Software-Qualitätssicherung durch Automatisierung – Ein modellbasierter Ansatz

Andreas Metzger

Technische Universität Kaiserslautern
research@andreas-metzger.net

Abstract: Die Qualitätssicherung von Softwareprodukten kann effizient durch eine Automatisierung von Entwicklungsaktivitäten erfolgen. In meiner Dissertation schlage ich hierzu ein verbessertes modellbasiertes Vorgehen vor. Gegenüber bisher eingesetzten Metamodellierungstechniken erlaubt mein Ansatz eine deutlich mächtigere und kompaktere Form der Multiebenenmodellierung. Zur Realisierung von Modelltransformationen und -analysen wird darüber hinaus eine angepasste Aktionssprache vorgestellt, welche die kompakte und allgemeingültige Beschreibung von Operationen auf unterschiedlichen Modellebenen erlaubt.

Angewendet werden diese Ansätze zur Automatisierung von Qualitätssicherungsmaßnahmen, zu welchen u.a. die automatische Identifikation von Inkonsistenzen in Entwicklungsdokumenten und die maschinelle Detektion von Feature-Interaktionen in Eingebetteten Systemen zählen. Ein „virtuelles Software-Labor“, welches durch die iterative Messung, Bewertung und Modifikation ausführbarer Modelle eine vollautomatische Untersuchung von Software-Eigenschaften ermöglicht, stellt die umfassendste Anwendung der Automatisierungstechniken in meiner Arbeit dar.

1 Einleitung

Komplexität und Größe moderner softwareintensiver Systeme steigen stetig. Wo z.B. im KFZ-Bereich im letzten Jahrzehnt Programmgrößen von wenigen 100kB üblich waren [Sc99], erreichen diese Systeme heute Code-Größen von über 60MB [Sa03]. Diese Zunahme stellt ein erhebliches Problem für die Software-Entwicklung dar, da die Entwicklungsmethoden und -techniken oft nicht skalieren. Es bleibt nicht genügend Zeit für die Qualitätssicherung, worunter die Qualität des fertigen Produkts leidet.

Als Lösung dieses Problems schlage ich in meiner Arbeit [Me04a] die *Automatisierung von Software-Entwicklungsaktivitäten* vor. Dies hat die folgenden Vorteile:

- Viele für den Menschen schwierige Aktivitäten können effizient und fehlerfrei von einer „Maschine“ durchgeführt werden.
- Der Zeitgewinn, der durch eine Automatisierung erzielt wird, kann bei gleich bleibendem Budget zusätzlich für Qualitätssicherungsmaßnahmen aufgewendet werden.
- Die in Automatisierungswerkzeugen realisierte Wiederverwendung führt zu einer höheren Qualität, da wiederverwendete Artefakte i.d.R. gründlich überprüft wurden.
- Die Automatisierung von Messungen erlaubt die kosteneffiziente, permanente Qualitätskontrolle und Aufwandsabschätzung während der Projektdurchführung.
- Durch eine automatische Konsistenzhaltung von Software-Artefakten (Modellen und Code) erhöht sich die Wartbarkeit des Software-Produkts.

Die Automatisierung von Entwicklungsaktivitäten wird in meiner Arbeit durch eine *modellbasierte Software-Entwicklung* erreicht. Modelle sind abstrakter und semantisch reichhaltiger als Programm-Code. Daher können die Spezifikation von Entwicklungsartefakten und die Erstellung von Automatisierungswerkzeugen effizienter als bei code-basierten Ansätzen erfolgen. Meine Arbeit leistet einen Beitrag zur weiteren Verbesserung der modellbasierten Software-Entwicklung. Mit dem in meiner Dissertation vorgestellten Ansatz lassen sich Modelle und Automatisierungsalgorithmen weit abstrakter und allgemeingültiger als bisher beschreiben. Damit sind die Automatisierungswerkzeuge über viele Projekte hinweg einsetzbar.

In dieser Kurzdarstellung liegt der Schwerpunkt auf den Verbesserungen der modellbasierten Software-Entwicklung. Die damit automatisierten Qualitätssicherungsmaßnahmen, die zugleich als Nachweis für die Tauglichkeit meines Ansatzes dienen, können wegen des beschränkten Platzes nur skizziert werden.

