Universität Leipzig
Institut für Informatik
Abteilung Betriebliche Informationssysteme (BIS)

Hausarbeit zum Thema

Abbildung von Constraints für Benutzeroberflächen

Betreuer: Hr. Johannes Schmidt
Bearbeiter: Hans-Georg Schladitz

Matr.-Nr.: 2137652
4. Semester

Eingereicht am: 29.06.2016

Abbildung von Constraints für Benutzeroberflächen

Hans-Georg Schladitz

Abstract

UML als Standard ist ein weit verbreiteter und in der Praxis häufig eingesetzter Ansatz zur Modellierung von Spezifikation, Konstruktion, Dokumentation von Software und Systemen. Das Klassendiagramm als bekanntester Vertreter von UML und gleichzeitig als Ausgangsmodelltyp der modellgetriebenen Softwareentwicklung, eignet sich auf Grund der hohen Abstrahierbarkeit auch zur Modellierung von Benutzeroberflächen. Während die Struktur und Beziehungen einer Software leicht über Modelle und Generatoren zum lauffähigen Programmen gebracht werden können, verhält es sich mit der Definition von Einschränkungen (Constraints) wegen der hohen Ausdrucksmächtigkeit und der Notwendigkeit der detaillierten Beschreibung, anders. Ziel dieser Arbeit ist es mögliche Ansätze (Frameworks und Konzepte) zu finden, um Constraints der Art zu definieren, dass aus ihnen, zusammen mit UML-Klassendiagrammen, ausführbarer Code generiert werden kann. Grundlage hierfür ist ein kurzer theoretischer Überblick über UML, Constraints und ihre Zusammenhänge. Außerdem soll ein Ansatz genauer untersucht und als Contept of Proof beispielhaft umgesetzt werden.

Schlüsselwörter

Constraint, OCL, UML, DSL, MDSD, MDE, Xtext

Inhaltsverzeichnis

Inhaltsverzeichnis III
Abbildungsverzeichnis IV
Abkürzungsverzeichnis IV
1 Einleitung 1
2 Theorie 1
3 Analyse 3
3.1 Object Constraint Language 3
3.2 UML-Profile 3
3.3 Domain Specific Language 4
4 Concept of Proof Implementation 6
5 Zusammenfassung 9
Literaturverzeichnis V

Abbildungsverzeichnis

Abbildung 1: Kompositionsbeziehung mit Multiplizitäten 2
Abbildung 2: Struktur bzw. Grammatik der DSL 7
Abbildung 3: Beispielmodell zur Definition von Constraints mit der UML-DSL 7
Abbildung 4: UML-Klassendiagramm Beispielmodell 8
Abbildung 5: Ausgeführtes Beispielformular aus Generat 8
Abbildung 6: Ausschnitt aus dem Codegenerator 9

Abkürzungsverzeichnis

ANTLR	Another Tool for Language Recognition
AST	Abstract Syntax Tree
DSL	Domain-Specific Language
EBNF	Erweiterte Backus Naur Form
EMF	Eclipse Modeling Framework
EMOF	Essential Meta Object Facility
EPL	Eclipse Public License
GPL	General Purpose Language
IDE	Integrated Development Environment
JVM	Java Virtual Machine
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MDWE	Model Driven Web Engineering
MDSD	Model Driven Software Development
MWE	Modeling Workflow Engine
MOF	Meta Object Facility
UI	User Interface
UML	Unified Modeling Language
UWE	UML-Based Web Engineering
OCL	Object Contraint Language
OMG	Object Management Group
oAW	openArchitectureWare
XMI	XML Metadata Interchange

1 Einleitung

Mit dem Ziel UML-Klassendiagramme als Ausgangsbasis zur Erstellung von Benutzeroberflächen wie z.B. Eingabeformulare zu verwenden, bedarf es einiger theoretischer Grundlagen. Das Klassendiagramm ist ein UML-Model. D.h. es ist ein Standard der OMG. Über Klassen und Beziehungen können Objekte aus der realen Welt (von Systemen) abstrahiert und abgebildet werden. So kann die Komplexität von Systemen, wie z.B. Software, reduziert werden, was die Analyse der Systeme oder die Entwicklung solcher erleichtert. Ein weiterer Verwendungszweck für das Klassendiagramm als Modell ist das Modell Driven Engineering (MDE). Hierbei geht es nach [Kraus et al 2007, 1], neben dem normalen Transformieren und Ausarbeiten von Modellen auch um die Erstellung von Software. Das Ziel des Model Driven Software Development (MDSD) ist ebenfalls die direkte Ausführung von Modellen. UML-Modelle eigenen sich wegen der hohen Verbreitung und der ausgeprägten Toolunterstützung. Aus diesem Grund hat sich unter anderem das UML-based Web Engineering (UWE) etabliert. UWE basiert auf Model Driven Web Engineering (MDWE), welches auf MDE beruht und sich auf Web-Systeme konzentriert und sich demzufolge auf webnahe Technologien bezieht. Der Vorteil von MDSD und MDE liegt in der Trennung der Aufgabenbereiche. Modellierer kümmern sich um ihren Problemraum - den Modellen - wohingegen Programmierer sich auf den Lösungsraum - den Code bzw. dem Generator - konzentrieren können. Ebenfalls wird eine Trennung von der Funktionalität des Systems und den Implementierungsdetails geschaffen. Auf die strikte Trennung von Technologie und Funktionalität beruht das Model Driven Architecture (MDA) (vgl. [OMG 2014, 9]). MDA ist ein Mittel zur Implementierung von MDSD und MDE.

