Entwurfsmustergesteuerte Erzeugung von OCL-Constraints

Thomas Baar[1], Reiner Hähnle[2], Theo Sattler[1]und Peter H. Schmitt[1]

[1] Fakultät für Informatik
Universität Karlsruhe
D-76128 Karlsruhe
{baar,sattler,pschmitt}@ira.uka.de
[2] Department of Computing Science
Chalmers University of Technology
S-41296 Göteborg
reiner@cs.chalmers.se

Zusammenfassung Eine der größten Hürden auf dem Weg zur formalen Verifikation von Software ist die Erstellung und Validierung der hierfür notwendigen formalen Spezifikation. Der Erfolg formaler Methoden bei der industriellen Softwareerstellung hängt also davon ab, ob es zum einen gelingt, die notwendigen Zugangsvoraussetzungen für die Erstellung und Verwendung formaler Objekte gering genug zu machen, und zum anderen davon, formale und informelle Softwaremodelle möglichst eng zu integrieren. Für letzteres bietet die weithin verwendete, objektorientierte Modellierungssprache *Unified Modeling Language* (UML) einen guten Ansatzpunkt durch ihre semi-formale Untersprache *Object Constraint Language* (OCL). In der vorliegenden Arbeit zeigen wir, daß es auch für Programmierer ohne formalen Hintergrund prinzipiell möglich ist, formale Teilspezifikationen in OCL zu erstellen. Dies erfolgt durch Auswahl, Instanziierung und Adaption von schematischen OCL-Constraints, die von Spezialisten sorgfältig definiert wurden. Durch die Bindung von OCL-Constraints an Entwurfsmuster wird ein hoher Grad an möglicher Hilfestellung und maschineller Unterstützung erreicht. Durch die Erhebung der Constraints im Rahmen eines konkreten Industrieprojekts wird der Praxisbezug des Ansatzes sichergestellt.

1 Einführung

Eine der größten Hürden auf dem Weg zur formalen Verifikation von Software ist die Erstellung und Validierung der hierfür notwendigen formalen Spezifikation. Der zukünftige Erfolg formaler Methoden bei der industriellen Softwareerstellung wird also entscheidend davon abhängen, ob es zum einen gelingt, die notwendigen Zugangsvoraussetzungen für die Erstellung und Verwendung formaler Objekte gering genug zu halten, und zum anderen davon, formale und informelle Softwaremodelle möglichst eng zu integrieren.

Für letzteres bietet die weithin verwendete, objektorientierte Modellierungssprache *Unified Modeling Language* (UML) [7, 8] einen guten Ansatzpunkt durch ihre semi-formale Untersprache *Object Constraint Language* (OCL) [7, 11]. Mit sogenannten *OCL-Constraints* lassen sich semantische Eigenschaften von UML-Diagrammen und ihren Elementen präzise ausdrücken.

Die Notation der OCL ist an die Gepflogenheiten der objektorientierten Modellierung (OOM) angepaßt, die Syntax ist einfach und verzichtet auf mathematische Symbole. Dies führt zu relativ leichter Lesbarkeit für Entwickler objektorientierter Software. Bei der tatsächlichen Formalisierung von semantischen Eigenschaften in OCL haben die meisten Benutzer aber dennoch massive Probleme.

In der vorliegenden Arbeit wenden wir uns daher der Frage zu, wie die Erzeugung von korrekten und sinnvollen OCL-Constraints im Rahmen der Softwaremodellierung mit UML so unterstützt werden kann, daß auch ein Nichtspezialist für formale Spezifikation davon profitiert. Wir zeigen, daß es auch für Programmierer ohne formalen Hintergrund prinzipiell möglich ist, formale Teilspezifikationen in OCL zu erstellen. Dies erfolgt durch Auswahl, Instanziierung und Adaption von schematischen OCL-Constraints, die von Spezialisten sorgfältig vorbereitet wurden.

Das kann nur gelingen, wenn die OCL-Constraints in hohem Maß vorstrukturiert und aufbereitet sind. Wenn man nach Vorlagen für eine solche Strukturierung Ausschau hält, bietet sich das in den letzten Jahren extrem erfolgreiche Paradigma der Entwurfsmuster für Software [3] an. Im folgenden demonstrieren wir, daß durch die Bindung von schematischen OCL-Constraints an Entwurfsmuster ein hoher Grad an Hilfestellung und maschineller Unterstützung bei der Erzeugung formaler Spezifikationen erreicht werden kann.