2 Modellbasierte Software-Entwicklung

Die Automatisierung von Entwicklungsaktivitäten setzt eine präzise Kenntnis des Software-Entwicklungsprozesses voraus. Daher werden explizite Modelle von Artefakten (*Produktmodelle*) und von Aktivitäten (*Prozessmodelle*) benötigt. Nur dadurch werden Entwicklungsaktivitäten wiederhol- und schließlich auch automatisierbar. In meiner Arbeit werden Produktmodelle als objektorientierte Metamodelle formalisiert, wobei eine verbesserte Form der Multiebenenmodellierung zum Einsatz kommt (siehe Abschnitt 2.1). Die Aktivitäten werden im Prozessmodell imperativ mit Hilfe der von mir konzipierten Aktionssprache AL++ spezifiziert (siehe Abschnitt 2.2) und sind damit direkt ausführbar.

2.1 Verbesserte Multiebenenmodellierung

Gängige Ansätze zur Metamodellierung basieren auf der Grundidee, dass Metamodelle mit denselben Modellierungskonzepten wie „normale“ Modelle beschrieben werden können. Ein bekanntes Beispiel ist die UML, bei welcher die abstrakte Syntax der Sprache (das Metamodell) mit der UML selbst (genauer mit MOF, einer „Teilmenge“ der UML [OM02]) beschrieben wird. Zur semantisch reichhaltigeren Beschreibung von Metamodellen fehlt es diesen Modellierungssprachen jedoch an Konzepten. Dies kann bereits an einem Beispiel zur Spezifikation einfacher Grafikprogramme illustriert werden. In Abb. 1 ist zunächst die Modellebene mit Hilfe eines UML Klassendiagramms dargestellt. Mit dem Grafikprogramm können zweidimensionale Häuser und Bäume gezeichnet werden. Diese Grafikelemente bestehen aus weniger komplexen Elementen, die durch die Koordinaten der Eckpunkte beschrieben werden. Um eine Orientierungshilfe in der Modellhierarchie zu

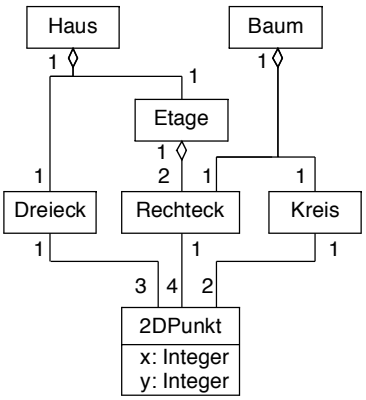

Abb. 1: Modellebene M_1

geben, werden die Ebenen der Hierarchie beginnend bei M_0 (Ebene der Laufzeitinstanzen) aufsteigend nummeriert.

Ein Metamodell (Produktmodell), das der Spezifikation aller sinnvoll möglichen Grafikprogramme (Produkte) dient, beinhaltet Typen von Elementen der Ebene M_1. Ein solches Metamodell (Modellebene M_2) ist in Abb. 2 gezeigt.

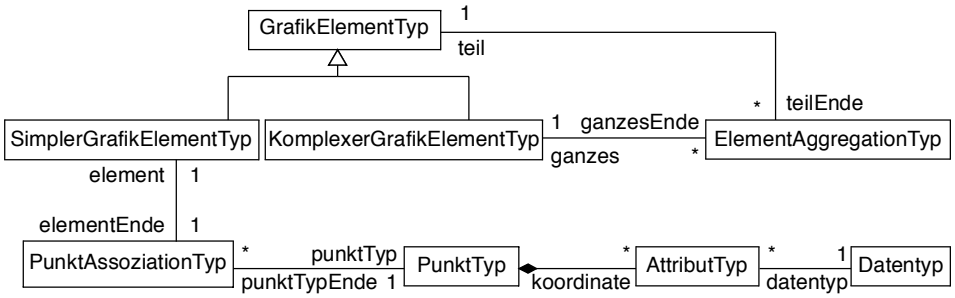

Abb. 2: Modellebene M_2

Es lassen sich hierbei zwei Typen von Grafikelementen identifizieren: simple Grafikelementtypen und komplexe Elementtypen. Des Weiteren werden in diesem Metamodell die Typen von Relationen zwischen M_1-Modellelementen angegeben. Mit den Konzepten der existierenden Modellierungssprachen erfolgt dies durch die Definition eines Objekttyps ElementAggregationTyp, der über die entsprechenden Assoziationen mit den beteiligten Elementtypen verbunden ist. Da sowohl auf der Seite von KomplexerGrafikElementTyp als auch auf der Seite von GrafikElementTyp eine Multiplizität von eins gefordert ist, beschreibt dies binäre Assoziationen auf M_1. Analog erfolgt die Modellierung der Beziehung zwischen simplen Grafikelementen und den Punkten, welche die Form des Elements angeben. Zuletzt müssen noch die möglichen Koordinaten eines Punkts spezifiziert werden, was durch eine Kompositionsbeziehung von PunktTyp zu AttributTyp erfolgt.

