Modell-zu-Modell-Transformation von Modellen von Benutzerschnittstellen

TECHNISCHE UNIVERSITÄT DARMSTADT

Vom Fachbereich Informatik
zur
Erlangung des Grades eines Doktor-Ingenieurs (Dr.-Ing.)
genehmigte

D i s s e r t a t i o n

vorgelegt von

Dipl.-Inform. Andreas Petter

aus Ölbronn-Dürrn

Berichterstatter:	Prof. Dr. Max Mühlhäuser
Mitberichterstatter:	Prof. Dr. Heiko Krumm
Tag der Einreichung:	22.11.2011
Tag der mündlichen Prüfung:	24.01.2012

Darmstadt, 2012
Hochschulkennziffer D17

Zusammenfassung

Durch die ständig fortschreitende Diversifizierung der Geräte in der Computertechnik ergeben sich für Softwarehersteller neue Herausforderungen im Bereich Benutzerschnittstellen. Benutzerschnittstellen sollten an die neuen Geräte angepasst werden, um größtmögliche Benutzbarkeit zu erzielen.

Aus diesem Grund wird schon seit längerem modell-getriebene Softwareentwicklung von Benutzerschnittstellen mit Modell-zu-Modell-Transformationen als möglichem Lösungsansatz untersucht. Hierbei werden Modelle, die von den spezifischen Eigenschaften der Geräte abstrahieren, in "konkrete" Modelle transformiert, die an deren Eigenschaften angepasst sind. Diese Modelle können dann direkt durch einen Interpreter dargestellt oder durch eine weitere Transformation in Quelltext überführt werden.

Dadurch, dass für viele Geräte auch viele konkrete Modelle erzeugt werden müssen, werden auch entsprechend viele Modell-zu-Modell-Transformationen benötigt. Üblicherweise werden diese Modell-zu-Modell-Transformationen von Entwicklern per Hand entwickelt. Der Entwicklungsaufwand ist entsprechend hoch. Um diesen zu reduzieren, können deklarative Modell-zu-Modell-Transformationssprachen eingesetzt werden, die die Programmierung erleichtern.

Diese Arbeit beschäftigt sich mit der Frage, welche Konzepte solche Sprachen unterstützen müssen, damit sie sich "gut" zur Programmierung von Transformationen eignen. Ein Schwerpunkt dieser Arbeit liegt dabei auf der Generierung von Modellen grafischer Benutzerschnittstellen, da diese den Markt dominieren. Um dieses Problem anzugehen, werden Constraint Solving, Optimierung und verschiedene weitere Erweiterungen in Transformationssprachen integriert. Diese Integration ermöglicht die Deklaration von Constraints auf Komponenten von Benutzerschnittstellen, wie sie häufig in Ansätzen zur Generierung von Benutzerschnittstellen ohne Transformationen schon verwendet wurden.

Die neuen Konzepte werden Bestandteil der neuen Transformationssprache namens "Solverational". Im Rahmen der Implementierung wird eine Architektur und ein Interpreter vorgestellt. Um Transformationen mit bewährten Vorgehensmodellen entwickeln zu können, werden mehrere Schritte vorgestellt, um die diese erweitert werden können. Zur möglichst anschaulichen Demonstration der Flexibilität der Transformationssprache wird eine Menge von Transformationen deklariert, die beispielhaft verschiedene sog. "Strategien" zur Erzeugung von Modellen von Benutzerschnittstellen implementieren. Diese beschränken sich nicht auf grafische Benutzerschnittstellen, sondern zeigen die breite Anwendbarkeit der Konzepte.

Zur Evaluation der Transformationssprache werden verschiedene Untersuchungen durchgeführt, die die Qualität von "Solverational" messen. Die Ergebnisse der Untersuchungen zeigen, dass die neuen Sprachkonstrukte Transformationssprachen zur Transformation von Modellen von Benutzerschnittstellen verbessern können.

Abstract

User interface software technology faces a steadily increasing diversification of computing devices. User interfaces should be able to adapt to these new devices to achieve maximum usability. Model-driven software development and especially model-to-model transformation provides a solution. Used for models of user interfaces, model-to-model transformations derive device-specific models of various user interfaces from models which abstract from device-specific properties. To be displayed, device-specific models can then be interpreted or transformed into source-code.

However, generating a large number of device-specific models implies that many model-to-model transformations must be available. Usually the model-to-model transformations are being written by hand. Enormous efforts must be invested for this many transformations as a result. To reduce complexity by facilitating programming of model-to-model transformations declarative languages can be used.

We investigate which concepts and language constructs should be integrated to achieve model-to-model transformation languages "well" suited for programming transformations. A key aspect are graphical user interfaces which still dominate the market. To tackle the problem we propose to integrate optimization and constraint solving into a standard model transformation language. Using this approach constraints can be used to generate user interface models.

The newly introduced language contructs have been integrated into the new model-to-model transformation language called "Solverational". As a proof-of-concept we present an interpreter as well as a set of transformations written in the language. However, the approach is not limited to graphical user interfaces and we provide some transformations used for other purposes.

We evaluate our concepts using a number of different analysis which evaluate the "quality" of the Solverational language. We compare Solverational with other approaches. Results show that Solverational is rather well suited for model-to-model transformation of user interface models.

Danksagung

Mit Abgabe der Doktorarbeit möchte ich mich bei einer großen Anzahl von Leuten bedanken. Die hier aufgeführten Personen und -gruppen stellen aus Platzgründen nur eine Teilmenge dar.

Ein besonderer Dank gilt meinem Doktorvater, Herrn Prof. Dr. Max Mühlhäuser, der nicht nur durch die Finanzierung der Dissertation diese überhaupt möglich machte. Auch die inspirierende Atmosphäre, die er in der Telekooperation geschaffen hat und das Ziel Persönlichkeiten auszubilden sind meiner Erfahrung nach von unschätzbaren Wert für Dissertation und weiteren Lebensweg. An dieser Stelle möchte ich mich auch bei Herrn Prof. Dr. Heiko Krumm für die Übernahme des Zweitgutachtens bedanken.

Besonders bedanken für Unterstützung und Entbehrungen muss ich mich bei meiner Familie und meinen Freunden. Stefanie, meine geliebte Frau, hat nicht wenige Stunden auf mich verzichtet um mir die Durchführung der Arbeit zu ermöglichen. In ähnlicher Weise dankbar bin ich auch meinen Eltern und weiteren Verwandten. Während der Dissertation kam auch Valentin Petter, mein "kleiner Großer" zur Welt, der manchmal auf Spielen mit Papa verzichtete und dem ich für ein extra an Motivation dankbar bin. Viele Freunde haben die Arbeit Korrektur gelesen, dafür sei allen gedankt.

Allen ehemaligen Kollegen der Telekooperation möchte ich für ihre Inspiration und Unterstützung danken. Meinen ehemaligem Kollegen Alexander Behring möchte ich als Kollege und Freund besonders erwähnen, ohne dessen Zusammenarbeit die vorliegende Dissertation nicht halb so viel Spaß gemacht hätte.

Schlussendlich möchte ich auch den Prüfern, Zuhörern und Lesern danken.

Ölbronn-Dürrn, den 17.12.2011

J. J.

Inhaltsverzeichnis

Abbildungsverzeichnis **11**

Tabellenverzeichnis **13**

Quelltextverzeichnis **15**

1 Einleitung **17**
1.1 Modelle und Transformationen 18
1.2 Transformationen und Benutzerschnittstellen 20
1.3 Forschungsfrage 22
1.4 Struktur dieser Arbeit 23

2 Verwandte Arbeiten **25**
2.1 Transformationen und Constraints 27
2.1.1 Fokus: Transformationen 29
2.1.2 Fokus: Constraints 33
2.2 Transformationen und Benutzerschnittstellen 36
2.2.1 Fokus: Graphentransformation und Layout 36
2.2.2 Fokus: Benutzerschnittstellen 40
2.3 Transformationen und Optimierung 45
2.3.1 Fokus: Modell- oder Graphentransformation 45
2.3.2 Fokus: Optimierung 47
2.4 Benutzerschnittstellen und Constraints 48
2.4.1 Fokus: Benutzerschnittstellen 49
2.4.2 Fokus: Constraints 50
2.5 Benutzerschnittstellen und Optimierung 52
2.5.1 Fokus: Benutzerschnittstellen 53
2.5.2 Fokus: Optimierung 56
2.6 Zusammenfassung 56
2.6.1 Fazit 58

3 Konzepte **61**
3.1 Transformationssprache . . . 63
3.1.1 Herleitung von Konzepten . . . 64
3.1.2 Constraint-Solving . . . 65
3.1.3 Constraint-Solving in Modell-zu-Modell-Transformationen . 68
3.1.4 Optimierung . . . 75
3.1.5 Derivatprobleme . . . 82
3.2 Transformation von Benutzerschnittstellen . . . 86
3.2.1 Benutzbarkeit . . . 87
3.2.2 Meta-Modelle . . . 89
3.2.3 Constraint-Solving und qualitative Regeln . . . 91
3.2.4 Optimierung mit quantitativen Regeln . . . 93
3.2.5 Scheduling bei Modellen von Benutzerschnittstellen . . . 95
3.2.6 Strategien zur Generierung von Benutzerschnittstellen . . . 96
3.3 Zusammenfassung . . . 97

4 Transformationssprache **99**
4.1 Transformationssprache . . . 99
4.1.1 QVT Relations . . . 100
4.1.2 Solverational . . . 103
4.2 Architektur zur Interpretation . . . 113
4.2.1 Vorteile der Architektur . . . 115
4.2.2 Nachteile der Architektur . . . 116
4.2.3 Vorteile der Abbildung auf CLP . . . 116
4.2.4 Nachteile der Abbildung auf CLP . . . 117
4.3 Implementierung . . . 117
4.3.1 Beschränkungen der Implementierung . . . 119
4.3.2 Fazit . . . 120
4.4 Integration in Vorgehensmodelle . . . 120
4.4.1 Entwicklungsphase . . . 121
4.4.2 Schritte . . . 123
4.4.3 Iterative Methodik und Beispiel . . . 124
4.5 Transformationen . . . 126
4.5.1 Hardware-Transformation . . . 127
4.5.2 Anoto-Transformation . . . 132
4.5.3 Panel-Transformation . . . 133
4.5.4 AUI2CUI-Transformation . . . 139
4.5.5 Performance-Transformation . . . 140
4.6 Zusammenfassung . . . 144

5 Evaluation **147**
5.1 Methodik zur Evaluation . . . 150
5.1.1 J. Howatts . . . 150
5.1.2 D. Olsen . . . 150
5.1.3 Kombination . . . 151
5.1.4 Ausgereift . . . 152
5.1.5 Benutzbar . . . 153
5.1.6 Domänen-gerecht . . . 154
5.1.7 Komplexes Entscheidungskriterium . . . 155
5.2 Verständlichkeit . . . 155
5.2.1 Hypothese . . . 156
5.2.2 Studiendesign . . . 157
5.2.3 Datenanalyse und Präsentation . . . 159
5.2.4 Bedeutendste Faktoren, die die Validität einschränken können 163
5.2.5 Interpretation . . . 164
5.3 Schreibbarkeit . . . 165
5.3.1 Hypothese . . . 166
5.3.2 Studiendesign . . . 166
5.3.3 Datenanalyse und Präsentation . . . 166
5.3.4 Bedeutendste Faktoren, die die Validität einschränken können 168
5.3.5 Interpretation . . . 168
5.4 Allgemeingültigkeit der Ansätze . . . 169
5.4.1 Untersuchte Transformationen . . . 169
5.4.2 Datenanalyse und Präsentation . . . 173
5.4.3 Bedeutendste Faktoren, die die Validität einschränken können 176
5.4.4 Interpretation und Fazit . . . 177
5.5 Ausdrucksmächtigkeit SGG-Solverational . . . 177
5.5.1 Räumliche Relationen . . . 178
5.5.2 Topology . . . 178
5.5.3 Direction . . . 180
5.5.4 Distance . . . 181
5.5.5 Alignment . . . 182
5.5.6 Beispiel zur Nutzung der Constraints . . . 183
5.5.7 Interpretation . . . 184
5.6 Effizienz des Constraint-Solving Ansatzes . . . 185
5.6.1 Datenanalyse und Präsentation . . . 186
5.6.2 Interpretation . . . 186
5.6.3 Fazit . . . 188
5.7 Abdeckung von Regeln aus dem Apple Style Guide . . . 188
5.7.1 Bedeutendste Faktoren, die die Validität einschränken können 190

5.7.2 Interpretation . . . 191
5.7.3 Fazit . . . 192
5.8 Ergebnisse und Entscheidungskriterium . . . 192
5.8.1 Ausgereift . . . 192
5.8.2 Benutzbar . . . 194
5.8.3 Domänen-gerecht . . . 197
5.8.4 Komplexes Entscheidungskriterium . . . 199
5.9 Zusammenfassung . . . 201

6 Zusammenfassung und Ausblick 203
6.1 Solverational . . . 204
6.1.1 Konzepte . . . 204
6.1.2 Umsetzung . . . 205
6.2 Ergebnisse der Evaluation . . . 206
6.2.1 Ausgereift . . . 206
6.2.2 Benutzbar . . . 207
6.2.3 Domänen-gerecht . . . 207
6.2.4 Formel . . . 208
6.3 Ausblick . . . 208
6.3.1 Weiterentwicklung der Theorie . . . 208
6.3.2 Implementierung . . . 209
6.3.3 Anwendung auf Benutzerschnittstellen und Evaluation . . . 210
6.3.4 Evaluation . . . 210

A Ergebnisse von Transformationen 211
A.1 Ergebnisse der Panel-Transformation . . . 211
A.1.1 Möglichst-Links-Strategie . . . 212
A.1.2 Möglichst-Oben-Strategie . . . 214
A.1.3 Relevanz-Strategie . . . 215
A.1.4 2D-Fitt's-Law-Strategie . . . 217
A.1.5 Behring-Strategie . . . 218

B Graphen zu Studienergebnissen 219
B.1 Approximierte Verteilungen . . . 220
B.1.1 Modelle . . . 220
B.1.2 Optimierung . . . 222
B.2 Boxplots der Antwortzeiten . . . 222

C Abkürzungsverzeichnis 227

D Literaturverzeichnis 229

Abbildungsverzeichnis

1.1 Aufbau Dissertation 24

2.1 Klassifikation und resultierender Aufbau des Kapitels. 27

3.1 Aufbau Dissertation, Kapitel 3 hervorgehoben 62
3.2 Beispielhaftes, abstraktes Meta-Modell 90
3.3 Beispielhaftes, konretes Meta-Modell 91

4.1 Aufbau Dissertation, Kapitel 4 hervorgehoben 100
4.2 Abstrakte Syntax, Constraints 105
4.3 Grafische Syntax, Constraints 107
4.4 Abstrakte Syntax, Optimierung und Derivatprobleme 109
4.5 Grafische Syntax, Optimierung. 111
4.6 Architektur zur Interpretation von Solverational 114
4.7 Architektur der Implementierung 118
4.8 Methodik zur Entwicklung von Transformationen, [89] angepasst . . 125
4.9 Spezialhardware zur Eingabe 126
4.10 Beispiel Hardware-Transformation 127
4.11 Abstraktes Modell, dargestellt in Mapache-Umgebung 129
4.12 Konkretes Modell, dargestellt in Mapache-Umgebung 130
4.13 Beispiel Anoto-Transformation 132

5.1 Aufbau Dissertation, Kapitel 5 hervorgehoben 148
5.2 Bereich Modelle: Mittelwerte, Standardabweichung und Konfidenzintervalle der Antwortzeiten 160
5.3 Bereich Constraints: Mittelwerte, Standardabweichung und Konfidenzintervalle der Antwortzeiten 161
5.4 Bereich Optimierung: Mittelwerte, Standardabweichung und Konfidenzintervalle der Antwortzeiten 162
5.5 Scheduling Transformation: Beispiel Schreibwarenfabrik 170
5.6 Service Composition: Workflow ökologische Herstellung 172
5.7 Punkte rechteckiger Objekte 178

5.8 Punkte in Geradensegmenten 180
5.9 Laufzeiten von Transformationsmaschinen und Constraint-Solving. . 185

A.1 Ergebnis Möglichst-Links-Strategie, 1500*1000 212
A.2 Ergebnis Möglichst-Links-Strategie, 1500*800 213
A.3 Ergebnis Möglichst-Oben-Strategie, 1500*1000 214
A.4 Ergebnis Möglichst-Oben-Strategie, 1100*1000 214
A.5 Ergebnis Relevanz-Strategie, 1500*1000 215
A.6 Ergebnis Relevanz-Strategie, 1000*1000 216
A.7 Ergebnis Fitt's-Law-Strategie, 1200*700 217
A.8 Ergebnis Fitt's-Law-Strategie, 1600*500 217
A.9 Ergebnis Behring-Strategie, 1500*1000, Gewichte=1 218
A.10 Ergebnis Behring-Strategie, 1500*1000, Gewicht unten=20 218

B.1 Approximierte Verteilung der Antwortzeiten der Probanden für unterschiedliche Notationen für Modelle 220
B.2 Approximierte Verteilung der Antwortzeiten der Probanden für unterschiedliche Notationen für Constraints 221
B.3 Approximierte Verteilung der Antwortzeiten der Probanden für unterschiedliche Notationen für Zielfunktionen 222
B.4 Boxplot der Antwortzeiten für unterschiedliche Notationen von Modellen 223
B.5 Boxplot der Antwortzeiten für unterschiedliche Notationen von Constraints 224
B.6 Boxplot der Antwortzeiten für unterschiedliche Notationen von Zielfunktionen 225

Tabellenverzeichnis

2.1 Kriterien zur Klassifikation verwandter Arbeiten 28
2.2 Klassifikation verwandter Arbeiten, Transformationen und Constraints 37
2.3 Klassifikation verwandter Arbeiten, Transformationen und Benutzerschnittstellen 44
2.4 Klassifikation verwandter Arbeiten, Transformationen und Optimierung 48
2.5 Klassifikation verwandter Arbeiten, Benutzerschnittstellen und Constraints 52
2.6 Klassifikation verwandter Arbeiten, Benutzerschnittstellen und Optimierung 56
2.7 Klassifikation verwandter Arbeiten, Übersicht 57
2.8 Anforderungen aus verwandten Arbeiten 60

3.1 Zusammenfassung der mathematischen Symbole in Kapitel 3 65

5.1 Kategorien für Kriterien der Ansätze zur Evaluation 152
5.2 p-Werte der Tests: Notation von Modellen 159
5.3 p-Werte der Tests der paarweisen Vergleiche 161
5.4 p-Werte der Tests: Notation von Zielfunktionen 162
5.5 Ergebnisse Schreibbarkeit 167
5.6 Scheduling Transformation: Eigenschaften der Maschinen 170
5.7 Service Composition: Eigenschaften der Tasks 171
5.8 Ergebnisse: Allgemeingültigkeit der Sprachen 174
5.9 Ergebnisse: Aufgaben, die in Transformationen adressiert werden können 177
5.10 Codes die in der Auswertung verwendet werden 189
5.11 Ergebnisse der Auswertung von Regeln aus [10] 190
5.12 Berechnete Konstanten für das komplexe Entscheidungskriterium . 200

Quelltextverzeichnis

3.1 Algorithmus Constraint-relationale-Transformationen 72
3.2 Algorithmus Derivatprobleme 84
4.1 Solverational-Constraints um Komponenten untereinander anzuordnen 131
4.2 Solverational-Constraints um Größe festzusetzen 133
4.3 Solverational-Derivatproblem um die Ausrichtung von Panels festzulegen 135
4.4 Solverational-Derivatproblem für die Transformation von Listen . . 141
4.5 Transformation für Benchmarks - Variante 1 142
4.6 Transformation für Benchmarks - Variante 2 143
4.7 Transformation für Benchmarks - Variante 3 144
5.1 Scheduling Transformation: Details des Solverational Quellcodes . . 175
5.2 Service Composition: Details des Solverational Quellcodes 176

Kapitel 1

Einleitung

"Ubiquitous Computing" ist eine Vision über die Zukunft der Computertechnik (s. [164]), deren Realisierung unter anderem viele verschiedene, leicht benutzbare Geräte erfordert. Auch wenn die Vision heute erst teilweise Realität ist [119], gibt es bereits "Vorboten für diese Entwicklung" [101] - Computer werden kleiner, SmartPhones und NetBooks diversifizieren die Auswahl an Geräten. Dabei müssen auf Grund der vielen Geräte auch entsprechend viele Benutzerschnittstellen für eine einzige Anwendung entwickelt werden, die auf mehreren unterschiedlichen Geräten verwendet werden soll. Zusätzlich sollten zur Verbesserung der Benutzbarkeit weitere Benutzerschnittstellen entwickelt werden, damit die Benutzerschnittstellen der Anwendung an Benutzer oder deren Wünsche, sowie verschiedene Umgebungen (**"Kontexte"**) angepasst sind. Der Ansatz, jede Benutzerschnittstelle einzeln zu entwickeln, steht dabei einer effizienten Entwicklung entgegen [105]. Deshalb wird schon länger modell-getriebene Softwareentwicklung von Benutzerschnittellen gefordert (z.B. [121, 93, 149]).

Bei einem modell-getriebenem Ansatz werden Modelle zur Beschreibung der Benutzerschnittstellen genutzt, die von den Unterschieden der Benutzerschnittstellen abstrahieren. Diese sogenannten **"abstrakten Modelle"** können vom Entwickler der Modelle, dem sog. **"Modellierer"**, entweder textuell oder grafisch mit einem entsprechendem Editor notiert werden. Sie werden dann in ein anderes, weniger abstraktes Modell überführt, das **"konkrete Modell"**, welches zusätzlich Informationen enthält, die zum Anzeigen der Benutzerschnittstelle in einem bestimmten Kontext benötigt werden. Um eine möglichst große Anzahl von Kontexten abzudecken, werden viele solche konkrete Modelle benötigt. Viele Modelle sind daher besonders für Anwendungen im Bereich des Ubiquitous Computing nötig.

Die manuelle Erstellung (**"Modellierung"**) einer Vielzahl von Modellen ist sehr aufwändig. Aus diesem Grund werden Programme entwickelt, die den Prozess der Erzeugung von Modellen automatisch ausführen. Diese Programme erzeugen aus einem oder mehreren Modellen ein neues oder verändertes Modell. Die-

se sog. **"Modell-zu-Modell-Transformationen"** erleichtern z.B. die Erstellung von Benutzerschnittstellen, indem sie die konkreten Modelle automatisch erzeugen: wenn die konkreten Modelle durch Modell-zu-Modell-Transformationen aus einem abstrakten Modell erzeugt werden, können die Modelle für viele verschiedene Applikationen mit wenig Aufwand erzeugt werden.

Werden viele verschiedene konkrete Modelle benötigt, um Benutzerschnittstellen optimal anzupassen, so werden auch viele Modell-zu-Modell-Transformationen benötigt (oder entsprechend lange Transformationen die alle Möglichkeiten und Kontexte abdecken). Werden die Transformationen mit klassischen Programmiersprachen entwickelt, müssen die Entwickler entsprechende Funktionen nutzen um auf Informationen in den Modellen zuzugreifen (z.B. Attribute, Typen, Klassen, Assoziationen, Werte von Attributen). Dadurch wird einerseits der Aufwand zur Entwicklung der Transformationen erhöht. Andererseits wird durch den längeren Quellcode die Verständlichkeit der Transformationen verringert, was den Aufwand zur Entwicklung der Transformationen weiter erhöhen kann.

Um den Aufwand zu vermindern kann zur Spezifikation und Programmierung der Transformationen eine **"Modell-zu-Modell-Transformationssprache"** verwendet werden. Solche Sprachen haben Sprachkonstrukte um auf die in Modellen notierten Informationen zugreifen zu können. Dadurch wird der Entwicklungsaufwand von Transformationen verringert. Dieser Vorteil von Modell-zu-Modell-Transformationssprachen kann zur Programmierung von Transformationen zur Transformation von Modellen von Benutzerschnittstellen im Bereich des Ubiquitous Computing genutzt werden. In dieser Arbeit werden daher Modell-zu-Modell-Transformationssprachen zur Entwicklung von Transformationen für Benutzerschnittstellen untersucht. Aufbauend auf neuen Konzepten zur Modelltransformation entsteht eine neue Transformationssprache. Die Konzepte werden mit Hilfe dieser Sprache im Kontext von grafischen Benutzerschnittstellen umgesetzt und evaluiert.

Grundsatz 1. *Der Ansatz soll die Transformation abstrakter Modelle von Benutzerschnittstellen in konkrete Modelle unterstützen.*

1.1 Modelle, Meta-Modelle und Modell-zu-Modell-Transformationen

In diesem Abschnitt werden einige allgemeine Begriffe aus der modell-getriebenen Softwareentwicklung eingeführt, die für das Verständnis der folgenden Kapitel notwendig sind.

Ein allgemein akzeptiertes Verständnis von Modellen im Software-Engineering entstand erst durch die von der Industrie voran getriebene Entwicklung der weithin

bekannten Unified Modeling Language (UML) [114]. Eine klare Definition für den Begriff "Modell" kann laut J. M. Favre aber nicht einfach gegeben werden [42]. Das aus der Modelltheorie [151] abgeleitete allgemeine Verständnis von Modellen von J. Ludewig [97] wird mit den Ansätzen von UML kombiniert und sei hier informell wie folgt definiert:

Ein **Modell** ist ein Graph, der zur Beschreibung eines Systems dient, einen Namen und eine Pragmatik hat. Er besteht aus Modellelementen (Knoten) und Assoziationen (Kanten). Assoziationen können verschiedene Beziehungen ausdrücken und haben eine Beschriftung (engl. "labelled edge"). Modellelemente können einen Typ, eine Menge von Attributen und eine Menge von Operationen haben.

Zur formalen Definition von Modellen können Meta-Modelle verwendet werden [41]. Diese können – verglichen mit textlich notierten Sprachen – als Grammatik für Modelle verstanden werden. Dazu wird ein Modell definiert, das die möglichen Typen definiert, einschließlich deren Mengen von Attributen, Operationen und Assoziationen, sog. **Klassen**. Ein **Meta-Modell** ist dann ein Modell aller Modelle, die seiner Beschreibung entsprechen, sogenannter "Instanzen des Meta-Modells" (oder auch "konformer Modelle") [42]. Um Meta-Modelle zu definieren, werden diese wiederum in formalen Sprachen notiert, in sog. Meta-Meta-Modellen. Die bekannteste ist Meta Object Facility (MOF) [113], zu der auch das in dieser Arbeit verwendete EMF/Ecore kompatibel ist [154]. Die drei Ebenen Modell, Meta-Modell und Meta-Meta-Modell wurden in der Model Driven Architecture (MDA) in einer Hierarchie geordnet. Dies ist der de-facto-Standard für modell-getriebene Softwareentwicklung mit Hilfe von Meta-Modellen [118], der auch als Basis für die vorliegende Arbeit genutzt wird.

Eine **Modell-zu-Modell-Transformation** ist eine Abbildung, die Modelle auf Modelle abbildet [118]. Diese Abbildung kann in Form eines Programms in einer nahezu beliebigen Sprache notiert werden. Programmiersprachen die speziell zur Entwicklung von Modell-zu-Modell-Transformationen dienen, heissen Modell-zu-Modell-Transformationssprachen. Modell-zu-Modell-Transformationssprachen bieten sich auf Grund der Nähe zur Domäne besonders an (s.o.; eine Klassifikation von Modell-zu-Modell-Transformationssprachen findet sich in [35]). Bei einem sog. "model-type-mapping" kann eine Abbildung von Modellelementen und Assoziationen der Meta-Modelle der Ausgangsmodelle in Modellelemente und Assoziationen der Meta-Modelle der Zielmodelle definiert werden [118]. Werden Meta-Modelle verwendet, können Transformationen unabhängig von den zu transformierenden Modellen notiert werden. Sie können damit auch für zukünftige Modelle verwendet werden.

Modell-zu-Modell-Transformationssprachen können deklarativ sein. Zustände werden in **deklarativen Sprachen** nicht explizit notiert. Zwischen deklarativen Sprachen bestehen teils erhebliche Unterschiede bzgl. der Frage inwieweit die Spra-

chen tatsächlich von Algorithmen und Zuständen abstrahieren. Einerseits sind nicht immer alle Konstrukte deklarativ (z.B. in der logischen Sprache Prolog, s. [52]) und andererseits ist manchmal ein Algorithmus zur Implementierung eines Programms erkennbar: so muss in Prolog zumindest eine Art Algorithmus zur Lösung eines Problems angegeben werden – zum Aufzählen einer Liste muss beispielsweise eine Rekursion definiert werden. Andere Programmierparadigmen, wie z.B. Constraint-Programmierung, bei dem das komplette Programm in Constraints zwischen Variablen notiert wird, benötigen dies nicht. Hier wird es dem Computer überlassen, eine Lösung zu finden, so dass alle Constraints erfüllt werden ("halten"). Als Folge solch eklatanter Unterschiede zwischen deklarativen Sprachen haben z.B. A. Gerber u.a. Prolog als nicht ausreichend deklarativ verworfen und stattdessen die Sprache Mercury verwendet [52]. Um diese Unterschiede ausdrücken zu können sei **Deklarativität** ein Begriff, der ausdrücken soll, wie stark eine deklarative Sprache von Algorithmen und Zuständen zu abstrahieren gestattet ("hohe" oder "niedrige Deklarativität"). Für die Entwicklung von Modell-zu-Modell-Transformationen ist es oft erwünscht, dass Sprachen möglichst deklarativ sind, da sie dann oft leichter zu verstehen sind (wie in der Modellierung allgemein angestrebt wird). Die Transformationssprache, die in dieser Arbeit zur Transformation von Modellen von Benutzerschnittstellen verwendet wird, sollte daher möglichst deklarativ sein.

Modell-zu-Modell-Transformationen können endogen oder exogen sein. Eine **endogene Transformation** verwendet für Ausgangs- und Zielmodelle dasselbe Meta-Modell. D.h. es wird nicht zwischen zwei verschiedenen Sprachen transformiert, sondern innerhalb einer Sprache. Eine **exogene Transformation** hingegen verwendet für Ausgangs- und Zielmodelle unterschiedliche Meta-Modelle.

1.2 Modell-zu-Modell-Transformationen und Modelle von Benutzerschnittstellen

Ein Modell zur Beschreibung einer Benutzerschnittstelle enthält Modellelemente, die Komponenten der Benutzerschnittstelle repräsentieren. Diese Modellelemente werden ebenfalls als Komponenten bezeichnet, da eine direkte Abhängigkeit zu Benutzerschnittstellen besteht. Es werden "abstrakte" und "konkrete" Komponenten unterschieden. **Konkrete Komponenten** können ohne weitere Interpretation dargestellt werden ("darstellen" ist nicht auf visuelle Modalitäten beschränkt). **Abstrakte Komponenten** müssen hingegen in einem bestimmten Kontext interpretiert werden bevor sie angezeigt werden können. **Abstrakte Modelle** enthalten abstrakte Komponenten und abstrahieren daher von Unterschieden mit Hilfe dieses Modells ableitbarer Benutzerschnittstellen. Hingegen enthalten **konkrete**

Modelle alle notwendigen Informationen zum Anzeigen der Benutzerschnittstelle ohne dass eine Interpretation notwendig ist.

Zum Beispiel kann ein abstraktes Modell unter anderem für verschieden große Bildschirme und eine Benutzerschnittstelle, die durch Sprache gesteuert werden kann, gleichermaßen verwendet werden. Deshalb werden abstrakte Komponenten genutzt, deren Ziel es ist, von den konkreten Komponenten grafischer und per Sprache gesteuerter Benutzerschnittstellen zu abstrahieren.

Die Interpretation von abstrakten Modellen kann mit Hilfe von Modell-zu-Modell-Transformationen geschehen. Modell-zu-Modell-Transformationen zur Transformation von Modellen von Benutzerschnittstellen überführen daher meist von einem abstrakten Modell in mehrere konkrete (s. [94], Kapitel 2).

Da das abstrakte Modell abstrakte Komponenten enthält, muss auch das Meta-Modell zu dem es konform ist, die abstrakten Komponenten in Form von Klassen zur Verfügung stellen. Hingegen enthalten die konkreten Modelle Komponenten, die in anderen Meta-Modellen zur Verfügung gestellt werden. Daher handelt es sich bei den Meta-Modellen für abstrakte Modelle um andere Meta-Modelle als die für konkrete Modelle. Eine Transformationssprache, die Transformationen von Modellen von Benutzerschnittstellen unterstützen soll, muss daher exogene Transformationen unterstützen.

Grundsatz 2. *Es soll eine deklarative Modell-zu-Modelltransformationssprache verwendet werden. Die Transformationssprache soll exogene Transformationen unterstützen.*

Erzeugte Zielmodelle entsprechen nur teilweise den Anforderungen, die Modellierer an ihre Modelle stellen. Wenn Modellierer dies als notwendig erachten, verändern sie die Modelle so lange, bis sie den Anforderungen genügen. Im Falle der Transformation von Modellen von Benutzerschnittstellen werden die konkreten Modelle so lange verändert, bis die generierten Benutzerschnittstellen die von den Modellierern (oder den Anwendern der Applikation oder HCI-Experten) geforderte Benutzbarkeit erreicht haben. Werden grafische Benutzerschnittstellen entwickelt, muss insbesondere auch das Layout den Anforderungen gerecht werden. Modellierer passen von der Transformation erzeugte Modelle per Hand an, deshalb ist die Effizienz der Entwicklung höher, wenn weniger Änderungen vorgenommen werden müssen. Dies ist der Fall, wenn die Modelle dem gewünschten Layout bereits bestmöglich entsprechen, da dann weniger Änderungen vorgenommen werden müssen. Da die Modelle durch Transformationen erzeugt werden, wird in dieser Arbeit versucht, Techniken zur Generierung von Benutzerschnittstellen in Transformationen zu integrieren, die die Voraussetzung schaffen, dass Layouts bereits bestmöglich in Transformationen erzeugt werden können[1]. Um den Grund-

[1]Die Erzeugung von Modellen von grafischen Benutzerschnittstellen ist nur ein Beispiel. In Kapitel 4 werden weitere Modalitäten betrachtet.

satz einer deklarativen Transformationssprache umzusetzen, sollten die Techniken zur Generierung von Benutzerschnittstellen, die in die Transformationen integriert werden sollen, möglichst auch deklarativ sein.

Bekannte deklarative Programmierparadigmen zur Generierung von Benutzerschnittstellen sind Constraint-Programmierung und Optimierung (z.B. [71, 44, 48]).

In der Constraint-Programmierung wird ein Programm durch eine Menge von Variablen und eine Menge von Constraints notiert. Ein **Constraint** setzt eine Teilmenge der Variablen mittels eines Komparators in Relation zueinander, wobei die Komperatoren $<$, $\leq$, $=$, $>$, $\geq$ und $\neq$ genutzt werden können. **Constraint-Programmierung** ist ein Programmierparadigma, bei dem ein Problem nicht durch einen Algorithmus beschrieben wird, sondern durch ein System von solchen Constraints. Diese schränken den Definitionsbereich einer ursprünglichen Menge so weit ein, dass die Lösungen für das System von Constraints die Lösungen für das gegebene Problem sind. Dies ist ein sog. "Constraint-Solving-Problem". Wie viele Arbeiten im Bereich der Mensch-Maschine-Interaktion anschaulich demonstrieren [71], handelt es sich dabei um einen in der Literatur sehr weit verbreiteten Ansatz, um Benutzerschnittstellen zu generieren. Viele, das Layout bestimmende Eigenschaften, wie Position und Größe, werden in Modellen klassischerweise als Attribute definiert. Um Layouts in Transformationen generieren zu können, soll in Transformationen Constraint-Programmierung bzw. Constraints-Solving insbesondere auf Attributen des Zielmodells ausgeführt werden können.

Grundsatz 3. *Die Transformationssprache soll Constraints-Solving auf Attributen des Zielmodells ermöglichen.*

Es wäre von Vorteil, wenn Transformationen Metriken zur quantitativen Messung von Benutzbarkeit nutzen könnten, um Benutzerschnittstellen zu erzeugen, die bzgl. Metriken "optimal" sind. **Optimierung** ermöglicht es, eine Funktion zu minimieren (oder maximieren) und so bzgl. der Funktion ein "optimales" Ergebnis zu erhalten. Da Metriken als Funktionen verstanden werden können, können Benutzerschnittstellen bzgl. dieser Metriken optimal erzeugt werden (z.B. [48, 44, 146]). Es bietet sich daher an, in der Transformationssprache die Deklaration von Funktionen zur Optimierung, sog. **"Zielfunktionen"**, zu ermöglichen.

Grundsatz 4. *Die Transformationssprache soll die Definition von Zielfunktionen unterstützen.*

1.3 Forschungsfrage

Welche Sprachkonstrukte muss eine "gute" Modell-zu-Modell-Transformationssprache zur Transformation von Modellen von Be-

nutzerschnittstellen haben?

"Gut" ist dabei ein relativer Begriff, der sich erst aus dem Vergleich von verschiedenen Modell-zu-Modell-Transformationssprachen ergibt und nach der hier vertretenen Ansicht aus zwei Teilen besteht:

1. In diesem Kapitel wurde festgestellt, dass Modell-zu-Modell-Transformationssprachen zur Transformation von Modellen von Benutzerschnittstellen auf der Basis von vier Grundsätzen entwickelt werden sollten:

   (a) Der Ansatz muss die Transformation von Modellen von Benutzerschnittstellen unterstützen.

   (b) Die Modell-zu-Modelltransformationssprache sollte deklarativ sein.

   (c) Die Transformationssprache sollte Constraint-Solving auf Attributen des Zielmodells ermöglichen.

   (d) Die Transformationssprache sollte die Definition von Zielfunktionen unterstützen.

   Um eine "gute" Modell-zu-Modell-Transformationssprache zur Transformation von Modellen von Benutzerschnittstellen zu finden, kann daher untersucht werden, inwieweit alle vier Grundsätze bei der Entwicklung der Sprache berücksichtigt wurden. In Kapitel 2 werden verschiedene Modell-zu-Modell-Transformationssprachen auf die Einhaltung dieser Grundsätze hin untersucht.

2. Darüber hinaus sollte eine "gute" Modell-zu-Modell-Transformationssprache zur Transformation von Modellen von Benutzerschnittstellen auf die Gegebenheiten von Projekten zugeschnitten sein, in denen sie eingesetzt wird. Sie soll in gewisser Hinsicht "gut" zu einem Projekt passen. Dabei gilt es zu beachten, dass verschiedene Projekte unterschiedliche Anforderungen an Modell-zu-Modell-Transformationssprachen stellen können. Im Rahmen der Evaluation (s. Kapitel 5) wird eine Metrik erarbeitet, die es ermöglicht, in Abhängigkeit verschiedener, gewichteter Eigenschaften zwischen drei verschiedenen Sprachen auszuwählen. Damit wird "gut" in Abhängigkeit von Projekten definiert.

1.4 Struktur dieser Arbeit

Abbildung 1.1 gibt einen Überblick über den Aufbau dieser Arbeit. Während in diesem Kapitel Grundsätze für mögliche Transformationssprachen entwickelt wurden, wird in Kapitel 2 an Hand der existierenden Literatur geprüft, ob es bereits

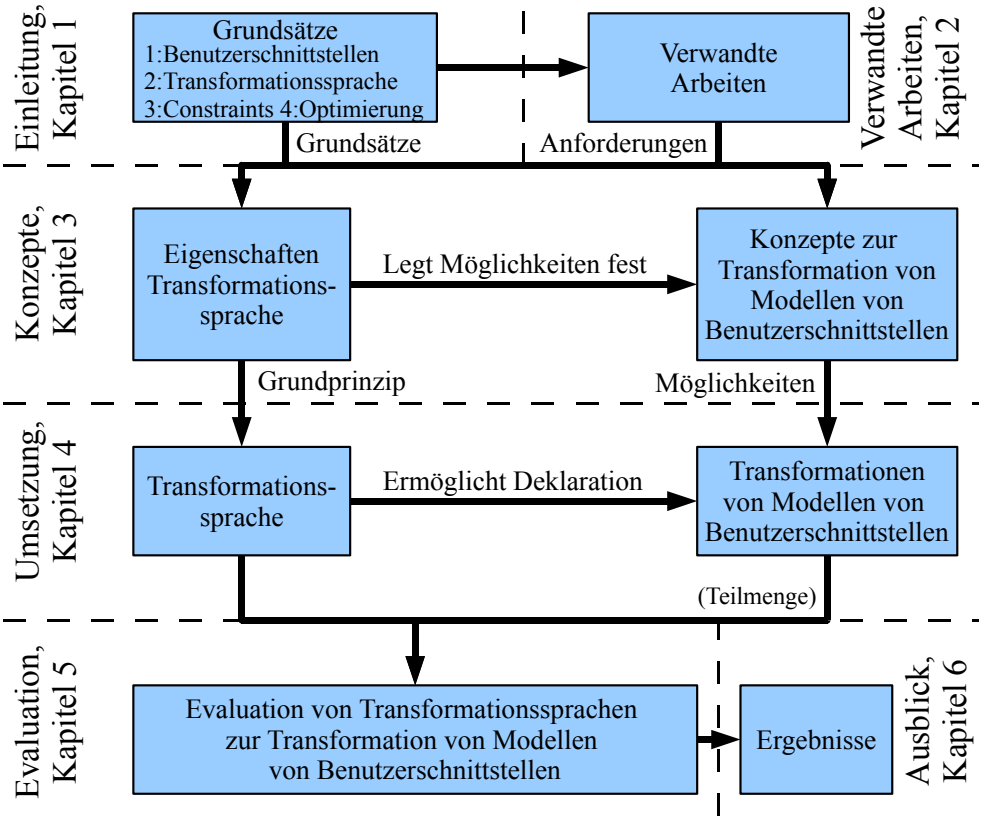


Abbildung 1.1: Aufbau Dissertation

existierende Ansätze gibt, die die Grundsätze berücksichtigen. Kapitel 3 gibt einen Überblick über theoretische Grundlagen und Konzepte für eine neue Sprache. Kapitel 4 beschreibt eine Erweiterung eines existierenden Standards für Modell-zu-Modell-Transformationssprachen und definiert dabei eine neue Sprache sowie eine Menge von Modell-zu-Modelltransformationen für Modelle von Benutzerschnittstellen. Eine Evaluation der Sprache in Hinblick auf die Transformation von Modellen von Benutzerschnittstellen erfolgt in Kapitel 5. Kapitel 6 fasst die Ergebnisse zusammen und gibt einen Ausblick auf mögliche weitere Forschungsvorhaben.

Kapitel 2

Verwandte Arbeiten

Überblick

In diesem Kapitel werden verwandte Arbeiten und aktuelle Entwicklungen im Bereich Constraint-Solving, Optimierung, Modell-zu-Modell-Transformation und Modell-Transformation von Benutzerschnittstellen vorgestellt. Diese vier Forschungsgebiete sind so umfangreich, dass nur der Ausschnitt der eng verwandten Arbeiten betrachtet werden kann. Im letzten Kapitel wurde bereits erläutert, dass

1. die Transformation von Modellen von Benutzerschnittstellen,
2. eine deklarative Modell-zu-Modell-Transformationssprache,
3. die Unterstützung von Constraint-Solving,
4. und die Unterstützung von Optimierung

als Grundsätze zur Entwicklung von Transformationssprachen zur Transformation von Modellen von Benutzerschnittstellen identifiziert wurden. Für eine "gute" Transformationssprache zur Transformation von Modellen von Benutzerschnittstellen ist nach der in dieser Arbeit vertretenen Ansicht eine Transformationssprache erforderlich, die alle vier Grundsätze gleichzeitig berücksichtigt. Es werden daher verwandte Arbeiten untersucht, die möglichst viele Grundsätze berücksichtigen, mindestens aber zwei Grundsätzen (zumindest teilweise) genügen. Dazu werden die Grundsätze paarweise geordnet:

1. Transformationen und Constraints (Grundsätze 2 und 3, s. Abschnitt 2.1)
2. Transformationen und Benutzerschnittstellen (Grundsätze 1 und 2, s. Abschnitt 2.2)

3. Transformationen und Optimierung (Grundsätze 2 und 4, s. Abschnitt 2.3)
4. Benutzerschnittstellen und Constraints (Grundsätze 1 und 3, s. Abschnitt 2.4)
5. Benutzerschnittstellen und Optimierung (Grundsätze 1 und 4, s. Abschnitt 2.5)

Die Kombination von Optimierung und Constraints (Optimierung unter Nebenbedingungen) wird nicht betrachtet und fehlt daher in oben stehender Liste. Ansätze, die die Kombination von Optimierung und Constraints erreichen, ohne einen weiteren Grundsatz zu erfüllen, sind zu allgemein und daher nicht mehr als eng verwandt zu bezeichnen. Nicht vorgestellt werden deshalb die vielen bekannten Optimierungsansätze aus dem Forschungsgebiet des Operations Research (z. B. [130, 134, 45]). Diese besitzen zwar teilweise grafische Notationen, die in Editoren mit Benutzerschnittstellen notiert werden, aber sie sind weder Modell-zu-Modelltransformationen noch werden die grafischen Notationen als Modelle im engeren Sinne (Software Engineering) verwendet. Ebenfalls aus diesem Grund nicht vorgestellt werden Ansätze aus der Graphentheorie, die spezielle grafische Optimierungsprobleme lösen (z.B. "Travelling Salesman") und deren Definition von "Modell" nicht mit der hier verwendeten übereinstimmt.

Da nicht alle eng verwandten Arbeiten im Bereich Modelltransformation und Benutzerschnittstellen Ansätze mit deklarative Modell-zu-Modell-Transformationssprachen aufweisen, werden für die Betrachtung der verwandten Arbeiten auch einige Ansätze einbezogen, mit denen "Transformationen" flexibel definiert werden können.

Die paarweisen Kombinationen der Grundsätze können zusätzlich von zwei Blickwinkeln betrachtet werden, da die verwandten Ansätze meist einen Fokus auf einen Grundsatz legen und bei der Entwicklung des Ansatzes eventuell erreichte weitere Grundsätze weniger wichtig erschienen. Die Abschnitte sind deshalb in zwei kleinere Abschnitte ("Unterabschnitte") unterteilt, die den Fokus der in ihnen enthalten Ansätze widerspiegelt.

Zur besseren Orientierung innerhalb des Kapitels verdeutlicht Abbildung 2.1 die Klassifikation. Die Grundsätze werden durch Balken dargestellt. Paarweise Kombinationen von Grundsätzen, die in der Klassifikation abgebildet werden, sind durch gestrichelte Linien gekennzeichnet. Die Linien sind mit der Nummer des Abschnitts beschriftet, in denen die Kombination untersucht wird. Existieren Ansätze, die einen Fokus auf einen Grundsatz der Kombination haben, befindet sich am Ende der Linie auf dessen Seite ein Kreis. Der an der Linie notierte Abschnitt enthält dann einen entsprechenden Unterabschnitt zum jeweiligen Fokus.

Zur Bewertung der untersuchten Ansätze werden die Kriterien aus Tabelle 2.1 angewendet, die sich unmittelbar aus den Grundsätzen ergeben. Die Qualität wird

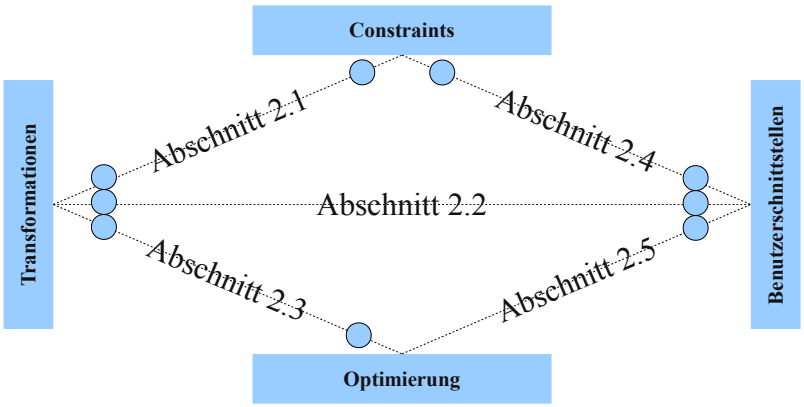


Abbildung 2.1: Klassifikation und resultierender Aufbau des Kapitels.

in drei Kategorien eingeteilt:

1. ✔, voll unterstützter Grundsatz
2. (✔), teilweise unterstützter Grundsatz
3. ✘, nicht unterstützter Grundsatz

In Form von Tabellen befindet sich in den Unterabschnitten jeweils eine Bewertung, die den im jeweiligen Unterabschnitt untersuchten Ansätzen Qualitäten zuordnet. Eine vereinfachende Übersicht über die Bewertung aller Ansätze befindet sich am Ende des Kapitels.

2.1 Transformationen und Constraints

In diesem Abschnitt werden Transformationssprachen und einige dazu verwandte Ansätze vorgestellt, die einerseits zur Modelltransformation oder Erzeugung von Modellen verwendet werden können und andererseits auch über Constraints verfügen. Daher werden in diesem Abschnitt von allen vorgestellten Ansätzen mindestens zwei Grundsätze erfüllt (Constraints, Modelltransformation).

In diesem Abschnitt nicht betrachtet werden Ansätze, die Graphen rein aus Constraints erzeugen. Dabei werden die Knoten und Kanten als Variablen notiert. Da Modelle eine Spezialisierung von Graphen sind, können Modelle theoretisch aus Systemen von Constraints erzeugt werden. R. Tamassia gibt einen Überblick über

Bewertung	Anforderung	Kriterien
✘	Transformationen	Es werden keine Transformationen verwendet.
	Constraints	Es werden keine Constraints verwendet.
	Optimierung	Es wird keine Optimierung verwendet.
	Benutzerschnittstellen	Dieser Ansatz erzeugt keine Benutzerschnittstellen.
(✔)	Transformationen	Es wird min. eine Transformation verwendet und nicht immer direkt interpretiert.
	Constraints	Constraints werden verwendet.
	Optimierung	Optimierung wird verwendet.
	Benutzerschnittstellen	Mit diesem Ansatz wurden für eine sehr beschränkte Domäne Benutzerschnittstellen erzeugt.
✔	Transformationen	Dieser Ansatz wurde dazu entwickelt, Modell-zu-Modell- oder Graphtransformationen frei zu definieren.
	Constraints	Constraint-Solving wird für das Zielmodell oder die Zielbenutzerschnittstelle durchgeführt und die Constraints können in einer deklarativen Sprache notiert werden.
	Optimierung	Beliebige Zielfunktionen können in einer deklarativen Sprache notiert werden.
	Benutzerschnittstellen	Mit diesem Ansatz wurden bereits Benutzerschnittstellen erzeugt.

Tabelle 2.1: Kriterien zur Klassifikation verwandter Arbeiten

einige Algorithmen zur Erzeugung von Graphen mit Constraints und Constraint Systemen [156]. Im Unterschied zu Modell-zu-Modell-Transformationen, die Graphen (bzw. Modelle) als Eingaben haben, werden die von R. Tamassia vorgestellten Algorithmen nur zur Erzeugung von Graphen verwendet.

Hingegen ist eine Graphentransformationssprache meist als Graphgrammatik ausgelegt, d.h. mit ihr können Graphen transformiert und auch über ein Regelwerk erzeugt werden. Vereinfacht betrachtet, bestehen Graphgrammatiken aus einem Startgraphen und einer Menge von Regeln, die aus diesem Startgraphen weitere Graphen erzeugen, indem Nachfolger wieder als Startgraphen verwendet werden. Manche Ansätze eignen sich auch zur Modell-zu-Modelltransformation von Graph-basierten Sprachen (d.h. speziell auch Modelle) in andere Graph-basierte Sprachen, z.B. [145]. Da es sich oft um Regelsysteme handelt, sind viele dieser Sprachen deklarativ. Eine umfassende Einführung gibt die dreibändige Buchreihe "Handbook of graph grammars and computing by graph transformation" ([136], [38], [39]).

Zusätzlich zu Graphgrammatiken existieren zur Transformation von Modellen auch speziell dafür entwickelte Sprachen, nämlich die bereits eingeführten Modell-zu-Modell-Transformationssprachen. Verschiedene Klassifikationen sowie eine allgemeine Einführung in Modell-zu-Modelltransformationssprachen bieten sowohl [35], [159] als auch [141].

Tabelle 2.2 fasst die Bewertung der Ansätze zusammen, indem alle vorgestellten Ansätze noch einmal tabellarisch verglichen werden.

2.1.1 Fokus: Transformationen

Es wurden bereits mehrere grundsätzlich verschiedene Ansätze entwickelt, Constraints und Transformationssprachen zu kombinieren. Diese Ansätze untersuchen den Einfluss von Constraints auf die Transformation oder Generierung von Graphen, wie in dieser Arbeit gefordert. Sie haben allerdings fast alle einen Fokus auf die Struktur von Graphen und ignorieren die in dieser Arbeit geforderten Constraints auf Attributen. Auch die effiziente Behandlung von Constraint-Solving-Problemen auf Attributen im Rahmen einer Erweiterung des Query View Transformations (QVT)-Standards [117] wurde bisher noch nicht untersucht.

2.1.1.1 Ansätze

Attributierte Graph Grammatiken (AGG) dienen der Erzeugung und Transformation von "attributierten" Graphen [137]. Ein attributierter Graph ist ein klassischer Graph mit einer Erweiterung um Attribute, ähnlich denen von Modellen. Eine AGG ermöglicht, zusätzlich zu den klassischen Graphentransformationen (d.h. dem Verändern nur der Kanten und Knoten eines Graphen), das

Setzen von Attributwerten unter Nutzung von Variablen und sog. Anwendungsbedingungen (engl. "application condition"). Da ein attributierter Graph einem Modell sehr ähnlich ist, eignen sich AGGs zur Durchführung von Modell-zu-Modelltransformationen. Q. Limbourg zeigt wie AGGs zur Generierung von Modellen von Benutzerschnittstellen eingesetzt werden können, ohne dass diese echte Constraints auf Attributwerten enthalten (s. [94]).

H. Ehrig erweitert AGGs und zeigt, wie Constraints über Graphen in die bereits erwähnten Anwendungsbedingungen (engl. "application condition") für Graphtransformationsregeln überführt werden können [37]. Dadurch können Constraints über Graphen z.B. mit AGGs notiert werden. Der durch die Transformation entstandene Graph entspricht dann den Anforderungen, die durch die Constraints an ihn gestellt wurden (der Graph ist "konsistent" bzgl. der Constraints).

Neben ihrem breit gefächerten industriellen Einsatz werden **Tripel Graph Grammatiken (TGG)** [145] in einigen Werkzeugen zur Definition von Modelltransformationen verwendet (z. B. [90, 6, 54]). Tripel-Graph-Grammatiken eignen sich zur grafischen, regelbasierten (und damit deklarativen) Notation von Modelltransformationen. In rudimentärer Form wird in einem Ansatz zur Verifikation auch Constraint-Solving unterstützt – ein entsprechender Ansatz wird in Abschnitt 2.1.2 vorgestellt.

U. Montanari beschreibt die Kombination von Graphentransformation und Constraint-Solving zur Auswahl von Transformationsregeln [103]. Er konzentriert sich auf die Transformation von Graphen von Prozessen (ähnlich Workflows), deren Ausführungsreihenfolge bestimmt werden soll. Dazu definiert er eine Menge von Constraints zur temporalen Beschränkung von Prozessen. Die Implementierung von U. Montanari wählt Transformationsregeln an Hand eines Constraint-Solving-Problems aus.

xMOF ist ein Vorschlag, der bei der OMG als QVT-Modell-zu-Modell-Transformationssprache zur Standardisierung von QVT [115] eingereicht wurde. Mit xMOF können Constraints notiert werden, die während der Ausführung von Transformationen berücksichtigt werden müssen. Es fehlt derzeit eine Implementierung, da die Entwickler von Computer Associates eine komplette Implementierung für den Fall versprachen, dass xMOF für den QVT Standard akzeptiert worden wäre. Da xMOF aber nicht akzeptiert wurde, implementierten die Entwickler stattdessen QVT Core, das als eine komplett andere Sprache gilt.

Alloy ist eine Logiksprache, in der Modelle und Transformationsregeln notiert werden [8]. Die Sprache ist auf die Verifikation und Validierung von Modellen spezialisiert worden. Zur praktischen Nutzung der Sprache existiert ein Werkzeug, dass die Transformation von Modellen, basierend auf UML, in diese Sprache automatisiert (UML2Alloy, [7]). Trotzdem werden die Transformationsregeln entweder in der Sprache selbst notiert, oder es wird die Transformation in einem Modell

notiert, das dann in Alloy überführt wird (die Transformaitonen werden also als Modelle betrachtet, s. [19]). Constraints können über Prädikate notiert werden. Ähnlich ist der Ansatz von **Kuester u.a.** [91]. Dieser verifiziert Modelltransformationen, indem ein Modell der Transformation in eine Algebra namens Communication Sequential Processes (s. [65], in der Literatur kurz CSP, nicht zu verwechseln mit Constraint-Solving-Problem, das ebenfalls mit CSP abgekürzt wird) transformiert wird.

2.1.1.2 Abgrenzung

Optimierung wird von allen hier aufgeführten Ansätzen nicht unterstützt. Constraint-Solving hingegen wird mit Einschränkungen von einigen Ansätzen unterstützt:

Die derzeitigen Implementierungen von **AGGs** bieten kein Constraint-Solving in Zielmodellen und operieren auf Graphen statt wie gefordert auf Modellen. Auch die Erweiterung von **H. Ehrig** ermöglicht zwar die Erfüllung von Constraints über Graphen aber es handelt sich bei der Realisierung der Konsistenz im Zielmodell nach Ehrig nicht um Constraint-Solving. Stattdessen wird die Erfüllung der Constraints im Zielmodell durch Einschränkung der Transformationsregeln erreicht: die Constraints werden so in Anwendungsbedingungen überführt, dass die Transformationsregeln keine inkonsistenten Graphen erzeugen können. Bei diesem Ansatz werden mögliche Werte für Attribute im Zielmodell nicht berechnet, sondern durch die Selektion von Transformationsregeln, deren Anwendung im jeweiligen Zustand des Graphen zulässig ist; dies bedeutet aber, dass das Setzen der Werte von Attributen nicht durch Constraints geschehen kann, sondern nur durch Zuweisung. Also verlässt sich der Ansatz bei der Definition der Werte für Attribute auf die Entwickler der Transformationsregeln und den Modellierer. Der Ansatz ist daher ausschließlich zur Beschränkung der Struktur von Graphen geeignet und nicht von Attributwerten.

Constraint-Solving wird im eingereichten Vorschlag für **xMOF** nicht erwähnt. Auch sonst werden keine Hinweise auf einen möglichen Algorithmus zur Implementierung von xMOF gegeben. Allerdings lässt die Art wie xMOF aufgebaut ist (Auflistung beliebiger Object Constraint Language (OCL)-Invarianten in einer Transformationsregel, s. [116]) den Schluss zu, dass für eine Implementierung ein Ansatz gewählt werden könnte, der Constraint-Solving verwendet. Es ist fraglich, ob eine effiziente Implementierung eines xMOF-Interpreters oder -Compilers überhaupt möglich ist, da die Constraints sehr allgemein gehalten sind und alle Aspekte von OCL abdecken können – die sehr allgemeinen Constraints müssten also auf ein Constraint-Solving-Problem abgebildet und dort effizient gelöst werden können.

Hingegen erscheinen **TGGs** für diese Arbeit prinzipiell als grundlegende Sprache für eine Implementierung in Frage zu kommen, da sich mehrere Grundsätze

mit Tripel Graph Grammatiken umsetzen lassen. Die reine Definition von einigen Constraint-Solving-Problemen wird unterstützt und ähnlich zu AGGs umgesetzt. Auch die standardisierte Sprache QVT lässt sich zumindest teilweise mit TGGs implementieren [58]. Allerdings liegt der Fokus von Tripel Graph Grammatiken nicht auf der Definition von Constraint-Solving-Problemen. In den bisher bekannten Werkzeugen für TGGs ist weder Constraint-Solving implementiert (ein Ansatz unterstützt zwar Constraint-Solving, ist aber ein Ansatz zur Verifikation, s. Abschnitt 2.1.2), noch ist es mit den Werkzeugen möglich, Optimierungsprobleme zu lösen.

Das von **U. Montanari** verwendete Prinzip könnte auf eine große Anzahl von verschiedenen Graphen verallgemeinert werden. Dadurch könnten Graphen mit temporalen Constraints mit Constraint-Solving verarbeitet werden. Allerdings werden wie beim Ansatz von H. Ehrig Zielgraphen nicht direkt durch das Lösen der Constraints berechnet und es findet daher kein Constraint-Solving für Attribute statt. Die möglichen Transformationsregeln werden durch den Constraint-Solver ausgewählt, sodass nur die Struktur der Graphen dem durch die Constraints deklarierten, gewünschten Ergebnis entsprechen.

Alloy und der Ansatz von **Kuester u.a.** definieren Constraints nicht über Sprach-Konstrukte und implementieren daher kein echtes Constraint-Programming, sondern sind formale Sprachen zur Verifikation von Transformationen. Auch wenn diese Ansätze Constraint-Solving auf diskreten Typen verwandt sind, können reelle Zahlen damit nicht gut abgebildet werden, da diese Ansätze auf diskreten Strukturen operieren. Auf Grund der Mechanismen zur Interpretation fehlt auch die Propagierung von Constraints, die die Ansätze – in Abhängigkeit der transformierten Transformationsregeln – beschleunigen würde. Hierbei werden mögliche Lösungen nicht allein durch rechenintensives Suchen gefunden sondern die Lösungen werden möglichst auf Grund von mathematischen Ableitungen effizient berechnet.

2.1.1.3 Fazit

Die theoretisch motivierten Ansätze, die vor allem zur Verifikation von Transformationen eingesetzt werden, sind für die Transformation von Modellen von Benutzerschnittellen nicht geeignet. Entweder sind die Ansätze nur Vorschläge, deren Umsetzbarkeit bezweifelt werden kann (**xMOF**) oder sie verwenden formale Sprachen, die nur über Umwege zur Definition von Constraints verwendet werden können (**Alloy**, **Kuester**). Eine Modell-zu-Modell-Transformationssprache zur Transformation von Modellen von Benutzerschnittellen sollte die Definition von Constraints direkt in der Sprache erlauben und sollte implementierbar sein um einen direkten praktischen Nutzen zu haben.

Einige Ansätze basierend auf Graph-Grammatiken (mit Anwendungsbedin-

gungen) sind prinzipiell zur Definition von Constraints geeignet (**TGG**, **Ehrig**). Der Ansatz von Ehrig erfordert jedoch erhebliches Ausprobieren von Lösungen, während effiziente Interpreter für TGGs mit Hilfe echter Constraint-Solver noch auf sich warten lassen (im nächsten Abschnitt wird ein ineffizienter Ansatz zur Verifikation von Modellen mit Constraint-Solving vorgestellt). Ansätze zur Transformation von Modellen von Benutzerschnittstellen sollten daher sog. Constraint-Propagation unterstützen, die unnötiges Ausprobieren verhindert. Aktuelle Ansätze basierend auf Graph-Grammatiken sind daher derzeit nicht als Grundlage für diese Arbeit geeignet.

Tabelle 2.2 fasst die Bewertung der Ansätze zusammen.

2.1.2 Fokus: Constraints

In diesem Abschnitt wird an Ansätzen demonstriert, wie nützlich Constraint-Solving im Zusammenspiel mit Modellen ist. Dabei können aus den vorgestellten Ansätzen Methoden entnommen werden, die zur Implementierung von Algorithmen zur Kombination von Modellen mit Constraint-Solving geeignet sind. Diese sind dann speziell auch für die Implementierung einer Transformationsmaschine (also eines Interpreters zur Interpretation von Modell-zu-Modell-Transformationen) mit Constraint-Solving anwendbar, die in den Kapitel 4 und 5 entwickelt wird.

2.1.2.1 Ansätze

J. Winkelmann u.a. erweitern den bereits vorgestellten Ansatz von H. Ehrig u.a. [37], indem sie OCL [116] in Anwendungsbedingungen für Graphentransformationen übersetzen [167]. Dies bedeutet, dass der Ansatz von H. Ehrig nun auch mit der standardisierten Constraint-Sprache OCL verwendet werden kann, was die Deklaration von Constraints für viele Modellierer einfacher macht. Mit dem Ansatz erzeugen die Entwickler dabei Modelle aus Meta-Modellen, die in UML [114] notiert werden und OCL-Constraints enthalten. Es handelt sich insofern um einen Ansatz, der Modelle durch Anwendung von klassischen Sprachen der modellbasierten Entwicklung erzeugen kann. Einen ähnlichen Ansatz verfolgen **M. Gogogolla u.a.**, indem sie in ihrem Werkzeug namens "USE" automatisiert Modelle erzeugen, um Meta-Modelle zu validieren (z.B. um festzustellen, ob unter den gegebenen Constraints überhaupt ein gültiges Modell existieren kann) [55].

Modelle für das Zusammenspiel von Komponenten, die für die komponentenbasierte Entwicklung von Anwendungen genutzt werden, können z.B. mittels der UML notiert werden. Ein UML-Komponentendiagramm enthält allerdings auf Grund seiner statischen Natur keine funktionalen Aspekte der Anwendung. Einige "Qualitätseigenschaften von Komponenten" können aber als ein Konglo-

merat aus Eigenschaften von Klassen und OCL-Ausdrücken modelliert werden. So können z.B. aus bereits bekannten Qualitätseigenschaften einer Menge von Diensten die Qualitätseigenschaften eines neuen Dienstes modelliert werden, der aus den Diensten zusammen gesetzt wurde. Dazu definieren **O. Defour u.a.** eine Erweiterung des Teils der UML mit dem sich Komponenten modellieren lassen (genannt "QoSCL", [74]). Dieses Modell wird dann durch eine Modell-zu-Code-Transformation in ein Constraint-Logisches-Programm übersetzt. Die Modell-zu-Code-Transformation ist speziell für die im Ansatz entwickelte Sprache QoSCL entwickelt worden und daher nicht allgemein für beliebige Meta-Modelle gültig.

Im Gegensatz dazu, aber trotzdem durch eine Abbildung auf ein Constraint-Solving-Problem, wählten **J. Cabot u.a.** einen eher allgemeinen Ansatz, der auf beliebige, mit der UML notierte Modelle anwendbar ist [28]. Ihr Ziel war die Verifikation von UML-Klasssendiagrammen. Diese bestehen aus Klassen und Assoziationen sowie einer Menge von Constraints. Der Ansatz transformiert das Klassendiagramm und die Constraints in ein Constraint Logisches Programm eines PROLOG-Dialekts. Danach wird ein in die gewählte PROLOG-Implementierung eingebauter Constraint Solver gewählt, der die Constraints im Modell zu lösen versucht. Derzeit ist der Ansatz noch auf UML-Klassendiagramme beschränkt.

Parallel zur Präsentation eines Teils dieser Arbeit [125] wurde der Ansatz von **J. Cabot auf Modelltransformationen** erweitert [26]. Es können somit Modelltransformationen verifiziert werden, die in den Sprachen QVT Relations [117] oder Tripel-Graph-Grammatiken [145] notiert werden. Dabei wird das im vorigen Absatz entwickelte Werkzeug verwendet, in [26] UML2CSP genannt. Über den Umweg der Definition der Transformationen über Modelle (für die Theorie dazu s. [76]) und Invarianten kann eine Constraint-basierte Darstellung der Transformation erstellt werden. Die gesamte Transformation besteht dann nur aus den von J. Cabot als "Invarianten" bezeichneten Constraints.

2.1.2.2 Abgrenzung

Optimierung wird von allen vorgestellten Ansätzen nicht unterstützt.

Die Ansätze **J. Winkelmann** und **M. Gogolla** interpretieren Constraints zur Erzeugung von Modellen. Dabei werden Modelle erzeugt und nicht mit Hilfe einer Transformationssprache transformiert.

Der Ansatz von **O. Defour** kann nicht zur Modelltransformation von Modellen im allgemeinen verwendet werden, da er nur für die Sprache QoSCL entwickelt wurde, die auf die Interpretation von Quality of Service spezialisiert ist.

Der Ansatz von **J. Cabot** verifiziert Modelle und Constraints auf Modellen indem er sie in PROLOG überführt ([28]). Während die Transformation von Modellen und Constraints in die Programmiersprache PROLOG einen guten Ansatz zur Implementierung einer Transformationsmaschine darstellen würde, wird der

Ansatz zur Verifikation und nicht zur Modell-zu-Modell-Transformation genutzt. Modelltransformationen erlauben normalerweise mehrere Ursprungsmodelle. Nicht jede Modelltransformation kann nur durch OCL-Constraints ausgedrückt werden; aus diesem Grunde definierte die OMG den QVT Standard, der eine Erweiterung der OCL darstellt. Dies bedeutet auch, dass der Ansatz von J. Cabot nicht als allgemeine Modell-zu-Modell-Transformationssprache verwendet werden kann.

Der zweite Ansatz von **J. Cabot auf Modelltransformationen** nutzt eine Transformation in PROLOG um dadurch Modelle zu transformieren. Die Transformation wird aber nicht als solche durch ein Programm durchgeführt, sondern durch das Lösen des Constraint-Solving-Problems. Die Ausgangs- und Zielmodelle sind nicht fest definiert (sie werden durch komplexe Constraints beschrieben) und die Abhängigkeiten zwischen Transformationsregeln sind "implizit" gegeben: soll eine Modelltransformation ausgeführt werden, muss der Entwickler die Modelle durch das Hinzufügen weiterer Invarianten (Constraints) auf das entsprechende Ausgangsmodell einschränken. Dieser Ansatz hat daher noch mehrere Probleme: die Invarianten für Modelle zur Modelltransformation werden nicht automatisch erzeugt und in einer komplett anderen Sprache als klassische Modelle notiert (nämlich als Constraints). Dies ist nicht komfortabel, da die Modellierungssprache zur Beschreibung von Modellen nicht verwendet werden kann und stattdessen Constraints notiert werden müssen. Die Verifikation testet verschiedene Eigenschaften, dafür sind aber Mengen von per Hand gewählter Gegenbeispiele erforderlich (bzw. die Einschränkung des Lösungsraumes durch weitere Constraints per Hand). Deshalb ist der Ansatz nur für eine begrenzte Menge von Anwendungsfällen anwendbar. Da die in diesem Ansatz als Ausgangssprachen gewählten Transformationssprachen QVT Relations und Tripel-Graph-Grammatiken nicht verändert wurden, ist der Ansatz auch nicht in der Lage, Optimierung zu verwenden, denn dafür wäre das hinzufügen von Sprachkonstrukten notwendig. Ferner bildet die Implementierung OCL-Constraints in QVT Relations nicht in der Weise ein, wie diese Arbeit die Constraints einbinden soll, da nur Pseudo-Constraints mit dem Komparator "=" zulässig sind. Die Abhängigkeiten zwischen Transformationsregeln werden implizit interpretiert und durch sog. "reified" Constraints gegeben, bei der der Solver den kompletten Zustandsraum durchsuchen muss (wegen der Verknüpfung durch "and") und zusätzlich keine Propagierung von Constraints verwenden kann. Wenn Abhängigkeiten zwischen Transformationsregeln existieren, ist dieser Ansatz daher in der Praxis auch als sehr ineffizient anzusehen. Diese Ansicht bestätigt J. Cabot in [27], nach der für die untersuchten Modelle maximal zwei Instanzen von Klassen erzeugt werden dürfen und diese Einschränkung als "wichtig" (engl. important) für dieses Konzept bezeichnet wird.

2.1.2.3 Fazit

Um den Grundsatz Rechnung zu tragen Modell-zu-Modell-Transformationen ausführen zu können, erscheinen Ansätze ungeeignet, die zur Generierung von Modellen auf den Prinzipien von **M. Gogolla** und **J. Winkelmann** aufbauen. Diese transformieren Modelle nicht, sondern erzeugen mögliche Modelle aus einer Menge von Constraints.

Die Ansätze von **O. Defour** und **J. Cabot** hingegen demonstrieren, dass Constraints von Modellen als Constraint-Logisches-Programm interpretiert werden können und somit könnten auch die Constraints der Implementierung der vorliegenden Arbeit auf diese Weise interpretiert werden. In der vorliegenden Arbeit soll daher diese Vorgehensweise als möglicher Weg zur Interpretation von Sprachen Berücksichtigung finden.

Tabelle 2.2 bietet eine Übersicht über die Bewertung der Ansätze.

2.2 Transformationen und Benutzerschnittstellen

In diesem Abschnitt werden Ansätze vorgestellt, die verschiedene Aspekte von Benutzerschnittstellen berechnen, wie z.B. deren Layout. Dies wird entweder für Modelle von Benutzerschnittstellen oder Editoren/Benutzerschnittstellen für Graphen (als allgemeinere Struktur) durchgeführt.

Einige Ansätze konzentrieren sich auf die Generierung von Graphen mit verschiedenen Layouts, die durch Kombination von verschiedenen Constraints aus einer festen Menge implementiert werden (s. Unterabschnitt 2.2.1).

Eine weitere Gruppe von Ansätzen konzentriert sich auf die Transformation von Modellen von Benutzerschnittstellen. Dabei werden entweder klassische Programmiersprachen oder fertige Transformationssprachen verwendet (s. Unterabschnitt 2.2.2).

Tabelle 2.3 fasst die Bewertung zusammen.

2.2.1 Fokus: Graphentransformation und Layout

Das Berechnen von Layouts für Graphen ist dem hier geforderten Ansatz insofern verwandt, als einerseits

- eine Graphentransformation durchgeführt wird, und andererseits
- ein Layout berechnet wird.

Ansatz	Klasse	Transformationen	Constraints	Optimierung	Benutzerschnittstellen
Transformationen und Constraints					
	Fokus: Transformationen				
AGGs, Rudolf [137]		✔	(✔)	✘	✘
Ehrig [37]		✔	(✔)	✘	✘
TGGs, Schürr [145]		✔	(✔)	✘	✘
Montanari [103]		✔	(✔)	✘	✘
xMOF, Compuware [33]		✔	(✔)	✘	✘
Alloy, Anastasakis [8]		✔	(✔)	✘	✘
Küster [91]		✔	(✔)	✘	✘
	Fokus: Constraints				
Winkelmann [167]		(✔)	(✔)	✘	✘
USE, Gogolla [55]		(✔)	(✔)	✘	✘
QoSCL, Defour [74]		(✔)	✔	✘	✘
UML2CSP, Cabot [28]		(✔)	✔	✘	✘
Cabot [26]		✔	✔	✘	✘

Tabelle 2.2: Klassifikation verwandter Arbeiten, Transformationen und Constraints

Die Unterschiede zwischen den vorgestellten Arbeiten und dem hier entwickeltem Ansatz liegen in der Umsetzung und im Fokus. Während der hier propagierte Ansatz unter anderem die Berechnung von Layouts für Benutzerschnittstellen zum Ziel hat, haben die Ansätze in diesem Unterabschnitt einen Fokus auf die Berechnung von Layouts zur Präsentation von Graphen. Da aber alle Ansätze dabei z.B. das Überlappen von Kanten und Knoten verhindern müssen, sind die Ansätze zum Geforderten ähnlicher, als ein kurzer Blick auf die Materie vermuten ließe. Z.B. können mit Spatial Graph Grammars (SGG) (s. letzter Absatz in diesem Abschnitt) auch Transformationen von Modellen von Benutzerschnittstellen durchgeführt werden. Allerdings mangelt es allen Ansätzen an einer Möglichkeit, mathematische Optimierung auszuführen, und sie verfügen nur über eine fest definierte Menge von Constraints, die von Entwicklern nicht erweitert werden können.