Es geht also zunächst darum, basierend auf konkreten Modellierungen mit UML und Mustern, relevante semantische Eigenschaften und die Entwurfsmuster, in deren Verbindung sie auftreten, zu identifizieren. Zur Gewährleistung von Signifikanz und Praxisnähe wurde die Untersuchung in einem konkreten Projekt eines mittelständischen Systemhauses angesiedelt. Die Analyse der Probleme, die die Entwickler bei ihren Verwendungsversuchen der OCL hatten, sowie allgemeine, durch den OCL-Sprachentwurf bedingte Hürden, motivieren die direkte Verknüpfung von Entwurfsmustern mit Constraints. Dies ist der Inhalt des folgenden Abschnitts 2. Die Auswahl geeigneter Muster und Constraints wird in Abschnitt 3 beschrieben. Eine Definition der Verknüpfung von Constraints mit Mustern sowie Beispiele, bei deren Formulierung wir allerdings eine Vertrautheit des Lesers auch mit fortgeschrittenden Konzepten der OCL voraussetzen mußten (zur Einführung in OCL siehe [11]), finden sich in Abschnitt 4. In Abschnitt 5 geben wir eine Zusammenfassung und zeigen auf, was für einen realistischen Einsatz zu tun bleibt.

2 Erhebung von geeigneten Mustern und Constraints

Zunächst beschreiben wir das Industrieprojekt samt Umfeld, in dem wir praxisrelevante Vorstellungen darüber, welche Eigenschaften von Entwürfen zu formalisieren sind, erhoben haben.

2.1 Projekt und Umfeld

Dieser Teil der Untersuchung wurde vor Ort in dem mittelständischen Systemhaus *fun communications* GmbH, Karlsruhe durchgeführt. Die Gründe für diese Wahl, neben der grundsätzlichen Bereitschaft der Firma zu dieser Art der Zusammenarbeit, waren:

- Entwurf und Entwicklung von Software findet bei der *fun communications* GmbH mit objektorientierten Methoden unter Verwendung von CASE-Werkzeugen statt.
- Die verwendete Modellierungssprache ist UML.
- Die Entwickler machen routinemäßig Gebrauch von Entwurfsmustern, was auch von den verwendeten CASE-Werkzeugen unterstützt wird.
- Wir hatten Gelegenheit, die Entwurfsarbeit an einem typischen Projekt von Anfang an mitzuverfolgen.

Ziel des Softwareprojekts war die Erstellung des eMail-Clients *keyMail/S*, der außer den Standardfunktionen, wie Senden, Empfangen, Verwalten von Nachrichten und Adressen (Kontakten) eine Konfigurierbarkeit für mehrere Benutzer, Unterstützung für Verschlüsselung und Signatur von Nachrichten, sowie Zertifikatsverwaltung bereitstellt.

Analyse und Entwurf des Projekts wurden weitgehend in UML erstellt. *keyMail/S* besteht im wesentlichen aus den fünf Paketen Benutzer-, Nachrichten-, Konto-, Kontakt- und Zertifikatsverwaltung. Die Benutzerverwaltung kontrolliert den Zugang. Wer *keyMail/S* benutzen will, muß sich über die Benutzerverwaltung anmelden. Ein Benutzer kann mehrere Konten besitzen. Jedes Konto ist einem Mail-Server zugeordnet und wird von der Kontoverwaltung gepflegt. Die Kontaktverwaltung stellt jedem Benutzer ein Adreßbuch zur Verfügung, in dem er seine Kontakte hierarchisch ordnen und gruppieren kann. Die Nachrichtenverwaltung stellt eine ähnliche Funktionalität für die Archivierung von Nachrichten in hierarchischen Ordnern zur Verfügung.

2.2 Analyse der UML-Modellierung

Das Analysemodell von *keyMail/S* umfaßt 47 Klassen, wobei die einzelnen Pakete zwischen 5 und 11 Klassen beinhalten. Wir konzentrieren uns auf die Kontaktverwaltung, weil diese die meisten Aspekte umfaßt und zum Zeitpunkt unserer Untersuchung hauptsächlich vorangetrieben wurde.