2 Theorie

Im Mittelpunkt steht in dieser Arbeit der Modelltyp der UML-Klassendiagramm. Eine Möglichkeit MDSD umzusetzen ist das Eclipse Modeling Framework (EMF). Es ist ein Java-Tool zur Generierung von Code, dass sich auf Klassendiagramme beschränkt. Dem EMF liegt das Ecore-Metamodell zu Grunde. Dadurch ist es möglich Objektmodelle mit unterschiedlicher Repräsentation zu importieren, indem sie in die kanonische Ecore-Form konvertiert werden. Die kanonische Ecore-Form lässt sich wiederrum in das Standard-Format „XML Metadata Interchange“ (XMI) von OMG transformieren. Dadurch schlägt es die Brücke zum MetaObject Facility (MOF), dem Metamodell von UML. Ecore und MOF sind beides Metamodelle die sich selbst definieren (vgl. [OMG 2015b, 9] und [Kuhn 2008]). Dadurch und wegen ihrer Ausdrucksmächtigkeit eignen sich beider für die Erstellung eigener bzw. neuer Modellierungssprachen. Besonderen Fokus hierbei haben domänenspezifische Sprachen (DSL). Eine DSL ist eine Sprache zur einfachen Darstellung von Sachverhalten aus einer bestimmten Domäne für bestimmte Personen (Domänenexperten), die sich in diesem Gebiet auskennen. Dieses Wissen fehlt dem Programmierer der Software wegen der unterschiedlicher Domänenpriorisierung. Im Sinne des MDSD können mittels Generatoren aus Modellen einer DSL, Generatoren erstellt werden, welche aus den Informationen der Modellen Code generieren. Der Vorteil hierbei liegt in der Trennung zwischen der fachlichen und technischen Sicht. Der Programmierer braucht nur die Generatoren für den jeweiligen Modelltyp zu erstellen. Dadurch reduziert sich die Programmierzeit und die Programmierfehler. Der Domänenexperte kann ohne Programmierkenntnisse sein Expertenwissen in Modellen übertragen. Per Kopfdruck wird aus den Modellen, abhängig von dem erstellten Generator, entsprechender Quellcode einer bestimmten Programmiersprache generiert. Das UML Klassendiagramm ist genau betrachtet bereits eine DSL zur Beschreibung von Anwendungssystemen. Über Klassen, die Konzepte

der realen Welt abbilden und Beziehungen zwischen diesen, wird auf konzeptioneller Ebene, die Struktur von domänenabhängigen Systemen definiert. Hierbei können über die Arten der Beziehungen konkrete Aussagen über die Konzepte getroffen werden. So ist es beispielsweise möglich über die Vererbungsbeziehung (Generalization) zwischen Klassen komplette Klassenhierarchien zu beschreiben. Klassen selbst haben neben ihren Namen auch Eigenschaften (Attributes) und ein Verhalten (Behavior). Das Verhalten wird über Methoden zum Ausdruck gebracht wird. Neben der Vererbungsbeziehung gibt es noch die Enthaltenseinbeziehung (Aggregation, Composition), Abhängigkeitenbeziehung (Dependency) und die normale Assoziation (Association).

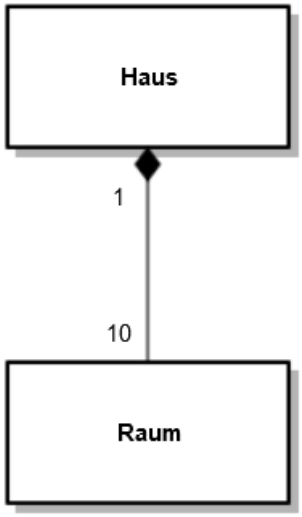

Abbildung 1: Kompositionsbeziehung mit Multiplizitäten

Grundform aller Beziehungen ist die normale binäre Assoziation (vgl. [OMG 2015, 200]). Sie ist definiert durch einen Anfang und ein Ende. An den Endpunkten sind jeweils Klassen referenziert. Dadurch ist es wie in Abb. 1 gezeigt möglich, mittels Intervallen (Multiplizitäten), Wertigkeiten der Beziehung zwischen diesen Klassen zu beschreiben. Dies entspricht einer Einschränkung und damit einer Art von Zusicherung (Constraint), da eine Ober- bzw. Untergrenze die Anzahl der Möglichkeiten begrenzt.