2.2.1.1 Ansätze

F. Brandenburg präsentiert eine Graph Grammatik ("Layout Graph Grammar", **LGG**), die zum Layout von Graphen dient [23]. Er präsentiert keine Implementierung, sondern stützt sich nur auf theoretische Grundlagen von kontextfreien Graph-Grammatiken.

G. Zinßmeister und C. McCreary präsentieren ein Framework, das Attributierte Graph Grammatiken um die Fähigkeit erweitert, Layouting-Algorithmen als sog. Attribut-Evaluatoren einzubinden, um Layouts für gerichtete, azyklische Graphen zu erzeugen [174].

Spatial Graph Grammars (**SGGs**) dienen der Definition von Graphentransformationen unter Nutzung räumlicher Constraints [88]. Deshalb eignen sie sich zur Definition von einer Menge von Benutzerschnittstellen, die den Anforderungen genügen, die die Entwickler der SGGs an die Struktur des Graphen stellen, der die Benutzerschnittstelle modelliert. Sie verwenden keinen Constraint Solver, sondern erweitern die Notation einer Graph-Grammatik namens RGG (s. [173]) um einige vordefinierte räumliche Constraints.

J. Rekers und A. Schürr versuchen das Problem, grafische Sprachen zu definieren, in mehrere Teile zu zerlegen [132]. Dabei beschreibt der "Abstract Syntax Graph" die Konstrukte, die mit Informationen über ihr Layout angereichert werden sollen. Das "Physical Layout" beschreibt die grafischen Primitive, die auf dem Bildschirm angezeigt werden. Der "Spatial Relations Graph" verbindet die beiden Teile der Sprache miteinander und bietet Konstrukte ähnlich denen von SGGs an.

In dem Papier von J. Rekers und A. Schürr ([132], Tabelle auf Seite 7) werden ferner sieben weitere Ansätze untersucht, die weitere Grundsätze nicht erfüllen (sie nutzen zur Eingabe textuelle Notationen statt Graphen) und sollen deshalb hier - bis auf Ansätze basierend auf Multisets (s. Abschnitt 2.4.2) - nicht vorgestellt werden.

2.2.1.2 Abgrenzung

Da Graphen eine Verallgemeinerung von Modellen sind, ist der Ansatz von F. Brandenburg dem in der vorliegenden Arbeit vorgestelltem insofern ähnlich, als dass das Ziel eine Art Transformationssprache in Kombination mit der Berechnung von Layouts von Graphen ist. Bei dem Ansatz, der in der vorliegenden Arbeit gefordert wird, geht es dabei um Modellelemente und deren Attribute, deren Werte theoretisch durch **LGGs** (teilweise) berechnet werden könnten. Allerdings ist die Transformationssprache von Brandenburg auf kontextfreie Graphtransformationen beschränkt und impliziert die Verwendung eines speziellen Algorithmus, der das Layout berechnet. Mit kontextfreien Graphtransformationen können Knoten nur nach deren Typ ausgewählt werden. Dadurch sind die Möglichkeiten der

Transformation und der Berechnung des Layouts von Graphen stark eingeschränkt. Modellelemente, die transformiert werden sollen, können z.B. nicht nach deren Assoziationen zu anderen Modellelementen ausgewählt werden ("kontextsensitiv").

Keiner der im Ansatz von **G. Zinsmeister und C. McCreary** verwendeten Algorithmen für Attribut-Evaluatoren kann allgemeine Constraints verarbeiten. Theoretisch wäre es aber möglich, einen Constraint-Solver nach der Transformation zu verwenden und einen Attribut-Evaluator zu verwenden, der Constraints in ein entsprechendes Constraint-System einfügt. Dieser Ansatz wird aber von G. Zinßmeister nicht erwähnt. Die in dieser Arbeit entwickelte Transformationssprache verwendet eine Sprache, um Entwicklern zu erlauben, beliebige Constraints zu deklarieren und ist daher nicht auf speziell entwickelte Attribut-Evaluatoren angewiesen.

Da in **SGGs** die Constraints bereits feststehen, können mit SGGs auch keine allgemeinen Constraints verarbeitet werden. In Kapitel 5 wird unter anderen auch dieser Ansatz untersucht, da er nach [88] zur Transformation von Modellen von Benutzerschnittstellen geeignet erscheint.

Der Ansatz von **J. Rekers und A. Schürr** konzentriert sich auf die Definition von neuen grafischen Sprachen und nicht auf deren Transformation. Um alle Grundsätze der geforderten Transformationssprache zu erfüllen, fehlen Konstrukte zur Beschreibung von Optimierung.

2.2.1.3 Fazit

Kontextfreie Transformationssprachen, wie der Ansatz von **J. Brandenburg** sind für den Gebrauch in Benutzerschnittstellenmodellen nicht geeignet, da z.B. die Komponenten von Benutzerschnittstellen räumlich von anderen abhängen und diese räumlichen Relationen teilweise durch Assoziationen beschrieben werden (z.B. Verschachteln von Komponenten).

Die hier geforderte Transformationssprache muss kontextsensitiv sein und soll sich nicht auf das Layout von Graphen beschränken.

SGGs werden in Kapitel 5 genauer untersucht, da der Ansatz zur Transformation von Modellen prinzipiell geeignet erscheint. Allerdings sind die Constraints nicht frei wählbar und auch in diesem Fall nur eine sehr eingeschränkte Menge von Graphen kann mit diesem Ansatz transformiert werden. Die geforderte Sprache soll daher frei wählbare Constraints unterstützen.

In Tabelle 2.3 werden die Ansätze mit denen aus dem nächsten Abschnitt verglichen.

2.2.2 Fokus: Benutzerschnittstellen

Eine Einführung in die Transformation von Modellen von Benutzerschnittstellen zum damaligen Stand gibt P. Silva in [147]. Modell-zu-Modell-Transformationssprachen wurden damals noch nicht verwendet.

In diesem Unterabschnitt werden Ansätze vorgestellt, die modellbasierte Entwicklung von Benutzerschnittstellen mit Modell-zu-Modell-Transformationssprachen verwenden, oder bei denen es sich um modellbasierte Ansätze mit beliebigen Transformationen handelt, die Constraints zur Entwicklung von Benutzerschnittstellen unterstützen.

2.2.2.1 Ansätze

Q. Limbourg definiert in seiner Arbeit [94] einen umfassenden modell-getriebenen Ansatz zur Entwicklung von Benutzerschnittstellen. Dazu definiert er mehrere XML-basierte Meta-Modelle (genannt "UsiXML"), die mittels Graphtransformationen ineinander überführt werden können. Die von ihm verwendete Graphtransformationssprache ist AGG (s.o.). Der Ansatz beleuchtet die Verwendung von Modell-zu-Modell-Transformationen mittels Graphtransformationen und bietet zum ersten Mal einen kompletten Prozess zur Entwicklung von Benutzerschnittstellen mit Transformationen.

TransformiXML ist ein Werkzeug, das auf der Sprache UsiXML aufbaut und das die Ausführung von Transformationen erleichtert [95, 29]. Zur Ausführung von Transformationen stehen sowohl die Möglichkeit, Regeln sequentiell auszuführen als auch eine grafische Benutzerschnittstelle zur Verfügung. TransformiXML transformiert die in UsiXML notierten Modelle und eine Menge von per Hand selektierten Transformationsregeln in AGG-kompatible Graphen und Regeln. Der Ansatz ermöglicht so die Transformation von Modellen von Benutzerschnittstellen mittels Graphentransformation, ohne dass Entwickler Kenntnisse in Graphentransformation haben müssen.

Die Arbeiten von **M. Trapp und M. Schmettow** erweitern den von Q. Limbourg bekannten Ansatz (s.o., [94]) um die Verwendung von formal beschriebenen Mustern über die Benutzbarkeit ("usability patterns") der Benutzerschnittstellen in den Transformationsregeln [158]. Zur Beschreibung der Transformationsregeln nutzen sie die bereits in UsiXML verwendeten AGGs, um Task-Modelle zu transformieren. Sie untersuchen vor allem die Konsistenz der Benutzerschnittstellen über verschiedene Geräte hinweg und behaupten, dass sich durch Nutzung von Transformationsregeln in Kombination mit den verwendeten Mustern die Nutzbarkeit der zu entwickelnden Benutzerschnittstellen erhöht.

F. Paternó u.a. präsentieren mit TERESA (s. [121]) und MARIA (s. [120]) zwei Ansätze, die auf XML-basierten Modellsprachen basieren. TERESA trans-

formiert von einem Modell, das die Aufgaben des Benutzers modelliert, in ein Modell mit abstrakten und konkreten Komponenten von Benutzerschnittstellen. Dafür wird von MARIA das Modell zur Laufzeit per XSLT transformiert – eine Transformationssprache für XML-Sprachen. Einen ähnlichen Ansatz verfolgt **J. S. Sottet**, der die Modelle zur Laufzeit transformiert [150]. Anstatt XSLT nutzt er ATL ([75]), eine Modell-zu-Modell-Transformationssprache.

ArtStudio von D. Thevenin definiert und nutzt verschiedene Meta-Modelle und Modell-zu-Modell-Transformationen [157]. Die meisten der Modell-zu-Modell-Transformationen werden bei diesem Ansatz mittels einer klassischen Regelsprache ("JAVA Expert System Shell, JESS"). Die Transformationen beschränken sich auf die Transformation der Struktur von Modellen. Für das nachfolgende Constraint-Solving wird der Ansatz von G. Badros (s.u. bzw. [13]) verwendet, der als ein nachfolgender Schritt behandelt wird.

Garnet verfügt über eine Reihe grafischer Editoren, die zusammen mit mehreren Generatoren und Interpretern die Modellierung, Generierung und Ausführung von interaktiven Systemen ermöglichen [106]. Constraints modellieren die Abhängigkeiten zwischen den modellierten Komponenten von Benutzerschnittstellen. Der Ansatz war einer der ersten mit modellbasierten Entwicklungswerkzeugen, um Benutzerschnittstellen zu spezifizieren, der auch Constraints verwendete. B. Myers erweitert den Ansatz in **Amulet** [104] um die Möglichkeit, beliebige Constraints in C++ zu definieren, sowie die Möglichkeit, von den verwendeten Komponenten der Benutzerschnittelle zu abstrahieren.

T. Browne untersuchte in **MASTERMIND** die Erzeugung von Benutzerschnittstellen an Hand von "textuellen" Modellen [24]. Obwohl es in der Literatur häufig zu den User Interface Management Systems (UIMS) gezählt wird, behandelt es auch die Erzeugung von C++ Quellcode für Benutzerschnittstellen und kann so methodisch als modellbasierter Ansatz gesehen werden. Es existieren verschiedene Modelle, in denen Temporale- und Layout-Constraints verwendet werden können. Diese werden dann in Constraints übersetzt, die die Laufzeitumgebung interpretiert.

Personal Universal Controller von **J. Nichols** u.a. ist ein modellbasierter Ansatz, der ursprünglich zur Definition von Layouts für universelle Fernbedienungen auf PDAs gedacht war. Der Ansatz ist auch zur Definition sonstiger Schnittstellen nutzbar [110]. Benutzerschnittstellen werden bei diesem Ansatz mit einer speziellen XML-basierten Notation gespeichert. Zur Definition von Transformationen in Modelle entsprechender Zielsysteme werden Templates verwendet. Diese verhindern, dass für jedes Zielsystem immer neue Transformationen entwickelt werden müssen. Sie können auch helfen, die Transformationen von entsprechenden Meta-Modellen zu abstrahieren, wie von N. Aquinto u.a. in [12] erkannt wurde, die einen ähnlichen Template-basierten Ansatz verfolgen.

2.2.2.2 Abgrenzung

Der Ansatz von **Q. Limbourg** ist der erste und bekannteste Ansatz, der eine Methodik zur Entwicklung von Modellen von Benutzerschnittstellen festlegt. Ebenso wie der darauf aufbauende Ansatz **TransformiXML** verzichtet der Ansatz auf die Verwendung von Constraint-Solving auf Attributen und Optimierung in der verwendeten Transformationssprache.

Im Gegensatz zu den Grundsätzen dieser Arbeit wird im Ansatz von **M. Trapp und M. Schmettow** keine Transformationssprache verwendet, die Constraint-Solving unterstützt. Vielmehr verlässt sich der Ansatz auf eine festgeschriebene Menge von Mustern. Auch **Garnet**, **Amulet** und **MASTERMIND** verwenden keine Modell-zu-Modell-Transformationssprache. Mit Garnet und Amulet werden die Transformationen in Sprachen wie z.B. C++ geschrieben. Daher wird auch nicht die Möglichkeit geboten, Constraints und Transformationsregeln in einer dafür geeigneten Sprache zu definieren, wie in dieser Arbeit gefordert.

MASTERMIND bietet zwar Sprachen zur Festlegung der Constraints aber nicht in Modell-zu-Modell-Transformationen. Trotzdem ist der Ansatz, Constraints mit Modellen zu verbinden, auch für diese Arbeit interessant. Die Integration von Constraint-Solving in den Entwicklungsprozess bedingt allerdings, dass die Constraints während des Entwicklungsprozesses interpretiert werden und nicht erst zur Laufzeit. Dadurch kann der Entwickler nach der Transformation unter Berücksichtigung der Constraints noch das Modell anpassen und ist nicht gezwungen, wie im Fall von MASTERMIND, das die Constraints zur Laufzeit interpretiert, Spezialfälle im Constraint-System zu berücksichtigen.

ArtStudio ist der einzige Ansatz der einen Constraint-Solver verwendet. Es handelt sich beim Constraint-Solving-Prozess von ArtStudio allerdings nicht wie gefordert um einen Teil einer Modell-zu-Modell-Transformationen, da er durch einen externen Constraint-Solver als ein externer Schritt ausgeführt wird. Dadurch können die Modell-zu-Modell-Transformationssprache und die Sprache zur Definition von Constraints nicht integriert werden. So müssen die Transformationen und das Constraint-Solving mit verschiedenen Sprachen durchgeführt werden, was die Benutzbarkeit des Ansatzes deutlich reduziert. Für Transformationen werden in ArtStudio nur Regeln eines Expertensystems zugelassen.

Die Ansätze von **M. Trapp und M. Schmettow**, **J. Nichols**, **J.S.-Sottet**, **Q. Limbourg** und **F. Paternó** unterstützen kein Constraint-Solving. Die anderen Ansätze, die Constraints erlauben, unterstützen Mechanismen, bei denen die Constraints bereits feststehen, oder mit Hilfe klassischer Sprachen kompiliert werden können. Dies erfordert höheren Aufwand bei der Programmierung der Transformationen.

Alle genannten Ansätze unterstützen Optimierung nicht.

2.2.2.3 Fazit

Ansätze, die keine Modell-zu-Modell-Transformationssprache verwenden, erschweren die Programmierung von Modell-zu-Modell-Transformationen. Die Verwendung klassischer Programmiersprachen würde den Aufwand erhöhen.

Q. Limbourgs vielfach verwendeter Ansatz deutet auf die Bedeutung dieses Ansatzes hin. Die Methodik der vorliegenden Arbeit kann die Prinzipien der Methodik des Ansatzes aufgreifen und sie um Constraint-Solving erweitern. So können Layouts von Benutzerschnittellen auch mit Hilfe von Constraints berechnet werden.

M. Trapp und M. Schmettow behaupten, dass sich durch Nutzung von Transformationsregeln in Kombination mit verschiedenen Mustern die Nutzbarkeit der zu entwickelnden Benutzerschnittstellen erhöht. Die Konsistenz zwischen erzeugten Benutzerschnittstellen ist potenziell höher, wenn diese mit einer Transformation erzeugt werden. Die Autoren folgen dem Ansatz, dass Benutzerschnittstellen sich ähnlicher sind, wenn sie mit dem gleichen Algorithmus – also nach dem gleichen Schema – erzeugt werden. Obwohl diese Behauptung nur durch ein einfaches Beispiel belegt wird, unterstützt diese doch das Ziel dieser Arbeit, Benutzbarkeit in Transformationen zu berücksichtigen.

ArtStudio – der einzige Ansatz, der einen Constraint-Solver einsetzt – motiviert, dass der Constraint-Solving-Prozess nicht (wie in ArtStudio) als ein von der Transformation getrennter Schritt durchgeführt wird. Vielmehr sollte dieser mit der Transformation integriert sein. Erstens können in ArtStudio die Sprachen zu Beschreibung von Modell-zu-Modell-Transformationen als auch die zur Deklaration von Constraint-Solving-Problemen nicht integriert werden, was die Benutzbarkeit des Gesamtsystems für Entwickler beeinträchtigt. Zweitens kann erst durch die Kombination der beiden Schritte in einer Sprache Transformationen und Constraint-Solving in einem einzigen Programm dieser Sprache kombiniert ausgeführt werden. Dies ermöglicht, dass das Constraint-Solving über verschiedene Constraint-Systeme hinweg durchgeführt werden kann, die durch den Transformationsprozess erzeugt werden. Constraints können in einem System voneinander abhängen, was in zwei getrennten Systemen nicht der Fall ist. Sind die Sprachen also nicht integriert, muss der Entwickler nach jedem erzeugten Modell gegebenenfalls abhängige Constraints neu festlegen und den Constraint-Solving-Prozess per Hand ausführen.

In Tabelle 2.3 befindet sich eine vergleichende Bewertung der Ansätze.

Ansatz	Klasse	Transformationen	Constraints	Optimierung	Benutzerschnittstellen
Transformationen und Benutzerschnittstellen					
	Fokus: Transformationen				
LGGs, Brandenburg [23]		✔	(✔)	✘	(✔)
Zinßmeister [174]		✔	(✔)	✘	(✔)
SGGs, Kong [88]		✔	(✔)	✘	✔
ASG/SRG, Rekers [132]		✔	(✔)	✘	(✔)
	Fokus: Benutzerschnittstellen				
UsiXML, Limbourg [94]		(✔)	(✔)	✘	✔
TransformiXML, ” [95]		✔	(✔)	✘	✔
Trapp [158]		(✔)	(✔)	✘	✔
Teresa, Paterno [121]		(✔)	✘	✘	✔
Maria, ” [120]		(✔)	✘	✘	✔
Sottet [150]		(✔)	✘	✘	✔
ArtStu., Thevenin [157]		(✔)	(✔)	✘	✔
Garnet, Myers [106]		(✔)	(✔)	✘	✔
Amulet, Myers [104]		(✔)	(✔)	✘	✔
Masterm., Browne [24]		(✔)	(✔)	✘	✔
PUC, Nichols [110]		(✔)	✘	✘	✔
Aquinto [12]		(✔)	(✔)	✘	✔

Tabelle 2.3: Klassifikation verwandter Arbeiten, Transformationen und Benutzerschnittstellen

2.3 Transformationen und Optimierung

Die Integration von Transformationen und Optimierung ist besonders interessant: durch die Transformation kann aus einer Menge von Zielmodellen eines ausgewählt werden, das nach einem definierten Kriterium optimal ist. Im folgenden werden die Sprachen und Ansätze aufgezählt, die Optimierung und Graph- oder Modell-zu-Modell-Transformationen kombinieren und entweder Transformationen (s. Unterabschnitt 2.3.1) oder Optimierung (s. Unterabschnitt 2.3.2) als Fokus haben. Ausgeschlossen werden Ansätze, die Optimierung nicht im Sinne dieser Arbeit verstehen, z.B. die "Optimierung" von Maschinensprache in Compilern mit Graph-Transformationen oder die "Optimierung" von Modellen, um z.B. hierarchische Strukturen in flache zu transformieren (z.B. [102, 92]). Solche Ansätze versuchen nicht, eine Funktion zu maximieren oder zu minimieren, sondern wenden nur eine einfache Graph-Transformation an um Graphen zu verändern.

Tabelle 2.4 fasst die Bewertung der Ansätze zusammen.

2.3.1 Fokus: Modell- oder Graphentransformation

Wenige Ansätze verbinden Optimierung mit einer Form von Graphen- oder Modelltransformationen. Solche Ansätze finden sich in der Literatur äußerst selten. Sie implementieren Optimierung nicht über Zielfunktionen, sondern über Gewichte, die beim Feuern von Regeln kumuliert werden. Jede Regel hat dabei ein eigenes Gewicht.

2.3.1.1 Ansätze

T. Ruys zeigt in [139], wie Optimierung mit einem Model-Checker durchgeführt werden kann. Dabei konzentriert er sich zwar nicht auf die Transformation von Modellen, sein Ansatz bietet aber eine theoretische Grundlage für die zwei folgenden Ansätze. Der dem Ansatz von Ruys zu Grunde liegende Model-Checker, SPIN (s. [67]), wird normalerweise eingesetzt um an Hand verschiedener Automaten zu prüfen, ob Modelle vorher definierte Eigenschaften erfüllen. Um optimale Modelle mit dem Model-Checker zu finden, bedient sich Ruys eines klassischen Algorithmus aus der Optimierung, genannt "Branch-And-Bound" und implementiert diesen mittels eines Tricks in der dem Model-Checker zu Grunde liegenden Sprache "Promela". Der Algorithmus sucht eine optimale Lösung und kann dabei einige ungültige oder nicht optimale Lösungen ausschließen.

Basierend auf dieser Idee untersuchen **S. Varró-Gyapay und D. Varró** Optimierung und Erreichbarkeit (d.h. dass gewisse Graphen erzeugt werden können) von Graphentransformationen von Petri-Netzen mit Zeitangaben [60]. Es handelt sich dabei um eine Kombination der Überführung von Transformationsregeln in

SPIN und dem Ansatz von Ruys. Es kann exakt nur ein modelliertes Attribut (die "Zeit") zur Optimierung genutzt werden.

Wiederum aufbauend auf diesem Ansatz präsentieren **S. Varró-Gyapay und D. Varró** eine Idee, bei dem ein "optimaler" Graph erzeugt wird, indem bei einer Graphentransformation jede Regel mit einem Gewicht ausgestattet wird [163]. Bei jedem Feuern einer Regel wird das Gewicht der Regel zum Gesamtgewicht addiert. Es werden alle Gesamtergebnisse verschiedener Reihenfolgen von Ausführungen von Transformationsregeln bestimmt. Am Schluss wird das beste Ergebnis verwendet. Die Idee basiert wieder auf dem Model-Checker SPIN.

2.3.1.2 Abgrenzung

Der Ansatz von **T. Ruys** kann keine Modell-zu-Modell-Transformationen ausführen, sondern demonstriert wie mit Model-Checkern diskrete Optimierungsprobleme gelöst werden können. Dies wird von **S. Varró-Gyapay und D. Varró** aufgegriffen und um die Transformation von Petri-Netzen mit Zeitangaben erweitert. Dieser Ansatz ist daher auf bestimmte Graphen (Petri-Netze) beschränkt, was die Nutzung für allgemeine Modell-zu-Modell-Transformationen erheblich erschwert.

Die darauf aufbauende Idee, einen "optimalen" Graphen zu erhalten, indem die Transformationsregeln mit Gewichten ausgestattet werden, ermöglicht nicht das explizite Notieren der Zielfunktion. Beim Ansatz von **S. Varró-Gyapay und D. Varró** ist die Zielfunktion implizit durch die Gewichte der Transformationsregeln gegeben und hat keinen direkten Bezug zur Domäne des Zielmodells. Da ein optimales Zielmodell erzeugt werden soll, bestimmt die Domäne des Zielmodells aber für die meisten Anwendungsfälle maßgeblich die Zielfunktion. Dies ist z.B. für die Transformation von Modellen von Benutzerschnittstellen der Fall, bei denen z.B. das Layout des Zielmodells berechnet werden soll. Damit Entwickler einen besseren Bezug zur Domäne des Zielmodells und zur Zielfunktion herstellen können, muss die Zielfunktion also explizit für das Zielmodell notiert werden können. Trotzdem handelt es sich um den einzigen bekannten Ansatz zur Integration von Optimierung in Transformationssprachen.

2.3.1.3 Fazit

Da die Sprachen von Model-Checkern keine Transformationssprachen sind, sollte die in dieser Arbeit entwickelte Transformationssprache nicht auf diesen Sprachen aufbauen. Vielmehr sollte sie auf einer Modell-zu-Modell-Transformationssprache aufbauen und die explizite Notation von Zielfunktionen ermöglichen.

In Tabelle 2.4 werden die Ansätze verglichen. Die Bewertung ist sehr ähnlich da die Ansätze in diesem Abschnitt stark aufeinander aufbauen.

2.3.2 Fokus: Optimierung

Die beiden in diesem Unterabschnitt vorgestellten Ansätze verbinden ebenfalls Optimierung mit Transformationen. Allerdings haben diese Ansätze keinen Fokus auf Transformationen, sondern auf Optimierung und kombinieren diese sehr unterschiedlich mit Transformationen (s.u.). Der Ansatz von J. White bietet dabei die explizite Angabe von Zielfunktionen. Allerdings werden Optimierung und Constraint-Solving ausschließlich für den Prozess des Model Weaving eingesetzt. M. Kessentini hingegen zeigt, wie Optimierung zur Implementierung von Transformationen genutzt werden kann, aber es fehlt die Möglichkeit, die Zielfunktion anzugeben und der Ansatz ist auf Partikelschwarmoptimierung begrenzt.

2.3.2.1 Ansätze

J. White u.a. konzentrieren sich auf die Integration von Constraint-Solving und Optimierung in den Prozess des Weben von Modellen [165] (engl. "Model Weaving"). Weben von Modellen kann als eine Art Übertragung von Teilen aspektorientierter Programmierung auf modell-getriebene Softwareentwicklung verstanden werden. Dabei werden die Constraints aller zu webenden Modelle in einem Constraint-Solving-Problem kombiniert. Zum besseren Verständnis: aus Sicht von Modelltransformationen wären diese Modelle die Ausgangsmodelle (auch wenn Constraint-Solving in dieser Form im Kontext von Modelltransformationen nicht sinnvoll erscheint). Zum Finden einer optimalen Lösung kann eine Zielfunktion angegeben werden.

M. Kessentini u.a. betrachten den Prozess der Modelltransformation als ein Optimierungsproblem (MOTOE, [79]). Im Gegensatz zum klassischen Ansatz, Modelltransformationen in Sprachen zu notieren, werden bei diesem Ansatz keine Transformationsregeln definiert, sondern die Modelltransformationen werden aus Beispielen inferiert. Deshalb kann MOTOE als eine Spezialisierung der Programmierung durch das Geben von Beispielen (engl. "Programming by example") auf Modelltransformationen gesehen werden. Zur Erzeugung einer Transformation wird ein kombinatorisches Optimierungsproblem gelöst, welches auf Partikelschwarmoptimierung (s. [78]) basiert.

2.3.2.2 Abgrenzung

Die Idee des Ansatzes von **J. White**, eine Zielfunktion für den Constraint-Solving-Prozess zu verwenden, entspricht einem Grundsatz der vorliegenden Arbeit. Allerdings kann mit dem Ansatz von J. White nicht die Zielfunktion des Zielmodells angegeben werden, da der Prozess des Webens derzeit die Anwendung der Zielfunktion nur auf zu webende Modelle erlaubt.

Transformationen und Optimierung				
Fokus: Transformationen				
Ruys [139]	(✔)	✘	(✔)	✘
Varró-Gyapay [60]	(✔)	(✔)	(✔)	✘
Varró-Gyapay [163]	(✔)	✘	(✔)	✘
Fokus: Optimierung				
Motoe, Kessentini [79]	✔	✘	(✔)	✘
White [165]	(✔)	✔	✔	✘

Tabelle 2.4: Klassifikation verwandter Arbeiten, Transformationen und Optimierung

Der Ansatz von **M. Kessentini** verwendet Optimierung zur Erzeugung von Modell-zu-Modell-Transformationen. Der in der vorliegenden Arbeit geforderte Ansatz unterstützt hingegen die Deklaration von Optimierungsproblemen als Teil von Modell-zu-Modell-Transformationen. Im Gegensatz zu MOTOE kann daher klar ausgedrückt werden, wie zu erzeugende Zielmodelle aussehen sollen. Mittels MOTOE – als Spezialfall von Programmierung durch Beispiele – können Transformationen nur auf Grundlage der vorliegenden Beispiele definiert werden.

2.3.2.3 Fazit

Die zu entwickelnde Transformationssprache soll die explizite Deklaration von Zielfunktionen erlauben und Modell-zu-Modell-Transformationen nicht nur durch Beispiele zu definieren sein.

In Tabelle 2.4 werden die Ansätze mit denen aus dem vorigen Abschnitt verglichen.

2.4 Benutzerschnittstellen und Constraints

In diesem Abschnitt werden stellvertretend für eine große Zahl von Constraint-basierten Ansätzen exemplarisch einige ausgewählt, die Constraints zur Generierung von Benutzerschnittstellen verwenden (Unterabschnitt 2.4.1) oder Systeme, die das Lösen von Constraints ermöglichen (Unterabschnitt 2.4.2).

Eine empfehlenswerte, sehr umfassende Übersicht mit mehr als 200 älteren Referenzen geben W. Hower und W. Graf in [71]. Aus diesem Grund werden nur einige für diese Arbeit besonders interessante Ansätze betrachtet.

Tabelle 2.5 fasst die Bewertung der Ansätze zusammen.

2.4.1 Fokus: Benutzerschnittstellen

Bereits seit 1964 gibt es Ansätze, die Benutzerschnittstellen mit Hilfe von Constraints erzeugen und manipulieren [155]. Viele Anforderungen an Benutzerschnittstellen werden meist in Form von umgangssprachlich formulierten Bedingungen notiert (ähnlich zu Constraints). Daher bieten sich zur maschinenlesbaren Notation ebenfalls Constraints an, da sich diese dann für Menschen vermeintlich einfacher formalisieren lassen. Insbesondere wurde auch die Erzeugung von Benutzerschnittstellen und speziell des Layouts für Constraint-basierten Systeme intensiv untersucht. Die hier vorgestellten Ansätze beschränken sich auf einige neuere Ansätze.

2.4.1.1 Ansätze

Lutteroth u.a. präsentieren einen Ansatz basierend auf einem "mathematischen Modell", um das Layout einer grafischen Benutzeroberfläche festzulegen [98]. Dabei verwenden sie vordefinierte sog. Tabulatoren, anstatt sich auf ein klassisches Layout mit Tabellen zu verlassen. Die Tabulatoren legen Zwischenräume zwischen verschiedenen GUI Komponenten fest. Ein mathematisches Modell definiert die Abhängigkeiten zwischen den Tabulatoren.

Stellvertretend für die Klasse von Editoren, die das Editieren von grafischen Elementen mit Hilfe von Constraints vereinfachen, soll hier **QOCA** vorgestellt werden (weitere siehe [71]). QOCA ermöglicht die Definition von Constraint-Solving-Problemen mit einem grafischen Editor [100]. Der Nutzer kann im grafischen Editor grafische Elemente hinzufügen und editieren. Der Editor erzeugt dabei automatisch Constraints. Der Definition entsprechend kann die Anzeige im Editor als Graph und der Graph als Modell verstanden werden. Damit wird der Editor automatisch eine interaktive Applikation, die Constraint-Solving unterstützt. Das Modell ist nicht explizit vorhanden, sondern liegt nur als Instanz grafischer Elemente im Editor vor.

S. Feuerstack u.a. nutzen den von Badros entwickelten Constraint-Solver, der im nächsten Abschnitt vorgestellt wird [43]. Bei diesem Ansatz, genannt "Serco", werden die Modelle von Benutzerschnittstellen direkt interpretiert und nicht in Quellcode überführt. Transformationen werden auf konkreten Modellen ausgeführt, wenn ein Modell auf Grund geänderter Anforderungen durch neue Geräte, die Umgebung oder neue Nutzer interpretiert werden soll. Zur Erzeugung eines neuen Layouts für einen speziellen Fall werden automatisiert Constraints für den Constraint-Solver erzeugt.

2.4.1.2 Abgrenzung

Beim Ansatz von **Lutteroth** sind die Constraints auf lineare Constraints beschränkt. Nicht-lineare Constraints werden aber von der vorliegenden Arbeit ge-

fordert, da diese zur Generierung vieler Benutzerschnittstellen mit Constraints benötigt werden (s. [69]).

Bei **QOCA** handelt es sich – mangels Transformationen – nicht um Modell-zu-Modelltransformation, aber um Constraint-Solving von bestimmten Constraints in einem grafischen Modell. Mit dem in dieser Arbeit geforderten Ansatz sollen zusätzlich Constraints in Modelltransformationen gelöst werden können.

Die Sprache zur Notation der Modelle kann beim Ansatz von **S. Feuerstack** nicht geändert werden, da die Interpretation der Constraints fest an ein einziges Meta-Modell gebunden wurde. Es handelt sich also nicht um eine Modell-zu-Modell-Transformationssprache.

2.4.1.3 Fazit

Alle vorgestellten Ansätze definieren keine neue Modell-zu-Modell-Transformationssprache die Constraint-Solving und Optimierung unterstützt. Die Sprache dieser Arbeit soll dies unterstützen.

Zusätzlich sollen im Gegensatz zum Ansatz von **Lutteroth** nicht-lineare Constraints unterstützt werden.

Tabelle 2.5 bietet eine Übersicht über die Ansätze.

2.4.2 Fokus: Constraints

Einige Constraint-Solver bieten spezielle Schnittstellen an, mit denen sich besonders leicht Constraint-Solving-Probleme deklarieren lassen. Es existiert eine ganze Reihe von Ansätzen, s. [71]. Mit den beiden ersten in diesem Unterabschnitt präsentierten Ansätzen seien nur einige bekannte neuere genannt. Der letzte Ansatz dieses Abschnitts stellt die bereits alten Constraint Multiset Grammars vor, die aber als textueller Vorläufer von modernen Transformationssprachen zur Transformation von Benutzerschnittstellen betrachtet werden können und bereits Constraints implementierten.

2.4.2.1 Ansätze

G. Badros präsentiert einen auf Benutzerschnittstellen optimierten Constraint-Solver, der in verschiedenen Anwendungen zum Einsatz kommt [13]. Es handelt sich dabei um einen Constraint-Solver, der lineare, ganzzahlige Constraints unterstützt (d.h. Constraints, die ausschließlich lineare Terme mit Ganzzahlen verwenden). Es können theoretisch quadratische Terme als Zielfunktionen verwendet werden, da der Algorithmus eine Erweiterung des Simplex Algorithmus ist. Als Anwendungen schlägt Badros sowohl einen Constraint-Solver für Scheme-Benutzeroberflächen (Scheme Constraint Window Manager [15]), als auch eine

Erweiterung von Cascading Style Sheets mit Constraints vor (Constraint Cascading Style Sheets [14]). Die bereits erwähnten Ansätze ArtStudio (s.o.) und Serco (s.o.) verwenden ebenfalls diesen Constraint-Solver, ohne ihn in eine Transformationssprache einzubinden.

H. Hosobe erweitert den Ansatz um nicht-lineare Constraints [69]. Er beschreibt mögliche nicht-lineare Constraints, die dem Ansatz von G. Badros noch fehlen, baut aber auf der Syntax des Constraint-Solvers von Badros auf. Der Ansatz ermöglicht die Integration der neuen Constraints und die Berücksichtigung von Constraints, die nicht unbedingt erfüllt werden müssen ("Soft-Constraints"). Ein weiterer Ansatz für einen Constraint-Solver, dessen Fokus auf Benutzerschnittstellen liegt, ist das "Flexible Widget Layout" (s. [171]), ein Spezialfall eines Solvers zum Lösen von "Fuzzy Constraint Satisfaction Problems" (s. [138]).

"Picture Layout Grammars" erlauben die Definition von Abhängigkeiten zwischen Multisets, d.h. unsortierten Mengen von grafischen Objekten, um Vektorgrafiken zu erzeugen [56]. Die auf Picture Layout Grammars aufbauenden **"Constraint Multiset Grammars"** ermöglichen die Definition von grafischen Sprachen mittels einer textuell notierten Syntax. Die Arbeiten von **R. Helm u.a.** untersuchen die Verwendung von solchen Grammatiken zur Definition von grafischen Eingaben von Diagrammen für "Notepad-Computers", einer frühen Form des Tablet-PCs [63]. Die Arbeiten von **S. Chock und K. Marriott** erweitern diesen Ansatz und implementieren einen Parser für Diagramme und einen entsprechenden grafischen Editor [32].

2.4.2.2 Abgrenzung

Alle vorgestellten Ansätze akzeptieren keine grafischen Modelle als Eingaben und sind daher keine Modell-zu-Modell-Transformationssprachen. Sie unterstützen jedoch das geforderte Contraint-Solving. Manche der vorgestellten Ansätze unterstützen sogar Optimierung (teilweise, **[13], [69]**).

In gewisser Weise ähneln die **Ansätze basierend auf Multisets** textuell notierten Graph-Grammatiken und können damit als Vorläufer auch von der hier geforderten Sprache gesehen werden. Auch die Generierung von Editoren ist ein Merkmal moderner modell-getriebener Softwareentwicklung. Der Vorteil von Constraint Multiset Grammars besteht in der Verwendung von Constraints, die beim Parsen berücksichtigt werden. Moderne modell-getriebene Softwareentwicklung von grafischen Editoren berücksichtigt Constraints bisher nicht. Allerdings handelt es sich bei Constraint Multiset Grammars nicht um Modell-zu-Modell-Transformationssprachen, wie in Kapitel 1 gefordert.

Ansatz	Klasse	Transformationen	Constraints	Optimierung	Benutzerschnittstellen
Benutzerschnittstellen und Constraints					
	Fokus: Benutzerschnittstellen				
Sutherland [155]		✘	✔	✘	✔
Lutteroth [98]		✘	✔	✘	✔
QOCA, Marriott [100]		✘	✔	✘	✔
MASP, Feuerstack [43]		(✔)	(✔)	✘	✔
	Fokus: Constraints				
Cassowary, Badros [13]		✘	✔	(✔)	✔
Hosobe [69]		✘	✔	✘	✔
PLG, Golin [56]		✘	✔	✘	(✔)
CMG, Helm/Chok [32]		✘	✔	✘	✔

Tabelle 2.5: Klassifikation verwandter Arbeiten, Benutzerschnittstellen und Constraints

2.4.2.3 Fazit

Die vorgestellten Ansätze bilden die Grundlage für viele weitere Ansätze, die Constraint-Solving zur Generierung von Benutzerschnittstellen verwenden. Es fehlen allerdings notwendige Sprachkonstrukte zur effizienten Deklaration von Modell-zu-Modell-Transformationen. Diese sollen von der in dieser Arbeit geforderten Sprache unterstützt werden.

Tabelle 2.5 fasst die Bewertung der klassischen Systeme zusammen.

2.5 Benutzerschnittstellen und Optimierung

Wie bereits in der Einleitung vorgestellt, bietet die Kombination von Benutzerschnittstellen und Optimierung weitreichende Vorteile, da die Benutzerschnittstellen dann an Hand eines Kriteriums “optimal” erzeugt werden können. Aus diesem Grund werden die wenigen derzeit bekannten Arbeiten auf diesem Gebiet vorge-

stellt.

In Tabelle 2.6 wird die Bewertung der Ansätze zusammengefasst.

2.5.1 Fokus: Benutzerschnittstellen

2.5.1.1 Ansätze

F. Bodart u.a. stellen in [21] eine Bewertung von zwei verschiedenen Strategien vor, nach denen benutzerfreundliche grafische Benutzerschnittstellen entworfen werden können. Danach kann eine Benutzerschnittstelle einerseits über ein 2-Spalten-Layout berechnet werden oder die Positionierung der nächsten Komponente kann mittels Heuristiken entweder neben oder unterhalb der aktuell platzierten Komponente erfolgen. Um das Layout für die beiden Strategien zu berechnen, kommt jeweils eine Metrik zum Einsatz. Da die Metriken sehr leicht berechnet werden können, kann auch das komplette Layout durch einen Algorithmus mit geringer Komplexität berechnet werden.

GADGET ist ein Framework zur Optimierung von grafischen Benutzerschnittstellen. So zeigen die Autoren mehrere Anwendungen, die mit Hilfe des Frameworks entwickelt wurden [44]. Diese Beispiele reichen vom Setzen von Buchstaben bis zur Entwicklung einer kompletten Benutzerschnittstelle. Diese Benutzerschnittstelle wird auf das Layout und insbseondere auf die relative Positionierung hin optimiert, d.h. ob die nächste anzuzeigende Komponente unterhalb oder neben der aktuellen dargestellt werden soll. Diese Metrik wurde aus [22] entnommen und bereits im vorhergehenden Absatz vorgestellt. Die Autoren verbinden die beiden notwendigen Metriken mit einer Normierungsfunktion, um vergleichbare Werte für die Zielfunktion zu erhalten.

A. Sears definiert eine Metrik "a metric for layout appropriateness" und setzt dabei Varianten von Fitt's Law (s. [99]) ein [146]. Damit kann berechnet werden, wie "gut" eine Benutzeroberfläche entsprechend der Metrik ist. Optimierung kann genutzt werden, um an Hand eines Rasters Komponenten auf der Benutzeroberfläche zu positionieren.

K. Gajos betrachtet im Projekt **SUPPLE** das Problem, Benutzerschnittstellen für verschiedene Geräte vollautomatisch zu berechnen, als ein Optimierungsproblem [48]. K. Gajos berechnet für ein bestimmtes Gerät das zu verwendende Layout und die darzustellenden Komponenten anhand der Minimierung einer Zielfunktion automatisch aus einem "Modell" der Benutzerschnittstelle.

Die Geräte werden mit einer Menge von Constraints beschrieben, die die Eigenschaften und Möglichkeiten des Gerätes formalisieren. Diese Constraints werden bei der Berechnung einer möglichen Benutzerschnittstelle berücksichtigt. In Abhängigkeit der Geräte wird automatisch eine passende Zielfunktion ausgewählt.

Die Zielfunktion, die die Zeit abschätzt, die ein Benutzer benötigt, um die

Benutzerschnittstelle zu bedienen, berechnet sich an Hand der Wechsel zwischen Komponenten der Benutzerschnittstelle ("Weg durch die Applikation"). Sie ist abhängig von der Nutzung der Applikation und dem persönlichen Geschmack des Benutzers (bzw. seines persönlichen Verständnisses von Benutzbarkeit). Daher werden in SUPPLE die Wege der Nutzer durch die Applikation aufgezeichnet und daraus Rückschlüsse auf die zukünftige Nutzung der Applikation gezogen. Es wird angenommen, dass der Nutzer zukünftig denselben Weg wie vorher "gehen" wird.

ARNAULD erweitert den aus SUPPLE bekannten Ansatz um die automatische Generierung von Parametern für die Zielfunktionen, arbeitet aber weiter auf der Basis von SUPPLE [49]. Um die Zielfunktionen anzupassen, müssen von Benutzern verschiedene Dialoge verwendet werden. Die Nutzung der Dialoge erzeugt Parameter für die Zielfunktionen, womit dann die Benutzeroberflächen von möglichen Applikationen an die Nutzer spezifisch angepasst werden können.

SUPPLE++ erweitert den Ansatz aus SUPPLE, um die Möglichkeit mehrere (vordefinierte) Zielfunktionen zu verwenden [50]. Dabei werden die Zielfunktionen an die motorischen Fähigkeiten der Anwender (in diesem Fall: Anwender mit physischen Einschränkungen) angepasst. K. Gajos zeigt mit diesem Ansatz, wie gut dieser für verschiedene Benutzer funktioniert.

Eine Alternative zum Ansatz von K. Gajos wird durch **Y. Yang und R. Klemmer** verfolgt, die ebenfalls einen Ansatz zur automatischen Generierung von Benutzerschnittstellen wählen [172]. Dieser Ansatz basiert auf einer Zielfunktion, die auf der sog. Gestalt-Theorie aufbaut [31]. Die bei diesem Ansatz verwendeten "Modelle" basieren auf einem speziell für diesen Ansatz entwickelten XML-Dialekt.

2.5.1.2 Abgrenzung

Die von **F. Bodart** vorgestellten Strategien basieren nicht auf allgemein anwendbaren Gesetzen aus dem Gebiet der Mensch-Maschine-Interaktion (wie Fitt's Law). Sie wurden von den Autoren des Ansatzes schlicht als gültig angenommen, indem sie sie an einem Beispiel validierten. Dadurch sind die Metriken nicht unbedingt ohne Modifikationen auf andere Benutzerschnittstellen als im untersuchten Beispiel anwendbar; da im Ansatz keine Modell-zu-Modell-Transformationssprache zum Einsatz kommt, kann der Ansatz auch nicht direkt auf Modelle angewendet werden.

GADGET verwendet zwar Optimierung, um ganze Benutzerschnittstellen zu erzeugen, aber es wurden keine Modelle und auch keine Transformationssprache verwendet. Dadurch verbleibt die Deklaration der Optimierung im Quellcode und es ist nicht möglich, Transformationen mittels einer dafür speziell entwickelten Sprache zu deklarieren, was die Notation vereinfachen würde.

"A metric for layout appropriateness" benutzt keine Modelle und keine Modell-zu-Modell-Transformationssprache. Der Ansatz ist wegweisend für die vorliegende

Arbeit, weil es der erste Ansatz ist, der Optimierung für Benutzerschnittstellen einsetzte. Die Zielfunktionen und Constraints können bei **A. Sears** nicht einfach ausgetauscht werden, da das System direkt in Code implementiert ist. Die Grundsätze der vorliegenden Arbeit erfordern hingegen die Möglichkeit der Deklaration von Zielfunktionen und Constraints durch Nutzung einer Modell-zu-Modell-Transformationssprache.

SUPPLE erfüllt den Grundsatz, Optimierung von Benutzerschnittstellen zu unterstützen, benutzt aber keine Modell-zu-Modell-Transformation. Die Zielfunktion berücksichtigt einerseits die Möglichkeiten der Geräte als auch eine Approximation der Zeit, die ein Benutzer zur Nutzung der Benutzerschnittstelle vermutlich benötigen könnte. SUPPLE verwendet keine allgemeine Modell-zu-Modell-Transformationssprache (sondern JAVA), und die Zielfunktion ist hart kodiert.

In **ARNAULD** werden zwar keine hart kodierten Parameter für die Zielfunktion mehr verwendet, aber es wird wiederum keine Modell-zu-Modell-Transformationssprache verwendet.

SUPPLE++ motiviert den Grundsatz der vorliegenden Arbeit, es zu ermöglichen, Zielfunktionen frei in Modelltransformationen zu deklarieren. Damit können diese mit noch einfacher an die Gegebenheiten der Benutzer angepasst werden, indem eine einzige Zeile Code getauscht wird. Der Ansatz von K. Gajos erlaubt hingegen nur die Nutzung einer Menge von vordefinierten Zielfunktionen (oder die neuen Zielfunktionen müssen in JAVA implementiert werden und können nicht deklarativ notiert werden) und verwendet keine Modell-zu-Modell-Transformationssprache.

Beim Ansatz von **Y. Yang und R. Klemmer** handelt sich wiederum nicht um eine Modell-zu-Modell-Transformationssprache, sondern es wird eine XML-Datei direkt interpretiert. Dies bedeutet übertragen auf Meta-Modelle, dass das Meta-Modell nicht frei gewählt werden kann, und somit nur das Meta-Modell von Y. Yang und R. Klemmer unterstützt wird.

2.5.1.3 Fazit

Die von **F. Bodart** entwickelten Metriken stellen ein Beispiel dafür dar, wie Layoutalgorithmen für Optimierungsprobleme entstehen können – durch das Treffen von Annahmen über von Nutzern akzeptierte Layouts (idealerweise durch Studien unterstützt). Die in dieser Arbeit geforderte Sprache soll daher die Deklaration von Zielfunktionen unterstützen, die es erlauben, solche Metriken sehr leicht zu verändern und in ähnlicher Weise zu erstellen.

SUPPLE++ und der Ansatz von **Y. Yang und R. Klemmer** erlauben mehrere Zielfunktionen. Daraus kann geschlossen werden, dass es günstig ist, möglichst viele Zielfunktionen zu unterstützen. Dies ist ein weiteres Argument für den Grundsatz der vorliegenden Arbeit, dass Zielfunktionen leicht veränderbar sein müssen.

Tabelle 2.5 bietet eine Übersicht über die Bewertung der Ansätze.

Ansatz	Klasse	Transformationen	Constraints	Optimierung	Benutzerschnittstellen
Benutzerschnittstellen und Optimierung					
Fokus: Benutzerschnittstellen					
Bodart [22]		✘	✘	(✔)	✔
Gadget, Fogarty [44]		✘	✘	(✔)	✔
Sears [146]		✘	✘	(✔)	✔
SUPPLE, Gajos [48]		✘	(✔)	(✔)	✔
ARNAULD, Gajos [49]		✘	(✔)	(✔)	✔
SUPPLE++, Gajos [50]		✘	(✔)	(✔)	✔
Yang [172]		✘	(✔)	(✔)	✔

Tabelle 2.6: Klassifikation verwandter Arbeiten, Benutzerschnittstellen und Optimierung

2.5.2 Fokus: Optimierung

Ansätze, die auf allgemeine Optimierung und Benutzerschnittstellen durch allgemeine Verfahren der Optimierung implementieren, sind außer der vorliegenden Arbeit derzeit keine bekannt.

2.6 Zusammenfassung

Keiner der untersuchten Ansätze unterstützt nach den Kriterien aus Tabelle 2.1 alle vier Grundsätze voll (entspricht vier ✔), d.h. dass insgesamt auch kein Ansatz der geforderten Transformationssprache entsprechen kann. Vielmehr verbleibt jeder Ansatz mit mindestens einem Grundsatz, den er nicht unterstützt (entspricht ✘) und einer Schwäche (entspricht (✔); Ausnahme ist Transformationen und Optimierung, Fokus: Optimierung). Dadurch können auch die allgemeinen Ansätze – die z.B. theoretisch auf Benutzerschnittstellen ausgeweitet werden könnten – nicht verwendet werden, da sie zusätzlich mindestens einen Grundsatz nicht voll unterstützen.

Fokus	Klasse	Transformationen	Constraints	Optimierung	Benutzerschnittstellen
Transformationen und Constraints					
Fokus: Transformationen		✔	(✔)	✘	✘
Fokus: Constraints		✔	✔	✘	✘
Transformationen und Benutzerschnittstellen					
Fokus: Transformationen		✔	(✔)	✘	✔
Fokus: Benutzerschnittstellen		✔	(✔)	✘	✔
Transformationen und Optimierung					
Fokus: Transformationen		(✔)	(✔)	(✔)	✘
Fokus: Optimierung		✔	✔	✔	✘
Benutzerschnittstellen und Constraints					
Fokus: Benutzerschnittstellen		(✔)	✔	✘	✔
Fokus: Constraints		✘	✔	(✔)	✔
Benutzerschnittstellen und Optimierung					
Fokus: Transformationen		✘	(✔)	(✔)	✔

Tabelle 2.7: Klassifikation verwandter Arbeiten, Übersicht

Tabelle 2.7 fasst die untersuchten Klassen (Fokus) von verwandten Arbeiten zusammen. Die Zusammenfassung zeigt in jeder Zeile und Spalte die "'beste" Qualität einer jeden Klasse (Fokus) von Ansätzen. Aufbauend auf den erarbeiteten Ergebnissen in den Fazits 2.1.1.3, 2.1.2.3, 2.2.1.3, 2.2.2.3, 2.3.1.3, 2.3.2.3, 2.4.1.3, 2.4.2.3 und 2.5.1.3 fasst Tabelle 2.8 Anforderungen zusammen, die sich aus den Ergebnissen ergeben. Diese werden dann in Kapitel 3 aufgegriffen und ein Konzept für Modell-zu-Modell-Transformationssprachen zur Transformation von Modellen von Benutzerschnittstellen vorgelegt.

2.6.1 Fazit

Kein Ansatz erreicht alle Ziele. Aus diesem Grund muss eine neue Transformationssprache entwickelt werden. Die Transformationssprache muss die Grundsätze (s. Kapitel 1) und die identifizierten Anforderungen (s. Tabelle 2.8) erreichen. Die Tabelle wiederholt das Fazit aus den vorigen Abschnitten und weist diesen Anforderungen zu. Zusätzlich erhalten die Anforderungen eine ID zum einfachen Verweis. Die Anforderungen werden in den Kapiteln 3 und 4 sortiert und verfeinert. Deshalb wird in der letzten Spalte die Nummer des entsprechenden Abschnitts angegeben, indem auf die Anforderung verwiesen wird ("Tracking").

Fazit	Anforderungen	ID	Track.
Eine Modell-zu-Modell-Transformationssprache zur Transformation von Modellen von Benutzerschnittellen sollte die Definition von Constraints direkt in der Sprache erlauben und sollte implementierbar sein.	Unterstützung von Constraints in der Sprache; Sprache muss maschinell zu interpretieren sein	A1	3.1.3.4
Ansätze zur Transformation von Modellen von Benutzerschnittstellen sollten daher sog. Constraint-Propagation unterstützen, die unnötiges Ausprobieren verhindert.	Interpreter von Sprachen mit Constraints sollten Constraint-Propagation unterstützen	A2	4.3
Aktuelle Ansätze basierend auf Graph-Grammatiken sind daher derzeit nicht als Grundlage für diese Arbeit geeignet.	Der Ansatz in dieser Arbeit sollte nicht auf aktuellen Graph-Grammatiken fußen	A3	3.1.3.3
In der vorliegenden Arbeit soll daher die Vorgehensweise von J. Cabot als möglicher Weg zur Interpretation von Sprachen Berücksichtigung finden.	Zur Implementierung eines Interpreters für vorgeschlagene Sprachen könnten logische Programmiersprachen genutzt werden	A4	4.2
Die hier geforderte Transformationssprache muss kontextsensitiv sein und soll sich nicht auf das Layout von Graphen beschränken.	keine Beschränkung auf das Layout von Graphen; kontextsensitive Sprache	A5	3.1.3.3

Die geforderte Sprache soll daher frei wählbare Constraints unterstützen.	Constraints sollen nicht vordefiniert sein, sondern sollen von Entwicklern von Transformationen frei definiert werden können	A6	3.1.3.4
ArtStudio – der einzige Ansatz der einen Constraint-Solver einsetzt – motiviert, dass der Constraint-Solving-Prozess nicht als ein getrennter Schritt durchgeführt wird, sondern mit der Transformation integriert sein soll.	Constraint-Solving sollte Bestandteil des Prozesses der Transformation sein	A7	3.1.3.5
Da die Sprachen von Model-Checkern keine Transformationssprachen sind, sollte die in dieser Arbeit entwickelte Transformationssprache nicht auf diesen Sprachen aufbauen.	Die Modell-zu-Modell-Transformationssprache sollte nicht auf Sprachen zur Spezifikation von Eigenschaften für Model-Checker aufbauen, da diese keine Transformationssprachen sind	A8	3.1.3.3
Vielmehr sollte sie auf einer Modell-zu-Modell-Transformationssprache aufbauen und die explizite Notation von Zielfunktionen ermöglichen.	Die Modell-zu-Modell-Transformationssprache sollte als Sprachkonstrukt die Deklaration von Zielfunktionen erlauben	A9	3.1.4.1
Die zu entwickelnde Transformationssprache soll die explizite Deklaration von Zielfunktionen erlauben und Modell-zu-Modell-Transformationen nicht wie im Fall von MOTOE nur durch Beispiele zu definieren sein.	Transformationen sollten nicht durch Programmierung auf Grund von Beispielen entwickelt werden; Zielfunktionen sollten explizit notiert werden	A10	3.1.4.1
Zusätzlich sollen im Gegensatz zum Ansatz von Lutteroth nicht-lineare Constraints unterstützt werden.	Es sollen nicht-lineare Constraints unterstützt werden	A11	3.1.3.4

Die in dieser Arbeit geforderte Sprache soll daher die Deklaration von Zielfunktionen unterstützen, die es erlauben, solche Metriken sehr leicht zu verändern und in ähnlicher Weise zu erstellen.	Zielfunktionen sollten leicht veränderbar sein	A12	3.1.4.1

Tabelle 2.8: Anforderungen aus verwandten Arbeiten

Kapitel 3

Entwicklung theoretischer Grundlagen und Konzepte

Überblick

In dieser Arbeit wird eine neue, deklarative Transformationssprache zur Transformation von Modellen von Benutzerschnittstellen deklariert. Dabei bietet die Einbettung von Constraint-Solving und Optimierung in die Sprache (s. Kapitel 1) neue Möglichkeiten bei der Deklaration von Modell-zu-Modell-Transformationen, die es z.B. erlauben, das Layout von Benutzerschnittstellen in Transformationen zu berechnen. Als Verfeinerung der Grundsätze und groben Anforderungen, die bereits in den Kapiteln 1 und 2 aufgestellt wurden, werden in diesem Kapitel Konzepte formuliert, die zur Festlegung einer Transformationssprache verwendet werden können.

Abbildung 3.1 soll die Einordnung des Kapitels in den gesamten Rahmen dieser Arbeit verdeutlichen. In Kapitel 1 wurden Begriffe definiert, die in diesem Kapitel verwendet werden. Zur Erfüllung der bereits präsentierten Anforderungen aus Kapitel 2 soll in diesem Kapitel ein neuartiges Konzept für Transformationssprachen zur Transformation von Modellen von Benutzerschnittstellen erarbeitet werden. In Kapitel 4 wird das Konzept beispielhaft in einer konkreten Transformationssprache umgesetzt.

Das Konzept ist wiederum in zwei Teile aufgeteilt: einem Teil, der sich mit der Herleitung von Anforderungen und Eigenschaften für Transformationssprachen befasst und einem weiteren, der sich mit der Spezialisierung des Konzepts auf die Transformation von Modellen von Benutzerschnittellen beschäftigt. Diese Aufteilung soll dann auch auf Kapitel 4 übertragen werden: Zuerst wird aufbauend auf dem ersten Teil eine konkrete Transformationssprache entwickelt. Im zweiten Teil wird die Sprache genutzt, um Transformationen, Methodik und Strategien

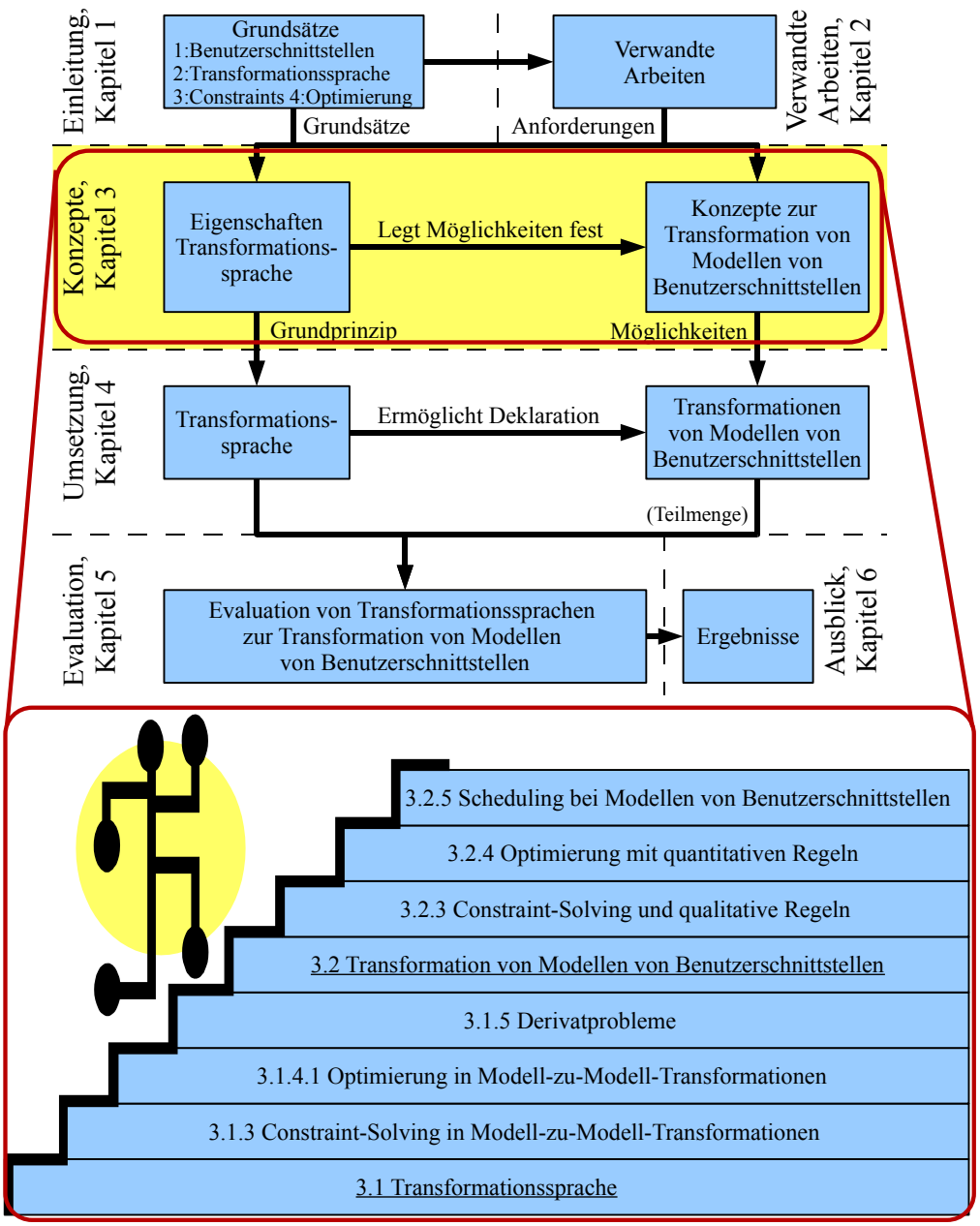


Abbildung 3.1: Aufbau Dissertation, Kapitel 3 hervorgehoben

zur Generierung von Modellen von Benutzerschnittstellen umzusetzen. Dabei wird das Konzept stufenweise aufgebaut, s. Abbildung 3.1. Aufbauend auf der Theorie der Verknüpfung von Constraint-Solving mit Transformationssprachen können Optimierung und Derivatprobleme in Transformationen eingeführt werden.

3.1 Transformationssprache

Modell-zu-Modell-Transformation handhabt Modelle auf der Basis spezieller Graphen. Meist wird mit den Modellen Software modelliert (im MDA guide [118] als "Systeme" bezeichnet), insofern kann modell-getriebene Entwicklung von Software und damit Modell-zu-Modell-Transformation als Teilgebiet des Software Engineering betrachtet werden.

Constraint-Solving und Optimierung kann dagegen so interpretiert werden, dass Probleme als Suche in einem Lösungsraum formuliert werden; Constraints bzw. Randbedingungen beschränken diesen Lösungsraum idealerweise letztlich so, dass nur noch Problemlösungen übrig bleiben. Constraint-Solving und Optimierung sind Teilgebiete der deklarativen Programmierung.

"Modell-zu-Modell-Transformation" und "Constraint-Solving und Optimierung" werden in der Informatik daher als disjunkte Teilgebiete betrachtet.

Zur besseren Unterscheidung sei der Unterschied zwischen Modell-zu-Modell-Transformation und Constraint-Solving/Optimierung wie folgt interpretiert:

- bei exogenen Modell-zu-Modell-Transformationen (s. Seite 20) führt eine Transformation zum Übergang von Wörtern einer Ausgangssprache in Wörter einer Zielsprache.

- bei Constraint-Solving und Optimierung mit Randbedingungen wird dagegen im Allgemeinen *eine* gegebene Sprache (sukzessiv) durch Constraints beschränkt; der Begriff "Sprache" ist hier weit gefasst, beispielsweise werden auch $\mathbb{N}$ (also die Menge der natürlichen Zahlen) und "Vielfache von 5" dieser Interpretation gerecht. Die (sukzessive) Beschränkung einer Sprache kann aber immer nur Teil dieser Sprache sein, womit es sich *immer* nur um endogene Transformationen (s. Seite 20) handelt.

Dadurch können im Fall von Constraint-Solving oder Optimierung keine exogenen Modell-zu-Modell-Transformationen vorliegen. Exogene Transformationen transformieren von einer Quell-Sprache in eine Ziel-Sprache, die nicht Teil der Quell-Sprache ist.

Zur Transformation von Modellen von Benutzerschnittstellen ist es jedoch von Vorteil, exogene Sprachtransformationen in Kombination mit Constraint-Solving oder Optimierung auszuführen, um deklarative Programmierung (d.h. die

Programmierung mit Constraints und Zielfunktionen) in Transformationen auszuführen. Als Beispiel für die Verwendung von Constraints sei die Berechnung der Position von Komponenten von Benutzerschnittstellen genannt, die bekanntlich (z.B. [155, 13, 43, 100, 140]) durch Constraints angegeben werden kann. Nach B. Myers bieten Constraints "einfache", "deklarative" Beschreibungen für verschiedene Aspekte von Benutzerschnittstellen (z.B. Layout) [105]. Sollen also z.B. Layouts in der Modell-zu-Modell-Transformation berechnet werden, ist es daher vorteilhaft, die Layouts anhand von "einfachen" Beschreibungen, d.h. z.B. Constraints, zu beschreiben. Da verschiedene Modell-zu-Modell-Transformationssprachen auch deklarativ sind, bietet es sich an, Constraints als native Beschreibungselemente in die Modell-zu-Modell-Transformationssprache zu integrieren. Soll bspw. das Layout nach einem oder mehreren Kriterien optimiert werden, so kann auch Optimierung in Transformationen integriert werden. Als Erweiterung zu den Ansätzen zur automatischen Optimierung von Benutzerschnittstellen (z.B. [146, 44, 47]) wird es durch die Integration erstmals möglich, Modelle von Benutzerschnittstellen mit Hilfe von Transformationen, die in einer Transformationssprache geschrieben wurden, zu optimieren.

Infolgedessen wird in diesem Kapitel vorgeschlagen, Constraint-Solving und Optimierung mit einer deklarativen (relationalen, s.u.) Modell-zu-Modell-Transformationssprache zu verbinden.

3.1.1 Herleitung von Konzepten

In diesem Kapitel werden Konzepte entwickelt, anhand derer sich Transformationssprachen deklarieren lassen, die Constraint-Solving und Zielfunktionen allgemein unterstützen. Aus diesen Konzepten lassen sich dann Transformationssprachen herleiten, mit denen dann Transformationen implementiert werden können. Eine solche Sprache und einige beispielhafte Transformationen werden in dieser Arbeit in Kapitel 4 definiert.

3.1.1.1 Iterative Herleitung der Konzepte

Im weiteren Verlauf des Kapitels wird zuerst ein Konzept für die Integration von Constraint-Solving mit Transformationen entwickelt, s. Abschnitt 3.1.3. Ein darauf aufbauendes Konzept integriert dann die Optimierung mit Transformationen, s. Abschnitt 3.1.4.1. Schließlich wird das Konzept um die Möglichkeit erweitert, verschiedene alternative Klassen für ein erzeugtes Modellelement des Zielmodells zuzulassen, s. Abschnitt 3.1.5.

3.1.2 Constraint-Solving

In diesem Abschnitt wird Constraint-Solving mit einer Klasse von deklarativen Modell-zu-Modell-Transformationssprachen kombiniert. Diese Kombination trägt den Namen "Constraint-relationale Transformationen".

Zum besseren Verständnis – insbesondere der Algorithmen zur Implementierung von Constraint-relationalen Transformationen – werden zuerst Constraints, die auf Modellen definiert wurden, auf klassische Constraint-Solving-Probleme abgebildet, d.h. auf Probleme, die nicht mit Modellen aus dem Software Engineering implementiert wurden. Eine Abbildung von Constraints auf Modellen auf klassische Constraint-Solving-Probleme wurde z.B. von J. Cabot u.a. entwickelt [28] und wird hier vereinfacht wiedergegeben und verallgemeinert.

3.1.2.1 Symbole für Modellelementen

Zunächst seien einige informelle Definitionen von mathematischen Symbolen gegeben, die in der Definition von Constraint-relationalen-Transformationen genutzt werden:

Symbol	Erklärung
V	Menge von Variablen eines Constraint-Solving-Problems
C	Menge der Constraints, die die Variablen V eines Constraint-Solving-Problems beschränken
$A(m)$	Menge der Attribute eines Modellelements m
$C_{local}(m)$	Menge der Constraints, die lokal zu m sind
M	Menge aller Modellelemente eines Modells

Tabelle 3.1: Zusammenfassung der mathematischen Symbole in Kapitel 3

Jedes **Modellelement** m aus der Menge aller Modellelemente besitzt eine Menge von Constraints $C_{local}(m)$ sowie eine **Menge von Attributen** $A(m)$. Assoziationen verbinden Modellelemente wie (gerichtete) Kanten in einem Graph. Constraints beschränken nahezu beliebige Eigenschaften von Modellen.

Constraints $C_{local}(m)$ werden hier als "lokal zu m" bezeichnet, wenn die Constraints an Modellelement m gebunden sind. Hier sei dies so definiert, dass die

Existenz der Constraints vom Modellelement m abhängt und der Namensbereich[1] für Variablen und Assoziationen der Constraints relativ zu m interpretiert wird. Das impliziert aber nicht, dass die Constraints nur auf Attributen von m definiert sind, da in Constraints auch Assoziationen verwendet werden können. Dadurch können Constraints auch Attribute anderer Modellelemente beschränken, festlegen, vergleichen oder zu anderen in Relation setzen. Indem Assoziationen zu anderen Modellelementen genutzt werden, werden ausgehend vom Namensbereich des aktuellen Modellelements die Namensbereiche anderer Modellelemente erreicht. Normalerweise werden in der Modellierung (z.B. UML) nur lokale Constraints verwendet, sodass sich die weitere Betrachtung allein auf die hier beschriebenen lokalen Constraints beschränken kann. Diese Vorgehensweise ist verbreitet und findet z.B. auch breite Akzeptanz in der Anwendung von UML. Sie hat den Vorteil, dass keine Constraints berücksichtigt werden müssen, die einen globalen Namensbereich benötigen und damit nur sehr umständlich spezielle Modellelemente beschränken können.

Wie bereits in Kapitel 1 erläutert, werden in der Modellierung Constraints meist in Meta-Modellen deklariert. In Meta-Modellen können Constraints für Klassen (im Meta-Modell deklarierte Typen von Modellelementen) festgelegt werden. Abhängig von der Klasse gelten dann die jeweiligen Constraints bei der Erzeugung eines neuen Modellelements im Modell. Sie sind nach obiger Definition damit lokal zu den jeweiligen Modellelementen.

3.1.2.2 Constraint-Solving-Probleme auf Modellen

Die vorliegende Arbeit stützt sich für die Definition eines Constraint-Solving-Problems auf die gängige Definition aus [160] und verwendet zur Erläuterung die bereits eingeführten Symbole: Ein **Constraint-Solving-Problem** besteht klassischerweise aus einer Menge von Variablen V und einer Menge von Constraints C auf V. Jede Variable hat einen definierten Typ (z.B. natürliche Zahl oder Fließkommazahl) und einen Wertebereich, der von den Constraints definiert wird. Sofern mindestens eine Lösung existiert, können **Constraint-Solver** meist eine Belegung von V finden so, dass alle Constraints halten. Um eine Lösung zu finden wird von vielen Solvern eine komplexe Suche durchgeführt. Solver sind daher Algorithmen zur Lösung von Constraint-Solving-Problemen.

Sind in einem Modell Constraints deklariert worden, so handelt es sich um ein

[1] Namensbereich: Ausgehend von einem Modellelement können Attribute und Assoziationen referenziert werden, indem deren Bezeichner verwendet werden. Beispiel: Modellelement a und b sind durch eine Assoziation c verbunden (bidirektional, in beiden Modellelementen gleich bezeichnet). a habe ein Attribut $a1$, b habe $b1$. Dann kann im Namensbereich von a $b1$ je nach Notation z.B. durch $c.b1$ referenziert werden, während $a1$ im Namensbereich von b mit $c.a1$ referenziert werden kann. Diese Art Namensbereich wird in der Sprache OCL als “context” bezeichnet.

"Constraint-Solving-Problem auf Modellen". Diese Probleme beschränken sich im Allgemeinen nicht nur auf Variablen sondern auf alle möglichen Modellelemente, Attribute und Assoziationen. Um bereits existierende Solver zur Lösung dieser Probleme zu nutzen, können diese als klassische Constraint-Solving-Probleme mit Variablen (V) und Constraints (C) dargestellt werden:

Jedes Modellelement erhält eine ID. Die Klasse (also das Meta-Modellelement), die den Typ des Modellelements bestimmt, wird als Attribut (also als Element von $A(m)$) betrachtet. Assoziationen werden als Attribute (also als Element von $A(m)$) betrachtet, die auf die IDs der assoziierten Modellelemente verweisen.

Die Attribute eines Modellelements können als Variablen im Constraint-Solving-Problem dargestellt werden, sind damit also Teilmenge von V, sodass gilt $A(m) \subseteq V$. Unter der Voraussetzung, dass keine Constraints betrachtet werden, die nicht im Modell notiert werden, ist V dann die Vereinigung aller $A(m)$ (also letztlich aller im Modell vorkommenden Attribute). Werden die Attribute in Meta-Modellen deklariert, so werden für jedes neu erzeugte Modellelement die entsprechenden Variablen (also eine neue Menge $A(m)$) erzeugt. Zur besseren Unterscheidung werden sie unterschiedlich benannt (z.B. gödelisiert).

Constraints können ebenso transformiert werden. Werden keine Meta-Modelle verwendet können die Constraints einfach verwendet werden. Im Fall, dass Constraints im Meta-Modell definiert werden, werden Constraints pro Modellelement erzeugt, entsprechend der Klasse aus dem Meta-Modell. Hierbei werden die Attribute, die in einem beliebigen Constraint $C_{local}(m)$ verwendet werden, durch die im vorigen Schritt erzeugten Variablen $A(m)$ substituiert und das neue Constraint dem Constraint-Solving-Problem hinzugefügt, sodass gilt $C_{local}(m) \subseteq C$. Da nur lokale Constraints berücksichtigt werden, ist die Gesamtmenge aller Constraints im erzeugten Constraint-Solving-Problem die Menge aller neu erzeugten lokalen Constraints.

Die so erzeugten lokalen Constraints und Variablen ergeben wieder ein klassisches Constraint-Solving-Problem, welches mit einem Solver gelöst werden kann.

3.1.2.3 Erweiterungen klassischer Constraint-Solving-Probleme

Im Fall der Verwendung von Constraints in Meta-Modellen ergeben sich folgende zusätzlichen Eigenschaften im Vergleich mit klassischen Constraint-Solving-Problemen:

- Constraints in Constraint-Solving-Problemen auf Modellen können auch über Assoziationen hinweg verwendet werden.

