL-Constraints, die sowohl für die Quellmodelle, als auch für die Zielmodelle angegeben werden können. Das letzte Teilprogramm beschäftigt sich mit der ATL Transformation und führt diese durch. Für alle drei Teilprogramme wird die Ausführungsdauer aufgezeichnet, wobei bei der Transformation noch detailliert die Zeiten für die Kompilierung, das Laden der Modelle und die eigentliche Transformationsdurchführung abgespeichert werden.

Die Evaluierung hat gezeigt, dass das Programm nicht nur Fehler im Umfeld der Transformation, wie zum Beispiel nicht valide Modelle oder Fehler bei der Prüfung von OCL-Constraints ausgibt, sondern auch Laufzeitfehler, die während der Transformation entstehen, erkennt und detailliert aufzeichnet. Das Programm benötigt, solange keine zu komplexen Testmodelle generiert werden sollen, wenig Speicher und läuft schnell

[1] Verfasser: Thomas Franz, BSc & Sabine Wolny, BSc
[2] Verfasser: Sabine Wolny, BSc

und effizient. Außerdem können aufgrund der automatischen Modellgenerierung Fehler aufgezeigt werden, die mit manuell erstellten Tests vielleicht erst später erkannt würden, da diese Modellinstanzen aus der Transformationsentwicklersicht möglicherweise nicht erstellt und daher auch nicht getestet würden.

In der modellgetriebenen Softwareentwicklung wird auch in Zukunft das Testen von Transformationen nicht an Bedeutung verlieren, da das frühe Erkennen von Fehlern Zeit und Kosten spart. Wenn Fehler nicht rechtzeitig erkannt werden, kann dies Schwierigkeiten verursachen, die nur schwer behoben werden können.

9.2 Ausblick[3]

Das, im Rahmen dieser Arbeit, entwickelte WBT-Tool ist in der Lage vollautomatisiert ATL-Modelltransformationen zu testen. Auch die automatische Erzeugung von Testmodellen wird vom Tool unterstützt. Dabei wird auch der Testprozess selbst transparent. So wird in Abschnitt 6.5 gezeigt, dass nicht nur die Aufzeichnung der definierten und ausgeführten Regeln ermöglicht wird. Darüber hinaus hat die Evaluierung (Kapitel 7) gezeigt, dass auch verschiedenste Fehler, die während oder im Umfeld der Transformation auftreten können, vom Programm entdeckt und detailliert ausgewiesen werden.

Da es sich beim Testen von Modelltransformationen um eine sehr komplexe Aufgabe handelt, gibt es für das WBT-Tool einige potenzielle Verbesserungen, die in der Folge vorgestellt werden.

Die Testmodelle werden momentan automatisch mit Hilfe vom DLV Solver erzeugt. Hierfür werden die Informationen des Metamodells mit *Xpand* in für den Solver verständliche Regeln transformiert. Allerdings wird momentan nur ein kleiner Ausschnitt von den möglichen Eigenschaften, die in einem *Ecore* Metamodell angegeben werden können, umgesetzt. Eine sinnvolle Erweiterung in diesem Bereich brächte den Vorteil, dass mehr Metamodelle, beziehungsweise mehr Eigenschaften getestet werden können. Ein Beispiel hierfür wäre, dass auch multi-valued Attribute verarbeitet werden oder ein Metamodell mehrere Packages haben kann.

Eine weitere Anpassung im Bereich der Testmodelle ist, dass die Werte für die Attribute nicht mehr random Werte sind, sondern dass hier eine Liste vorgegeben wird, aus der zufällig ein Wert genommen wird. Es kann zum Beispiel für ein String Attribut auch der Sonderfall getestet werden, dass dieser String leer ist. Somit ist es möglich zu überprüfen, wie sich ein Programm in Randsituationen verhält. Bezüglich Kriterien zur Erstellung dieser Werte gibt es schon einige Überlegungen, unter anderem unter dem Namen “Parameter Generation”[4].