Es werden sechs Arten von Contraints unterschieden:

- Invariants,
- Pre-/Postconditions,
- Initial & derived Values,
- Definitions,
- Body Definitions und
- Guards.

Invariants sind Zusicherungen, die zu jeder Zeit für eine Instanz oder Assoziation gelten. Im Fall der Multiplizität heißt dies am Beispiel der Abb. 1, dass ein Haus in diesem Fall genau 10 Zimmer enthalten muss. Die Vor- bzw. Nachbedingung müssen zur Ausführung einer bestimmten Funktion davor bzw. danach gelten. Initial & derived Values beschreiben Bedingungen für Ausgangs- und abgeleitete Werte. So kann z.B. festgelegt werden, dass innerhalb eines Formular-Klassendiagramm, das Attribute „Value“ vom Objekt mit dem Namen „name“ des Typs „Textarea“ mit dem Startwert „John Doe“ initialisiert wird. UML bietet bereits die Möglichkeit Default-Values zu beschreiben, wodurch eine extra Definition über Constraints nicht nötig ist. Über ‚Definitions‘ können Attribute und Operationen

definiert werden, die nicht im Klassendiagramm enthalten sind. Body Definitions beschreiben die Art der Operation. Mit z.B. „isQuery = true“ wird eine Operation zur Abfrageoperation. Guards sind Zusicherung, die bei einem Zustandsübergang gelten müssen. In dieser Arbeit wird, aufgrund des kleingehaltenen Rahmens, der Focus auf Invarianten gelegt. Invarianten werden von dem UML-Klassendiagramm bereits teilweise unterstützt. D.h. sie können über die UML-Sprachelemente ausgedrückt werden.

3 Analyse

Grundsätzlich gibt es drei Möglichkeiten um aus UML-Klassendiagrammen zusammen mit definierten Constraints Code zu generieren. Diese sind:

- OCL,
- UML-Profiles und
- DSLs.

Im Folgenden werden die Ansätze untersucht und geprüft ob sie für den Zweck der Constraintdefinition gestützte Benutzeroberflächengenerierung geeignet sind. Als Grundlage für weitere Betrachtungen dient ein Framework zu Modellierung von UML-Modellen. Hierfür wurde Papyrus gewählt, weil es Open Soruce und weit verbreitet ist, sowie dem UML-Standard konforme UML-Files erstellen kann. Papyrus ist eine Modellierungsumgebung die auf der Entwicklungsumgebung (IDE) Eclipse basiert. Aus diesen Grund können die erstellten UML-Klassendiagramme auch als Ecore konforme Modellartefakte gespeichert werden. Aktuell wird der UML 2.5.0 und OCL 2.3.1 Standard unterstützt.

3.1 Object Constraint Language

Eine Sprache zur Beschreibung von Constraings für UML-Klassendiagramme ist OCL.
OCL erweitert UML um die Möglichkeit zusätzliche Randbedingungen beschreiben zu können. OCL-Ausdrücke sind widerspruchsfrei und können darum von Programmen verarbeitet werden. Bei der Codegenerierung von einem UML-Klassendiagramm zu Programmcode, dient OCL der Überwachung. Mittels OCL wird demnach überprüft, ob die definierten Constraints beim Generieren von Code erfüllt sind. Weitere Beispiele die mittels OCL in Form von Constraints definiert werden können sind Wertebreiche oder einzuhaltende Restriktionen zwischen Objekten. Beispiel hierfür ist das Alter einer Person, der nicht negativ sein darf, oder die Bedingung, dass eine Person jünger sein muss als die Eltern. Problematisch bei OCL ist die Komplexität. Aufgrund der Ausdrucksmächtigkeit der Sprache können Constraints sehr detailliert beschrieben werden. Dies führt bei der bestehenden schweren Syntax oft zu einem hohen Zeitaufwand und erfordert hinreichendes Know-how zu OCL, sowie zur Domäne die es abzubilden gilt. Jedoch, führt der Einsatz von OCL zu einem höheren Generierungsgrad. Aufgrund der Komplexität und der schwachen Toolunterstützung ist OCL für die Lösung der Aufgabenstellung nicht hinreichend geeignet.