- Die Menge der Variablen ist durch die Attribute der Modellelemente gegeben und steht nicht bei Deklaration des Constraint-Solving-Problems fest, da oft

eine beliebige Anzahl von Modellen aus dem Meta-Modell erzeugt werden kann.

- Analog dazu steht die Menge der Constraints erst zum Zeitpunkt der Modellierung des Modells fest, da für die Modellelemente jeweils Constraints erzeugt werden. Die Constraints sind dann lokal zu den jeweiligen Modellelementen.

Die folgende Definition ist eine Übersetzung der Definition von Constraint-Solving-Problemen auf UML-Klassendiagrammen aus [28] in den allgemeineren Fall der Modellierung mit Meta-Modellen:

Begriffsdefinition 3.1. *Ein* **Constraint-Solving-Problem auf Modellen** *ist ein Constraint-Solving-Problem, bei dem die Menge der Variablen des Constraint-Solving-Problems gegeben ist durch* $V = \{\forall m \in M | A(m) \subseteq V\}$ *und die Menge der Constraints gegeben ist durch* $C = \{\forall m \in M | C_{local}(m) \subseteq C\}$.

Sofern mindestens eine Lösung zu einem Constraint-Solving-Problem auf Modellen existiert, versucht ein dafür entwickelter Solver eine Belegung der Attribute zu finden so, dass alle lokalen Constraints im Modell erfüllt sind. Constraint-Solving-Probleme auf Modellen können, wie klassische Constraint-Solving-Probleme auch (s. [16]), entweder

- *überspezifiziert* sein, d.h. es existiert kein Modell, welches die Constraints erfüllt
- oder *genau spezifiziert* sein, d.h. es existiert genau ein Modell, welches die Constraints erfüllt
- oder *unterspezifiziert* sein, d.h. es existieren mehrere Modelle, welche die Constraints erfüllen.

3.1.3 Constraint-Solving in Modell-zu-Modell-Transformationen

Eine Modell-zu-Modell-Transformation transformiert eine Menge von Ausgangsmodellen in ein oder mehrere Zielmodelle. Wie bereits erwähnt, können Constraints auch in Modell-zu-Modell-Transformationen verwendet werden. Dabei können die Constraints zwei Aufgaben erfüllen:

1. Sie können zur Selektion von Attributen oder Modellelementen im Ausgangsmodell dienen.
2. Sie können zur Beschränkung von Attributen oder Modellelementen im Zielmodell genutzt werden.

3.1.3.1 Selektion

Die erste Aufgabe kann durch zwei verschiedene Lösungen implementiert werden. Normalerweise werden Constraints zur Selektion so verwendet, dass mit einem Algorithmus alle Modellelemente selektiert werden, bei denen die Constraints erfüllt sind. Im Fall der zweiten Lösung wird die Selektion auf ein Constraint-Solving-Problem abgebildet, und kann so mittels eines Constraint-Solver etwas effizienter durchgeführt werden. Daher kann die erste Aufgabe zwar durch Constraint-Solving gelöst werden (eine Implementierung wird in [137] beschrieben), aber in klassischen Ansätzen werden die Constraints auf den Modellelementen nur geprüft und dann das Element selektiert. Beide Methoden haben unterschiedliche Effizienz bei der Selektion von Modellelementen und bieten gegenseitig keine weiteren Funktionen zur Transformation an. Daher kann sowohl der klassische Ansatz, wie auch der zur Selektion mit Constraint-Solving verwendet werden – ohne den Funktionsumfang einzuschränken. Für die prototypische Implementierung des Interpreters der in dieser Arbeit entwickelten Transformationssprache wurde der klassische Ansatz verfolgt.

3.1.3.2 Beschränkung von Attributen in Zielmodellen

Die zweite Aufgabe ist ein Constraint-Solving-Problem, da dort die Werte der Attribute unter Beschränkungen gesetzt werden sollen. Aus Kapitel 2 ist bereits bekannt, dass zur Lösung der zweiten Aufgabe keine deklarative Modell-zu-Modell-Transformationssprache und Implementierung (mit Optimierung, s. 3.1.4.1) in der Literatur existiert. Also wurde im Rahmen dieser Arbeit ein Konzept und eine Sprache entwickelt. Auf der bis zu diesem Punkt diskutierten Ebene ist die Darstellung der Transformationen aber zu abstrakt, um eine Modell-zu-Modell-Transformationssprache zu definieren. Zuerst muss eine Familie von Transformationssprachen gewählt werden.

3.1.3.3 Entwurfsentscheidung: Relationale Transformationssprachen

Um einen rein deklarativen Ansatz von Modell-zu-Modell-Transformationen zu erhalten, wurden in dieser Arbeit relationale Transformationssprachen ausgewählt. Da sie deklarativ sind, vereinfacht sich die Integration mit den Konzepten des Constraint-Solving und der Optimierung (s. Kapitel 1). Dieser Ansatz wird im nächsten Abschnitt konkretisiert. Ein alternativer Ansatz dazu wären Graph-Transformationssprachen, die ebenfalls deklarativ und mit Constraints verwendet werden können. Dies widerspricht aber der Herleitung von Anforderung A3 aus Kapitel 2 (d.h. keine aktuelle Graph-Grammatik). Mit einer relationalen Sprache werden zusätzlich die Anforderungen A5 (kontextsensitiv) und A8 (keine Model-Checker) umgesetzt.

Mit QVT Relations [117] liegt erstmals eine standardisierte Transformationssprache vor, welche deklarative Konzepte implementiert. Es handelt sich um eine relationale Transformationssprache, sie ist also nicht regelbasiert. Im Hinblick auf die Verwendbarkeit der vorgestellten Konzepte in Kombination mit der Transformationssprache aus dem Standard sind daher relationale Sprachen besonders interessant, und in den folgenden Abschnitten wird die Kombination dieser Sprachfamilie mit Constraint-Solving und Optimierung untersucht.

Sprachen die diese Kombination unterstützen, erreichen Grundsätze 2 und 3 aus Kapitel 1, da es sich um Transformationssprachen handelt und andererseits Constraint-Solving-Probleme deklariert werden können.

3.1.3.4 Constraint-relationale-Transformationen

Eine **relationale Modell-zu-Modell-Transformation** transformiert Ausgangsmodelle mittels einer Anzahl von Relationen in ein Zielmodell. Eine **Relation** erzeugt aus einer Menge von Modellelementen der Ausgangsmodelle eine durch die Relation festgelegte Menge von Modellelementen des Zielmodells, indem sie diese Modellelemente miteinander in Beziehung setzt[2]. Die Modellelemente entsprechen sowohl in den Ausgangsmodellen als auch im Zielmodell einem sog. **Muster**. Im Ausgangsmodell wird das Muster dazu verwendet, zur Transformation geeignete Modellelemente auszusuchen (die Modellelemente müssen also den im Muster deklarierten Eigenschaften entsprechen), während im Zielmodell die Elemente dem Muster entsprechend erzeugt werden. Insgesamt kann eine relationale Transformation als eine Art "große Relation zwischen Modellen" verstanden werden.

Relationale Transformationen werden in der Regel auf der Ebene von Meta-Modellen deklariert. Dabei werden Relationen zwischen Klassen aus den Meta-Modellen möglicher Ausgangsmodelle und dem Meta-Modell möglicher Zielmodelle deklariert. Die Menge von Modellelementen aus einem Modell, die in der Transformation selektiert, verändert oder erzeugt werden, wird durch die Klassen eingeschränkt, da nur Modellelemente konformer Klassen betrachtet werden. Die Muster beschränken die Menge der Modellelemente des Ausgangsmodells weiter oder bestimmen Attribute und Assoziationen des Zielmodells.

Werden in den Mustern des Zielmodells Constraints verwendet, so handelt es sich um Constraint-relationale-Transformationen. Die Constraints werden für Modellelemente des Zielmodells in den Mustern aufgeschrieben, die für das Zielmodell gelten sollen. Dadurch werden sie auf alle erzeugten Modellelemente des Zielmodells angewendet und sind somit für den Prozess der Transformation lokal zu diesen. In der relationalen Sprache werden die Klassen mit den Constraints verknüpft,

[2] Alternativ kann eine Transformation nur zum Prüfen verwendet werden. Dann wird nur geprüft, ob die beiden Mengen der Relation entsprechen, und es werden keine Elemente erzeugt. Dies wird im Folgenden nicht weiter betrachtet.

ähnlich den Constraints von Constraint-Solving-Problemen auf Modellen.

Die aus Kapitel 1 bekannten Grundsätze lassen sich somit auf relationale Transformationssprachen einschränken. Es ergeben sich zwei formale Anforderungen und eine Definition:

1. Es muss eine relationale Transformationssprache verwendet werden.

2. In der relationalen Sprache sollen Constraints über Attributen und Assoziationen zulässig sein.

Diese Anforderungen an eine Sprache erzwingen auch die Einhaltung der Anforderungen A1 (Constraints in der Sprache), A6 (frei wählbare Constraints) und A11 (allgemeine, nicht-lineare Constraints) aus Kapitel 2.

Constraint-relationale-Transformationen können z.B. eingesetzt werden um die Mindestgrößen von Modellelementen zu setzen, die Komponenten von Benutzerschnittstellen repräsentieren. Dabei wird z.B. einem Meta-Modellelement, das für Buttons verwendet wird, ein Constraint hinzugefügt, so dass das Attribut für die Höhe von Buttons in Pixeln mindestens 40 Pixel betragen muss. Wird nun eine Transformation ausgeführt werden im Zielmodell ausschließlich Modellelemente dieses Typs erzeugt, die mindestens 40 Pixel hoch sind.

Begriffsdefinition 3.2. Constraint-relationale-Transformationen *sind relationale Transformationen mit Constraints, die die Attribute und Typen von Modellelementen des Zielmodells festlegen.*

Constraint-relationale-Transformationssprachen *sind Transformationssprachen, die die Notation von Constraint-relationalen-Transformationen unterstützen.*

Zur Veranschaulichung findet sich in Listing 4.6 auf Seite 143 ein konkretes Beispiel für eine relationale Transformation, die sogar eine Constraint-relationale-Transformation ist und in einer Constraint-relationalen-Transformationssprache notiert ist. Relationen sind am Schlüsselwort "relation" zu erkennen; diese bestehen aus Mustern; das Muster wiederum beinhaltet mehrere Zeilen, wobei in den Zeilen die Constraints zu erkennen sind. Die genaue Syntax der Sprache wird im folgenden Kapitel festgelegt, das Beispiel dient daher nur zur Veranschaulichung.

3.1.3.5 Algorithmus

Ein Algorithmus (s. Listing 3.1) zur Ausführung von Constraint-relationalen-Transformationen führt relationale Modell-zu-Modell-Transformation durch und konstruiert dabei ein Constraint-Solving-Problem (hier kurz CSP) auf Modellen (s. Abschnitt 3.1.2).

```
1  Lese vorhandene Modelle ein
2  Lese Relationen aus der Transformation in Liste L_R ein
3      foreach(Relation R1 in L_R)
4          foreach(Relation R2 in L_R)
5              if(R1 hängt von R2 ab)
6                  erzeuge Constraint R2 vor R1
7          foreach(Relation R3 in L_R)
8              if(R3 häangt von R1 ab)
9                  erzeuge Constraint R1 vor R3
10     Führe Solver für Constraints aus, Ergebnis = L_Sort(L)
11 foreach(Relation R in L_Sort(L))
12     Suche Teilmodelle (TM_(A)) in Ausgangsmodellen
13     foreach(TM_(A))
14         Erzeuge (TM_(Z))
15             // d.h. temporäre Modellelemente und Assoziationen
16             // des Zielmodells, Attribute noch offen
17         Suche Teilmodell TM_(Z) als TEMP_(Z) im Zielmodell
18             // muss Mustern entsprechen (Attribute fest)
19             if(exists(TEMP_(Z)) im Zielmodell)
20                 Lösche TEMP_(Z) im Zielmodell
21                 Ersetze durch TM_(Z)
22                 // da Attribute durch CSP gesetzt werden sollen
23         foreach(Modellelement m aus TM_(Z))
24             foreach(Attribut a von Attributen A(m))
25                 Füge eine Variable für a zu CSP hinzu
26             Speichere C_local(m) in Speicher S_temp
27             Speichere Assoziationen von m in Speicher S_temp
28             Füge m dem Zielmodell hinzu
29 foreach(Assozation l aus S_temp)
30     foreach(Modellelement m_temp das in l verwendet wird)
31         Suche Variablen v für m_temp des Zielmodells in CSP
32     Füge a dem Zielmodell mit allen v hinzu
33 foreach(Constraint c aus S_temp)
34     foreach(Modellelement m_temp das in c verwendet wird)
35         Suche Variablen v für m_temp des Zielmodells in CSP
36     Ersetze Variablen in c mit allen v
37     Füge c dem CSP hinzu
38 Führe Solver für CSP aus
39 Ersetze Werte der Attribute mit Werten der Lösung des CSP
40 Gebe Zielmodell aus
```

Listing 3.1: Algorithmus Constraint-relationale-Transformationen

Zeilen 2 bis 10: Die Relationen werden zuerst durch einen Constraint-Solver sortiert. Dafür werden die Relationen zuerst auf ihre Abhängigkeiten untersucht und für jede Abhängigkeit ein temporales Constraint eingefügt. Die Relationen dürfen für diesen Algorithmus daher keine zyklischen Abhängigkeiten haben (zyklische Abhängigkeiten werden auch durch den Constraint-Solving Schritt am Ende des gesamten Algorithmus unnötig, da diese durch das Constraint-Solving aufgelöst werden).

Zeile 11 bis 28: Dann werden für jede Relation (die manchmal auch als Regeln bezeichnet werden) im Ausgangsmodell Teilmodelle gesucht, die den Mustern und assoziierten Klassen entsprechen. Constraints, die in Mustern notiert worden sind, werden dabei bei der Suche berücksichtigt und nur passende Teilmodelle in den Ausgangsmodellen zur Transformation selektiert (Zeile 12).

Es wird zu diesem Teilmodell des Ausgangsmodells ein passendes temporäres Teilmodell des Zielmodell erzeugt, das den Vorgaben aus der betrachteten Relation entspricht (Zeile 14). Dabei werden Modellelemente des Zielmodells erzeugt, sodass diese den vorgegebenen Klassen und Assoziationen entsprechen; wenn es sich um eine Transformation in ein bereits existierendes Zielmodell handelt werden "alte Modellelemente" durch die neu erzeugten ersetzt um die neuen Werte zu übernehmen (Zeilen 14 bis 22).

Zeilen 23 bis 32: Constraints, die in Mustern des Zielmodells definiert werden (s.o.), werden im Prozess der Transformation als Constraints betrachtet, die lokal zu den erzeugten Modellelementen sind (Zeilen 23 bis 28). D.h. sie können als Constraints verstanden werden, die in einem entsprechend konstruierten Meta-Modell definiert wurden. Hierbei wird beachtet, dass Constraints auch Assoziationen über mehrere Modellelemente hinweg verwenden können. (Zeilen 29 bis 32) Bei Erzeugung neuer Modellelemente in der Transformation werden für $A(m)$ entsprechend dem Algorithmus für Constraint-Solving-Probleme auf Modellen neue Variablen als Elemente von V generiert (Typen und Assoziationen werden als Attribute betrachtet). Die Modellelemente werden bei der Transformation aber nacheinander erzeugt (jeweils während der Bearbeitung einer Relation). Bei einer naiven Implementierung des Algorithmus kann es vorkommen, dass bei Instantiierung eines Modellelements assoziierte Modellelemente noch nicht erzeugt worden sind. Bei Constraints, die mehrere Modellelemente betreffen (also Assoziationen verwenden), muss deshalb gewährleistet sein, dass diese Modellelemente bei Erzeugung des Constraints für das Zielmodell bereits vorhanden sind. Entsprechend werden bei der Erzeugung eines neuen Modellelements die Constraints $C_{local}(m)$ gespeichert und erst nach der Erzeugung aller Modellelemente mit Modellelementen verbunden (Zeilen 33 bis 37). Die Attribute (also auch die Typinformation und assoziierten Modellelemente), die in den Constraints verwendet werden, werden durch die erzeugten Variablen für die Attribute des Modellelements substituiert.

Zeilen 38 bis 40: Zum Setzen der Werte der Attribute kann ein klassischer Solver eingesetzt werden.

Der Algorithmus setzt Anforderung A7 (Constraint-Solving sollte Bestandteil der Transformation sein) aus Kapitel 2 um.

3.1.3.6 Eigenschaften von Constraint-relationalen-Transformationen

Ein klassisches Constraint-Solving-Problem ist ein statisches System, bei dem die Menge der Constraints von Anfang an festgelegt ist. Die Darstellung des Problems ist deklarativ, da explizit keine Zustände deklariert werden. Es wird also kein imperativer Programmcode geschrieben. Einige Problemstellungen haben aber keine feste Anzahl von Constraints und die Constraints müssen in einem Programm dynamisch erzeugt werden. Dabei kommen beliebige Programmiersprachen zum Einsatz (z.B. verschiedene Varianten der Sprache PROLOG [11]). Dadurch reduziert sich aber die Qualität des Ansatzes bzgl. der erreichten Deklarativität, denn selbst Programmiersprachen wie PROLOG sind als nicht rein deklarativ bekannt ([52]).

Im Fall von Constraint-relationalen-Transformationen kann auch die Erzeugung der Constraints durch rein deklarative Transformationen durchgeführt werden. Tatsächlich muss der Modellierer – also nicht der Entwickler der Transformationen – die in der Transformation verwendeten und erzeugten Constraints nicht einmal kennen. Durch Hinzufügen neuer Modellelemente zum Ausgangsmodell verändert er das entstehende Constraint System trotzdem automatisch, da die Transformation bereits Constraints für das Zielmodell deklariert. Wird die Transformation ausgeführt, gelten die Constraints automatisch für alle Modellelemente im Zielmodell (die aus den Modellelementen der vom Modellierer modellierten Ausgangsmodelle erzeugt werden), ohne dass der Modellierer die Constraints kennt. Dadurch muss er keinen Quellcode schreiben und profitiert trotzdem von den Constraints. Auch der Entwickler der Transformationen verwendet die rein deklarativen, relationalen Konstrukte aus der relationalen Transformationssprache. Damit wird das gesamte System rein deklarativ erzeugt, und es muss kein imperativer Programmcode zum Erzeugen des Constraint-Solving-Problems geschrieben werden.

Constraint-relationale-Transformationen mit mehreren Zielmodellen (Algorithmus aus Abschnitt 3.1.3.5 erzeugt nur ein Zielmodell) können trotzdem durch Algorithmen mit nur einem Zielmodell und Solvern berechnet werden:

Satz 3.3. *Die Zielmodelle von Constraint-relationalen-Transformationen können als Constraint-Solving-Problem dargestellt werden.*

Beweis. (durch Konstruktion) Die Zielmodelle werden nach dem Algorithmus aus Abschnitt 3.1.3.5 erzeugt. O.B.d.A. kann die Vereinigung aller Zielmodelle einer Transformation als ein Modell betrachtet werden, wenn auch deren Meta-Modelle

als ein gemeinsames Meta-Modell betrachtet werden (indem die Vereinigung aller Knoten und Kanten auf beiden Meta-Ebenen durchgeführt wird). Nach Abschnitt 3.1.3.5 ist V die Vereinigung aller Attribute der Modellelemente ($A(m)$) und entspricht damit V aus Definition 3.1. Die Menge der Constraints ist die Vereinigung aller Constraints der Transformation, die nun lokal zu den jeweiligen Modellelementen sind, entsprechend Definition 3.1. □

Da Constraint-relationale-Transformationen als Constraint-Solving-Probleme dargestellt werden können, sind daher analog zu Constraint-Solving-Problemen auf Modellen (s. Abschnitt 3.1.2) drei mögliche Fälle für Ergebnisse (d.h. Modelle) von Constraint-relationalen-Transformationen zu unterscheiden. Da die Constraints bei der Durchführung der Transformation erzeugt werden, hängen diese Fälle nicht nur von den Transformationen ab, sondern auch von den verwendeten Ausgangsmodellen. *Überspezifizierte Transformationen* können kein Zielmodell erzeugen, weil das Constraint-System keine Lösung hat. *Genauspezifizierte Transformationen* produzieren genau ein Zielmodell, während *unterspezifizierte Transformationen* mehrere verschiedene gültige Zielmodelle produzieren können.

Begriffsdefinition 3.4. Spezifiziertheit *bezeichnet im Folgenden die Ausprägung eines Modells; ein Modell kann unterspezifiziert, genauspezifiziert oder überspezifiziert ausgeprägt sein.*

Unterspezifizierte Transformationen sind besonders interessant, da der Solver entscheiden muss, welches Modell erzeugt werden soll. Mittels Optimierung kann durch Angabe einer Zielfunktion ein bestimmtes Modell bevorzugt werden (s. Abschnitt 3.1.4.1).

3.1.4 Optimierung

Mathematische Optimierung hat zum Ziel, für die Variablen einer Funktion eine Belegung zu finden so, dass die Funktion entweder einen maximalen oder minimalen Wert annimmt. Diese Funktion wird Zielfunktion genannt. Ein Optimierungsproblem besteht aus einer Menge von Variablen V, einer Menge von Constraints C und einer Zielfunktion f, für die globale oder lokale Extremwerte gesucht werden. Ein „globales Optimierungsproblem“ ist das Problem, eine Belegung der Variablen aus V zu finden so, dass f einen globalen Extremwert annimmt. Es existieren Solver, die in der Lage sind, globale Optimierungsprobleme zu lösen. Im folgenden werden mit “Solver” nur Solver bezeichnet, die diese Funktionalität besitzen.

3.1.4.1 Optimierung in Modell-zu-Modell-Transformationen

Optimierung lässt sich auf Modell-zu-Modell-Transformation anwenden. Dann kann aus einer Auswahl möglicher Zielmodelle das “optimale” ausgesucht werden.

Begriffsdefinition 3.5. Optimierende Constraint-relationale- Transformationen *sind Constraint-relationale-Transformationen mit einer Zielfunktion. Optimierende Constraint-relationale-Transformationssprachen sind Modell-zu-Modell-Transformationssprachen, die die Notation von optimierenden Constraint-relationalen-Transformationen erlauben.*

Das Konzept der Zielfunktion ist eine Ergänzung zu den bereits eingeführten Constraints, da alle Constraints erfüllt werden müssen, während die Zielfunktion nur "bestmöglich" erfüllt werden muss. Optimierung kann zwar theoretisch mit Constraint-Solving-Problemen abgebildet werden, da Constraint-Solving-Probleme im Allgemeinen Turing-vollständig sind, dies wäre für Entwickler aber nicht komfortabel.

Sprachen zur Deklaration von Optimierungsproblemen, wie z.B. OPL ([161]), sind meist deklarativ. Durch die Einführung der Zielfunktion wird die Deklarativität eines Ansatzes, der Constraint-Solving unterstützt, deshalb nicht negativ beeinflusst (Zustände werden nicht eingeführt).

Aufbauend auf den Anforderungen A9 (Zielfunktionen), A10 (explizite Notation von Zielfunktionen), A12 (leichte Veränderbarkeit) und den allgemeinen Grundsätzen aus Kapitel 1 lassen sich zwei Anforderungen für optimierende Constraint-relationale-Transformationen notieren:

1. Es soll eine Constraint-relationale-Transformationssprache verwendet werden.

2. Die Sprache muss die Möglichkeit der Deklaration von Zielfunktionen bieten.

Ein Beispiel für eine optimierende Constraint-relationale-Transformation wäre die bereits beschriebene Transformation mit Button-Modellelementen, die min. 40 Pixel hoch sein sollten. Wenn es sich "nur" um Constraint-relationale-Transformationen handelt und keine weiteren Constraints existieren, so ist 2000 ein gültiger Wert für das Attribut der Höhe des Buttons. Eine optimierende Constraint-relationale-Transformation könnte die Höhe unter der Randbedingung minimieren und würde so in diesem einfachen Fall 40 Pixel als Höhe wählen.

3.1.4.2 Algorithmus

Der Algorithmus zur Berechnung von Zielmodellen von Constraint-relationalen-Transformationen kann auch zur Berechnung von Zielmodellen von optimierenden Constraint-relationalen-Transformationen verwendet werden. Die Zielfunktion wird ähnlich einem speziellen Constraint behandelt; sie wird nach den Constraints in das Constraint-Solving-Problem eingefügt. Dadurch kann sie auf allen Modellelementen operieren.

Der Solver muss in der Lage sein, Optimierungsprobleme zu lösen. Hierbei lassen sich verschiedene Klassen von Problemen unterscheiden, für die jeweils spezielle Algorithmen erforderlich sind (z.B. Varianten des Branch-And-Bound-Algorithmus).

3.1.4.3 Eigenschaften von optimierenden Constraint-relationalen-Transformationen

Durch die Möglichkeit, Zielfunktionen deklarieren zu können, wird ein höherer Grad an Ausdrucksstärke (engl. "expressiveness") in den Modelltransformationen erreicht. Die Menge möglicher Zielmodelle kann durch die Angabe der Zielfunktion beschränkt werden. Dies kann sich auch in der Spezifiziertheit der Zielmodelle äußern.

Einige Erklärungen und Hilfsdefinitionen, die zur Definition von "reduziert" dienen:

Wie im Abschnitt 3.1.3.4 beschrieben, sind in Constraint-relationalen-Transformationen Muster mit Typen (Klassen) und Constraints assoziiert. Bei der Berechnung des Zielmodells werden die in den Mustern notierten Constraints lokal zu den Modellelementen, die mit dem Muster erzeugt werden. Die Constraints, die eine bestimmte Klasse betreffen, sind einerseits die Constraints, die in allen assoziierten Mustern des Zielmodells notiert sind; d.h. die Constraints, die lokal zu dieser Klasse sind. Zusätzlich beschränken die Constraints die Attribute des Modellelements, die, ausgehend von anderen Klassen, Attribute der Klasse durch Assoziationen verwenden. Sei das Ergebnis einer Transformation ein Modell mit einer Menge von Modellelementen. Jedes Modellelement besitzt eine Menge von Constraints $C_{localall}$. Diese besteht einerseits aus den erzeugten Constraints C_{local}, die zu dem Modellelement lokal sind (erzeugt von den Mustern der Klasse). Andererseits besteht sie aus den Constraints, die die Attribute des Modellelements verwenden und zu anderen Modellelementen lokal sind (erzeugt von Mustern von Klassen, die die Attribute der Klasse verwenden). $C_{localall}$ besteht dann - umgangssprachlich formuliert - aus Constraints, die Attribute des Modellelements und Attribute von anderen Modellelementen nutzen. Ist diese Betrachtung auf die Attribute des Modellelementes eingeschränkt, so gilt für die sog. Spezifiziertheit des Modellelements entsprechend der Definition für Constraint-relationale-Transformationen: Gibt es für die Belegung der Attribute genau eine (mindestens zwei / keine) Lösung so ist das Modellelement genau spezifiziert (unterspezifiziert / überspezifiziert).

Begriffsdefinition 3.6. *Ein Modellelement m heißt von einer Zielfunktion einer optimierenden Constraint-relationalen-Transformation* **reduziert**, *wenn durch die Anwendung einer Zielfunktion die Spezifiziertheit des Modells, das nur m enthält, genau spezifiziert ist.*

Einige Erklärungen und Hilfsdefinitionen, die zum einfacheren Verständnis des folgenden Beweises dienen sollen: Die Funktion $\# : \mathbb{X} \to \mathbb{N}$ bildet die Anzahl der Lösungen verschiedener Mengen $\mathbb{X}$:

- eine beliebig strukturierten Menge von Modellelementen (Modell),
- oder eine Menge von Lösungen eines Constraint-Solving-Problems,
- oder sie berechnet die Anzahl der Lösungen eines einzelnen Modellelements (s. Absatz davor)

Bildet # die Kardinalität einer Menge von Mengen, bildet sie # auf jeder Menge und multipliziert die Ergebnisse der einzelnen Mengen (Kardinalität des Kreuzproduktes).

Es folgen mehrere Lemma und Beweise die für Satz 3.10 benötigt werden. Es soll gezeigt werden, dass sich Transformationen auch nach Einführung von Zielfunktionen ähnlich verhalten wie klassische Constraint-Solving-Probleme. Zusätzlich kann durch die Lemma bestimmt werden, in welchen Fällen besondere Maßnahmen zur Fehlerbehandlung eingeführt werden müssen. Es stellt sich heraus, dass tendenziell weniger unterspezifizierte Modelle vorliegen.

Lemma 3.7. *Eine* optimierende Constraint-relationale-Transformation *ist für ein gegebenes Ausgangsmodell* **unterspezifiziert** *genau dann, wenn die Constraint-relationale-Transformation ohne Zielfunktion unterspezifiziert ist und die Zielfunktion nicht alle Modellelemente reduziert.*

Beweis. Sei S die Menge aller Lösungen der Constraint-relationalen-Transformation T für ein beliebiges Modell M und f die Zielfunktion, die in der optimierenden Constraint-relationalen-Transformation T_f verwendet wird, die sich ergibt, wenn T um f erweitert wird. S' sei die Menge aller Lösungen von T_f. Mit F wird der Solver bezeichnet, der Lösungen für Optimierungsprobleme berechnet. Dann gilt $T(M) = S$ und $T_f(M) = S'$. S und S' bestehen jeweils aus einer Menge M_* bzw. M'_* von Modellelementen und einer Menge von Mengen von möglichen Werten der Attribute der Modellelemente L bzw L', also $S = \{M_*, L\}$, sowie $S' = \{M'_*, L'\}$. Da F ein Solver ist, wählt F aus S bzw. M_* Kandidaten für S' bzw. M'_* aus. Obwohl die Anwendung der Zielfunktion auf die gesamte Lösungsmenge nicht effizient ist, kann F auf S und die Teilemengen L und M_* angewendet werden: $F(S) = S'$ bzw. $F(L) = L'$ und $F(M_*) = M'_*$. Da F eine Auswahl auf S trifft, gilt $S' \subseteq S$, d.h. $L' \subseteq L$, und damit $\#(L') \leq \#(L)$ sowie $\#(S') \leq \#(S)$. Da sich die Anzahl und Typen der Modellelemente des Modells für T_f durch F gegenüber T nicht ändert, da F nur "optimale" Werte aus L selektiert, gilt sogar: $M_* = M'_*$. Sei m ein Modellelement aus M. Bezeichne l die möglichen Werte für die Attribute von m für das Constraint-Solving-Problem

auf dem Modell, das nur m enthält, kurz $CSP(m)$. Wird F auch auf $CSP(m)$ angewendet, kann F die Anzahl möglicher Werte der Attribute $\#(l)$ einschränken. Da $M_* = M'_*$ gilt $m \in M'_*$ und $F(l) \in L'$. Reduziert bedeutet, dass $F(l)$ genau spezifiziert ist, also $\#(F(l)) = 1$.

$\Longrightarrow$: T_f unterspezifiziert, d.h. $\#(S') > 1 \Leftrightarrow \#(S') \geq 2$. Da $\#(S') \leq \#(S)$ gilt $\#(S) \geq 2$ und damit ist T unterspezifiziert (Teil 1).

(Teil 2) Reductio ad absurdum: Angenommen alle Modellelemente sind reduziert. Für ein Modell S' und ein beliebiges Modellelement $m \in M$ und entsprechendes l gilt dann $\bigcup_{F(l) \in L' \in S'} \{F(l)\} = L'$ und $\forall F(l) \in L' \Rightarrow \#(F(l)) = 1$.

Zeige dass $\{\forall m \in M'_* | F(l) \in L' | \#(F(l)) = 1\} \Rightarrow S'$ genau spezifiziert: Induktionsanker: $\#(F(l)) = 1 \Rightarrow \#(S') = 1$ wegen der Voraussetzung $\#(F(l)) = 1$ und $L' = \{F(l)\} \Rightarrow \#(L') = 1 \Rightarrow S' = \{M, L'\} \Rightarrow \#(S') = 1$ (weil $\#(M) = 1$). Induktionsschritt: Zur Menge komme jetzt ein Modellelement hinzu: $((M'_* = \{m_1, \ldots, m_n\} \wedge \#(\{F(l_1), \ldots, F(l_n)\}) = 1 \Rightarrow \#(S') = 1) \overset{Schritt}{\Rightarrow} ((M'_* = \{m_1, \ldots, m_n, m_{n+1}\} \wedge \#(\{F(l_1), \ldots, F(l_n), F(l_{n+1})\}) = 1 \Rightarrow \#((L' \in S') \cup \{F(l_{n+1})\}) = 1)$. Angenommen $(L' \in S') \cup \{F(l_{n+1})\}$ wäre überspezifiziert, dann muss gelten $\#(F(l_{n+1})) = 0$, da wegen vorherigem Schritt $\#(L') = 1$ gilt und nun keine Lösung mehr existiert. Da aber die Voraussetzung der Induktion $\#(F(l_{n+1})) = 1$ ist, gilt S' nicht überspezifiziert. Gleiches gilt für $(L' \in S') \cup \{F(l_{n+1})\}$ wäre unterspezifiziert, denn dann würde gelten $\#(F(l_{n+1})) \geq 2$. Widerspruch. Da also S' weder über- noch unterspezifiziert ist, muss S' genau spezifiziert sein. Also $\#(S') = 1$.

Dies ist aber im Widerspruch zur Annahme von Teil 2 der Aussage $\#(S') \geq 2$ und daher sind nicht alle Modellelemente durch die Zielfunktion reduziert (Teil 2).

$\Longleftarrow$: Voraussetzungen: T ist unterspezifiziert und $\{\exists F(l) \in L' \in S' | \#(F(l)) > 1\}$, weil die Zielfunktion nicht alle Modellelemente reduziert. Nach der Definition von $\#$ gilt $\#(L') = \prod_{l \in L'} F(l)$. Angenommen T_f genau spezifiziert, dann gilt $1 = \#(L') = \prod_{l \in L'} F(l)$. Da $\# \rightarrow \mathbb{N}$ muss gelten $\forall F(l) \in L' \Rightarrow F(l) = 1$. Dies ist aber nach Voraussetzung nicht der Fall. Widerspruch. Wäre T_f überspezifiziert, dann würde $\#(S') = 0$ gelten. Nach der Definition von Zielfunktion sucht F die bestmögliche Lösung aus einer Menge aus. Dies bedeutet, dass bei gegebenem $\#(S) > 0$ immer $\#(F(S)) > 0$ gelten muss, da F dann keine optimale Lösung liefern kann, wenn $\#(S) = 0$ gilt; ansonsten sucht F nicht die bestmögliche Lösung, sondern schränkt die Menge weiter ein. Da $\#(S) \geq 2$ muss gelten $\#(F(S)) > 0 \Rightarrow \#(S') > 0$. Widerspruch zur Annahme T_f sei überspezifiziert. Da S' also weder genau noch überspezifiziert sein kann, muss S' unterspezifiziert sein. □

Lemma 3.8. *Eine optimierende Constraint-relationale-Transformation ist für ein gegebenes Ausgangsmodell* ***überspezifiziert*** *genau dann, wenn die Transformation ohne Zielfunktion überspezifiziert ist.*

Beweis. Es werden die Erkenntnisse und Definitionen aus dem Beweis zu Lemma 3.7 verwendet.

$\Longrightarrow$: T_f überspezifiziert, d.h. $\#(S') = 0 \Leftrightarrow \#(L') = 0$. Es gilt $0 = \#(S') = \#(F(S))$ und $0 = \#(L') = \#(F(L))$. Da F aus der Menge S optimale Werte sucht und wegen $\#(F(S)) = 0$ kein solcher Wert existiert, muss $\#(S) = 0$ gelten. Denn angenommen es existiert zumindest ein Wert, also $\#(S) \geq 1$, dann existiert auch mindestens ein optimaler Wert, also $\#(F(S)) \geq 1$.

$\Longleftarrow$: T überspezifiziert, d.h. $\#(S) = 0 \Rightarrow \#(L) = 0$. Wird aus S ein optimaler Wert gesucht, kann dieser dann auch nicht existieren, da F ein optimales Element auf der leeren Menge sucht. Daher gilt $\#(F(S)) = 0 = \#(S')$. □

Lemma 3.9. *Eine optimierende Constraint-relationale-Transformation ist für ein gegebenes Ausgangsmodell genau dann* ***genau spezifiziert****, wenn die Transformation ohne Zielfunktion entweder genau spezifiziert ist, oder unterspezifiziert ist und die Zielfunktion alle Modellelemente reduziert.*

Beweis. Es werden die Erkenntnisse und Definitionen aus dem Beweis zu Lemma 3.7 und 3.8 verwendet.

$\Longrightarrow$: T_f ist genau spezifiziert, d.h. $\#(S') = 1 = \#(F(S))$. Da F auf S angewendet die "optimalen" Lösungen sucht, muss F eine Lösung finden, sofern eine in S existiert; d.h. $\#(S) \neq 0$. f kann entweder aus einer großen Menge eine Lösung aussuchen, d.h. $\#(S) > 1$ oder die Funktion f gibt aus einer ein-elementigen Lösungsmenge genau das einzige Element zurück, d.h. $\#(S) = 1$.

$\Longleftarrow$: Fall 1: T genau spezifiziert: Dann $\#(S) = 1 = \#(L)$. Da f optimale Lösungen auf L sucht und $f(S)$ nach Lemma 2 nicht überspezifiziert sein kann (da sonst $\#(S) = 0$ gelten müsste), kann f in einer ein-elementigen Lösungsmenge maximal eine Lösung finden und es muss gelten: $f(L) = 1 = L' \Rightarrow \#(S') = 1$. Fall 2: T ist unterspezifiziert und alle Modellelemente sind reduziert, also gilt $\{\forall f(l) \in L' | f(l) = 1\}$ und mit $\bigcup_{f(l) \in L' \in S'} \{f(l)\} = L'$ gilt dann, da für jede Teilmenge von L' nur eine Lösung existiert dass für die gesamte Menge L' auch nur eine Lösung existieren kann. □

Satz 3.10. *Die Spezifiziertheit von Constraint-Solving-Problemen kann auf die Spezifiziertheit von optimierenden Constraint-relationalen-Transformationen übertragen werden.*

Beweis. folgt unmittelbar aus einer Fallunterscheidung mit Lemma 3.7 bis 3.9. □

Es gibt verschiedene Arten von Graphtransformationen. Werden Modelle als Graphen betrachtet können Constraint-relationale-Transformationen als eine spezielle Art von algebraischen Graphtransformationen gesehen werden (die aber nicht auf klassischen Graph-Transformationsregeln fußen). Dies dient hier dem Zweck, die Definition der sog. "Konfluenz" auf Constraint-relationale-Transformationen zu

erweitern. "Nicht-Konfluent" bedeutet, dass eine Transformation potentiell mehrere Lösungen haben kann (für eine genaue Definition von Konfluenz im Zusammenhang mit Graphtransformationen s. [61]). Dies ist nicht wünschenswert, da bei einigen regelbasierten Graphtransformationsansätzen auf Grund von unterschiedlicher Reihenfolge der Anwendung von Regeln potentiell nicht-deterministische Ergebnisse (also unterschiedliche Graphen) entstehen können. Deshalb werden Graphtransformationen möglichst so definiert, dass sie konfluent sind. Es ist ein bekanntes Problem, Transformationen so zu definieren, dass sie beweisbar konfluent sind. Mit Optimierung bietet diese Arbeit dazu eine neue Lösung an, mit Transformationen Konfluenz einfacher zu erreichen:

Begriffsdefinition 3.11. *Eine genau-spezifizierte Constraint Relationale Transformation heiße für ein gegebenes Ausgangsmodell* **konfluente Transformation**, *eine unterspezifizierte Constraint Relationale Transformation heiße für ein gegebenes Ausgangsmodell* **nicht-konfluente Transformation**.

Satz 3.12. *Existiert eine geeignete Zielfunktion, können nicht-konfluente Constraint-relationale-Transformationen durch Hinzufügen dieser Zielfunktion, also der Umwandlung in optimierende Constraint-relationale-Transformationen, in konfluente Transformationen umgewandelt werden.*

Beweis. Aus Lemma 3.9 und Definition 3.11 folgt unmittelbar, dass dies möglich ist. □

Es ist zu vermuten, dass der Ansatz auf viele Graphtransformationssprachen erweitert werden kann [3]. Bei diesen lassen sich ebenfalls Konstrukte zur Definition von Zielfunktionen definieren. Graphen sind diskrete Strukturen. Daher kann bei Selektion von Regeln ein Branch-And-Bound-Verfahren angewendet werden, sodass der Zielgraph minimal wird. Es bleibt offen inwieweit sich bekannte Verfahren zur Erhöhung der Effizienz (wie z.B. Constraint Propagation) benutzen lassen.

Folgende Transformationen profitieren besonders von dem in diesem Abschnitt vorgestelltem Ansatz:

- Nicht-konfluente Transformationen: Durch den Einsatz einer Zielfunktion kann eine ursprünglich nicht konfluente Transformation in eine konfluente Transformationen überführt werden, indem die Zielfunktion so gewählt wird, dass sie alle Modellelemente reduziert (siehe Satz 3.12).

[3] Der Vollständigkeit halber sei erwähnt, dass mit den vorgestellten Konzepten nicht alle nicht-konfluenten Graphtransformationen (also nicht nur Constraint-relationale-Transformationen) in konfluente umgewandelt werden können.

- Constraint-relationale-Transformationen bei denen Werte noch nicht feststehen: Attribute, die in Constraint-relationalen-Transformationen unterspezifiziert wurden, können nun vom Entwickler auf einen Wert festgelegt werden, indem eine Zielfunktion für die Suche der "optimalen" Werte genutzt wird.
- Neue Anwendungsfälle: Der Einsatz von Zielfunktionen bietet neue Anwendungsfälle für die Nutzung von Modell-zu-Modell-Transformationssprachen und Graphtransformationen insgesamt, da Zielfunktionen in diesem Forschungsbereich noch nicht erforscht wurden. Es können nun bzgl. einer Zielfunktion "optimale" Modelle erzeugt werden.

3.1.5 Derivatprobleme

In einer klassischen relationalen Modelltransformation stehen die verwendeten Klassen oder Typen des Zielmodells durch Definition in der Modelltransformation bereits fest. Es gibt für verschiedene Modellelemente des Zielmodells daher keine Möglichkeiten, aus einer Auswahl an verschiedenen Typen einen passenden Typ auszuwählen. Manche Anwendungsfälle, wie z.B. die Transformation von Modellen von Benutzerschnittstellen, erfordern diese Funktionalität. In der Transformation soll z.B. festgelegt werden, ob eine Liste in der Benutzerschnittstelle als eine klassische Listbox oder als Liste von Radiobuttons dargestellt wird. In diesem Beispiel wäre "Liste" ein Modellelement des Meta-Modells für abstrakte Modelle; "Listbox" und "Liste von Radiobuttons" Modellelemente des konkreten Meta-Modells. Die Transformation kann nun auswählen ob für eine "Liste" eine "Listbox" oder eine Liste von "Radiobuttons" erzeugt wird. Diese haben unterschiedliche Constraints, die von anderen Modellelementen abhängen, was mit QVT Relations nicht realisierbar wäre. Dieses Beispiel wird auch in Kapitel 4 in Form einer Transformation implementiert und dient daher auch als Implementierungsbeispiel.

Begriffsdefinition 3.13. *Eine relationale Modell-zu-Modell-Transformation heißt* **Derivatproblem***, wenn die Transformation im Zielmodell für min. ein zu erzeugendes Modellelement des Zielmodells min. zwei mögliche Klassen (oder Typen) aus dem Meta-Modell des Zielmodells zulässt. Ein* **Derivat** *ist eine Modell-zu-Modell-Transformation, die jedem Modellelement genau eine im Derivatproblem vorgesehene Klasse (oder Typ) zuordnet.*

Derivatprobleme werden in der Regel mit Constraints und Zielfunktionen kombiniert. Als Anforderungen an eine Transformationssprache, mit der sich Derivatprobleme notieren lassen, ergeben sich:

1. Es soll eine Constraint-relationale-Transformationssprache oder eine optimierende Constraint-relationale-Transformationssprache als Grundlage für die Deklaration von Derivatproblemen Verwendung finden.

2. Durch die Sprache muss die Möglichkeit bereitgestellt werden, für ein Modellelement eines Ausgangsmodells mehrere Typen im Zielmodell zu definieren.

Dabei sind zwei Details bei der Implementierung interessant, weil sie erst durch Derivatprobleme möglich werden:

1. Da an jede Klasse unterschiedliche Constraints gebunden werden, können unterschiedliche Derivate unterschiedliche Constraints nutzen. Dabei kann jedes Derivat entweder unterspezifiziert, überspezifiziert oder genau spezifiziert sein.

2. Eine Zielfunktion für ein Derivatproblem kann für verschiedene Derivate unterschiedliche Werte annehmen. Die Zielfunktion muss das optimale Modell aller Derivate liefern.

Ein Algorithmus zur Lösung von Derivatproblemen muss diese beiden Anforderungen berücksichtigen und wird im nächsten Abschnitt vorgestellt.

3.1.5.1 Algorithmus

Jedes Derivat kann durch eine optimierende Constraint-relationale-Transformation implementiert werden. Dies liegt daran, dass ein Derivat eine Modelltransformation mit einer gültigen Zuweisung aller möglichen Klassen oder Typen an die Modellelemente ist. Da bei einem Derivat bereits alle Klassen und Typen in der Modelltransformation feststehen, ist dieses nicht anders zu behandeln als eine klassische Modelltransformation. Weil Zielfunktionen und Constraints Erweiterungen relationaler Modell-zu-Modell-Transformationen sind, können Derivatprobleme auch als optimierende Constraint-relationale-Transformationen dargestellt werden.

Unter der in der Definition gegebenen Voraussetzung, dass die Anzahl der zulässigen Klassen oder Typen für alle Modellelemente endlich ist, kann das "optimale" Derivat durch Iteration über alle Derivate gefunden werden. Ist keine Zielfunktion in der Transformation vorhanden, wird das zuerst gefundene Zielmodell des ersten gefundenen nicht überspezifizierten Derivates zurückgegeben. Ist hingegen eine Zielfunktion vorhanden, müssen die Ergebnisse der Zielfunktion für verschiedene Derivate verglichen werden. Dazu wird zuerst der Raum aller möglichen Derivate durch Generierung des Kreuzproduktes aller möglichen Klassen der Derivate erzeugt (Schritt 1 aus Listing 3.2). Alle Derivatprobleme werden durch einen Solver gelöst und das Ergebnis einer eventuell vorhandenen Zielfunktion gespeichert (Schritte 2 bis 5). Wird die Zielfunktion der Transformation maximiert (bzw. minimiert) wird das Zielmodell des Derivates, dessen Zielfunktion den maximalen (bzw. minimalen) Wert hat, zurückgegeben (Schritte 7 bis 11).

```
1  Erzeuge alle Derivate durch Permutation und speichere in D_temp
2  foreach(Derivat d aus D_temp)
3      Führe Algorithmus aus Abschnitt 3.1.4.2
4      Speichere Wert der Zielfunktion in Z_temp
5      Speichere Zielmodell in Z_temp
6  Sortiere Z_temp aufsteigend nach den Werten der Zielfunktionen
7  if(Zielfunktion soll maximiert werden)
8      E = letzter Eintrag der Modelle in Z_temp
9  else
10     E = erster Eintrag der Modelle in Z_temp
11 Gebe E aus
```

Listing 3.2: Algorithmus Derivatprobleme

3.1.5.2 Eigenschaften von Derivatproblemen

Aus der Definition von unter-, über- und genau spezifizierten Constraint-relationalen-Transformationen lässt sich eine Definition für die Spezifiziertheit für Derivatprobleme ableiten.

Lemma 3.14. *Ein Derivatproblem ist genau dann* ***unterspezifiziert****, wenn mindestens eines seiner Derivate unterspezifiziert ist oder mindestens zwei seiner Derivate genau spezifiziert sind.*

Beweis. Sei # eine Funktion, die die Anzahl der Lösungen eines Derivates oder einer Menge von Derivaten liefert und speziell $\#(D_p)$ die Anzahl der Lösungen eines Derivatproblems. Sei U die Menge der unterspezifizierten Derivate, G die Menge der genau spezifizierten Derivate und B die Menge der überspezifizierten Derivate. $|M|$ sei die Kardinalität einer Menge M.

$\Longrightarrow$: Ist das Derivatproblem unterspezifiziert, so gilt $\#(D_p) > 1$.
Es gilt $\{\forall b \in B | \#(b) = 0\} \Rightarrow \#(B) = \sum_{b \in B} \#(b) = 0$;
$\{\forall g \in G | \#(g) = 1\} \Rightarrow \#(G) = \sum_{g \in G} 1 = |G|$ sowie
$\{\forall u \in U | \#(u) > 1\} = \{u \in U | \#(u) \geq 2\}$
$\Rightarrow \#(U) = \sum_{\forall u \in U} \#(u) \geq \sum_{u \in U} 2 = 2 * \sum_{u \in U} 1 = 2 * |U|$.
$\#(D_p) > 1 \Rightarrow \#(D_p) \geq 2$
und $\#(D_p) = \#(U) + \#(G) + \#(B) \geq 2 * |U| + |G| + 0$.
Dieses Constraint-System hat nach Lemma 3.14 genau drei Lösungsmengen:
Fall 1: $|U| = 0$ und $|G| \geq 2 \Rightarrow 2 * 0 + |G| + 0 \geq 2$;
Fall 2: $|U| \geq 1$ und $|G| = 0 \Rightarrow 2 * |U| + 0 + 0 \geq 1$;
Fall 3: $|U| \geq 1$ und $|G| \geq 1$; s. Fall 2.
$\Longleftarrow$: Es werden die aus $\Longrightarrow$ bekannten Bezeichner verwendet.
Dann Fall 1, $|U| \geq 1$, $|G| = 0$: $\#(D_p) = \#(U) + \#(G) + \#(B) \geq 2 * |U| + |G| + |B|$

$= 2 * 1 + 0 + 0 = 2$.
Fall 2, $|U| = 0, |G| \geq 2$: $\#(D_p) \geq 2 * 0 + 2 + 0 = 2$.
Fall 3, $|U| > 1, |G| > 0$, s. Fall 1. □

Selbst wenn ein Derivat genau spezifiziert ist, bedeutet dies also nicht automatisch, dass das Derivatproblem genau spezifiziert ist. Aus diesem Grund drängt sich in einem solchen Fall die Verwendung von Zielfunktionen besonders auf.

Lemma 3.15. *Ein Derivatproblem ist* ***überspezifiziert****, genau dann, wenn alle seine Derivate überspezifiziert sind.*

Beweis. Es werden die aus Lemma 3.14 bekannten Bezeichner verwendet.
$\Longrightarrow$: Überspezifiziert $\Rightarrow \#(D_p) = 0$.
Also $0 = \#(U) + \#(G) + \#(B) \geq 2 * |U| + |G| + 0$.
Da $|U| \geq 0$ für alle U bzw. $|G| \geq 0$ für alle G gilt $|U| = 0$ und $|G| = 0$.
$\Longleftarrow$: Für $|U| = 0$ und $|G| = 0$ und $|B| \geq 0$ gilt $\#(D_p) = 2 * 0 + 0 + 0 = 0$. □

Lemma 3.16. *Ein Derivatproblem ist genau dann* ***genau spezifiziert****, wenn genau eines seiner Derivate genau spezifiziert ist und alle seine anderen Derivate überspezifiziert sind.*

Beweis. Es werden die aus Lemma 3.14 bekannten Bezeichner verwendet.
$\Longrightarrow$: Genau spezifiziert $\Rightarrow \#(D_p) = 1$.
Also $1 = \#(U) + \#(G) + \#(B) \geq 2 * |U| + |G| + 0$.
Wenn $|U| > 0$ folgt $\#(D_p) \geq 2$, Widerspruch.
Also $|U| = 0$ und $0 + |G| + 0 * |B| = 1 \Leftrightarrow |G| = 1$ und $|B| \geq 0$.
$\Longleftarrow$: $\Rightarrow$ in umgekehrter Reihenfolge. □

Satz 3.17. *Die Spezifiziertheit von Constraint-Solving-Problemen kann auf die Spezifiziertheit von Derivatproblemen übertragen werden.*

Beweis. Folgt unmittelbar aus einer Fallunterscheidung mit Lemma 3.14 bis 3.16. □

Schedulingprobleme können über Derivatprobleme implementiert werden (eine Definition von Schedulingproblemen findet sich in [160]). Ein Schedulingproblem ist ein Problem, bei dem Ressourcen zu Prozessen zugeteilt werden sollen so, dass die Ressourcen möglichst effizient (bzgl. einer bestimmten Metrik) genutzt werden. Z.B. können bei der Transformation von Modellen von Benutzerschnittstellen "Panels" als ein komplexer Prozess und der benötigte Platz auf der Bildschirmfläche – der von konkreten Komponenten verbraucht wird – als Ressource modelliert werden; s. Abschnitt 3.2.5. Solche Transformationen können mit Hilfe der vorgestellten Kombination aus Zielfunktion und Derivatproblem häufig sehr elegant dargestellt

werden. Der Prozess wird im Ausgangsmodell durch ein Modellelement repräsentiert, z.B. "Panel". In der Transformation werden mögliche Ressourcen (d.h. Panels mit Komponenten zum Editieren oder Anklicken) dem Prozess zugeteilt und als mögliche alternative Klassen oder Typen des Modellelementes im Zielmodell modelliert. Die Transformation wählt dann die beste Kombination von Ressourcen anhand der Zielfunktion aus. Sind zusätzlich Constraints vorhanden, kann das Problem als ein Derivatproblem mit optimierenden Constraint-relationalen-Transformationen dargestellt werden.

Ein weiteres Einsatzgebiet für Derivatprobleme ist das Finden gültiger Constraint-relationaler-Transformationen. Der Entwickler einer Transformation kann verschiedene Constraints zu alternativen Typen und Klassen zuordnen. Werden mehrere solche Modellelemente mit Constraints verwendet, ist es für Entwickler schwer zu erkennen, ob eine Transformation überspezifiziert ist. Mit Hilfe eines Derivatproblems und einer entsprechenden Transformationsmaschine kann der Entwickler dem Computer die Aufgabe überlassen, Kombinationen mit mindestens einer gültigen Lösung zu finden. Dadurch braucht er sich nicht selbst darum bemühen, Transformationen zu deklarieren, die nur gültige Lösungen haben.

Entwickler, die folgende Transformationen entwickeln, profitieren besonders von Derivatproblemen:

- Transformationen, die Schedulingprobleme lösen sollen. Also insbesondere Entwickler von Transformationen von Modellen von Benutzerschnittstellen.

- Transformationen, die verschiedene Constraints für Typen von Modellelementen zulassen. Dabei haben möglicherweise nur bestimmte Kombinationen solcher Constraints mindestens eine Lösung. Dies sind z.B. Transformationen von Modellen von Benutzerschnittstellen, bei denen z.B. "Listboxen" andere Mindestgrößen (Constraint, das die minimale Größe beschreibt) haben als "Radiobuttonlisten".

3.2 Transformation von Modellen von Benutzerschnittstellen

Transformationen von Modellen von Benutzerschnittstellen überführen abstrakte Modelle von Benutzerschnittstellen in konkrete (dies ist eine Spezialisierung des MDA-Musters, s. [118]). Dabei sind die **abstrakten Modelle der Benutzerschnittstellen** (im MDA-Muster auch als "Platform Independent Model" oder "PIM" bezeichnet) bezüglich einer Menge von Eigenschaften unabhängiger von der Implementierung als es die konkreten sind (im MDA-Muster auch als "Platform Specific Model" oder PSM bezeichnet). Im Falle von Benutzerschnitt-

stellen sind dies z.B. Modalitäten oder spezielle Eigenschaften der Komponenten verschiedener Betriebssysteme.

Zur Verbesserung der Effizienz des Prozesses der Modellierung ist es erforderlich, das Zielmodell mit Transformationen so zu erzeugen, dass Modellierer weniger Änderungen am Zielmodell durchführen müssen als notwendig wären, um ein komplett neues von Hand zu erzeugen. Dazu sollten die Transformationen Modelle von Benutzerschnittstellen erzeugen können, die bereits nach der Transformation den Anforderungen an die von den Modellierern angestrebte Benutzbarkeit genügen. Andernfalls müssen die Modelle von Modellierern per Hand so verändert werden, dass sie die Anforderungen erfüllen. Eine Voraussetzung dafür ist, dass Benutzerschnittstellen durch einen Algorithmus den Anforderungen entsprechend generiert werden können. Derzeit ist der Stand der Forschung im Bereich HCI noch nicht so weit fortgeschritten, dass die Generierung von Benutzerschnittstellen selbst mit einer Turing-vollständigen Programmiersprache immer zufriedenstellend durchgeführt werden kann. Dies hat vielschichtige Ursachen, sodass sich praktisch nutzbare Lösungen auf einen Teil aller Benutzerschnittstellen beschränken. Trotzdem gibt es interessante Ansätze, die mit Hilfe von deklarativen Beschreibungen bereits in vielen Fällen Benutzerschnittstellen liefern, die zumindest niedrigen Ansprüchen an die Benutzbarkeit genügen (z.B. [48]).

Diese Schnittstellen können bei aktuellen Ansätzen nachher nicht mehr verändert werden (d.h. sie wurden direkt angezeigt), sodass eine alleinige Abhängigkeit vom Algorithmus zur Generierung der Benutzerschnittstellen besteht. Im Gegensatz dazu können Modelle von Benutzerschnittstellen von Modellierern verbessert werden, auch wenn die Modelle in einer Transformation generiert wurden. Die Transformation dient also der (Teil-)Automatisierung des Modellierungsvorgangs, da die Modelle von den Modellierern ohne Transformationen selbst erstellt werden müssen. Bisher gab es allerdings keine deklarative Transformationssprache, die es erlaubte, die aus der Generierung von Benutzerschnittstellen bekannten, deklarativen Beschreibungen (Constraints und Optimierung) in deklarativen Transformationen zu nutzen.

3.2.1 Benutzbarkeit

Damit die generierten Modelle von Benutzerschnittstellen Ansprüchen an die Benutzbarkeit genügen können, müssen diese Ansprüche zuerst formalisiert werden. Weil Benutzbarkeit naturgemäß sehr schwer zu formulieren ist, geschieht dies in der HCI auf sehr unterschiedliche Weise. Ansätze zur Formulierung von Benutzbarkeit finden sich z.B. in [2] und [169], sind aber für Entwickler und Modellierer zu ungenau, da sie Benutzbarkeit auf sehr abstrakte Weise definieren. Es können so keine formalen Anforderungen an Benutzerschnittstellen extrahiert werden. In Ermangelung einer genügend großen Anzahl von theoretisch fundierten Ansätzen

entstanden Regelsätze, die zur Formalisierung von Benutzbarkeit auf Regeln setzen, die von Designern von Benutzerschnittstellen bei der Gestaltung der Nutzerschnittstelle in praktischen Anwendungen beachtet werden (z.B. [10]). Für diese Arbeit von besonderem Interesse sind Regeln, mit denen sich Benutzerschnittstellen deklarativ erzeugen lassen.

Es lassen sich zwei Klassen von Regeln bilden:

Qualitative Regeln basieren meist auf Erfahrungswerten von Designern und liegen meist nicht in einer Form vor, die ein maschinelles Interpretieren erlauben würde. Dadurch können viele dieser Regeln nicht ohne die Erfahrung von Designern in Benutzerschnittstellen umgesetzt werden. Da bis heute nicht bekannt ist, wie eine ausreichend große Menge an Erfahrungswerten von Designern in Computern im Allgemeinen abgebildet werden kann, sind viele dieser Regeln nicht im Computer abzubilden und somit auch nicht mit in dieser Arbeit vorgestellten Ansatz. Qualitative Regeln haben keine Abbildung auf einen Wert, mit dem sich die Güte der Benutzerschnittstelle bzgl. dieses Wertes messen lässt. In den Fällen, in denen sie dargestellt werden können, werden sie daher nur durch Regeln dargestellt, die eine qualitative Entscheidung zulassen (Regel erfüllt: „Ja“ oder „Nein“). Eine Untermenge dieser Regeln kann in Form von Constraints formuliert werden. In Abschnitt 5.7 wird eine Untersuchung vorgestellt, mit der geprüft wurde, wie viele solcher Regeln eines bekannten Style Guides sich mit der in dieser Arbeit entwickelten Transformationssprache abbilden lassen.

Quantitative Regeln bilden Eigenschaften von Benutzbarkeit auf quantitative Werte ab. Die bekannteste und am meisten erforschte Regel dieser Kategorie ist Fitt's Law. Fitt's Law approximiert die Zeit, die ein Mensch zum Navigieren in einem ein-dimensionalem Raum in Abhängigkeit von der Größe und der Entfernung des Ziels benötigt. Es wurde auf zwei-dimensionale Räume erweitert und dieser ist für klassische grafische Benutzerschnittstellen relevant. Erstaunlicherweise konnten viele andere quantitative Regeln auf Fitt's Law zurückgeführt oder davon abgeleitet werden (z.B. Accot-Zhai Steering-Law [3] oder SMALLER-OF Varianten [99]). Card und Newell sind der Meinung, dass es eine der besonderen Herausforderungen der HCI ist, quantitative Regeln aufzustellen. Leider gibt es immer noch wenige solcher Regeln [166]. Mit dem in dieser Arbeit vorgestellten Ansatz können nur solche quantitativen Regeln abgebildet werden, die mit Optimierungsalgorithmen effizient berechnet werden können.

Es sind nicht alle Regeln für alle Anwendungsfälle gleichermaßen relevant, sodass nicht alle Regeln in jeder Transformation benötigt werden. Trotzdem sollte

eine Transformationssprache in der Lage sein, möglichst viele Regeln abzubilden, da dann viele verschiedene Aspekte von Benutzbarkeit in Transformationen für die verschiedenen Anwendungsfälle berücksichtigt werden können. Auf Grund ihrer Natur lassen sich quantitative Regeln meist mit Zielfunktionen formalisieren (s. Abschnitt 3.2.4), während eine Menge von qualitativen Regeln als Constraints notiert werden kann (s. Abschnitt 3.2.3).

Entsprechend den neuen Anforderungen an eine Transformationssprache können Modelle mit Constraint-Solving, Optimierung und Derivatproblemen transformiert werden. Auf Grund der groben Anforderungen aus Kapitel 1 können so die qualitativen und quantitativen Regeln in Constraint-Solving- und Optimierungsprobleme übersetzt werden. Ebenfalls können Derivatprobleme zum Lösen von Schedulingproblemen in Benutzerschnittstellen genutzt werden.

Transformationen von Modellen von Benutzerschnittstellen hängen von Meta-Modellen ab. Um beispielhaft Transformationen zu notieren, müssen deshalb Meta-Modelle definiert werden, an deren Beispiel Regeln über die Benutzbarkeit (z.B. aus Style Guides) in Transformationen notiert werden können.

3.2.2 Meta-Modelle

In Modell-zu-Modell-Transformationen werden Modelle von Benutzerschnittstellen transformiert, die durch Meta-Modelle definiert werden. In der Transformation werden dann Instanzen dieser Meta-Modelle transformiert. Dabei werden Modellelemente des Ausgangsmodells in Modellelemente des Zielmodells transformiert. Relationale Transformationen sind vom Meta-Modellen abhängig, da sie Klassen aus Meta-Modellen zueinander in Relation setzen. Die hier vorgestellten Meta-Modelle sind daher nur ein mögliches Beispiel, um konkrete Transformationen notieren zu können. Trotzdem sind sich die meisten Meta-Modelle von klassischen Ansätzen zur Transformation von Benutzerschnittstellen sehr ähnlich.

Es wird ein spezielles Metamodellelement definiert, von dem alle anderen Metamodellelemente erben, die die Komponenten von Benutzerschnittstellen modellieren. Abbildung 3.2 zeigt den Kern eines Meta-Modell, das im Rahmen der vorliegenden Arbeit entwickelt wurde. Es dient als Ausgangsmodell für die Modelltransformation. Die verwendete Notation für die Meta-Modelle entspricht der MOF [113]. Klassen werden als Kasten modelliert. Diese Kästen bestehen aus vier Teilen, wobei hier nur die obersten beiden Teile relevant sind: Der Name der Klasse steht im obersten Teil eines Kastens (z.B. “Interactor”). Darauf folgen die Attribute. Die Vererbung wird durch Pfeile dargestellt, deren Spitzen weiß gefüllt sind (zeigen auf Klassen von denen geerbt wird). Assoziationen zwischen Klassen haben einen Namen und Kardinalitäten. Sie können nur in eine Richtung existieren (Pfeil mit Spitze aus Strichen), oder bidirektional sein (kein Pfeilende). Sollte es sich dabei um eine Komposition von Klassen handeln wird dies durch einen ausgefüllten

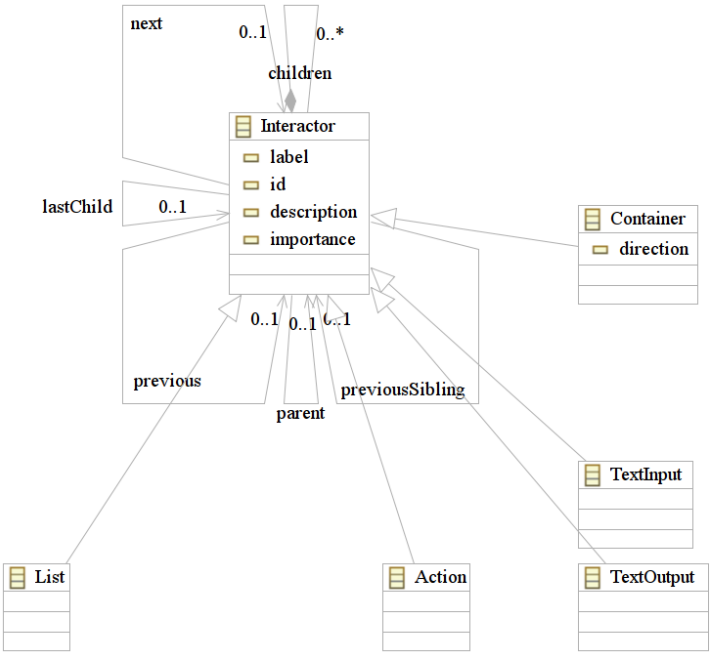


Abbildung 3.2: Beispielhaftes, abstraktes Meta-Modell

Diamant dargestellt ("besteht aus").

Man erkennt das Metamodellelement "Interactor", von dem alle anderen Metamodellelemente erben. Dieser klassische Ansatz wird auch im Zielmodell verfolgt. Hier erben alle Metamodellelemente, die Komponenten der Benutzerschnittstelle modellieren, von einem Element "Component" (Abbildung 3.3). Man erkennt dass wesentlich mehr Modellelemente im Zielmodell existieren.

Im Zielmodell sollen durch die Modelltransformation Eigenschaften so verändert oder hinzugefügt werden, dass dem Modellierer Arbeiten abgenommen werden, die er normalerweise per Hand hätte durchführen müssen. Die Modelltransformation soll daher genau die Eigenschaften erzeugen, die der Modellierer ausgehend vom Ausgangsmodell im Zielmodell konkretisieren würde. Im vorliegenden (beispielhaften) Meta-Modell werden dabei zu "Component" neue Attribute hinzugefügt, die diese zusätzlich erzeugten Eigenschaften repräsentieren. Es sind insbesondere die Position (x, y) sowie die Größe (w, h) der Komponente zu erkennen, wie sie in modernen zwei-dimensionalen grafischen Benutzerschnittstellen verwendet werden. Diese sollten am besten in der Transformation unter Berücksichtigung passender qualitativer oder quantitativer Regeln berechnet werden.

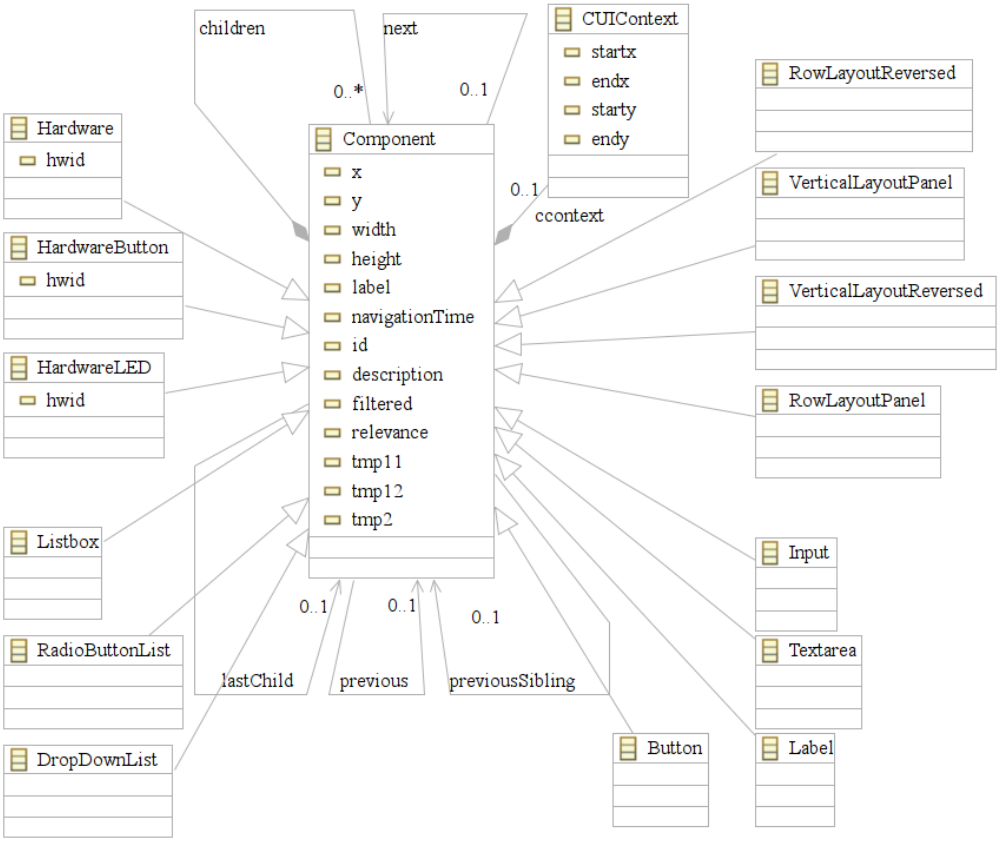


Abbildung 3.3: Beispielhaftes, konretes Meta-Modell

3.2.3 Constraint-Solving und qualitative Regeln

Werden verschiedene Eigenschaften eines Modells berechnet, sollen qualitative Regeln als Constraints in der Berechnung berücksichtigt werden. Die Idee, Regeln zur Benutzbarkeit generierter Benutzerschnittstellen als Constraints zu formulieren, wurde bereits im Jahre 1964 im Projekt SketchPad [155] verfolgt und hat daher eine lange Tradition in der automatischen Generierung von Benutzerschnittstellen.

Um ein System von Constraints zu entwickeln, werden zuerst die Regeln identifiziert, die nachher vom System unterstützt werden sollen. Hierzu werden z.B. Designer befragt oder es werden Regeln aus Style Guides extrahiert.

Bei dieser Vorgehensweise muss eine qualitative Regel in der generierten Benutzerschnittstelle erfüllt sein. Entwickler versuchen dann, sie in ein Constraint

zu übersetzen [4]. Die Constraints stellen deshalb Anforderungen an das generierte Modell, die die Anforderungen der Designer oder Style Guides umsetzen. Diese Anforderungen müssen durch die Modelltransformation eingehalten werden.

Ein prominentes Beispiel sind Mindestgrößen für Komponenten. Diese werden meist in Pixel angegeben und sind somit sehr einfach in Constraints zu übersetzen. Ein Constraint umfasst jeweils die Breite (im Metamodell: "*w*") oder die Höhe (im Metamodell: "*h*") der Komponente (im Metamodell: "*Component*") und eine Zahl, die angibt wie groß Höhe (1) oder Breite (2) mindestens sein müssen. Das folgende Beispiel ist dem Apple Style Guide [10], Seite 261 (12 entspreche der Höhe in Pixeln des Mac OS X "Regular Size" Systemfont) entnommen:

$$h \geq 12$$
$$w > 0$$

3.2.3.1 Vorteile

Ein großer Vorteil dieses Ansatzes ist, dass die Regeln relativ unabhängig voneinander deklariert werden können. Treffen mehrere Constraints auf eine Eigenschaft zu, werden diese einfach hintereinander deklariert und die Aufgabe, den richtigen Wert zu finden, bei dem noch alle Constraints gelten, an den Solver übertragen. Der Entwickler macht durch die Constraints nur Zielvorgaben an den Solver. Dadurch handelt es sich um eine rein deklarative Darstellungsform, denn es müssen keine Algorithmen entwickelt werden. Der Prozess der Lösungsfindung ist für die Entwicklung von Constraints nicht relevant, solange auf Laufzeiten keine Rücksicht genommen werden muss (also z.B. bei kleinen Lösungsräumen).

3.2.3.2 Nachteile

Nachteile sind die schlechten Möglichkeiten, das Constraint System zu untersuchen, ("debuggen", [105]) insbesondere, wenn das Constraint System schlechte Ergebnisse liefert, sowie die geringe Vorhersagbarkeit des "Aussehens" der berechneten Benutzerschnittstelle ([105]). Beide Nachteile sind hier aber weniger relevant, da der Modellierer die Modelle nachher weiter verfeinern und die schlechten Ergebnisse per Hand verbessern kann. Dabei erhöht sich der Aufwand maximal zu dem

[4] Es gibt auch erste Ansätze, die darüber hinaus zulassen, dass nur eine bestimmte Menge von Constraints (oder eine, die maximale Anzahl von Elementen enthält) halten müssen ("Constraint-Hierarchies" [53], "Soft-Constraints" [20]). Diese haben sich zur Generierung von Benutzerschnittstellen allerdings noch nicht durchgesetzt, da z.B. nicht klar ist, wie "wichtig" ein oder mehrere Constraints bewertet werden sollen. Es ist daher noch erheblicher Forschungsbedarf vorhanden.

Punkt bei dem der Modellierer die komplette Schnittstelle mit Größen- und Positionsangaben versehen muss. Dadurch ist der Aufwand nicht höher als mit Transformationen ohne Constraint-Solving. Zusätzlich handelt es sich vor allem bei dem Problem der Fehlersuche um ein allgemeines Problem, das hier nicht weiter betrachtet werden soll. Es existieren dazu bereits Ansätze (z.B. spezielle Editoren für Constraints, s. [71, 72, 43]), die es dem Entwickler vereinfachen, Fehler im Constraint System zu finden. Die geringe Vorhersagbarkeit ist vielen bekannten Systemen zur Generierung von Benutzerschnittstellen inhärent ([105]). In der vorliegenden Arbeit wird dieser Nachteil wegen der hohen Deklarativität billigend in Kauf genommen.