Das Defizit einer solchen Metamodellierung ist, dass nicht formal angegeben ist, ob ein Objekttyp auf M_2 zu einem Objekttyp, zu einer Relation oder zu einem Attribut auf M_1 instanziiert wird. Diese Information steckt alleine im Namen der Modellelemente, die jedoch keinen formalisierten Teil des Metamodells darstellen. Mit einem einfachen Gedankenexperiment lässt sich dieses Problem nachweisen. Ersetzt man alle Namen in Abb. 2 durch unsinnige Bezeichner (z.B. in der Form xyz), dann lässt sich nicht mehr nachvollziehen, ob und zu welcher Art von Element das jeweilige M_2-Element instanziiert wird.

In meiner Dissertation wird dieses Defizit durch einen Ansatz zur semantisch reichhaltigeren Beschreibung von Metamodellen beseitigt [Me04a]. Als Grundlage dient der Ansatz der *tiefen Instanziierung* von Atkinson et al. [At01]. Die Grundidee dieses Ansatzes ist, die instanziierbaren Elemente einer jeden Modellebene an Hand von Eigenschaften abzuleiten, die formal im *Modell* spezifiziert wurden. Im Gegensatz dazu muss, wie wir gesehen haben, beim „traditionellen“ Ansatz eine im Modell nicht sichtbare *Vereinbarung* getroffen werden, ob und wie ein Element der Ebene M_n erneut zu einer Instanz der Ebene M_{n-1} instanziiert wird. Zur Beschreibung der möglichen Instanziierungstiefe wird das

Konzept der *Potenz* eingeführt. Die Potenz gibt an, bis zu welcher Tiefe ein Modellelement instanziiert werden kann. Jede Instanziierung verringert die Potenz. Ein Element der Potenz 0 entspricht einer Instanz, die nicht weiter instanziiert werden kann (also z. B. einer Attributbelegung oder einem Objekt). Der Ansatz der tiefen Instanziierung wurde von Atkinson et al. als Lösung für andere als die hier diskutierten Problemfelder erarbeitet. Die Ergänzung und Anpassung des Ansatzes zur Lösung der oben beschriebenen Probleme ist ein Hauptbeitrag meiner Arbeit. Die Lösungsansätze werden im Folgenden vorgestellt und anhand des Grafikbeispiels illustriert.

Das Modellelement AttributTyp im Grafikbeispiel kann durch Angabe einer Potenz von 2 formal als Attributtyp spezifiziert werden. Hierdurch wird ausgedrückt, dass die Attribute auf M_1 und die Attributbelegungen auf M_0 zu finden sind. Im Ansatz von Atkinson et al. ist hierbei allerdings nur die *implizite* Form der Instanziierung vorgesehen (bei der Instanziierung des Kontext-Objekttyps werden alle Attributtypen automatisch und mit dem auf der Meta-Ebene definierten Namen instanziiert). Attribute, deren Namen nicht im Voraus bekannt sind, können damit nicht spezifiziert werden. In meiner Arbeit führte ich daher zusätzlich *explizit instanziierbare* Attribute ein. Diese werden „manuell" instanziiert, wobei jeweils ein neuer Name bei der Instanziierung angegeben wird. Zur Unterscheidung werden die Namen implizit instanziierter Elemente in den Modellen unterstrichen.

Analog zu Attributtypen können auch Relationstypen formal im Metamodell beschrieben werden. Die Spezifikation von Relationen mit einer höheren Potenz als 1 wird von Atkinson et al. nicht detailliert beschrieben. In meiner Dissertation schlage ich die folgenden Konventionen vor: Die Art der Relation (Assoziation, Aggregation, Komposition oder Generalisierung) wird bei der Relationstypdefinition auf der höchsten Meta-Ebene festgelegt. Die Einschränkungen, die mit der jeweiligen Art der Relation verbunden sind (z. B. keine Zykeln in einer Generalisierungsrelation) greifen erst für die „traditionellen" Ebenen. Multiplizitäten und Rollennamen gelten hingegen jeweils für die nächst tiefere Meta-Ebene. Für jede Relation auf dieser tieferen Ebene können Rollennamen und Multiplizitäten wiederum völlig frei angegeben werden. Die M_2-Ebene des verbesserten Multiebenenmodells, das sich hieraus ergibt, ist in Abb. 3 gezeigt.