Im Prinzip werden mit Hilfe von DLV alle möglichen Kombinationen der Elemente vom Metamodell erzeugt und als Modell zurückgeliefert. Allerdings umfasst dies auch symmetrische Lösungen. Diese können nur mit Hilfe eines *reasoning* über alle Answer Sets eliminiert werden, was mit DLV so nicht möglich ist. Allerdings könnte dies mit Hil-

[3] Verfasser: Thomas Franz, BSc

[4] `http://msdn.microsoft.com/en-us/library/ee620448.aspx`, Zuletzt besucht: 17-08-2013

fe der Erweiterung von DLV, nämlich *dlvhex*[5], gelöst werden. Eine weitere Möglichkeit wäre einen anderen ASP Solver zu nutzen, bei dem ausgewählte Answer Sets zurückgeliefert werden. Somit könnten zum Beispiel 100 Modelle generiert werden, jedoch werden nicht die ersten 100 Answer Sets dafür verwendet. Der Vorteil ist, wenn die Modelle von beliebiger Stelle genommen werden, dass sie sich in den Eigenschaften mehr unterscheiden. Der DLV Solver generiert die Modelle systematisch, und daher haben die ersten 100 Answer Sets auch sehr ähnliche Eigenschaften.

Beim Testen des Programms selbst hat sich herausgestellt, dass die Prüfung der OCL-Constraints eine sehr zeitintensive Aufgabe ist. Die Evaluierung hat gezeigt, dass die Ausführung der Transformation ohne die Berücksichtigung zusätzlicher Bedingungen bereits einige Zeit in Anspruch nimmt. Daher stellt die Überprüfung von Constraints eine wichtige Aufgabe hinsichtlich einer weiteren Optimierung des Tools dar.

Hier würde sich eine Integration der Bedingungen in den DLV basierten Ansatz zur Modellgenerierung anbieten, da dieser mit sehr geringem Zeitaufwand ausgeführt wird. Auch die in Unterabschnitt 6.6.2.2 angedachte Integration der Constraints in die jeweiligen Metamodelle könnte Verbesserungen bringen. Letzteres setzt voraus, dass dadurch der Aufwand zur Validierung der Modelle (der aktuell für die Zielmodelle vernachlässigbar gering ausfällt) nicht signifikant erhöht wird.

Auch bei der Verarbeitung der Transformationen besteht Verbesserungspotenzial. Das Programm ist zwar in der Lage bestimmte Variationen hinsichtlich der Formatierung, wie Einrückungen und Benennungen, zu erkennen und zu verarbeiten, allerdings schließt dies nicht aus, dass gewisse Spezialfälle zu ungewollten Fehlern führen. Auch die Aufzeichnungen der definierten und ausgeführten Regeln variieren zum Teil. So ist nicht immer gewährleistet, dass ein Regelname auch mit der Aufzeichnung (Datentyp der Klasse des Modells) während der Laufzeit übereinstimmt.

Eine Erweiterung des Programms, um diese Unterschiede auszugleichen, würde auch die Ermittlung von Abdeckungskriterien (z.B. Regelabdeckung) erleichtern. Allerdings besteht hier die besondere Herausforderung darin, dass eine solche Angleichung nicht die Vorteile, die die Aufzeichnung der Typen der Quell- und Zielklassen bietet (siehe Unterabschnitt 6.4.3), aufhebt.

In der ATL-Komponente (siehe Abschnitt 6.4) des WBT-Tools wird ein Helper verwendet um die ausgeführten Regeln aufzuzeichnen. Dieser wird in die Transformation integriert. Da aber auch viele Aufgaben der Transformation in solche Helper ausgelagert werden, würde eine Aufzeichnung der aufgerufenen Helper die Informationen, die aus der Transformationsausführung gewonnen werden, sinnvoll ergänzen.

Das einheitliche Aussehen der Log-Dateien des Programms soll auch die Berechnung diverser Kennzahlen unterstützen (z.B. Regelabdeckung). Eine weitere Möglichkeit wäre zu ermitteln, welche Konzepte des Quellmetamodells wie oft in den erzeugten Testinstanzen vorkommen, beziehungsweise in welchem Ausmaß diese durch die Testmodelle abgedeckt werden. Hierfür würde sich auch eine aufbereitete grafische Übersicht in Form einer eigenen GUI anbieten.

[5] `http://www.kr.tuwien.ac.at/research/systems/dlvhex/`, Zuletzt besucht: 17-08-2013

Das Wissen darüber, welche Konzepte des Quellmetamodells von den mit DLV generierten Testmodellen verwendet werden, kann auch genützt werden um zu ermitteln, welche Elemente *nicht* abgedeckt wurden. Mit diesen Informationen ist es möglich, Teile der DLV-Komponente dynamisch und automatisiert so zu erweitern, dass neue Testmodelle erzeugt werden, die genau diese, nicht in den ursprünglichen Testmodellen enthaltenen Konzepte, abdecken.