3.2.3.3 Minimal unterstütze Constraints

Um die Tauglichkeit der im nächsten Kapitel entwickelten, allgemeinen Constraint-relationalen-Transformationssprache für Modelle von Benutzerschnittstellen zu testen, sollen mehrere Transformationen entwickelt werden, die häufig verwendete Constraints benutzen. Dabei soll es sich um Constraints für grafische Benutzerschnittstellen handeln, denn noch immer werden grafische Benutzerschnittstellen häufiger implementiert als Benutzerschnittstellen für andere Modalitäten. Die Constraints erscheinen auf den ersten Blick relativ trivial, denn sie werden von den meisten Constraint-Solvern unterstützt. Ein Vergleich mit einer anderen neueren Transformationssprache, die eine große Menge fester qualitativer Regeln für Layouts grafischer Benutzerschnittstellen implementiert und damit einer großen Menge von Constraints, erfolgt in Kapitel 5.5. Diese Arbeit konzentriert sich bei der Implementierung verschiedener Transformationen in Kapitel 4 auf eine Menge von häufig verwendeten Constraints, die mindestens implementiert werden müssen:

1. Mindestgrößen von Komponenten setzen.
2. Größen von Container-Komponenten sollen aus den in ihnen enthaltenen Komponenten berechnet werden können.
3. Positionen sollen berechnet werden können; z.B. mit dem bereits von [47] verwendetem Verfahren.

3.2.4 Optimierung mit quantitativen Regeln

Quantitative Regeln sind besonders interessant, da sie auch eine Aussage über die erreichte Qualität der Benutzerschnittstelle bzgl. der Regel ermöglichen. Durch die Nutzung einer Zielfunktion können die generierten Modelle so optimiert werden, dass sie bzgl. der quantitativen Regel (bzw. des erzielten Wertes) optimal sind. Es werden die qualitativen Regeln weiterhin beachtet, während die Zielfunktion

nur unter den – durch die qualitativen Regeln gegebenen – Rahmenbedingungen das optimale Zielmodell sucht. Weil in optimierenden Constraint-relationale-Transformationen Zielfunktionen definiert werden können, sollten optimierende Constraint-relationale-Transformationen geeignet sein, Optimierung von Modellen von Benutzerschnittstellen mit quantitativen Regeln durchzuführen.

Im folgenden werden die bereits in Kapitel 2 vorgestellten Arbeiten von A. Sears [146] und K. Gajos aufgegriffen und verbessert.

Der Ansatz von A. Sears nutzt Fitt's Law, um die Zeit zum Bearbeiten einer Benutzerschnittstelle zu berechnen. Fitt's Law – oder vielmehr eine zwei-dimensionale Erweiterung davon – ist eine quantitative Regel zur Approximation der Zeit, die ein Nutzer in Abhängigkeit der Größe und Entfernung zum Treffen einer Fläche benötigt. Die Annahme von A. Sears ist, dass die Zeit zum Benutzen einer Benutzerschnittstelle bei einer "besser" benutzbaren Benutzerschnittstelle geringer ist. Diese Annahme wird durch die vielen, in der HCI durchgeführten, manuellen Evaluationen von Benutzerschnittstellen gestützt, die sich ebenfalls auf die Zeit zur Benutzung der Schnittstelle verlassen ("Task-Completion-Time"). Die Benutzbarkeit der Benutzerschnittstellen ist in diesen Fällen dann durch diese Zeit definiert.

Der Ansatz von Gajos verwendet ein Keystroke Level Model (KLM), um die Zeit zum Bearbeiten der Benutzerschnittstelle zu approximieren und eine Summe als Approximation für die Zeit, die zum Navigieren benötigt wird. D.h. im Ansatz von Gajos existieren insgesamt zwei Zielfunktionen: eine, die die Zeit zum Navigieren minimiert und eine zweite, die die Zeit zum Bearbeiten minimiert. Mit den bereits vorgestellten optimierenden Constraint-relationalen-Transformationen wird dies auch in Modelltransformationen machbar.

3.2.4.1 Behandlung multipler Kriterien

Ein klassisches Problem der Optimierung tritt auch bei Modelltransformation von Modellen von Benutzerschnittstellen auf: Wie können die verschiedenen quantitativen Regeln zu einem Optimum verrechnet werden? Leider ist das einfache Aufzählen der Regeln wie im Fall von Abschnitt 3.2.3 nicht möglich, denn um ein bestimmtes Ergebnis als optimal auszuzeichnen, müssen die verschiedenen Regeln (also die Zielfunktionen) ein gemeinsames Optimum bilden. Ein klassischer Lösungsansatz aus der Optimierung wird auch in dieser Arbeit verfolgt ("Utility Theory", s. [46], "Simple MultiAttribute Rating Technique" (SMART)). Dieser kombiniert die verschiedenen Zielfunktionen (quantitativen Regeln) aus der Menge $R = \{f_1, \ldots, f_n\}, n = |R|$ miteinander, indem eine Menge von Gewichten $A = \{a_1, \ldots, a_n\}$ definiert wird. Dann wird mittels

$$f^* = \sum_{i=1}^{|R|} a_i * f_i$$

das Gesamtergebnis f^* der quantitativen Regeln berechnet. Das Ergebnis kann dann entweder minimiert oder maximiert werden. Die Wahl der Gewichte obliegt dem Entwickler der Transformation oder dem Anwender der Transformation. Mit a_i wird festgelegt, wie wichtig ein Anteil einer Zielfunktion f_i (also einer quantitativen Regel) am Gesamtergebnis f^* ist.

Zur Evaluation, ob die im nächsten Kapitel entwickelte optimierende Constraint-relationale-Transformationssprache auch zur Transformation und Optimierung von Modellen von Benutzerschnittstellen eingesetzt werden kann, sollen verschiedene Optimierungsprobleme, die bei der Transformation von Benutzerschnittstellen eine Rolle spielen, implementiert werden. Tatsächlich wird in Kapitel 4 eine ganze Reihe von Transformationen bereitgestellt, die Optimierung erfolgreich mit verschiedenen Zielfunktionen und unterschiedlichen Zielmodellen von Benutzerschnittstellen nutzen. Die vielen Beispiele und Zielfunktionen zeigen die Flexibilität und Ausdrucksstärke des hier vorgestellten Ansatzes.

3.2.5 Scheduling bei Modellen von Benutzerschnittstellen

Die in Abschnitt 3.1.5.2 bereits vorgestellten Scheduling-Probleme sind Probleme, bei denen Ressourcen einer Menge von Prozessen zugeordnet werden. Diese Art von Problem existiert auch bei der Erzeugung von Benutzerschnittstellen. Eine abstrakte Komponente, d.h. z.B. ein “Interactor” (s. Meta-Modell in Abbildung 3.2), kann nämlich als Ressource verstanden werden, die von verschiedenen konkreten Komponenten, d.h. z.B. “Components” (s. Meta-Modell in Abbildung 3.3), implementiert wird, die die Prozesse darstellen. Es soll also eine gültige oder sogar optimale Zuordnung von “Components” zu “Interactors” gefunden werden.

Ein Beispiel soll genauer untersucht werden. Hierbei werden für eine abstrakte Komponente – z.B. Listen-Eingabe – konkrete Komponenten gesucht, die diese abstrakte Komponente implementieren – z.B. eine Listbox oder eine Drop-Down-Liste. Die beiden konkreten Komponenten implementieren prinzipiell den gleichen Prozess – die Auswahl eines Objektes in einer Liste von Objekten. Dies ist ein Scheduling-Problem denn sie benötigen unterschiedlich viel Platz auf der Ressource Bildschirm. So können sie normalerweise nicht beliebig gegeneinander ausgetauscht werden, sondern sollten abhängig vom zur Verfügung stehenden Platz gewählt werden: Eine Drop-Down-Liste braucht relativ wenig Platz, aber Benutzer können nicht so viele Einträge auf einmal sehen, bzw. diese müssen erst einmal auf die Drop-Down-Liste klicken. Die Listbox verbraucht in der Regel viel mehr Platz, aber

dadurch können Benutzer besser die vorhandenen Objekte in der Liste einsehen. Sie ist also bei zur Verfügung stehendem Platz prinzipiell zu bevorzugen. Das Beispiel lässt sich leicht zu einer allgemeinen Problemstellung für Typen von Komponenten erweitern (z.B. Texteingabe → { Textfeld, Textfläche }, oder Beschriftung einer Komponente → { Label, Tooltip }).

Zur Evaluation, ob die erstellte Transformationssprache auch zum Lösen von Scheduling-Problemen von Benutzerschnittstellen einsetzbar ist, soll das genannte Beispiel die Implementierung von abstrakten Komponenten (z.B. Listen-Eingabe) durch konkrete Komponenten möglich sein und in einer Transformation implementiert werden.

3.2.6 Strategien zur Generierung von Benutzerschnittstellen

Durch die Flexibilität, die eine Transformationssprache für die Transformation von Modellen von Benutzerschnittstellen bietet, ist der Aufwand besonders gering, verschiedene Methoden und Ansätze zur Generierung von Benutzerschnittstellen auszuprobieren. Entwickler können sich dann für die Methode entscheiden, die für ihre Modelle von Benutzerschnittstellen die besten Resultate liefert. Ein Ansatz besteht dabei aus einer Menge von qualitativen Regeln (Constraints), sowie einer Menge von quantitativen Regeln (Zielfunktion(en)). Die qualitativen Regeln schränken die Menge möglicher Benutzerschnittstellen ein; d.h. je restriktiver die Constraints gewählt werden, desto weniger verschiedene Benutzerschnittstellen können erzeugt werden. Die qualitativen Regeln wählen dann aus den möglichen Benutzerschnittstellen die "Beste" oder "Passende" aus. Begünstigt die Kombination dieser Mittel ein bestimmtes Layout, ist die Kombination aus Sicht der Entwickler der Transformationen eine Strategie zum Aufbau einer Benutzerschnittstelle. Die folgende Definition wird in den folgenden Kapiteln wieder aufgegriffen:

Begriffsdefinition 3.18. Strategien zur Erzeugung von Benutzerschnittstellen *sind Kombinationen von Derivaten, Constraints und Zielfunktionen, die ein bestimmtes Layout einer Benutzerschnittstelle propagieren.*

Es sollen mit der erstellten Transformationssprache mehrere Benutzerschnittstellen erzeugt werden können, die trotz weniger Änderungen (Änderung der Strategie) einer zu implementierenden Transformation relativ starke Änderungen im Layout der Benutzerschnittstellen zeigen sollten.

3.3 Zusammenfassung

In diesem Kapitel wurden theoretische Grundlagen für deklarative Transformationssprachen entwickelt, die Constraint-Solving, Zielfunktionen und (oder) Derivate implementieren. Das Konzept umfasst die Nutzung von Constraints, Zielfunktionen und Derivatproblemen als strukturelle Konzepte in der Transformationssprache, insb. zum Setzen von Werten von Attributen, auch über Assoziationen hinweg und die Bestimmung der Klasse eines Modellelements im Zielmodell.

Hierbei wurden folgende wissenschaftliche Beiträge erzielt:

- Constraint-relationale-Transformationen sind eine Erweiterung aktuell bekannter Transformationssprachen. Sie erlauben das Setzen von Attributwerten nicht nur durch Gleichungen sondern auf Grund von Constraints und können Constraint Propagation zur Steigerung der Effizienz nutzen.

- optimierende Constraint-relationale-Transformationen erweitern relationale Transformationssprachen um die Möglichkeit in Transformationen Optimierungsprobleme lösen zu können. Sie können zur Reduktion der Anzahl der Lösungen von Constraint-relationalen-Transformationen genutzt werden.

- Derivatprobleme erweitern Transformationen um eine einfache Möglichkeit verschiedene Klassen und Constraints für dasselbe Element des Zielmodells zu verwenden.

Durch diese, in Kapitel 3 erarbeiteten, Erweiterungen bisheriger Transformationssprachen wird die Anwendung von Algorithmen zur Generierung von Benutzerschnittstellen mittels Optimierung zur Generierung von Modellen von Benutzerschnittstellen selbst für deklarative Transformationssprachen unterstützt. Hierbei werden zur Verbesserung der Benutzbarkeit Constraints für die Definition von Transformationen mit qualitativen Regeln als auch Zielfunktion(en) für die Definition von Transformationen mit quantitativen Regeln genutzt. Derivatprobleme erlauben die Definition einer Abbildung von abstrakten Komponenten der Benutzerschnittstelle auf konkrete. Zusammen bilden diese Konzepte eine Strategie zur Erzeugung von Modellen von Benutzerschnittstellen.

Folgende neue Begriffe wurden in diesem Kapitel definiert und Eigenschaften sowie Algorithmen zur Implementierung untersucht:

- Constraint-relationale-Transformationen

- Optimierende Constraint-relationale-Transformationen

- Derivatprobleme

- Strategien zur Erzeugung von Benutzerschnittstellen

Nach der Entwicklung theoretischer Grundlagen kann in Kapitel 4 eine Transformationssprache entwickelt werden, die die Konzepte umsetzt. Basierend auf dieser Sprache können beispielhafte Transformationen zum Beweis der Umsetzbarkeit der Anforderungen aus diesem Kapitel deklariert werden.

Kapitel 4

Eine Transformationssprache zur Transformation von Modellen von Benutzerschnittstellen

Überblick

In diesem Kapitel wird in Abschnitt 4.1 die Transformationssprache "Solverational" vorgestellt. Solverational ist eine optimierende Constraint-relationale-Transformationssprache mit Unterstützung für Derivatprobleme. Es wird eine abstrakte, eine textliche und eine grafische Syntax vorgestellt und erläutert wie Solverational die Anforderungen aus den Kapiteln 2 und 3 erfüllt (s. Abbildung 4.1 für eine Einordnung des Kapitels in die gesamte Arbeit). Zusätzlich werden eine Architektur für einen Interpreter für Solverational (s. Abschnitt 4.2) und eine Erweiterung in Form von Schritten, die in verschiedene Methodiken zur Entwicklung von Software mit Modell-zu-Modell-Transformationen integriert werden können, vorgestellt (s. Abschnitt 4.4).

Ausgehend von dieser Transformationssprache wird ermittelt, welche Klasse von Transformationen zur Transformation von Modellen von Benutzerschnittstellen unterstützt wird (s. Abschnitt 4.5). Dazu werden einige Transformationen identifiziert, die Modelle verschiedener Benutzerschnittstellen transformieren können. Zusätzlich wird gezeigt, dass nicht nur die Transformation von Modellen von Benutzerschnittstellen unterstützt wird.

4.1 Transformationssprache

Im Rahmen der vorliegenden Arbeit wurde eine optimierende Constraint-relationale Tranformationssprache entwickelt: "Solverational". Diese implemen-

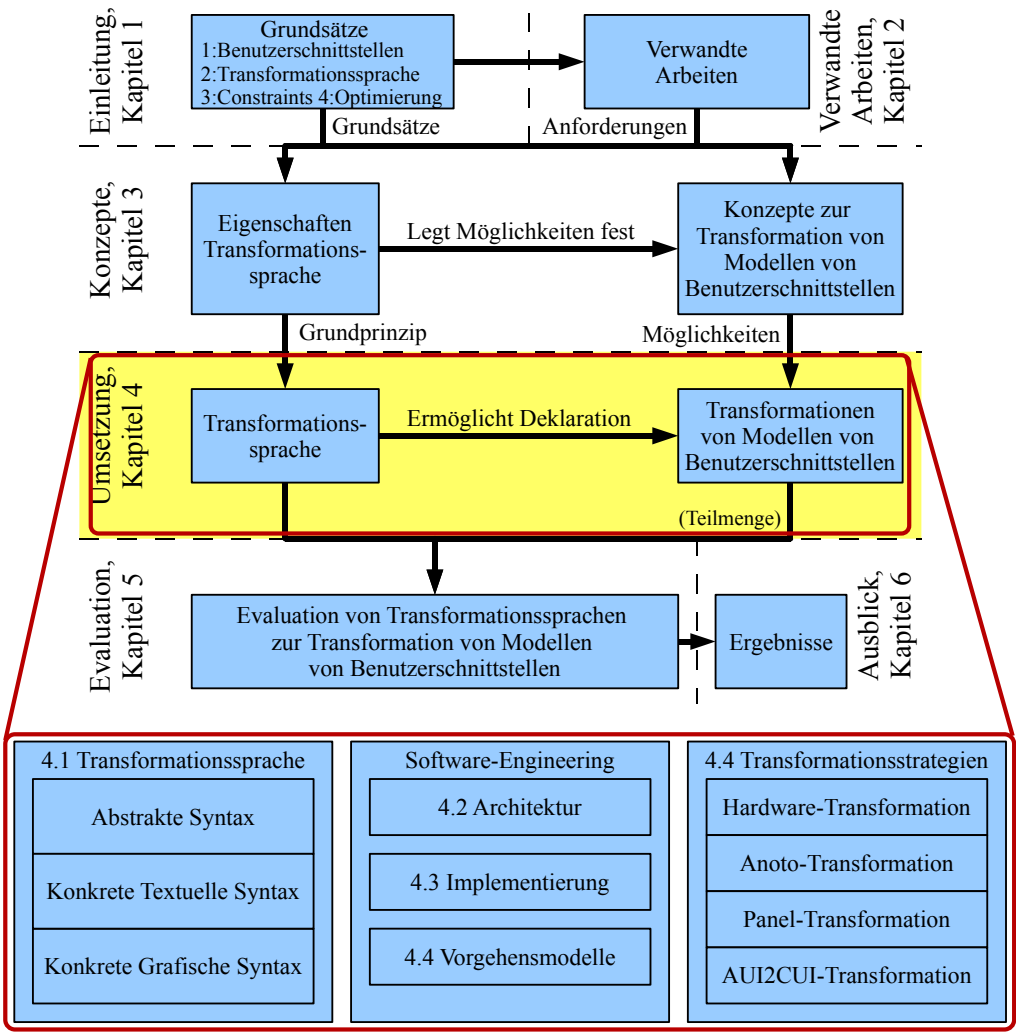


Abbildung 4.1: Aufbau Dissertation, Kapitel 4 hervorgehoben

tiert die Konzepte aus dem vorigen Kapitel. Sie basiert auf der Sprache **QVT Relations** [117].

4.1.1 QVT Relations

QVT Relations ist eine der drei Modell-zu-Modell-Transformationssprachen aus dem QVT Standard der OMG. Bei der Definition der Sprache lag der Fokus dar-

auf, eine im Verhältnis zu QVT Core benutzbare, deklarative Transformationssprache zu erhalten. QVT Relations ist damit die einzige standardisierte, deklarative Transformationssprache, die zur Programmierung durch Entwickler von Transformationen gedacht ist[1].

Zum besseren Verständnis der Arbeit sei hier eine kurze Einführung der Sprache gegeben. Der folgende Text ist daher eine Zusammenfassung von [117]. Eine ausführlichere, leicht verständliche Einführung findet sich in [111].

In QVT Relations wird eine abstrakte von zwei konkreten Syntaxen unterschieden. Alle Syntaxen können zur Definition von Transformationen verwendet werden. Dabei wird entweder direkt in der abstrakten Syntax notiert (entsprechend [19]) oder in einer der beiden konkreten Syntaxen, die dann in die abstrakte überführt werden.

4.1.1.1 Abstrakte Syntax

Die abstrakte Syntax wurde in Form eines Modells notiert. Da QVT Teil des MOF-Standards [113] ist, wurde auch die abstrakte Syntax in Form eines MOF-Meta-Modells deklariert. Die komplette abstrakte Syntax findet sich in Kapitel 7 des QVT-Standards [117].

Eine Transformation besteht in QVT Relations aus einer Menge von Relationen, genannt "relations". Eine Relation wird im Fall von QVT Relations auch als Transformationsregel[2] bezeichnet. Relationen definieren Abbildungen, die von Mengen von Modellelementen eines oder mehrerer Ausgangsmodelle auf Mengen von Modellelementen eines Zielmodells abbilden.

Eine Menge von Modellelementen wird als "Domain" bezeichnet und ist mit einer Klasse aus dem Meta-Modell und einem Muster assoziiert. Eine Domain kann entweder zur Selektion von Modellelementen in möglichen Ausgangsmodellen oder zur Deklaration von Modellelementen des Zielmodells verwendet werden:

- Bei der *Selektion* von Modellelementen beschränkt die assoziierte Klasse die Domain auf Modellelemente, die Instanzen der Klasse sind. Das Muster schränkt die möglichen Instanzen weiter ein, indem deklariert wird, welche Werte die Attribute und Assoziationen der Instanzen annehmen müssen, um in die Menge aufgenommen zu werden.

[1] QVT Core ist aus Sicht von QVT Relations vergleichbar mit einer Maschinensprache für Transformationen, und QVT Operational ist imperativ. Andere Transformationssprachen sind nicht standardisiert.

[2] Trotzdem ist eine Relation - anders als eine klassische Transformationsregel (meist "Graph-Morphismus" genannt) - auf kompletten Mengen von Modellelementen definiert und nicht auf einzelnen Modellelementen. Dadurch können aufeinander aufbauende "Regeln" mit den gleichen Ersetzungsoperationen in beiden Darstellungsformen trotzdem unterschiedliche Ergebnisse liefern.

- Bei der *Deklaration* einer Menge von Modellelementen des Zielmodells legt die assoziierte Klasse fest, von welchem Typ alle Instanzen (Modellelemente) sind. Bezogen auf eine Instanz des Zielmodells (also ein Element der Zielmenge) legt das Muster fest, welche Werte und Assoziationen die Instanz im Zielmodell haben soll.

Ein Muster besteht aus mehreren sog. "PropertyTemplateItems". Diese bestehen entweder:

- Aus einem Attribut mit einem Vergleich (Selektion) bzw. einer Zuweisung (Deklaration)

- Oder einer Eigenschaft für eine Assoziation und einem Muster, um andere Modellelemente zur Assoziation auszusuchen (Selektion) oder zu verbinden (Deklaration).

Dies wird in der QVT Relations abstrakten Syntax wie folgt notiert:

- Attribute: Das Vergleichen oder Zuweisen der Werte der Attribute geschieht mittels eines Komparators, per "=" und eines OCL Ausdrucks, mit dem der Wert berechnet wird (s. [116]). Nach Auswertung des OCL Ausdrucks wird der Wert dem Attribut zugewiesen oder mit diesem verglichen.

- Assoziationen: Für die Eigenschaften, die für Assoziationen genutzt werden, kommen sog. "ObjectTemplates" zum Einsatz. ObjectTemplates sind ihrerseits wieder Muster für Modellelemente, die mit den Modellelementen der Domain assoziiert sind. In ObjectTemplates können wieder PropertyTemplateItems genutzt werden. Dadurch können Anforderungen an die assoziierten Modellelemente gestellt und deren Werte verglichen oder gesetzt werden.

Um Abhängigkeiten darzustellen, können Relationen mit anderen Relationen verknüpft werden. Dies bewirkt eine Modularisierung der Transformation und wird mittels "when"- und "where"-Klauseln realisiert. Dazu werden die Relationen in sog. "when"- und "where"-Klauseln anderer Relationen eingetragen. Die "when"-Klausel bewirkt, dass eine Relation nur hält, wenn die in der "when"-Klausel eingetragenen Relationen ebenfalls halten. Die "where"-Klausel erfordert, dass auch die in der "where"-Klausel eingetragenen Relationen halten werden. Deshalb werden "when"-Klauseln zur Einschränkung der Selektion von Modellelementen verwendet, während "where"-Klauseln häufig zur Modularisierung der Zuweisung von Werten zu Attributen verwendet werden.

4.1.1.2 Konkrete Syntaxen

Für QVT Relations existiert sowohl eine textliche als auch eine grafische Syntax, genannt "textuelle Notation" bzw. "grafische Notation". Beide Syntaxen werden zur Ausführung in die abstrakte Syntax transformiert. Dabei sind die Namen der nicht-terminalen Symbole (engl. "nonterminal symbols") der textuellen Notation an die Namen der Modellelemente der abstrakten Syntax angelehnt. Die grafische Notation lehnt sich hingegen an UML Objektdiagramme [114] an und verwendet zum Teil Elemente der textuellen Notation.

Die textuelle Notation ist im Standard durch eine Grammatik in Backus-Naur-Form (BNF) definiert. Häufig wird diese Syntax synonym mit dem Begriff QVT Relations assoziiert, was die Dominanz der textuellen Syntax unterstreicht. Zum weiteren Verständnis des nächsten Kapitels seien die wichtigsten Symbole hier erläutert. Eine Transformation (nicht-terminales Symbol: "transformation") kann demnach entsprechend der abstrakten Syntax mehrere Relationen deklarieren. PropertyTemplateItems werden direkt als solche abgebildet (nicht-terminales Symbol: "propertyTemplate") und können "=" zur Zuweisung von Ergebnissen von OCL Ausdrücken an Attribute verwenden. Relationen können mehrere Domains umfassen, für jedes Ausgangsmodell jeweils genau eine, sowie genau eine für das Zielmodell (nicht-terminales Symbol: "domain").

Die grafische Notation wird in der Literatur sehr selten verwendet. Die Syntax wird im QVT Standard nur sehr knapp vorgestellt. Um eine interpretierbare Sprache zu erhalten, muss immer noch ein großer Teil der Sprache textlich notiert werden. Es handelt sich bei der grafischen Notation um eine Erweiterung der UML Objektdiagramme, die um ein Symbol zur Definition von Transformationen erweitert wurden. Zusätzlich können "when"- und "where"-Klauseln definiert werden, die textlich notiert werden.

4.1.2 Solverational

Die hier definierte Modell-zu-Modell-Transformationssprache "Solverational" erweitert die Syntaxen aus QVT Relations um Constraints, Zielfunktionen und Derivatprobleme. Dabei wird QVT Relations zuerst zu einer Constraint-relationalen-Transformationssprache weiterentwickelt (s. Kapitel 3). Danach wird durch Hinzufügen von Sprachkonstrukten zur Definition von Zielfunktionen eine optimierende Constraint-relationale-Transformationssprache definiert. Derivatprobleme ergeben sich durch die Erweiterung um "alternative Domains".

4.1.2.1 Constraint-relationale-Transformationssprache

Da in QVT Relations PropertyTemplateItems die Werte von Attributen nur durch Gleichheit ("=") belegen können, ist QVT Relations keine Constraint-relationale-Transformationssprache. Aus diesem Grund wird die Sprache um die Komparatoren $<$, $\leq$, $>$, $\geq$ und $\neq$ erweitert. Durch die Verwendung solcher Komparatoren werden Attribute nicht mehr fest mit einem möglichen Wert belegt, sondern können mit einem beliebigen Wert aus einer durch die Komparatoren definierten Menge belegt werden.

Constraints Im Rahmen von QVT Relations hat es keinen Sinn, das gleiche Attribut mit PropertyTemplates zwei Mal zu belegen, da dabei entweder der gleiche Wert verwendet werden muss oder ein Widerspruch entsteht, wenn gleichzeitig ein anderer Wert zugewiesen wird. Werden allerdings Komparatoren verwendet, um ein Attribut einzuschränken, sind entsprechend viele verschiedene Werte möglich und es ist sogar sinnvoll die Anzahl der Möglichkeiten noch weiter zu beschränken, um ein möglichst eindeutiges Ergebnis zu erhalten.

Solverational Es ist mit Solverational möglich beliebig viele Einschränkungen auf einem Attribut zu definieren, solange die Einschränkungen sich nicht widersprechen. Aus diesem Grund ist es auch in Solverational erlaubt, Attribute mehrmals einzuschränken, d.h. in einem Muster darf ein Attribut auf der linken Seite eines PropertyTemplates mehrfach vorkommen.

Beispiel Als Beispiel sei das Setzen von Mindest- und Maximalgrößen für Komponenten von Benutzerschnittstellen erwähnt. Zu kleine Buttons werden leicht übersehen und sehr große Buttons sind für Benutzerschnittstellen eher ungewöhnlich, weshalb sie der Konsistenz wegen generell vermieden werden sollten. Im Beispiel soll die Breite ("width", s. Abbildung 3.3) von modellierten Komponenten der Klasse Button ("Button", s. Abbildung 3.3) nach unten (> 40) und oben (< 250) begrenzt werden:

```
domain a b:concreteModel::Button {
        width > 40,
        width < 250
}
```

Zuweisungen In QVT Relations können auf der rechten Seite Variablen vorkommen, diese werden belegt, wenn sie noch nicht belegt wurden. Es handelt sich also um ein Konzept sehr ähnlich der aus der logischen Programmierung bekannten Unifikation. Ist die Variable bereits belegt, wird diese nicht neu zugewiesen, sondern der Wert zum Berechnen anderer Werte verwendet.

In Solverational können Zuweisungen von Werten zu Attributen zusätzlich auch dann erfolgen, wenn die Attribute nicht auf der linken Seite des Komparators von PropertyTemplateItems stehen, sondern auch wenn sie auf der rechten Seite des

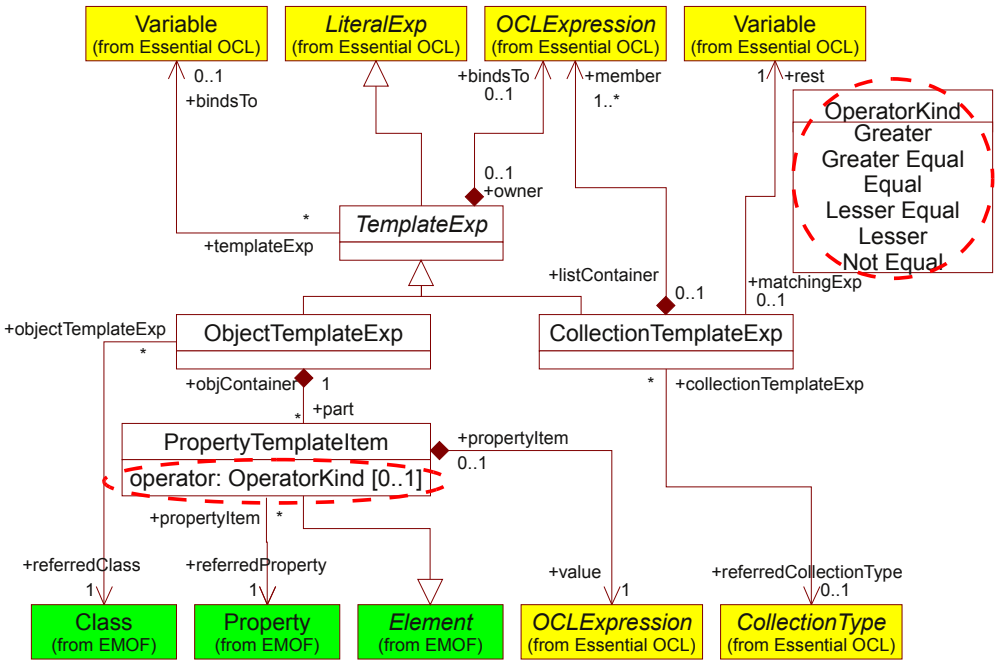


Abbildung 4.2: Abstrakte Syntax, Constraints

Komparators in einem Ausdruck vorkommen. Dies ist eine Folge des Übergangs von QVT Relations zu einem Ansatz, der Constraint-Solving unterstützt.

Anforderung 1 aus Kapitel 3 erfüllt sich automatisch aus der unterliegenden Transformationssprache QVT Relations, die bereits eine relationale Transformationssprache ist. Durch die Erweiterung um Komparatoren wird nun ebenfalls Anforderung 2 aus Kapitel 3 erfüllt.

Erweiterungen zur Abstrakten Syntax: Die abstrakte Syntax wird um die Möglichkeit erweitert, andere Komparatoren zu verwenden. Abbildung 4.2 zeigt das im QVT Relations Standard definierte Modell (s. [117]) mit den in dieser Arbeit vorgeschlagenen Erweiterungen. Das Modell basiert auf einem EMOF-Modell (Essential MOF [113]). Die Erweiterungen sind mit roten Kreisen markiert.

Zur Definition der Komparatoren wird eine Aufzählung (engl. Enumeration) "OperatorKind" eingeführt. "OperatorKind" enthält alle in Solverational verwendbaren Komparatoren. Wird ein PropertyTemplateItem instantiiert kann das Attribut "operator" auf einen entsprechenden Komparator gesetzt

werden.

Wird "operator" in einer Instanz des Modells – also einer Transformation – nicht deklariert, so handelt es sich um eine Gleichheitsrelation. Dadurch ist sichergestellt, dass die erweiterte Syntax mit der originalen Syntax kompatibel ist (dies ist möglich, da die Kardinalität von "operator" 0 oder 1 sein darf).

Erweiterungen zu den konkreten Syntaxen: Die konkreten Syntaxen werden ähnlich erweitert. In die BNF der textlichen Syntax werden die Operatoren direkt als Alternativen eingefügt. Aus der Definition des bereits vorgestellten "PropertyTemplate"

```
<propertyTemplate> ::= <identifier> '=' <OCLExpressionCS>
```

wird daher die neue Definition:

```
<propertyTemplate> ::= <identifier>
                       ('<' | '<=' | '=' | '>=' | '>' | '<>')
                       <OCLExpressionCS>
```

Die grafische Syntax muss um eine Liste von Symbolen erweitert werden, die die verschiedenen Komparatoren darstellen. Da die grafische Notation auf den UML Objektdiagrammen aufbaut, muss die betreffende Klasse aus dem UML-Standard, die UML "Instance Specification" (oder genauer dessen assoziierte "Slots"), erweitert werden (s. [114], Seite 62ff). Dies geschieht analog zur vorgestellten Erweiterung der abstrakten Syntax. Abbildung 4.3 zeigt ein Klassendiagramm mit der Erweiterung der UML "Instance Specification". Die Erweiterungen sind wieder mit roten Kreisen markiert.

Es wird eine Aufzählung "OperatorKind" eingeführt, die die möglichen Komparatoren zur Verfügung stellt. Diese Aufzählung wird in "Slot" verwendet (Attribut "operator"), der die Attribute der definierenden Klasse eines Objektes mit Werten verbindet. Dabei können nun Komparatoren verwendet werden.

Wie in der Erweiterung zur abstrakten Syntax kann auch in der grafischen Syntax der Komparator weggelassen werden. Dadurch handelt es sich implizit um eine Gleichheitsrelation, und die Syntax wird damit kompatibel mit der originalen Syntax, die keine Komparatoren unterstützt.

Erweiterungen zur Sprache OCL: OCL bietet einige klassische Mengenoperationen (d.h. Summe, Multiplikation, Zählen) auf sog. "Collections" an.

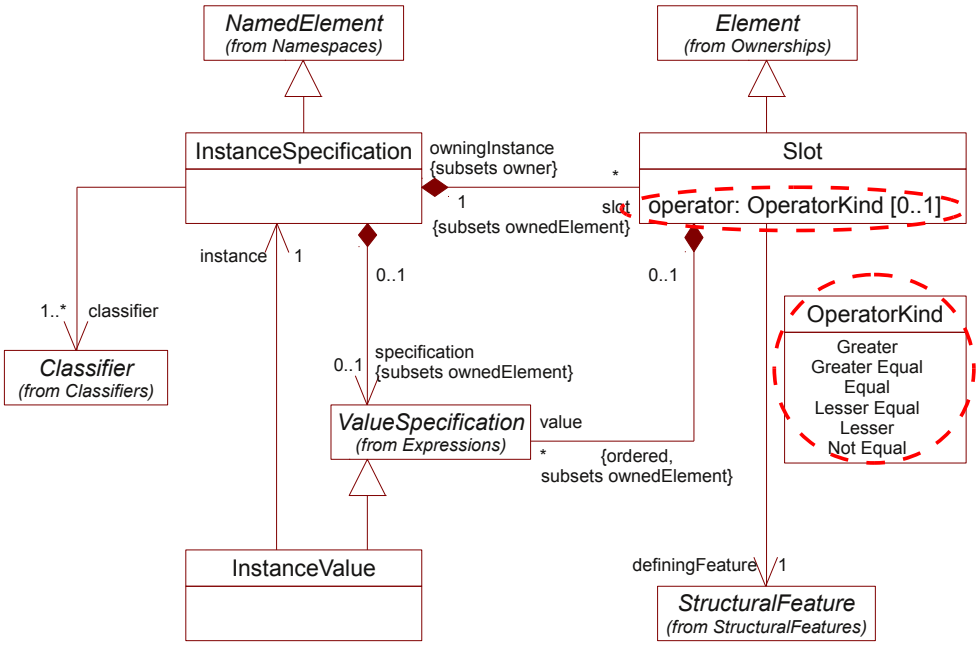


Abbildung 4.3: Grafische Syntax, Constraints

Collections sind Mengen von Modellelementen, Attributen oder Assoziationen. Klassische Mengenoperationen sind insbesondere für Constraint-Solving interessant, da mit Constraint-Solving Werte unter anderem über diese Operationen in Beziehung gesetzt werden. Die Benutzung dieser Rechenoperationen ist in OCL allerdings nur über die Bildung von sog. Collections möglich, die die betroffenen Modellelemente in einem Objekt vereinen. Auf diese Collections werden dann die Operationen wie “sum” oder “max” angewendet. Dies ist bei klassischen Constraint-Solvern nicht der Fall, denn dort werden Aggregationsfunktionen genutzt, wodurch Constraint Programmierer diese Schreibweise gewöhnt sind und die Aggregationsfunktionen intuitiv nutzen können. Aggregationsfunktionen sind wie die klassischen Mengenoperationen zu verwenden, d.h. es muss keine Collection gebildet werden. Zusätzlich ist die Abbildung auf die Aggregationsfunktionen von Contraint Solvern einfacher, wenn diese direkt abgebildet werden können. Dies vereinfacht die Implementierung der Transformationssprache, da sie direkt auf eine Constraint-

Logische-Programmiersprache abgebildet wird, s. Abschnitt 4.2.

Aus diesen Gründen bietet diese Arbeit Aggregationsfunktionen an. Diese verstehen sich als eine Erweiterung der klassischen OCL Syntax. Die Aggregationsfunktionen können ebenfalls über OCL Konstrukte wie folgt definiert werden:

Sei T ein numerischer Datentyp (in OCL sind numerische Datentypen Spezialisierungen des Typs "Real"). Für eine gegebene Collection X vom Typ T, $X = Collection(T)$, gilt:

- $max(X) : T = X- > iterate(x : T; acc : T = X- > first()|x.max(acc))$
- $min(X) : T = X- > iterate(x : T; acc : T = X- > first()|x.min(acc))$
- $sum(X) : T = X- > sum()$
- $prod(X) : T = X- > prod()$
- $count(X) : Integer = X- > size()$

4.1.2.2 Erweiterung zu optimierenden Constraint-relationalen-Transformationen

In der Sprache QVT Relations ist die Definition von Zielfunktionen nicht vorgesehen. Damit erfüllt QVT Relations nicht die Anforderungen aus Abschnitt 3.1.4.1. Aus diesem Grund wird in der vorliegenden Arbeit die Definition von Zielfunktionen in QVT Relations ermöglicht, um eine optimierende Constraint-relationale-Transformationssprache zu erhalten (Anforderung 2 aus Abschnitt 3.1.4.1 wird erfüllt). Als unterliegende Sprache wird die im letzten Abschnitt beschriebene Erweiterung von QVT Relations verwendet, genannt "Solverational", die bereits Constraints implementiert (d.h. Anforderung 1 aus Abschnitt 3.1.4.1 wird erfüllt).

In QVT Relations wird OCL verwendet. Daher werden keine neuen Anforderungen an das Wissen von Entwicklern gestellt, wenn die Sprache auch als Basis für neue Konstrukte in Solverational verwendet wird. Hier dient sie als vorgeschlagene Erweiterung zu QVT Relations für die Deklaration von Zielfunktionen.

Zusätzlich zur Deklaration der Funktion muss für eine Zielfunktion angegeben werden, ob sie minimiert oder maximiert werden soll.

Im Gegensatz zu lokalen Constraints haben Zielfunktionen an sich keinen Bezug zu bestimmten Modellelementen, also muss bei der Deklaration einer Zielfunktion eine Menge von Modellelementen beschrieben werden, auf denen Berechnungen ausgeführt werden sollen. Für die Beschreibung einer solchen Menge ist die OCL Operation "instanceOf" von Interesse, die es ermöglicht, alle Instanzen einer bestimmten Klasse des Meta-Modells zu selektieren. Ist eine solche Menge

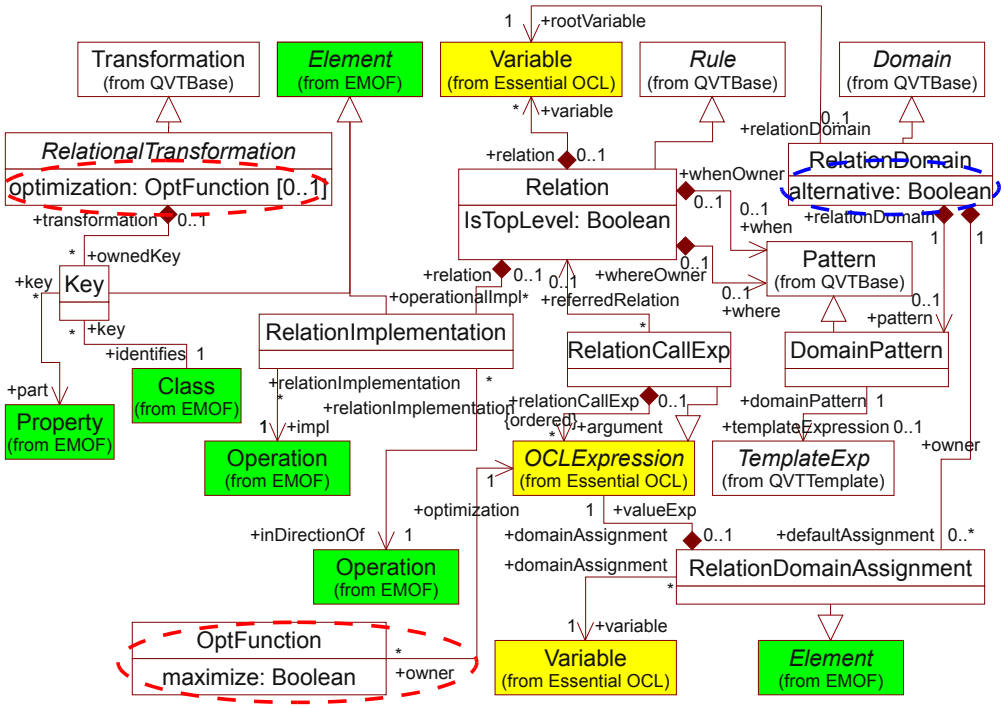


Abbildung 4.4: Abstrakte Syntax, Optimierung und Derivatprobleme

bestimmt, kann diese weiter eingeschränkt werden, und die weiter oben bereits vorgestellten Aggregationsfunktionen können zur Berechnung eines Funktionswertes aus der Menge verwendet werden. Die Aggregationsfunktionen sollen auch für Zielfunktionen Deklaration und Lesen erleichtern. Ein Beispiel findet sich in Abschnitt 4.5.3. In diesem Beispiel wird die Summe über alle Werte des Attributs "x" der Klasse "Component" gebildet.

Erweiterungen zur Abstrakten Syntax: Die Erweiterungen, die zur Einführung von Zielfunktionen notwendig sind, werden in Abbildung 4.4 dargestellt. Die Erweiterungen sind mit roten Kreisen markiert. Die Klasse "RelationalTransformation" wird um das Attribut "optimization" erweitert, das eine Assoziation zu einer Instanz der Klasse "OptFunction" enthalten kann. Dadurch kann maximal eine Zielfunktion pro Transformation angegeben werden. Die Einführung weiterer Zielfunktionen (sog. Vektoroptimierungsprobleme) erschwert das Finden einer optimalen Lösung, sodass darauf verzichtet wird. Da alle hier berücksichtigten Optimierungs-

probleme nur eine Zielfunktion benötigen werden, erfordert dies keine Einschränkung der Ziele der Arbeit.

Ein Objekt der Klasse "OptFunction" enthält die Information, ob die Zielfunktion entweder maximiert oder minimiert werden soll (wenn Attribut "maximize" auf "false" gesetzt ist, wird die Zielfunktion minimiert, ansonsten maximiert). Zur Deklaration einer OCLExpression, die die Zielfunktion definiert, enthält die Klasse "OptFunction" eine Assoziation "optimization". Es ist bei der Deklaration der Zielfunktion darauf zu achten, dass die Zielfunktion einen numerischen Wert als Ergebnis liefert. Ist dies nicht der Fall, kann nicht optimiert werden, auch wenn der deklarierte Ausdruck der Grammatik entspricht.

Erweiterungen zu den konkreten Syntaxen: Die textliche Syntax wird um Zielfunktionen erweitert. Die Klasse "RelationalTransformation" kann nach der abstrakten Syntax eine Zielfunktion erhalten. Entsprechend wird in der konkreten Syntax

```
<transformation> ::= 'transformation' <identifier>
    '(' <modelDecl> (',' <modelDecl>)* ')'
    ['extends' <identifier>]
    '{' <keyDecl>* ( <relation> | <query> )* '}'
```

zu

```
<transformation> ::= 'transformation' <identifier>
    '(' <modelDecl> (',' <modelDecl>)* ')'
    ['extends' <identifier>]
    '{' [<optFunction>] <keyDecl>*
    ( <relation> | <query> )* '}'
```

wobei gilt:

```
<optFunction> ::= ('maximize' | 'minimize')
    <OCLExpressionCS> ';'
```

Das Symbol "transformation" hat nun die bereits aus der abstrakten Syntax bekannte Möglichkeit eine Zielfunktion zu deklarieren: "optFunction" kann angegeben werden, die wiederum eine "OCLExpressionCS" enthalten muss. Durch Angabe von "maximize" oder "minimize" wird "OCLExpressionCS" entweder maximiert oder minimiert.

Abbildung 4.5: Grafische Syntax, Optimierung.

Die Erweiterung der grafischen Syntax erfolgt analog: das in QVT Relations definierte Symbol für Transformationen (s. [117], Tabelle 7.1) wird um die Möglichkeit erweitert, eine Zielfunktion zu definieren. Abbildung 4.5 zeigt links des blauen Pfeils das originale grafische Symbol und rechts das neue Symbol mit der Möglichkeit, eine Zielfunktion anzugeben.

Um eine Transformation in grafischer Syntax zu deklarieren wird das neue Symbol mit der entsprechenden Zielfunktion bestückt und als Ersatz für das originale Symbol eingesetzt.

Erweiterungen zur Sprache OCL: Zusätzlich zu den Erweiterungen für Constraint-relationale-Transformationen waren keine erforderlich. Die Aggregationsfunktionen können in der Erweiterung zu optimierenden Constraint-relationale-Transformationen zur Deklaration von Zielfunktionen verwendet werden, da diese bereits für die Unterstützung von Constraint-relationale-Transformationen in OCL definiert wurden.

4.1.2.3 Erweiterung zu Derivatproblemen

Derivatprobleme müssen zwei Anforderungen erfüllen:

1. es muss die Möglichkeit geben, für ein Modellelement des Ausgangsmodells mehrere Typen im Zielmodell zu definieren und

2. als Basis muss eine Constraint-relationale-Transformationssprache oder eine optimierende Constraint-relationale-Transformationssprache verwendet werden.

Um dies zu realisieren wird die im vorigen Abschnitt entwickelte Sprache um sog. “alternative Domänen” erweitert (engl. “alternative domain”).

Wie im Abschnitt zu QVT Relations (s. 4.1.1) beschrieben, werden Domains verwendet, um die Menge der Modellelemente einzuschränken. In QVT Relations wird hierzu zuerst eine Klasse mit der Domain assoziiert. Im Zielmodell führt das dazu, dass alle Modellelemente vom Typ der Klasse erzeugt werden.

Die vorgeschlagene Erweiterung ermöglicht die Zuordnung einer Menge von mehreren Klassen für das gleiche Modellelement des Zielmodells. Dadurch sind Entwickler von Transformationen nicht auf einen Typ festgelegt, sondern können eine endliche, abzählbare Menge von Klassen aus dem Meta-Modell des Zielmodells auswählen[3].

Es kommt häufig vor, dass Constraints in einer Transformation von Typen abhängig sind, d.h. dass nicht nur die Typen unterschiedlich sind, sondern auch wie die Werte für die unterschiedlichen Attribute der Typen berechnet werden. Beispielsweise soll ein Button mindestens eine Zeile hoch sein (wenn er mit Text beschriftet wird), während eine Liste meist wesentlich höher sein sollte, da mehrere Zeilen dargestellt werden müssen. Um dies zu realisieren, ist es in der vorliegenden Erweiterung vorgesehen, unterschiedliche Constraints bei Verwendung verschiedener Klassen verwenden zu können.

Erweiterungen zur Abstrakten Syntax: Die abstrakte Syntax wird um ein boolesches Attribut "alternative" der Klasse "RelationDomain" erweitert. Domains gelten dann als Alternativen für die Erzeugung von Modellelementen im Zielmodell, wenn sie einerseits Teilmenge des Zielmodells sind (d.h. das Attribut "typedModel" zeigt auf das gleiche Modellelement vom Typ "TypedModel", das in der aktuellen Transformation als Zielmodell verwendet wird) und andererseits "alternative" auf "true" gesetzt ist.

Die abstrakte Syntax wird wie in Abbildung 4.4 blau markiert erweitert. Man erkennt das neue Attribut "alternative" am Element "RelationDomain", welches zur Deklaration von Domains in QVT Relations verwendet wird.

Erweiterungen zu den konkreten Syntaxen: Die textliche Syntax wird analog erweitert. Das nicht-terminale Symbol "domain" wird um die Möglichkeit erweitert, die Domain als "alternative" zu deklarieren. Dazu wird das Schlüsselwort direkt in der Deklaration der Domain angegeben.

Aus

```
<domain> ::= [<checkEnforceQualifier>] 'domain'
    <modelId> <template>
    ['implementedby' <OperationCallExpCS>]
    ['default_values' '{' (<assignmentExp>)+ '}'] ';'
```

wird:

[3]Dies stellt eine Beschränkung der Allgemeinheit dar. Allerdings können im Moment für die Praxis relevante Transformationssprachen nur endliche Modelle transformieren.

```
<domain> ::= [<checkEnforceQualifier>] ['alternative'] 'domain'
    <modelId> <template>
    ['implementedby' <OperationCallExpCS>]
    ['default_values' '{' (<assignmentExp>)+ '}'] ';'
```

Die grafische Syntax wird formal nicht erweitert. Die Erweiterung besteht darin, dass auf der Seite des Transformationssymbols (s.o.) auf der das Zielmodell notiert wird mehrere Domains verwendet werden können. Diese werden dann als "alternative domains" erkannt.

Erweiterungen zur Sprache OCL: Die Sprache OCL muss zusätzlich zu den bereits eingeführten Erweiterungen nicht erweitert werden.

4.2 Architektur zur Interpretation

Im Rahmen dieser Arbeit wurde eine Architektur eines Systems entwickelt, mit der die vorgestellte Transformationssprache interpretiert werden kann. Es kommen zwei grundsätzlich verschiedene Ansätze in Frage:

1. Architektur für einen Interpreter
2. Architektur für einen Compiler

Es wird eine Architektur für einen Compiler vorgestellt (diese Entwurfsentscheidung wird in den Unterabschnitten 4.2.1 und 4.2.2 diskutiert). Zusammen mit der im folgenden Abschnitt vorgestellten Implementierung dient sie als ein Beispiel zur Demonstration, dass eine Implementierung möglich ist. Die Architektur diente als Grundlage zur Implementierung eines Interpreters. Um eine allgemeinere Form der Architektur zu erhalten, enthält diese noch keine Bezüge zu verwendeten Plattformen oder Frameworks.

Abbildung 4.6 zeigt einen Überblick über die vorgeschlagene Architektur für eine sog. "Transformationsmaschine" zur Interpretation von Solverational. Transformationsmaschinen für die in Kapitel 1 erläuterten model-type-mappings besitzen implizit zwei Meta-Ebenen. Dies liegt daran, dass sie erstens Abhängigkeiten zwischen der Modelltransformation und den Meta-Modell-Elementen auflösen müssen und zweitens die Modelltransformation auf Instanzen der Meta-Modelle anwenden müssen.

1. Die obere Meta-Ebene erhält die Meta-Modelle zur Ein- und Ausgabe (in gewisser Hinsicht also Grammatiken für die Ausgangs- und Zielmodelle), sowie die Beschreibung der Transformation in Solverational selbst und wird deshalb Meta-Modell-Ebene genannt.

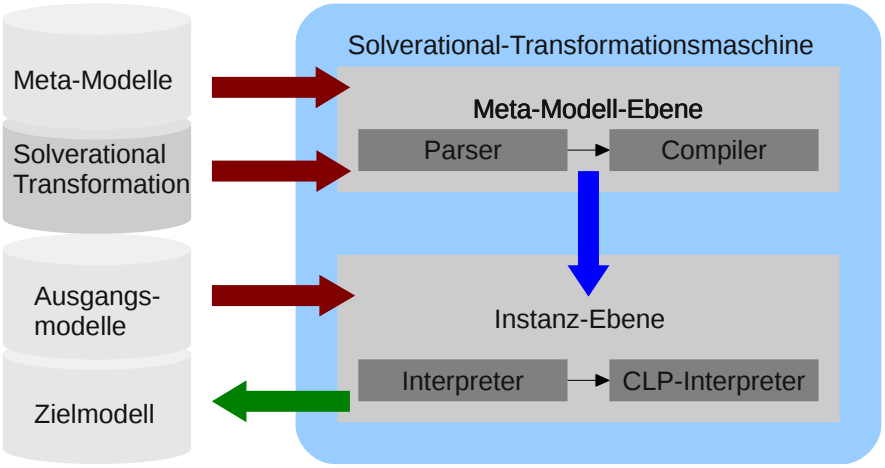


Abbildung 4.6: Architektur zur Interpretation von Solverational

2. Die untere Meta-Ebene verarbeitet und produziert Instanzen der Meta-Modelle, also die "Modelle", indem es die Transformation interpretiert und wird deshalb Instanz-Ebene genannt.

Entsprechend diesen beiden Meta-Ebenen wird die Architektur der Transformationsmaschine in zwei Schichten eingeteilt. Wie bei einem Compiler für klassische Programmiersprachen können diese Schichten später getrennt voneinander ausgeführt werden (dies wird in der Literatur als "transformation compiler" bezeichnet). Die Meta-Modell-Ebene wird zur Entwicklungszeit der Transformationen ausgeführt; die Instanz-Ebene, wenn die entsprechenden Ausgangsmodelle in Zielmodelle transformiert werden sollen.

Eingaben zur Transformationsmaschine sind mit roten Pfeilen dargestellt, Ausgaben mit grünen. Die Meta-Modell-Ebene wird mit den Meta-Modellen und der Beschreibung der Transformation in Solverational bestückt. Sie führt zunächst einen Parser aus. Der Parser erweitert seine Grammatik, mit der die Transformationssprache analysiert wird (z.B. die nicht-terminalen Symbole), anhand der Meta-Modelle. Konnte die Transformation grammatikalisch analysiert werden, gibt der Parser einen Syntaxbaum an den Compiler weiter. Der Compiler bildet diesen auf ein ausführbares Programm in einer unterliegenden Programmiersprache ab. In der vorliegenden Arbeit wurde eine Constraint-Logische-Programmiersprache (häufig

auch mit "Constraint Logische Programmierung/Programmiersprache (CLP)" bezeichnet) als unterliegende Programmiersprache verwendet (s. Abschnitt 4.2.3). Dies setzt Anforderung A4 aus Kapitel 2 um.

Das vom Compiler erzeugte Programm wird Teil der Instanz-Ebene (s. blauer Pfeil). Diese erhält die zu transformierenden Modelle als Eingabe (also Instanzen der Meta-Modelle). Der Interpreter formt die Modelle so um, dass sie als Eingabe für das erzeugte Programm verwendet werden können. D.h. das eine für die Constraint-Logische-Programmiersprache interpretierbare Darstellung entsteht (z.B. eine Term-basierte Darstellung). Der CLP-Interpreter führt das Programm aus und nutzt die neu entstandene Darstellung der Modelle als Eingabe. Als Ergebnis entsteht wiederum ein Modell in einer speziellen Darstellungsform (z.B. wieder eine Term-basierte Darstellung des Zielmodells). Am Schluss wird das Zielmodell durch den Interpreter in eine für Entwickler intuitiv verständliche Darstellungsform für Modelle überführt und als Zielmodell zur Verfügung gestellt.

4.2.1 Vorteile der Architektur

Die vorgestellte Architektur kompiliert die Transformation in ein Constraint-Logisches-Programm. Es existieren damit für Entwickler von Transformationen zwei Möglichkeiten, Änderungen an Transformationen durchzuführen: In Solverational und in der Constraint-Logischen-Programmiersprache. Dadurch können die Teile der Transformation, die in der Constraint-Logischen-Programmiersprache besser entwickelt werden können, auch in dieser entwickelt werden. Programmierer können daher je nach ihren Fähigkeiten entscheiden, ob sie die Transformation komplett in der Transformationssprache entwickeln wollen, oder ob sie Teile der Transformation in der unterliegenden Sprache entwickeln. Dies kann insbesondere zur Verbesserung der Effizienz des generierten Quellcodes des Constraint-Logischen-Programmes nützlich sein. Das Verändern muss bei erneutem Kompilieren der Transformation allerdings wiederholt werden.

Dadurch, dass es sich um einen Compiler handelt muss die Transformationssprache nicht bei jeder Transformation neu durch den Parser und den Compiler interpretiert werden. So ist insgesamt bei mehrmaligem Ausführen eine kürzere Ausführungszeit zu erwarten. Hierbei braucht der erste Prozess für das Kompilieren zwar etwas mehr Zeit (durch das Schreiben des Programms auf die Festplatte). Allerdings ist die Ausführung nachfolgender Transformationen dann um die Zeit zum Ausführen des Parsers reduziert – dieser müsste bei einer Interpretation immer ausgeführt werden. Werden also viele Transformationen mit dem gleichen Quelltext und unterschiedlichen Ausgangsmodellen ausgeführt, ergibt sich nach einer Anzahl von Ausführungen ein Geschwindigkeitsvorteil.

4.2.2 Nachteile der Architektur

Da es sich um eine Architektur für einen Compiler handelt, ist die Ausführung von selbst-modifizierenden Transformationen nur unter hohem technischem Aufwand möglich. Selbst-modifizierende Transformationen werden normalerweise als Modelle interpretiert und zwar solche, die durch die Transformationen selbst wieder modifiziert werden können. Da im vorliegenden Fall die Transformation aber bereits in der Meta-Modell-Ebene in ein CLP-Programm übersetzt wurde, ist das Modell der Transformation und dessen Syntaxbaum zur Zeit der Interpretation nicht mehr vorhanden. Allerdings werden selbst-modifizierende Tansformationen extrem selten verwendet, da es nur wenige sinnvolle Anwendungsfälle gibt (Virenprogrammierung, Genetische Programmierung), s. [9]. Aus diesem Grund wurde die Entwicklung selbst-modifizierender Transformationen nicht weiter verfolgt.

4.2.3 Vorteile der Abbildung auf CLP

In dieser Arbeit wurde eine Constraint-Logische-Programmiersprache als Zielsprache verwendet. Diese Entscheidung beruht auf mehreren Vorteilen:

- Als mögliche Programmiersprachen kommen Sprachen in Frage, die Optimierung und Constraint-Programmierung unterstützen. Klassischerweise werden in solchen Fällen Constraint-Logische-Programmiersprachen ("CLP") verwendet, da die Integration zwischen logischen Programmiersprachen und Constraint-Programmierung hoch ist und dadurch potentiell Entwicklungszeit eingespart werden kann.

- Logiksprachen sind von Natur aus deklarativ[4]. Solverational - als eine deklarative Sprache - wird auf eine unterliegende Sprache abgebildet. Es ist günstig, diese unterliegende Sprache so zu wählen, dass die Abbildung von Solverational möglichst einfach ist. Da Logiksprachen deklarativ verwendet werden können, muss der deklarative Anteil von Solverational nicht in imperative Programme übersetzt werden, wie es z.B. bei JAVA der Fall wäre. Durch die konsistente Verwendung deklarativer Sprachen gestaltet sich daher die Entwicklung von Abbildungen deklarativer Konstrukte einfacher.

- Mehrere wissenschaftliche Arbeitsgruppen haben bereits Logiksprachen zur Implementierung von Transformationen eingesetzt (z.B. [52, 142]). Daher kann zur Implementierung der Transformationen (ohne Constraints und Optimierung) auf den Erfahrungen bestehender Arbeiten aufgebaut werden. Es muss nicht, wie bei neuen Programmiersprachen, ein völlig neues Konzept

[4]Es gibt Konstrukte, die als nicht deklarativ gelten [52], allerdings werden Logiksprachen gemeinhin trotzdem als deklarativ bezeichnet.

zur Darstellung von Modellen und Transformationen entwickelt werden. Insbesondere die Abbildung der Meta-Modelle und Modelle auf eine textuelle Sprache kann von den bestehenden Ansätzen übernommen werden.

4.2.4 Nachteile der Abbildung auf CLP

Constraint-Logische-Programmiersprachen werden in der Regel nicht direkt in Maschinensprache übersetzt, was dazu führt, dass diese nicht ganz so schnell ausgeführt werden können wie Programme in Maschinensprache. Allerdings werden heute durch die breite Verwendung von interpretierten Sprachen (z.B. Perl, Python, PHP oder JavaScript) und die mindestens ebenso breite Verwendung von Sprachen die in virtuellen Maschinen ausgeführt werden (z.B. Java, C#, Visual Basic .NET, J#, JScript .NET, C++/CLI) viele Programme nicht direkt in Maschinensprache übersetzt. Dies ist darauf zurückzuführen, dass der Nachteil der schlechteren Effizienz in der Praxis entweder gering ist, oder durch die Vorteile von interpretierten Sprachen (z.B. direktes Ausführen ohne Kompilieren) kompensiert werden. Fortschritte sowohl in der Entwicklung neuer Interpreter als auch neuer Prozessoren haben die Effizienz der Interpreter weiter verbessert.

4.3 Implementierung

Dieser Abschnitt stellt eine Implementierung der in Abschnitt 4.2 beschriebenen Architektur vor. Tatsächlich können mit der vorliegenden Implementierung Constraint-relationale-Transformationen, optimierende Constraint-relationale-Transformationen und Derivatprobleme in Constraint-Logische-Programme überführt werden. Die Architektur wurde im Rahmen dieser Arbeit wie in Abbildung 4.7 implementiert. Die Ähnlichkeit zu Abbildung 4.6 ist deutlich zu erkennen.

Die Architektur wurde in Form eines Plugins für die JAVA-Programmierumgebung Eclipse ([68]) implementiert. Dieses Plugin ist in der Lage, Modelle zu transformieren, die mittels Meta-Modellen definiert wurden. Diese wiederum liegen in EMF/Ecore ([154]) vor und müssen nicht in Code übersetzt werden ("dynamic EMF"). Die Transformationen werden in Solverational definiert. Dies bedeutet dass die Implementierung – wie andere Modell-zu-Modell-Transformationssprachen auch – unabhängig von Modellierungssprachen verwendet werden kann.

Die gesamte Implementierung basiert auf einer starken Modifikation der OCL/EMFT-Plugins (z.B. [59]), einem Plugin der Eclipse-Programmierumgebung zur Interpretation der OCL. Die Grammatik wurde stark verändert, sodass die textuelle Syntax von Solverational interpretiert werden kann. Das OCL-Modell wurde

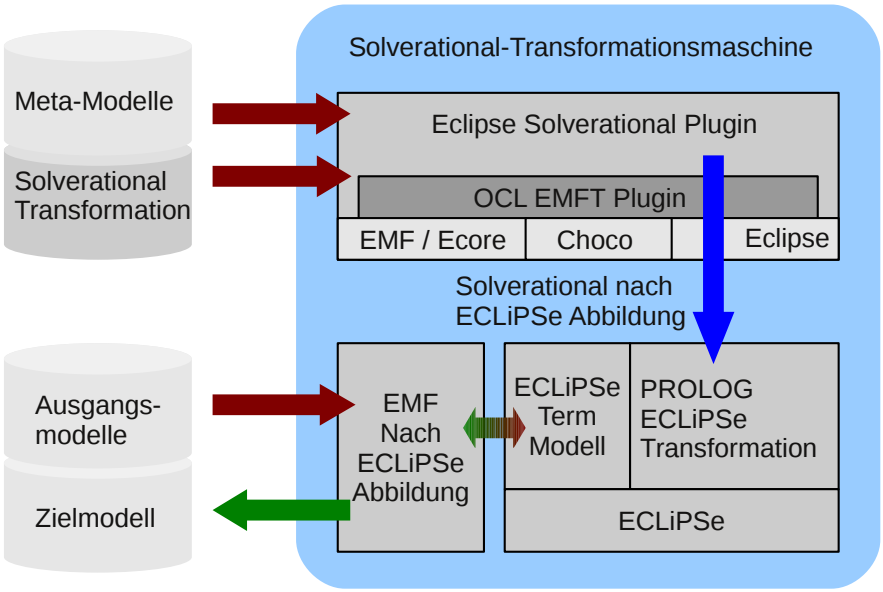


Abbildung 4.7: Architektur der Implementierung

so verändert, dass es mit der abstrakten Syntax von Solverational übereinstimmt. Die Evaluation der OCL Ausdrücke wurde komplett neu implementiert, da sie nun in eine andere Sprache übersetzt und nicht mehr direkt interpretiert werden.

Die Transformationsmaschine steht also als Teil (sog. Plugin) der Eclipse-IDE zum Entwickeln direkt zur Verfügung. Um die Flexibilität dieses Ansatzes aufzuzeigen und eine bessere Integration mit Modellierungswerkzeugen anzubieten, wurde das Plugin zusätzlich noch in die Mapache-Programmierumgebung integriert [18]. Einige Beispiele für Transformationen innerhalb dieser Programmierumgebung folgen in Kapitel 4.5.

Als Zielsprache für die Kompilierung ("Solverational-nach-ECLiPSe-Abbildung") wurde ECLiPSe ([11]) gewählt, das nicht mit dem oben genannten Eclipse verwechselt werden sollte. ECLiPSe ist eine Constraint logische Programmiersprache, die auf PROLOG aufbaut. Es existiert ein Interpreter für die Sprache, der auch für diese Implementierung genutzt wurde.

Während der Kompilierung muss die Reihenfolge zum Ausführen der Transformationen bestimmt werden. Diese wurde mit einem Constraint-Solver ("Choco"

[77]) berechnet. Dieser Ansatz ermöglicht die Ausführungsreihenfolge der Relationen im Vorfeld zu bestimmen. Dadurch kann die Ausführungsreihenfolge serialisiert werden, indem die "when"- und "where"-Klauseln in Relationen als Constraints in einer temporalen Logik betrachtet werden. Daher muss die Ausführungsreihenfolge nicht wie in anderen Ansätzen zur Laufzeit berechnet werden.

Das Eclipse-Solverational-Plugin erzeugt ein ECLiPSe-Programm ("PROLOG-ECLiPSe-Transformation"), das die kompilierte Transformation darstellt. Übergibt der Entwickler der Solverational-Transformationsmaschine eine Menge von Ausgangsmodellen, werden diese mit der "EMF-Nach-ECLiPSe-Abbildung" in eine Term-basierte Darstellung überführt ("ECLiPSe Term Modell"). Diese Darstellung kann vom erzeugten ECLiPSe-Programm verwendet werden, um wiederum ein Zielmodell in Term-basierter Darstellung zu erzeugen. Dieses wird wieder in EMF übersetzt. ECLiPSe unterstützt Constraint-Propagation, wodurch Anforderung A2 aus Kapitel 2 umgesetzt wird.

Das erzeugte ECLiPSe-Programm implementiert Algorithmen 3.1 und 3.2. Optimierung wird ebenfalls unterstützt. Das ECLIPSe-Programm wird durch den Aufruf des Terms "top_relations" gestartet. Zuerst wird dann die term-basierte Darstellung aus einer externen Datei nachgeladen. Die Terme werden daher als dynamische ECLiPSe-Terme zur Laufzeit integriert. Relationen werden in mehrere Terme aufgeteilt. Diese Unterteilung entspricht grob der Aufteilung von Algorithmus 3.1. Während der Ausführung der Terme für die Relationen werden die Constraints gesammelt und später – nach Erzeugung aller Modellelemente – in das Constraint-System eingefügt. Am Ende wird ein Constraint-Solver ausgeführt, der für das System Lösungen sucht (evtl. auch mittels eines Branch-And-Bound-Algorithmus im Falle dass es sich um optimierende Constraint-relationale Transformationen handelt).

4.3.1 Beschränkungen der Implementierung

Es wurden nicht alle OCL Ausdrücke implementiert. Insbesondere gibt es Beschränkungen bei verschiedenen "iterate"-Konstrukten. Aggregationsfunktionen implementieren entsprechende Konstrukte in anderer Form. Insofern zeigen diese, dass es einerseits kein Problem darstellt, diese zu implementieren und andererseits stehen damit bereits einige Ausdrücke zur Verfügung, die teilweise an Stelle der "iterate"-Konstrukte verwendet werden können.

Mehrere Konstrukte, die in QVT Relations verwendet werden können, stehen derzeit noch nicht zur Verfügung: Die Implementierung beherrscht keine "Collection-Templates", ähnlich zum QVT-Interpreter medini QVT (Collection-Templates werden selten verwendet).

Die Berechnung der Ausführungsreihenfolge der Relationen berücksichtigt keine rekursiven Aufrufe von Relationen. Diese werden allerdings in der Regel zur

Transformation von Modellen von Benutzerschnittstellen nicht benötigt.

Derzeit wird aus einem Ausgangsmodell ein Zielmodell erzeugt. Ein einzelnes Ausgangsmodell zu verwenden ist keine Beschränkung der Allgemeinheit, da mehrere Meta-Modelle leicht zu einem großen Meta-Modell kombiniert werden können, indem alle Meta-Modelle als sog. "Packages" (ähnlich einem Modul) eines Meta-Modells betrachtet werden. Es können keine existierenden Zielmodelle geladen und verarbeitet werden. Es wird kein Tracemodell erzeugt. Dies ist allerdings nicht schwieriger zu implementieren als für QVT Relations.

"Where"-Klauseln sind nicht implementiert. Diese werden auf Grund des Constraint-Solving Ansatzes allerdings in Solverational im Gegensatz zu QVT Relations auch meist nicht benötigt, da Abhängigkeiten zwischen Werten in einer Domain behandelt werden können.

4.3.2 Fazit

Die gezeigte Implementierung einer Solverational Transformationsmaschine ist eine Implementierung der in Abschnitt 4.2 vorgeschlagenen Architektur. Insbesondere können optimierende Constraint-relationale-Transformationen und Derivatprobleme mit dem vorgestellten Plugin in Constraint-Logische-Programme überführt werden. Dabei werden Transformationen, die in Solverational geschrieben wurden, in ECLiPSe-Programme überführt. Modelle, die in EMF/Ecore vorliegen, werden in eine Term-basierte Darstellung überführt, mit ECLiPSe transformiert und wieder in EMF/Ecore abgebildet. Dadurch konnten die Konzepte aus den Abschnitten 3.1 und 4.1 implementiert werden. Die Implementierung bietet allerdings noch zahlreiche Beschränkungen, die in einer kommerziell verwertbaren Transformationsmaschine nicht auftreten dürfen.

4.4 Integration in Vorgehensmodelle

Es existieren mehrere Vorgehensmodelle zur modell-getriebenen Entwicklung verschiedenster Software mit Modell-zu-Modelltransformationen (z.B. [4, 87, 89]). Auch speziell für den Fall der Entwicklung von Benutzerschnittstellen mit Transformationen existieren Ansätze (z.B. [94, 152, 121]). Jedoch mangelt es allen Ansätzen an Unterstützung für Constraints und Optimierung, da bisher keine Notwendigkeit bestand, diese in ein Vorgehensmodell zur modell-getriebenen Softwareentwicklung mit Transformationen zu integrieren. Da nun eine optimierende Constraint-relationale-Transformationssprache vorliegt, ist ein solches Vorgehensmodell wünschenswert, um Entwickler bei der Erstellung von Software zu unterstützen.

Besser noch als ein spezielles Vorgehensmodell ist die Möglichkeit, bestehende so zu erweitern, dass sie auch zur Entwicklung mit optimierenden Constraint-relationalen-Transformationen genutzt werden können. Dazu sollen Bausteine oder Schritte definiert werden, die sich mit bekannten Bausteinen einer Menge bestehender Methodiken integrieren lassen. In Abschnitt 4.4.3 werden die entwickelten Schritte beispielhaft auf ein spezielles Vorgehensmodell zur Entwicklung von Transformationen angewendet.

Der Ansatz kann auf Grund der Abhängigkeit von Vorgehensmodellen, die Modelltransformationen zulassen, jedoch nicht den Anspruch erheben, dass die Schritte in jedes beliebige Vorgehensmodell integriert werden können. Er beschränkt sich stattdessen auf eine Untermenge von Vorgehensmodellen, die die Entwicklung von Applikationen mittels Modelltransformationen unterstützen. Eine vollständige Aufzählung würde den Rahmen dieser Arbeit sprengen. Beispiele sind: [89], [4], [87], [94], [152], [121], ...

4.4.1 Entwicklungsphase

Bei Transformationen handelt es sich um Entwicklungswerkzeuge. Daher lässt sich die Entwicklung mit Modell-zu-Modell-Transformationen grob in zwei Unterphasen einteilen:

1. Die Entwicklung der Transformationen selbst und
2. die Entwicklung von Modellen von Applikationen mit Transformationen.

Es sind deshalb zwei verschiedene Gruppen von Entwicklern zu unterscheiden: Entwickler der Transformationen, in [1] auch als “transformation specifier” bezeichnet, und Entwickler der Modelle, die die Transformationen nutzen („Modellierer“ – in der Literatur [1] auch als “application designer” bezeichnet).

Die Entwicklung von Transformationen kann wiederum in drei spezielle Zeitpunkte eingeteilt werden, an denen diese im Entwicklungsprozess weiterentwickelt werden. Hierbei unterscheidet man Transformationen einer Bibliothek, Transformationen für eine bestimmte Nutzergruppe und Transformationen, die für die Durchführung eines individuellen Projekts entwickelt werden.

Transformationen einer Bibliothek werden als Werkzeug oder Teil einer Menge von Werkzeugen ausgeliefert, die für möglichst viele Applikationen oder Projekte Verwendung finden. Daher werden sie meist so ausgeliefert, dass keine Veränderungen notwendig sind. Sie werden fast ausschließlich von den Entwicklern des Werkzeugs entwickelt. Ein Beispiel sind die Benutzerschnittstellentransformationen, die mit der sog. Mapache-Programmierumgebung ausgeliefert und im nächsten Abschnitt vorgestellt werden. Der Nutzen der

Transformationen liegt in der wiederholten Anwendung der Transformationen, sodass ab einer gewissen Anzahl von durchgeführten Transformationen ein break-even der eingesetzten Entwicklungszeit gegenüber der ohne Transformationen erreicht wird.