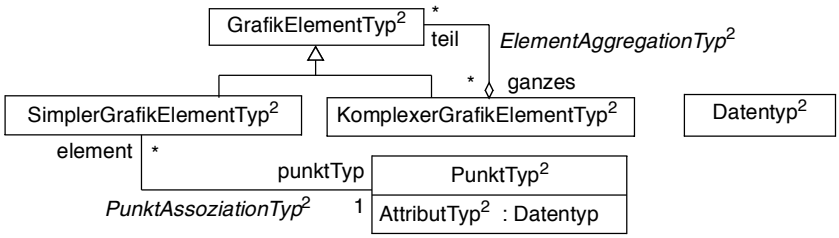

Abb. 3: Modellebene M_2 in verbesserter Multiebenenmodellierung

Im Vergleich zu dem Metamodell aus Abb. 2 zeigt sich deutlich der Gewinn an Kompaktheit und Verständlichkeit. Attribut- und Relationstypen werden in einer analogen Darstellung zu deren Instanzen repräsentiert, weshalb die Bedeutung dieser Modellelemente auch ohne Kenntnis des jeweiligen Namens ersichtlich ist. Die Potenz ist hochgestellt am Ende des Namens angegeben.

2.2 Aktionssprache AL++

Bei einer modellbasierten Software-Entwicklung entsprechen die Entwicklungsaktivitäten einzelnen *Modelltransformationen.* In meiner Arbeit entschied ich mich gegen die weit verbreitete Graphtransformation [An99] als Vertreter der deklarativen Modelltransformationsansätze. Stattdessen wählte ich einen imperativen Ansatz, da ein solcher Ansatz im Kontext der verbesserten Multiebenenmodellierung eine deutlich kompaktere und generischere Formulierung der Transformationsalgorithmen erlaubt. Das Ergebnis ist die Aktionssprache AL++, welche grundsätzliche Abstraktionen bestehender Aktionssprachen beinhaltet (siehe z.B. [Su01]), darüber hinaus aber spezielle Konstrukte zur Bearbeitung von Relations- und Attributtypen besitzt.

Die generische Beschreibung von Modelltransformationen erreicht man in AL++ durch die Definition von Operationstypen auf einer der höheren Meta-Ebenen. Wie bei den Attributtypen habe ich hierbei zwei Arten der Instanziierung unterschieden: implizit und explizit instanziierte Operationstypen. Implizit instanziierte Operationstypen werden verwendet, falls sämtliche zur Formulierung des Transformationsalgorithmus notwendigen Informationen bei der Definition des Operationstyps bekannt sind. Die explizite Form wird genutzt, falls zusätzliche Informationen bei der Instanziierung angegeben werden müssen.

Die wichtigsten Konstrukte der AL++ werden im Folgenden an einem Ausschnitt des Grafikbeispiels eingeführt. In Abb. 4 sind dafür zusätzlich die Instanziierungsbeziehungen zur Ebene M_2 durch @ gekennzeichnet.

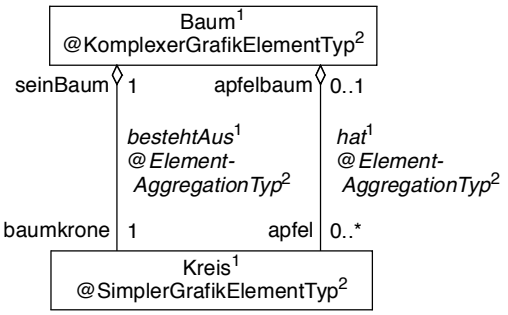

Abb. 4: Modellebene M_1 (Ausschnitt)

Zur Erzeugung neuer Relationstypen mit der AL++ wird der Name des Typs, die Multiplizitäten der Enden und die Rollennamen zusätzlich zu dem Ursprungs- und Zielobjekttyp als Parameter angegeben. Im Beispiel wird die bestehtAus-Relation durch den Ausdruck typBaum.teil += typKreis, ("bestehtAus", 1, 1, "seinBaum", "baumkrone") erzeugt. Dabei sind typBaum und typKreis Verweise auf die jeweiligen Objekttypen und teil der Rollenname, der auf Ebene M_2 definiert wurde (siehe Abb. 3). Der Zugriff auf eine existierende Relation erfolgt umgekehrt durch den Ausdruck (typBaum.ElementAggregationTyp, "baumkrone"), der ("bestehtAus", 1, 1, "seinBaum", Kreis) liefert. Die Angabe des Rollennamens (baumkrone) wird hierbei zur eindeutigen Auswahl der gewünschten Relation aus der Menge aller instanziierten Relationen benötigt. Die Angabe von typBaum.ElementAggregationTyp alleine genügt nicht, da mehr als eine Relationsinstanz dieses Typs vom Ursprungsobjekttyp typBaum ausgehen kann (so z.B. die hat-Relation in Abb. 4). Kennt man die Rollennamen am Ziel des Relationstyps nicht, so kann man diese über Anweisungen der Form typBaum.ElementAggregationTyp bestimmen. Im Beispiel liefert dies die Menge {"baumkro-