Eine weitere Herausforderung, die sich in dieser frühen Entwicklungsphase des WBT-Tools stellt, ist die Auswertung der aufgezeichneten Daten, wie definierte/ausgeführte Regeln, Zeitaufwand und Auslastung des RAM-Speichers. Je umfangreicher der Testlauf ist, desto umfangreicher wird die Datenmenge. Der große Vorteil der aktuell umgesetzten Datengewinnung ist der hohe Grad an Automatisierung: Es ist keine Benutzerinteraktion erforderlich.

Eine mögliche Vereinfachung dieser Aufgabe, die auch zusätzliche Informationen, sowie eine Visualisierung der Daten bieten würde, wäre der Einsatz der "Java Visual VM"[6,7]. Deren Verwendung erfordert allerdings manuelle Interaktionen durch den Benutzer um die Informationen zu speichern. Bietet die Visual VM aber eine geeignete Schnittstelle, über die man diesen Vorgang automatisieren und in das WBT-Tool integrieren könnte, wäre sie eine sinnvolle Ergänzung (bzgl. RAM-Auslastung sogar Alternative) zu den bestehenden Lösungen.

[6] `https://en.wikipedia.org/wiki/VisualVM`, Zuletzt besucht: 03-08-2013
[7] `http://visualvm.java.net/`, Zuletzt besucht: 03-08-2013

ANHANG A

Abkürzungsverzeichnis

API Application Programming Interface

ASP Answer Set Programming

ASSL A Snapshot Sequence Language

ATL Atlas Transformation Language

CIM Computational Independent Model

CMOF Complete Meta-Object Facility

CSP Constraints Satisfaction Problem

DLV Datalog with disjunction

DSL Domain Specific Language

edlp extended disjunctive logic program

EMF Eclipse Modeling Framework

EMFT Eclipse Modeling Framework Technologies

EMOF Essential Meta-Object Facility

ER Entity-Relationship

GCO Guess/Check/Optimize

GUI Graphical User Interface

IDE Integrated Development Environment

LHS Left-Hand Side

MBT Model Based Testing

MDA Model Driven Architecture

MDD Model Driven Development

MDE Model Driven Engineering

MDSD Model Driven Software Development

MDSE Model Driven Software Engineering

MDT Model Development Tools

MMT Model to Model Transformation

M2C Model to Code

MTL Model Transformation Language

M2M Model to Model

M2T Model to Text

MOF Meta-Object Facility

MWE Modeling Workflow Engine

NAC Negative Application Condition

ndlp normal disjunctive logic program

nlp normal logic program

NP non-deterministic polynomial-time

oAW openArchitectureWare

OCL Object Constraint Language

OMG Object Management Group

PAC Positive Application Condition

PIM Platform Independent Model

PSM Platform Specific Model

QVT Query View Transformation

RAM Random Access Memory

RHS Right-Hand Side

SAT satisfiability

TGG Triple Graph Grammar

T2M Text to Model

UML Unified Modeling Language

WBT White-Box Testing

XML Extensible Markup Language

XMI XML Metadata Interchange

ANHANG B

Beispiel *"Families2Persons"*

In diesem Teil des Anhangs werden Teile des Beispiels *"Families2Persons"* präsentiert. Dabei handelt es sich nicht um das Original-Beispiel (das in den ATL-Tutorials[1] beschrieben wird und auch als Download[2] zur Verfügung steht), sondern um eine erweiterte Version. Diese wurde auch in Unterabschnitt 6.5.3 ab Seite 132 zur Dokumentation der Funktionsweise des WBT-Tools verwendet.

In Abbildung B.1 ist das (Quell)Metamodell für Familien, in Abbildung B.2 ist das (Ziel)Metamodell für Personen grafisch dargestellt.

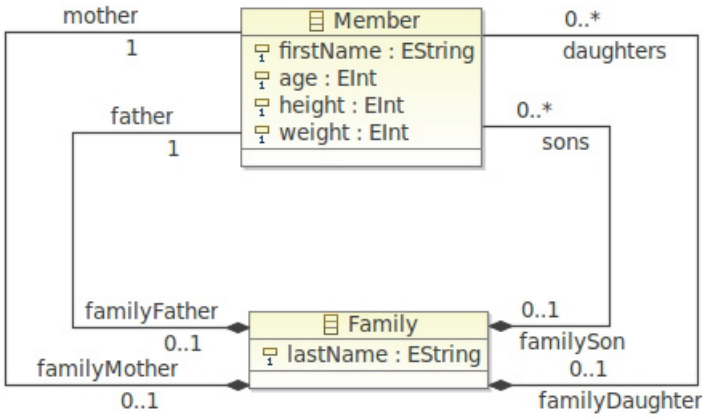

Abbildung B.1: Grafische Darstellung des Metamodells *"Families.ecore"*

[1] `http://wiki.eclipse.org/ATL/Tutorials_-_Create_a_simple_ATL_transformation`, Zuletzt besucht: 30-07-2013