Transformationen einer bestimmten Nutzergruppe dienen der Unterstützung zur Lösung einer bestimmten Aufgabe, die im Rahmen von Arbeiten einer speziellen Nutzergruppe wiederholt auftritt. Der Übergang zu Bibliotheken von Transformationen ist fließend und die beiden Fälle grenzen sich vor allem durch die Allgemeingültigkeit und die Anzahl der Nutzer gegeneinander ab. Im Gegensatz zu Transformationen aus einer Bibliothek werden sie durch den Nutzerkreis selbst auf spezielle Probleme angepasst. Ein Beispiel sind Transformationen, die von einer Beratungsfirma zur Lösung wiederkehrender Problemstellung der verschiedenen Kunden der Firma entwickelt und gewartet werden.

Transformationen, die zur Durchführung von Aufgaben in Projekten genutzt werden, dienen der Lösung eines speziellen Problems. Sie werden meist einmalig angewendet. Die Transformationen erledigen dabei innerhalb des Schrittes der Modellierung meist Aufgaben, die gut automatisierbar sind, aber zeitintensiv sind, wenn sie per Hand ausgeführt werden. Ein Beispiel ist die Veränderung vieler Modellelemente nach einem bestimmten Muster. Dadurch lohnt sich die Entwicklung von Transformationen, die diese Aufgaben übernehmen, bereits nach zeitlichen Gesichtspunkten. Diese Transformationen werden von den Nutzern selbst oder von Spezialisten entwickelt, die in ein Team zur Lösung spezieller Aufgaben eingebunden sind (z. B. innerhalb eines Projekts).

4.4.1.1 Phasen, Entwickler und Nutzergruppen

Die Entwicklung von Modellen von Applikationen mit Transformationen lässt sich ebenfalls in diese drei Fälle einteilen:

1. Im Falle der Bibliothek werden die Transformationen von den Modellierern nur angewendet und nicht verändert. Die Modellierer sind deshalb nur Nutzer. Die Anwendung der Transformationen zur Erzeugung von Modellen geschieht daher zeitlich versetzt und unabhängig von deren Entwicklung.

2. Im Fall, dass eine Nutzergruppe eine Aufgabe wiederholt bearbeitet, werden die Transformationen genutzt, die diese Nutzergruppe bereits vorher entwickelt hat und vorher ggf. entwickelt und modifiziert, s.o. Meist werden

Modifikationen notwendig, die von den Modellierern selbst oder einem Spezialisten möglichst zeitnah durchgeführt werden, um die Modellierer nicht zu behindern.

3. Im Fall dass die Durchführung einer ganz speziellen Aufgabe mit Transformationen unterstützt werden soll, geht die Entwicklung der Transformation und deren Nutzung zur Entwicklung des Modells der Applikation Hand in Hand. Die Modellierer identifizieren die Aufgaben innerhalb des Prozesses der Modellierung, die mit Transformationen automatisiert werden sollen und geben diese Aufgabe an Entwickler von Transformationen weiter oder entwickeln die Transformationen selbst.

4.4.2 Schritte

Es sind maximal drei wesentliche Änderungen an bestehenden Vorgehensmodellen notwendig, um die Entwicklung von (optimierenden) Constraint-relationalen-Transformationen aufzunehmen:

1. Die Anforderungen an die Constraints und eine eventuell vorhandene Zielfunktion müssen aufgenommen werden. Dabei ist festzulegen, ob es sich um eine Constraint-relationale-Transformation, eine optimierende Constraint-relationale-Transformation oder ein Derivatproblem handelt.

2. Danach findet die Entwicklung der Constraints statt. Findet sich eine Transformation in einer Bibliothek, oder existiert eine Nutzergruppe die bereits eine ähnliche Problemstellung mit Transformationen gelöst hat, kann versucht werden, existierende Transformationen zu verwenden und anzupassen.

3. Parallel zu, vor oder nach Punkt zwei ist es möglich, eine Zielfunktion zu definieren.

Diese Schritte können in klassische Vorgehensmodelle integriert werden. Nahezu in allen Vorgehensmodellen, die den gesamten Entwicklungszyklus von Software umfassen, sind mindestens folgende Schritte, Phasen oder Zeiträume vorgesehen (Beispiele: Wasserfallmodell, V-Modell, RUP, Spiralmodell, OpenUP, Kanban/Scrum, Feature Driven Development):

1. Anforderungsanalysephase

2. Systemdesignphase

3. Implementierungsphase

4. Testphase

Die Schritte lassen sich gemäß folgendem Algorithmus in solche Vorgehensmodelle integrieren:

1. füge Schritt 1 zu "Anforderungsanalyse" hinzu
2. füge Schritt 2 zu "Systemdesignphase" und "Implementierungsphase" hinzu, wenn
   (a) nach "Anforderungen" aus "Anforderungsanalyse" Transformationen und
   (b) nach "Anforderungen" aus "Anforderungsanalyse" Constraints zu entwickeln sind
3. füge Schritt 3 zu "Systemdesignphase" und "Implementierungsphase" hinzu, wenn
   (a) nach "Anforderungen" aus "Anforderungsanalyse" Transformationen und
   (b) nach "Anforderungen" aus "Anforderungsanalyse" Zielfunktionen zu entwickeln sind

4.4.3 Iterative Methodik und Beispiel

Auch wenn die Schritte in Form des Wasserfallmodells eingeführt werden, ist eine iterative Vorgehensweise heute Standard, da sich das Wasserfallmodell für Entwicklungsprozesse als zu statisch erwiesen hat (s. [135]). Die vorgestellten Schritte zur Erweiterung von Vorgehensmodellen lassen sich leicht in iterativen Vorgehensmodellen anwenden: Die Schritte 2 und 3 werden im Schritt der Entwicklung von Transformationen zusätzlich benötigt, während Schritt 1 in der Anforderungsanalyse benötigt wird. Als Beispiel werden die vorgeschlagenen Schritte in das (allgemeinere) Vorgehensmodell zur Entwicklung von Transformation von J. Küster u.a. [89] integriert und diese damit beispielhaft für Constraint-relationalen-Transformationen, optimierenden Constraint-Relationen-Transformationen und Derivatprobleme spezialisiert. Abbildung 4.8 zeigt ein auf deutsch übersetztes, leicht angepasstes und teilweise vereinfachtes Bild aus [89], das die Integration verdeutlichen soll. Es sind die von J. Küster u.a. eingeführten Phasen zur Entwicklung von Transformationen zu erkennen: Analyse, Design und Implementierung. In der Analyse werden die Anforderungen an die Transformationen aufgenommen. In der Phase des Designs werden Transformationsregeln schrittweise von einer groben grafischen Skizze (high-level Design) bis zu konkreten Transformationsregeln hin entwickelt. Anschließend werden die Regeln auf syntaktische und semantische Gültigkeit geprüft. In der letzten Phase werden die Regeln

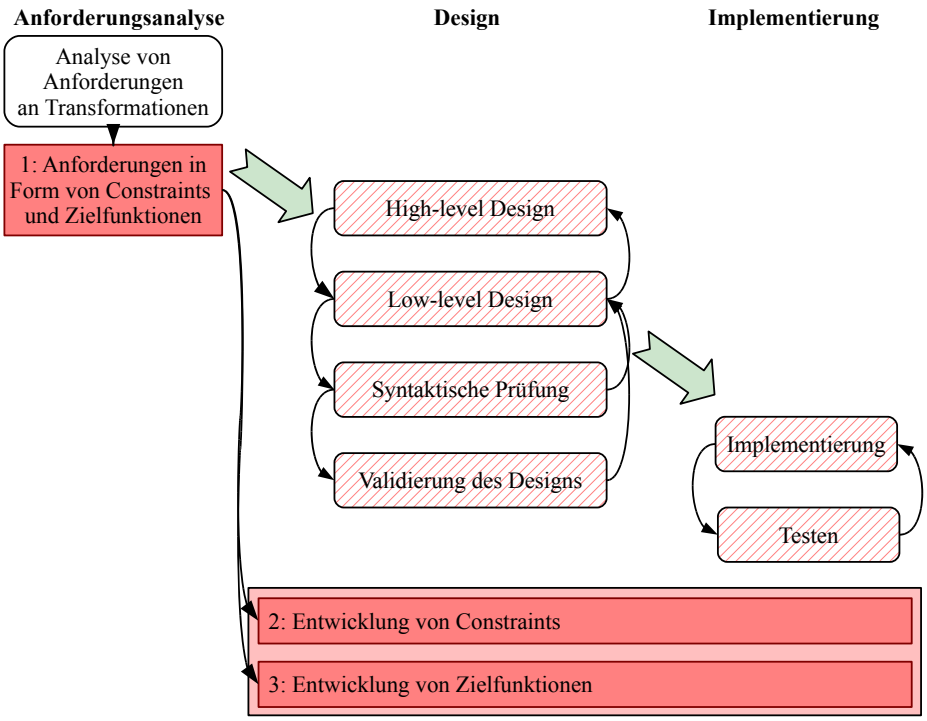


Abbildung 4.8: Methodik zur Entwicklung von Transformationen, [89] angepasst

implementiert und getestet. Mögliche Korrekturen oder Iterationen im Prozess sind durch die dünnen schwarzen Pfeile dargestellt. Die großen Pfeile symbolisieren die Transitionen zwischen den einzelnen Phasen.

Die entwickelten und neu hinzugefügten Schritte zur Entwicklung von Constraint-relationalen-Transformationen, optimierenden Constraint-Relationen-Transformationen und Derivatproblemen sind rot unterlegt. Die Analyse der Anforderungen an Transformationen kann um Schritt 1 ergänzt werden – die Analyse der Anforderungen an die Constraints und Zielfunktionen. Während des Designs und der Implementierung werden Schritte 2 und 3 in allen Schritten des Vorgehensmodells von J. Küster u.a. benötigt, da Constraints und Zielfunktionen als Erweiterungen von Transformationsregeln gesehen werden können.

Das vorgestellte Vorgehensmodell von J. Küster behandelt nur die Entwicklung von Transformationen. Entsprechend ist auch das Beispiel zur Integration der Schritte nur auf Transformationen beschränkt. Dies ist besonders für die Entwicklung von Transformationen für eine Bibliothek (s. Abschnitt 4.4.1) ausreichend, da

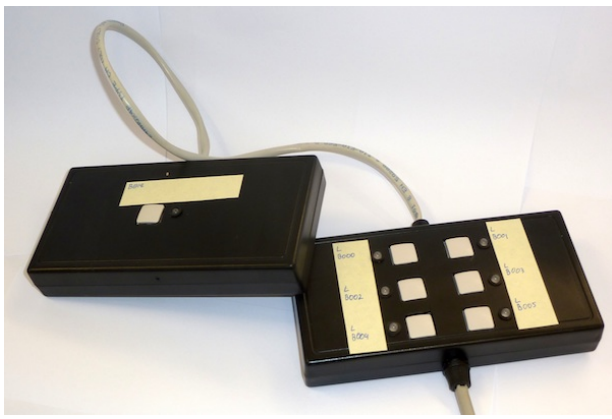


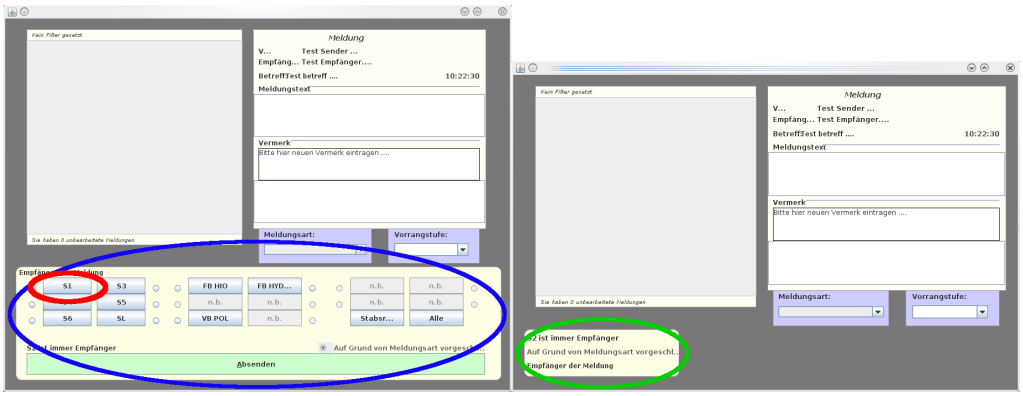
Abbildung 4.9: Spezialhardware zur Eingabe

diese meist getrennt von Anwendungen für die Transformationen entwickelt werden. In der Praxis werden Transformationen häufig erst im Rahmen von Projekten als notwendige Artefakte erkannt. Dies trifft besonders auf Transformationen zu, die von einer bestimmte Nutzergruppe benötigt werden und Transformationen, die für die Durchführung spezieller Aufgaben in Projekten verwendet werden. In diesen Fällen können die in den Projekten bereits verwendeten Vorgehensmodelle z.B. um das hier angepasste erweitert werden, um die Transformationen zu entwickeln.

4.5 Strategien zur Transformation von Modellen von Benutzerschnittstellen und deren Anwendung auf Beispiele

Der Nutzen von Constraints und Zielfunktionen für die Transformation von Modellen von Benutzerschnittstellen wurde bereits in den Kapiteln 1 und 3 erläutert. Die Constraints dienen der Deklaration von Beschränkungen, die in Zielmodellen halten müssen. Hingegen stellen Zielfunktionen entsprechende Kriterien zur Erzeugung von “optimalen” Zielmodellen dar. Beide Ansätze können nach Definition 3.18 gemeinsam in Form von Strategien verwendet werden. Zusammenfassend lässt sich eine Strategie nach dieser Definition als eine Kombination einer Vorgehensweise und eines Entwurfsmusters für Transformationen beschreiben.

Im Rahmen dieser Arbeit wurden verschiedene klassische und neue Strategien verfolgt, deren Theorien und Umsetzungen im Folgenden vorgestellt werden sollen. Die Strategien sollen einen Eindruck vermitteln, welche Flexibilität Zielfunktionen



(a) Sichterapplikation (b) Benutzerschnittstelle mit Hardware

Abbildung 4.10: Beispiel Hardware-Transformation

in Kombination mit Transformationen von Modellen von Benutzerschnittstellen bieten.

Die Strategien werden im Rahmen von Szenarien vorgestellt, an denen sie beispielhaft erläutert werden sollen. Um die Relevanz der Szenarien zu erhöhen, wurden einige im Rahmen eines industriellen Forschungsprojektes entwickelt und die hier vorgestellten Lösungen Projektpartnern aus der Wirtschaft präsentiert (s. [148]). Die Strategien einiger der hier vorgestellten Transformationen (Hardware-Transformation, Anoto-Transformation und Performance-Transformation) sind relativ simpel; es wird im Rahmen dieser Arbeit nicht der Anspruch erhoben, dass dies neue Strategien sind. Allerdings ist es das erste Mal, dass diese Strategien mit einer Modell-zu-Modelltransformationssprache realisiert wurden. Die Strategien mit Zielfunktionen wurden neu entwickelt. Die aus den Szenarien und Strategien entwickelten Transformationen werden im Mapache-System (s. [18]) ausgeliefert (Ausnahmen: Aui2Cui, Performance-Transformation).

4.5.1 Hardware-Transformation

Szenario

In den ersten drei Szenarien soll jeweils ein Modell einer bereits fertigen Benutzerschnittstelle modifiziert werden, das auf die speziellen Eigenschaften eines Desktops (d.h. Maus und Tastatur) optimiert wurde. Ein wichtiger Aspekt ist, dass die Transformationen nicht auf die gegebenen Ausgangsmodelle beschränkt sind, sondern für alle Ausgangsmodelle anwendbar sind, die dem Meta-Modell des Ausgangsmodell entsprechen. Im ersten Szenario soll eine Hardware eingeführt

werden, die über Knöpfe zur Eingabe verfügt. Dabei werden Komponenten in der Benutzerschnittstelle überflüssig und diese Komponenten zur Eingabe durch reale Knöpfe ersetzt. Dies ist eine Benutzerschnittstelle, die für Nachrichtensysteme von Katastrophenschutz-Krisenstäben entwickelt wurde. Ein sog. Sichter untersucht mit Hilfe der Software Nachrichten auf ihren Inhalt und leitet die Nachrichten an die entsprechenden Stabsmitglieder weiter. Wie alle Stabsmitglieder sind Sichter an klassische, reale Knöpfe gewöhnt und entsprechend versiert in deren Benutzung: die Einführung der Knöpfe soll daher die Akzeptanz erhöhen und die Fehlerrate senken.

Realisierung

Die entsprechende Hardware wurde im Rahmen dieser Arbeit entwickelt (s. Abbildung 4.9). Ein Mikrocontroller wurde mit einer Reihe von Knöpfen ausgerüstet. Zum besseren Verständnis findet sich in Abbildungen 4.10a und 4.10b ein Beispiel für eine Darstellung des Ausgangs- und Zielmodells als Benutzerschnittstellen. Die Modelle werden als Instanzen des Meta-Modells aus Abbildungen 3.2 und 3.3 implementiert. In den vorliegenden Fällen wurden die Meta-Modelle des Mapache-Systems in diese Modelle überführt. Die Transformation wird als Teil des sog. Mapache-Systems ausgeliefert [18]. Abbildung 4.11 zeigt das abstrakte Mapache-Modell. Dieses wird in das Modell aus Abbildung 4.12 transformiert. Die Modelle sind identisch mit den leichter verständlichen Bildern der Benutzerschnittstellen, sodass die Modelle ausschließlich bei dieser Transformation als Beispiel gezeigt werden.

Es werden alle “Action”-Modellelemente (ein Beispiel für eine Komponente für das Modellelement ist mit einer roten Ellipse markiert) des Ausgangsmodells in “Hardwarebuttons” des Zielmodells überführt. Dabei wird Platz auf der grafischen Benutzerschnittstelle frei, da die Buttons dort nicht mehr erscheinen (blaue Ellipse). Dies verbessert die Übersichtlichkeit der grafischen Benutzeroberfläche. Im Vorfeld ist nicht klar, welche von den restlichen Komponenten noch benötigt werden. Deshalb wird es dem Modellierer überlassen die restlichen Komponenten zu löschen oder neu zu platzieren. Die restlichen Komponenten des veränderten Teils der Benutzerschnittstelle werden daher übereinander an einer Linie ausgerichtet (grüne Ellipse).

Strategie 1: Komponenten eines Teils einer Benutzerschnittstelle untereinander oder nebeneinander anordnen

Bei dieser Strategie ist vorgesehen, die verbleibenden, vermutlich nicht mehr benutzten Elemente (“Component”) eines veränderten Teils der Benutzerschnittstelle durch Constraints untereinander anzuordnen. Der Modellierer kann nach der Transformation entscheiden ob er die Komponenten überhaupt noch braucht. Die Constraints in Solverational werden als Ausschnitt aus einer größeren Transformation in Listing 4.1 präsentiert. Es wird eine übergeordnete Komponente (“Com-

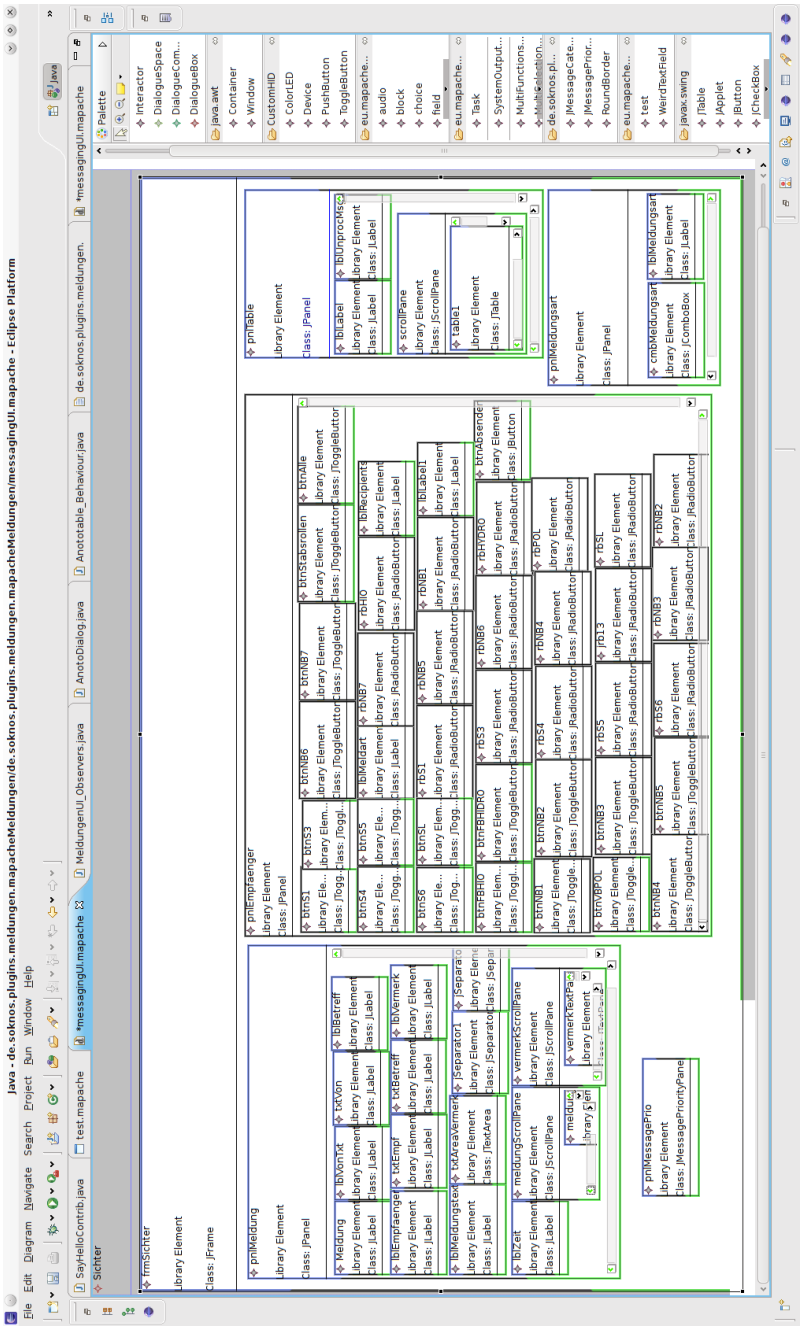


Abbildung 4.11: Abstraktes Modell, dargestellt in Mapache-Umgebung

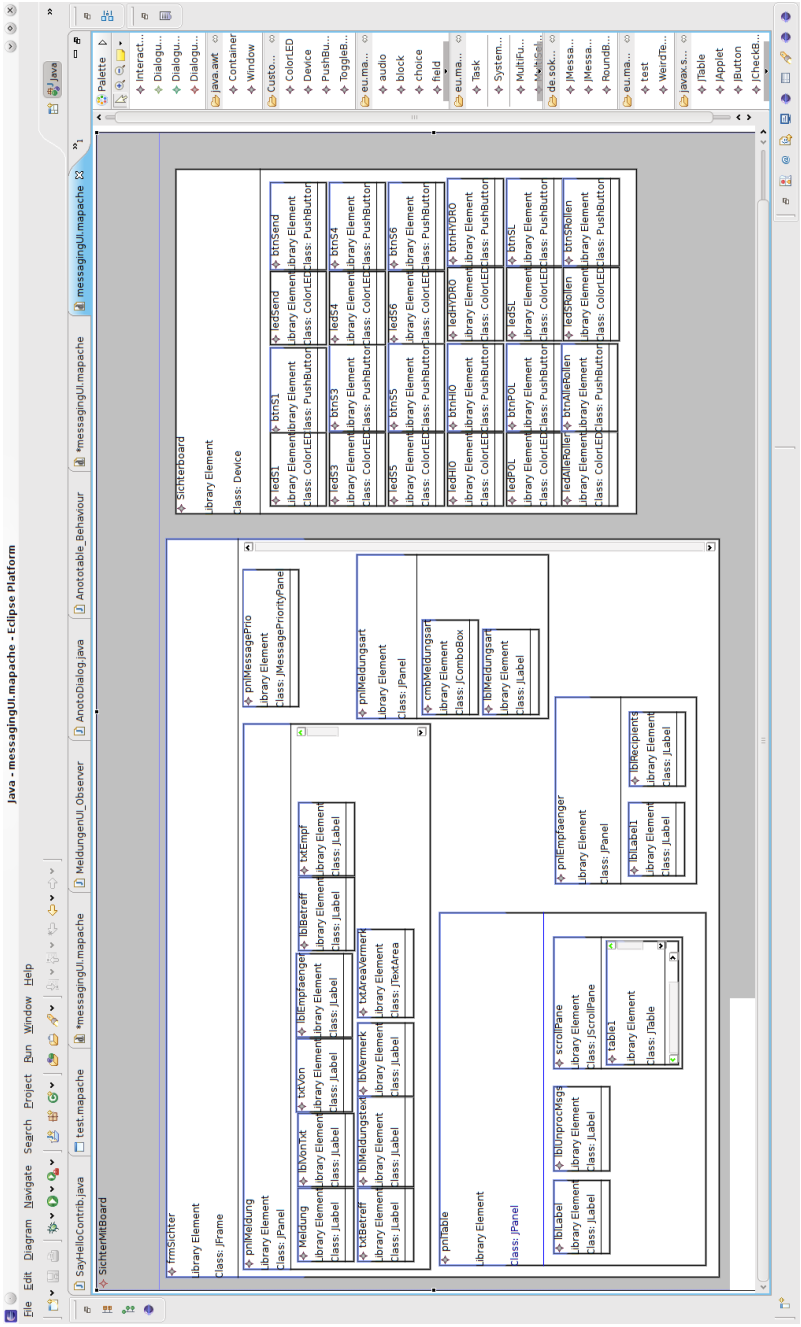


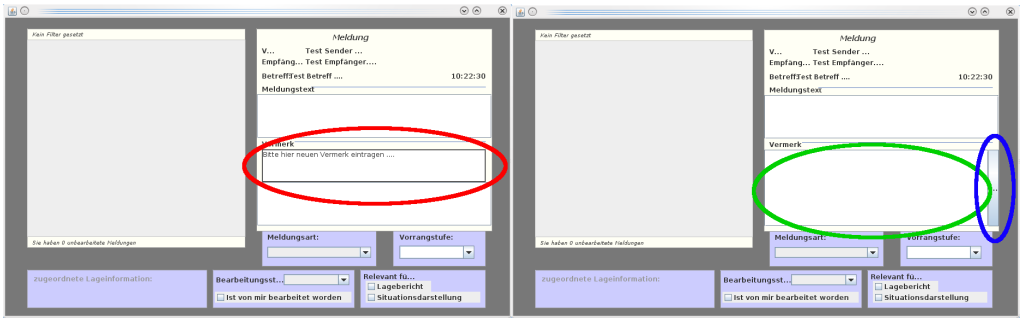
Abbildung 4.12: Konkretes Modell, dargestellt in Mapache-Umgebung

```
top relation setSizes2 {

    w: Integer;
    h: Integer;

    domain a i:abstractsoknosuimodel::Container {
        children=d:abstractsoknosuimodel::Interactor {
            previousSibling <> null,
            wDefault = w,
            hDefault = h
        }
    };

    domain c o:concretesoknosuimodel::Component {
        children=e:concretesoknosuimodel::Component {
            x = 5,
            y = previousSibling.y + previousSibling.height + 5,
            width = w,
            height = h
        }
    };
    ...
}
```

Listing 4.1: Solverational-Constraints um Komponenten untereinander anzuordnen

ponent") aus einem "Container" und eine beliebige Anzahl innerer Komponenten (Assoziation *children*) erzeugt. Die Position der inneren Komponenten wird in horizontaler Richtung auf fünf Pixel vom linken Rand des Containers festgelegt ($x = 5$). Dadurch beginnen die inneren Komponenten alle an einer imaginären vertikalen Linie. Die Position in vertikaler Richtung wird aus der Position des vorigen Elements ($previousSilbing.y$), dessen Höhe ($previousSibling.height$) und einem Abstand von 5 Pixeln berechnet. Dies bewirkt, dass die Komponenten untereinander angeordnet werden. Breite ($width$) und Höhe ($height$) werden von den Komponenten der ursprünglichen Benutzerschnittstelle übernommen. Die Breite und Höhe der übergeordneten Komponente wird jeweils aus Breite und Höhe der in ihr enthaltenen Komponenten berechnet. Durch einfaches Vertauschen von x und y sowie $height$ und $width$ können die Komponenten nebeneinander statt übereinander angeordnet werden.

Eine Zielfunktion wird mit dieser Strategie nicht benötigt, da nur Komponenten entfernt werden sollen und die restlichen Komponenten absichtlich nicht



(a) Eine Nachrichtenapplikation

(b) Benutzerschnittstelle mit Anotokompatibilität

Abbildung 4.13: Beispiel Anoto-Transformation

angeordnet werden sollen.

4.5.2 Anoto-Transformation

Szenario

Es soll wieder ein Modell einer bereits fertigen Benutzerschnittstelle (s. Abbildung 4.13a) modifiziert werden, die auf die speziellen Eigenschaften des Desktop optimiert wurde. Die Benutzerschnittstelle dient Stabsmitgliedern zum Verschicken von Nachrichten im Katastrophenfall und wird daher besonders selten eingesetzt. Daher ist es wichtig besonders intuitiv zu bedienende Benutzerschnittstellen bereitzustellen, da Stabsmitglieder die Verwendung zwischen Einsätzen verlernen. In diesem Szenario soll daher ein Anoto-Digitalstift zur Eingabe verwendet werden [153], der ähnlich wie Stift und Papier zu bedienen ist. Der Einfachheit halber werden Nachrichten als “electronic ink” ohne Handschrifterkennung eingegeben. Um dies zu realisieren wird im Zielmodell statt der in der ursprünglichen Benutzerschnittstelle vorhandenen Texteingabefelder ein Button eingefügt. Wird der Button betätigt kann mittels eines weiteren Fensters die Eingabe der “electronic ink” erfolgen (Beispiel s. Abbildung 4.13b).

Realisierung

Ein entsprechender Treiber für einen Stift wurde auf Basis des an der Fachgruppe Telekooperation entwickelten Letras-Systems realisiert [62].

Es werden alle Textinput-Modellelemente des Ausgangsmodells im Zielmodell in Buttons überführt (rote Ellipse, blaue Ellipse). Dabei wird Platz auf der ursprünglichen Benutzerschnittstelle frei, da die Textinput-Modellelemente dort

```
top relation filterInput {
    ...
    domain c o:concretesoknosuimodel::Component {
        x = parent.x + parent.width - 20,
        y = parent.y,
        width = 19,
        height = parent.height,
        label = '+'
    };
    ...
}

top relation calculateSizes {
    ...
    domain c o:concretesoknosuimodel::Component {
        y = parent.y,
        x = parent.x + 1,
        width = parent.width - 21,
        height =  parent.height
    };
    ...
}
```

Listing 4.2: Solverational-Constraints um Größe festzusetzen

nicht mehr vorhanden sind und Buttons in der Regel kleiner sind. Der Platz wird daher automatisch von umliegenden Komponenten verwendet (grüne Ellipse).

Strategie 2: Größenanpassung durch Constraints

In einem Schritt der Vorverarbeitung werden sowohl die Komponente, die durch den Button ersetzt werden soll, als auch die Komponente, die vergrößert werden soll, mit einem Attribut markiert. Es wird eine nicht sichtbare, übergeordnete Komponente erzeugt. In der Transformation wird dann durch Constraints die Größe und Position des Buttons und der zu vergrößernden Komponente angepasst (s. Listing 4.2).

4.5.3 Panel-Transformation

Szenario

Es sollen wieder Modelle bereits fertiger Benutzerschnittstellen modifiziert werden. Es handelt sich um das gleiche Beispiel wie im vorigen Abschnitt (s. Abbildung 4.13a). Die Benutzerschnittstelle wurde auf die speziellen Eigenschaften des

Desktop optimiert. In diesem Szenario soll das bestehende Modell an einen neuen Kontext angepasst werden. Kontext bezeichnet hier den Plattform-Kontext, insbesondere die Auflösung des Bildschirms. Dabei soll die Position von imaginären Komponenten, die andere Komponenten gruppieren, sogenannter Panels, an den Kontext angepasst werden. Für diese Aufgabe stehen eine ganze Reihe neu entwickelter Strategien zur Auswahl.

Realisierung

In der Transformation wird der Kontext – in diesem Fall die Bildschirmauflösung – anhand eines Modellelementes festgelegt. Die Anfangskoordinaten plus deren Breite und Höhe von allen Komponenten darf diese Werte nicht überschreiten, sodass entsprechende $\leq$-Ungleichungen in der Transformation implementiert wurden.

Zuerst werden die Panels in einem Vorverarbeitungsschritt (z.B. einer weiteren Transformation) jeweils paarweise in ein neues Panel eingefügt. Hierbei ist die Wahl "paarweise" (also jeweils zwei) willkürlich getroffen. Es kann jede beliebige Anzahl gewählt werden, jedoch bietet paarweise die größte Flexibilität, da es die kleinste Menge darstellt, die noch mehr als eine Komponente enthält. Die Transformation kann die Reihenfolge der paarweise gruppierten Panels verändern. Dadurch kann die Reihenfolge aller Panels nahezu beliebig geändert werden. Dies wird durch das Hinzufügen von zusätzlichen "Alternativen Domains" erreicht – die paarweise gruppierten Panels werden entweder aufsteigend oder absteigend sortiert.

Zusätzlich dazu kann die Transformation wählen, ob die gruppierten Panels horizontal oder vertikal angeordnet werden sollen.

Insgesamt ergeben sich also vier Alternativen pro neuem Panel, von denen ein Derivatproblem jeweils eine aussuchen muss:

1. aufsteigend sortiert, horizontal angeordnet
2. absteigend sortiert, horizontal angeordnet
3. aufsteigend sortiert, vertikal angeordnet
4. absteigend sortiert, vertikal angeordnet

Ein Auszug des Derivatproblem wird in Listing 4.3 dargestellt.

Mit diesen Constraints wird erreicht, dass die angeordneten Panels immer an imaginären Linien beginnen. Häufig werden klare Linien als ästhetisch empfunden, weshalb sich z.B. Strukturen wie Tabellen für Layouts bewährt haben. Mit dieser Transformation kann dies automatisch realisiert werden.

Durch Angabe des Kontextes (d.h. der Bildschirmauflösung) können allerdings nicht alle Möglichkeiten von der Transformation berücksichtigt werden. Panels sollen z.B. innerhalb des sichtbaren Bereichs liegen. Constraints stellen sicher,

```
top relation Container2ConcreteContainers {

    domain a i:abstractsoknosuimodel::Container {...};

    alternative domain c
            o:concretesoknosuimodel::RowLayoutPanel {
        width>=sum(children.width) +
                (5 * (count(children) - 1)) + 2 * 11,
        height>=max(children.height) + 2 * 11,
        children=e:concretesoknosuimodel::Component {
            y=parent.y + 11,
            x>=parent.x + 11,
            x<=parent.x + parent.width - 11,
            x>previousSibling.x + previousSibling.width + 4
        }
    };

    alternative domain c
            o:concretesoknosuimodel::VerticalLayoutPanel {
        height>=sum(children.height) +
                (5 * (count(children) - 1)) + 2 * 11,
        width>=max(children.width) + 2 * 11,
        children=e:concretesoknosuimodel::Component {
            x=parent.x + 11,
            y>=parent.y + 11,
            y<=parent.y + parent.height - 11,
            y>previousSibling.y + previousSibling.height + 4
        }
    };

    alternative domain c
        o:concretesoknosuimodel::RowLayoutReversed {...};

    alternative domain c
        o:concretesoknosuimodel::VerticalLayoutReversed {...};
    ...
}
```

Listing 4.3: Solverational-Derivatproblem um die Ausrichtung von Panels festzulegen

dass die x-Position und y-Position plus die Breite (*width*) und Höhe (*height*) von Komponenten kleiner als die angegebene Breite und Höhe der Bildschirmauflösung sind.

Die Bildschirmauflösung bietet trotzdem meist Platz für verschiedene Anordnungen und Positionen der Panels. Um Sortierung und Ausrichtung der Panels in der Transformation festzulegen, sollte daher eine Zielfunktion hinzugefügt werden. Im Rahmen der Transformation wurden verschiedene Zielfunktionen verwendet. In Kombination mit den obigen Constraints ergeben sich mehrere Strategien, von denen hier

- Möglichst-links- bzw. Möglichst-oben-Strategie,
- Relevanz-Strategie,
- 2D-Fitt's-Law-Strategie,
- Behring-Strategie,

vorgestellt werden sollen. Weitere Strategien können sehr leicht entwickelt werden, indem die Zielfunktion geändert wird. Diese macht in Solverational nur eine einzige Zeile Quellcode aus, wodurch der Aufwand zur Implementierung sehr gering ist.

Visuelle Beispiele für alle diese Strategien befinden sich auf Grund des benötigten Platzes im Anhang in Abschnitt A.1.

Strategie 3: Möglichst-links- bzw. Möglichst-oben-Strategie

Bei dieser Strategie sollen die Panels (oder Komponenten) möglichst links oder möglichst oben angeordnet werden.

Um dies mit einer Zielfunktion zu erreichen, können z.B. die x-Positionen der Panels (oder die allgemeine Form Komponente) addiert werden und die Zielfunktion minimiert werden:

```
minimize
 sum(concretesoknosuimodel::Component.allInstances()->x);
```

Als Ergebnis werden die Panels (oder Komponenten) so angeordnet, dass diese möglichst links platziert werden (da die Koordinatenachsen x und y nach rechts und unten wachsen). Wird x durch y getauscht, werden die Komponenten möglichst oben platziert.

Strategie 4: Relevanz-Strategie

Dem Entwickler wird bei dieser Strategie die Möglichkeit gegeben, Komponenten im Ausgangsmodell zu gewichten. Komponenten mit höherem Gewicht sollen von der Transformation möglichst nahe an der linken oberen Ecke platziert werden; dabei können beliebig viele verschiedene oder gleiche Gewichte verteilt werden, die dadurch potentiell unterschiedlich nahe an der linken oberen Ecke platziert werden.

Um dies mit einer Zielfunktion zu erreichen, können z.B. die x-Positionen der Panels (oder Komponenten) addiert und die Zielfunktion minimiert werden:

```
minimize
 sum(concretesoknosuimodel::Component.allInstances()->tmp2);
```

wobei tmp2 für jede Komponente wie folgt berechnet wird:

```
tmp2=relevance * x + relevance * y
```

Strategie 5: 2D-Fitt's-Law-Strategie

Bei dieser Strategie sollen die Panels (oder Komponenten) möglichst so angeordnet werden, dass die Zeit zur Navigation zwischen den Panels durch Fitt's-Law approximiert werden kann. Die Idee, die zur Navigation erforderliche Zeit durch Fitt's Law zu approximieren ist eine Idee aus der Metrik "Layout Appropriateness" von A. Sears [146].

Die zur Navigation erforderliche Zeit in der gesamten Nutzerschnittstelle wird durch deren Summe zwischen den einzelnen Panels approximiert. Innerhalb der Panels sei die Zeitspanne vernachlässigt um das Beispiel überschaubar zu halten.

Für jedes Panel muss nun zusätzlich das Attribut $navigationTime$ berechnet werden. Handelt es sich bei einer Komponente nicht um ein Panel, so ist die Zeit gleich Null. Andernfalls soll Fitt's Law angewendet werden. Dazu muss eine Variante von Fitt's Law gewählt werden, die auch für zwei Dimensionen gültig ist. Eine Auswahl an Varianten und entsprechende Bewertungen finden sich in [99]. Im vorliegenden Fall wurde die Variante "SMALLER-OF" approximiert; die ursprüngliche Form lautet:

$$Fitt's_{(A,W,H)} = a + b * \log_2(\frac{A}{\min(W,H)} + 1) \tag{4.1}$$

Da a und b Konstanten sind, die z.B. durch die Hardware vorgegeben sind (s. [143]), können sie für Zielfunktionen ignoriert werden. Dies liegt daran, dass Zielfunktionen minimiert oder maximiert werden und die beiden Konstanten nur eine Achsenverschiebung (a) oder eine Änderung der Steigung (b) bewirken. Gleiches gilt für den Logarithmus, der weggelassen werden kann: ein größerer Wert des Logarithmus eines Terms bedeutet auch einen größeren Wert des Terms, wodurch sich die Ergebnisse bzgl. Minimum oder Maximum nicht ändern.

W und H sind jeweils Breite und Höhe der mit dem Mauszeiger zu treffenden Komponente. Diese sind bereits wegen der Größe der Panels bekannt.

A bezeichnet die Entfernung vom Startpunkt des Mauszeigers zum Ziel. Die Entfernung kann leicht berechnet werden, da die Panels entweder horizontal oder vertikal angeordnet sind. Mit fünf Pixeln Abstand zum Rand und der Annahme, dass im Schnitt ein Klick in die Mitte einer Komponente erfolgt, gilt:

```
navigationTime=min((5+height/2+previous.height/2)/height,
                   (5+width/2+previous.width/2)/width),
```

Die 1 des inneren Terms, die in der Shannon-Formulierung von Fitt's Law verwendet wird (s. Gleichung 4.1), dient vor allem dazu, negative Werte des Logarithmus zu verhindern und kann daher ignoriert werden, da der Logarithmus nicht mehr benötigt wird.

Sollen nun alle *navigationTime*-Attribute in der finalen Zielfunktion berücksichtigt werden, so müssen diese multipliziert werden (Logarithmen-Regel: $\log(t_1)+\log(t_2) = \log(t_1 * t_2)$). Dabei handelt es sich nicht um eine lineare Zielfunktion. Mit Solverational und der vorgestellten Implementierung ist es möglich diese prinzipiell zu berechnen. Um allerdings Ergebnisse effizient zu berechnen ist es sinnvoll eine ungenaue, aber lineare, Approximation vorzunehmen. Dies ist für akzeptable Modelle nicht entscheidend: die Erzeugung von Benutzerschnittstellen kann naturgemäß sowieso nicht exakt durchgeführt werden und große Werte können nicht vorkommen (sodass die Addition die eigentliche Zielfunktion noch relativ gut approximieren kann). Als zusätzlicher Vorteil dieser Vorgehensweise kann später ein "Keystroke-Level-Model" (hier kurz KLM) nach der in Abschnitt 4.5.4 präsentierten Weise in die Zielfunktion integriert werden.

Entsprechend lautet die Zielfunktion:

```
minimize
 sum(concretesoknosuimodel::Component.allInstances()
   ->navigationTime);
```

Strategie 6: Behring-Strategie

Die Grundidee dieser Strategie geht auf einen Mitarbeiter zurück, der während des Verfassens dieser Arbeit am Fachgebiet Telekooperation tätig war, weshalb diese Strategie nach ihm benannt wurde. Ziel seiner Idee ist es, alle Panels möglichst oberhalb einer bestimmten, imaginären Linie platziert werden (d.h. möglichst oben links, mit bestimmten Gewichtungen). Umgangssprachlich formuliert soll die Zielfunktion der Transformation diese Linie so weit links oben wie möglich platzieren können. Dies konnte mit Hilfe der vorliegenden Transformationssprache umgesetzt werden.

Dazu werden die Panels gesucht, deren untere rechte Ecken am weitesten nach rechts und unten ragen. Dabei kann gewichtet werden, ob "am weitesten rechts" oder "am weitesten nach unten" wichtiger ist. Aus der Addition der beiden maximalen Koordinaten wird die Zielfunktion berechnet, die minimiert wird. Es werden dabei verschiedene Anordnungen von Panels berechnet, bis die Variante für die Anordnung der Panels gefunden wird, sodass die Zielfunktion minimal wird. Da es sich um eine lineare Zielfunktion handelt müssen nicht alle Varianten berechnet werden.

Dies kann wie folgt realisiert werden:

```
minimize
 max(concretesoknosuimodel::Component.allInstances()->rechts) +
 max(concretesoknosuimodel::Component.allInstances()->unten);
```

wobei für jedes Panel gilt:

```
rechts=1 * (width + x),
unten=20 * (height + y),
```

Eins und zwanzig entsprechen hierbei den Gewichtungen. Als Ergebnis werden in diesem Fall die Komponenten möglichst hinter einer Linie mit der Steigung zwanzig platziert, sodass sie möglichst weit links oben liegen.

4.5.4 AUI2CUI-Transformation

Szenario

A. Sears entwickelte eine Metrik zur Optimierung von Benutzerschnittstellen [146]. Diese Metrik konnte bislang nicht durch Modell-zu-Modelltransformationen implementiert werden, da eine ausreichend ausdrucksstarke deklarative Modell-zu-Modelltransformationssprache nicht existierte. Eine Realisierung kann mit Solve-rational erfolgen.

Realisierung

Hierzu wurde bereits in Abschnitt 4.5.3 die Optimierung mit Fitt's Law vorgestellt. Anstatt Optimierung auf ganze Panels anzuwenden, wird sie in diesem Abschnitt auf Komponenten angewendet. Eine Laufzeit-effiziente Berechnung erfordert spezielle Algorithmen. Dadurch dass die Transformation zur Entwicklungszeit ausgeführt wird läßt sich die Verwendung allgemeiner Algorithmen rechtfertigen. Im Rahmen dieser Arbeit wurde daher kein spezieller Algorithmus entwickelt.

Strategie 7: KLM-Fitt's Law-Strategie

Komponenten werden mit dieser Strategie entsprechend Abschnitt 4.5.3 nebeneinander oder untereinander angeordnet.

Ebenso wird die Zielfunktion für Fitt's Law übernommen und auf alle Komponenten gleichermaßen angewendet. Die Berechnung für $navigationTime$ erfolgt nun für alle Komponenten wie für die Panels aus Abschnitt 4.5.3.

Zusätzlich zur Navigation benötigten Zeit (diesmal zwischen allen Komponenten) soll die Zeit berücksichtigt werden, die zum Bedienen der Komponenten benötigt wird. Dazu wird das Keystroke Level Model (kurz "KLM") verwendet. Eine Einführung in die Methodik zur Verbesserung von Benutzerschnittstellen mit dem KLM findet sich in [80]. Dabei werden verschiedenen Benutzereingaben verschiedene Zeiten zugeordnet, die Tabellen aus [30] entnommen werden können.

Zur Approximation dieser Zeiten wird jeder Komponente eine *inputTime* zugeordnet, die einen vermeintlich durchschnittlichen Wert für Eingaben mit diesen Komponenten repräsentieren soll.

Für die Implementierung eines Modellelementes des abstrakten Modells stehen verschiedene alternative Modellelemente im Zielmodell (konkretes Modell) zur Verfügung. So kann eine Liste (Modellelement des abstrakten Modells) beispielsweise entweder als Listbox oder als Dropdown-Liste implementiert werden. Während bei der Listbox mehr Einträge zu sehen sind und der Nutzer somit eine Eingabe potenziell schneller durchführen kann, braucht sie auch deutlich mehr Platz als eine Dropdown-Liste. Daher ist die Implementierung der Liste nicht immer gleich: steht wenig Platz zur Verfügung, oder ist die Navigation mit der Maus relativ teuer, so kann eine Dropdown-Liste zu schnelleren Eingaben führen als die Listbox. Dazu werden in der Transformation verschiedene alternative Domains eingerichtet, die die verschiedenen alternativen Implementierungen, deren *inputTime* und Mindestgrößen als Constraints beinhalten. Ein Teil einer Transformation, der das vorgestellte Beispiel die Liste in eine Dropdownliste oder eine Listbox zu transformieren implementiert, befindet sich in Listing 4.4.

Das Abwägen zwischen den verschiedenen alternativen Domains, den Constraints, dem KLM und dem Fitt's Law wird dem Solver überlassen.

Die Zielfunktion dieser Strategie wird einfach um die Addition der *inputTime* erweitert - ein weiterer Grund, die Zielfunktion mit Fitt's Law aus Abschnitt 4.5.3 durch eine Addition linear zu approximieren.

```
minimize
 sum(concretesoknosuimodel::Component.allInstances()
    ->inputTime) +
 sum(concretesoknosuimodel::Component.allInstances()
    ->navigationTime);
```

4.5.5 Performance-Transformation

Szenario

Diese Transformation wird genutzt, um die Performanz des Constraint-Solving zu testen. Dabei wird die Struktur des Ursprungsmodells transformiert und ein Zielmodell erzeugt, das die gleiche Struktur besitzt. Es handelt sich in gewissem Sinne nicht um eine Strategie, sondern um drei Transformationen, die zum Vergleichen von Transformationsmaschinen zur Transformation von Modellen von Benutzerschnittstellen dienen sollen (s. Abschnitt 5.6 bzgl. erreichter Ergebnisse).

Realisierung

Ein Modell soll mit unterschiedlicher Anzahl von Modellelementen transformiert werden. Transformiert werden 3, 100, 500, 1000, 1500, 2000 und 2500 Mo-

```
...
top relation List2Lists {

    domain a i:abstractsoknosuimodel::List { };

    alternative domain c o:concretesoknosuimodel::DropDownList {
        width>=150,
        height>=40,
        navigationTime=2*100
    };
    ...
    alternative domain c o:concretesoknosuimodel::Listbox {
        width>=130,
        height>=125,
        navigationTime=1*100
    };

    when {
        AUIInteractor2CUIInteractor(i, o);
    }
}
...
```

Listing 4.4: Solverational-Derivatproblem für die Transformation von Listen

dellelemente. Alle sind wiederum mit einem ausgezeichneten Element assoziiert. Es soll möglichst schnell ein neues Modell erzeugt werden, das die gleiche Struktur besitzt. Im Bezug auf Modelle von Benutzerschnittstellen kann dies als eine Transformation verstanden werden, die einen Container mit unterschiedlicher Anzahl von darin enthaltenen Komponenten (Klasse "Action") von Benutzerschnittstellen transformiert. Die Modellelemente – also der Container und die darin enthaltenen Komponenten von Benutzerschnittstellen – haben jeweils eine Breite und eine Höhe, die in einer ersten Transformation kopiert werden sollen. In der ersten Transformation werden keine Constraints verwendet, die die minimale Breite und Höhe der Komponenten festlegen. Dadurch kann die Transformation auch mit Interpretern für klassisches QVT Relations ausgeführt werden. Im konkreten Modell wurden der Einfachheit halber die Komponenten "Button" (erzeugt aus Action) und "Window" erzeugt (erzeugt aus Container). Die Transformation wurde in Listing 4.5 abgedruckt.

In der zweiten Transformation werden ebenfalls die Modellelemente erzeugt, die minimale Breite und Höhe durch Constraints festgelegt sowie die Breite und

```
transformation trafo
        (a:SemiAbstractUIMetaModel, b:ConcreteUIMetaModel) {

    top relation SACopm2CComp  {
        theID:String;
        theName:String;
        theChID:String;
        theChName:String;
        domain a c:SemiAbstractUIMetaModel::
                    ContainerComponent {
            id=theID,
            name=theName,
            children=f:SemiAbstractUIMetaModel::
                     ActionComponent {
                id=theChID,
                name=theChName
            }
        };
        domain b d:ConcreteUIMetaModel::Window {
            id=theID,
            name=theName,
            children=g:ConcreteUIMetaModel::Button {
                id=theChID,
                name=theChName
            }
        };
    }
}
```

Listing 4.5: Transformation für Benchmarks - Variante 1

Höhe des assoziierten Modellelementes (des "Window") berechnet, indem die Breite aller Elemente addiert und die Höhe als Maximum der Höhe aller Komponenten gebildet wird. Es handelt sich also um einen horizontal ausgerichteten Container. Die Transformation wurde in Listing 4.6 abgedruckt.

Die dritte Transformation erfolgt identisch der zweiten, nur dass Größe und Breite des assoziierten Elements *mindestens* der Summe (bzw. des Maximums) der Werte der darin enthaltenen Komponenten entsprechen müssen. Dadurch kann ein Constraint-Solver potentiell einfacher einen gültigen Wert finden, indem es Größe und Breite erheblich größer wählt als mindestens gefordert. Die Transformation wurde in Listing 4.7 abgedruckt.

Eine Zielfunktion wird in dieser Strategie nicht benötigt. Für die erste kann

```
transformation trafo
        (a:SemiAbstractUIMetaModel, b:ConcreteUIMetaModel) {

    top relation SACopm2CComp  {
        theID:String;
        theName:String;
        theChID:String;
        theChName:String;
        domain a c:SemiAbstractUIMetaModel::
                    ContainerComponent {
            id=theID,
            name=theName,
            children=f:SemiAbstractUIMetaModel::
                    ActionComponent {
            id=theChID,
            name=theChName
            }
        };
        domain b d:ConcreteUIMetaModel::Window {
            id=theID,
            name=theName,
            w = sum(children.w) + 2*(4 - 1) + 6,
            h = max(children.h) + 13,
            children=g:ConcreteUIMetaModel::Button {
                id=theChID,
                name=theChName,
                w <= 60,
                w >= 30,
                h <= 20,
                h >= 10
            }
        };
    }
}
```

Listing 4.6: Transformation für Benchmarks - Variante 2

eine Transformationssprache ohne Constraint-Solving verwendet werden, während insbesondere die letzte Constraint-Solving erfordert.

```
transformation trafo
        (a:SemiAbstractUIMetaModel, b:ConcreteUIMetaModel) {

    top relation SACopm2CComp  {
        theID:String;
        theName:String;
        theChID:String;
        theChName:String;
        domain a c:SemiAbstractUIMetaModel::
                    ContainerComponent {
            id=theID,
            name=theName,
            children=f:SemiAbstractUIMetaModel::
                    ActionComponent {
            id=theChID,
            name=theChName
            }
        };
        domain b d:ConcreteUIMetaModel::Window {
            id=theID,
            name=theName,
            w >= sum(children.w) + 2*(4 - 1) + 6,
            h >= max(children.h) + 13,
            children=g:ConcreteUIMetaModel::Button {
                id=theChID,
                name=theChName,
                w <= 60,
                w >= 30,
                h <= 20,
                h >= 10
            }
        };
    }
}
```

Listing 4.7: Transformation für Benchmarks - Variante 3

4.6 Zusammenfassung

In diesem Kapitel wurde die Transformationssprache Solverational vorgestellt. Sie basiert auf der Modell-zu-Modelltransformationssprache QVT Relations [117] und erweitert QVT Relations hin zu einer optimierenden Constraint-relationalen-

Transformationssprache. Entsprechend den Syntaxen von QVT Relations, existieren auch für Solverational eine abstrakte, eine grafische und eine textliche Syntax. Es wurde eine Architektur für einen Interpretr und eine Implementierung von Solverational präsentiert. Um Transformationen, die in der Sprache Solverational entwickelt werden sollen, mit bestehenden Vorgehensmodellen entwickeln zu können, wurden drei Schritte entwickelt, die in Methodiken integriert werden können.

Zusätzlich wurden anhand von Beispielen sieben Strategien zur Erzeugung von Modellen von Benutzerschnittstellen präsentiert. Diese Strategien stellen eine Kombination von Constraints und Zielfunktionen dar und können in gewisser Hinsicht als Muster für die Erstellung von neuen Transformationen verwendet werden. Dabei wurden sowohl neue Zielfunktionen definiert als auch lange bekannte und etablierte Gesetze aus dem Gebiet der Mensch-Maschine-Interaktion, wie Fitt's Law, verwendet.

Das folgende Kapitel soll die vorgestellte Sprache Solverational auf ihre Tauglichkeit zur Verwendung für die Transformation von Modellen von Benutzerschnittstellen evaluieren.

Kapitel 5

Evaluation

Überblick

Die vorliegende Arbeit erhebt den Anspruch, eine allgemeine Transformationssprache zu definieren, die sich "gut" zur Transformation und Generierung von Modellen von Benutzerschnittstellen eignet. Um den Grad der Erfüllung dieser Ziele zu Evaluieren, wird im vorliegenden Kapitel eine Vorgehensweise angewendet, die die Arbeiten von J. Howatts einerseits und D. Olsen andererseits kombiniert. Hierzu wird in Abschnitt 5.1 eine Methodik zur Evaluation von Transformationssprachen entwickelt. Die Methodik ermöglicht die Aufstellung einer Formel zur "objektiven" Wahl von Transformationssprachen. Dieses sog. "objektiven Entscheidungskriterium" ist mit einer Metrik zur Wahl von Transformationssprachen vergleichbar (s. Abschnitt 5.8.4).

In den folgenden Abschnitten werden verschiedene Untersuchungen vorgestellt, die zur Aufstellung der Formel notwendig waren (s. Abbildung 5.1:

- Abschnitt 5.2 stellt eine Studie zur Verständlichkeit der untersuchten Sprachen vor.
- Die Quelltexte aus den Fragen der Studie wurden in Abschnitt 5.3 genutzt, um die Schreibbarkeit der Sprachen zu untersuchen.
- In Abschnitt 5.4 wird anhand einiger Beispiele geprüft, welche verschiedenen Typen von Transformationen in den Sprachen implementiert werden können.
- In Abschnitt 5.5 findet sich die Implementierung einiger "räumlicher Relationen" in der in dieser Arbeit entwickelten Transformationssprache um zu zeigen, dass diese Relationen auch in dieser Sprache implementiert werden können.

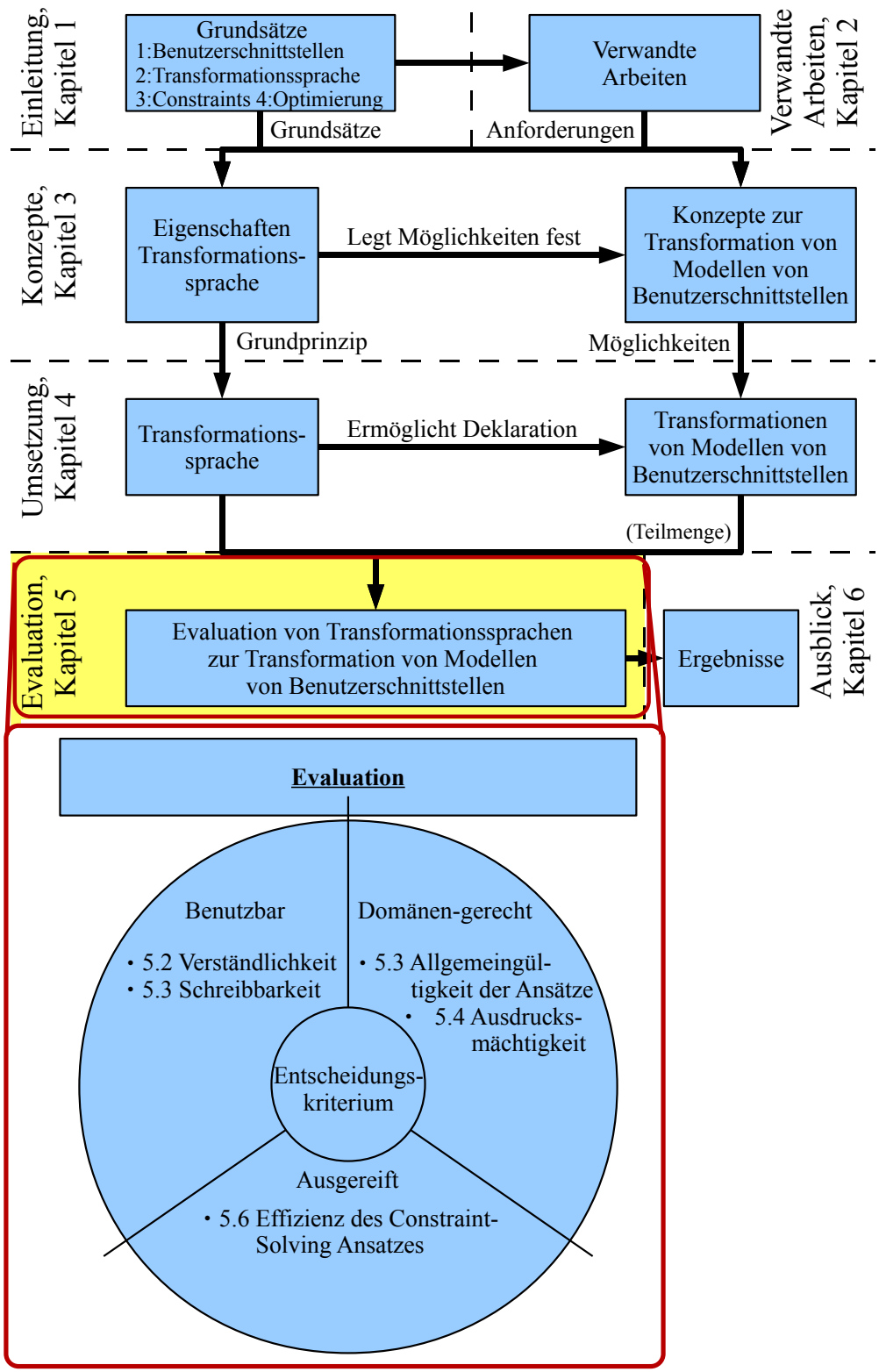


Abbildung 5.1: Aufbau Dissertation, Kapitel 5 hervorgehoben

- Die Effizienz des Constraint-Solving Ansatzes wird in Abschnitt 5.6 diskutiert.
- In der letzten Untersuchung (s. Abschnitt 5.7) wird geprüft, wie viele Regeln eines Style Guides sich in dieser Sprache implementieren lassen.

Abschnitt 5.9 fasst die Ergebnisse kurz zusammen.

Kandidaten zur Evaluation

Im Prinzip kommen für die Evaluation verschiedene Systeme in Frage (z.B. [47, 23, 44, 88, 13, 132]). Sinnvollerweise kann sich diese aber auf die offensichtlich "stärksten Konkurrenten" beschränken. Dabei handelt es sich um

1. Spatial Graph Grammars als Transformationssprache
2. und SUPPLE als ein System, das auch Optimierung unterstützt,

Diesen beiden hier untersuchten Systemen ist gemein, dass sie beide relativ neu sind. Das System SUPPLE wurde z.B. von K. Gajos erst 2008 zur Dissertation eingereicht [47]. Spatial Graph Grammars (SGGs) wurden 2006 vorgestellt [88]. Andere Systeme sind teilweise deutlich älter und bieten daher nicht den in den aktuellen Entwicklungen gegebenen Funktionsumfang (z.B. bietet [23] nur kontextfreie Grammatiken und das System GADGET [44] ist noch auf eine gegebene gültige Lösung angewiesen).

SGGs sind Graph-Grammatiken, die zur Transformation von Modellen von Benutzerschnittstellen verwendet werden können. Sie können auf eine Menge von festen Constraints zurückgreifen, die sog. "räumlichen Relationen". Diese bilden Relationen zwischen den mit einer Graph-Grammatik erzeugten Objekten ab. Eine Implementierung liegt bei der Erstellung dieser Arbeit nicht öffentlich vor, wird aber für die Evaluation auch nicht benötigt, da nur die Sprachen verglichen werden sollen und nicht die Implementierungen. Wie alle anderen Transformationssprachen (außer Solverational) können SGGs keine mathematische Optimierung nutzen.

SUPPLE ist ein Framework, das auf der Programmiersprache JAVA aufsetzt und aus abstrakten Beschreibungen Benutzerschnittstellen generiert. SUPPLE kann keine echten Modell-zu-Modell-Transformationen ausführen, da die Benutzerschnittstellen direkt nach deren Erzeugung angezeigt werden. Dafür können Optimierung und Constraint-Solving durchgeführt werden. Auch wenn SUPPLE keine Sprache im engeren Sinne ist, so sind Frameworks wie SUPPLE in gewisser Hinsicht eine "Erweiterung" der bestehenden Grundsprache und können daher im weitesten Sinne auch als Sprachen bezeichnet werden. "Sprache" bezeichnet daher in den folgenden Abschnitten - zusätzlich zu Solverational und SGGs - auch SUPPLE.

5.1 Methodik zur Evaluation

Die Ansätze werden anhand intuitiv verständlicher Kriterien untersucht. Diese sind Spezialisierungen von [70] und [112]. Andere Ansätze wie [5, 66, 109] sind auf allgemeine Programmiersprachen zugeschnitten und können für Modell-zu-Modell-Transformationssprachen (mit ihren Eigenheiten wie Typen aus Meta-Modellen, Deklarativität, mehrere Sprachen, etc.) nicht gut verwendet werden.

5.1.1 J. Howatts

J. Howatts identifiziert in [70] Kriterien, die allgemein zur Evaluation domänenspezifischer Programmiersprachen verwendet werden können. Er wählt "Language Design and Implementation Criteria" (syntaktisches Kriterium), "Human Factors Criteria" (Benutzbarkeit der Sprache), "Software Engineering Criteria" (methodische Implikationen der Sprache) und "Application Domain Criteria" (domänenspezifische Aspekte) als grobe Kriterien zur Evaluation von Sprachen. Diese mussten im Rahmen dieser Arbeit an die Gegebenheiten von modernen Modell-zu-Modell-Transformationssprachen zur Transformation von Benutzerschnittstellen angepasst werden: Die vorliegenden Sprachen sollen nicht den gesamten Teil von Aufgaben großer Softwareprojekte bewältigen können, sondern lediglich Modell-zu-Modelltransformation von Benutzerschnittstellen.

5.1.2 D. Olsen

Während sich die Analyse von J. Howatts vor allem mit Kriterien für Sprachen des Software Engineering befasst, beschäftigt sich D. Olsen in [112] mit Kriterien zur Evaluation von Systemen zur Entwicklung von Benutzerschnittstellen. D. Olsen beschreibt, dass die Evaluation von Systemen zur Entwicklung von Benutzerschnittstellen nicht mit Analysen der Benutzbarkeit vom System produzierter Benutzerschnittstellen durchgeführt werden kann: "The Usability Trap". Stattdessen schlägt er vor, abhängig vom Ziel des Systems mehrere Kriterien zu evaluieren: "Situations, Tasks, Users" / "Importance" (Größe der gewählten Endnutzergruppe, Aufgaben und Situationen), "Problem not previously solved" (Lösung für ein ungelöstes Problem), "Generality" (Lösung löst viele verschiedene Probleme), "Reduce solution viscosity" (Lösung hat eine Sprache, die den zu lösenden Problemen gut angepasst ist), "Empowering new design participants" (es können neue Akteure für das Design eingespannt werden), "Power in combination" (durch Kombination diverser Grundbausteine kann hohe Abdeckung erreicht werden), "Can it scale up?" (kann es auf große Probleme angewendet werden?). Die vorliegende Arbeit konzentriert sich auf den wichtigsten Teil der Kriterien, also die für die vorliegende Modell-zu-Modell-Transformationssprache und die Konkurrenten relevant sind.

5.1.3 Kombination

Wie Tabelle 5.1 anschaulich darstellt (Anzahl der Kriterien in jeder Zelle der Tabelle), konzentrieren sich die Kriterien von J. Howatt mehr auf technische Aspekte von Sprache und die von D. Olsen vor allem auf die Benutzbarkeit und die speziellen Aspekte von Domänen. Die Kriterien müssen einheitlich in einem System zusammengefasst werden, damit eine komplette Evaluation einer Sprache durchgeführt werden kann, die alle Kriterien berücksichtigt. Die Kriterien können grob - wie hier vorgestellt – in drei "Meta-Kriterien" eingeordnet werden (s. Tabelle 5.1). Die Benennung der "Meta-Kriterien" erfolgte willkürlich aber entsprechend dem Ziel der Evaluation des jeweiligen Kriteriums.

Als Ergebnis der Synthese der Kriterien zum Zwecke der Evaluation von Modell-zu-Modell-Transformationssprachen zur Transformation von Benutzerschnittstellen müssen diese daher

- ausgereift (s. Abschnitt 5.1.4),
- benutzbar (s. Abschnitt 5.1.5) und
- domänen-gerecht (s. Abschnitt 5.8.3)

sein. In den entsprechenden Abschnitten findet sich eine eingehende Betrachtung dieser Kriterien.

Nach dem Goal-Question-Metric-Modell (s. [17]) läßt sich das Ziel des Evaluationkriteriums mit der in diesem Modell entwickelten, sog. Schablone[1], wie folgt charakterisieren:

Analysiere Modell-zu-Modell-Transformationssprachen

zum Zwecke des direkten Vergleichs

in Bezug auf die Kriterien "ausgereift", "benutzbar" und "domänen-gerecht"

vom Blickwinkel eines Entwicklers oder Projektleiters

im Kontext einer allgemein auf verschiedene Projekte anwendbarer Metrik

Die hier entwickelten Kriterien können als eine Spezialisierung der Kriterien beider Analysen verstanden werden:

- Auf der einen Seite werden die Kriterien zur Evaluation von allgemeinen Sprachen hin zur Evaluation von Modelltransformationssprachen zur Entwicklung von Benutzerschnittstellen spezialisiert.

[1] Anmerkung: Die Schablone dient vor allem dazu, verschiedene Metriken zu kategorisieren; Kategorien sind fett herausgestellt. Anwender können so leichter erkennen, ob die Metriken für die ihnen vorliegenden Fragestellungen einsetzbar sind.

vorliegende Arbeit	J. Howatt	D. Olsen
ausgereift	Language Design and Implementation Criteria, Software Engineering Criteria	Can it scale up?
benutzbar	Human Factors Criteria	Reduce solution viscosity, Empowering new design participants, Power in combination
domänen-gerecht	Application Domain Criteria	Situations, Tasks, Users / Importance; Generality; Problem not previously solved

Tabelle 5.1: Kategorien für Kriterien der Ansätze zur Evaluation

- Auf der anderen Seite werden Kriterien zur Evaluation von Systemen zur Entwicklung von Benutzerschnittstellen hin zu Kriterien für eine Modelltransformationssprache zur Entwicklung von Benutzerschnittstellen spezialisiert.

5.1.4 Ausgereift

"Ausgereift" ist ein Qualitätskriterium von Programmiersprachen, das die rein technischen und Software-Engineering-Aspekte von Programmiersprachen bezeichnet. Auf den ersten Blick scheinen diese Kriterien auf der Basis fester Kriterien aus der Literatur verhältnismässig einfach vergleichbar zu sein, aber bereits die Kriterien von N. Holtz und W. Rasdorf [66] können nicht gut auf Modell-zu-Modell-Transformationssprachen übertragen werden (da z.B. das Typ-System von den Meta-Modellen abhängig ist). Auch die Kriterien für die "neue Generation von Entwicklerwerkzeugen" von J. Vanthienen und S. Poelmans (s. [162]) können nicht verwendet werden, da es sich bei Solverational und SGGs um deklarative und nicht um objektorientierte Sprachen handelt. Daher wird hier nicht auf feste Kriterien aus der Literatur zurückgegriffen, sondern aus den Kriterien von J. Howatt und D. Olsen ausgewählt.

Das Kriterium "ausgereift" ist eine Kombination aus den Kriterien "Language Design and Implementation Criteria" und "Software Engineering Criteria" von J. Howatt. Nach J. Howatt soll eine Sprache implementierbar sein. Sie soll keine zweideutigen Sprachkonstrukte enthalten, es soll möglich sein, einen schnellen Compiler dafür zu schreiben, der kompakten, effizienten Code erzeugt. Zusätzlich sollte eine

Sprache portablen (über mehrere Plattformen hinweg verwendbarer Code), zuverlässigen, erweiterbaren, wartbaren und wiederverwendbaren Code fördern und "alle anderen Aspekte von Software Engineering umfassen". Es sollten für eine Sprache "gute" Compiler und viele "gute" Programmierer vorhanden sein.

Es wäre gewagt zu behaupten, es existiere eine Möglichkeit, Sprachen nach allen diesen Kriterien untersuchen zu können (es steht noch nicht einmal die Anzahl der Kriterien fest). Aus diesem Grund beschränkt sich diese Arbeit auf die Existenz eines Compilers (Stellvertretend für "implementierbar") und auf zweideutige Sprachkonstrukte; wohlwissend dass dies nicht für einen umfassenden Vergleich ausreicht. Alle anderen Aspekte sind im Rahmen dieser Arbeit entweder deshalb nur schwer vergleichbar, weil die Implementierung von SGGs nicht vorliegt (Interpreter), weil SGGs nur grafisch notiert werden, oder weil SUPPLE ein Framework und kein Compiler ist. Viele "gute" Programmierer existieren für keinen der drei Ansätze, da es sich ausschlisslich um Prototypen handelt.

5.1.5 Benutzbar

Ob eine Sprache benutzbar ist, liegt nach der in dieser Arbeit vertretenen Ansicht vor allem an deren Verständlichkeit ("lesen") und wie zügig neue Transformationen implementiert werden können ("schreiben"). Verständlichkeit bezeichnet den Prozess, eine bestehende Transformation oder Teil einer Transformation zu verstehen. Mit Schreibbarkeit werde die Effizienz bezeichnet, mit der Transformationen geschrieben werden können.

5.1.5.1 Verständlichkeit

Verstehen ist ein kognitiver Prozess. Die Verständlichkeit als solche ist daher als zeitliche Verzögerung beim Lesen eines Textes messbar. Um die Verständlichkeit zu messen, kann daher einer Anzahl Probanden ein Quelltext präsentiert werden, die dann versuchen müssen diesen schnellstmöglich zu verstehen. Um die Zeit maschinell messen zu können, müssen die Probanden in einer vorher festgelegten Art und Weise auf den gelesenen Text reagieren, die vom Inhalt abhängig ist. Dies kann eine Frage sein, die auf den Inhalt Bezug nimmt und zu der eine Menge möglicher Antworten zur Auswahl stehen. Wird die Frage schneller verstanden, so wird die Frage schneller beantwortet.

Die Auswahl der Fragen und Quelltexte ist durch die Forschungsfrage vorgegeben. Um die Verständlichkeit von Transformationssprachen im Rahmen der These zu testen, dass Contraint-Solving und Optimierung "gute" Sprachkonstrukte darstellen, sollten drei Bereiche untersucht werden: Modelle, Constraints und Optimierung.

5.1.5.2 Schreibbarkeit

Aufbauend auf den Fragen einer Studie zur Verständlichkeit kann untersucht werden, wie viel Code für die verschiedenen Quelltexte implementiert werden mussten. Dies stellt eine sehr grobe Approximation dar, die einen Anhaltspunkt über die Schreibbarkeit der Sprachen geben soll und nicht den Anspruch auf wissenschaftliche Perfektion erheben kann.

Um die "Menge" Code zu messen kann z.B. die Anzahl relevanter terminaler Symbole gezählt werden, die zum Notieren minimal benötigt wird.

5.1.6 Domänen-gerecht

Laut [70] ist es nicht möglich "Application Domain Criteria" zu definieren ohne sich auf eine Domäne festzulegen. Aus diesem Grund macht J. Howatts auch keine weiteren Angaben zu diesen Kriterien.

Da diese Arbeit sich auf Modelltransformation von Benutzerschnittstellen konzentriert, können diese Kriterien aber weiter spezifiziert werden. Es darf nicht außer acht gelassen werden, dass diese Kriterien auch evaluiert werden müssen, daher muss berücksichtigt werden, dass nur begrenzte Ressourcen zur Verfügung stehen, was eine umfassende Betrachtung unmöglich macht.