ne", "apfel"}. Diese Menge lässt sich bei Bedarf weiter präzisieren, indem man den Meta-Relationsnamen angibt. Damit wird die Richtung, in welcher die möglichen Relationsinstanzen betrachtet werden sollen, beschrieben (z.B. typBaum.teil).

Die AL++-Syntax zur imperativen Handhabung explizit instanziierter Attribute ist analog zu der Syntax für Relationstypen (siehe hierzu [Me04a, S.73]).

Um die verbesserte Multiebenenmodellierung voll nutzen zu können, werden weitere AL++-Konstrukte definiert:

- Der Typ eines jeden Modellelements kann durch den type-Operator bestimmt werden.
- Der #-Operator erlaubt an allen Stellen, an denen ein Identifier (z.B. ein Rollenname) erwartet wird, diesen zur Laufzeit aus einem String zu „erzeugen".
- Der anyinstance-Operator liefert einen „gedachten" Supertyp. Dies vereinfacht den programmatischen Zugriff auf die Instanzen verschiedener Subtypen eines Metatyps. So ist GrafikElementTyp.anyinstance der „gedachte" Supertyp für Baum und Kreis.

2.3 Beispiel „Multiplizitätsprüfung"

Die Mächtigkeit und Generizität der Sprache soll an einem Beispiel zur automatischen Prüfung von Multiplizitäten dargestellt werden. Um die höchstmögliche Generizität und damit langfristige Wiederverwendbarkeit zu erreichen, wird eine weitere Meta-Ebene definiert. Diese ist in Abb. 5 gezeigt. In diesem Meta-Metamodell werden die grundlegenden objektorientierten Konzepte definiert, wie z.B. Objekttypen und Relationstypen. Damit kann die Multiplizitätsprüfung dann wie folgt in der implizit instanziierten check()-Operation realisiert werden:

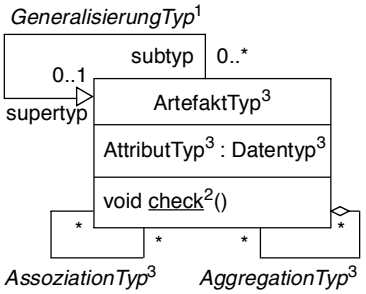

Abb. 5: Modellebene M_3

```
void check2() {
    ArtefaktTyp aTyp = this.type;
    while(aTyp != null) {
        foreach(String rName in aTyp.AggregationTyp) {
            Multiplicity rMult;
            (null, null, rMult, null, null) = (aTyp.AggregationTyp, rName);
            Set rSet = this.#rName;
            Integer rSize = rSet.size();
            if((rSize < rMult.lower) || (rSize > rMult.upper)) /* Fehlermeldung */
            /* analog für AssoziationTyp */
        }
        aTyp = aTyp.supertyp;
    }
}
```

Durch die Definition der check-Operation mit der Potenz 2 erhält man Operationsinstanzen auf der Ebene M_1. Bei der Ausführung der Operation wird zunächst der Typ des aktuellen Elements (this) bestimmt, für welches die Relationen untersucht werden. Da eine Generalisierungsbeziehung zwischen M_2-Elementen erlaubt ist, muss die Generalisierungshierarchie durchlaufen werden, um alle Relationstypdefinitionen zu identifizieren. Im AL++-Code ist dies durch die while-Schleife realisiert, die beim „Wurzelelement" terminiert. Für jede Ebene in dieser Hierarchie werden alle Instanzen von AggregationTyp analysiert. Dazu wird in zwei Schritten vorgegangen. Zuerst wird die Menge aller Rollennamen über aTyp.AggregationTyp abgefragt und für jeden Rollennamen einzeln die Multiplizität auf der Seite des Rollennamens ermittelt. Dann wird die tatsächliche Anzahl an Elementen, die auf der jeweiligen Seite der Relation auf Ebene M_1 zu finden sind, bestimmt. Dies erfolgt mit den Mitteln herkömmlicher Aktionssprachen, wobei # als Operator für den dynamischen Zugriff auf die Relationsenden eingesetzt wird. Analog wird für die Instanzen von AssoziationTyp vorgegangen.