Das Analysemodell der Kontaktverwaltung umfaßt 11 Klassen. In der Analysephase wurden keine OCL-Constraints verwendet. Einzelne Klassen wurden umgangssprachlich beschrieben. Methoden wurden zum Teil nur durch Wahl einschlägiger Bezeichner, wie *addElement()*, *removeElement()*, dokumentiert. Das verfeinerte Entwurfsmodell der Kontaktverwaltung umfaßt 48 Klassen. Insgesamt wurden 11 verschiedene Entwurfsmuster aus [3, 1] verwendet (Abstract Factory, Builder, Bureaucracy, Composite, Counted Pointer, Iterator, Observer, Proxy, Serializer, Singleton, Strategy). Jede Klasse war Teil mindestens eines Musters.

2.3 Erhebung von Constraints

Während der Verfeinerung des Entwurfsmodells wurden die Entwickler dazu aufgefordert, ihre bislang schriftlich oder auch nur mündlich ausgedrückten Einschränkungen in Form von OCL-Constraints auszudrücken, insbesondere im Zusammenhang mit der Verwendung von Entwurfsmustern. Dabei wurden sie von uns unterstützt. Hier ist festzuhalten, daß zunächst der Sinn des Vorgehens in Frage gestellt wurde. In diesem Zusammenhang ist erwähnenswert, daß nur ein Teil der Systementwickler eine Informatik-Ausbildung hinter sich hat, während, wie derzeit für kleinere Unternehmen typisch, viele einen Hintergrund in den Natur- oder Ingenieurwissenschaften haben und keinerlei Erfahrung mit formalen Sprachen besitzen.

Die zögerliche Haltung zu den Formalisierungsversuchen änderte sich rasch, nachdem es uns gelang, Unklarheiten und Mehrdeutigkeiten im Entwurf mit Hilfe der präziseren OCL-Notation aufzuzeigen. Danach stieß OCL als formale Spezifikationssprache auf weitgehende Akzeptanz, was u.E. zu einem großen Teil auf die an OO-Programmiersprachen angelehnte Syntax zurückzuführen ist.

Es zeigte sich jedoch auch deutlich, daß Entwickler trotz guten Willens erhebliche Schwierigkeiten mit der korrekten Verwendung von OCL haben, was zum Teil gerade durch die Nähe zur OO-Syntax bedingt ist. Zum Beispiel wurde mehrfach versucht, in einem OCL-Ausdruck den Zustand eines Objekts zu modifizieren, was durch die Seiteneffektfreiheit von OCL ausgeschlossen ist. Die Formalisierung komplizierter Sachverhalte überforderte die meisten Entwickler.

Zusammenfassend ist zu sagen, daß auch Entwickler ohne Hintergrund in formalen Methoden den Sinn und die Vorteile von präziseren Modellierungen durchaus schätzengelernt haben und gerne anwenden würden. Das bestätigt die Erfahrung anderer Wissenschaftler bei der Etablierung formaler Methoden im Bereich industrieller Softwareentwicklung [5]. Für die Anwendung der OCL brauchen Systementwickler eine grundsätzliche Einführung in OCL, die im Rahmen eines OOM-Kurses mit UML erfolgen kann, und CASE-Werkzeuge, die eine (richtige) Verwendung von OCL unterstützen durch:

- integrierte OCL-Parser und -Editoren;
- Bibliotheken mit OCL-Idiomen;
- Verknüpfung von Entwurfsmustern mit typischen OCL-Constraints.

Im folgenden versuchen wir zu zeigen, daß dies in realistischer Weise möglich ist.

3 Auswahl und Verwendung von OCL-Constraints

3.1 OCL-Idiome

In allen Systemteilen von *keyMail/S* werden Daten in diversen Graphen- und Listenstrukturen verwaltet. Die jeweiligen Methoden zum Löschen und Einfügen von Elementen beinhalten regelmäßig solche Einschränkungen, die in UML-Diagrammen höchstens in Form von unstrukturierten Kommentaren auftauchen.