Es werden folgende Kriterien festgelegt:

1. Allgemeingültigkeit der Sprachen
2. Grad der Abdeckung von Regeln zur Benutzbarkeit

5.1.6.1 Allgemeingültigkeit der Sprachen

Als erstes soll untersucht werden, wie allgemeingültig die drei Sprachen sind. Die Untersuchung zeigt wie stark Sprachen auf eine Domäne beschränkt sind. Damit wird entsprechend [112] der Grad der Abdeckung von allen denkbaren Transformationen bezeichnet (s. Abschnitt 5.4). Ein bekanntes Kriterium dazu ist Turing-Vollständigkeit, die aber nur für allgemeine Programmiersprachen, in diesem Fall JAVA/SUPPLE gegeben ist. Für andere Sprachen kann anhand von verschiedenen Transformationen untersucht werden, wie Allgemeingültig Sprachen sind. D. Olsen schlägt vor Beispiele aus mindestens drei verschiedenen Domänen zu verwenden, damit die Aussage glaubwürdig erscheint.

5.1.6.2 Grad der Abdeckung von Regeln zur Benutzbarkeit

Ein Kriterium für Transformationssprachen zur Transformation von Modellen von Benutzerschnittstellen ist, wie "gut" Benutzbarkeit mit ihnen abbildbar ist. Hierzu

kann beispielsweise ein bestehender Style Guide als Vorlage verwendet werden. Die Apple Human Interface Guidelines (s. [10]) stellen durch ihre breite Verwendung einen guten Kandidaten dar. Experten können diese Guidlines ("Regeln") auf ihre Abbildbarkeit in Solverational untersuchen. Als Ergebnis entsteht ein Richtwert ("Grad") wie gut eine Transformationssprache die Regeln abbilden können.

5.1.7 Komplexes Entscheidungskriterium

Entwickler stehen oft vor dem Problem, eine geeignete Programmiersprache für ein Projekt auszuwählen. Dabei werden häufig sehr emotionale Debatten geführt, deren Emotionalität oft aus der persönlichen subjektiven Weltanschauung bestimmter Entwickler für oder gegen eine bestimmte Sprache herrührt (die Trivialliteratur hat hier durchaus amüsante Diskussionen zu bieten z.B. [36]). Solche subjektiven Kriterien können zwar vielleicht die Zufriedenheit der Entwickler in einem Projekt beeinflussen, stellen aber kein objektives Entscheidungskriterium dar. Bislang existiert auch kein objektives Entscheidungskriterium, welche der drei zu vergleichenden Modell-zu-Modell-Transformationssprachen in Projekten verwendet werden sollte, in denen eine solche Sprache zur Erzeugung von Modellen von Benutzerschnittstellen verwendet wird. Die vorliegende Arbeit versucht daher auf der Basis einfacher Kriterien ein komplexes Entscheidungskriterium für die drei Sprachen herzustellen. Das komplexe Entscheidungskriterium soll es Entwicklern erlauben, für verschiedene einfache Kriterien Gewichte anzugeben und so die für ihre Anwendung (die die Gewichte bestimmt) bestmögliche Sprache zu finden. Nach Aufstellung der einzelnen Kriterien für "ausgereift", "benutzbar" und "domänengerecht" kann das gesamte komplexe Entscheidungskriterium berechnet werden. Eine entsprechende Formel wird am Ende des Kapitels entwickelt.

5.2 Verständlichkeit

Verständlichkeit einer Sprache kann durch kognitive Modelle abgebildet werden. Diese kognitiven Modelle bieten eine Theorie oder eine Formel mit der die Verständlichkeit von Sprachen qualitativ oder quantitativ eingeschätzt werden kann – z.B. können Texte auf Klassenstufen abgebildet werden um z.B. zu entscheiden, ob ein Text für den Unterricht für eine gewisse Altersstufe von Jugendlichen geeignet ist [64]. Für kognitive Modelle für die Verständlichkeit von objektorientierten Programmiersprachen findet sich in K. El-Eman [40] eine Übersicht, deren Arbeiten sich zum Teil in den Arbeiten über die Verständlichkeit von M. Genero u.a. [51] zu Klassendiagrammen im Allgemeinen spezialisieren lassen (die die Basis von grafischen Modellen bilden), oder gar zur Verständlichkeit von OCL-Ausdrücken [133] (die die Basis von Ausdrücken in Solverational bilden). Insofern

existieren zumindest kognitive Modelle für Solverational-Ausdrücke. Damit aber die Unterschiede in der Verständlichkeit der drei zu vergleichenden Sprachen ausreichend durch Theorien oder Formeln differenziert werden können, müssten für alle drei Sprachen kognitive Modelle vorliegen, die die Verständlichkeit des kompletten Sprachumfangs abbilden können. Diese kognitiven Modelle müssten zusätzlich vergleichbar sein, da die Sprachen direkt verglichen werden sollen. Derzeit existieren für die drei zu vergleichenden Sprachen aber keine kognitiven Modelle, die den kompletten Sprachumfang der Sprachen abbilden oder gar einen direkten Vergleich zulassen würden.

Aus diesem Grund wurde eine vergleichende Studie durchgeführt, um die Verständlichkeit an Probanden zu messen. In dieser Studie wurden Probanden verschiedene Ausschnitte von Transformationen vorgelegt. Die Probanden mussten aus einer Auswahl von möglichen Antworten entscheiden, was der ihnen vorliegende Teil der Transformation bewirkt. Es wurde nicht gemessen wie schnell die gesamten Transformationen verstanden werden können[2].

Bei der Durchführung und Auswertung der Studie wurden die von D. Perry u.a. [122] vorgeschlagenen Schritte zur Durchführung von Studien in der empirischen Evaluation von Ansätzen des Software Engineering angewendet. Demnach soll eine Studie folgende Punkte umfassen, die die Struktur der folgenden Abschnitte vorgeben:

1. Forschungskontext (s. voriger Absatz)
2. Hypothese
3. Studiendesign / Durchführung
4. Bedeutendste Faktoren, die die Validität einschränken können
5. Datenanalyse und Präsentation
6. Ergebnisse und Interpretation

5.2.1 Hypothese

Es waren drei Bereiche der Sprachen zu testen, soweit die Sprachen Konstrukte zur Implementierung vorsahen:

[2]Dies hätte mehrtägige Schulungen der Teilnehmer erfordert, da die Teilnehmer in der Verwendung von deklarativen Sprachen unterrichtet werden müssen. Zusätzlich wären die Ergebnisse wesentlich schlechter zu vergleichen gewesen, da die Implementierungen kompletter Transformationen in den drei Sprachen noch größere Unterschiede aufweisen als nur Teile der Transformationen. Dadurch sind noch mehr Faktoren zu berücksichtigen (schlechtere interne Validität).

1. Bereich Modelle: Lassen sich grafisch notierte Modelle von Benutzerschnittstellen leichter verstehen oder die in SUPPLE textuell notierten? In diesem Fall wurden Solverational und SGG mit einer identischen (grafischen) Notation mit der Notation von SUPPLE verglichen.

2. Bereich Constraints: Welche Notation für die Constraints lässt sich am einfachsten verstehen? Constraints werden in allen drei Sprachen unterschiedlich notiert, sodass alle Sprachen untersucht wurden.

3. Bereich Optimierung: Welche Notation für Zielfunktionen lässt sich am einfachsten verstehen? In SGGs können Zielfunktionen nicht notiert werden, sodass nur Solverational und SUPPLE verglichen werden konnten.

Da die Studienteilnehmer die Programmiersprache JAVA kannten und sie diese Sprache damit vermutlich einfacher verstehen konnten, blieb zu hoffen, dass zwischen der JAVA/SUPPLE Repräsentation der Lösung und den Solverational- und SGG-Lösungen keine statistisch nachweisbar signifikanten Unterschiede bestanden.

5.2.2 Studiendesign

Es wurde eine quantitative Studie basierend auf einem speziell entwickelten Fragebogen durchgeführt. Auf Grund mangelnder verfügbarer Experten, die sowohl Expertise mit Modell-zu-Modelltransformationen als auch der Generierung von Benutzerschnittstellen vorweisen konnten, konnten die Teilnehmer nicht gebeten werden, eigene Transformationen zu entwickeln. Stattdessen wurden den Teilnehmern fertige Teile von Transformationen vorgelegt. Die Studie basierte auf einem speziell entwickelten Fragebogen. Die Frage "What does the following source code do?" wurde zu einem jeweils in sich abgeschlossenen Ausschnitt eines Quellcodes einer Transformation in einer bestimmten Sprache gestellt. Die Probanden mussten anhand einer Auswahl möglicher Antworten entscheiden, was dieser Teil der Transformation bewirkt. Das Design des Fragebogens richtete sich nach Vorschlägen aus der Serie [129, 82, 81, 85, 83, 84], die sich mit der Durchführung von Umfragen im Software Engineering beschäftigt.

Die Studienteilnehmer wurden zum größten Teil aus Mitarbeitern der Fachgruppe Telekooperation an der TU-Darmstadt ausgewählt (90%); alle Teilnehmer hatten bereits mehrere Programme in der Sprache JAVA geschrieben – Probanden, die bereits Erfahrung mit Solverational hatten, wurden von der Studie ausgeschlossen, um eine Verzerrung zu Gunsten von Solverational zu vermeiden. Den Studienteilnehmern wurde nicht mitgeteilt, welches das am Lehrstuhl entwickelte System war. Allen Studienteilnehmer wurde ein kleines Präsent offeriert, um einerseits potentielle Probanden zur Teilnahme zu bewegen (Annahmerate war 100%) und andererseits erhöhte dies die Rate derer die alle Fragen beantworteten (Rate

vollendeter Fragebögen war 100%). Obwohl es sich um eine Internet-basierte Studie handelte, wurden die Teilnehmer bei der Durchführung per Telefon geführt oder direkt beobachtet. Insgesamt nahmen 19 Probanden an der Studie teil. Den Teilnehmern wurde mitgeteilt, dass die Ergebnisse in einer Form präsentiert werden würden, die nicht auf die Identität der Teilnehmer schließen läßt.

Für die Durchführung wurden drei Gruppen von Studienteilnehmern gebildet, die jeweils die Aufgaben in einer Sprache (also entweder Solverational (6 Teilnehmer), SGGs (7 Teilnehmer) oder SUPPLE (6 Teilnehmer)) beantworteten. Dadurch konnten Lerneffekte weitestgehend ausgeschlossen werden, die durch wiederholtes Durchführen der Fragen in einer anderen Sprache entstanden worden wären.

Als Operationalisierung wurde die Zeit zum Verstehen gewählt. Dabei wird die intuitiv richtig erscheinende Annahme getroffen, dass schnelleres Verstehen prinzipiell "besser" ist. Es ist nicht klar, wann ein Proband Zeit brauchte um zu Lesen, zu Verstehen oder auf die Antwort zu klicken. Deshalb wurde die Zeit zum Verstehen nur indirekt gemessen, indem die Gesamtzeit pro Frage gemessen wurde. Als Werkzeug für die Umfrage wurde Limesurvey [144] verwendet und der Quellcode erweitert, sodass die Zeit gemessen werden konnte.

In allen Bereichen wurden jedem Proband jeweils drei Fragen gestellt (insgesamt also 9 für Solverational und SUPPLE und 6 für SGGs), zu der es jeweils vier Antworten gab:

1. Modelle: Den Studienteilnehmern wurden mehrere Modelle präsentiert, und sie sollten entscheiden, wie die Benutzerschnittstelle zu diesem Modell aussah. Die Sprachen in denen die Modelle notiert wurden, waren entweder grafisch (SGG und Solverational) oder JAVA im Fall von SUPPLE.

2. Constraints: In allen drei Sprachen wurden die gleichen Constraints implementiert, die den Probanden präsentiert wurden. Die Probanden sollten entscheiden, was genau die Constraints beschränken.

3. Optimierung: Es wurde untersucht, wie gut Optimierung von den Studienteilnehmern verstanden wurde. Dazu wurden den Studienteilnehmern unterschiedliche Zielfunktionen präsentiert. SGGs implementieren keine Unterstützung für Optimierung und wurden deshalb nicht betrachtet.

Alle Fragen und Antworten hatten – bis auf den jeweils angezeigten Quellcode – in jeder Sprache den gleichen Wortlaut. Auch das Modell, das Constraint oder die Zielfunktion hatten jeweils die gleiche Funktion, nur wurden diese in unterschiedlichen Sprachen dargestellt. Entsprechend waren die richtigen Antworten in allen drei Sprachen identisch. Dadurch konnte sichergestellt werden, dass zeitliche Unterschiede nur im Lesen und Verstehen des Quellcodes entstehen konnten.

Vergleich		T-Test (Welch)	Wilcoxon-Mann-Whitney	F-Test
grafische Modelle SUPPLE	-	$0,0006$	$5 * 10^{-7}$	$3 * 10^{-13}$

Tabelle 5.2: p-Werte der Tests: Notation von Modellen

Im Vorfeld wurden die Fragen nach aufsteigender Schwierigkeit sortiert. Diese Reihenfolge wurde bei jedem Studienteilnehmer identisch präsentiert. Schließlich wurde mit drei Teilnehmern eine Pilotstudie durchgeführt, die aber keine Änderungen im Studiendesign und der Reihenfolge erforderlich machte.

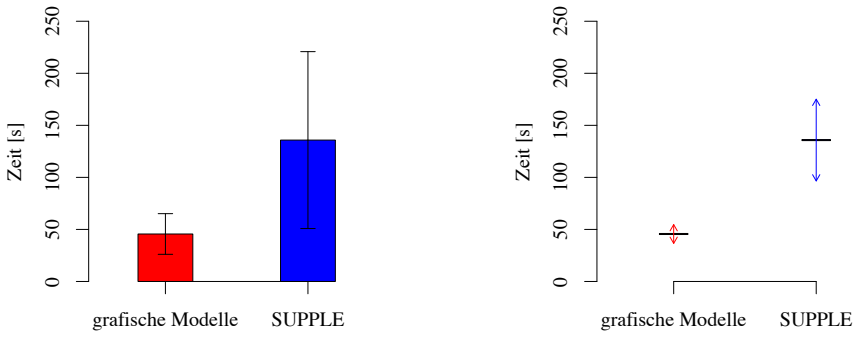
5.2.3 Datenanalyse und Präsentation

Zur Datenanalyse wurden alle Tests gegen ein Signifikanz-Niveau von $\alpha = 0,95$ durchgeführt. Im Anhang finden sich Abbildungen zu den berechneten stetigen Verteilungen der Populationen (Abbildungen B.1a bis B.3b), die teilweise nicht der Normalverteilung entsprechen. Deshalb wurden alle Tests mit Populationen, die nicht der Normalerteilung entsprachen, mit parameterlosen Tests wiederholt. In allen Fällen wurden für solche Fälle die klassischen Null- (Effekt nicht statistisch signifikant = Effekt nicht vorhanden) und Alternativhypothesen (Effekt statistisch signifikant = Effekt vorhanden) verwendet.

Im **Bereich Modelle** wurden die grafischen Modelle von Benutzerschnittstellen (die Modelle in Solverational und SGG repräsentierten) mit den textuell notierten von SUPPLE verglichen. Abbildung 5.2 präsentiert die berechneten Mittelwerte und Standardabweichungen der grafischen Modelle und den textuellen von SUPPLE (zusätzlich werden Konfidenzintervalle präsentiert). Der Mittelwert über den genommenen Zeiten für grafische Modelle war niedriger als der für SUPPLE.

Es blieb statistische Signifikanz zu beweisen. Im Fall der grafischen Modelle konnte die Hypothese, es handle sich um Normalverteilung, beibehalten werden, während für SUPPLE keine Normalverteilung vorlag (Shapiro-Wilk-Test p-Werte $= 0,23$ bzw. $0,002$). Aus diesem Grund wurden zwei Tests durchgeführt: der klassische T-Test als Welch-Test (klassischer T-Test erfordert Homoskedastizidität, es lag aber Heteroskedastizität vor, p-Wert des F-Tests $= 0,006$) und ein Wilcoxon-Mann-Whitney-Test, der von Verteilungen unabhängig ist. Tabelle 5.2 zeigt die hohe statistische Signifikanz.

Im **Bereich Constraints** lagen Daten aus allen drei Populationen vor. Entsprechend der Terminologie von C. Wohlin u. a. [170] wurde somit "Standard design 3" ausgeführt. In Abbildung 5.3 lässt sich anhand der Mittelwerte erkennen, dass Probanden die Constraints die in Solverational notiert waren, am schnell-



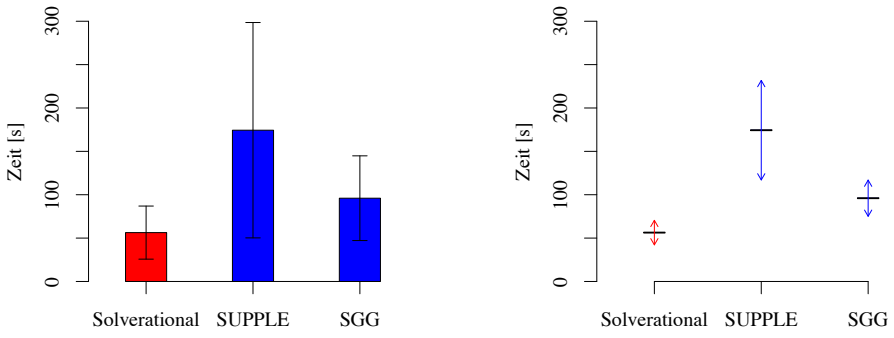
(a) Standardabweichung

(b) Konfidenzintervalle

Abbildung 5.2: Bereich Modelle: Mittelwerte, Standardabweichung und Konfidenzintervalle der Antwortzeiten

sten verstehen konnten – gefolgt von SGG und SUPPLE. Die drei Populationen wurden nur unter Vorbehalt als normal verteilt angenommen, da einige Ausreißer vorlagen, die Mediane nicht mittig in den Boxen lagen (s. Abbildung B.5) und Anderson-Darling und Shapiro-Wilk-Tests einen signifikanten Unterschied zur Normalverteilung bestätigten. Mit einem Levene-Test wurde geprüft ob Heteroskedastizität vorlag, was nicht der Fall war (p-Wert $= 0,75$, Homoskedastizidität ist eine Voraussetzung für ANOVA).

Der Nachweis, dass es sich tatsächlich um drei verschiedene Populationen handelte, erfolgte zunächst durch eine einfaktorielle ANOVA, da es bei mehrfacher Ausführung des T-Tests zu einer Alphafehler-Kumulierung kommen kann. Das Ergebnis war statistisch hoch signifikant (p-Wert $= 1,1 * 10^{-4}$). Da die Annahme, es handle sich um Normalverteilungen im Fall von Solverational und SGG nur unter Vorbehalt getroffen wurde, wurde zur Überprüfung ein Kruskal-Wallis-Test durchgeführt, der von der Normalverteilung unabhängig ist. Dieser bestätigte das Ergebnis der ANOVA (p-Wert $= 2,2 * 10^{-16}$). Der Nachweis statistischer Signifikanz zwischen den Populationen erfolgte paarweise mit dem T-Test, sowie zur Überprüfung mit dem Wilcoxon-Mann-Whitney-Test wegen fehlender Normalverteilungen. Um eine Alphafehler-Kumulierung zu vermeiden, wurde die konservative Bonferroni-Korrektur verwendet (Signifikanz-Niveau $= 0,9833$). Alle paarweisen



(a) Standardabweichung (b) Konfidenzintervalle

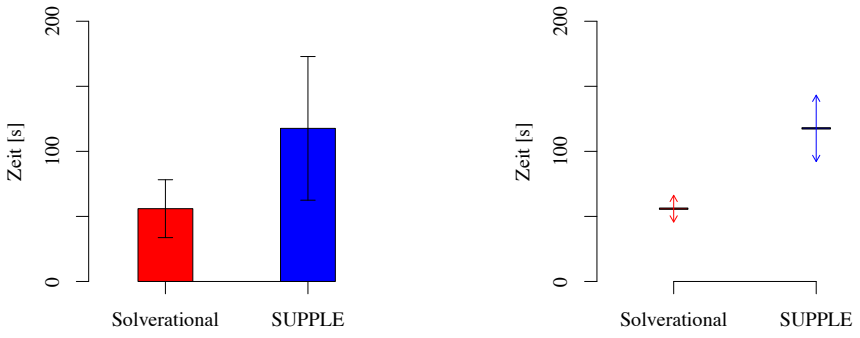
Abbildung 5.3: Bereich Constraints: Mittelwerte, Standardabweichung und Konfidenzintervalle der Antwortzeiten

Vergleich	T-Test (Welch)	Wilcoxon-Mann-Whitney	F-Test
Solverational - SGG	$0,004$	$0,003$	$0,05$
Solverational - SUPPLE	$0,0009$	$0,0002$	$0,0000005$
SUPPLE - SGG	$0,019$	$0,042$	$0,0001$

Tabelle 5.3: p-Werte der Tests der paarweisen Vergleiche

Tests waren statistisch signifikant und entsprechende p-Werte finden sich in Tabelle 5.3. Heteroskedastizität lag vor (F-Tests) und entsprechend wurden Welch-Tests durchgeführt. Die Ergebnisse dieser Tests wurden allerdings nur mit Vorbehalt akzeptiert, da der F-Test sehr anfällig für nicht normale Verteilungen ist. Da der Welch-Test somit korrekt gewählt wurde, ist die Reihenfolge der Ergebnisse aus Abbildung 5.3 somit statistisch signifikant.

Im **Bereich Optimierung** konnten nur Quelltexte verglichen werden, die in Solverational und SUPPLE notiert wurden, da SGG keine Unterstützung für Optimierung anbot. Abbildung 5.4 fasst die Ergebnisse in Mittelwerten und Standardabweichungen zusammen. Es ist leicht zu erkennen, dass im Verstehen von



(a) Standardabweichung (b) Konfidenzintervalle

Abbildung 5.4: Bereich Optimierung: Mittelwerte, Standardabweichung und Konfidenzintervalle der Antwortzeiten

Vergleich	T-Test (Welch)	F-Test
Solverational - SUPPLE	$0,0002$	$0,0005$

Tabelle 5.4: p-Werte der Tests: Notation von Zielfunktionen

Zielfunktionen Solverational einen klaren Vorteil bietet.

Um Signifikanz nachzuweisen wurde für die Populationen mit Zielfunktionen in Solverational und SUPPLE getestet, ob es sich nicht um Normalerteilungen handelt (Shapiro-Wilk-Tests, p-Werte: $0,08$ bzw. $0,07$). Entsprechend wurde die Annahme, es handle sich um Normalerteilungen, nicht verworfen. Es konnten also F-Test und Welch-Test ohne Einschränkung verwendet werden. Der F-Test wies Heteroskedastizität nach. Es wurde also ein Welch-Test durchgeführt mit dem hohe statistische Signifikanz nachgewiesen werden konnte. Tabelle 5.4 fasst die Ergebnisse zusammen.

Lerneffekte sind im Rahmen von Studien prinzipiell eher unerwünscht, da sie die Ergebnisse verfälschen (siehe nächster Abschnitt). Im vorliegenden Fall war von einem Lerneffekt auszugehen, da die Sprachen für die Teilnehmer unbekannt waren. Als Maß für einen Lerneffekt kann die Korrelation dienen, denn ein Lerneffekt

würde eine Abnahme der Antwortzeiten zwischen den drei Fragen jedes Bereiches erwarten lassen. Im vorliegenden Fall war die Korrelation zwischen jeweils der ersten und der zweiten Frage 0,64 sowie 0,6 zwischen der ersten und der dritten. Ein Lerneffekt konnte damit über die Korrelation nachgewiesen werden. Er wäre aber auch nach der in dieser Arbeit vertretenen Ansicht nicht zu vermeiden gewesen, wenn – wie in dieser Studie – auch mehrere Fragen gestellt werden sollen, um eine breitere Abdeckung von verschiedenen Constraints, Modellen und Zielfunktionen zu erhalten.

5.2.4 Bedeutendste Faktoren, die die Validität einschränken können

Reliabilität war in der vorliegenden Studie nicht sinnvoll zu berechnen: Die Antworten waren nicht eine Skala zur Operationalisierung z.B. des "persönlichen Geschmacks", sondern vielmehr war eine Antwort aus vielen die einzig "richtige". Mit den gestellten Fragen wurde die Zeit gemessen, sodass die klassischen statistischen Methoden zur Bestimmung der Reliabilität nicht angewendet werden können (keine Skala von 1 bis 10, sondern nur ein Wert richtig, der nicht den Messwert repräsentierte). Die im Anhang dargestellten Verteilungen lassen aber vermuten, dass Teilnehmer ähnliche Zeiten benötigten – sodass eine Gausskurve entstand – und daher zumindest eine gewisse Wiederholbarkeit der Studie angenommen werden kann. Auch die Korrelation zwischen den Zeiten verschiedener Fragen zum selben Bereich (jeweils Modelle, Constraints oder Optimierung) der gleichen Nutzer betrug trotz des Lerneffektes noch 0,64 bei der ersten und 0,6 bei der zweiten Frage (Korrelation). Nach klassischen Annahmen wird bei einer Analyse der Reliabilität für die Korrelation 0,7 erwartet, sie war aber wegen der Lerneffekte kleiner als 0,7.

Die interne Validität wird durch die Wahl der Fragen stark beeinflusst. Allerdings müssen für empirische Studien naturgemäß Beispiele gewählt werden, die nicht den gesamten Umfang aller Programme abdecken können, die in den Sprachen geschrieben werden können. Die Wahl der Beispiele bestimmt dabei stark die Ergebnisse. Jede Aussage, die durch quantitative Studien überprüft wird, die Sprachen mit hinreichend großer Anzahl verschiedener Quelltexte untersuchen, wird streng genommen nur unter der Wahl der Beispiele gültig sein. Dies ist für die gegebene Anzahl möglicher Transformationen der drei Sprachen sehr einschränkend. Eine Alternative wäre eine qualitative Studie gewesen, die aber auf Grund mangelnder Experten nicht durchgeführt werden konnte. Aus diesem Grund wurde die beschriebene Studie durchgeführt. Die Ergebnisse beschränken sich demnach auf die Transformation von Benutzerschnittstellen im Rahmen der gewählten Quelltexte der Fragen. Da die Quelltexte der Fragen Übersetzungen von Beispielen aus dem

SUPPLE-System waren, wurden sie allerdings zu großen Teilen nicht willkürlich gewählt, sondern entsprachen der Vorgabe die das SUPPLE-System machte. Die interne Validität der Studie entspricht also der Qualität der Beispiele aus dem SUPPLE-System.

Eine Einschätzung der Gültigkeit der Verallgemeinerung der Ergebnisse bzgl. der befragten Probanden ist nicht einfach (externe Validität). Dies beruht zum Teil auf der Tatsache, dass es sich um eine Online-Umfrage handelt [131]; allerdings ist bereits die Wahl der Teilnehmer durch die beschränkten Ressourcen schwierig. Eine Studie mit Programmierern aus der Wirtschaft wäre zu teuer und zu aufwändig gewesen. Daher musste auf Doktoranden ausgewichen werden. Allerdings gaben alle – bis auf einen Teilnehmer – der Studie an, bereits umfangreiche Erfahrung in der Programmierung mit JAVA zu haben, aber alle hatten bereits mehrere Programme in JAVA geschrieben (Schnitt: 4,53 von 5 Punkten (1 Sprache unbekannt, 5 Experte), Standardabweichung 0,67). Insofern waren die Unterschiede zu klassischen Entwicklern aus der Industrie zumindest in den Programmierkenntnissen als relativ klein zu bewerten. Auch die Frage, ob sich die Probanden mit Graph-Grammatiken und Modell-zu-Modelltransformationssprachen auskennen würden, beantworteten die Probanden größtenteils damit dass diese ihnen unbekannt seien (Schnitt: 1,68 von 5 Punkten (1 Sprache unbekannt, 5 Experte), Standardabweichung 0,67).

5.2.5 Interpretation

Die durchgeführte Studie bestätigte die Vermutung, dass bei den gewählten Beispielen ein signifikanter Unterschied zwischen grafischen Modellen und textuell notierten Modellen in JAVA/SUPPLE besteht. Dies stimmte mit den grundsätzlichen Annahmen überein, die zur Verwendung von Graphen zur Notation von Benutzerschnittstellen führten. Grafische Benutzerschnittstellen werden seit langem in sogenannten GUI-Buildern grafisch notiert. Die grafischen Komponenten, die in GUI-Buildern verwendet werden, werden in gewisser Hinsicht ähnlich zu grafischen Modellen von Benutzerschnittstellen notiert. Die Probanden dürften auf Grund ihrer Programmierkenntnisse an GUI-Builder gewöhnt sein, was den gemessenen Effekt erklären könnte. Da in der Studie nur Modelle von Benutzerschnittstellen geprüft wurden, sind die Ergebnisse nicht auf beliebige Modelle zu verallgemeinern. Allerdings steht auf Grund der häufigen Verwendung von grafischen GUI-Buildern und deren Ähnlichkeiten zu grafischen Modellen zu vermuten, dass sich die Ergebnisse auf grafische Modelle von Benutzerschnittstellen verallgemeinern lassen. Modelle von Benutzerschnittstellen sollten wegen besserer Verständlichkeit in grafischen Modellen notiert werden.

Beim Verstehen notierter Constraints lag die Geschwindigkeit des Verstehens von Solverational leicht vor der für SGGs. Constraints, die in SUPPLE notiert

wurden, konnten nur deutlich langsamer verstanden werden. Dies ist insofern überraschend, weil die Probanden mit der Sprache JAVA (SUPPLE) vertraut waren, aber nicht mit Transformationssprachen! Es ist außerdem verblüffend, dass die Probanden im Fall von Solverational alle sehr ähnliche Geschwindigkeiten erzielten. Hingegen waren die Ergebnisse von SUPPLE sehr breit gestreut, was an unterschiedlichen Kenntnissen der Programmiersprache JAVA liegen könnte. Die Selbsteinschätzung, wie qualifiziert sich ein Proband im Entwickeln mit einer Programmiersprache fühlt, ist subjektiv, sodass die Einschätzung "gut" aus der Sichtweise eines anderen Probanden "schlecht" bedeuten kann.

Der klare Geschwindigkeitsvorteil beim Verstehen von Zielfunktionen in der Sprache Solverational gegenüber SUPPLE war zu erwarten, während SGGs gar keine Zielfunktionen erlaubten. In SUPPLE müssen Zielfunktionen im Allgemeinen direkt in JAVA-Code implementiert werden. Solverational bietet direkt Sprachkonstrukte an, mit denen sich Zielfunktionen klar als solche deklarieren lassen – und zusätzlich deklarativ notiert werden können. Es kann daher angenommen werden, dass dieses Ergebnis sowohl über die Probanden, als auch über die gewählten Beispiele hinaus verallgemeinert werden kann. SGGs erlauben hingegen gar keine Zielfunktionen. Beim Verstehen der Zielfunktionen kann sich daher Solverational klar gegen die beiden anderen Sprachen durchsetzen.

Solverational konnte sich sogar in allen drei Bereichen als Sieger positionieren. Nach Maßgabe der Ergebnisse der durchgeführten Untersuchung kann also davon ausgegangen werden, dass Modelle, Constraints und Zielfunktionen, die in Solverational notiert wurden, einfacher zu verstehen sind als in SGGs oder JAVA/SUPPLE.

5.3 Schreibbarkeit

Schreibbarkeit ist wie Verständlichkeit im vorliegenden Fall nicht durch kognitive Modelle zu erfassen. Aufbauend auf den Fragen aus dem letzten Abschnitt lässt sich aber messen, wie viel Code für die verschiedenen Fragen implementiert werden mussten. Dies ist eine klassische Metrik, um den Aufwand von Softwareprojekten zu schätzen (s. z.B. [34]). Im vorliegenden Fall ist das nur eine sehr grobe Approximation, die nur einen Anhaltspunkt geben soll und nicht den Anspruch erhebt, in jedem Fall richtige Ergebnisse zu liefern. Es ist allerdings der erste Versuch überhaupt, die Schreibbarkeit der drei Modelltransformationssprachen für Modelle von Benutzerschnittstellen zu schätzen und damit auch der erste Richtwert.

Als Operationalisierung für die kürzeste Notation wurde entsprechend die Anzahl relevanter terminaler Symbole gewählt, die zum Notieren minimal benötigt wurde.

Wie im vorigen Abschnitt auch, wurden die von D. Perry u.a. [122] vorge-

schlagenen Schritte für Studien in der empirischen Evaluation von Ansätzen des Software Engineering angewendet.

5.3.1 Hypothese

Ähnlich zur Studie über die Verständlichkeit der drei Sprachen waren für Schreibbarkeit drei Bereiche der Sprachen zu testen, soweit die Sprachen Konstrukte zur Implementierung vorsahen:

1. Bereich Modelle: Lassen sich grafische oder in SUPPLE textuell notierte Modelle von Benutzerschnittstellen kürzer notieren? In diesem Fall werden wieder Solverational und SGG mit einer identischen (grafischen) Notation mit der Notation von SUPPLE verglichen.

2. Bereich Constraints: Welche Notation für Constraints lässt sich mit den wenigsten terminalen Symbolen notieren? Constraints werden in allen drei Sprachen unterschiedlich notiert, sodass alle Sprachen untersucht wurden.

3. Bereich Optimierung: Welche Notation für Zielfunktionen lässt sich am kürzesten notieren? In SGGs können Zielfunktionen nicht notiert werden, sodass nur Solverational und SUPPLE verglichen werden.

5.3.2 Studiendesign

Basierend auf den Fragen des Fragebogens aus Abschnitt 5.2 wurde ein Vergleich auf Basis notwendiger terminaler Symbole durchgeführt.

Für die Durchführung des Vergleichs standen in jedem Bereich drei Fragen zur Verfügung. In jeder Sprache wurden diese drei Fragen möglichst einfach und kurz implementiert. Zum Vergleichen wurden nur die terminalen Symbole gezählt, die tatsächlich zur Implementierung benötigt wurden und nicht durch nicht-terminale Symbole implizit vorgegeben waren (d.h. z.B. bei Funktionsaufrufen wurden "(" und ")" nicht gezählt).

Alle Fragen und Antworten hatten – bis auf den jeweils angezeigten Quellcode – in jeder Sprache den gleichen Wortlaut. Auch die Modelle, die Constraints oder die Zielfunktionen hatten jeweils gleiche Funktionen, nur wurden diese in den drei unterschiedlichen Sprachen dargestellt. Dadurch war Vergleichbarkeit der verschiedenen Quellcodes hergestellt.

5.3.3 Datenanalyse und Präsentation

Die Ergebnisse sind in Tabelle 5.5 zusammengefasst.

Vergleich	Größe	SGG	Solverational	SUPPLE
Modelle	Mittelwert		9	33,67
	Standardabweichung		5	19,09
	Konfidenzintervall		5,66	21,6
Constraints	Mittelwert	16,67	17,33	18
	Standardabweichung	5,77	4,62	3,46
	Konfidenzintervall	6,53	5,23	3,92
Optimierung	Mittelwert		9,33	7,67
	Standardabweichung		5,77	2,89
	Konfidenzintervall		6,53	3,27

Tabelle 5.5: Ergebnisse Schreibbarkeit

Im **Bereich Modelle** wurden die grafischen Modelle von Benutzerschnittstellen (die Modelle in Solverational und SGG repräsentierten) mit den textuell notierten von SUPPLE verglichen. Die Anzahl terminaler Symbole, die für das Notieren von grafischen Modellen benötigt wird, ist erheblich geringer. Auch wenn das Ergebnis nicht statistisch signifikant ist, so ist eine starke Tendenz zugunsten grafisch notierter Modelle erkennbar. Diese These wird durch die zahlreichen grafischen GUI-Builder unterstützt, die zeigen, dass sich grafische Repräsentationen durchgesetzt haben. Dies liegt vermutlich auch an höherer Effizienz bei der Entwicklung von grafischen Benutzerschnittstellen mit grafischen Werkzeugen, die dem Ergebnis dadurch näher sind als textuelle.

Im **Bereich Constraints** lagen Mittelwerte über der Anzahl terminaler Symbole der Quellcodes aller drei Sprachen vor. Die Mittelwerte in Tabelle 5.5 lassen erkennen, dass Constraints, die in SGG notiert waren, am wenigsten terminale Symbole für die Notation benötigten – gefolgt von Solverational und SUPPLE. Allerdings liegen die Mittelwerte sehr nahe beisammen und eine Interpretation der Ergebnisse ist bei so wenigen Messpunkten nicht statistisch signifikant. Die Mittelwerte unterscheiden sich in weniger als einem terminalen Symbol, während die Standardabweichung mindestens drei terminale Symbole betrug.

Im **Bereich Optimierung** konnten nur Quelltexte verglichen werden, die in Solverational und SUPPLE notiert wurden, da SGG keine Unterstützung für Optimierung anbot. Tabelle 5.5 zeigt dass der Mittelwert der Anzahl terminaler Symbole, die zum Notieren in SUPPLE benötigt wurden, etwas niedriger ist. Die Ergebnisse sind statistisch nicht signifikant.

5.3.4 Bedeutendste Faktoren, die die Validität einschränken können

Die Qualität der Untersuchung leidet an der geringen Anzahl untersuchter Beispiele. Es wurden jeweils nur drei Messpunkte genommen, dadurch konnte keine statistische Signifikanz nachgewiesen werden. Zusätzlich beschränkte die geringe Anzahl von Beispielen die Möglichkeit, die Ergebnisse auf eine größere Gruppe von Transformationen zu verallgemeinern.

Relevante terminale Symbole zu identifizieren, stellte sich im Fall von SUPPLE als schwierig heraus, da im Falle von SUPPLE der Unterschied zwischen Constraints, Zielfunktionen, JAVA und existierendem Framework von K. Gajos nicht klar definiert wurde. So war nicht klar, welche terminalen Symbole benötigt wurden und welche Funktionen tatsächlich neu zu entwickeln waren. Beim Zählen der Symbole bei den vorliegenden Fragen wurde von der minimalen Anzahl ausgegangen – dies könnte eine deutliche Unterschätzung der real benötigten Zeilen Code darstellen. Da diese Angabe auf persönlichen Entscheidungen zum Design von Applikationen beruht, ist die Reliabilität der Untersuchung vermutlich gering.

Auch der Vergleich von grafischen Notationen mit textuellen ist recht schwierig zu bewerten (z.B. [57, 108, 123]). SUPPLE ist eine Erweiterung der Sprache JAVA, die eine allgemeine Programmiersprache ist, während es sich bei den anderen beiden Sprachen (bzw. Modellen) um domänenspezifische Sprachen handelt. Entsprechend weniger Symbole werden für die domänenspezifischen Sprachen benötigt, da diese bereits auf die Domänen eingeschränkt sind und die Einschränkung nicht durch weitere Symbole im Quelltext geschehen muss.

5.3.5 Interpretation

Die Ergebnisse zeigen, dass das Schreiben von grafischen Modellen gegenüber Modellen, die in SUPPLE notiert wurden, deutlich weniger terminale Symbole benötigt. Dies liegt daran, dass es sich bei den grafischen Modellen um domänenspezifische Sprachen handelt, bei SUPPLE/JAVA nicht.

Im Bereich Constraints konnten keine klaren Unterschiede zwischen den Sprachen gefunden werden. Die Mittelwerte unterschieden sich zu wenig. Es kann also davon ausgegangen werden, dass die drei Sprachen im Bereich Constraints ähnlich gute Schreibbarkeit aufweisen.

Der Bereich Optimierung ergab, dass Zielfunktionen (nicht signifikant) kürzer in SUPPLE notiert werden können. Während Zielfunktionen in Solverational über beliebig große Mengen von Komponenten iterieren können, sind Zielfunktionen in SUPPLE immer über allen beteiligten Komponenten der Benutzerschnittstelle definiert. Dies ist nicht immer erwünscht, führt aber bei manchen Zielfunktionen zur Nutzung von weniger terminalen Symbolen. Das Ergebnis war statistisch nicht

signifikant. Intuitiv wäre zu erwarten gewesen, dass für Sprachen, die Konstrukte für Zielfunktionen anbieten, weniger terminale Symbole benötigt worden wären als für eine allgemeine Programmiersprache. Die Ergebnisse der Untersuchung entsprechen nicht dieser Intuition – Abschnitt 5.3.4 gibt Hinweise auf die Gründe. Die Ergebnisse werden nur unter Vorbehalt verwendet, denn zum Zeitpunkt des Verfassens dieser Arbeit lagen in der Literatur keine Ergebnisse vor, die Hinweise auf die Schreibbarkeit der drei Sprachen geben würden.

Die Verallgemeinerung der Ergebnisse ist problematisch, da zu wenige Beispiele untersucht wurden. Auch wenn die Ergebnisse hier für die Schreibbarkeit zur Ableitung einer optimalen Sprache verwendet werden, sollte die Untersuchung zur Überprüfung der Ergebnisse in Zukunft mit mehr Messungen wiederholt werden.

Fazit: Solverational konnte sich im Bereich Modelle gegen den Konkurrenten JAVA/SUPPLE statistisch signifikant durchsetzen. In den Bereichen Constraints und Zielfunktionen waren die Ergebnisse nicht signifikant und es kann daher keine klare Aussage getroffen werden.

5.4 Allgemeingültigkeit der Ansätze

Domänenspezifische Sprachen sind nicht automatisch Turing-vollständig, da sie speziell für eine Domäne entwickelt wurden und damit nicht überall eingesetzt werden. Wenn in einem Projekt viele domänenspezifische Sprachen eingesetzt werden, führt dies potenziell dazu, dass sich Entwickler in verschiedene Sprachen einarbeiten müssen. Will man diesen Aufwand einsparen, ist es daher notwendig, die Sprachen nach deren Allgemeingültigkeit zu klassifizieren. Dazu sollen die drei vorgestellten Sprachen nach möglichen Einsatzgebieten klassifiziert werden. Nach D. Olsen [112] kann damit Allgemeingültigkeit (engl. “Generality”) nachgewiesen werden.

5.4.1 Untersuchte Transformationen

In Abschnitt 4.5 wurden bereits mehrere Transformationen zur Transformation von Modellen von Benutzerschnittstellen vorgestellt. Es wird untersucht ob diese sich auch mit SUPPLE/JAVA und SGGs implementieren lassen.

Zwei neue, hier vorgestellte Typen von Transformationen, können nur in Transformationssprachen notiert werden, die Zielfunktionen unterstützen. Sie befassen sich mit den Domänen Scheduling und Service Composition nicht-funktionaler Eigenschaften.

Zusätzlich werden drei theoretische Typen von Transformationen vorgestellt, die sich derzeit nicht in Solverational implementieren lassen.

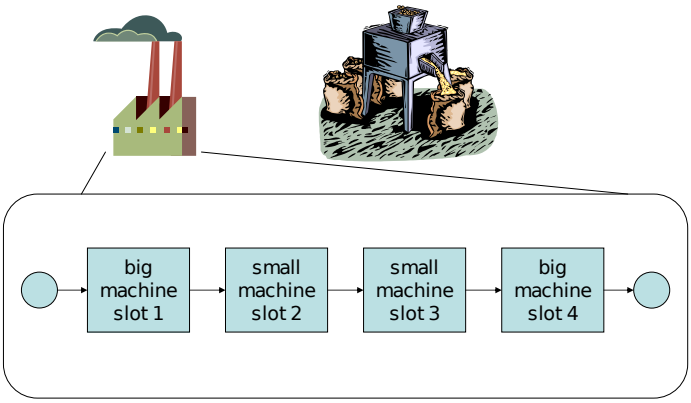


Abbildung 5.5: Scheduling Transformation: Beispiel Schreibwarenfabrik

Typ	Leihgebühren	Produktionsgeschwindigkeit
DrillerStandard	1500	200
DrillerCheap	1000	100
DrillerPremium	2000	300

Tabelle 5.6: Scheduling Transformation: Eigenschaften der Maschinen

5.4.1.1 Scheduling Transformation

Stellvertretend für eine Klasse von Scheduling Problemen zur Prozessoptimierung soll eine Transformation ein spezielles Scheduling-Problem lösen, dass den Gewinn eines Unternehmens maximiert. Dazu soll die Transformation ein abstraktes Modell einer Fabrik in ein Modell der Fabrik überführen, das mit Repräsentationen von Maschinen bestückt ist.

Im vorliegenden Beispiel sollen Bleistiftanspitzer in einer Schreibwarenfabrik produziert werden. Der Manager der Schreibwarenfabrik möchte für die Stellplätze des abstrakten Modells der Fabrik aus Abbildung 5.5 Maschinen finden, die den Gewinn maximieren. Es stehen Stellplätze für zwei große und zwei kleine Maschinen zur Verfügung. Die Maschinen kosten unterschiedliche Leihgebühren und produzieren unterschiedlich schnell. Da das Problem bereits als Modell vorliegt, kann eine passende Modell-zu-Modelltransformation ein Modell mit einer Liste von Maschinen ausgeben.

Den Maschinen werden Leihgebühren und Produktionsgeschwindigkeiten zuge-

Task	Leihgebühren	Ausführungszeit	Zuverlässigkeit
AdelphiChemicalLab	1500	200	300
BiblisChemicalLab	150	2000	300
BushmillsMaltChemicalLab	500	350	300

Tabelle 5.7: Service Composition: Eigenschaften der Tasks

ordnet. Zum besseren Verständnis findet sich in Tabelle 5.6 ein Beispiel für verschiedene Bohrer. Eine Produktionslinie ist genau so schnell wie das langsamste Glied in der Produktionskette; also ist das Minimum der Produktionsgeschwindigkeiten der Maschinen die Produktionsgeschwindigkeit der Produktionskette. Jeder Bleistiftanspitzer bringt dem Unternehmen 5 Euro Umsatz. Auf Grund dieser Angaben kann die Zielfunktion für den Gewinn wie folgt angegeben werden:

$$5 * \min(Produktionsgeschwindigkeiten) - \sum_{i=1}^{4} Leihgebühr(Maschine_i)$$

Das Ergebnis der Transformation soll ein Modell sein, das diese Zielfunktion maximiert.

5.4.1.2 Service Composition

Gegeben ist ein Modell eines Prozesses in Form eines Workflows (einer sog. "Composition"), der aus mehreren Teilprozessen besteht. Zusätzlich wurden die Teilprozesse bereits in verschiedener Weise in Software implementiert. Eine Transformation soll nun eine Zuordnung dieser Implementierungen, die "Tasks" genannt werden, zu den Teilprozessen finden. Jedem "Task" sind zusätzlich einige nicht-funktionale Eigenschaften zugeordnet. Diese umfassen im vorliegenden Beispiel die Gebühren, den Task im Internet zu "leihen", die Zeit zum Ausführen und einen Wert über die Zuverlässigkeit des Tasks (die Zeit, die der Task im Jahr nicht verfügbar ist). Ein Nutzer der Service Composition ist daran interessiert, die Kosten für die Benutzung der Composition gering zu halten; da alle aufgeführten nicht-funktionalen Eigenschaften in gewisser Weise Kosten darstellen, soll eine Kombination aus den funktionalen Eigenschaften der verschiedenen Tasks minimiert werden, sodass die nach Kosten günstige Composition automatisch gewählt wird.

Das Beispiel wurde einem Szenario des Projekts Theseus entnommen (für eine detaillierte Einführung s. [126]). Es sollte eine Zuordnung von Implementierungen zu Tasks für die Herstellung eines Produktes gefunden werden. Im vorliegenden Fall wurden bei der Herstellung drei Prozesse verwendet, s. Abbildung 5.6. Eine

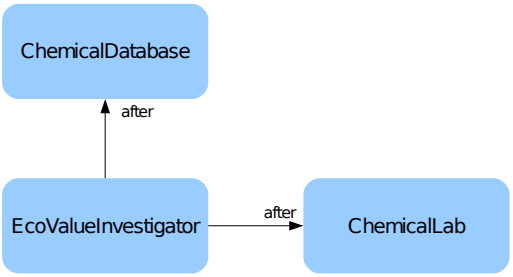


Abbildung 5.6: Service Composition: Workflow ökologische Herstellung

Zuordnung von Tasks zu den drei Prozessen konnte durch eine Gewichtung von Leihgebühren, Ausführungszeit und Zuverlässigkeit durch die Anwender der Transformation auf deren spezielle Bedürfnisse angepasst werden. Im vorliegenden Fall werden beispielhaft die Gewichte 5 für Leihgebühren, 2 für die Ausführungszeit und 3 für die Zuverlässigkeit angenommen. Zum besseren Verständnis finden sich in Tabelle 5.7 die Beispiele für drei verschiedene Tasks. Mittels SMART (einer einfachen Technik, mehrere Parameter einer Zielfunktion zu verrechnen, s. [46]) können die verschiedenen nicht-funktionalen Eigenschaften für n Tasks in einer Zielfunktion verwendet werden:

$$5 * \sum_{i=1}^{n} Leihgebühr_i + 2 * \sum_{i=1}^{n} Ausführungszeit_i + 3 * \sum_{i=1}^{n} Zuverlässigkeit_i$$

Das Ergebnis der Transformation sollte ein Modell sein, das diese Zielfunktion minimiert. Eine ausführliche Beschreibung der Transformation findet sich in [126].

5.4.1.3 Schwierige Transformationen

Im folgenden werden drei Typen von Transformationen vorgestellt, die sich auf Grund spezieller Anforderungen nicht in Solverational oder SGGs implementieren lassen. Diese drei Typen wurden ausgewählt, da sie nach Meinung dieser Arbeit die wesentlichen Nachteile der Sprache Solverational erklären, und die in einigen wenigen praktischen Beispielen zu Problemen bei der Implementierung von Transformationen führen könnten.

Hierbei handelt es sich um drei bekannte Nachteile:

Mutliple Vergleiche Es ist schwer in Solverational ein Attribut eines Elements mit beliebigen Attributen beliebiger anderer Elemente zu vergleichen. Dies

liegt daran, dass Solverational bis auf die Definition von Zielfunktionen keine globalen Constraints unterstützt, sondern nur lokale. Das bedeutet, dass ein Constraint ein Attribut eines Modellelements mit Attributen anderer Elemente nur dann vergleichen kann, wenn diese über Assoziationen verfügen. Andernfalls muss das Constraint beliebig viele weitere Assoziationen nutzen um transitiv mit den Modellelementen verglichen zu werden. Dafür müssen die Assoziationen einzeln in der Transformation aufgeführt werden, was zu sehr umfangreichen Quellcodes führen kann.

Planungsalgorithmen Planungsalgorithmen können z.B. für die Transformation von Goal-Modellen (also Modellen, die Ziele einer modellierten Applikation in deklarativer Weise festhalten) in Task-Modelle eingesetzt werden. Dabei liegen im Goal-Modell eine Menge von Zielen (Goals) vor, die durch eine Menge von Tasks des Task-Modells (oder Workflows) implementiert werden sollen. Der Planungsalgorithmus erstellt dabei aus einer Menge von elementaren Aufgaben eine Reihenfolge von elementaren Aufgaben, die die Ziele aus dem Goal-Modell erfüllen können. Solverational unterstützt in seiner derzeitigen Implementierung nur Constraints, die Attribute beschränken. Planungsalgorithmen können daher nicht mit Solverational implementiert werden, da dafür Constraints über der Struktur des Modells (Graphen) notwendig wären.

Viele Komponenten Die derzeitige Implementierung von Solverational kann unter der Verwendung von Constraints sehr schnell Ergebnisse berechnen (s. Abschnitt 5.6). Dies ist leider nicht der Fall, wenn komplexe nicht-lineare Constraints verwendet werden. Hierbei sind auch einige der festen Constraints betroffen, die in SGG notiert werden können, da z.B. zur Berechnung der Entfernung eine euklidische Distanz berechnet werden muss (s. Abschnitt 5.5). Abhängig von den Constraints und der Anzahl der Komponenten kann die Berechnungsdauer exponentiell steigen. Zusätzlich ist nicht-lineare Optimierung – sollte kein spezieller Algorithmus zur Lösung oder Approximation entwickelt worden sein – als NP-vollständig bekannt. Aus diesem Grund ist die Verwendung von Zielfunktionen nur für wenige Komponenten praktisch nutzbar.

5.4.2 Datenanalyse und Präsentation

SUPPLE/JAVA baut auf der Sprache JAVA auf und ist damit als Erweiterung einer klassischen Programmiersprache zu verstehen. Da JAVA an sich bereits Turing-vollständig ist, ist es auch die Kombination aus SUPPLE und JAVA. Aus diesem Grund sind alle denkbaren mathematischen Funktionen (und damit

Domäne	Transformation	SGG	Solverational	SUPPLE
Benutzer-schnitt-stellen	Benchmark	+	+	+
	Anoto	+	+	+
	Spezifische Hardware	+	+	+
	Panel		+	+
	AUI2CUI		+	+
Scheduling	Schreibwarenfabrik		+	+
Service Composi-tion	Nicht-funktionale Eigenschaften		+	+
Benutzer-schnitt-stellen	Multiple Vergleiche			+
	Planungsalgorithmen			+
	Viele Komponenten	+		+
Gesamt		4	7	10

Tabelle 5.8: Ergebnisse: Allgemeingültigkeit der Sprachen

Transformationen) implementierbar. Allerdings wird SUPPLE dabei nicht direkt verglichen und SUPPLE/JAVA wird daher in gewisser Hinsicht durch die Pauschale Annahme bevorzugt, dass die Transformationen immer implementierbar sind. JAVA/SUPPLE wird daher außer Konkurrenz betrachtet.

SGGs sind speziell für die Entwicklung von Benutzerschnittstellen gut geeignet. Obwohl die unterliegende Sprache "RGG" für beliebige Domänen entwickelt wurde, lassen sich nicht alle Graphen transformieren [88]. Die Sprache ist auf eine bestimmte Menge von Constraints beschränkt. Insbesondere sind Zielfunktionen nicht mit SGGs und RGGs zu implementieren. Aus diesem Grund können die Transformationen "Panel" und "AUI2CUI" nicht mit SGGs implementiert werden, da diese Zielfunktionen enthalten. Auch das Scheduling-Problem und die Service Composition mit nicht-funktionalen Eigenschaften erfordern Optimierung und können nicht mit SGGs implementiert werden.

Hingegen sind SGGs gut geeignet, sehr große Benutzerschnittstellen zu transformieren, denn der Algorithmus zur Interpretation von SGGs kann in polynomieller Laufzeit ausgeführt werden [88]. Die Graphen müssen dabei aber die "Selection-free"-Bedingung (s. [88]) erfüllen. Es ist nicht klar wie viele Graphen tatsächlich transformiert werden können. Es können nicht beliebige Constraints formuliert werden und Zielfunktionen können weder definiert noch automatisch optimiert werden.

Für Solverational wurden mögliche Implementierungen der ersten fünf Trans-

```
transformation Sharpener
        (a:factory, b:concretefactory) {
    ...
    top relation SmallMachine2Driller  {
        domain a c:factory::SmallMachine {
            position=2
        };
        alternative domain b d:concretefactory::DrillerStandard{
            leasing=1500,
            outputPerMonth=200
        };
        alternative domain b d:concretefactory::DrillerCheap{
            leasing=1000,
            outputPerMonth=100
        };
        alternative domain b d:concretefactory::DrillerPremium{
            leasing=2000,
            outputPerMonth=300
        };
    ...
    }
}
```

Listing 5.1: Scheduling Transformation: Details des Solverational Quellcodes

formationen bereits in Abschnitt 4.5 vorgestellt.

Auszüge aus den Implementierungen für die zwei neuen Transformationen werden in Listings 5.1 und 5.2 gegeben.

In Listing 5.1 findet sich ein Auszug aus dem Quellcode der Scheduling-Transformation. Man erkennt die Implementierung der Zielfunktion, sowie eine einfache Abbildung der Kosten für die Maschinen. Die Zielfunktion kann nahezu identisch aus Abschnitt 5.4.1.1 übernommen werden. Die verschiedenen Maschinen wurden in Form von alternativen Domains implementiert. Es handelt sich also um ein Derivatproblem. Die Solverational Transformationsmaschine kann dann mit Hilfe der Zielfunktion die "optimalen" Maschinen aussuchen.

In ähnlicher Weise wurde die Transformation mit Service Composition und nicht-funktionalen Eigenschaften implementiert (s. Listing 5.2). Die Zielfunktion konnte wieder übernommen werden. Die alternativen Domains repräsentieren in dieser Transformation die verschiedenen implementierten Tasks. Die Transformationsmaschine sucht die "optimalen" Tasks anhand der Zielfunktion aus.

Die "schwierigen Transformationen" können nicht mit Solverational implemen-

```
    ...
    minimize
      5*sum(concreteTaskMM::Task.allInstances()->rentingCosts)+
      2*sum(concreteTaskMM::Task.allInstances()->executionTime)+
      3*sum(concreteTaskMM::Task.allInstances()->reliability);
    ...
    top relation ChemicalLabToLabService {
        domain s i:abstractTaskMM::ChemicalLab { };
        alternative domain t
                o:concreteTaskMM::AdelphiChemicalLab {
            rentingCosts=1500,
            executionTime=200,
            reliability=300};
        alternative domain t
                o:concreteTaskMM::BiblisChemicalLab {
            rentingCosts=150,
            executionTime=2000,
            reliability=300};
        alternative domain t
                o:concreteTaskMM::BushmillsMaltChemicalLab {
            rentingCosts=500,
            executionTime=350,
            reliability=300};
    ...
    }
}
```

Listing 5.2: Service Composition: Details des Solverational Quellcodes

tiert werden.

5.4.3 Bedeutendste Faktoren, die die Validität einschränken können

Die untersuchten Transformationen sind nur eine Auswahl von möglichen Beispielen. Beispiele können nicht nach repräsentativen Methoden ausgewählt werden, da für alle drei Sprachen unendlich viele Beispiele existieren. Dadurch können die Ergebnisse nicht auf den kompletten Umfang der Sprachen verallgemeinert werden. Ferner könnte der Wahl der Beispiele unterstellt werden, dass sie zu Gunsten von Solverational gewählt wurden, da sich diese Arbeit vorwiegend mit Solverational befasst. Diesem Argument steht allerdings entgegen, dass absichtlich eine ganze

Aufgabe	SGG	Solverational	JAVA/SUPPLE
Benutzerschnittstellen	+	+	+
Viele Komponenten	+		+
Planungsalgorithmen			+
Scheduling		+	+
Service Composition		+	+
Multiple Vergleiche			+

Tabelle 5.9: Ergebnisse: Aufgaben, die in Transformationen adressiert werden können

Reihe von Transformationen untersucht wurden, die sich nicht mit Solverational implementieren lassen.

5.4.4 Interpretation und Fazit

Die vorgestellten Transformationen geben einen Eindruck, welche Aufgaben in welcher Sprache implementiert werden können. JAVA/SUPPLE schneidet besonders gut ab, da die Sprache Turing-vollständig ist. Allerdings ist JAVA keine Modell-zu-Modell-Transformationssprache und daher nicht mit den anderen Sprachen vergleichbar. Im Allgemeinen bleibt für SGGs und Solverational unklar, welche Aufgaben in Transformation gelöst werden können, auch wenn Teilmengen lösbarer und unlösbarer Aufgaben in dieser Arbeit für Solverational klassifiziert werden. Basierend auf den hier verglichenen Transformationen ist die Allgemeingültigkeit von Solverational gegenüber SGGs größer.

Tabelle 5.9 fasst die Möglichkeiten der Sprachen kurz zusammen, wobei SUPPLE hier außer Konkurrenz betrachtet wird, da SUPPLE keine Modell-zu-Modell-Transformationssprache zu Grunde liegt.

5.5 Ausdrucksmächtigkeit von SGG gegenüber Solverational

Im Rahmen dieser Arbeit wurde untersucht, inwieweit sich räumliche Relationen, die in SGG zur Definition von Abhängigkeiten zwischen zwei Modellelementen verwendet werden, in Solverational abbilden lassen.

Dazu wurde jede der in SGGs zur Verfügung stehenden Relationen in Constraints übersetzt. Am Ende des Abschnitts wird ein Beispiel gegeben, wie diese Constraints in Solverational verwendet werden können. Die Implementierbarkeit

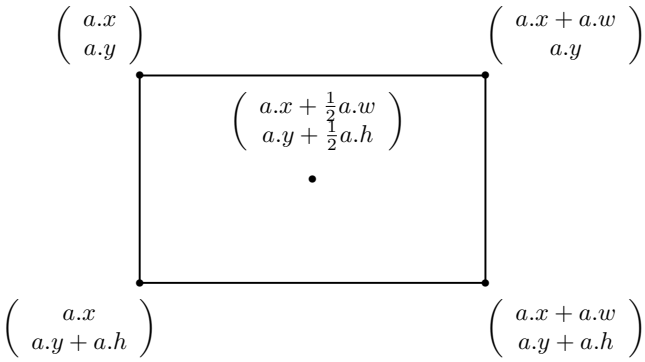


Abbildung 5.7: Punkte rechteckiger Objekte

der räumlichen Relationen zeigt daher, dass mit Solverational die gleiche Anzahl von Relationen dargestellt werden kann, wie mit SGGs. Dies bedeutet, dass im Allgemeinen die räumlichen Relationen von SGGs nicht mächtiger sind als Constraints in Solverational.

In den folgenden Abschnitten wird jede Relation aus Kapitel 3 aus Kong u.a. [88] in Constraints übersetzt und die Struktur oder Klassifikation zum einfacheren Vergleich von Kong u.a. übernommen. Die Überschriften der folgenden Abschnitte sind deshalb Englisch.

5.5.1 Räumliche Relationen

Die räumlichen Relationen wurden von Kong u.a. [88] identifiziert. Wenn dem in [88] nicht ausdrücklich widersprochen wurde, wird auch in der vorliegenden Arbeit angenommen, dass alle Objekte Rechtecke sind. Fast alle in klassischen WIMP-Benutzerschnittstellen verwendeten Objekte sind rechteckig (daher werden vermutlich auch von Kong u.a. Rechtecke verwendet).

Die Menge aller Objekte ist $\mathcal{O}$. Wie in den Beispielen aus Abschnitt 4.5 wird angenommen, dass die Koordinatenachsen nach rechts und unten wachsen. Die Koordinaten der linken oberen Ecke eines rechteckigen Objektes $a \in \mathcal{O}$ mit positiver Breite und Höhe ($a.w$ bzw. $a.h$) sind daher $(a.x, a.y)$, die der unteren rechten Ecke sind $(a.x + a.w, a.y + a.h)$. Abbildung 5.7 stellt den Sachverhalt grafisch dar.

5.5.2 Topology

Abbildung 4 in Kong u.a. [88] illustriert die Relationen aus dem Abschnitt "Topology" durch grafische Beispiele. Im Prinzip stellen die Relationen aus diesem

Abschnitt topologische Beziehungen her, d.h. z.B., dass ein Objekt "in" einem anderem liegen soll.

Touch: a touch $b \Longleftrightarrow$
$[(a.x + a.w = b.x) \wedge (a.y + a.h \geq b.y) \wedge (a.y \leq b.y + b.h)] \vee$
$[(a.x = b.x + b.w) \wedge (a.y \leq b.y + b.h) \wedge (a.y + a.h \geq b.y)] \vee$
$[(a.x + a.w \geq b.x) \wedge (a.x \leq b.x + b.w) \wedge (a.y + a.h = b.y)] \vee$
$[(a.x \leq b.x + b.w) \wedge (a.x + a.w \geq b.x) \wedge (a.y = b.y + b.h)]$

In: a in $b \Longleftrightarrow$
$(a.x \geq b.x) \wedge (a.y \geq b.y) \wedge (a.x + a.w \leq b.x + b.w) \wedge (a.y + a.h \leq b.y + b.h)$

Overlap: a overlap $b \Longleftrightarrow$
$(a.x < b.x + b.w) \wedge (a.y < b.y + b.h) \wedge (a.x + a.w > b.x) \wedge (a.y + a.h > b.y)$

Disjoint: a disjoint $b \Longleftrightarrow$
$(a.x > b.x + b.w) \vee (a.y > b.y + b.h) \vee (a.x + a.w < b.x) \vee (a.y + a.h < b.y)$

Cross: ist ein Spezialfall, da dies der einzige Fall ist, in dem Kong explizit Geradensegmente benutzt. In Solverational können Geraden und Rechtecke ebenfalls häufig gegeneinander ausgetauscht werden, indem eine Gerade in 2-Punkte-Form angegeben wird: $(a.x, a.y)$ und $(a.x + a.w, a.y + a.h)$. Für ein Geradensegment a, wird die passende Gerade als l_a bezeichnet und eine rechteckige Hülle mit $\bar{a}$, die das Geradensegment exakt enthält. $a.h$ kann dann auch negative Werte annehmen, da mit $a.h$ die Steigung bezeichnet wird. Abbildung 5.8 soll den Sachverhalt verdeutlichen.

$$l_a := \{(x_l, y_l) \in \mathbb{R}^2 | (a.h)(x_l - a.x) + (-a.w)(y_l - a.y) = 0\}$$

und

$$\bar{a} := \{(x_r, y_r) \in \mathbb{R}^2 | [a.x \leq x_r \leq a.x + a.w] \wedge [(a.y \leq y_r \leq a.y + a.h) \vee (a.y + a.h \leq y_r \leq a.y)]\}.$$

Für ein Geradensegment a und ein Rechteck b soll sichergestellt werden, dass gilt

$$a \text{ cross } b \Longleftrightarrow \{l_a \text{ geht durch } b\} \wedge \{\bar{a} \text{ overlap } b\}.$$

Die Constraints für "overlap" wurden bereits hergeleitet. Die Bedingung $\{l_a$ geht durch $b\}$ ist äquivalent zu

$$[\langle (a.h, -a.w), (b.x, b.y) - (a.x, a.y) \rangle > 0] \wedge [\langle (a.h, -a.w), (b.x + b.w, b.y + b.h) - (a.x, a.y) \rangle < 0] \text{ wenn } a.h \leq 0$$

oder

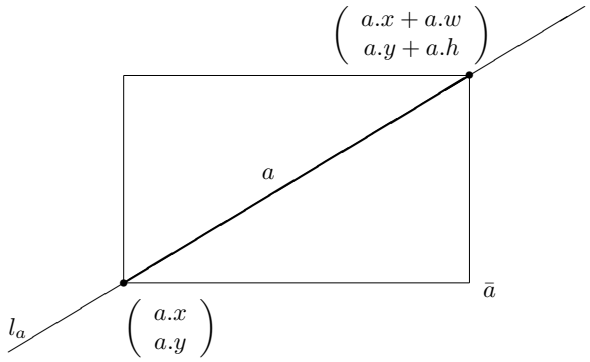


Abbildung 5.8: Punkte in Geradensegmenten

$$[\langle (a.h, -a.w), (b.x + b.w, b.y) - (a.x, a.y) \rangle > 0] \wedge$$
$$[\langle (a.h, -a.w), (b.x, b.y + b.h) - (a.x, a.y) \rangle < 0] \text{ wenn } a.h \geq 0$$

wobei mit $\langle ., . \rangle$ das Skalarprodukt bezeichnet wird. Die beiden Bedingungen stellen sicher, dass in beiden Halbebenen von l_a jeweils eine Ecke des Rechtecks b liegt. D.h.

das Geradensegment a cross Rechteck $b \Longleftrightarrow$
$\{[(a.h)(b.x) + (a.h)(b.w) - (a.h)(a.x) - (a.w)(b.y) - (a.w)(b.h) + (a.w)(a.y) > 0] \wedge [(a.h)(b.x) - (a.h)(a.x) - (a.w)(b.y) + (a.w)(a.y) > 0] \wedge$
$[a.x < b.x + b.w] \wedge [a.x + a.w > b.x] \wedge [a.y > b.y] \wedge$
$[a.y + a.h < b.y + b.h] \wedge [a.h \leq 0]\}$
$\vee$
$\{[(a.h)(b.x) + (a.h)(b.w) - (a.h)(a.x) - (a.w)(b.y) + (a.w)(a.y) > 0] \wedge$
$[(a.h)(b.x) - (a.h)(a.x) - (a.w)(b.y) - (a.w)(b.h) + (a.w)(a.y) > 0] \wedge$
$[a.x < b.x + b.w] \wedge [a.x + a.w > b.x] \wedge [a.y + a.h > b.y] \wedge$
$[a.y < b.y + b.h] \wedge [a.h \geq 0]\}$

5.5.3 Direction

Kong definiert mit "Direction" eine Relation um anzugeben, dass sich ein Objekt in einer bestimmten Richtung zu einem Zweiten befinden soll. Um die Direction-Relation zu beschreiben approximieren die Entwickler von SGGs Objekte als Punkte und wenden den kegelförmigen Ansatz von Peuquet u.a. [128] an. In Solverational können die von Kong u.a. vorgegebenen Direction-Relationen dann wie folgt beschrieben werden:

- Für $dir_4 : \mathcal{O} \times \mathcal{O} \longrightarrow D_4 := \{\mathrm{N}, \mathrm{E}, \mathrm{S}, \mathrm{W}\}$,
  setze

$\bar{x} = b.x + 0.5 * (b.w) - a.x - 0.5 * (a.w)$ und
$\bar{y} = b.y + 0.5 * (b.h) - a.y - 0.5 * (a.h)$.
Dann gilt:
$dir_4(a, b) = N \Longleftrightarrow (abs(\bar{y}) \geq \text{abs}(\bar{x})) \wedge (\bar{y} \leq 0)$,
$dir_4(a, b) = E \Longleftrightarrow (\text{abs}(\bar{y}) < \text{abs}(\bar{x})) \wedge (\bar{x} > 0)$,
$dir_4(a, b) = W \Longleftrightarrow (\text{abs}(\bar{y}) < \text{abs}(\bar{x})) \wedge (\bar{x} < 0)$ und
$dir_4(a, b) = S \Longleftrightarrow (\text{abs}(\bar{y}) \geq \text{abs}(\bar{x})) \wedge (\bar{y} > 0)$.

- Für $dir_8 : \mathcal{O} \times \mathcal{O} \longrightarrow D_8 := \{\text{N}, \text{NE}, \text{E}, \text{SE}, \text{S}, \text{SW}, \text{W}, \text{NW}\}$,
  setze
  $\bar{x} = b.x + 0.5 * (b.w) - a.x - 0.5 * (a.w)$ und
  $\bar{y} = b.y + 0.5 * (b.h) - a.y - 0.5 * (a.h)$.
  Dann gilt:
  $dir_8(a, b) = N \Longleftrightarrow (\bar{y} \leq -d_2 * \text{abs}(\bar{x}))$,
  $dir_8(a, b) = E \Longleftrightarrow (\bar{y} > -d_1 * \bar{x}) \wedge (\bar{y} \leq d_1 * \bar{x}) \wedge (\bar{x} > 0)$,
  $dir_8(a, b) = W \Longleftrightarrow (\bar{y} > d_1 * \bar{x})) \wedge (\bar{y} \leq -d_1 * \bar{x}) \wedge (\bar{x} < 0)$,
  $dir_8(a, b) = S \Longleftrightarrow (\bar{y} > d_2 * \text{abs}(\bar{x}))$,
  $dir_8(a, b) = NE \Longleftrightarrow (\bar{y} > -d_2 * \bar{x}) \wedge (\bar{y} \leq -d_1 * \bar{x})) \wedge (\bar{x} > 0)$,
  $dir_8(a, b) = NW \Longleftrightarrow (\bar{y} > d_2 * \bar{x}) \wedge (\bar{y} \leq d_1 * \bar{x}) \wedge (\bar{x} < 0)$,
  $dir_8(a, b) = SE \Longleftrightarrow (\bar{y} > -d_1 * \bar{x}) \wedge (\bar{y} \leq -d_2 * \bar{x}) \wedge (\bar{x} < 0)$ and
  $dir_8(a, b) = SW \Longleftrightarrow (\bar{y} > d_1 * \bar{x}) \wedge (\bar{y} \leq d_2 * \bar{x}) \wedge (\bar{x} > 0)$,
  wobei d_1 und d_2 numerische Approximationen von $\tan(\frac{\pi}{8})$ und $\tan(\frac{3\pi}{8})$ sind.

Das zweite Beispiel zeigt, wie beliebig viele komplexere kegelförmige Relationen beschrieben werden können, indem weitere numerische Approximationen von trigonometrischen Funktionen hinzugefügt werden. Umgangssprachlich formuliert kann der Entwickler durch die Anzahl der kegelförmigen Relationen selbst definieren, wie "genau" er die Richtung angeben will.

5.5.4 Distance

In SGG können Entfernungen zwischen Objekten in Form von Relationen festgelegt werden. Eine Entfernung ist eine positive reelle Zahl, die den Abstand zweier Objekte bezeichnet. Entfernungen können in Entfernungs-Granularitäten geordnet werden, die jeweils ein Intervall von Entfernungen bezeichnen. Dabei ist eine durch den Entwickler deklarierte Menge von Entfernungs-Granularitäten $Q = \{q_0, q_1, \ldots, q_n\}$ steigend und total geordnet, sodass sich in einer Entfernungs-Granularität immer nur Entfernungen "ähnlicher" Größenordnungen befinden können. Die zugehörigen Intervalle werden mit δ_i $(i = 0, \ldots, n)$ bezeichnet.

$(\forall \delta_i)(\exists l_i, u_i \in \mathbb{R} \cup \{\infty\} \mid 0 \leq l_i \leq u_i)$
$\quad \delta_i = (l_i, u_i) \vee \delta_i = [l_i, u_i) \vee \delta_i = (l_i, u_i] \vee \delta_i = [l_i, u_i]$.

Die Relation kann dann zwischen zwei Objekten a und b wie folgt definiert werden:

$$\begin{array}{ll}
(\forall q_i \mid \delta_i = (l_i, u_i))(\forall a, b \in \mathcal{O}) & aq_ib \Longleftrightarrow l_i < d(a,b) \wedge d(a,b) < u_i \\
(\forall q_i \mid \delta_i = [l_i, u_i))(\forall a, b \in \mathcal{O}) & aq_ib \Longleftrightarrow l_i \leq d(a,b) \wedge d(a,b) < u_i \\
(\forall q_i \mid \delta_i = (l_i, u_i])(\forall a, b \in \mathcal{O}) & aq_ib \Longleftrightarrow l_i < d(a,b) \wedge d(a,b) \leq u_i \\
(\forall q_i \mid \delta_i = [l_i, u_i])(\forall a, b \in \mathcal{O}) & aq_ib \Longleftrightarrow l_i \leq d(a,b) \wedge d(a,b) \leq u_i
\end{array}$$

wobei $d(a, b)$ für die Entfernung zwischen a und b steht und durch

$$d(a,b) = [(b.x + 0.5 * b.w - a.x - 0.5 * a.w)^2 + (b.y + 0.5 * b.h - a.y - 0.5 * a.h)^2]^{0.5}$$

definiert ist.

5.5.5 Alignment

Die mit "Alignment" bezeichneten Relationen modellieren, wie zwei Objekte zueinander angeordnet sein sollen, also z.B. ob zwischen ihnen ein Zwischenraum ("gap") sein soll, ob sie sich berühren ("meet"), usw. Die Beispiele in Abbildung 6 von Kong u.a. illustrieren die verschiedenen Relationen.

In diesem Abschnitt wird für jede der in Kong u.a. aufgeführten Relationen nur die horizontale Richtung betrachtet, da sich die vertikale durch Substitution von $.x$ und $.w$ durch $.y$ und $.h$ ergibt.