[2] `http://www.eclipse.org/atl/documentation/basicExamples_Patterns/`, Zuletzt besucht: 30-07-2013

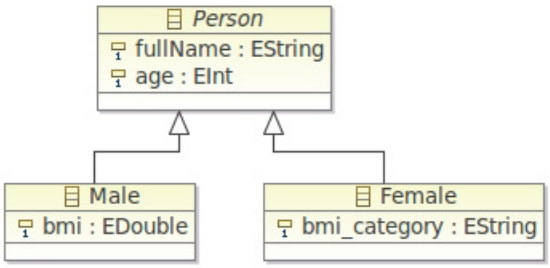

Abbildung B.2: Grafische Darstellung des Metamodells *"Persons.ecore"*

In Listing B.1 wird die ATL-Transformation für dieses Beispiel angeführt. Deren Regeln wurde im Vergleich zum Original nicht verändert. Lediglich die Helper (inkl. der neu hinzugefügten) wurden in eine eigene Bibliothek ausgelagert um den Funktionsumfang des Tools zu dokumentieren. Diese Bibliothek findet sich in Listing 6.1 auf Seite 133.

Abschließend werden in diesem Abschnitt noch zwei Modelle beschrieben. Dabei handelt es sich einerseits um ein manuell erstelltes, mögliches Quellmodell für die veränderte Transformation (siehe Listing B.2) sowie das daraus erzeugte Zielmodell (siehe Listing B.3). Alle verwendeten Werte für die einzelnen Attribute wurden hier zufällig ausgewählt und folgen auch nicht unbedingt logischen Erwägungen (z.B. die 75-jährige Tocher der Beispielfamilie).

```
109 -- @path Families=/Families2Persons/Families.ecore
110 -- @path Persons=/Families2Persons/Persons.ecore
111
112 module Families2Persons;
113 create OUT : Persons from IN : Families;
114
115 uses Lib4F2P;
116
117 helper context Families!Member def: familyName : String =
118   if not self.familyFather.oclIsUndefined() then
119     self.familyFather.lastName
120   else
121     if not self.familyMother.oclIsUndefined() then
122       self.familyMother.lastName
123     else
124       if not self.familySon.oclIsUndefined() then
125         self.familySon.lastName
126       else
127         self.familyDaughter.lastName
128       endif
129     endif
130   endif;
131
132
133 rule Member2Male {
134   from
135     s : Families!Member (not s.isFemale())
136   to
137     t : Persons!Male (
138       fullName <- s.firstName + ' ' + s.familyName,
139       age <- s.age,
140       bmi <- s.calculateBMI()
141     )
142 }
143
144 rule Member2Female {
145   from
146     s : Families!Member (s.isFemale())
147   to
148     t : Persons!Female (
149       fullName <- s.firstName + ' ' + s.familyName,
150       age <- s.age,
151       bmi_category <- s.bmiCategory()
152     )
153 }
```

Listing B.1: ATL-Transformation (ohne ausgelagerte Helper) für “Families2Persons”

```
<?xml version="1.0" encoding="ASCII"?>
<families:Family xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:families="http://families/1.0" xsi:schemaLocation="http://families/1.0 Families.ecore" lastName="Family">
  <father firstName="father" age="55" height="178" weight="93"/>
  <mother firstName="mother" age="52" height="185" weight="68"/>
  <sons firstName="son_1" age="24" height="147" weight="258"/>
  <sons firstName="son_2" age="28" height="286" weight="155"/>
  <daughters firstName="daughter_1" age="75" height="147" weight="87"/>
  <daughters firstName="daughter_2" age="25" height="246" weight="85"/>
</families:Family>
```