Ubiquitär sind Forderungen, wie zum Beispiel, daß bestimmte Elemente nicht in einer Liste auftauchen dürfen oder, umgekehrt, auftauchen müssen, etc. Folgendes Constraint[1] legt fest, daß die Variable *collection* vom OCL-Typ *Collection*, was sowohl eine Menge, Mehrfachmenge oder Folge sein kann, tatsächlich eine Menge ist:

```
collection->forAll(e | collection->count(e) = 1)
```

Solche Forderungen stehen nicht im Zusammenhang mit einem bestimmten Muster. Wegen ihrer Häufigkeit ist es dennoch angebracht, sie in allgemeiner Form zu formalisieren und in einer Bibliothek bereitzustellen. Wegen der starken Abhängigkeit von den in OCL vorgefundenen Konstrukten, nennen wir solche Constraints **OCL-Idiome**. Außer ihrer universellen Verwendbarkeit empfehlen sich OCL-Idiome als übersichtliche und praxisrelevante Formalisierungsbeispiele. Eine längere Liste mit von uns entdeckten und formalisierten OCL-Idiomen ist in [9] zu finden.

3.2 Mustergesteuerte OCL-Constraints

Als nächstes wurden systematisch die mit der Verwendung von Mustern einhergehenden Einschränkungen gesammelt und in OCL-Constraints umgesetzt.

Zum Beispiel findet bei rekursiven Datenstrukturen, wie Bäumen oder gerichteten Graphen-Strukturen, das Composite Muster Anwendung. Neben den OCL-Idiomen gibt es nun zusätzliche Einschränkungen, die an die *Struktur des Musters* geknüpft sind, wie zum Beispiel die Forderung nach Zyklenfreiheit der Instanzen.

Etliche Daten sollen dem Benutzer in verschiedener Sicht angeboten werden. Das Observer Muster realisiert einen effizienten Mechanismus, der die Daten und deren Sichten konsistent hält. Die Forderung nach Konsistenz kann im Rahmen dieses Musters allgemein in OCL formalisiert werden.

Teile der Daten müssen über die Laufzeit des Systems hinaus erhalten bleiben. Daher ist es notwendig, die Daten in eine linearisierte Form zu bringen, die z.B. in einer Datei abgelegt werden kann. In diesem Zusammenhang findet das Serializer Muster Anwendung. Dabei wird im wesentlichen gefordert, daß die Linearisierung umkehrbar ist.

Eine vollständige Liste der aus dem *keyMail/S*-Entwurf extrahierten und als OCL-Constraint formalisierten Anforderungen ist in der Arbeit [9] enthalten. Bei der weiteren Verfeinerung des Entwurfs traten von den formalisierten Entwurfsmustern besonders Composite, Singleton und Observer häufiger auf. Die Formalisierung ergab zwischen 4 (Observer) und 16 (Composite) Constraints, die mit einem Muster verknüpft werden konnten, wobei diese nicht notwendigerweise alle Aspekte des Musters abdecken.

---

[1] Das Constraint für die Methode *isUnique()* in der OCL-Sprachdefinition soll zwar dasselbe leisten, ist jedoch fehlerhaft formuliert, vgl. [9, Abschnitt 4.4].

3.3 Von Constraints zu Constraint-Schemata

Ein Constraint, das mit einem Muster verknüpft wird, muß wie dieses auf verschiedene Diagramme anpaßbar sein. Weiterhin sollten sich alle Anpassungsschritte von einem Muster zu einem Diagramm in uniformer Weise auf das Constraint übertragen lassen. Die Analyse der in der Praxis vorkommenen Anpassungsschritte eines Musters ist deshalb ein erster Schritt, um die Struktur und Verwendung der mit Mustern verknüpften Constraints zu motivieren. Moderne CASE-Werkzeuge wie GDPro, TogetherJ, etc. bieten dem Softwareentwickler bereits eine Muster-Bibliothek an. Aus einem ausgewählten Muster kann sich der Entwickler sein gewünschtes Diagramm generieren lassen. Bei diesem Vorgang, den wir Musterinstanziierung nennen, bieten die Werkzeuge folgende Möglichkeiten der Einflußnahme:

Strukturanpassung: Die mit einem Muster assoziierte Klassenstruktur stellt nur in seltenen Fällen hundertprozentig die Lösung für das aktuelle Entwurfsproblem dar. Notwendig sind üblicherweise strukturelle Anpassungen wie das Hinzufügen von Klassen, Attributen, Methoden.