Gap: a gaps $b \Longleftrightarrow (a.x + a.w < b.x) \vee (a.x > b.x + b.w)$

Meet: a meets $b \Longleftrightarrow (a.x + a.w = b.x) \vee (a.x = b.x + b.w)$

Equal: a equals $b \Longleftrightarrow (a.x = b.x) \wedge (a.x + a.w = b.x + b.w)$

Interleaving: a interleaving $b \Longleftrightarrow$
$[(a.x < b.x) \wedge (a.x + a.w > b.x) \wedge (a.x + a.w < b.x + b.w)] \vee$
$[(a.x > b.x) \wedge (a.x > b.x + b.w) \wedge (a.x + a.w > b.x + b.w)]$

Middle: a middle $b \Longleftrightarrow (a.x > b.x) \wedge (a.x + a.w < b.x + b.w)$

Left (or Top): a und b sind "left aligned" $\Longleftrightarrow (a.x = b.x)$

Right (or Bottom): a und b sind "right aligned" $\Longleftrightarrow (a.x + a.w = b.x + b.w)$

5.5.5.1 Spatial Granularity and Spatial Hierarchy

Spatial Granularity: In SGGs ist es möglich, verschiedene Abstufungen ("Granularität") für "Direction"- und "Distance"-Relationen anzugeben. Diese Relationen können auch in Solverational leicht realisiert werden, indem mehr Richtungen – also kleinere kegelförmige Strukturen – oder mehr Intervalle zur Unterscheidung weiterer Entfernungen definiert werden. Die Abstufungen der Relationen "Direction" und "Distance" wurden bereits in Abschnitten 5.5.3 und 5.5.4 beschrieben.

Spatial Hierarchy: Die Relation $\bar{R}$ ist eine Abstufung einer Relation R genau dann, wenn für alle a und b, die die Relation R in Relation zueinander setzt, $\bar{R}$ auch a und b in Relation zueinander setzt. Ergibt sich kein Widerspruch durch das Hinzufügen weiterer Constraints, kann $\bar{R}$ aus beliebig vielen verschiedenen Relationen R zusammengesetzt werden, indem die Constraints von R auch für $\bar{R}$ verwendet werden. Daraus folgt, dass die Constraints von R eine Untermenge von $\bar{R}$ sind.

$$(\text{Constraints von } \bar{R}) \equiv [(\text{Constraints von } R) \wedge \ldots]$$

Um "Spatial Hierarchy" herzustellen, können Entwickler daher beliebige Relationen R zu neuen Relationen $\bar{R}$ kombinieren, indem die Constraints aus R einfach für $\bar{R}$ verwendet werden.

5.5.6 Beispiel zur Nutzung der Constraints

Ein Beispiel, wie die hier entwickelten Constraints in Solverational genutzt werden, werde für die Relation "gap" gegeben:

```
transformation t(a:inputMM, c:outputMM) {
    relation relationGap {
        domain a Node {
            gap = b:Node {}
        };
        alternative domain c NewNode {
            c.x < c.w + d.x,
            gap = d:NewNode {}
        };
        alternative domain c NewNode {
            c.x > d.x + d.w,
            gap = d:NewNode {}
        };
    }
}
```

5.5.7 Interpretation

In diesem Abschnitt wurden für alle Relationen aus [88] entsprechende Constraints angegeben. Ein Beispiel zeigt, wie die Constraints in Solverational übersetzt werden können. Dies impliziert:

1. Die Ausdrucksmächtigkeit von Constraints, die in Solverational verwendet werden können, ist bzgl. der räumlichen Relationen mindestens genauso groß wie die von SGGs.

2. Umgekehrt können mit Solverational aber zusätzlich Constraints und Zielfunktionen notiert werden, was mit SGGs nicht möglich ist.

Zum besseren Vergleich werden hier bereits bekannte, charakteristische Unterschiede von SGGs und Solverational nochmals aufgeführt und damit in die Interpretation aufgenommen:

SGGs erlauben die Suche von Teilgraphen in speziellen Graphen in polynomieller Laufzeit. Dies ist sehr schnell, da das Problem der Suche von Teilgraphen in Graphen im Allgemeinen als NP-vollständig bekannt ist. Für die Laufzeit von Transformation ist dies von Vorteil, allerdings funktioniert dies nur für sog. "selection-free"-Graphen. Nach den Entwicklern von SGGs [88] ist unbekannt, wie viele Graphen tatsächlich in diese Klasse von Graphen einzuordnen sind. Sogar so einfache Graphen, wie voll vermaschte Graphen können mit dem Algorithmus zur Transformation von SGGs nicht transformiert werden. Für alle Graphen, die nicht die "selection-free"-Bedingung erfüllen, ist die Laufzeit von SGGs unbekannt und ein Algorithmus zur Interpretation solcher Graphen, muss noch entwickelt werden (und liegt damit vermutlich wieder in NP).

Mit Solverational können Zielfunktionen verwendet werden. Dies ist für die Deklaration von Transformationen für Modelle von Benutzerschnittstellen von Vorteil, s. Abschnitt 3.2.

Es wurde nicht untersucht, inwieweit sich die beiden unterliegenden Transformationssprachen (RGG und QVT Relations) in ihrer Ausdrucksmächtigkeit unterscheiden. Allerdings ist der derzeit bekannte Algorithmus zur Interpretation von SGGs auf die Transformation von Graphen beschränkt, die der "Selection-free"-Bedingung genügen. Dies ist eine starke Einschränkung (z.B. können im Gegensatz zu QVT Relations keine "gitterähnlich" vermaschten Graphen transformiert werden). Es steht daher zu vermuten, dass zumindest unter Anwendung dieses Algorithmus SGGs weniger Graphen transformieren können als es mit QVT Relations möglich ist.

Im Vergleich der beiden Sprachen lässt sich leicht feststellen, dass SGGs auf die Transformation von speziellen Graphen und speziellen Relationen optimiert wurde. Mit Solverational können alle Relationen implementiert und zusätzlich weitere Constraints und Zielfunktionen definiert werden.

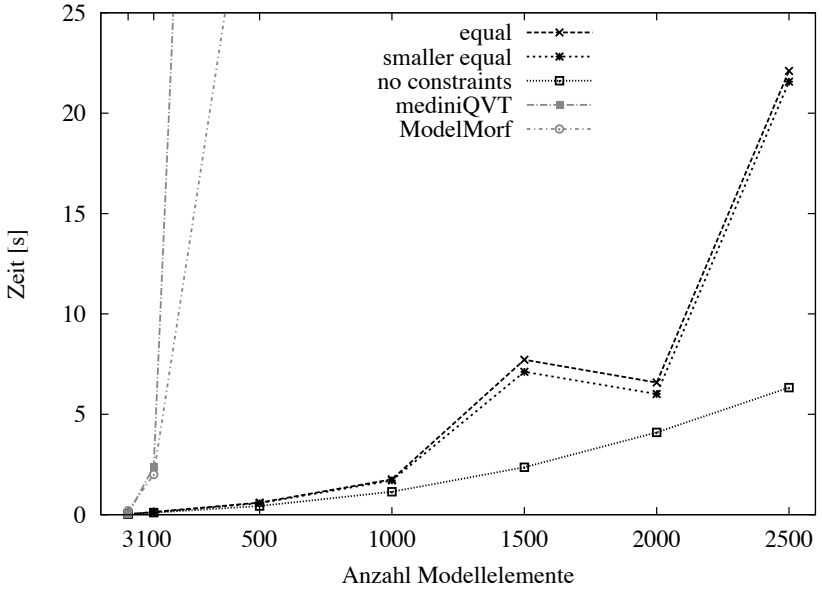


Abbildung 5.9: Laufzeiten von Transformationsmaschinen und Constraint-Solving.

5.6 Effizienz des Constraint-Solving Ansatzes

Constraint-Solving ist ein Prozess, der häufig als sehr aufwändig beschrieben wird, insbesondere im Hinblick auf die Rechenzeit, die zum Auswerten von Constraint-Solving-Problemen benötigt wird. Dieser Verdacht ist berechtigt, wenn komplexe Constraints-Solving-Probleme gelöst werden sollen. Da oftmals angenommen wird, dass Modelltransformationen komplexe Prozesse sind, wurde anhand eines einfachen Beispiels überprüft, ob Constraint-Solving in Modelltransformationen *immer* als "langsam" zu bezeichnen ist. Abbildung 5.9 stellt dar, wie sich implementierte Transformationsmaschinen für unterschiedlich viele Modellelemente im Ausgangsmodell verhalten. Auf der x-Achse ist die Anzahl der transformierten Modellelemente aufgetragen, auf der y-Achse die Laufzeit der Transformationsmaschinen (ohne Laden und Speichern der Modelle, also die reine Laufzeit der Transformation). Zum Vergleichen wurde ein mit Windows ausgestatteter Computer verwendet (CPU mit 1,8GHz und 1 GB RAM, keiner der Ansätze benötigte den gesamten Hauptspeicher, sodass kein Swapping benötigt wurde). Es sind fünf verschiedene

Graphen dargestellt. Alle Transformationen, die in diesem Abschnitt verwendet wurden finden sich in Abschnitt 4.5.5.

5.6.1 Datenanalyse und Präsentation

Graph "no constraints" stellt die einfachste der in Abschnitt 4.5.5 beschriebenen Transformationen dar. Diese kann direkt in drei verschiedenen Transformationsmaschinen ausgeführt werden: der medini QVT-Transformationsmaschine [73], der ModelMorf-Transformationsmaschine [96] und der Solverational-Transformationsmaschine aus dieser Arbeit. Die Abbildung suggeriert, dass die Solverational-Transformationsmaschine die Transformation vor allem für große Modelle deutlich schneller ausführen kann. Bereits ab 500 Modellelementen benötigen die Transformationsmaschinen medini QVT und ModelMorf schon mehr als 25 Sekunden um zu terminieren, und die Zeiten können daher nicht mehr bei der gewählten Skala berücksichtigt werden. Mit medini QVT war es nicht möglich Modelle mit mehr als 500 Modellelementen in akzeptabler Zeit zu transformieren. Die Tests mit medini QVT mussten daher bei 1000 Modellelementen abgebrochen werden. Mit ModelMorf konnten die Tests zwar durchgeführt werden, allerdings brauchte ModelMorf knapp 15 Minuten für 2500 Modellelemente – deutlich länger als der Solverational-Interpreter, der gut 6 Sekunden für die gleiche Transformation benötigte.

Graph "equal" nutzt Eigenschaften von Constraint-Programming und kann deshalb weder in der medini QVT noch in der ModelMorf-Transformationsmaschine ausgeführt werden. Die Constraints erlauben allerdings trotzdem nur eine Lösung. Sie erzwingen allerdings das Setzen von Werten von Attributen aus einer mathematischen Beziehung heraus (s. Abschnitt 4.5.5). Dieser Graph zeigt, dass das Ausführen von Modelltransformationen mit Constraints im vorliegenden Fall langsamer ist als ohne Constraints, aber im Verhältnis zum schlechten Abschneiden von ModelMorf und medini QVT auch nicht so sehr viel länger dauert.

Graph "smaller equal" erlaubt mehrere Lösungen für die Constraint-relationale-Transformation, indem auch kleinere Werte erlaubt sind als vorgegeben (s. Abschnitt 4.5.5). Man erkennt, dass die Transformation nahezu gleich schnell ausgeführt wird, wie die "equal" Transformation.

5.6.2 Interpretation

Auf den ersten Blick erscheint überraschenderweise die Solverational-Transformationsmaschine deutlich effizienter zu sein als ihre beiden Kontrahenten. Die Transformationen nicht direkt zu interpretieren, sondern erst in eine andere Sprache zu übersetzen (Transformation Compiler), ist ähnlich zum Vergleich von

interpretierten Sprachen gegenüber kompilierten Sprachen. Zusätzlich implementiert die Solverational-Transformationsmaschine nicht den kompletten Umfang von Funktionen von QVT Relations. Insbesondere die Funktion "check then enforce" wurde nicht implementiert, die bei jedem Einfügen eines neuen Elements prüft, dass nicht ein weiteres Element vorhanden ist, dessen Attribute die gleichen Werte haben. Diese Funktion scheint sehr zeitaufwändig, da bei jedem Einfügen das komplette Modell auf Existenz des Modellelements geprüft werden muss. Dabei handelt es sich um eine Suche nach einem Teilgraphen, deren Algorithmus in NP liegt. Trotzdem stellt sich die Frage, warum die vorliegenden, kommerziellen Transformationsmaschinen (medini QVT und ModelMorf) exponentielle Laufzeiten haben – für den praktischen Gebrauch sind lange Wartezeiten wenig komfortabel.

Dieses Beispiel stellt daher heraus, dass Constraint-Solving nicht *immer* der entscheidende Faktor für die Laufzeit von Modelltransformationen ist. Insbesondere die Verwendung von Constraints, die die sog. "Constraint Propagation" unterstützen, sind sehr effizient ausführbar. Hierbei erkennt der Constraint-Solver, welche Constraints von welchen anderen Constraints abhängen. Ein einfaches Beispiel ist $5 > A + B, B > 15$, sodass der Costraint-Solver $A < -10$ inferieren kann, ohne zu probieren. Auch im vorliegenden Beispiel kam Constraint Propagation zum Einsatz, sodass der Constraint-Solver sehr schnell terminierte. Weiterhin spielt die Wahl des Solvers eine entscheidende Rolle. Werden nur bestimmte Constraints verwendet, können spezialisierte Solver verwendet werden (im linearen Fall z.B. Solver, die auf dem in der Praxis sehr schnellen Simplex-Algorithmus basieren) die wesentlich schneller Ergebnisse liefern als der vorliegende allgemeine, nicht-lineare Solver der ECLiPSe Umgebung. ECLiPSe bietet dazu eine einfache Schnittstelle an, die Solver können daher schon in ECLiPSe ersetzt werden. Muss der Solver eine Lösung suchen, so ist die Reihenfolge, mit der Lösungen probiert werden, oftmals sehr wichtig. Es sollte also, wenn der Solver zu viel Rechenzeit benötigt, getestet werden, ob eine Lösung schneller gefunden werden kann, wenn der Solver Lösungen in einer anderen Reihenfolge probiert. Die Reihenfolge kann im generierten ECLiPSe Code eingestellt werden.

In einigen Fällen wird der Constraint-Solving-Prozess jedoch wesentlich mehr Zeit benötigen als in den vorliegenden Transformationen. Beispielsweise ist folgendes Constraint für gegebenes "A","B" und "C" eine Implementierung des diskreten Logarithmus:

```
A = B ^ C mod D
```

Verschiedene Verfahren aus der asymmetrischen Kryptographie basieren auf der Schwierigkeit, den diskreten Logarithmus zu ziehen (z.B. Diffie-Hellman, ElGamal oder Schnorr). Daher dauert eine Berechnung des obigen Constraints nach heutigen Wissenstand für große Zahlen sehr lange – bei "schwierigen" Constraints ist der

Prozess des Constraint-Solving daher sehr wohl ein entscheidender Faktor für die Laufzeit einer Transformation.

5.6.3 Fazit

Die vorliegende Implementierung verhält sich für die untersuchten Beispiele bereits ohne Constraint-Solving viel effizienter als die beiden Konkurrenten. Sie implementiert allerdings einige sehr rechenintensive Funktionen der Konkurrenten nicht und kann deshalb an sich nicht mit diesen verglichen werden. Trotzdem ist sie auch mit Constraint-Solving in den angegebenen Beispielen wesentlich effizienter. Es lässt sich also schlussfolgern, dass Constraint-Solving nicht pauschal als der langsamste Prozess in der Transformation bezeichnet werden kann. Vielmehr bietet sich selbst bei Betrachtung der Laufzeit eine Integration an, da andere Prozesse sehr viel mehr Zeit in Anspruch nehmen. Im vorliegenden Fall liegt die Vermutung nahe, dass die Funktion "check then enforce" wesentlich zum schlechten Ergebnis der Konkurrenten beiträgt. Es kann also oftmals ohne große Veränderung der Laufzeit Constraint-Solving in Transformationsmaschinen eingefügt werden.

Zusätzlich wurde "check then enforce" für die reine Generierung von Modellen von Benutzerschnittstellen im Rahmen dieser Arbeit gar nicht benötigt. Unter der Annahme, dass die Beispiele einen gewissen repräsentativen Charakter für die Generierung von Modellen von Benutzerschnittstellen haben, kann diese Funktion für die Generierung von Benutzerschnittstellen einfach weggelassen und so die Effizienz der Interpreter gesteigert werden.

Trotzdem sollte auf die Art der Constraints Rücksicht genommen werden. Constraint-Programmierung ist deklarativ und selbst aufwändige Prozesse wie das Suchen eines diskreten Logarithmus lassen sich sehr kompakt darstellen. Diese Prozesse brauchen aber sehr viel Rechenzeit und sollten daher nicht in Constraint-relationalen-Transformationen verwendet werden.

5.7 Abdeckung von Regeln aus dem Apple Style Guide

Constraint-Solving und Zielfunktionen werden zur Generierung von Benutzerschnittstellen eingesetzt. Auf Grund der Möglichkeit, Constraints und Zielfunktionen in Transformationen zu integrieren, scheint Solverational daher für die Generierung von Modellen von Benutzerschnittstellen gut geeignet. Es bleibt die Frage, wie gut Regeln aus der Praxis in Solverational implementiert werden können. Dazu bietet dieser Abschnitt eine subjektive Einschätzung.

Wie in Abschnitt 3.2.1 beschrieben, werden viele Regeln für das Design und Layout von Benutzerschnittstellen in sogenannten Style Guides beschrieben. Um

eine Abschätzung für die Anzahl von Regeln aus Style Guides zu erhalten, die mit der Transformationssprache implementiert werden können, wurde exemplarisch ein Style Guide ausgewählt (Apple Human Interface Guidelines [10]).

Im Rahmen dieser Arbeit wurde untersucht, wie viele der Regeln mit Solverational implementiert werden können. Hierbei werden verschiedene Modelle vorausgesetzt, sodass Ausgangs- und Zielmodelle auch tatsächlich die in den Regeln gewünschten Eigenschaften modellieren können. Um die Benutzerschnittstellen zur Anzeige zu bringen wird angenommen, dass nach der Transformation ein Interpreter die erzeugten Modelle interpretiert (z.B. der Mapache-Modell-Interpreter [18] bzw. [124]). Alternativ könnte eine Transformation die Modelle in Quellcode überführen.

Die Untersuchung der Regeln fand aus zeitlichen Gründen auf einer informellen Ebene statt, d.h. die Regeln wurden nicht implementiert. Vielmehr schätzten zwei Experten die Möglichkeit ein, ob und wie sich die im Style Guide gegebenen Regeln mit Solverational implementieren ließen. Die gesamte Liste der untersuchten Regeln findet sich auf Grund der Länge in [127], dort werden auch Hinweise auf eine mögliche Implementierung der jeweiligen Regel gegeben. Es wurden aus jedem Abschnitt der Apple Human Interface Guidelines die ersten sieben Regeln untersucht, sofern der Abschnitt mindestens sieben Regeln umfasste.

Option	Code
nicht anwendbar	0
kann mit Solverational implementiert werden	1a
erfordert Annahmen aber kann mit Solverational implementiert werden	1b
kann implementiert werden, nur nicht mit Solverational	1c
kann auch mit einer klassischen Transformationssprache implementiert werden	2a
erfordert Annahmen, aber kann auch mit einer klassischen Transformationssprache implementiert werden	2b
kann nicht mit Solverational oder einer klassischen Transformationssprache implementiert werden	2c

Tabelle 5.10: Codes die in der Auswertung verwendet werden

Tabelle 5.11 zeigt das Ergebnis, während in Tabelle 5.10 die möglichen Codes aufgelistet sind, um die Auswertung zu codieren. In Tabelle 5.11 ist die Häufigkeit aufgeführt, mit der die Codes in der Liste Nr. 3 in [127] vorkommen.

Bei verschiedenen Regeln wurden Annahmen getroffen, die die Implementierung erleichtern, insbesondere über die zur Transformation zur Verfügung stehenden Modelle und Meta-Modelle. Eine komplette Implementierung aller Regeln

Code	Summe	%, incl. 0	%, excl. 0
0	34	24.46%	
1a	10	7.19%	9.52%
1b	4	2.88%	3.81%
1c	3	2.16%	2.86%
2a	39	28.06%	37.14%
2b	24	17.27%	22.86%
2c	25	17.99%	23.81%

Tabelle 5.11: Ergebnisse der Auswertung von Regeln aus [10]

setzt die Implementierung der entsprechenden Interpreter (oder Modell-zu-Code Transformationen), Meta-Modelle und Modelle voraus. Tabelle 3 in [127] gibt dazu Anhaltspunkte, welche Modellelemente, Interpreter und Transformation zur Implementierung einer bestimmten Regel benötigt werden. Dabei lassen sich die Modelle wie folgt zusammenfassen: Task-Modell, abstraktes Modell der Benutzerschnittstellen, konkretes Modell der Benutzerschnittstellen. Zusätzlich wurde die Existenz folgender Modelle vorausgesetzt: Konfigurationen-Modell, ein Modell benötigter Dateiformate, eines zur Beschreibung verwendeter und herstellbarer Inhalte oder Artefakte, ein Plattform-Modell mit standardmäßig verwendeten Pfaden als Modellelemente, ein Workflow-Modell, das die Arbeitsgänge des Nutzers beschreibt, ein Modell verwendeter Cursor (oftmals Teil des konkreten Modells der Benutzerschnittstelle) und ein Tasten-Modell (mit Tasten-Kurzbefehlen, oftmals Teil des konkreten Modells der Benutzerschnittstelle).

5.7.1 Bedeutendste Faktoren, die die Validität einschränken können

Die Objektivität dieser Untersuchung ist durch die geringe Anzahl von Experten beschränkt, da die Implementierbarkeit der Regeln nur von zwei Experten untersucht wurde, wobei ein Experte der Verfasser dieser Arbeit ist. Da es mehrere Wochen dauerte die Regeln auszuwerten, ist eine Wiederholung mit anderen Experten bei gegebenen Ressourcen nicht durchführbar. Unter den Experten herrschte allerdings Konsens über das Ergebnis der meisten der untersuchten Regeln. Die Regeln wurden nicht implementiert, sondern durch die Experten entschieden, ob die jeweiligen Regeln in Solverational implementiert werden können. Dadurch sind die Ergebnisse eher subjektiv.

Auf Grund der geringen Objektivität leidet auch die Reliabilität, da nur die zwei Experten befragt werden können, um die gleichen Ergebnisse zu erhalten.

Zusätzlich liegen die Regeln informell vor und es besteht Spielraum für verschiedene Interpretationen. Die Ergebnisse werden aber zumindest ähnlich ausfallen, da die Experten die Regeln vermutlich ähnlich interpretieren würden.

5.7.2 Interpretation

Es zeigt sich, dass mehr als 50% (55,4%) der Regeln mit Solverational formalisiert werden können. Von den 139 untersuchten Regeln können damit 78 mit Solverational implementiert werden. Zehn Regeln können sogar ausschließlich mit Solverational implementiert werden, da sie nur mit einer Modell-zu-Modell Transformationssprache implementiert werden können, die Constraints und Constraint-Solving unterstützt (vier weitere unter Annahmen). Diese Regeln betreffen vor allem das Layout von Benutzerschnittstellen. Es ist nicht weiter verwunderlich, dass sich gerade solche Regeln in Solverational implementieren lassen, da Solverational vor allem für diesen Zweck Constraint-Solving unterstützt.

Mit klassischen Transformationssprachen können im Gegensatz dazu 47,5% der Regeln umgesetzt werden. Rein nach Zahlen ist Solverational damit um 8% besser geeignet, den Regelsatz zu implementieren.

Es zeigt sich auch, dass ein guter Teil der Regeln leider nicht umgesetzt werden kann. Andererseits handelt es sich dabei zu großen Teilen (34 Regeln, Typ 0) um Aufgaben, die Menschen durchführen müssen[3] oder die bereits von IDEs und anderen Werkzeugen unterstützt werden[4]. Insofern wäre eine Umsetzung in einer Transformation auch nicht möglich oder doppelter Aufwand zur Implementierung. Werden diese Regeln außer Betracht gelassen, also nur relevante Regeln betrachtet, können sogar fast drei Viertel der Regeln (73,33%) mit Solverational formalisiert werden.

Ein weiterer Teil von Regeln lässt sich gar nicht in Transformationssprachen abbilden, obwohl diese prinzipiell dafür geeignet wären. Dies betrifft z.B. Regeln, die Auswertungen zur Laufzeit benötigen[5].

Einige Regeln (drei, 2,16%) können speziell nicht mit Solverational umgesetzt werden, da Solverational keine Operationen auf Zeichenketten unterstützt, die Regeln diese aber erforderten. Solverational könnte aber um diese Funktionalität leicht erweitert werden.

---

[3] z.B. Regel 65: "Your product development team should include a skilled writer who is responsible for reviewing all user-visible onscreen text as well the instructional documentation. The writer should refer to the Apple Publications Style Guide for guidance on Apple-specic terminology."

[4] z.B. Regel 31: "Include all required resources inside your application bundle."

[5] z.B. Regel 29: "Named resources that might potentially be accessable to an application from multiple user sessions should incorporate the session ID into the name of the resource."

5.7.3 Fazit

Es können mehr als 50% (50,4%) der Regeln des Apple Human Interface Guideline [10] Style Guides mit Solverational implementiert werden. Betrachtet man nur die relevanten Regeln sind es sogar 73,33%. Damit scheint Solverational als Basis zur Implementierung von Regeln aus Style Guides prinzipiell geeignet. Mit klassischen Transformationssprachen können stattdessen nur 8% weniger Regeln implementiert werden.

Die vorliegende Untersuchung umfasst nur einen einzigen Style Guide als Beispiel, der von zwei Experten analysiert wurde. Die Ergebnisse sind daher nur bedingt allgemein gültig.

5.8 Ergebnisse und Entscheidungskriterium

In diesem Abschnitt werden die Ergebnisse der einzelnen Untersuchungen im Rahmen der Methodik aus Abschnitt 5.1 interpretiert.

5.8.1 Ausgereift

Ergebnisse für die drei Sprachen

Für alle drei Ansätze existieren Implementierungen, d.h. alle sind implementierbar, womit dieses Kriterium erfüllt ist. Die Implementierungen von SUPPLE und SSGs werden in [47] und [88] vorgestellt. Die Implementierung einer Transformationsmaschine für Solverational findet sich in Abschnitt 4.3. Allerdings handelt es sich bei allen dreien um Prototypen. SUPPLE besitzt in gewissem Sinne eine ausgereifte Sprache, wenn nur JAVA betrachtet wird.

Da SUPPLE als Erweiterung der Sprache JAVA verstanden wird, existieren für SUPPLE keine zweideutigen Sprachkonstrukte, da JAVA keine zweideutigen Sprachkonstrukte zulässt.

Solverational setzt auf der OCL (s. [116]) auf. Für die OCL ist bekannt, in einigen extremen Fällen sogar nicht entscheidbar zu sein (s. [25]). Deshalb gilt dies auch für Solverational. Allerdings werden die Sprachkonstrukte in der Praxis nur sehr selten verwendet.

SGGs sind strukturell sehr einfach aufgebaut und besitzen keine mehrdeutigen Sprachkonstrukte.

Für SGGs und JAVA/SUPPLE ist bekannt, dass relativ effiziente Compiler und Interpreter vorliegen. Im Rahmen von Abschnitt 5.6 wurde untersucht, inwiefern einfache Constraints die Geschwindigkeit von Transformationen in Solverational beeinflussen. Es wurde festgestellt, dass andere Eigenschaften von QVT Relations,

der Sprache von der Solverational abgeleitet wurde, die Effizienz des Solverational-Interpreters wesentlich stärker beeinflussen können. Es wird daher – für einfache Constraints – von ähnlich effizienten Interpretern ausgegangen, auch wenn für manche Constraints SGGs und SUPPLE – auf Grund der hohen Spezialisierung auf eine Teilmenge von Constraints – effizientere Interpreter anbieten.

Interpretation

Alle Sprachen sind maschinell interpretierbar und damit implementierbar. SUPPLE besitzt – unter Berücksichtigung der starken Verflechtung mit der Sprache JAVA – "ausgereiftere" Sprachkonstrukte.

Auf Grund der Nicht-Entscheidbarkeit von OCL ist Solverational in einigen Fällen nicht entscheidbar. In der Praxis kommen diese Fälle allerdings sehr selten vor. Im Kriterium "ausgereift" besitzt Solverational gegenüber den beiden anderen Ansätzen also zumindest einen theoretisch vorhandenen Nachteil.

5.8.1.1 Entscheidungskriterium

Um ein einfaches Kriterium für die Integration in das komplexe Entscheidungskriterium zu erhalten, werden die Ergebnisse aus diesem Kapitel mit willkürlich (subjektiv) gewählten Zahlen zwischen 0 (schlecht) und 1 (gut) belegt:

- Für die Implementierbarkeit erhalten alle Sprachen eine 1.
- Weil es sich bei den Interpretern von SGGs und Solverational nur um Prototypen handelt, erhalten beide eine 0,9. SUPPLE nutzt den JAVA-Interpreter und erhält eine 1.
- Wegen der in der Theorie existierenden, zweideutigen Sprachkonstrukte erhält die Sprache Solverational eine 0,9. SUPPLE hat keine zweideutigen Sprachkonstrukte und für SGGs sind keine bekannt, somit erhalten beide eine 1.

Um das Gesamtergebnis zu berechnen werden die drei Konstanten multipliziert. Damit gilt insgesamt für die drei Sprachen: $T_{SUPPLE} = 1$ für die Sprache SUPPLE, $T_{SGG} = 0,9$ für die Sprache SGG und $T_{Solverational} = 0,9 * 0,9 = 0,81$ für die Sprache Solverational. Alle berechneten Konstanten finden sich in Tabelle 5.12.

Damit sei das Kriterium "ausgereift", abgekürzt durch A, mit einem Gewicht $w_{ausgereift}$ wie folgt definiert:

$$A = w_{ausgereift} * T_{Sprache} \tag{5.1}$$

5.8.2 Benutzbar

Nach der eingeführten Methodik besteht das Kriterium "Benutzbar" aus Untersuchungen zur Verständlichkeit und Schreibbarkeit.

5.8.2.1 Verständlichkeit

Um ein Maß für die Verständlichkeit zu erhalten, wurde eine Studie durchgeführt, um die Verständlichkeit an Probanden zu messen. Eine ausführliche Darstellung der Studie findet sich in Abschnitt 5.2.

Es wurden drei Bereiche der drei Sprachen untersucht: Modelle, Constraints und Optimierung. Die Studie wurde als quantitative Studie durchgeführt, basierend auf einem speziellen Fragebogen. Die Frage "What does the following source code do?" wurde zu einem jeweils in sich geschlossenen Ausschnitt eines Quellcodes einer Transformation in einer bestimmten Sprache gestellt. Die Probanden mussten anhand einer Auswahl möglicher Antworten entscheiden, was dieser Teil der Transformation bewirkt.

Als Studienteilnehmer wurden Mitarbeiter der Fachgruppe Telekooperation an der TU-Darmstadt ausgewählt. Es handelte sich um einen Online-Fragebogen bei dem die Teilnehmer bei der Beantwortung der Fragen per Telefon geführt oder direkt beobachtet wurden. Insgesamt nahmen 19 Probanden an der Studie teil, die in drei Gruppen von Studienteilnehmern eingeteilt wurden, die jeweils die Aufgaben in einer Sprache (also entweder Solverational (6 Teilnehmer), SGGs (7 Teilnehmer) oder SUPPLE (6 Teilnehmer)) durchführten.

Als Operationalisierung wurde die Zeit zum Verstehen gewählt. Eine Frage wurde demnach schneller verstanden, wenn sie schneller beantwortet wurde.

In jedem Bereich wurden den Probanden jeweils drei Fragen gestellt (insgesamt also 9 für Solverational und SUPPLE und 6 für SGGs), zu der es jeweils vier Antworten gab:

1. Modelle: Den Studienteilnehmern wurden mehrere Modelle präsentiert. Sie sollten entscheiden, wie die Benutzerschnittstelle zu diesem Modell aussah. Die Sprachen, in denen die Modelle notiert wurden, waren entweder grafisch (SGG und Solverational) oder JAVA im Fall von SUPPLE.

2. Constraints: In allen drei Sprachen wurden die gleichen Constraints implementiert, die den Probanden in einer Sprache präsentiert wurden. Die Probanden sollten entscheiden, was genau die Constraints beschränken.

3. Optimierung: Es wurde untersucht, wie gut Optimierung von den Studienteilnehmern verstanden wurde. Dazu wurden den Studienteilnehmern unterschiedliche Zielfunktionen präsentiert. SGGs implementieren keine Unterstützung für Optimierung und wurden deshalb nicht betrachtet.

Alle Fragen und Antworten hatten - bis auf den jeweils angezeigten Quellcode - in jeder Sprache den gleichen Wortlaut. Auch das Modell, das Constraint oder die Zielfunktion hatten jeweils die gleiche Funktion, nur wurden diese in unterschiedlichen Sprachen dargestellt. Entsprechend waren die richtigen Antworten identisch. Dadurch konnte sichergestellt werden, dass zeitliche Unterschiede nur im Lesen und Verstehen des Quellcodes entstehen konnten.

Im **Bereich Modelle** wurden die grafischen Modelle von Benutzerschnittstellen (die Modelle in Solverational und SGG repräsentierten) mit den textuell notierten von SUPPLE verglichen. Der Mittelwert über den genommenen Zeiten für grafische Modelle war niedriger als der für SUPPLE (grafische Modelle: 52s, SUPPLE: 136s). Ein T-Test zwischen den beiden Populationen ergab hohe statistische Signifikanz (p-Wert $= 0,0006$).

Im **Bereich Constraints** lagen Daten aus allen drei Populationen vor. anhand der Mittelwerte lässt sich erkennen, dass Probanden die Constraints, die in Solverational notiert waren, am schnellsten verstehen konnten – gefolgt von SGG und SUPPLE (Solverational: 56s, SGG: 96s, SUPPLE:174s). ANOVA wies nach dass es mindestens zwei unterschiedliche Populationen gab (p-Wert $= 1,1 * 10^{-4}$) und mittels T-Test (mit Bonferroni-Korrektur) wurde geprüft, dass es sich um drei Populationen handelte (p-Werte: $0,004$; $0,0009$; $0,042$).

Im **Bereich Optimierung** konnten nur Quelltexte verglichen werden, die in Solverational und SUPPLE notiert wurden, da SGG keine Unterstützung für Optimierung anbot. Solverational konnte im Mittel schneller verstanden werden (Solverational: 56s, SUPPLE: 118s). Ein T-Test wies statistische Signifikanz nach (p-Wert $= 0,0002$).

5.8.2.2 Schreibbarkeit

Aufbauend auf den Fragen der Studie zur Verständlichkeit aus Abschnitt 5.2 wurde in Abschnitt 5.3 untersucht, wie viel Code für die verschiedenen Fragen implementiert werden musste. Dies stellt eine sehr grobe Approximation dar, die einen Anhaltspunkt über die Schreibbarkeit der Sprachen geben soll und nicht den Anspruch auf wissenschaftliche Perfektion erhebt. Um die Menge Code zu messen wurde die Anzahl relevanter terminaler Symbole gezählt, die zum Notieren minimal benötigt wurde.

Analog zum Vorgehen zur Studie zur Verständlichkeit waren für Schreibbarkeit drei Bereiche der Sprachen zu untersuchen:

1. Bereich Modelle: Benötigen grafische oder in SUPPLE textuell notierte Modelle von Benutzerschnittstellen weniger terminale Symbole?

2. Bereich Constraints: Welche Notation für die Constraints lässt sich mit den wenigsten terminalen Symbolen notieren?

3. Bereich Optimierung: Benötigt die Notation von Zielfunktionen in Solverational oder SUPPLE weniger terminale Symbole?

Für die Durchführung des Vergleichs standen in jedem Bereich drei Fragen zur Verfügung. In jeder Sprache wurden diese drei Fragen möglichst einfach und kurz implementiert. Zum Vergleichen wurden nur die terminalen Symbole gezählt, die tatsächlich zur Implementierung benötigt wurden und nicht durch nicht-terminale Symbole implizit vorgegeben waren (d.h. z.B. bei Funktionsaufrufen wurden "(" und ")" nicht gezählt).

Alle Fragen und Antworten hatten – bis auf den jeweils angezeigten Quellcode – in jeder Sprache den gleichen Wortlaut. Auch das Modell, das Constraint oder die Zielfunktion hatten jeweils die gleiche Funktion, nur wurden diese in den drei unterschiedlichen Sprachen dargestellt. Dadurch war Vergleichbarkeit der verschiedenen Quellcodes hergestellt.

Bereich Modelle: Die Anzahl terminaler Symbole, die für das Notieren von grafischen Modellen benötigt wird, ist erheblich geringer als für textuell notierte in SUPPLE (Mittelwerte waren: 9 Symbole für grafische Modelle und $33,67$ Symbole für SUPPLE).

Im **Bereich Constraints** waren die Mittelwerte über der Anzahl terminaler Symbole $16,67$ Symbole für SGGs, $17,33$ Symbole für Solverational und 18 Symbole für SUPPLE. Die Mittelwerte haben daher sehr geringe Differenzen und es konnte kein signifikanter Unterschied beobachtet werden.

Der Vergleich der Anzahl terminaler Symbole, die für die Notation von **Zielfunktionen** benötigt wurden, zeigt, dass für SUPPLE (Mittelwert: $7,67$) etwas weniger Symbole benötigt wurden als für Solverational (Mittelwert: $9,33$). Dieses Ergebnis ist nicht realistisch, da Schleifen und Konstanten in SUPPLE nicht berücksichtigt werden konnten.

Dies zeigt auch, dass die Verallgemeinerung der Ergebnisse problematisch ist, da zu wenige Beispiele untersucht wurden. Da nicht mehr Datenpunkte vorlagen, werden die Ergebnisse trotzdem im Folgenden verwendet.

5.8.2.3 Entscheidungskriterium

Die Integration von "benutzbar", abgekürzt durch B, in das komplexe Entscheidungskriterium muss die beiden Teilkriterien "Verständlichkeit" (kurz V) und "Schreibbarkeit" (kurz S) berücksichtigen. Beide Teilkriterien umfassen jeweils die Bereiche "Modelle" (kurz M), "Constraints" (kurz C) und "Zielfunktionen" (kurz Z). Es werden sechs Gewichte benötigt, die linear in B eingehen:

$$\begin{aligned} B = {} & w_M^V * M_{Sprache}^V + w_C^V * C_{Sprache}^V + w_Z^V * Z_{Sprache}^V \\ & + w_M^S * M_{Sprache}^S + w_C^S * C_{Sprache}^S + w_Z^S * Z_{Sprache}^S \end{aligned} \tag{5.2}$$

Um die Konstanten $X^{Y}_{Sprache}.X \in \{M, C, Z\}.Sprache \in \{Solverational, SGG, SUPPLE\}.Y \in \{V, S\}$ zu berechnen muss berücksichtigt werden, dass die Konstanten größer werden, je kleiner die berechnete Zeit zum Verstehen oder je weniger terminale Symbole verwendet werden mussten. Aus diesem Grund wurden die Konstanten mittels folgender Formel berechnet:

$$1 - \frac{E_{Sprache}}{\max(E_{Solverational}, E_{SGG}, E_{SUPPLE})} + \frac{\min(E_{Solverational}, E_{SGG}, E_{SUPPLE})}{\max(E_{Solverational}, E_{SGG}, E_{SUPPLE})}$$

Hierbei ist $E_{Sprache}$ das Ergebnis der jeweiligen Untersuchung für die jeweilige Sprache. Wenn eine Sprache eine Eigenschaft nicht unterstützte (z.B. können Zielfunktionen nicht mit SGGs implementiert werden) erhält die Konstante den Wert 0. Auf Grund der großen Menge von Konstanten finden sich die berechneten Konstanten nur in Tabelle 5.12 und nicht direkt in Gleichung 5.2.

5.8.3 Domänen-gerecht

Laut [70] ist es nicht möglich "Application Domain Criteria" zu definieren ohne sich auf eine Domäne festzulegen. Aus diesem Grund macht J. Howatts auch keine weiteren Angaben zu diesen Kriterien.

Da diese Arbeit sich auf Modelltransformation von Benutzerschnittstellen konzentriert, können diese Kriterien aber weiter spezifiziert werden. Es darf nicht außer acht gelassen werden, dass diese Kriterien auch evaluiert werden müssen, daher muss berücksichtigt werden, dass nur begrenzte Ressourcen zur Verfügung stehen, was eine umfassende Betrachtung unmöglich machte.

Es wurden folgende Kriterien festgelegt:

1. Allgemeingültigkeit der Sprachen
2. Grad der Abdeckung von Regeln zur Benutzbarkeit

5.8.3.1 Allgemeingültigkeit der Sprachen

Als erstes wurde untersucht, wie allgemeingültig die drei Sprachen sind. Damit wird entsprechend [112] der Grad der Abdeckung von allen denkbaren Transformationen bezeichnet (s. Abschnitt 5.4). Ein bekanntes Kriterium dazu ist Turingvollständigkeit, die aber nur für JAVA/SUPPLE gegeben ist. Für SGGs und Solverational wurde deshalb anhand von zehn Transformationen untersucht, welche Transformationen von SGGs und Solverational implementiert werden können.

Hierbei ergaben sich 4 von 10 möglichen Transformationen für SGGs und 7 von 10 möglichen für Solverational. In SGGs lassen sich weder neue Constraints

definieren, noch ist es möglich Zielfunktionen zu spezifizieren. In JAVA/SUPPLE lassen sich alle Transformationen implementieren (Turing-vollständig).

Auch wenn die Untersuchung bzgl. der Abdeckung unzureichend ist und Optimierung in der Untersuchung über-repräsentiert ist, gibt sie einen groben Anhaltspunkt. Dieser erste Anhaltspunkt wird noch durch die Untersuchung in Abschnitt 5.5 unterstützt, die deduktiv zeigt, dass alle räumlichen Relationen (Constraints) von SGGs in Solverational implementiert werden können. Es werden allerdings teilweise nicht-lineare Constraints gefordert, die die Ausführungsgeschwindigkeit der Solverational Transformationsmaschine erheblich beinträchtigen können.

5.8.3.2 Grad der Abdeckung von Regeln zur Benutzbarkeit

Es wurde zusätzlich untersucht, wie gut sich Regeln überhaupt mit Programmiersprachen abbilden lassen (s. Abschnitt 5.7). Dies wurde vorwiegend für Solverational untersucht, aber da JAVA/SUPPLE eine Turing-vollständige Sprache ist, können alle für Solverational beobachteten Regeln auch in JAVA/SUPPLE implementiert werden. Für SGGs wurde eine Abschätzung angegeben.

Von den untersuchten Apple Human Interface Guidelines (s. [10]) können 55,4% mit Solverational formalisiert werden. Von den 139 untersuchten Regeln können damit 78 mit Solverational implementiert werden. Zehn Regeln können sogar ausschließlich mit Solverational deklarativ implementiert werden, da sie nur mit einer deklarativen Modell-zu-Modell Transformationssprache deklarativ implementiert werden können, die Constraints und Constraint-Solving unterstützt (vier weitere unter Annahmen). Es ist nicht weiter verwunderlich, dass sich gerade solche Regeln in Solverational implementieren lassen, da Solverational vor allem für diesen Zweck Constraint-Solving unterstützt. Mit SGGs können nur 47,5% der Regeln umgesetzt werden.

5.8.3.3 Entscheidungskriterium

Es lassen sich wieder einige Konstanten definieren, die im komplexen Entscheidungskriterium verwendet werden können.

Das Kriterium "domänen-gerecht", abgekürzt mit D berechnet sich demnach aus der Allgemeingültigkeit der Sprachen und der Abdeckung von Regeln zur Benutzbarkeit wie folgt:

$$D = w_A * A_{Sprache} + w_R * R_{Sprache} \tag{5.3}$$

Die Gewichte w_A und w_R werden den Entwicklern gewählt. Dabei bezeichnet w_A das Gewicht für die Allgemeingültigkeit von Sprachen, während w_R das Gewicht für die Abdeckung von Regeln zur Benutzbarkeit bezeichnet.

Da diesmal die Abdeckung größer ist, wenn mehr Transformationen oder Regeln implementiert werden können, können die Konstanten ($A_{Sprache}$ und $R_{Sprache}$) direkt abgelesen werden. D.h. als Konstanten werden die Anzahl der implementierbaren Transformationen in Prozent als auch die Anzahl der implementierbaren Regeln in Prozent verwendet. Die berechneten Konstanten finden sich in Tabelle 5.12.

5.8.4 Komplexes Entscheidungskriterium

Entwickler stehen oft vor dem Problem, eine geeignete Programmiersprache für ein Projekt auszuwählen. Dabei werden häufig sehr emotionale Debatten geführt. Diese subjektiven Kriterien stellen kein objektives Entscheidungskriterium dar. Bislang existiert kein objektives Entscheidungskriterium, welche der drei zu vergleichenden Modell-zu-Modell-Transformationssprachen in Projekten verwendet werden sollte. Hier wird daher versucht auf der Basis der bereits eingeführten Kriterien und Untersuchungen ein komplexes Entscheidungskriterium für die drei Sprachen herzustellen. Das komplexe Entscheidungskriterium soll es Entwicklern erlauben, für die verschiedenen, einfachen Kriterien Gewichte anzugeben und so die für ihre Anwendung (die die Gewichte bestimmt) bestmögliche Sprache zu finden. Nach Aufstellung der einzelnen Kriterien für "ausgereift", "benutzbar" und "domänengerecht" kann das gesamte komplexe Entscheidungskriterium, bezeichnet durch S, wie folgt aus den Gleichungen 5.1, 5.2 und 5.3 zusammengesetzt werden:

$$\begin{aligned} S &= A + B + D \\ S &= w_{ausgereift} * T_{Sprache} \\ &\quad + w_M^V * M_{Sprache}^V + w_C^V * C_{Sprache}^V + w_Z^V * Z_{Sprache}^V \\ &\quad + w_M^S * M_{Sprache}^S + w_C^S * C_{Sprache}^S + w_Z^S * Z_{Sprache}^S \\ &\quad + w_A * A_{Sprache} + w_R * R_{Sprache} \end{aligned} \tag{5.4}$$

Die Gewichte w_x^y werden von Entwicklern gewählt. Die Entwickler bestimmten damit den Fokus der Berechnung nach den Anforderungen ihrer Projekte. Die Konstanten können Tabelle 5.12 entnommen werden.

Um zu entscheiden, welche Sprache verwendet wird, werden die drei Ergebnisse für S berechnet. Dabei wird jeweils für eine Sprache mit den angenommenen Gewichten und den für die jeweils zu berechnende Sprache passenden Konstanten gerechnet. Um die passende Sprache zu erhalten, werden die Ergebnisse verglichen und die Sprache, die das Ergebnis mit dem größten Wert erzielt wird als "beste" Sprache gewählt.

Beispiel: Ein Entwickler möchte das komplexe Entscheidungskriterium verwenden. Er verwendet pauschal für alle Gewichte den Wert 1. Dadurch erhält er die

Konstante	SGG	Solverational	JAVA/SUPPLE
$T_{Sprache}$	$0,9$	$0,81$	1
$M^V_{Sprache}$	1	1	$0,38$
$C^V_{Sprache}$	$0,77$	1	$0,32$
$Z^V_{Sprache}$	0	1	$0,48$
$M^S_{Sprache}$	1	1	$0,27$
$C^S_{Sprache}$	1	$0,96$	$0,93$
$Z^S_{Sprache}$	0	$0,82$	1
$A_{Sprache}$	$0,4$	$0,7$	1
$R_{Sprache}$	$0,48$	$0,55$	1

Tabelle 5.12: Berechnete Konstanten für das komplexe Entscheidungskriterium

Werte $5,55$ für SGG, $7,84$ für Solverational und $6,38$ für JAVA/SUPPLE. Er wählt die Sprache mit dem maximalen Wert, d.h. Solverational mit $7,84$. Das entspricht dem erwarteten Ergebnis, denn "benutzbar" ist durch die Wahl aller Gewichte mit 1 deutlich überbewertet (es gibt mehr Konstanten im Kriterium "benutzbar"). Da Solverational nach den Untersuchungen als besonders benutzbar gilt, scheint das Ergebnis mit der Erwartung überein zu stimmen.

Es soll nicht verschwiegen werden, dass dieses Entscheidungskriterium von der Qualität der Untersuchungen abhängt, sowie eventuell weiteren Faktoren, die hier keine Berücksichtigung finden konnten. Daher ist dieses Entscheidungskriteriums nur als ein Hinweis zu verstehen.

5.8.4.1 Interpretation der Tabelle 5.12

Die Tabelle 5.12 stellt die drei Kontrahenten noch einmal vergleichend dar. Da höhere Werte bessere Ergebnisse bedeuten, können die "Qualitäten der Sprachen" abgelesen werden. Diese Qualitäten sind auf die Tauglichkeit als Sprache zur Modell-zu-Modell-Transformation von Modellen von Benutzerschnittstellen zu beziehen. Betrachtet man die Werte der Tabelle, so stellt man fest, dass Solverational am besten abschneidet. Dieser Vergleich ist Äquivalent zur Berechnung des objektiven Entscheidungskriteriums, bei dem alle Gewichte auf 1 gesetzt wurden. Solverational kann sich demnach insgesamt mit mehr als einem Bewertungspunkt von den Konkurrenten absetzen. Betrachtet man dies unter dem Aspekt, dass möglicherweise nicht alle Untersuchungen richtig bewertet wurden, so kann man selbst wenn mehr als eine Untersuchung komplett falsche Ergebnisse geliefert haben sollte, immer noch Solverational als Gewinner des Vergleiches bezeichnen. Solverational geht

damit aus diesem Vergleich als "beste" Modell-zu-Modell-Transformationssprache zur Transformation von Modellen von Benutzerschnittstellen hervor.

5.9 Zusammenfassung

In diesem Kapitel wurde eine Methodik zur Evaluation von Modell-zu-Modelltransformationssprachen zur Transformation von Modellen von Benutzerschnittstellen entwickelt. Es wurde ein Entscheidungskriterium entwickelt, welches die möglichst objektive Wahl einer entsprechenden Transformationssprache unterstützt. Das Entscheidungskriterium liegt als Formel vor, die von potentiellen Entwicklern mit passenden Gewichten versehen werden kann, um eine passende Sprache zu wählen, die an ihre Projekte angepasst ist. Die Formel in Abschnitt 5.8.4 unterstützt die Wahl von SGGs, JAVA/SUPPLE und Solverational.

Im Rahmen der Methodik wurden in den Abschnitten 5.2 bis 5.7 verschiedene Kriterien untersucht, deren Ergebnisse in die Formel eingingen:

Ausgereift, Abschnitt 5.1.4 Für das Kriterium ausgereift wurde untersucht, ob die Sprachen mehrdeutige Sprachkonstrukte erlauben und ob es sich bei den implementierten Interpretern um reine Prototypen handelte. Solverational hat auf Grund seiner Abhängigkeit von OCL eine mehrdeutige Sprache. Der Solverational Interpreter wurde in Abschnitt 4.3 vorgestellt. Ferner wurde der Einfluss von Constraints auf die Effizienz eines Solverational Interpreters untersucht (s. Abschnitt 5.6). Es wurde festgestellt, dass "einfache" Constraints weniger entscheidend für die Geschwindigkeit des Interpreters sind als andere Funktionen, die bereits in QVT Relations vorhanden sind.

Benutzbar, Abschnitt 5.1.5 Benutzbarkeit von Sprachen wurde im Rahmen dieser Arbeit anhand von Verständlichkeit (s. Abschnitt 5.2) und Schreibbarkeit (s. Abschnitt 5.3) untersucht. Es wurde bestätigt, dass grafische Modelle schneller zu verstehen und zu notieren sind. Solverational lässt sich insgesamt am leichtesten verstehen - die Notation von Constraints scheint aber in allen drei Sprachen ähnlich effizient.

Domänen-gerecht, Abschnitt 5.8.3 Es wurde untersucht, wie allgemein die Transformationssprachen einsetzbar sind (s. Abschnitt 5.4). Ferner wurde abgeschätzt dass $55,4\%$ der Regeln über die Benutzbarkeit aus [10] in Solverational implementiert werden können (s. Abschnitt 5.7). In Abschnitt 5.5 wurde gezeigt, dass alle "räumlichen Relationen", die aus SGGs bekannt sind, auch in Solverational implementiert werden können.

Die Formel als Gesamtergebnis findet sich in Abschnitt 5.8.4. Im Rahmen dieser Formel entstand eine Tabelle mit Konstanten. Die Sprache Solverational erhielt

im Gegensatz zu SGGs durchweg gute Werte, sodass sie sich als Sieger platzieren konnte. JAVA/SUPPLE, das keine Modell-zu-Modell-Transformationssprache ist und deshalb eigentlich außer Konkurrenz betrachtet werden muss, unterlag vor allem im Kriterium Benutzbar.

Kapitel 6

Zusammenfassung und Ausblick

In dieser Arbeit wurde eine Transformationssprache entwickelt, die die Transformation von Modellen von Benutzerschnittstellen unterstützt. Die Sprache muss dazu vier Grundsätze umsetzen:

1. Die Sprache unterstützt die Transformation von Benutzerschnittstellen.

2. Es handelt sich um eine deklarative Modell-zu-Modell-Transformationssprache, die exogene Transformationen unterstützt.

3. Die Transformationssprache unterstützt Constraints-Solving auf Attributen des Zielmodells.

4. Die Transformationssprache unterstützt die Deklaration von Zielfunktionen.

Aufbauend auf unterschiedlichen Kombinationen dieser Grundsätze lassen sich verschiedene Typen von Transformationen unterscheiden:

1. Constraint-relationale-Transformationen, die Constraint-Solving mit einem deklarativen (relationalen) Programmierparadigma für Modell-zu-Modell-Transformationen kombinieren;

2. Optimierende Constraint-relationale-Transformationen, die zusätzlich die Deklaration von Zielfunktionen erlauben;

3. Derivatprobleme, die verschiedene Typen oder Klassen im Zielmodell für dasselbe Element des Ausgangsmodells zulassen.

6.1 Solverational

In der in dieser Arbeit neu entwickelten Transformationssprache, "Solverational" genannt, sind alle drei Typen von Transformationen ermöglicht worden. Die Transformationssprache basiert auf der Sprache QVT Relations, die als Standard der OMG verabschiedet wurde [117]. Daher können Entwickler neuer Transformationen auf potentiell vorhandene Grundkenntnisse von QVT Relations zurückgreifen und müssen nur die neuen Konstrukte erlernen. Die Anzahl der neu eingeführten Konstrukte ist gering:

1. In Zuweisungen ist jetzt nicht mehr ausschließlich = zugelassen, sondern Wertzuweisungen können implizit durch Ungleichungen erfolgen ($<$, $\leq$, $=$, $\geq$, $>$, $\neq$).

2. Aggregationsfunktionen wurden eingeführt, mit denen Entwickler, die mit Constraint-Programmierung vertraut sind, wie gewohnt Operationen auf Mengen ausführen können. So müssen sie nicht mehr auf OCL-Collections zurückgreifen (Addieren, Multiplizieren, Elemente zählen, u.a.)

3. Zielfunktionen können verwendet werden, um "optimale" Zielmodelle zu erhalten. Zielfunktionen können mittels eines von Entwicklern deklarierten OCL-Ausdrucks festgelegt werden.

4. Zur Deklaration von alternativen Klassen und Constraints für Modellelemente des zu erzeugenden Zielmodells können mehrere Alternativen durch das Schlüsselwort "alternative" eingeführt werden.

6.1.1 Konzepte

Solverational stellt lange bekannte Techniken zur Entwicklung von Benutzerschnittstellen in einer Transformationssprache zur Verfügung. Damit wird die Nutzung dieser Techniken im Zusammenhang mit modell-getriebener Softwareentwicklung von Benutzerschnittstellen in Transformationen ermöglicht:

Constraint-relationale-Transformationen wurden eingeführt, da Constraint-Solving bereits vielfach zur Erzeugung von Benutzerschnittstellen eingesetzt wurde und die Deklaration von Constraint-Solving-Problemen ebenfalls deklarative Eigenschaften hat. Durch die Integration von Constraint-Solving in Solverational bietet Solverational nun ähnliche Möglichkeiten, Modelle von Benutzerschnittellen zu generieren. Dies war vorher nur mittels Ansätzen mit einfachen Constraint-Solvern möglich, die allerdings nur Benutzerschnittstellen generierten und direkt anzeigten. Deren Notationen waren aber nicht mit denen von Modell-zu-Modell-Transformationssprachen konsistent und nicht auf Modelle spezialisiert. Entwickler waren gezwungen mehrere Sprachen zu erlernen.

Die Nutzung von optimierenden Constraint-relationalen-Transformationen erlaubt Entwicklern von Transformation die Deklaration von Zielfunktionen. Dadurch können Zielfunktionen, die bereits zur Generierung von Benutzerschnittstellen verwendet wurden, auch mit Solverational genutzt werden. Modelle von Benutzerschnittellen sind dadurch nicht nur gültige Modelle bzgl. der Erfüllung von Constraints, sondern auch "optimal" bzgl. der Zielfunktion. Die Zielfunktionen können z.B. zum Finden eines "optimalen" Layouts dienen: durch Entwicklung von passenden Zielfunktionen für das vorliegende Projekt, durch Anpassung oder Nutzung der in Kapitel 4 beispielhaft entwickelten Zielfunktionen oder aber durch Nutzung von Zielfunktionen, ähnlich denen aus der Literatur bekannten, z.B. [146, 44, 47].

Derivatprobleme bieten Entwicklern die Möglichkeit für ein zu erzeugendes Modellelement im Zielmodell mehrere Klassen und Constraints als Alternativen zu erlauben. Dies ermöglicht z.B. die Transformation von einer abstrakten Liste in verschiedene konkrete Listen, z.B. eine Drop-Down-Liste oder eine Listbox. Die Zielelemente können dann einerseits zu unterschiedlichen Klassen im Meta-Modell konform sein und andererseits unterschiedliche Constraints verwenden, z.B. um unterschiedliche Mindestgrößen anzugeben.

6.1.2 Umsetzung

Zur Umsetzung der Sprache durch einen Interpreter wurde zuerst eine Architektur entwickelt. Die Implementierung selbst ist ein Übersetzer von Solverational auf ECLiPSe (ein PROLOG-Dialekt mit Unterstützung für Constraint Programmierung). Der Interpreter wurde in Eclipse (eine JAVA IDE) und inklusive einiger Transformationen in eine Entwicklungsumgebung für Benutzerschnittellen (die Mapache-Entwicklungsumgebung) integriert.

Zum Nachweis der Realisierbarkeit von Transformationen zur Transformation von Modellen von Benutzerschnittellen durch Solverational wurden verschiedene Strategien entwickelt. Sie dienen als Orientierungshilfe zur Implementierung weiterer Transformationen und können ähnlich wie Entwurfsmuster für Solverational verwendet werden.

Um Entwicklern die ingenieursmäßige Entwicklung von Software mit Solverational zu erleichtern wurde eine Menge von Schritten/Phasen entwickelt, die in Vorgehensmodelle integriert werden können. Dabei ist es unerheblich ob es sich um "wasserfallartige" oder iterative Vorgehensmodelle handelt.

6.2 Ergebnisse der Evaluation

Zur Durchführung der Evaluation wurde eine neue Methodik entwickelt, die es ermöglicht, Transformationssprachen zur Transformation von Modellen von Benutzerschnittstellen quantitativ zu vergleichen. Für den Vergleich mit Solverational wurden SUPPLE von K. Gajos ([47]) und Spatial Graph Grammars von Kong u.a. ([88]) ausgewählt. Die Methodik ist aber so allgemein, dass eine Erweiterung durch Dritte um weitere Sprachen kein Problem darstellt.

Die Methodik legt drei Kriterien fest, nach welchen Transformationssprachen untersucht werden. Die Sprachen sollten möglichst

1. ausgereift,
2. benutzbar und
3. domänen-gerecht

sein.

Für die drei untersuchten Sprachen wurden für die Kriterien jeweils eine Reihe von Konstanten ermittelt. Diese Konstanten geben Aufschluss darüber, wie gut die Sprache für das jeweilige Kriterium geeignet erscheint, d.h. z.B. ob die Sprache im Verhältnis als "ausgereift", "benutzbar" oder "domänen-gerecht" bezeichnet werden kann. Die Konstanten wurden mittels verschiedener Untersuchungen bestimmt: theoretische Ergebnisse aus der Literatur, Benutzerstudien um die Benutzbarkeit der Sprachen zu untersuchen und Beispiele um den Passgrad der domänen-spezifischen Sprachen an die Domäne zu bestimmen.

6.2.1 Ausgereift

Das Kriterium "ausgereift" umfasste verschiedene sprachliche Aspekte der Transformationssprachen, insbesondere ob eine Grammatik einer Sprache zweideutige Konstrukte enthält und ob für die Sprachen ein Interpreter implementiert werden konnte. Solverational hat hierbei auf Grund dessen, dass die Sprache auf QVT Relations aufbaut, eine direkte Abhängigkeit zur Sprache OCL [116]. Die OCL ist dafür bekannt, dass sie in gewissen, praktisch kaum relevanten Fällen nicht entscheidbar ist, und damit mehrdeutig ist. Dadurch schnitt auch Solverational nicht so gut ab wie die etablierte Konkurrenz, die allerdings weniger sprachliche Möglichkeiten bot (SGGs) oder nur eine API-Erweiterung von JAVA waren (SUPPLE). In diesem Rahmen wurde auch ein Solverational-Interpreter präsentiert und es wurde festgestellt, dass "einfache" Constraints unter Umständen weniger entscheidend für die Geschwindigkeit des Interpreters sind als andere Funktionen, die bereits in QVT Relations vorhanden sind. Der Solverational-Interpreter kann als relativ effizient bezeichnet werden.

6.2.2 Benutzbar

Das Kriterium "benutzbar" ist aus den Teilkriterien "Lesbarkeit" und "Schreibbarkeit" zusammengesetzt. Lesbarkeit bezeichnet die Geschwindigkeit mit der Entwickler Sprachen verstehen können. Die Fragen, die in einer Studie zur Lesbarkeit von Notationen, Constraints und Zielfunktionen der Sprachen gestellt wurden, konnten von besonders vielen Nutzern mit Solverational am schnellsten beantwortet werden, was darauf schließen lässt, dass Solverational eine besonders lesbare Sprache ist. Schreibbarkeit bezeichnet einen Wert, der die Anzahl der nichtterminalen Symbole repräsentiert, die zum Schreiben von Transformationen durchschnittlich benötigt werden. Die Unterschiede zwischen den Ergebnissen waren bei dieser Untersuchung statistisch nicht signifikant, sodass von ähnlicher Schreibbarkeit bei den drei Sprachen auszugehen ist.

Insgesamt lassen die Studienergebnisse darauf schließen, dass Solverational eine besonders benutzbare Sprache ist. Insbesondere die Verständlichkeit von SGG und SUPPLE wurde deutlich schlechter bewertet.

6.2.3 Domänen-gerecht

Nach der in dieser Arbeit vertretenen Ansicht ist eine Sprache immer für einen bestimmten Zweck entwickelt worden und damit in gewisser Hinsicht "domänenspezifisch". Insbesondere Modell-zu-Modell-Transformationssprachen werden immer für die Domäne (den Zweck) der Transformation von Modellen entwickelt. Die Frage ist allerdings, wie gut die Sprachen Aufgabenstellungen innerhalb der Domäne (Transformation von Modellen von Benutzerschnittstellen) abdecken und vielleicht darüber hinaus weitere Aufgaben erfüllen können. Deshalb wurde subjektiv anhand eines bekannten Style Guides [10] abgeschätzt, wie viele Regeln (Aufgabenstellungen innerhalb einer Transformation) mit Solverational entwickelt werden können. Es konnten $55,4\%$ der Regeln implementiert werden. Zusätzlich konnten die räumlichen Relationen der Sprache SGG in Solverational durch Constraints abgebildet werden. So ist davon auszugehen, dass Solverational mindestens so ausdrucksstark ist wie die räumlichen Relationen der Sprache SGG.

Im Vergleich der beiden Transformationssprachen kann Solverational also Regeln für die Benutzbarkeit besser abbilden und ist ausdrucksstärker bzgl. räumlicher Relationen. Dies ist ein starker Hinweis, dass die Theorie der vorliegenden Arbeit – also die Unterstützung von Constraint-Solving und Optimierung – zu "besseren" Transformationssprachen zur Transformation von Modellen von Benutzerschnittstellen führt.

6.2.4 Formel

Die drei Kriterien und die berechneten Konstanten konnten in eine Formel ("Metrik") integriert werden, die Entwicklern hilft, für ein bestimmtes Projekt eine bestimmte Transformationssprache auszuwählen. Dadurch wird eine Sprache nicht allein auf Grund persönlicher Präferenzen ausgewählt, sondern auf der Basis einer Formel. Die Entwickler können dabei verschiedene Gewichte für die Kriterien vergeben, sodass sich die Formel gut an die Gegebenheiten der Projekte der Entwickler anpassen lässt. Die Konstanten entsprechen den erwähnten Konstanten für die verschiedenen Kriterien. Die gesamte Formel findet sich in Abschnitt 5.8.4.

6.3 Ausblick

Es bieten sich vier Felder für weitere Forschungen an: Weiterentwicklung der Theorie, Weiterentwicklung der Sprache, weitere Anwendungen auf Modelle von Benutzerschnittstellen und Weiterentwicklung der Methodik zur Evaluation bzw. der Evaluation selbst.

6.3.1 Weiterentwicklung der Theorie

Planungsaufgaben sind Aufgabenstellungen, bei denen ein Ziel und eine Menge von möglichen "Aktionen" vorgegeben werden. Durch Ausführen einer Teilmenge der Menge von Aktionen kann das Ziel erreicht werden. Z.B. wurden Lösungen für Planungsaufgaben für autonome Robotersysteme entwickelt. Hier soll z.B. ein autonomer Roboter eine Cola-Dose in einen bestimmten Mülleimer werfen. Um sein Ziel zu erreichen muss der Roboter seine Route planen und z.B. die Aktionen "Türe öffnen" und "Fahrstuhl fahren" einplanen.

Im Bereich der modell-getriebenen Softwareentwicklung von Benutzerschnittstellen können sog. Zielmodelle verwendet werden, um die Ziele der zu modellierenden Applikation für die Entwickler transparent zu machen (z.B. [168]). Die Entwickler passen dann die anderen Modelle per Hand so an, dass sie dem Zielmodell genügen. Insbesondere wird das Task-Modell angepasst, sodass nach der Anpassung Aufgaben oder Aktionen die Ziele aus dem Zielmodell implementieren. Die automatische Erzeugung des Task-Modells ist also eine Planungsaufgabe.

Das Lösen von Planungsaufgaben kann durch Constraint-Solving implementiert werden [107]. Da Solverational Constraint-Solving implementiert, scheint die Lösung von Planungsaufgaben mit Solverational möglich. Die Sprache bietet aber derzeit keine Möglichkeiten, die Ziele der Planungsaufgaben als relevante Teile der Transformation zu identifizieren. Zusätzlich kann der implementierte Interpreter derzeit noch keine Assoziationen über Constraints festlegen, da die Eigenschaf-

ten von Klassen zum Setzen von Assoziationen in Solverational an feste Muster gebunden werden.

Um die Sprache auch zur Durchführung von Planungsaufgaben verwenden zu können, müssen diese Möglichkeiten geschaffen werden. Beispielsweise könnte ein Konstrukt der Sprache Teile von Modellen als Ziele definieren. Hierbei könnte es sich um Constraints handeln, die während der Transformation an das Zielmodell gebunden werden. Diese wären dann die zu erreichende Zielvorgabe – ähnlich der Idee aus [107] die diesen Ansatz ohne Modelle propagiert.

Derzeit ist die Nutzung von Constraints nur zwischen Modellelementen möglich, die über Assoziationen "verbunden" sind. Sind die Modellelemente aber nicht direkt über Assoziationen verbunden, so können die Constraints die Modellelemente nicht zueinander in Relation setzen (z.B. Attribute vergleichen). Die Einführung von globalen Constraints würde es hingegen ermöglichen, beliebige Constraints in Relation zu setzen, da diese prinzipiell für alle Modellelemente gelten, sofern die Constraints die Wahl nicht weiter einschränken. Dieses Konzept könnte ähnlich zu den bereits entwickelten Zielfunktionen implementiert werden.

6.3.2 Implementierung

Wie bereits in Abschnitt 4.3 beschrieben, ist nicht der komplette Funktionsumfang von QVT Relations implementiert – dies ist nicht weiter verwunderlich, da bis jetzt noch keine einzige komplette Implementierung von QVT Relations existiert. Einige Fragestellungen werden allerdings erst mit Solverational besonders herausfordernd:

- Wie lassen sich sog. In-Place Transformationen implementieren? Klassische Implementierungen können auf bestehende Ansätze zurückgreifen. Im Fall von Solverational muss der bestehende Algorithmus erweitert werden. Dies könnte geschehen, in dem Modelle in Terme und Constraints transformiert und die erzeugten Terme und Constraints zu denen vom jetzigen Algorithmus erzeugten hinzugefügt werden. Der Algorithmus und das Constraint System müssten zusätzlich so erweitert werden, dass vor jeder Erzeugung auf Existenz geprüft wird – was gleichzeitig einer Implementierung des aus dem QVT Relations-Standard bekannten Check-Then-Enforce entsprechen würde, bei dem vor der Erzeugung eines neuen Modellelements geprüft wird, ob eines mit den gleichen Werten und Assoziationen bereits existiert.

- Im Fall von Solverational ist die Effizienz der Implementierung besonders wichtig, da zusätzlich Constraint-Solving- und Optimierungsprobleme gelöst werden müssen. Dabei sollte sich die verbesserte Implementierung nicht an der Geschwindigkeit bereits existierender QVT Implementierungen orientieren, da diese für größere Modelle nicht effizient zu sein scheinen (s. Abschnitt

5.6). Ein möglicher Ansatzpunkt ist die Entwicklung eines speziellen Solvers, der auf die speziellen Eigenschaften des Constraint-Solving und Optimierungsproblems eingeht. Hier ist besonders interessant auf die Erzeugung von Assoziationen und alternativen Constraints genauer einzugehen, da es sich um Constraints handelt, die nur in speziellen Fällen aktiv sind. D.h. hier kann ausgenutzt werden, dass der Suchraum kleiner wird, wenn nicht alle Möglichkeiten in allen Fällen betrachtet werden (z.B. [160]).

6.3.3 Anwendung auf Benutzerschnittstellen und Evaluation

Im Verhältnis zum Sprachumfang ist die bisher implementierte Anzahl von Transformationen gering. Um die Möglichkeiten der Transformationssprache zum Implementieren von Transformationen genauer zu bestimmen, müsste festgestellt werden, welche Klassen von Transformationen existieren. Eine Klassifikation könnte auf der Basis eines bestimmten Meta-Modells eine Menge von Strategien, Zielfunktionen, Constraints und Transformationen nach Modalitäten, Kontexten und Plattformen kategorisieren. Es wäre interessant durch die Klassifikation jeweils möglichst viele Modalitäten, Kontexte und Plattformen abzudecken. Entsprechend der Klassifikation könnten Transformationen implementiert und so eine komplette Bibliothek von Transformationen zur Verfügung gestellt werden, die die Möglichkeiten des Sprachumfangs besser abdeckt. Dies würde es Entwicklern ermöglichen sehr schnell, bei geringem eigenem Aufwand, Benutzerschnittstellen für sehr unterschiedliche Modalitäten, Kontexte und Plattformen zu entwickeln, indem diese die Transformationen aus der Bibliothek nutzen.

6.3.4 Evaluation

Bisher wurde nur eine Formel (Metrik) als Entscheidungskriterium aufgestellt. In Zukunft kann eine Studie durchgeführt werden, die die Ergebnisse der Formel mit den tatsächlichen Ergebnissen der Projekte vergleicht, d.h. die Formel würde weiter validiert werden. Dazu bietet es sich an, mehrere Teams von Entwicklern in den verschiedenen Sprachen zu unterrichten und dann die Teams mit gleichen Projekten zu beauftragen. Die Teams könnten dann die Formel nutzen, um die für ihr Projekt passende Sprache zu bestimmen. Nach der Durchführung des Projektes mit der jeweiligen Sprache wird dann mit Hilfe der Entwickler untersucht (z.B. Fragebogen), wie gut die jeweilige Sprache deren Anforderungen entsprochen hat und damit, ob das Ergebnis der Formel ihren Erwartungen entsprach.

Anhang A

Beispiele für Ergebnisse von Transformationen

A.1 Ergebnisse der Panel-Transformation

Die folgenden Abbildungen sind beispielhafte Ergebnisse der Panel-Transformation, die in Abschnitt 4.5.3 vorgestellt wurde. Hierbei kommen verschiedene Strategien zum Einsatz, für die hier jeweils zwei Beispiele aufgeführt werden sollen; die Erklärungen zu den Strategien finden sich im entsprechenden Abschnitt in Kapitel 4. Jede Benutzerschnittstelle entstand durch die Transformation des abstrakten Modells der Benutzerschnittstelle aus Abbildung 4.13a.

Die Ergebnisse wurden für verschiedene Kontexte ausgeführt. Gezeigt werden ausschließlich Beispiele, die den Nutzen der jeweiligen Zielfunktion besonders gut herausstellen. Die Vielfalt an Benutzerschnittstellen, die durch das Austauschen der Zielfunktion, d.h. einer einzigen Zeile Quellcode, erreicht werden kann, erscheint auf den ersten Blick überraschend, ist aber dem in dieser Arbeit kompromisslos deklarativen Ansatz geschuldet.