Die check()-Operation ist für alle Objekttypen derjenigen M_1-Modelle definiert, die aus dem generischen M_3-Modell entstanden sind. Damit kann diese Operation in vielen Projekten eingesetzt werden, was einen Effizienzgewinn für jedes Folgeprojekt bedeutet.

3 Qualitätssicherung durch Automatisierung

Die Multiplizitätsprüfung ist eine recht einfache Maßnahme zur Qualitätssicherung, die hier nur der Veranschaulichung der verbesserten Metamodellierung und der Aktionssprache AL++ dienen sollte. Ein zweiter Teil meiner Arbeit beschäftigt sich mit automatisierten Qualitätssicherungsmaßnahmen, die in einer solchen Form bisher nicht realisiert wurden. Zwei Vertreter dieser Ansätze werden im Folgenden vorgestellt.

3.1 Detektion von Feature-Interaktionen in Eingebetteten Systemen

Eine Feature-Interaktion (eine unerwünschte Wechselwirkung zwischen Produktmerkmalen) tritt immer dann auf, wenn das Gesamtverhalten eines Systems nicht die Spezifikation seiner einzelnen Produktmerkmale erfüllt [Gi97]. In der Telekommunikationsdomäne ist dieses Problem seit geraumer Zeit bekannt [Ca03]. Für Eingebettete Systeme wurde diese Problemklasse bisher nicht systematisch betrachtet. Im Gegensatz zu Telekommunikationssystemen sind solche Systeme immer in eine physikalische Umgebung eingebettet. Es genügt daher nicht, Feature-Interaktionen nur innerhalb eines Software-Systems zu behandeln. Auch Wechselwirkungen, die durch die Umgebung entstehen, müssen zwingend berücksichtigt werden.

In meiner Arbeit schlage ich Algorithmen vor, welche die Detektion von Feature-Interaktionen bereits auf der Ebene der Anforderungen und der Grobarchitektur erlauben (siehe auch [Me04b]). Hierbei werden Nachvollziehbarkeitsinformationen zwischen Modellelementen ausgenutzt und Modelle der physikalischen Umgebung eingesetzt, welche die Wechselwirkungen zwischen verschiedenen physikalischen Effekten beschreiben. Vorteil

der AL++ zur Formulierung der Detektionsalgorithmen ist, dass sich die Traversion der Anforderungsgraphen rekursiv und damit sehr kompakt spezifizieren lässt. Der gesamte Detektionsalgorithmus benötig lediglich 35 Zeilen AL++-Code [Me04a, S.211f.].

Der Detektionsansatz wurde in drei Fallstudien eingesetzt, wovon die umfangreichste die automatische Analyse eines komplexen Gebäudeautomationssystems mit 331 Anforderungen war. Es wurden hierbei 44 Interaktionen identifiziert, von denen sich zwei als kritisch herausstellten und auf Fehler in der Realisierung des Software-Systems zurückgeführt werden konnten. Bei der Anwendung der Detektionswerkzeuge zeigt sich deutlich die Komplexitätsbeherrschung durch eine Automatisierung. Ohne die Werkzeugunterstützung hätte man 371 Modellelemente und 298 Nachvollziehbarkeits-Links manuell untersuchen müssen. Abb. 6 zeigt einen Ausschnitt aus dem Anforderungsgraphen zur Illustration dieser Komplexität (graue Knoten sind Interaktionspunkte).