Signaturanpassung: Die im Muster verwendeten Bezeichner charakterisieren zwar die Rollen der bezeichneten Entitäten innerhalb des Musters prägnant, als Bezeichner im resultierenden Entwurf eignen sie sich allerdings kaum. Der Entwickler wird durch Umbenennung eine Anpassung vornehmen.

Variantenbildung: Ein Muster beinhaltet den Lösungsansatz für eine Vielzahl ähnlicher Probleme, die sich in Details unterscheiden. Oft wird in der Beschreibung des Musters auch darauf eingegangen, wie zusätzliche Details bei der Problemstellung sich auf die Ausgestaltung des Lösungsansatzes auswirken (zum Beispiel im Abschnitt *Implementation* der Muster aus [3]). CASE-Werkzeuge reflektieren diesen Sachverhalt und geben dem Softwareentwickler die Möglichkeit, über zusätzliche Parameter die Generierung des Diagramms zu steuern. Ein typisches Beispiel ist die Entscheidung darüber, ob eine Assoziation zwischen zwei Muster-Klassen in dem resultierenden Entwurf als Aggregation oder Komposition realisiert werden soll.

Diese Techniken zur Musterinstanziierung, Strukturanpassung, Signaturanpassung und Variantenbildung, spielen auch bei der Anpassung der an ein Muster annotierten Constraints eine wichtige Rolle. Es hat sich für annotierte Constraints als zweckmäßig erwiesen eine spezielle Syntax zu verwenden. Deshalb verwenden wir für die mit Mustern vor ihrer Instanziierung bzw. Anpassung verbundenen Constraints den Begriff **Constraint-Schema**. Die formale Syntax von Constraint-Schemata, ihre Anpassung und Instanziierung zu Constraints sowie ihre praktische Verwendung sind Thema des folgenden Abschnitts.

4 Constraint-Schemata

4.1 Syntax

Um eine flexible Anpassung an Diagramme zu erreichen, haben Constraint-Schemata eine verglichen mit normalen OCL-Constraints leicht erweiterte Syntax. Wie bei diesen beziehen sich auch Constraint-Schemata auf ein Klassendiagramm, welches die verwendbaren Bezeichner festlegt. Im Falle von Constraint-Schemata ist dies das zum entsprechenden Muster P gehörende Klassendiagramm D_P.

Definition 4.1 (Syntax Constraint-Schema). Sei D_P ein gegebenes Klassendiagramm. Ein Ausdruck der Form

schema ***Name*** (***Parameters***)
    [**precond:** ***Guard***]
    ocl: ***OCLConstraint***

wird **Constraint-Schema** genannt. Dabei ist ***Name*** ein eindeutiger Bezeichner für das Constraint-Schema im Kontext von D_P; ***Parameters*** ist eine Liste von formalen Parametern der Form „***PType***   ***PName***“. Dabei muß ***PType*** ein gültiger OCL-Typbezeichner sein und ***PName*** ein noch nicht verbrauchter Bezeichner. ***Guard*** gibt die Vorbedingung für die Anwendbarkeit des Constraint-Schemas in OCL-Syntax an. Es expliziert Annahmen über die Struktur des Entwurfs. Die Angabe ist optional. ***OclConstraint*** ist ein unter Berücksichtigung von D_P und ***Guard*** syntaktisch korrekter OCL-Ausdruck, der zusätzlich die in ***Parameters*** eingeführten formalen Parameter gemäß ihrer Typisierung verwenden kann. Von großer praktischer Bedeutung ist weiterhin eine informelle Beschreibung dessen, was mit dem Constraint-Schema inhaltlich ausgedrückt werden soll. Dieser Kommentar hat für die Instanziierung des Constraint-Schemas formal gesehen keine Relevanz und wurde aus diesem Grunde nicht in der Syntax berücksichtigt. Bei der Verwendung der Constraint-Schemata in der Praxis erfolgt die Auswahl der benötigten Schemata durch den Softwareentwickler in starkem Maße jedoch auf Grundlage des zusätzlichen, beschreibenden Kommentars.

Zur Illustration betrachten wir das Singleton Muster, welches nur die eine Klasse *Singleton* enthält.