3.2 UML-Profile

UML-Profile werden in der Praxis häufig eingesetzt um UML zu erweitern, damit auch nicht abgedeckte Anforderungen erfüllt werden können. Hierfür werden Stereotypen genutzt. Im Metamodell werden diese mittels Erweiterungsbeziehungen definiert. Da hier in der Metaebene agiert wird, muss ein profundes Wissen über die Terminologie von UML und deren Zusammenhänge vorhanden sein. Die Anpassungen am Metamodell bedürfen gegebenenfalls auch Beschreibungen von Constraints als Text. Damit ist ein domänenspezifisches Wissen notwendig. Eine neukonzipierte gemeinsame Notationssprache bietet neben der Möglichkeit der Dokumentation eines Systems auch

einige Toolunterstützung, die durch Stereotypen angepasste Generierung von Code. Trotz der großen Anzahl an bereitgestellten Standardprofilen für viele Anwendungsgebiete ist die Verwendung von UML-Profilen, vor allem für Domänenexperten, zu kompliziert und aufwendig. Insbesondere für kleine Softwareentwicklungsprojekte wie die Erstellung von Benutzeroberflächen, ist der Einsatz von UML-Profilen ein überhöhter Aufwand. UML-Profile sind dafür sehr gut dafür geeignet, um UML als Standard derart anzupassen, dass es auch für neue Anwendungsgebiete jenseits der Standard-Systemmodellierung passt.

3.3 Domain Specific Language

Eine weitere Möglichkeit Constraints für UML-Klassendiagramme zu definieren um daraus Benutzeroberflächen generieren zu können sind DSLs. DSLs werden für eine bestimmte Domäne, für bestimmte Personen (Domänenexperten) und für einen bestimmten Zweck erstellt. Dadurch sind diese Sprachen wenig komplex, leicht zu erlernen und zu lesen. Es gibt nach [Voelter et al 2010, 26f] zwei Varianten von DSLs. Zum einen gibt es interne DSLs, die bestehende Sprachen wiederverwenden, um eine neue DSL zu erstellen. Zum anderen gibt es externe DSLs, die komplett selbst definiert sind und deshalb ihre eigenen Tools (Editor, Compiler, Parser etc.) benötigen. Eine interne DSL wäre z.B. eine Sprache für Datenbankabfragen, welche mit einer General Purpose Languages (GPL) wie Java programmiert wurde. Da diese DSL auf einer GPL basieren, können die dafür vorgesehenen Tools verwendet werden. Um eine DSL zu erstellen werden bestimmte Tools, sogenannte Language Workbenches eingesetzt. Sie enthalten alle notwendigen Bestandteile zur Erstellung einer eigenen DSL. Die bekanntesten und weit verbreitetsten Language Workbenches sind das Meta Programming System (MPS) von Jetbrains und das auf Eclipse basierte Xtext. In dieser Arbeit wird für die Erstellung einer DSL XText verwendet. Grund hierfür ist, dass die in Eclipse-Papyrus erstellten UML-Modelle direkt in Eclipse verwendet werden können, weil sie Ecore konform sind. MPS dagegen bietet nicht die Möglichkeit direkt UML-Modelle zu importieren. Dafür wurde es als Workbench nicht designt. Theoretisch wäre es jedoch möglich eine DSL zu entwickeln um UML-Modelle zu verarbeiten. Dies würde jedoch den gegebenen Rahmen verlassen.
Aus den designten DSLs werden für bestimmte Probleme der jeweiligen Domäne Modelle modelliert. Diese Modelle dienen als Ausgangspunkt zur Generierung von Anwendungscode. Im unseren Anwendungsfall handelt es sich um eine DSL um Constraints für UML-Klassendiagramme definieren zu können. Im Sinne des Model-Driven Software Development (MDSD) dienen Modelle dieser Art nicht allein der Dokumentation, sondern werden nach [Stahl/Völter et al 2007, 10f] auch als Quellcode verstanden. Die Modelle stellen Sachverhalte der Domäne stark abstrahiert dar. Damit ist eine formale Modellierung und damit hohe Automatisierung möglich. Daraus resultiert wiederum eine Produktivitätssteigerung, sowie eine Qualitäts- und Wartbarkeitsverbesserung von Softwaresystemen. Model Driven Architecture (MDA) ist hingegen eine Spezialisierung des MDSD, da das MDSD-Konzept ausschließlich mit den offenen Standards der OMG (UML, MOF, QVT) implementiert wird. Ziel von MDA ist, die Interoperabilität zwischen Werkzeugen und die Standardisierung populärer Anwendungsbereiche zu erhöhen (vgl. [Stahl/Völter 2007,12).
Eine Plattform für die modellgetriebene Softwareentwicklung ist die openArchitectureWare (oAW). Sie ist unter der Open-Source-Lizenz Eclipse Public License (EPL) frei verfügbar. Die oAW bietet die Möglichkeit für eine Vielzahl von Modelltypen Codegeneratoren zu erzeugen um beliebigen Quellcode zu erzeugen. Modelle die verarbeitet werden können sind unter Anderem EMF- und UML-Modelle. Aktuell wird oAW unter dem neuen Namen Modeling Workflow Engine (MWE) im Eclipse Modelling Projects weiterentwickelt. XText ist Teil dieses Framework zur Entwicklung von DSLs. MWE besteht im Kern aus einer Sprachfamilie, die aus drei Teilen zusammengesetzt wird:

- Xtend
- Check und
- Xpand.