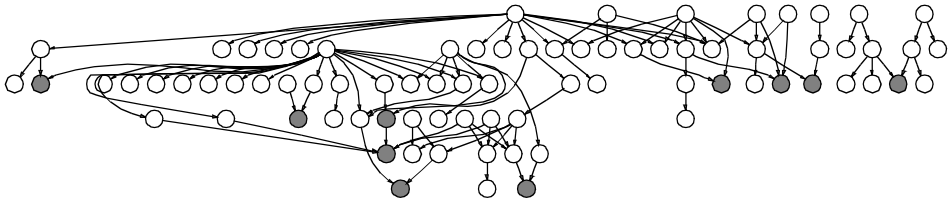

Abb. 6: Ausschnitt aus einem Anforderungsgraphen

3.2 „Virtuelles Software-Labor“

Modellbasierte Entwicklungsansätze erlauben durch die frühe Verfügbarkeit ausführbarer Artefakte (Prototypen) bereits am Anfang der Software-Entwicklung Validierungen durchzuführen. Damit können solche Benutzeranforderungen identifiziert werden, welche die Systemkomplexität erhöhen, ohne einen für den Benutzer relevanten Nutzen darzustellen (sog. „Vergoldung“ des Produkts). Modellbasierte Ansätze tragen somit indirekt zur *Vermeidung* einer zu hohen Komplexität bei, da irrelevante Anforderungen eliminiert werden können. In meiner Arbeit schlage ich die Technik des „virtuellen Software-Labors“ vor, in welchem die vollautomatische Durchführung von Experimenten (in der Form von Testläufen) möglich ist. Damit erreicht man eine umfassende Sammlung von Daten (Resultate der Experimente), die als Grundlage für die Auswahl relevanter Anforderungen dienen können. Abb. 7 zeigt schematisch, welche Aktivitäten in diesem „virtuellen Labor“ vollautomatisch durchgeführt werden können.

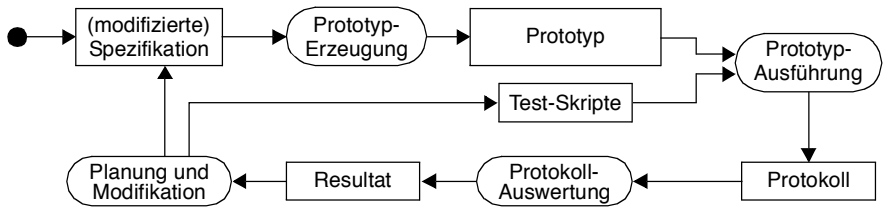

Abb. 7: Modellebene M_3

Zunächst wird aus einer initialen Spezifikation (einem Modell) ein Prototyp erzeugt. Während der maschinellen Ausführung, die durch Test-Skripte getrieben ist, werden festgelegte Messdaten gesammelt. Diese werden anhand einer jeweils auf die Fragestellung angepassten Heuristik automatisch ausgewertet. Auf Basis dieser Resultate wird die maschinelle Planung und Modifikation der Modelle und der Test-Skripte vorgenommen. Dieser Zyklus wird solange wiederholt bis die gewünschten Ergebnisse vorliegen oder das Experiment vom Experimentator abgebrochen wird.

Ein spannendes Resultat konnte ich durch die Analyse einer Raumtemperaturregelung in diesem „virtuellen Labor“ erzielen. Das Ziel war die Erstellung einer energieoptimalen Regelung, die einen gewünschten Benutzerkomfort erreicht. Hierzu wurden 4140 Einzelexperimente vollautomatisch durchgeführt und sowohl der Benutzerkomfort als auch der Energieverbrauch unterschiedlicher Regelungsstrategien gemessen. Es stellte sich heraus, dass Regelungsstrategien, die sich dynamisch an das Benutzerverhalten anpassen, nicht notwendig sind. Die Software-Realisierung wird daher deutlich einfacher. Für Eingebettete Systeme kann damit zusätzlich eine Einsparung der Kosten verbunden sein, weil eventuell ein „kleinerer“ Mikrokontroller eingesetzt werden kann.

4 Zusammenfassung

In meiner Dissertation konnte ich zeigen, wie durch eine modellbasierte Automatisierung von Software-Entwicklungsaktivitäten ein Effizienz- und Qualitätsgewinn erzielt und die Komplexität von Software-Systemen beherrscht werden kann. Neue wissenschaftliche Ergebnisse wurden hierbei sowohl für die Grundlagen einer modellbasierten Entwicklung als auch für die Anwendung dieser modellbasierten Techniken zur Realisierung automatisierter Qualitätssicherungsmaßnahmen erzielt. Die wichtigsten Resultate meiner Arbeit sind:

- Die *verbesserte Multiebenenmodellierung*, welche eine semantisch reichhaltigere Beschreibung von Metamodellen als bisherige Metamodellierungsansätze (wie sie z. B. zur Definition der UML verwendet werden) erlaubt. Dies wird durch die explizite Spezifikation der Instanziierbarkeit aller Modellelemente (Objekttypen, Relationen, Attribute und Operationen) erreicht.
- Die *Aktionssprache AL++* zur imperativen Beschreibung von Modelltransformationen im Kontext der verbesserten Multiebenenmodellierung. Durch die Einbettung der Sprache in die verbesserte Multiebenenmodellierung wird die Beschreibung sehr generischer Algorithmen möglich, welche daher langfristig einsetzbar sind.
- Die *automatische Detektion von Feature-Interaktionen*, die in dieser Form erstmalig für den Bereich der Eingebetteten Systeme durchgeführt wurde. Realisiert wird diese Detektion durch Algorithmen in AL++, welche auf Anforderungs- und Architekturmodellen arbeiten.
- Ein „*virtuelles Software-Labor*“, in welchem vollautomatische Experimente durchgeführt werden können. Dies geschieht auf Basis der Generierung von Prototypen aus Spezifikationen (Modellen), der vollständig automatisierten Ausführung solcher Prototypen, der Evaluierung der Prototyp-Läufe und schließlich der maschinellen Modifikation der Spezifikationen.