Abbildung1. Klassendiagramm des Musters Singleton

Die in der Beschreibung des Musters informell formulierte Anforderung, daß es von der Klasse *Singleton* nie mehr als eine Instanz gibt, läßt sich durch ein Constraint-Schema folgendermaßen formalisieren:

```
schema isSingleton()
   ocl: context Singleton inv:
      Singleton.allInstances->size <= 1
```

Manchmal wird Singleton leicht abgewandelt verwendet, wobei die Anzahl der erlaubten Instanzen durch eine größere Zahl als Eins beschränkt ist. Dies läßt sich einfach durch ein Constraint-Schema formulieren; die Obergrenze der Instanzenanzahl wird durch den formalen Parameter MAX angezeigt:

```
schema boundInstances(Integer MAX)
  ocl: context Singleton inv:
    Singleton.allInstances->size <= MAX
```

Bei der Instanziierung dieses Schemas (siehe Abschnitt 4.2) muß der Softwareentwickler den formalen Parameter MAX durch einen aktuellen Wert ersetzen.

Neben der Variantenbildung durch Parameter ist die Fähigkeit zur Erweiterung des im Muster vorgegebenen Klassendiagramms eine weitere wichtige Eigenschaft von Constraint-Schemata. Zur Illustration betrachten wir einen zu *boundInstances* alternativen Vorschlag *boundInstancesAlt*. Informell ausgedrückt besteht dieser darin, daß die Klasse *Singleton* um ein Attribut *id* vom Typ *Integer* erweitert wird. Als äquivalentes Constraint zu *boundInstances* kann man formulieren, daß der Wert von *id* in jeder Instanz von *Singleton* eindeutig und kleiner als MAX ist.

Um das Constraint für *boundInstancesAlt* zu formalisieren, muß die Klasse *Singleton* um das Attribut *id* erweitert werden. Eine scheinbare Lösung wäre, das Muster Singleton selbst zu erweitern und das Attribut *id* in die Klasse *Singleton* aufzunehmen. Das hätte aber zur Folge, daß bei jeder Anwendung des Musters das genannte Attribut erzeugt würde, auch wenn es die Situation nicht erfordert. Mit Hilfe von Constraint-Schemata kann man dies auf flexiblere Weise lösen:

```
schema boundInstancesAlt(Integer MAX)
  precond:
    Singleton.attributes->includes('id') and
    Singleton.allInstances->
              forAll(x | x.id.oclIsKindOf(Integer))

  ocl: context  Singleton inv:
    Singleton.allInstances->
      forAll(x | 0 < x.id and x.id <= MAX and
        forAll(y | x <> y implies x.id <> y.id))
```

Das Muster bleibt erhalten, aber wenn das Constraint-Schema sich auf zusätzliche Elemente (wie das Attribut *id*) bezieht, wird dies in der Bedingung, die auf das Schlüsselwort **precond:** folgt, festgehalten. Diese Bedingung muß garantieren, daß das eigentliche OCL-Constraint des Schemas nach der Instanziierung für ein gegebenes Klassendiagramm syntaktisch korrekt ist.

4.2 Instanziierung

Die Instanziierung von Constraint-Schemata lehnt sich an die Instanziierung von Mustern an, siehe Abschnitt 3.3. Im folgenden wird sowohl die Instanziierung von Mustern als auch die Instanziierung von Constraint-Schemata formal definiert.

Wir benutzen hierbei die Begriffe *Diagramm* und *Term* in der Bedeutung „UML-Diagramm“ und „Term einer formalen Sprache“.

Instanziierung von Mustern Diagramme werden üblicherweise in graphischer Form dargestellt, haben jedoch auch eine Termrepräsentation, zum Beispiel [10]. Dadurch kann das Verfahren der Musterinstanziierung uniform auf die Instanziierung von Constraint-Schemata, die ebenfalls Terme sind, übertragen werden. Die genaue Art der Termrepräsentation von Diagrammen spielt für unsere Untersuchungen dabei keine Rolle.

Unter der Signatur eines Terms wird im allgemeinen die Menge aller vorkommenden Funktionssymbole und Variablen verstanden. Wir benutzen im folgenden den Begriff der Signatur in adäquater Weise für Termrepräsentationen von Diagrammen. Die Signatur eines Diagramms ist somit das in ihm vorkommende Vokabular oder, präziser ausgedrückt, die Gesamtheit aller vorkommenden Bezeichner (z.B. für Klassen, Attribute, Methoden, Assoziationsenden, etc.).