Xtend ist eine funktionale (Programmier-)Sprache die zur Erweiterung (Extention) bestehender Metamodelltypen dient. Sie ist ähnlich zu Java und bietet jedoch eine kompaktere Syntax und erweiterte Konzepte bzw. Funktionalitäten. Dadurch ist er Umgang mit ihr schnell zu erlernen, vor allem, wenn bereits Vorkenntnisse in Java bestehen. Check ist äquivalent zu OCL. Jedoch kann Check, im Gegensatz zu OCL nicht nur auf MOF-kompatible Modelle angewendet werden. Beispielsweise kann Check auf Erweiterungen von Xtend zugreifen. Check ist wegen der Ähnlichkeit zu OCL ebenfalls ungeeignet Constraints für UML-Modelle so zu definieren, dass sie leicht in der Codegenerierung mit inbegriffen werden können. Check wird an dieser Stelle der Vollständigkeit halber erwähnt, spielt aber in der zu erstellenden DSL keine weitere Rolle. Xpand ist eine Templatesprache die speziell für die Codegenerierung entwickelt wurde und bietet dafür wichtige Features. Die Workflow-Engine ist ein Komponentenframework, dass zur Erstellung komplexer Generatoren dient. Dafür wird ein Generator in einzelne Cartridges aufgeteilt. Ein Cartridge ist ein beliebiger Teil eines Generators. Diese Aufteilung in eine Art Modulen ermöglicht eine Wiederverwendung bereits erstellter Generatorteile. Eine Workflowbeschreibung entspricht hier einem Bauplan des Generators und definiert die Schnittstellen der verwendeten Cartridges. Dadurch kann ein Generator als Ganzes innerhalb einer Jar-Datei an Dritte weitergegeben werden und mit der Workflow-Schnittstelle ausgeführt werden. Das Zusammensetzen der Komponenten und das anschließende Ausführen, geschieht über Workflows. Workflows werden über eine einzelne Java Virtual Machine (JVM) ausgeführt. Es werden ebenso Transformationen und Artefakt-Erzeugungen über Workflows definiert (vgl. [Lorenzo 2013, 28ff]).
XText benutzt die MWE(2) DSL um die Erstellung von Artefakten zu konfigurieren.
Wird ein Workflow ausgeführt, erzeugt XText alle notwendigen Artefakte für ein UI-Editor der betrachteten DSL. Außerdem wird eine ANTLR-Spezifikation für die DSL abgeleitet. Dieses Tool unterstützt die Erzeugung von Parsern, Lexern und TreeParsern für Gramatiken. Bei XText sorgt ANTLR dafür, dass Abstrakte Syntaxbäume (ASTs) beim Parsen erzeugt werden. XText beruht außerdem auf dem Eclipse Modeling Framework (EMF). Dem EMF zu Grunde, liegt das Meta(meta)modell Ecore. Ecore entspricht einer abstrakten Sprache zur Definition von Metamodellen ohne Technologieabhängigkeit, sowie einem Framework. Diese Metamodelle können wiederum Sprachen anderer Modelle sein. Es dient, angelehnt an der Meta Object Facility (MOF) von der OMG, der Überbrückung zwischen unterschiedlichen Metamodellen, durch Schaffung einer allgemeinen Grundlage. Sind zwei verschiedene Metamodelle MOF- bzw. Ecore-konform, dann können Modelle die auf ihnen basieren gemeinsam, durch z.B. Modelltransformation, verarbeitet werden. Eine Untermenge von MOF 2.0 ist EMOF (Essential MOF). Diese ist weitgehend kompatibel zu Ecore. Folglich sind alle Java-Klassen die von EMF erzeugt werden Unterklassen von EObject, einem Element von Ecore, das als EMF-Äquivalent von java.lang. Object gesehen werden kann. Analog hierzu korrespondiert EClass zu java.lang.Class. Bei der Erstellung einer DSL mit XText wird gewöhnlich mit der Struktur (Grammatik) begonnen. Diese wird intern automatisch in einem AST umgewandelt. Die Grammatik besteht aus Features bzw. Regeln. Die erste Regel entspricht dem Wurzelknoten vom AST. Die Beschreibungsausdrücke einer Regel werden hierbei angelehnt an der erweiterten Backus-Naur-Form (EBNF) rekursiv aufgelöst. Aus dem resultierenden AST können anschließend EMF-Klassen generiert werden. Instanzen der Klasse werden durch eine statische Factory (Fabrikmethode) erzeugt. Aus diesem Grund existieren innerhalb der Klassen keine

Konstruktoren. Die Attribute (Features) werden durch Getter- und Setter-Methoden initialisiert (vgl. [Lorenzo 2013, 34ff]). Grund für die detaillierte Beschreibung Framework-Bestandteile von Xtext ist der im nächsten Abschnitt erstellte Prototyp, der als Concept of Proof dient.