A.1.1 Möglichst-Links-Strategie

Kontext: Bildschirmauflösung 1500*1000

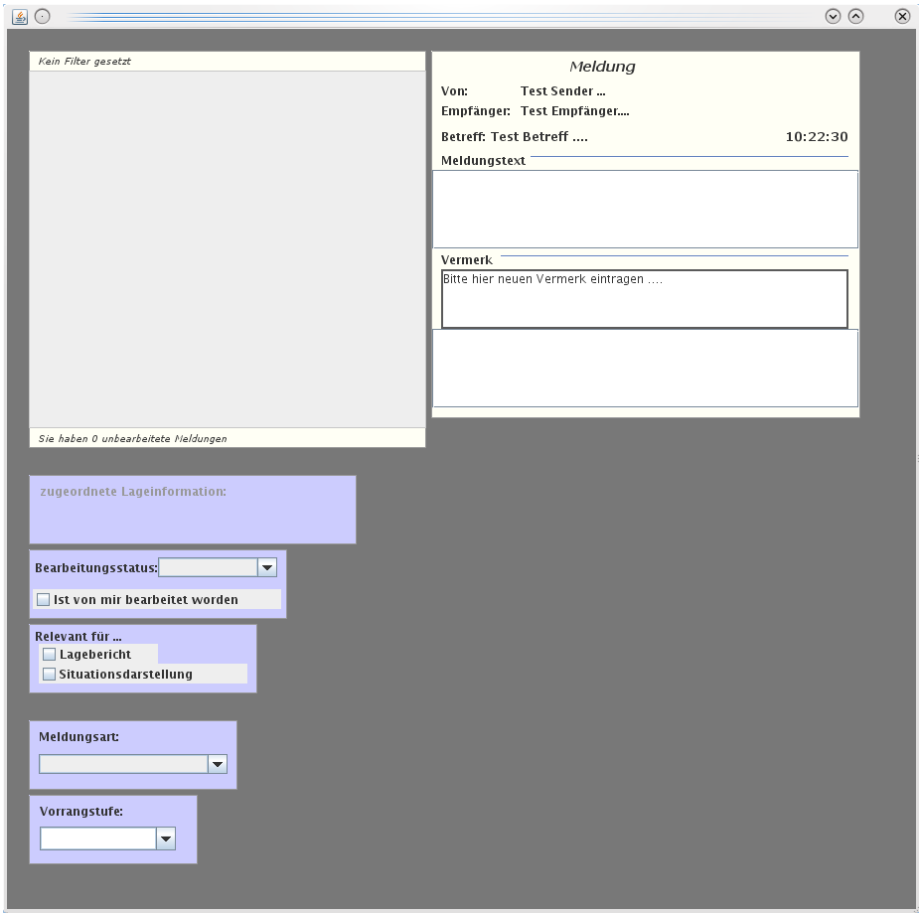


Abbildung A.1: Ergebnis Möglichst-Links-Strategie, 1500*1000

Kontext: Bildschirmauflösung 1500*800

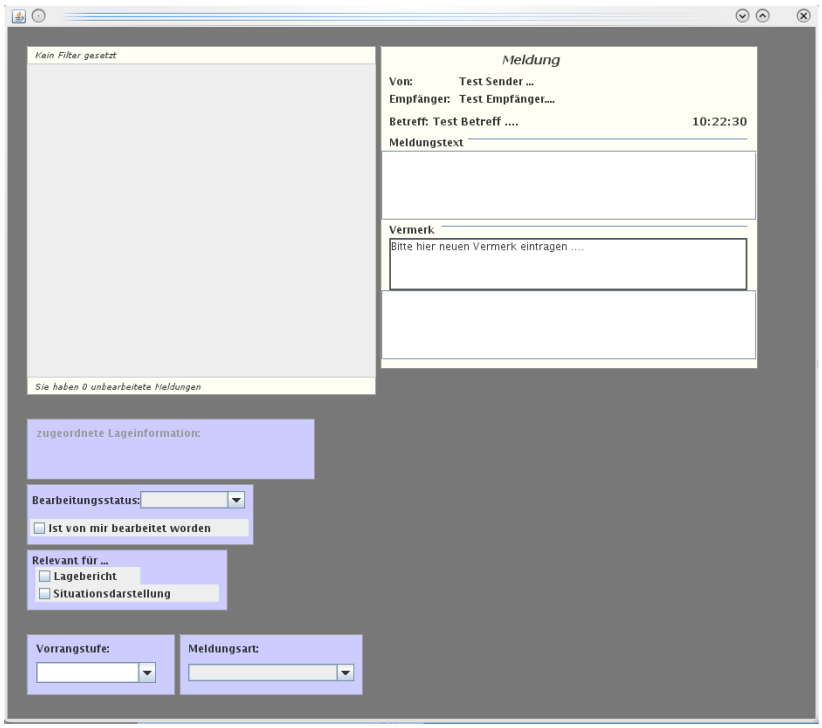


Abbildung A.2: Ergebnis Möglichst-Links-Strategie, 1500*800

A.1.2 Möglichst-Oben-Strategie

Kontext: Bildschirmauflösung 1500*1000

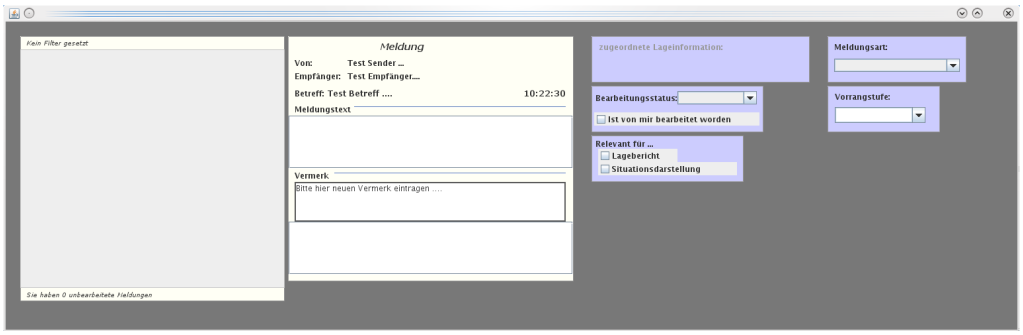


Abbildung A.3: Ergebnis Möglichst-Oben-Strategie, 1500*1000

Kontext: Bildschirmauflösung 1100*1000

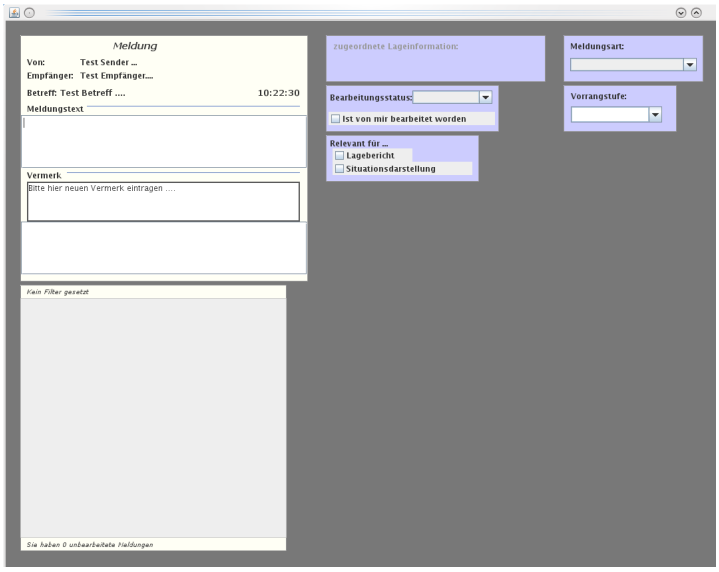


Abbildung A.4: Ergebnis Möglichst-Oben-Strategie, 1100*1000

A.1.3 Relevanz-Strategie

Kontext: Bildschirmauflösung 1500*1000

Relevanz wurde für pnlMeldung (hier links unten) auf 20, für pnlMessagePrio gesetzt (links oben)

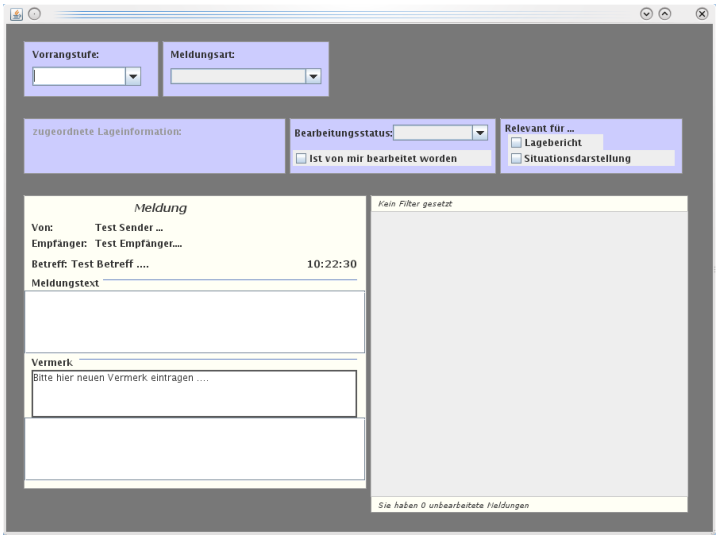


Abbildung A.5: Ergebnis Relevanz-Strategie, 1500*1000

Kontext: Bildschirmauflösung 1000*1000
Relevanz wurde für pnlMeldung (hier links oben) auf 20 gesetzt

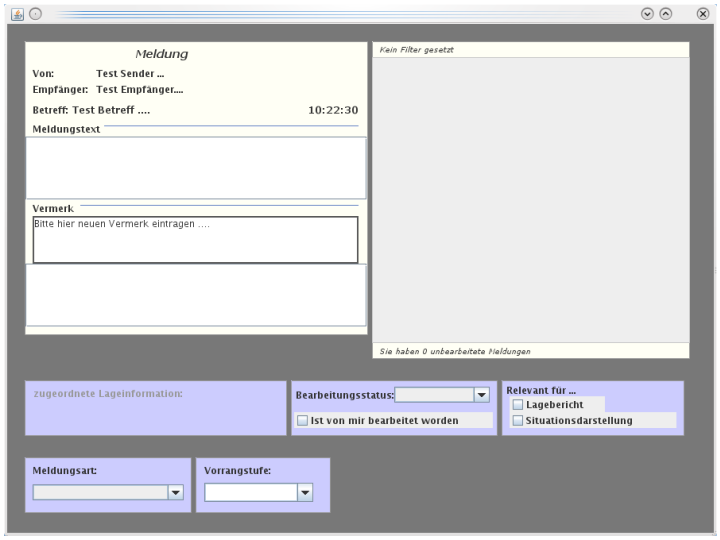


Abbildung A.6: Ergebnis Relevanz-Strategie, 1000*1000

A.1.4 2D-Fitt's-Law-Strategie

Kontext: Bildschirmauflösung 1200*700

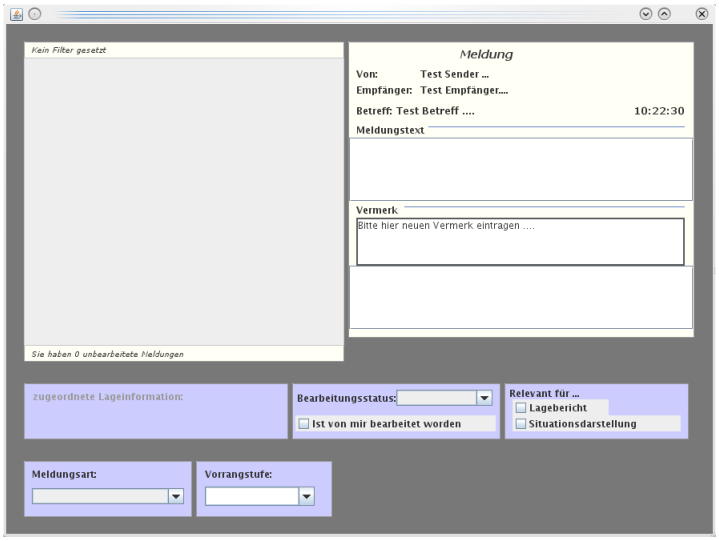


Abbildung A.7: Ergebnis Fitt's-Law-Strategie, 1200*700

Kontext: Bildschirmauflösung 1600*500

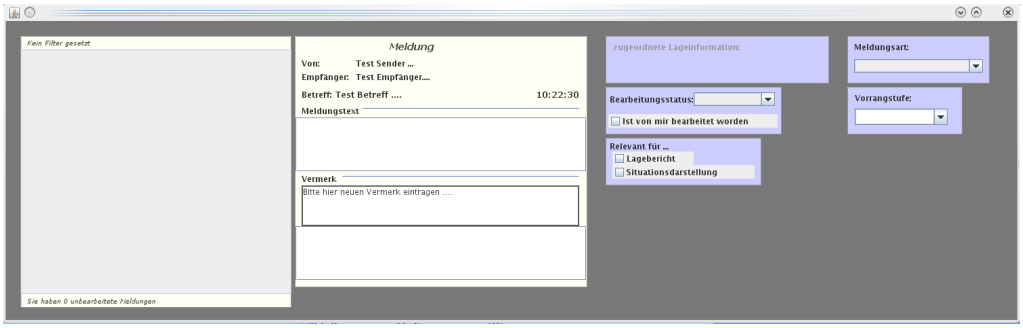


Abbildung A.8: Ergebnis Fitt's-Law-Strategie, 1600*500

A.1.5 Behring-Strategie

Kontext: Bildschirmauflösung 1500*1000
Gewicht rechts: 1, Gewicht unten: 1

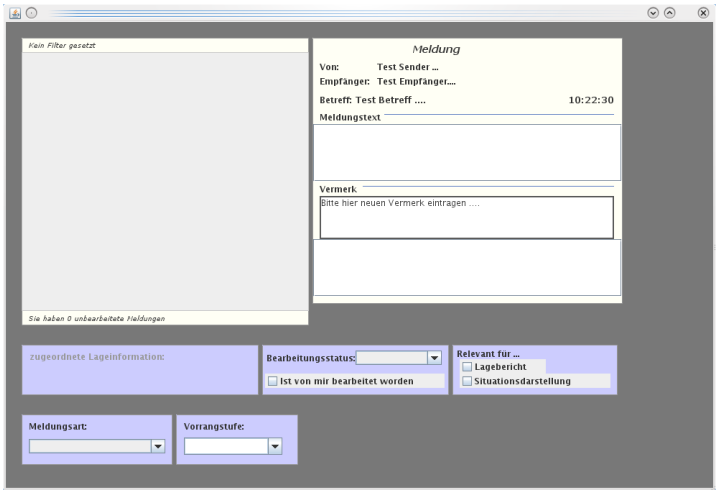


Abbildung A.9: Ergebnis Behring-Strategie, 1500*1000, Gewichte=1

Kontext: Bildschirmauflösung 1500*1000
Gewicht rechts: 1, Gewicht unten: 20

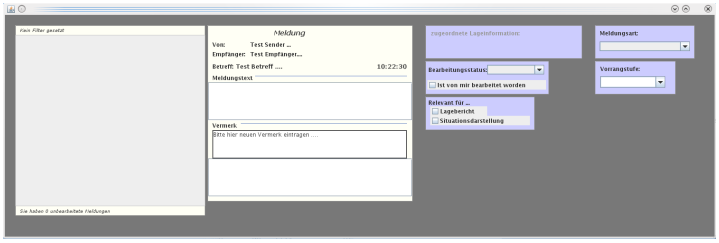


Abbildung A.10: Ergebnis Behring-Strategie, 1500*1000, Gewicht unten=20

Anhang B

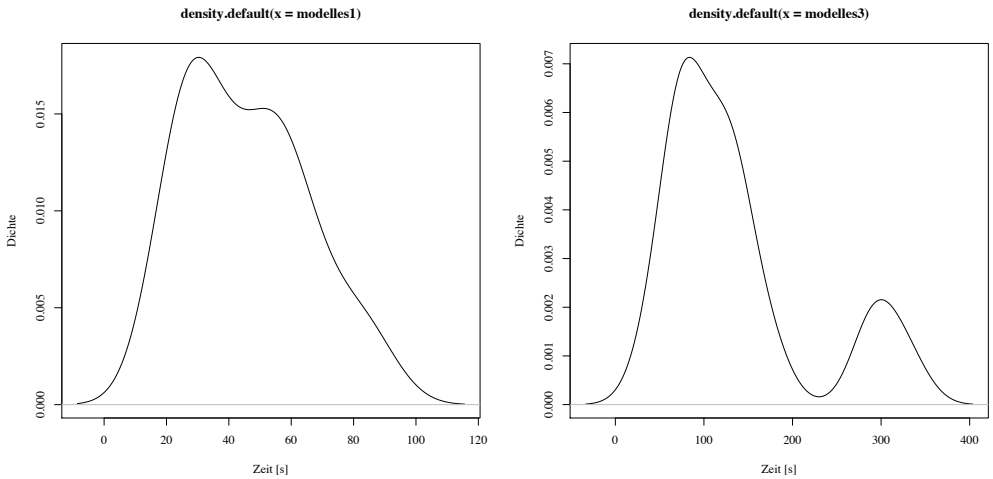
Graphen zu Studienergebnissen

B.1 Approximierte Verteilungen

Die Abbildungen B.1 bis B.3 stellen approximierte Verteilungen der Dichte der Antwortzeiten dar. Es sollte gemessen werden, wie leicht verständlich die verschiedenen Fragen für die Probanden waren. Bei ausreichend großzügiger Betrachtung der Verteilungen ist bei allen Graphen eine Ähnlichkeit zur Normalverteilung erkennbar, auch wenn einige Graphen Ausreißer enthalten.

B.1.1 Modelle

Die folgenden Grafiken zeigen berechnete approximierte Verteilungsfunktionen der Dichte der Antwortzeiten der Probanden auf die Fragen zur Messung des Verständnisses der Modelle.

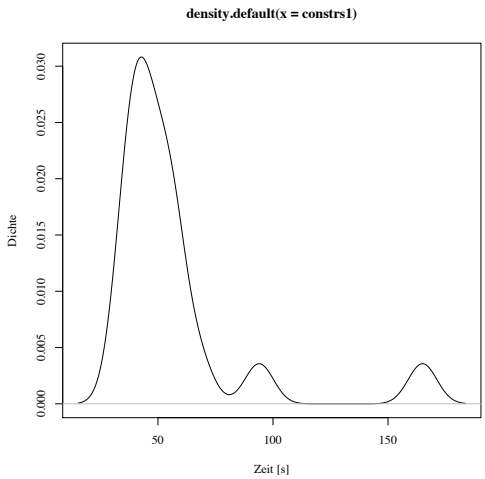


(a) Grafische Modelle (b) SUPPLE

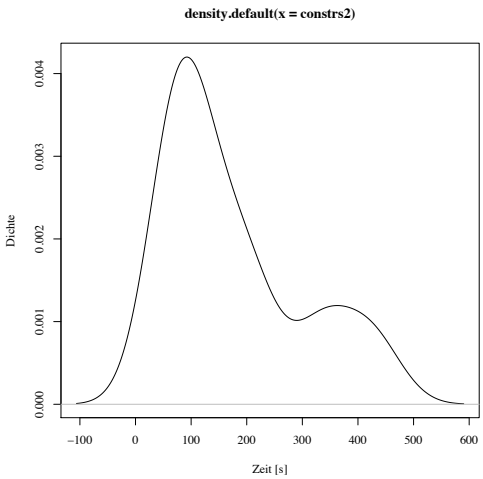
Abbildung B.1: Approximierte Verteilung der Antwortzeiten der Probanden für unterschiedliche Notationen für Modelle

B.1.1.1 Constraints

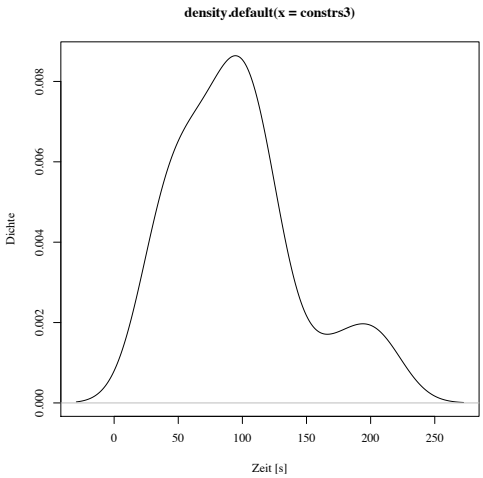
Die folgenden Grafiken zeigen berechnete approximierte Verteilungsfunktionen der Dichte der Antwortzeiten der Probanden auf die Fragen zur Messung des Verständnisses der Constraints.



(a) Solverational



(b) SUPPLE

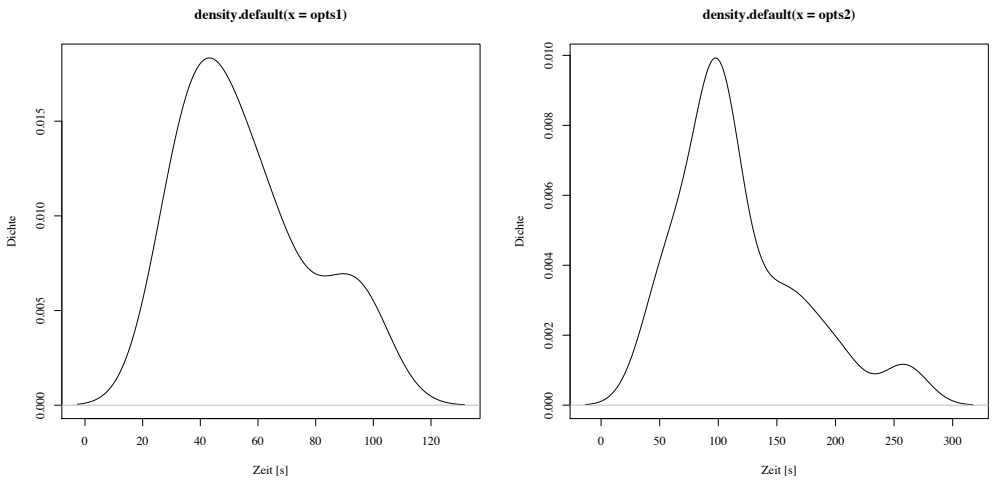


(c) SGG

Abbildung B.2: Approximierte Verteilung der Antwortzeiten der Probanden für unterschiedliche Notationen für Constraints

B.1.2 Optimierung

Die folgenden Grafiken zeigen berechnete approximierte Verteilungsfunktionen der Dichte der Antwortzeiten der Probanden auf die Fragen zur Messung des Verständnisses der Zielfunktionen.



(a) Solverational (b) SUPPLE

Abbildung B.3: Approximierte Verteilung der Antwortzeiten der Probanden für unterschiedliche Notationen für Zielfunktionen

B.2 Boxplots der Antwortzeiten

In den Abbildungen B.4 bis B.6 werden die Daten der Antwortzeiten grafisch in Form von Box-Plots (auch "Box-Whisker-Plot") dargestellt. Man erkennt den Median und bekommt einen Überblick über die Verteilung der Daten. Es zeigt sich, dass die Daten nur zum Teil als normal verteilt angenommen werden dürfen. Aus diesem Grund wurden in Kapitel 5 die Ergebnisse mit Tests wiederholt, die von den Verteilungen unabhängig sind.

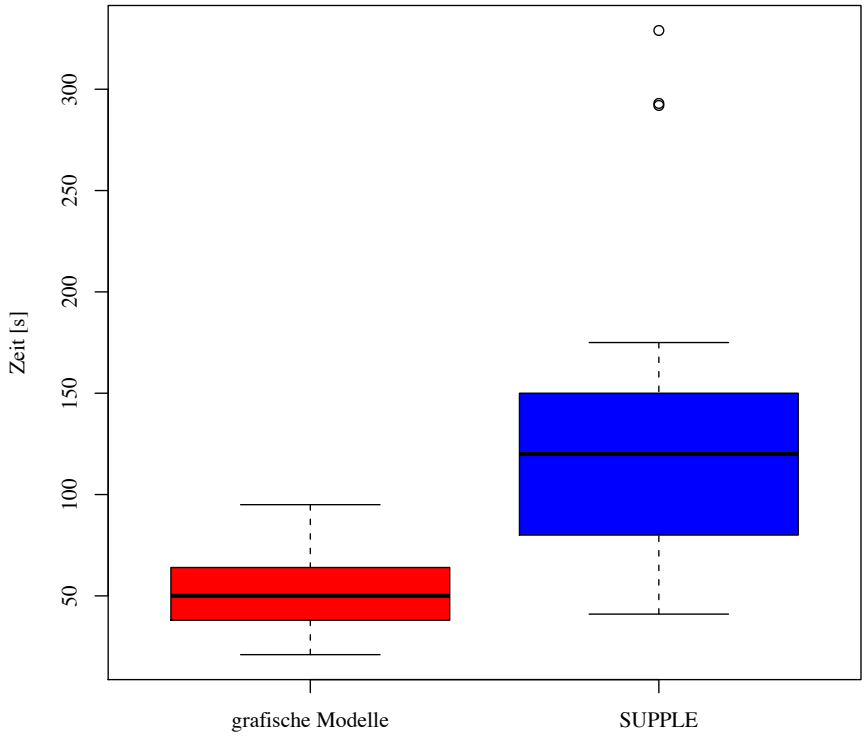


Abbildung B.4: Boxplot der Antwortzeiten für unterschiedliche Notationen von Modellen

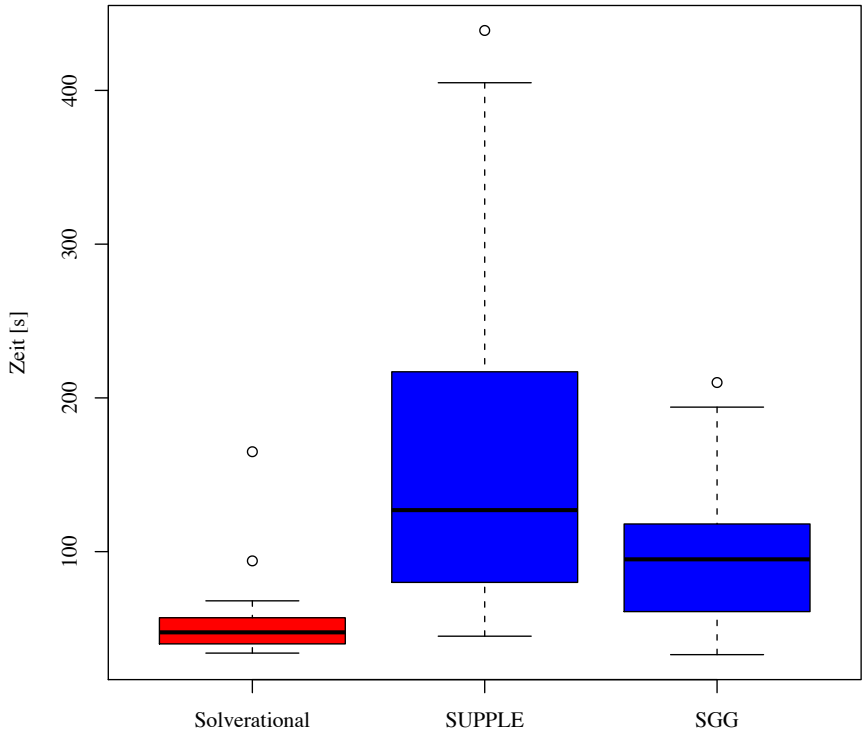


Abbildung B.5: Boxplot der Antwortzeiten für unterschiedliche Notationen von Constraints

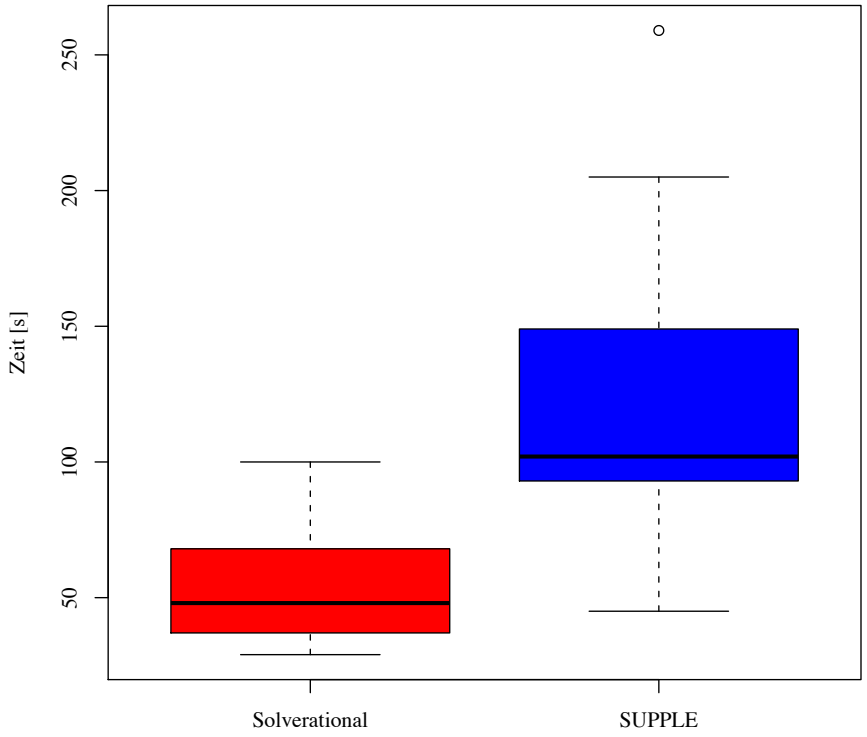


Abbildung B.6: Boxplot der Antwortzeiten für unterschiedliche Notationen von Zielfunktionen

Anhang C

Abkürzungsverzeichnis

QVT Query View Transformations, standardisierte Familie von Modell-zu-Modell-Transformationssprachen, s. [117]

BNF Backus-Naur-Form, Notation für Grammatiken, s. [86]

UML Unified Modeling Language, relativ allgemeines Meta-Modell, s. [114]

MOF Meta Object Facility, Meta-Meta-Modell s. [113]

KLM Keystroke Level Model, einfaches Modell zur Approximation von Eingabezeiten von Benutzerschnittstellen, s. [80]

MDA Model Driven Architecture, Konzept zur Entwicklung von Software mit Transformationen, s. [118]

OCL Object Constraint Language, Sprache zur Notation von Constraints in Modellen, s. [116]

CLP Constraint Logische Programmierung/Programmiersprache, deklaratives Programmierparadigma auf Basis von Constraints, s. z.B. [11]

CSP Constraint Solving Problem, Beschränkung einer Menge, sodass die beschränke Menge die Lösung eines Problem ist, s. z.B. [160]

SGG Spatial Graph Grammars, können u.a. räumliche Relationen notieren, [88]

AGG Attributierte Graph Grammatiken, s. [137]

TGG Tripel Graph Grammatiken, s. [145]

Anhang D

Literaturverzeichnis

[1] Aagedal, J. O. ; Solheim, I.: New roles in model-driven development. In: *Proccesdings of Second European Workshop on Model Driven Architecture (MDA)*. Canterbury, England, 2004

[2] Abowd, G. D. ; Coutaz, J. ; Nigay, L.: Structuring the Space of Interactive System Properties. In: *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction*, North-Holland, 1992. – ISBN 0–444–89904–9, S. 113–129

[3] Accot, J. ; Zhai, S.: Beyond Fitts' law: models for trajectory-based HCI tasks. In: *CHI '97: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA : ACM, 1997. – ISBN 0–89791–802–9, S. 295–302

[4] Akehurst, D. H. (Hrsg.): *Second European Workshop on Model Driven Architecture (MDA) with an emphasis on Methodologies and Transformations*. Canterbury, England, September 2006

[5] AlGhamdi, J. ; Urban, J.: Comparing and assessing programming languages: basis for a qualitative methodology. In: *SAC '93: Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing*. New York, NY, USA : ACM, 1993. – ISBN 0–89791–567–4, S. 222–229

[6] Amelunxen, C. ; Königs, A. ; Rötschke, T. ; Schürr, A.: MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In: Rensink, A. (Hrsg.) ; Warmer, J. (Hrsg.): *Model Driven Architecture - Foundations and Applications: Second European Conference* Bd. 4066. Heidelberg : Springer-Verlag, 2006 (Lecture Notes in Computer Science (LNCS)), S. 361–375

[7] Anastasakis, K. ; Bordbar, B. ; Georg, G. ; Ray, I.: UML2Alloy: A Challenging Model Transformation. In: *Model Driven Engineering Languages and Systems*, 2007, S. 436–450

[8] Anastasakis, K. ; Bordbar, B. ; Küster, J. M.: Analysis of Model Transformations via Alloy. In: *ModeVVa 2007*, 2007, S. 47–56

[9] Anckaert, B. ; Madou, M. ; De Bosschere, K.: A model for self-modifying code. In: *Proceedings of the 8th international conference on Information hiding*. Berlin, Heidelberg : Springer-Verlag, 2007 (IH'06). – ISBN 978–3–540–74123–7, 232–248

[10] Apple, I.: *Apple Human Interface Guidelines*. `http://developer.apple.com/documentation/UserExperience/Conceptual/OSXHIGuidelines/OSXHIGuidelines.pdf`. Version: 2005

[11] Apt, K. R. ; Wallace, M.: *Constraint Logic Programming using ECLiPSe*. New York, NY, USA : Cambridge University Press, 2007. – ISBN 0521866286

[12] Aquinto, N. ; Vanderdonckt, J. ; Pastor, O.: Transformation Templates: Adding Flexibility to Model-Driven Engineering of User Interfaces. In: *Proceedings on the 25th Symposium of Applied Computing*. Sierre, Switzerland : ACM, März 2010

[13] Badros, G. J.: *Extending Interactive Graphical Applications with Constraints*, University of Washington, Diss., 2000

[14] Badros, G. J. ; Borning, A. ; Marriott, K. ; Stuckey, P.: Constraint cascading style sheets for the web. In: *UIST '99: Proceedings of the 12th annual ACM symposium on User interface software and technology*. New York, NY, USA : ACM, 1999. – ISBN 1–58113–075–9, S. 73–82

[15] Badros, G. J. ; Nichols, J. ; Borning, A.: Scwm: An Extensible Constraint-Enabled Window Manager. In: *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. Berkeley, CA, USA : USENIX Association, 2001. – ISBN 1–880446–10–3, S. 225–234

[16] Barták, R.: Constraint Programming: In Pursuit of the Holy Grail. In: *Proceedings of the Week of Doctoral Students (WDS99), Part IV*. Prague : MatFyzPress, Juni 1999, S. 555–564

[17] Basili, V. R. ; Rombach, H. D.: Towards improvement-oriented software environments. In: *IEEE Transactions on Software Engineering* 14 (1988), Nr. 6, S. 728–738

[18] Behring, A. ; Petter, A. ; Mühlhäuser, M.: Rapidly Modifying Multiple User Interfaces of one Application. In: *ICSOFT (SE)*, INSTICC Press, 2009

[19] Bézivin, J. ; Büttner, F. ; Gogolla, M. ; Jouault, F. ; Kurtev, I. ; Lindow, A.: Model Transformations? Transformation Models! In: *Proceedings of the 9th International Conference, MoDELS 2006* Bd. 4199. Berlin : Springer-Verlag, 2006 (Lecture Notes in Computer Science), 440–453

[20] Bistarelli, S. ; Frühwirth, T. ; Marte, M.: Soft constraint propagation and solving in CHRs. In: *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*. New York, NY, USA : ACM, 2002. – ISBN 1–58113–445–2, S. 1–5

[21] Bodart, F. ; Hennebert, A. ; Leheureux, J. ; Vanderdonckt, J.: Towards a dynamic strategy for computer-aided visual placement. In: *AVI '94: Proceedings of the workshop on Advanced visual interfaces*. New York, NY, USA : ACM, 1994. – ISBN 0–89791–733–2, S. 78–87

[22] Bodart, F. ; Vanderdonckt, J.: On the Problem of Selecting Interaction Objects. In: Cockton, G. (Hrsg.) ; Draper, S. W. (Hrsg.) ; The University of Glasgow (Veranst.): *Proceedings of HCI'94 "People and Computers IX"* The University of Glasgow, 1994

[23] Brandenburg, F.: Designing graph drawings by layout graph grammars. In: Tamassia, R. (Hrsg.) ; Tollis, I. G. (Hrsg.): *Graph Drawing* Bd. 894, Springer-Verlag, 1995 (Lecture Notes in Computer Science), S. 416–427

[24] *Kapitel* Using Declarative Descriptions to Model User Interfaces with MASTERMIND. In: Browne, T. ; Davila, D. ; Rugaber, S. ; Stirewalt, K.: *Formal Methods in Human Computer Interaction*. Springer-Verlag, 1997

[25] Brucker, A. D. ; Doser, J. ; Wolff, B.: Semantic Issues of OCL: Past, Present, and Future. In: *Electronic Communications of the EASST* 5 (2006), 213–228. `http://www.brucker.ch/bibliography/abstract/brucker.ea-semantic-2006-b`. – ISSN 1863–2122

[26] Cabot, J. ; Clarisó, R. ; Guerra, E. ; Lara, J.: An Invariant-Based Method for the Analysis of Declarative Model-to-Model Transformations. In: *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*. Berlin, Heidelberg : Springer-Verlag, 2008. – ISBN 978–3–540–87874–2, S. 37–52

[27] Cabot, J. ; Clarisó, R. ; Guerra, E. ; Lara, J. de: Verification and Validation of Declarative Model-to-Model Transformations Through Invariants. In: *Journal of Systems and Software* 83 (2010), Ferbuar, Nr. 2, S. 283–302

[28] Cabot, J. ; Clarisó, R. ; Riera, D.: Verification of UML/OCL Class Diagrams using Constraint Programming. In: *Model Driven Engineering, Verification, And Validation: Integrating Verification And Validation in MDE (MoDeVVA 2008)*, 2008

[29] Calleros, J. M. G. ; Stanciulescu, A. ; Vanderdonckt, J. ; Delacre, J. ; Winckler, M.: A Comparative Analysis of Graph Transformation Engines for User Interface Development. In: *Proceedings of the 4th International Workshop on Model-Driven Web Engineering, MDWE*, 2008, S. 16–30

[30] Card, S. ; Moran, T. P. ; Newell, A.: *The Psychology of Human-Computer Interaction.* London : Lawrence Erbaum Associates, 1983. – ISBN 0-89859-243-7

[31] Chang, D. ; Dooley, L. ; Tuovinen, J. E.: Gestalt theory in visual screen design: a new look at an old subject. In: *CRPIT '02: Proceedings of the Seventh world conference on computers in education conference on Computers in education: Australian topics.* Darlinghurst, Australia, Australia : Australian Computer Society, Inc., 2002. – ISBN 0–909925–86–0, S. 5–12

[32] Chok, S. S. ; Marriott, K.: Automatic construction of user interfaces from constraint multiset grammars. In: *Visual Languages, IEEE Symposium on* 0 (1995), S. 242. `http://dx.doi.org/10.1109/VL.1995.520815`. – DOI 10.1109/VL.1995.520815. – ISSN 1049–2615

[33] Compuware-Corporation ; SUN-Microsystems: *XMOF Queries, Views and Transformations on Models using MOF, OCL and Patterns.* OMG. `http://www.omg.org/docs/ad/03-08-07.pdf`. Version: August 2003. – OMG Document ad/2003-08-07

[34] Conte, S. D. ; Dunsmore, H. E. ; Shen, V. Y.: *Software engineering metrics and models.* Redwood City, CA, USA : Benjamin-Cummings Publishing Co., Inc., 1986. – ISBN 0–8053–2162–4

[35] Czarnecki, K. ; Helsen, S.: Classification of Model Transformation Approaches. In: Bettin, J. (Hrsg.) ; Emde Boas, G. van (Hrsg.) ; Agrawal, A. (Hrsg.) ; Willink, E. (Hrsg.) ; Bezivin, J. (Hrsg.): *2nd OOPSLA workshop on Generative Techniques in the Context of Model-driven Architecture*, ACM Press, Oktober 2003

[36] Devshed.com: *But which is better, Java or C#..* Internet, besucht 12.07.2010, Juni 2003. – http://forums.devshed.com/net-development-87/but-which-is-better-java-or-c-65293.html

[37] Ehrig, H. ; Ehrig, K. ; Habel, A. ; Pennemann, K.: Constraints and Application Conditions: From Graphs to High-Level Structures. In: *ICGT* Bd. 3256, Springer-Verlag, 2004, S. 287–303

[38] Ehrig, H. (Hrsg.) ; Engels, G. (Hrsg.) ; Kreowski, H.-J. (Hrsg.) ; Rozenberg, G. (Hrsg.): *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools.* River Edge, NJ, USA : World Scientific Publishing Co., Inc., 1999

[39] Ehrig, H. (Hrsg.) ; Kreowski, H.-J. (Hrsg.) ; Montanari, U. (Hrsg.) ; Rozenberg, G. (Hrsg.): *Handbook of graph grammars and computing by graph transformation: vol. 3: concurrency, parallelism, and distribution.* River Edge, NJ, USA : World Scientific Publishing Co., Inc., 1999 . – ISBN 9–810240–21–X

[40] El-Emam, K.: Object-Oriented Metrics: A Review of Theory and Practice / National Research Council Canada, Institute for Information Technology. 2001. – Forschungsbericht

[41] Favre, J.: Foundations of Meta-Pyramids: Languages vs. Metamodels – Episode II: Story of Thotus the Baboon. In: Bezivin, J. (Hrsg.) ; Heckel, R. (Hrsg.): *Language Engineering for Model-Driven Software Development.* Dagstuhl, Germany : Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005 (Dagstuhl Seminar Proceedings 04101). – ISSN 1862–4405

[42] Favre, J.: Foundations of Model (Driven) (Reverse) Engineering : Models – Episode I: Stories of The Fidus Papyrus and of The Solarus. In: Bezivin, J. (Hrsg.) ; Heckel, R. (Hrsg.): *Language Engineering for Model-Driven Software Development.* Dagstuhl, Germany : Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005 (Dagstuhl Seminar Proceedings 04101). – ISSN 1862–4405

[43] Feuerstack, S. ; Blumendorf, M. ; Schwartze, V. ; Albayrak, S.: Model-based layout generation. In: *AVI '08: Proceedings of the working conference on Advanced visual interfaces.* New York, NY, USA : ACM, 2008. – ISBN 0–978–60558–141–5, S. 217–224

[44] Fogarty, J. ; Hudson, S.: GADGET: A Toolkit for Optimization-Based Approaches to Interface and Display Generation. In: *Proceedings of the ACM*

Symposium on User Interface Software and Technology (UIST 2003), 2003, S. 125–134

[45] Fourer, R. ; Gay, D. M. ; Kernighan, B. W.: *AMPL: A Modeling Language for Mathematical Programming.* Brooks/Cole Publishing Company / Cengage Learning, 2002

[46] Fülöp, J.: Introduction to decision making methods. In: *Workshop on Biodiversity and Ecosystem Informatics.* Washington, 2004

[47] Gajos, K. Z.: *Automatically Generating Personalized User Interfaces*, University of Washington, Diss., 2008

[48] Gajos, K. Z. ; Weld, D. S.: SUPPLE: automatically generating user interfaces. In: *IUI '04: Proceedings of the 9th international conference on Intelligent user interfaces.* New York, NY, USA : ACM Press, 2004. – ISBN 1–58113–815–6, S. 93–100

[49] Gajos, K. Z. ; Weld, D. S. ; Wobbrock, J. O.: Decision-Theoretic User Interface Generation. In: *AAAI'08*, AAAI Press, 2008, S. 1532–1536

[50] Gajos, K. Z. ; Wobbrock, J. O. ; Weld, D. S.: Automatically generating user interfaces adapted to users' motor and vision capabilities. In: *UIST '07: Proceedings of the 20th annual ACM symposium on User interface software and technology.* New York, NY, USA : ACM, 2007. – ISBN 978–1–59593–679–2, S. 231–240

[51] *Kapitel* Defining and Validating Metrics for UML Class Diagrams. In: Genero, M. ; Poels, G. ; Manso, E. ; Piattini, M.: *Metrics for Software Conceptual Models.* Imperial College Press, 2005, S. 99–159

[52] Gerber, A. ; Lawley, M. ; Raymond, K. ; Steel, J. ; Wood, A.: Transformation: The Missing Link of MDA. In: *ICGT '02: Proceedings of the First International Conference on Graph Transformation.* London, UK : Springer-Verlag, 2002. – ISBN 3–540–44310–X, S. 90–105

[53] Geske, U. ; Wolf, A.: *Using Constraint Hierarchies and Dynamic Constraint Solving for Industrial Design Problems.* `http://citeseer.ist.psu.edu/701951.html`

[54] Giese, H. ; Wagner, R.: Incremental Model Synchronization with Triple Graph Grammars. In: Nierstrasz, O. (Hrsg.) ; Whittle, J. (Hrsg.) ; Harel, D. (Hrsg.) ; Reggio, G. (Hrsg.): *Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS),*

Genova, Italy Bd. 4199, Springer-Verlag, 10 2006 (Lecture Notes in Computer Science (LNCS)), S. 543–557

[55] Gogolla, M. ; Büttner, F. ; Richters, M.: USE: A UML-based specification environment for validating UML and OCL. In: *Science of Computer Programming* 69 (2007), Nr. 1-3, 27 - 34. http://dx.doi.org/10.1016/j.scico.2007.01.013. – DOI 10.1016/j.scico.2007.01.013. – ISSN 0167–6423. – Special issue on Experimental Software and Toolkits

[56] Golin, E. J. ; Reiss, S. P.: The specification of visual language syntax. In: *J. Vis. Lang. Comput.* 1 (1990), Nr. 2, S. 141–157. http://dx.doi.org/10.1016/S1045-926X(05)80013-8. – DOI 10.1016/S1045–926X(05)80013–8. – ISSN 1045–926X

[57] Green, T. R. G. ; Petre, M.: When Visual Programs are Harder to Read than Textual Programs. In: *Human-Computer Interaction: Tasks and Organisation, Proceedings ECCE-6 (6th European Conference Cognitive Ergonomics)*, 1992

[58] Greenyer, J. ; Kindler, E.: Reconciling TGGs with QVT. Version: September 2007. http://dx.doi.org/10.1007/978-3-540-75209-7_2. In: *Model Driven Engineering Languages and Systems*. Springer-Verlag, September 2007 (LNCS). – DOI 10.1007/978–3–540–75209–7_2, S. 16–30

[59] Gronback, R. C.: *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. 1. Addison-Wesley Professional, 2009. – 736 S. – ISBN: 0321534077

[60] Gyapay, S. ; Schmidt, A. ; Varró, D.: Joint Optimization and Reachability Analysis in Graph Transformation Systems with Time. In: *Electronic Notes in Theoretical Computer Science* 109 (2004), 137 –147. http://www.sciencedirect.com/science/article/B75H1-4F1H7PT-D/2/a5b2341f248b8aa35dfc4f77a41530c6

[61] Heckel, R. ; Küster, J. ; Taentzer, G.: Confluence of Typed Attributed Graph Transformation Systems. In: *ICGT '02: Proceedings of the First International Conference on Graph Transformation* Bd. 2505. London, UK : Springer-Verlag, 2002, S. 161 –176. – ISBN 3-540-44310-X

[62] Heinrichs, F. ; Steimle, J. ; Schreiber, D. ; Mühlhäuser, M.: Letras: An Architecture and Framework For Ubiquitous Pen-and-Paper Interaction.

In: *EICS '10: Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*. New York, NY, USA : ACM, Juni 2010

[63] Helm, R. ; Marroitt, K. ; Odersky, M.: Building visual language parsers. In: *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA : ACM, 1991. – ISBN 0–89791–383–3, S. 105–112

[64] *Kapitel* Standard, assessment, and text difficulty. In: Hiebert, E. H.: *What research has to say about reading instruction*. 3rd ed. International Reading Association, 2002, S. 337–369. – ISBN: 978-0872071773

[65] Hoare, C. A. R.: Communicating sequential processes. In: *Commun. ACM* 21 (1978), Nr. 8, S. 666–677. http://dx.doi.org/10.1145/359576.359585. – DOI 10.1145/359576.359585. – ISSN 0001–0782

[66] Holtz, N. M. ; Rasdorf, W. J.: An evaluation of programming languages and language features for engineering software development. In: *Engineering with Computers* 3 (1988), Nr. 4, S. 183 –199. http://dx.doi.org/10.1007/BF01202140. – DOI 10.1007/BF01202140

[67] Holzmann, G.: *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, 2003. – ISBN 0–321–22862–6

[68] Holzner, S.: *Eclipse*. 1. O'Reilly Media, 2004. – 317 S. – ISBN: 0596006411

[69] Hosobe, H.: A modular geometric constraint solver for user interface applications. In: *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*. New York, NY, USA : ACM Press, 2001. – ISBN 1–58113–438–X, S. 91–100

[70] Howatt, J.: A project-based approach to programming language evaluation. In: *SIGPLAN Not.* 30 (1995), Nr. 7, S. 37–40. http://dx.doi.org/10.1145/208639.208642. – DOI 10.1145/208639.208642. – ISSN 0362–1340

[71] Hower, W. ; Graf, W. H.: A bibliographical survey of constraint-based approaches to CAD, graphics, layout, visualization, and related topics. In: *Knowledge-Based Systems* 9 (1996), Nr. 7, 449 - 464. http://dx.doi.org/10.1016/S0950-7051(96)01055-6. – DOI 10.1016/S0950–7051(96)01055–6. – ISSN 0950–7051

[72] Hudson, S. E. ; Mohamed, S. P.: Interactive specification of flexible user interface displays. In: *ACM Trans. Inf. Syst.* 8 (1990), Nr. 3, S. 269–288.

http://dx.doi.org/10.1145/98188.98201. – DOI 10.1145/98188.98201. – ISSN 1046–8188

[73] IKV++ TECHNOLOGIES AG: *QVT medini*. Internet. http://www.ikv.de/ikv_movies/mediniQVT.swf. Version: 2008

[74] JÉZÉQUEL, J. ; DEFOUR, O. ; PLOUZEAU, N.: An MDA approach to tame component based software development. In: BOER, J. S. (Hrsg.): *Post Proceedings of Formal Methods for Components and Objects (FMCO'03)*, Springer-Verlag, 2004 (LNCS 3188)

[75] JOUAULT, F. ; KURTEV, I.: Transforming Models with ATL. In: *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*. Montego Bay, Jamaica, 2005

[76] JOUAULT, F. ; KURTEV, I.: On the Architectural Alignment of ATL and QVT. In: *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*. New York, NY, USA : ACM Press, 2006. – ISBN 1–59593–108–2, 1188–1195

[77] JUSSIEN, N. ; ROCHART, G. ; LORCA, X.: The CHOCO constraint programming solver. In: *CPAIOR'08 workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*. Paris, France, Juni 2008

[78] KENNEDY, J. ; EBERHART, R. C.: Particle Swarm Optimization. In: *Proceedings of the IEEE International conference on neural Networks*. Perth, Australia, 1995, S. 1942–1948

[79] KESSENTINI, M. ; SAHRAOUI, H. A. ; BOUKADOUM, M.: Model Transformation as an Optimization Problem. In: CZARNECKI, K. (Hrsg.) ; OBER, I. (Hrsg.) ; BRUEL, J. (Hrsg.) ; UHL, A. (Hrsg.) ; VÖLTER, M. (Hrsg.): *Proceedings of Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008* Bd. 5301, Springer-Verlag, September 2008 (Lecture Notes in Computer Science), S. 159–173

[80] KIERAS, D.: Using the keystroke-level model to estimate execution times / University of Michigan. Version: 2001. ftp://www.eecs.umich.edu/people/kieras/GOMS/KLM.pdf. 2001. – Forschungsbericht

[81] KITCHENHAM, B. A. ; PFLEEGER, S. L.: Principles of survey research part 3: constructing a survey instrument. In: *SIGSOFT Softw. Eng. Notes* 27 (2002), Nr. 2, S. 20–24. http://dx.doi.org/10.1145/511152.511155. – DOI 10.1145/511152.511155. – ISSN 0163–5948

[82] Kitchenham, B. ; Pfleeger, S. L.: Principles of survey research part 4: questionnaire evaluation. In: *SIGSOFT Softw. Eng. Notes* 27 (2002), Nr. 3, S. 20–23. http://dx.doi.org/10.1145/638574.638580. – DOI 10.1145/638574.638580. – ISSN 0163–5948

[83] Kitchenham, B. ; Pfleeger, S. L.: Principles of survey research: part 5: populations and samples. In: *SIGSOFT Softw. Eng. Notes* 27 (2002), Nr. 5, S. 17–20. http://dx.doi.org/10.1145/571681.571686. – DOI 10.1145/571681.571686. – ISSN 0163–5948

[84] Kitchenham, B. ; Pfleeger, S. L.: Principles of survey research part 6: data analysis. In: *SIGSOFT Softw. Eng. Notes* 28 (2003), Nr. 2, S. 24–27. http://dx.doi.org/10.1145/638750.638758. – DOI 10.1145/638750.638758. – ISSN 0163–5948

[85] Kitchenham, B. A. ; Pfleeger, S. L.: Principles of survey research part 2: designing a survey. In: *SIGSOFT Softw. Eng. Notes* 27 (2002), Nr. 1, S. 18–20. http://dx.doi.org/10.1145/566493.566495. – DOI 10.1145/566493.566495. – ISSN 0163–5948

[86] Knuth, D. E.: backus normal form vs. Backus Naur form. In: *Commun. ACM* 7 (1964), Nr. 12, S. 735–736. http://dx.doi.org/10.1145/355588.365140. – DOI 10.1145/355588.365140. – ISSN 0001–0782

[87] Koch, N.: Transformation techniques in the model-driven development process of UWE. In: *ICWE '06: Workshop proceedings of the sixth international conference on Web engineering*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–435–9, S. 3

[88] Kong, J. ; Zhang, K. ; Zeng, X.: Spatial graph grammars for graphical user interfaces. In: *ACM Trans. Comput.-Hum. Interact.* 13 (2006), Nr. 2, S. 268–307. http://dx.doi.org/10.1145/1165734.1165739. – DOI 10.1145/1165734.1165739. – ISSN 1073–0516

[89] Küster, J. M. ; Ryndina, K. ; Hauser, R.: A Systematic Approach to Designing Model Transformations / IBM Research GmbH. 2005 (RZ 3621). – Forschungsbericht

[90] Königs, A. ; Tratt, L. (Hrsg.): *Model Transformation with Triple Graph Grammars*. Website of Model Transformations in Practice Workshop. http://sosym.dcs.kcl.ac.uk/events/mtip05/. Version: Oktober 2005

[91] Küster, J.: Definition and validation of model transformations. In: *Software and Systems Modeling* 5 (2006), Nr. 3, S. 233 –259. http://dx.doi.org/10.1007/s10270-006-0018-8. – DOI 10.1007/s10270–006–0018–8

[92] Lara, J. de ; Vangheluwe, H.: Defining visual notations and their manipulation through meta-modelling and graph transformation. In: *Journal of Visual Languages & Computing* 15 (2004), Nr. 3-4, 309 - 330. http://dx.doi.org/10.1016/j.jvlc.2004.01.005. – DOI 10.1016/j.jvlc.2004.01.005. – ISSN 1045–926X. – Domain-Specific Modeling with Visual Languages

[93] Limbourg, Q. ; Vanderdonckt, J. ; Michotte, B. ; Bouillon, L. ; Florins, M. ; Trevisan, D.: UsiXML: A User Interface Description Language for Context-Sensitive User Interfaces. In: Luyten, K. (Hrsg.) ; Abrams, M. (Hrsg.) ; Limbourg, Q. (Hrsg.) ; Vanderdonckt, J. (Hrsg.): *Proceedings of the ACM AVI'2004 Workshop "Developing User Interfaces with XML: Advances on User Interface Description Languages" (Gallipoli, May 25, 2004)*, 2004, 55-62

[94] Limbourg, Q.: *Multi-Path Development of Multimodal Applications*, Université Catolique de Louvin, Diss., 2004

[95] Limbourg, Q. ; Vanderdonckt, J.: Addressing the mapping problem in user interface design with UsiXML. In: *TAMODIA '04: Proceedings of the 3rd annual conference on Task models and diagrams*. New York, NY, USA : ACM, 2004. – ISBN 1–59593–000–0, S. 155–163

[96] Liu, Z. ; Mencl, V. ; Ravn, A. P. ; Yang, L.: Harnessing Theories for Tool Support. In: *ISOLA '06: Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. Washington, DC, USA : IEEE Computer Society, 2006. – ISBN 978–0–7695–3071–0, S. 371–382

[97] Ludewig, J.: Models in software engineering – an introduction. In: *Models in software engineering – an introduction* 2 (2003), S. 5–14

[98] Lutteroth, C. ; Strandh, R. ; Weber, G.: Domain Specific High-Level Constraints for User Interface Layout. In: *Constraints* 13 (2008), Nr. 3, S. 307 –342. http://dx.doi.org/10.1007/s10601-008-9043-2. – DOI 10.1007/s10601–008–9043–2

[99] MacKenzie, I. S. ; Buxton, W.: Extending Fitts' law to two-dimensional tasks. In: *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA : ACM, 1992. – ISBN 0–89791–513–5, S. 219–226

[100] Marriott, K. ; Sen Chok, S.: QOCA: A Constraint Solving Toolkit for Interactive Graphical Applications. In: *Constraints* 7 (2002), Nr. 3, S. 229 –254. http://dx.doi.org/10.1023/A:1020513316058. – DOI 10.1023/A:1020513316058

[101] Mattern, F. ; Flörkemeier, C.: Vom Internet der Computer zum Internet der Dinge. In: *Informatik-Spektrum* 33 (2010), Nr. 2, S. 107–121. http://dx.doi.org/10.1007/s00287-010-0417-7. – DOI 10.1007/s00287–010–0417–7

[102] Mens, T.: On the use of graph transformations for model refactoring. In: Saraiva, J. (Hrsg.) ; Lammel, J. V. R. (Hrsg.): *Generative and transformational techniques in software engineering*, Springer-Verlag, 2006, S. 219–257

[103] Montanari, U. ; Rossi, F.: Graph rewriting, constraint solving and tiles for coordinating distributed systems. Applied Categorical Structures. In: *Applied Categorical Structures* 7 (1999), S. 7–333

[104] Myers, B. ; McDaniel, R. ; Miller, R. ; Ferrency, A. ; Faulring, A. ; Kyle, B. ; Mickish, A. ; Klimovitski, A. ; Doane, P.: The Amulet environment: new models for effective user interface software development. In: *Software Engineering, IEEE Transactions on* 23 (1997), Juni, Nr. 6, S. 347–365. http://dx.doi.org/10.1109/32.601073. – DOI 10.1109/32.601073. – ISSN 0098–5589

[105] Myers, B. ; Hudson, S. E. ; Pausch, R.: Past, present, and future of user interface software tools. In: *ACM Trans. Comput.-Hum. Interact.* 7 (2000), Nr. 1, S. 3–28. http://dx.doi.org/10.1145/344949.344959. – DOI 10.1145/344949.344959. – ISSN 1073–0516

[106] Myers, B. A.: The garnet user interface development environment. In: *CHI '94: Conference companion on Human factors in computing systems*. New York, NY, USA : ACM, 1994. – ISBN 0–89791–651–4, S. 25–26

[107] Nareyek, A. ; Freuder, E. C. ; Fourer, R. ; Giunchiglia, E. ; Goldman, R. P. ; Kautz, H. ; Rintanen, J. ; Tate, A.: Constraints and AI Planning. In: *IEEE Intelligent Systems* 20 (2005), Nr. 2, S. 62–72. http://dx.doi.org/10.1109/MIS.2005.25. – DOI 10.1109/MIS.2005.25. – ISSN 1541–1672

[108] Neary, D. ; Woodward, M.: An Experiment to Compare the Comprehensibility of Textual and Visual Forms of Algebraic Specifications. In: *Journal of Visual Languages & Computing* 13 (2002), Nr. 2, 149 - 175. http:

//dx.doi.org/10.1006/jvlc.2001.0213. – DOI 10.1006/jvlc.2001.0213. – ISSN 1045–926X

[109] Nelson, M. L.: Considerations in choosing a concurrent/distributed object-oriented programming language. In: *SIGPLAN Not.* 29 (1994), Nr. 12, S. 66–71. http://dx.doi.org/10.1145/193209.193223. – DOI 10.1145/193209.193223. – ISSN 0362–1340

[110] Nichols, J. ; Myers, B. A.: Creating a lightweight user interface description language: An overview and analysis of the personal universal controller project. In: *ACM Trans. Comput.-Hum. Interact.* 16 (2009), Nr. 4, S. 1–37. http://dx.doi.org/10.1145/1614390.1614392. – DOI 10.1145/1614390.1614392. – ISSN 1073–0516

[111] Nolte, S.: *QVT - Relations Language.* Springer-Verlag, 2009. – ISBN: 978-3-540-92170-7

[112] Olsen, D. R.: Evaluating user interface systems research. In: *UIST '07: Proceedings of the 20th annual ACM symposium on User interface software and technology.* New York, NY, USA : ACM, 2007. – ISBN 978–1–59593–679–2, S. 251–258

[113] OMG: *Meta Object Facility 2.0 Core Final Adopted Specification.* OMG, Oktober 2003

[114] OMG: *Unified Modeling Language 2.0 Infrastructure Final Adopted Specification.* September 2003. – ptc/03-09-15

[115] OMG: *MOF 2.0 Query / Views / Transformations RFP.* April 2004

[116] OMG: *Object Constraint Language OMG Available Specification Version 2.0.* OMG, Mai 2006

[117] OMG: *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification.* OMG, July 2007. – ptc/07-07-07

[118] OMG, J. M. J. Miller M. J. Miller: *MDA Guide Version 1.0.1.* OMG, Juni 2003. – document number: omg/2003-06-01

[119] Oulasvirta, A.: FEATURE When users "do" the Ubicomp. In: *interactions* 15 (2008), Nr. 2, S. 6–9. http://dx.doi.org/10.1145/1340961.1340963. – DOI 10.1145/1340961.1340963. – ISSN 1072–5520

[120] Paternò, F. ; Santoro, C. ; Spano, L. D.: MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. In: *ACM Trans. Comput.-Hum. Interact.* 16 (2009), Nr. 4, S. 1–30. http://dx.doi.org/10.1145/1614390.1614394. – DOI 10.1145/1614390.1614394. – ISSN 1073–0516

[121] Paternò, F. ; Santoro, C.: One Model, Many Interfaces. In: *Proceedings of CADUI 2002*. Valenciennes, France, Mai 2002

[122] Perry, D. E. ; Porter, A. A. ; Votta, L. G.: Empirical studies of software engineering: a roadmap. In: *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*. New York, NY, USA : ACM, 2000. – ISBN 1–58113–253–0, S. 345–355

[123] Petre, M. ; Green, T. R. G.: Learning to Read Graphics: Some Evidence that 'Seeing' an Information Display is an Acquired Skill. In: *Journal of Visual Languages & Computing* 4 (1993), Nr. 1, 55 - 70. http://dx.doi.org/10.1006/jvlc.1993.1004. – DOI 10.1006/jvlc.1993.1004. – ISSN 1045–926X

[124] Petter, A. ; Behring, A. ; Steinmetz, J.: Efficient Modelling of Highly Adaptive UbiComp Applications. In: Kortuem, G. (Hrsg.): *Workshop on Software Engineering Challenges for Ubiquitous Computing*, 2006, S. 47–48

[125] Petter, A. ; Behring, A. ; Zlatkov, M. ; Steinmetz, J. ; Mühlhäuser, M.: Modeling Usability in Model-Transformations. In: Bošković, M. (Hrsg.) ; Gašević, D. (Hrsg.) ; Pahl, C. (Hrsg.) ; Schätz, B. (Hrsg.): *Proceedings of the 1st International Workshop on Non-functional System Properties in Domain Specific Modeling Languages, NFPinDSML-2008* Bd. 394, CEUR, September 2008. – ISSN 1613-0073

[126] Petter, A. ; Borgert, S. ; Aitenbichler, E. ; Behring, A. ; Mühlhäuser, M.: Optimizing non-functional Properties of a Service Composition using a Declarative Model-to-Model Transformation. In: *Acta Universitates Apulensis* 18 (2009), S. 15

[127] Petter, A. ; Zlatkov, M. ; Behring, A.: The Solverational Grammar and a Set of Evaluation Results / Technische Universität Darmstadt. Hochschulstr. 10, 64289 Darmstadt, Januar 2010. – Forschungsbericht. – ISSN 1864–0516

[128] Peuquet, D. J. ; Ci-Xiang, Z.: An algorithm to determine the directional relationship between arbitrarily-shaped polygons in the plane. In: *Pattern Recognition* 20 (1987), Nr. 1, 65 - 74. http://dx.doi.org/10.1016/

0031-3203(87)90018-5. – DOI 10.1016/0031–3203(87)90018–5. – ISSN 0031–3203

[129] Pfleeger, S. L. ; Kitchenham, B. A.: Principles of survey research: part 1: turning lemons into lemonade. In: *SIGSOFT Softw. Eng. Notes* 26 (2001), Nr. 6, S. 16–18. http://dx.doi.org/10.1145/505532.505535. – DOI 10.1145/505532.505535. – ISSN 0163–5948

[130] Pirkul, H. ; Gupta, R. ; Rolland, E.: VisOpt: a visual interactive optimization tool for P-median problems. In: *Decision Support Systems* 26 (1999), September, Nr. 3, S. 209–223

[131] Punter, T. ; Ciolkowski, M. ; Freimut, B. ; John, I.: Conducting online surveys in software engineering. In: *Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 International Symposium on*, 2003, S. 80–88

[132] Rekers, J. ; Schürr, A.: Defining and Parsing Visual Languages with Layered Graph Grammars. In: *Journal of Visual Languages and Computing* 8 (1997), S. 27–55

[133] *Kapitel* Measuring OCL Expressions: an Approach Based on Cognitive Techniques. In: Reynoso, L. ; Genero, M. ; Piattini, M.: *Metrics for Software Conceptual Models*. Imperial College Press, 2005, S. 161–206

[134] Roelofs, M.: *AIMMS 3.9 - User's Guide*. Lulu.com, 2009. – ISBN 0557063604, 9780557063604

[135] Royce, W.: Managing the development of large software systems. In: *IEEE WESCON*, 1970, S. 1–9

[136] Rozenberg, G. (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997 . – ISBN 9810228848

[137] *Kapitel* Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching. In: Rudolf, M.: *Lecture Notes in Computer Science*. Bd. 1764/2000,: *Theory and Application of Graph Transformations*. Springer-Verlag, 2000, 381 –394

[138] Ruttkay, Z.: Fuzzy constraint satisfaction. In: *Proceedings of the 3rd IEEE Conference on Fuzzy Systems* Bd. 2. Orlando, Florida : IEEE, 1994, S. 1263–1268

[139] Ruys, T. C.: Optimal scheduling using branch and bound with SPIN 4.0. In: *In Proceedings of SPIN-03*, Springer-Verlag, 2003, S. 1–17

[140] Sannella, M. ; Maloney, J. ; Freeman-Benson, B. ; Borning, A.: Multi-way versus one-way constraints in user interfaces: experience with the DeltaBlue algorithm. In: *Softw. Pract. Exper.* 23 (1993), Nr. 5, S. 529–566. – ISSN 0038–0644

[141] Schaefer, R.: A Survey on Transformation Tools for Model Based User Interface Development. In: Jacko, J. A. (Hrsg.): *HCI (1)* Bd. 4550, Springer-Verlag, 2007 (Lecture Notes in Computer Science). – ISBN 978–3–540–73104–7, S. 1178–1187

[142] Schätz, B.: Formalization and Rule-Based Transformation of EMF Ecore-Based Models. In: *Proceedings of the 1st International Consference on Software Language Engineering*, 2008

[143] Schedlbauer, M.: Completion Time Predictions of Mobile Touch-Screen Interactions in Dual-Task Situations. In: *Proceedings of the ITI 2007 29th Int. Conf. in Information Technology Interfaces*. Cavtat, Croatia, Juni 2007

[144] Schmitz, C.: *LimeSurvey - the open source survey appliation*. Internet, Oktober 2009. – visited 11.03.2010

[145] Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: *WG '94: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science* Bd. 903/1995. London, UK : Springer-Verlag, 1995 (Lecture Notes in Computer Science). – ISBN 3–540–59071–4, S. 151–163

[146] Sears, A.: Layout Appropriateness: A Metric for Evaluating User Interface Widget Layout. In: *IEEE Trans. Softw. Eng.* 19 (1993), Nr. 7, S. 707–719. `http://dx.doi.org/10.1109/32.238571`. – DOI 10.1109/32.238571. – ISSN 0098–5589

[147] Silva, P. P.: User Interface Declarative Models and Development Environments: A Survey. In: Palanque, P. (Hrsg.) ; Paternò, F. (Hrsg.): *Proceedings of DSV-IS2000* Bd. 1946. Limerick, Ireland : Springer-Verlag, Juni 2000 (LNCS), 207–226

[148] SoKNOS-Konsortium: *Service-orientierte ArchiteKturen zur Unterstützung von Netzwerken im Rahmen Oeffentlicher Sicherheit*. Internet, 2009. – http://www.soknos.de, visited 08.07.2010

[149] Sottet, J.-S. ; Calvary, G. ; Favre, J.-M.: Towards Model Driven Engineering of Plastic User Interfaces. In: Pleuss, A. (Hrsg.) ; Van den Bergh, J. (Hrsg.) ; Hussmann, H. (Hrsg.) ; Sauer, S. (Hrsg.): *Proceedings of Model Driven Design of Advanced User Interfaces 2005* Bd. 159. Montego Bay, Jamaica : online CEUR-WS.org/Vol-159/paper5.pdf, Oktober 2005 (CEUR Workshop Proceedings). – ISSN 1613–0073

[150] Sottet, J.-S. ; Calvary, G. ; Favre, J.-M.: Models at Run-time for Sustaining User Interface Plasticity. In: *Proceedings of the Workshop Models at Run Time*. Glasgow, Oktober 2006

[151] Stachowiak, H.: *Allgemeine Modelltheorie*. Book, 1973

[152] Stanciulescu, A.: *A Transformational Approach for Developing Multimodal Web User Interfaces*, UNIVERSITE CATHOLIQUE DE LOUVAIN SCHOOL OF MANAGEMENT BELGIAN LABORATORY OF COMPUTER-HUMAN INTERACTION, Diss., 2006

[153] Steimle, J.: *Integrating Printed and Digital Documents: Interaction Models and Techniques for Collaborative Knowledge Work*, Technische Universität Darmstadt, Diss., 2009

[154] Steinberg, D. ; Budinsky, F. ; Paternostro, M. ; Merks, E. ; Gamma, E. (Hrsg.) ; Nackman, L. (Hrsg.) ; Wiegand, J. (Hrsg.): *EMF: Eclipse Modeling Framework*. 2. Amsterdam : Addison-Wesley, 2009. – 744 S. – ISBN : 0321331885

[155] Sutherland, I. E.: Sketch pad a man-machine graphical communication system. In: *DAC '64: Proceedings of the SHARE design automation workshop*. New York, NY, USA : ACM Press, 1964, S. 6.329–6.346

[156] Tamassia, R.: Constraints in Graph Drawing Algorithms. In: *Constraints* 3 (1998), Nr. 1, S. 87 –120. `http://dx.doi.org/10.1023/A:1009760732249`. – DOI 10.1023/A:1009760732249

[157] Thevenin, D.: *Adaptation en Interaction Homme-Machine : le cas de la Plasticité*, Université de Grenoble, Diss., 2001

[158] Trapp, M. ; Schmettow, M.: Consistency in use through Model based User Interface Development. In: *Proceedings of CHI 2006 Workshop "The Many Faces of Consistency in Cross-Platform Design"*, 2006

[159] Tratt, L.: Model transformations and tool integration. In: *Software and Systems Modeling* 4 (2005), Mai, Nr. 2, S. 112–122. `http://dx.doi.org/10.1007/s10270-004-0070-1`. – DOI 10.1007/s10270–004–0070–1

[160] Tsang, E. P. K.: *Foundations of Constraint Satisfaction.* Academic Press, Inc., 1993. – ISBN 0-12-701610-4

[161] Van Hentenryck, P.: *The OPL optimization programming language.* Cambridge, MA, USA : MIT Press, 1999. – ISBN 0–262–72030–2

[162] Vanthienen, J. ; Poelmans, S.: A general framework for positioning, evaluating and selecting the new generation of development tools. In: *EUROMICRO 96. 'Beyond 2000: Hardware and Software Design Strategies'., Proceedings of the 22nd EUROMICRO Conference*, 1996, S. 233–240

[163] Varró-Gyapay, S. ; Varró, D.: Optimization in graph transformation systems using Petri net based techniques. In: *Workshop on Petri Nets and Graph Transformations, PNGT06*, 2006

[164] Weiser, M.: The Computer for the 21st Century. In: *Scientific American* 265 (1991), Nr. 9, S. 66–75,

[165] White, J. ; Gray, J. ; Schmidt, D. C.: Constraint-based Model Weaving. In: *Transactions on Aspect-Oriented Software Development* open (2009), S. open. – to appear

[166] Wikipedia: *Fitt's law.* Internet. `http://en.wikipedia.org/wiki/Fitts%27s_law`. Version: März 2010. – visited 8.3.10

[167] Winkelmann, J. ; Taentzer, G. ; Ehrig, K. ; Küster, J. M.: Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars. In: *Electr. Notes Theor. Comput. Sci.* 211 (2008), S. 159–170

[168] Winkler, M. ; Heinrich, M. ; Behring, A. ; Steinmetz, J. ; Dargie, W.: EMODE – ein Ansatz zur werkzeugunterstützten Modellierung multimodaler, adaptiver Benutzerschnittstellen. In: *Workshop Modellbasierte Entwicklung von Benutzerschnittstellen, 37. Jahrestagung der Gesellschaft für Informatik.* Bremen, 2007

[169] Winter, S. ; Wagner, S. ; Deissenboeck, F.: A Comprehensive Model of Usability. In: *Proc. Engineering Interactive Systems 2007.* Salamanca, Spain : Springer-Verlag, 2007

[170] *Kapitel* Empirical Research Methods in Software Engineering. In: Wohlin, C. ; Höst, M. ; Henningsson, K.: *Empirical Methods and Studies in Software Engineering.* Bd. 2765. Springer-Verlag, 2003, 7–23

[171] Yanagida, T. ; Nonaka, H.: Flexible Widget Layout Formulated as Fuzzy Constraint Satisfaction Problem. In: *New Advances in Intelligent Decision Technologies*, Springer-Verlag, 2009, S. 73 –83

[172] Yang, Y. ; Klemmer, S. R.: Aesthetics matter: leveraging design heuristics to synthesize visually satisfying handheld interfaces. In: *CHI EA '09: Proceedings of the 27th international conference extended abstracts on Human factors in computing systems*. New York, NY, USA : ACM, 2009. – ISBN 978–1–60558–247–4, S. 4183–4188

[173] Zhang, D.-Q. ; Zhang, K. ; Cao, J.: A Context-sensitive Graph Grammar Formalism for the Specification of Visual Languages. In: *The Computer Journal* 44 (2001), Nr. 3, S. 186–200. http://dx.doi.org/10.1093/comjnl/44.3.186. – DOI 10.1093/comjnl/44.3.186

[174] Zinssmeister, G. ; McCreary, C.: Drawing graphs with attribute graph grammars. In: *Graph Grammars and Their Application to Computer Science* 1073 (1996), S. 443 –453. http://dx.doi.org/10.1007/3-540-61228-9-104. – DOI 10.1007/3–540–61228–9–104

S. D. G.

Wissenschaftlicher Werdegang des Verfassers[1]

1997-2002	Abschluß des Studiums der Informatik an der Technischen Universität Darmstadt als Dipl.-Inform.
2006-2008	Wiss. Mitarbeiter am Fachgebiet Telekooperation des Fachbereichs Informatik der Technischen Universität Darmstadt, ITEA/BMBF-Projekt EMODE
2008-2010	Wiss. Mitarbeiter am Fachgebiet Telekooperation des Fachbereichs Informatik der Technischen Universität Darmstadt, BMBF-Projekt SoKNOS
2010-2012	Researcher ICT, SEEBURGER AG, Projekte Theseus / B2B In The Cloud, Software Cluster / InDiNet und Trusted Cloud / PeerEnergyCloud

[1] gemäß §20 Abs. 3 der Promotionsordnung der Technischen Universität Darmstadt