Der Prozeß der Musteranpassung, so wie er in vielen CASE-Werkzeugen realisiert ist, kann grob wie folgt beschrieben werden (wobei die in einigen CASE-Werkzeugen realisierte Form der Variantenbildung unberücksichtigt bleibt). Gegeben sei ein Muster P, das strukturell durch ein Klassendiagramm D_P beschrieben ist.

Schritt 1: Strukturanpassung: Änderung von D_P zu D'_P
In diesem Schritt paßt der Softwareentwickler das Diagramm des Musters seinen Erfordernissen an. Ein Beispiel ist im Singleton Muster das Hinzufügen eines Attributes *id* in der Klasse *Singleton*.

Schritt 2: Definition der Signaturabbildung σ
In diesem Schritt paßt der Softwareentwickler das im Muster verwendete Vokabular durch Umbenennung seinen Wünschen an. Formal gesehen handelt es sich bei der Umbenennung um eine Signaturabbildung, die wir mit σ bezeichnen.

Schritt 3: Generierung des resultierenden Entwurfs D
Der resultierende Entwurf D entsteht durch Anwendung der Signaturabbildung σ auf das strukturell angepaßte Diagramm D'_P. Dieser Schritt wird automatisch vom CASE-Werkzeug auf Grundlage der Beziehung $D = \sigma'(D'_P)$ durchgeführt. Dabei ist σ' die kanonische Erweiterung der Signaturabbildung σ auf Terme.

Instanziierung von Constraint-Schemata Die Instanziierung eines Constraint-Schemas vollzieht sich in analoger Weise zur Instanziierung von Mustern, erfordert jedoch zusätzlich die Behandlung der mit ihm verknüpften Vorbedingung und der formalen Parameter.

Gegeben sei wiederum ein Muster P, das zugehörige Klassendiagramm D_P und ein an Muster P annotiertes Constraint-Schema CS_P. Der Instanziierungsvorgang vollzieht sich in folgenden Schritten:

Schritt 1.1: Strukturanpassung von D_P zu D'_P

Schritt 1.2: Prüfung der Anwendbarkeit von CS_P auf D'_P
Die in CS_P formulierten Vorbedingungen müssen von D'_P erfüllt werden. Dazu muß der **precond:** – Teil von CS_P entweder leer sein oder der angegebene Guard sich bzgl. D'_P zu *true* evaluieren lassen. Falls das nicht der Fall ist, wird der Prozeß der Instanziierung an dieser Stelle abgebrochen.

Schritt 1.3: Bindung der formalen Parameter an aktuelle Werte
In diesem Schritt wird der Softwareentwickler nach konkreten Werten für die im Constraint-Schema CS_P aufgeführten formalen Parameter gefragt. Im **ocl:** – Teil von CS_P werden anschließend die formalen Parameter syntaktisch durch die entsprechenden konkreten Werte ersetzt und man erhält einen OCL-Ausdruck C_P.

Schritt 2: Definition der Signaturabbildung σ

Schritt 3: Erzeugung des resultierenden Entwurfs D mit Constraint C durch Anwendung der Signaturabbildung σ auf D'_P und C_P wobei gilt:
$D = \sigma'(D'_P)$ und $C = \sigma'(C_P)$

Anpassungsschritte am einfachen Beispiel Zur Illustration passen wir das Singleton Muster aus Abschnitt 4.1 für ein denkbares Szenario an. Ziel ist ein Entwurf mit einer Klasse *StandardFolder*, die ein Attribut *nthNumber* besitzt und von der höchstens 7 Instanzen existieren dürfen, was mit Hilfe von *boundInstancesAlt* ausgedrückt werden soll. In diesem Fall sind folgende Anpassungsschritte vonnöten:

Schritt 1.1: Strukturanpassung des Musters durch Einfügung des Attributs *id*

$D'_P =$

Singleton
id: Integer

Schritt 1.2: Test der Vorbedingung
Die in *boundInstancesAlt* enthaltende Vorbedingung

```
Singleton.attributes->includes('id') and
Singleton.allInstances->
            forAll(x | x.id.oclIsKindOf(Integer))
```

muß für das Diagramm D'_P ausgewertet werden. Diese Auswertung ergibt *true*. Somit kann mit dem Schritt 1.3 fortgefahren werden.