4 Concept of Proof Implementation

Die Erstellung einer DSL mit Xtext hat sich bei der Analyse als geeigneter Ansatz herausgestellt. Es für die Erstellung einer DSL ebenfalls Wissen erforderlich. Jedoch ist die Menge an möglichen Constraint-Arten die in der Praxis innerhalb von einem UML-Klassendiagramm definiert werden müssen überschaubar. Aus diesen Grund kann eine DSL zur Definition von Constraints für UML-Modelle relativ schnell definiert und erweitert werden. Bei der Erstellung der DSL gibt es jedoch zwei Probleme. Das erste Problem besteht in dem Zugriff auf die UML-Klassendiagrammelemente. In der DSL muss ein Zugriff auf die Elemente gewährleistet werden, damit Regeln/Constraints für diese beschrieben werden können. Gelöst wird dieses Problem durch ein von Christrian Dietrich, Berater und Entwickler der ItemisAG, bereitgestelltes Projekt. Das Projekt bietet eine in XText generierte DSL, die direkte Referenzen (qualifizierte Pfadname) auf UML-Elemente ermöglicht (vgl. [Dietrich 2011]). Dietrich stellt das Projekt über Github zur Verfügung (vgl. [Dietrich 2015]). Die Referenzierung funktioniert hierbei über die Ecore-konformität. Das zweite Problem bezieht sich auf die Darstellung von Constraints innerhalb der DSL. Die UML-DSL, welche auf UML-Elemente referenzieren kann muss derart erweitert werden, dass Constraints möglichst einfach definiert werden können. Da Constraintdefinitionen sowohl aus Literalen verschiedenster Standardtypen (INT, STRING, BOOLEAN) bestehen, als auch aus Objekten die den UML-Elementen entsprechen, erweist die DSL-Erweiterung um diese Aspekte als äußerst herausfordernd. Eine Sprache zu erstellen, die mit generell wirkenden Ausdrücken Constraints beschreiben kann, wobei eine Validierung z.B. die Typisierung überprüft, sprengt hierbei den Rahmen. Darum werden, wie Abb. 2 im unteren Bereich zeigt, der Einfachheit halber zwei Arten von Constraints definiert.

```
Model:
    elements+=Element*
;

Element:
    Class | Property | Association | PropertyEqualityConstraint | NumberRestrictionConstraint
;

Class:
    "class" name=ID "mapsTo" ref=[uml::Class|FQN]
;
FQN returns ecore::EString:
    ID ("." ID)*
;

// UML-Mapping: ------------------------------------------------------
Property:
    "property" name=ID "mapsTo" ref=[uml::Property|FQN]
;

Association:
    "association" name=ID "mapsTo" ref=[uml::Association|FQN]
;

// Constraint:-------------------------------------------------------------
PropertyEqualityConstraint:
    "PropertyEqualityConstraint:" leftSite=[Property] (equality=STRING)? rightSite=[Property]
;

NumberRestrictionConstraint:
    "NumberRestrictionConstraint:" leftSite=[Property] (equality=STRING)? rightSite=INT
;
```

Abbildung 2: Struktur bzw. Grammatik der DSL

Zusätzlich zu den constraintbedingten Sprachelementen sind in der Abbildung 2 die für die DSL notwendigen Sprachdefinitionen zur Referenzierung der UML-Modellelemente definiert. Aktuell können Assoziationen, Properties und Classen direkt referenziert werden. Für die zwei Constraintarten wird jedoch nur die Property-Referenzierung verwendet. Das erste Constraint beschreibt einen Property-Vergleichsausdruck. D.h. es werden zwei Properties mit einem Vergleichs- bzw. Ungleichzeichen in Beziehung gesetzt. Abb. 3 zeigt den Constrainttyp am Beispiel.

```
property Name mapsTo Model.TextArea_Name.text
property Password mapsTo Model.TextArea_Password.text
property Age mapsTo Model.TextArea_Age.text

PropertyEqualityConstraint: Name "!=" Password
NumberRestrictionConstraint: Age ">" 18
```

Abbildung 3: Beispielmodell zur Definition von Constraints mit der UML-DSL

Die Constraint besagt, dass das Attribute ‘text‘ mit dem Alias ‘Name‘ von der Klasse ‘TextArea_Name‘ nicht gleich sein darf mit dem Attribut ‘text‘ mit dem Alias ‘Password‘ von der Klasse ‘TextArea_Password‘. Die zweite Constraint besagt, dass das Attribut ‘text‘ mit dem gesetzten Alias ‘Age‘ von der Klasse ‘TextArea_Age‘ größer als 18 sein muss. Die Definition von Aliase dienen der Lesbarkeit und Wiederverwendbarkeit. Eine Änderung der Sprache, dass Constraints direkt ohne Aliase beschrieben werden können wäre ohne weiteres möglich. Bei der Definition der ‘NumberRestrictionConstraint‘ ist die Reihenfolge festgelegt. Der Ausdruck wird hierbei nicht innerhalb der Sprache ausgewertet um beispielsweise die Typisierung zu prüfen. Dies sind mögliche zukünftige Optimierungen der Sprache. Abb. 4 Zeigt das UML-Klassendiagramm, welches für die Generierung eines

Formulars, wie es in Abb. 5 gezeigt wird, zusammen mit den definierten Constraints, eingesetzt wird.