Listing B.2: Mögliches Quellmodell mit Beispielfamilie

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:persons="http://persons/1.0">
  <persons:Male fullName="father Family" age="55" bmi="0.5224719101123596"/>
  <persons:Male fullName="son_1 Family" age="24" bmi="1.7551020408163265"/>
  <persons:Male fullName="son_2 Family" age="28" bmi="0.541958041958042"/>
  <persons:Female fullName="mother Family" age="52" bmi_category="under"/>
  <persons:Female fullName="daughter_1 Family" age="75" bmi_category="normal"/>
  <persons:Female fullName="daughter_2 Family" age="25" bmi_category="under"/>
</xmi:XMI>
```

Listing B.3: Erzeugtes Zielmodell mit Beispielpersonen

Literaturverzeichnis

[1] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A Challenging Model Transformation. In *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*, pages 436–450. Springer Verlag Berlin, Heidelberg, 2007.

[2] A. Andrews, R. France, S. Ghosh, and G. Craig. Test adequacy criteria for UML design models. *Software Testing, Verification and Reliability*, 13(2):95–127, 2003.

[3] O. Balci. Verification, validation, and accreditation. In *Winter Simulation Conference*, pages 41–4, 1998.

[4] B. Baudry, T. Dinh-Trong, J.M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. Le Traon. Model Transformation Testing Challenges. In *Proceedings of IMDT Workshop in conjunction with ECMDA'06*, 2006.

[5] B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, and J.M. Mottu. Barriers to systematic model transformation testing. *Communications of the ACM*, 53(6):139–143, 2010.

[6] C. Beierle, O. Dusso, and G. Kern-Isberner. Using answer set programming for a decision support system. In *Proceedings of the 8th international conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, pages 374–378. Springer Verlag Berlin, Heidelberg, 2005.

[7] C. Beierle and G. Kern-Isberner. Logisches Programmieren und Anwortmengen. In *Methoden wissensbasierter Systeme : Grundlagen — Algorithmen — Anwendungen*, pages 269–301. Friedr. Vieweg & Sohn Verlagsgesellschaft / GWV Fachverlage GmbH, Wiesbaden, 3. edition, 2006.

[8] S. Biffl, D. Winkler, and D. Frast. *Skriptum: Qualitätssicherung, Qualitätsmanagement und Testen in der Softwareentwicklung.* Vienna University of Technology, 2011. `http://qse.ifs.tuwien.ac.at/courses/skriptum/script.htm`.

[9] R. Bihlmeyer, W. Faber, G. Ielpa, V. Lio, and G. Pfeifer. DLV - User Manual. `http://www.dlvsystem.com/html/DLV_User_Manual.html`, Zuletzt besucht: 15-03-2013.

[10] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison Wesley Publishing Company Inc, 1. edition, 1999.

[11] International Software Testing Qualifications Board. *Standard glossary of terms used in Software Testing*. 2.2 edition, 2012. `http://www.istqb.org/downloads/finish/20/101.html`.

[12] C. Bommer, M. Spindler, and V. Barr. *Software-Wartung: Grundlagen, Management und Wartungstechniken*. dpunkt.verlag, 1. edition, 2008.

[13] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012.

[14] E. Bringmann and A. Krämer. Model-Based Testing of Automotive Systems. In *Proceedings of the First International Conference on Software Testing, Verification and Validation (ICST 2008), April 9-11, Lillehammer, Norway*, pages 485–493. IEEE CS Press, 2008.

[15] A. D. Brucker and B. Wolff. On theorem prover-based testing. *Formal Aspects of Computing*, 25(5):683–721, 2013.

[16] J. Cabot, R. Clarisó, and D. Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 547–548, New York, USA, 2007. ACM.

[17] S. Citrigno, T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The dlv System: Model Generator and Application Frontends. In *Proceedings of the 12th Workshop on Logic Programming*, pages 128–137, 1997.

[18] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1. edition, 1999.

[19] L. Copeland. *A Practitioner's Guide to Software Test Design*. Artech House Inc, 1. edition, 2003.

[20] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J.F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In R. F. Paige, editor, *Theory and Practice of Model Transformations*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283. Springer Berlin Heidelberg, 2009.