- Der Nachweis, dass eine *Effizienzsteigerung durch Automatisierung* möglich ist. Alleine die automatische Erzeugung und Konsistenzprüfung von Entwicklungsdokumenten führten in einer Fallstudie zu einem Produktivitätsgewinn von 54 % gegenüber einer Fallstudie ohne eine solche Automatisierung [Me04a, S. 301ff.].

Literatur

[An99] M. Andries, G. Engels, A. Habel et al. "Graph Transformation for Specification and Programming" *Science of Computer Programming*. 34(1). Amsterdam: Elsevier Science. 1999. S. 1–54

[At01] C. Atkinson, T. Kühne. "The Essence of Multilevel Metamodeling" in *Proceedings UML 2001*. LNCS 2185. Heidelberg: Springer-Verlag. 2001. S. 19–33

[Ca03] M. Calder, M. Kolberg, E. Magill. "Feature Interaction: A Critical Review and Considered Forecast" *Computer Networks*. 41(1). 2003. S. 115–141

[Gi97] J. P. Gibson. "Feature Requirements Models: Understanding Interactions" in *Feature Interactions In Telecommunications IV*. Amsterdam: IOS Press. 1997

[Me04a] A. Metzger. *Software-Qualitätssicherung durch Automatisierung – Ein modellbasierter Ansatz*. Dissertation. Fachbereich Informatik. TU Kaiserslautern. 2004. Schriftenreihe Informatik. Band 21. ISBN 3-936890-60-9

[Me04b] A. Metzger. "Feature Interactions in Embedded Control Systems" *Computer Networks*. 45(5). 2004. S. 625–644

[OM02] OMG. *Meta Object Facility (MOF)*. Version 1.4. Document Number: formal/02-04-03. Object Management Group. 2002

[Sa03] A. Saad. "Prototyping bei der BMW Car IT GmbH" *JavaSPEKTRUM*. 2. Troisdorf: SIGS-DATACOM. 2003. S. 49–53

[Sc99] M. Schütze. *Eine musterbasierte Methode zur domänenspezifischen Modellierung und Generierung von Softwarekomponenten*. Dissertation. Aachen: Shaker Verlag. 1999

[Su01] G. Sunye, F. Pennaneac'h, W.-M. Ho. "Using UML Action Semantics for Executable Modeling and Beyond" in *Proceedings CAiSE 2001*. LNCS 2068. Heidelberg: Springer-Verlag. 2001. S. 433–447

Andreas Metzger wurde am 4. April 1973 in Annweiler geboren. Er besuchte dort das Trifelsgymnasium, wo er das Abitur mit einem Notendurchschnitt von 1,2 absolvierte. Im Anschluss studierte er Informatik mit Nebenfach Wirtschaftswissenschaften an der Universität Kaiserslautern. Im Jahr 1998 erhielt er das Diplom in Informatik mit der Note „sehr gut". Herr Metzger promovierte als wissenschaftlicher Mitarbeiter in der Arbeitsgruppe von Prof. Dr. G. Zimmermann (Fachbereich Informatik) und im Sonderforschungsbereich 501 „Entwicklung großer Systeme mit generischen Methoden" der TU Kaiserslautern. Während dieser Zeit engagierte sich Herr Metzger rege als Vertreter der wissenschaftlichen Mitarbeiter im Fachbereichsrat und im Fachausschuss für Studium und Lehre. Im Jahr 1999 nahm er einen sechswöchigen Forschungsaufenthalt an der Carnegie Mellon University, Pittsburgh, USA wahr. Für seine Dissertation und die anschließende Disputation im Dezember 2004 erhielt er den Dr.-Ing. mit der Note „mit Auszeichnung bestanden". Seit November 2004 ist Andreas Metzger in der Arbeitsgruppe „Software Systems Engineering" (Prof. Dr. Klaus Pohl) der Universität Duisburg-Essen beschäftigt, wo er seit Januar 2005 als wissenschaftlicher Assistent forscht und lehrt.