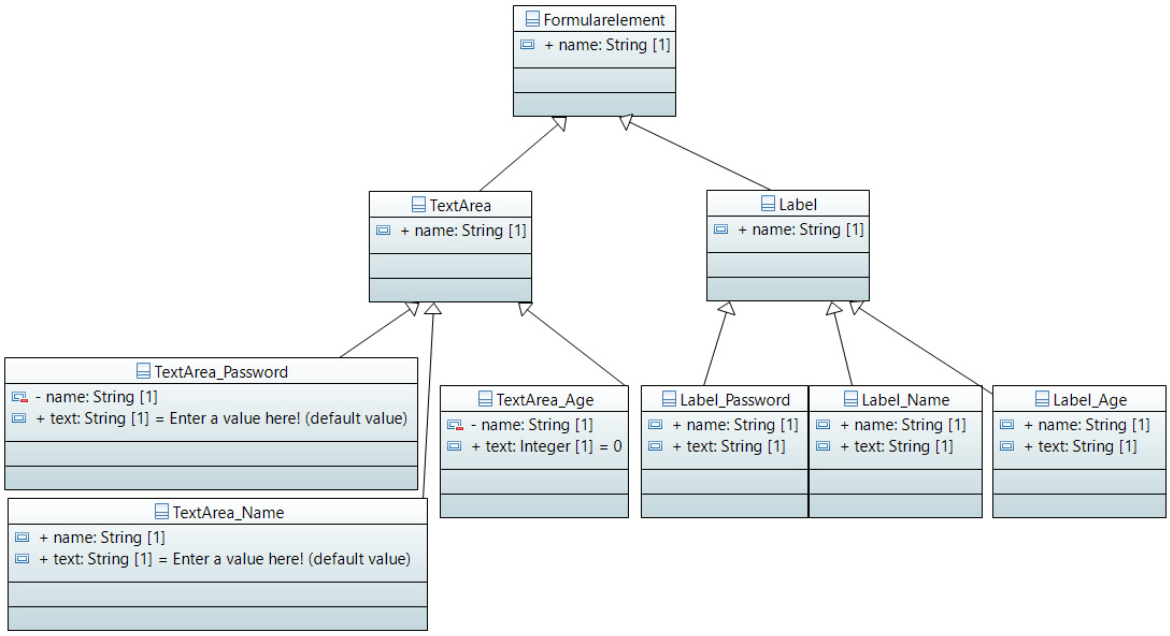

Abbildung 4: UML-Klassendiagramm Beispielmodell

Das Modell beschreibt beispielhaft ein kleines Formularfeld. Das Formularfeld besteht aus den Textfeldern Name, Password und Age, sowie den dazugehörigen Labels. Das Namen-Attribute ist hierbei optional. Es wird nicht in der Codegeneration verwendet und müsste nicht in den abgeleiteten Klassen angezeigt werden, weil es implizit klar ist, dass das Attribut von der Formularelement-Klasse nach unten weitervererbt wird. Der Name ist insofern wichtig, dass es Anwendungsfälle geben könnte wo der eindeutige Name Verwendung findet. Z.B. bei eventbasierten Anwendungen. Das Text-Attribute entspricht hierbei den aktuellen Text des jeweiligen Formularements. Erkennbar sind auch die Default-Values der Felder. Diese entsprechen als Startwert ebenfalls einer Zusicherung. Abb. 5 zeigt das ausgeführte Programm aus dem beispielhaft generierten Code.