[21] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *Proceedings of OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[22] R.A. DeMilli and A.J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[23] Z. Drey, C. Faucher, F. Fleurey, V. Mahé, and D. Vojtisek. Kermeta language Reference manual, 2011. `http://www.kermeta.org/docs/fr.irisa.triskell.kermeta.documentation/build/pdf.fop/KerMeta-Manual/index.pdf`, Zuletzt besucht: 06-08-2013.

[24] E. Dustin, J. Rashka, and J. Paul. *Software automatisch testen*. Springer Berlin Heidelberg, 1. edition, 2000.

[25] S. Efftinge, P. Friese, A. Hase, D. Hübner, C. Kadura, B. Kolb, J. Köhnlein, D. Moroff, K. Thoms, M. Völter, P. Schönbach, M. Eysholdt, S. Reinisch, and A. Jockel, D. Arnoldet. XPand Documentation, 2010. `http://ditec.um.es/ssdd/xpand_reference.pdf`, Zuletzt besucht: 05-04-2013.

[26] K. Ehrig, J. M. Küster, G. Taentzer, and J. Winkelmann. Generating Instance Models from Meta Models. In R. Gorrieri and H. Wehrheim, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 4037 of *Lecture Notes in Computer Science*, pages 156–170. Springer Berlin Heidelberg, 2006.

[27] T. Eiter, G. Ianni, and T. Krennwallner. Answer Set Programming: A Primer. In S. Tessaris, E. Franconi, T. Eiter, C. Gutierrez, S. Handschuh, M. Rousset, and R. Schmidt, editors, *Reasoning Web. Semantic Technologies for Information Systems*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer Berlin Heidelberg, 2009.

[28] C. Fermüller, R. Freund, B. Gramlich, A. Leitsch, M. Oswald, and G. Salzer. Von Termen zu Programmiersprachen: Syntax versus Semantik. In *Theoretische Informatik und Logik*, pages 81–95. Institut für Computersprachen Technische Universität Wien, 2010.

[29] F. Fleury, J. Steel, and B. Baudry. Validation in Model-Driven Engineering: Testing Model Transformations. In *Proceedings of MoDeVa'04 (Model Design and Validation Workshop associated to ISSRE'04), November 2, Rennes, France*, pages 29–40. IEEE CS Press, 2004.

[30] G. Fraser, F. Wotawa, and P.E. Ammann. Testing with model checkers: a survey. *Software: Testing, Verification and Reliability*, 19(3):215–261, 2009.

[31] Jounault Frédéric, Allilaire Freddy, Bézivin Jean, and Kurtev Ivan. ATL: A model transformation tool. *Science of Computer Programming*, 72(1–2):31–39, 2008.

[32] D. Gašević, D. Djurić, and V. Devedžić. *Model Driven Engineering and Ontology Development*. Springer Berlin Heidelberg, 2. edition, 2009.

[33] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam Answer Set Solving Collection. *AI Communications*, 24(2):105–124, 2011.

[34] M. Gelfond. Answer Sets. In F. Van Harmelen, V. Lifschitz, and B. Porter, editors, *Handbook of Knowledge Representation*, pages 285–316. Elsevier Science, 2008.

[35] M. Gelfond and V. Lifschitz. The Stable Model Semantics For Logic Programming. In *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[36] M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Journal on Software and System Modeling*, 4:386–398, 2005.

[37] M. Gogolla, F. Büttner, and M. Richters. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69(1-3):27–34, 2007.

[38] C.A. Gonzalez and J. Cabot. ATLTest: A White-Box Test Generation Approach for ATL Transformations. In *Proceedings of 15th International Conference on Model Driven Engineering Languages and Systems (MODELS 2012), September 30 – October 5, Innsbruck, Austria*, pages 449–464. Springer-Verlag, 2012.

[39] C. A. González Pérez, F. Büttner, R. Clarisó, and J. Cabot. EMFtoCSP: A Tool for the Lightweight Verification of EMF Models. In *Proceedings of Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA)*, Zürich, Schweiz, 2012.

[40] H. Götz, M. Nickolaus, T. Rošner, and K. Salomon. *iX Studie Modellbasiertes Testen - Modellierung und Generierung von Tests - Grundlagen, Kriterien für Werkzeugeinsatz, Werkzeuge in der Übersicht.* Heise Zeitschriften Verlag, 2009.

[41] P. Grässle, H. Baumann, and P. Baumann. *UML 2 projektorientiert.* Galileo Computing, 4. edition, 2007.

[42] T. Grechenig, M. Bernhart, R. Breiteneder, and K. Kappel. *Softwaretechnik: Mit Fallbeispielen aus realen Entwicklungsprojekten.* Pearson Studium, 1. edition, 2009.

[43] V. Gruhn, D. Pieper, and C. Röttgers. *MDA. Effektives Software-Engineering mit UML 2 und Eclipse.* Springer Verlag, 2006.