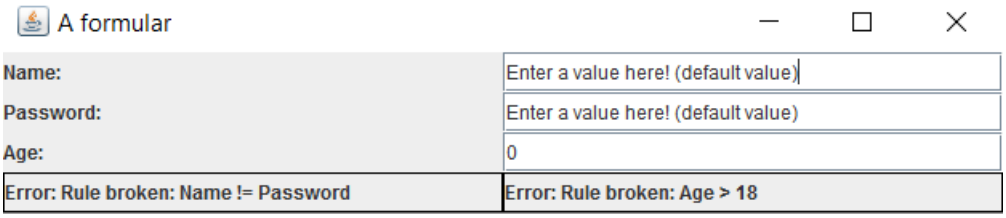

Abbildung 5: Ausgeführtes Beispielformular aus Generat

An den Fehlermeldungen unten ist zu sehen, dass die definierten Constraints im ausgeführten Generat aktiv werden. Name und Passwort sind mit dem gleichen Startwert belegt. Da sie nach dem Constraint nicht gleich sein sollen wird der Fehler unten links eingeblendet. Wird die Zeile geändert und die Constraint erfüllt, verschwindet auch die Fehlermeldung. Zur Steigerung des Verständnisses zeigt Abb. 6 einen Ausschnitt aus dem Codegenerator.

```
«FOR rule : numberRestrictionConstraint»
try{
    if(!(Integer.valueOf(ta_«rule.leftSite.name».getText()) «rule.equality»(«rule.rightSite»))){
            errorLabel2.setText("Error: Rule broken: «rule.leftSite.name» «rule.equality» «rule.rightSite»");
        }else{
                errorLabel2.setText("");
            }
        }catch(Exception e){
            errorLabel2.setText("Invalid value for the field: «rule.leftSite.name»; -> «rule.rightSite.class» expected");
        }
«ENDFOR»
```

Abbildung 6: Ausschnitt aus dem Codegenerator

Der Codegenerator ist mit Xtend implementiert. Die grau unterlegten Bereiche entsprechen zu generierenden Freitext. Der Abschnitt zeigt Code aus der Update-Funktion, die jedes Mal aufgerufen wird, wenn sich etwas in einem TextArea-Feld ändert. Im Template kann direkt auf die Ecore-Elemente zugegriffen werden, welche aus der Struktur der DSL resultieren.

5 Zusammenfassung

Alle Ansätze basieren auf UML-Klassendiagramme. Darum muss davon ausgegangen werden, dass auch der Domänenexperte bei der Definition von Constraints die UML versteht. Die Implementierung des Prototyps zeigt wie relativ einfach Constraints über eine eigene DSL zu einem oder mehrere UML-Modellen definiert werden können. Ist der Generator für eine bestimmte Art von UML-Struktur vollständig entwickelt, muss anschließend nur noch das UML selbst oder das DSL-Modell vom Modellexperten angepasst werden um schnell und einfach neuen Code zu generieren. Eine kritische Betrachtung lässt ein weiteres Problem erkennen. Mit Steigenden Anforderungen und den damit folgenden Anpassungen an die UML/Constraint-DSL, kann es schnell geschehen, dass Constraintarten modellierbar sind, die im Codegenerator nicht evaluiert werden. Die Programmierer müssen bei der Entwicklung von Generatoren darauf achten alle für die Domäne notwendigen Constraints und mit der UML-DSL ausdrückbaren Constraint-Typen anforderungsgerecht umzusetzen. Andernfalls werden von dem Domänenexperten Constraints definiert die keinen Einfluss auf das Generat haben. Die Dokumentation von Expertenwissen ist in der modellgetriebenen Softwareentwicklung jedoch nicht das oberste Ziel. Der Prototyp zeigt wie Constraints für Klassendiagramme abgebildet werden können. Die Lösung ist erweiterbar, anpassbar und für den zukünftigen User leicht erlernbar.

Literaturverzeichnis

[Damus 2015] Damus, C. W., Fun with OCL in Papyrus Mars 2015, URL: http://www.damus.ca/blog/2015/6/15/fun-with-ocl-in-papyrus-mars.

[Dietrich 2011] Dietrich, C., Christian´s Blog: Xtext 2.0 and UML, 2011, UML: https://christiandietrich.wordpress.com/2011/07/17/xtext-2-0-and-uml/.

[Dietrich 2015] Dietrich, C., Xtext-uml-example, 2015, URL: https://github.com/cdietrich/xtext-uml-example.

[Kraus et al 2007] Kraus, A, Knapp, A, Koch, N., Model-Driven Generation of Web Applications in UWE, 2007.

[Kuhn 2008] Kuhn, S., Diplomarbeit, Entwicklung eines domänenspezifischen UML Diagramms zur Benutzeroberflächenmodellierung, 2008, URL: https://wiki.eclipse.org/images/d/d0/DA_StefanKuhn.pdf.

[Lorenzo 2013] Bettini, Lorenzo, Implementing Domain-Specific Languages with Xtext and Xtend, 2013.

[OMG 2015] OMG, UML Specification, 2015, URL: http://www.omg.org/spec/UML/2.5/PDF.

[OMG 2015b] OMG, MOF Specification, 2015, URL: http://www.omg.org/spec/MOF/2.5/PDF.

[OMG 2014] OMG, Model Driven Architecture (MDA), MDA Guide rev.2.0, 2014, URL: http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf.

[OMG 2014b] OMG, OCL Specification, 2014, URL: http://www.omg.org/spec/OCL/2.4/PDF.

[Stahl/Völter et al 2007] Stahl, T., Völter, M., Efftinge, S., Haase, A., Bettin, J., Helsen, S., Kunz, M., Modellgetriebene Softwareentwicklung, Engineering, Management, 2007.

[Voelter et al 2010] Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L., Visser, E., Wachsmuth, G., DSL Engineering, Designing, Implementing and Using Domain-Specific Languages, 2010, URL:

http://voelter.de/data/books/markusvoelter-dslengineering-1.0.pdf.