[44] L. Grunske, L. Geiger, and M. Lawley. A Graphical Specification of Model Transformations with Triple Graph Grammars. In A. Hartman and D. Kreische, editors, *Model Driven Architecture – Foundations and Applications*, volume 3748 of *Lecture Notes in Computer Science*, pages 284–298. Springer Berlin Heidelberg, 2005.

[45] S. Hildebrandt, L. Lambers, and H. Giese. Complete Specification Coverage in Automatically Generated Conformance Test Cases for TGG Implementations. In *Theory and Practice of Model Transformations (Proceedings of 6th International Conference, ICMT 2013), June 18–19, Budapest, Hungary*, pages 174–188. Springer-Verlag, 2013.

[46] P. Hofstedt and A. Wolf. *Einführung in die Constraint-Programmierung*. Cambridge University Press, 1. edition, 2007.

[47] P. Huber. The Model Transformation Language Jungle - An Evaluation and Extension of Existing Approaches. Master's thesis, Vienna University of Technology, 2008.

[48] IEEE Standards Association. IEEE SA - 829-2008 - IEEE Standard for Software and System Test Documentation. `http://standards.ieee.org/findstds/standard/829-2008.html`, Zuletzt besucht: 15-04-2013.

[49] R. Irrgang. *Entscheidungstabellen-Technik: Entscheidungstabellen erstellen und analysieren*. expert, 1. edition, 1995.

[50] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[51] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: The Alloy constraint analyzer. In *Proceedings of the 22nd international conference on Software engineering (ICSE'2000), Limerick Ireland, June 4-11*, pages 730–733. ACM, 2000.

[52] F. Jouault and I. Kurtev. Transforming Models with ATL. In *MoDELS Satellite Events*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2006.

[53] C. Koch. The DLV Tutorial. `http://www.dbai.tuwien.ac.at/proj/dlv/tutorial/`, Zuletzt besucht: 05-03-2013.

[54] T. Kühne. Matters of (Meta-)Modeling. *Software and Systems Modeling*, 5(4):369–385, 2006.

[55] J.M. Küster and M. Abd-El-Razik. Validation of Model Transformations – First Experiences using a White Box Approach. In *Proceedings of the MoDeVa'06 (Model Design and Validation Workshop associated to MoDELS'06), October 2, Genova, Italy*. Springer-Verlag, 2006.

[56] H. Le Guen. *Validation d'un logiciel par le test statistique d'usage : de la modelisation de la decision à la livraison*. PhD thesis, l'université de Rennes 1, 2005.

[57] N. Leone, G. Pfeifer, W. Faber, T. Eiter, Gottlob G., S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 7:499–562, 2002.

[58] V. Lifschitz. What is answer set programming? In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17*, pages 1594–1597. AAAI Press, 2008.

[59] P. Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, 2. edition, 2009.

[60] J. Link. *Softwaretests mit JUnit*. dpunkt.verlag, 2. edition, 2005.

[61] J. A. Mcquillan and J. Power. White-Box Coverage Criteria for Model Transformations, 2009.

[62] T. Mens and P. Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[63] J. Minker and C. Ruiz. Semantics for Disjunctive Logic Programs with Explicit and Default Negation. *Fundamenta Informaticae*, 20, 1994.

[64] R. Mitchell and J. McKim. *Design by Contract, by Example*. Addison Wesley Publishing Company Inc, 1. edition, 2001.

[65] P. Morgan, A. Samaroo, and G. Thompson. *Software Testing: An Istqb-Iseb Foundation Guide*. British Informatics Society Ltd, 1. edition, 2010.

[66] J.-M. Mottu, S. Sen, M. Tisi, and J. Cabot. Static Analysis of Model Transformations for Effective Test Generation. In *Proceedings of IEEE 23rd International Symposium on Software Reliability Engineering, November 27–30, Dallas, Texas, USA*, pages 291–300. IEEE CS Press, 2012.

[67] P.-A. Muller, F. Fleurey, D. Vojtisek, Z. Drey, D. Pollet, F. Fondement, and J.-M. Jézéquel. On executable meta-languages applied to model transformations. In *Proceedings of the Model Transformations In Practice Workshop (MTIP)*, 2005.

[68] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An A-Prolog decision support system for the Space Shuttle. In *PADL 2001*, pages 169–183. Springer Verlag Berlin, Heidelberg, 2000.

[69] J.R. Norris. *Markov Chains*. Cambridge University Press, 1. edition, 1998.

[70] Object Management Group. MOF 2.4.1. `http://www.omg.org/spec/MOF/2.4.1/PDF/`, Zuletzt besucht: 30-07-2013.

[71] Object Management Group. Object Constraint Language, Version: 2.3.1. `http://www.omg.org/spec/OCL/2.3.1/PDF`, Zuletzt besucht: 21-01-2013.

[72] Object Management Group. OMG. `http://www.omg.org/gettingstarted/gettingstartedindex.htm`, Zuletzt besucht: 30-07-2013.

[73] Object Management Group. OMG Formal Specifications. `http://www.omg.org/spec/index.htm`, Zuletzt besucht: 30-07-2013.

[74] Object Management Group. QVT 1.1. `http://www.omg.org/spec/QVT/1.1/PDF/`, Zuletzt besucht: 30-07-2013.

[75] Object Management Group. UML 2.4.1. `http://www.omg.org/spec/UML/2.4.1/`, Zuletzt besucht: 30-07-2013.

[76] J. Oetsch, J. Pührer, M. Seidl, H. Tompits, and P. Zwickl. VIDEAS: A Development Tool for Answer-Set Programs Based on Model-Driven Engineering Technology. In J. P. Delgrande and W. Faber, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 6645 of *Lecture Notes in Computer Science*, pages 382–387. Springer Berlin Heidelberg, 2011.

[77] M. Ornaghi, C. Fiorentini, A. Momigliano, and F. Pagano. Applying ASP to UML Model Validation. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR '09)*, pages 457–463. Springer Verlag Berlin, Heidelberg, 2009.

[78] T.J. Ostrand and M.J. Balcer. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, 31(6):676–686, 1988.

[79] M. Pol, T. Koomen, and A. Spillner. *Management und Optimierung des Testprozesses*. dpunkt.verlag, 2. edition, 2002.

[80] M. Richters and M. Gogolla. OCL: Syntax, Semantics, and Tools. In T. Clark and J. Warmer, editors, *Object Modeling with the OCL*, pages 42–68. Springer-Verlag, 2002.

[81] T. Rošner. Von höherer Ebene - Modellbasiertes Testen - eine Einführung. *iX Magazin für professionelle Informationstechnik*, 11:125–130, 2009.

[82] T. Rošner, C. Brandes, H. Götz, and M. Winter. *Basiswissen Modellbasierter Test*. dpunkt.verlag, 1. edition, 2010.

[83] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[84] J. Rushby. Automated Test Generation and Verified Software. In B. Meyer, editor, *Verified Software: Theories, Tools, Experiments*, pages 161–172. Springer-Verlag, 2008.

[85] I. Schieferdecker. Modellbasiertes Testen. *OBJEKTspektrum*, 3:39–45, 2007.

[86] J. Schönböck. *Testing and Debugging of Model Transformations*. PhD thesis, Technische Universität Wien, 2011.

[87] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '94), Herrsching (D)*. Springer-Verlag, 1995.

[88] W. Schwinger and N. Koch. P=?=NP. In *Algorithmen und Komplexität*, pages 138–152. Carl Hanser Verlag, 2003.

[89] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

[90] S. Sendall and W. Kozaczynski. Model transformation: The Heart and Soul of Model-Driven Software Development. *Software, IEEE*, 20:42–45, 2003.

[91] C. Sensler, M. Kunz, and P. Schnell. Testautomatisierung mit modellgetriebener Testskript- Entwicklung. *OBJEKTspektrum*, 3:75–83, 2006.

[92] H.M. Sneed, M. Baumgartner, and R. Seidl. *Der Systemtest - Von den Anforderungen zum Qualitätsnachweis*. Carl Hanser Verlag, 3. edition, 2011.

[93] A. Spillner and T. Linz. *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester - Foundation Level nach ISTQB-Standard*. dpunkt.verlag, 5. edition, 2012.

[94] T. Stahl, M. Völter, S. Efftinge, and A. Haase. *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management*. dpunkt.verlag, 2. edition, 2007.

[95] G.E. Thaller. *Software-Test: Verifikation und Validation*. Heise Medien, 2. edition, 2002.

[96] M. Utting and B. Legeard. *Practical Model-Based Testing. A Tools Approach*. Morgan Kaufmann, 1. edition, 2007.

[97] Lifschitz; V. Answer Set Programming and Plan Generation. *Artificial Intelligence*, 138:39–54, 2002.

[98] U. Vigenschow. *Objektorientiertes Testen und Testautomatisierung in der Praxis*. dpunkt.verlag, 1. edition, 2005.

[99] I. Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. PhD thesis, RWTH Aachen University, 2012.

[100] M. Wieber and A. Schürr. Systematic Testing of Graph Transformations: A Practical Approach Based on Graph Patterns. In *Theory and Practice of Model Transformations (Proceedings of 6th International Conference, ICMT 2013), June 18–19, Budapest, Hungary*, pages 205–220. Springer-Verlag, 2013.

[101] J. Zander, I. Schieferdecker, and P.J. Mostermann. *Model-Based Testing for Embedded Systems (Computational Analysis, Synthesis, and Design of Dynamic Systems)*. CRC Press, 1. edition, 2011.