Schritt 1.3: Bindung der formalen Parameter
Für den formalen Parameter MAX wird vom Softwareentwickler ein aktueller Wert erfragt, in diesem Beispiel der Wert 7. Nach syntaktischer Ersetzung im **ocl:** – Teil von *boundInstancesAlt* ergibt sich für C_P:

```
context Singleton inv:
   Singleton.allInstances->
      forAll(x | 0 < x.id and x.id <= 7 and
        forAll(y | x <> y implies x.id <> y.id))
```

Schritt 2: Angabe der Signaturabbildung

Der Benutzer wählt $\sigma = \{\mathit{Singleton} \mapsto \mathit{StandardFolder},\ \mathit{id} \mapsto \mathit{nthNumber}\}$.

Schritt 3: Automatische Generierung des resultierenden Diagramms mit gewünschten Constraints:


```
context StandardFolder inv:
   StandardFolder.allInstances->
      forAll(x | 0 < x.nthNumber and x.nthNumber <= 7 and
        forAll(y | x <> y implies
                    x.nthNumber <> y.nthNumber))
```

4.3 Ein reales Anwendungsbeispiel

Constraint-Schemata haben sich im *keyMail/S*-Projekt vielfach bewährt. In diesem Abschnitt wird ein Ausschnitt daraus vorgestellt. Die hierbei verwendeten Constraint-Schemata sind in ihrem Aufbau typisch. Sie verwenden mit den sogenannten *Metakonstrukten* fortgeschrittene Konzepte der OCL. Die Anwendung der Metakonstrukte wird in Abschnitt 4.4 näher diskutiert.

Das folgende Beispiel basiert auf dem Composite Muster. Es dient zur Realisierung graphenartiger Datenstrukturen bestehend aus Knoten und Kanten. In der Beschreibung des Musters werden die Knoten in innere Knoten (*Composite*) und Blattknoten (*Leaf*) unterteilt.

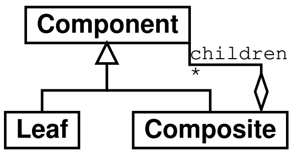
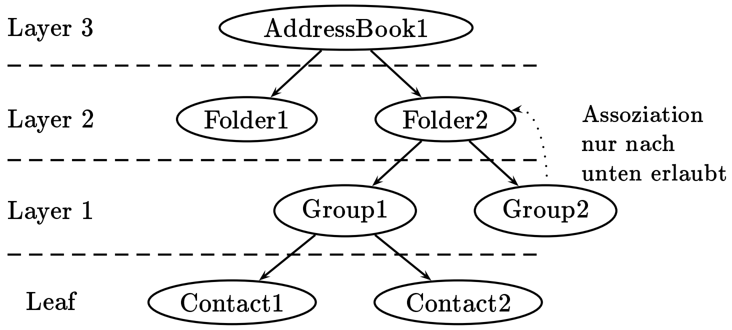


Abbildung2. Klassendiagramm des Musters Composite

In *keyMail/S* wird eine Datenstruktur „Geschichteter Baum“ für die Realisierung eines Adreßbuches benötigt. Ein Adreßbuch besteht aus Verzeichnissen, jedes Verzeichnis aus sogenannten Kontakt-Gruppen, jede Kontakt-Gruppe aus Kontakten. Diese strenge Hierarchisierung soll bei Bedarf etwas abgeschwächt werden, indem auch Sprünge zu weiter unten liegenden Knotentypen erlaubt sind.

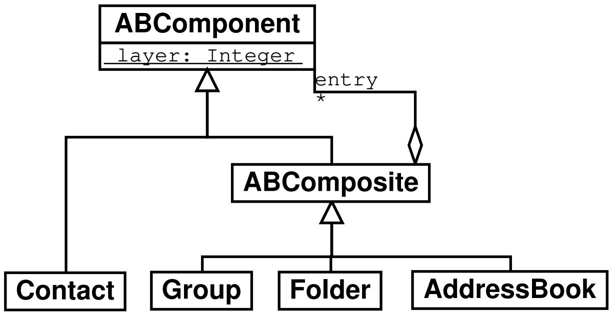


Die Spezifikation der informell beschriebenen Datenstruktur gelingt sehr elegant mit folgendem Constraint-Schema. Dabei wird *layer* als neues Klassenattribut von *Component* mit Typ *Integer*, welches die Hierarchiestufe einer Klasse codiert, vorausgesetzt. Die Hierarchiestufe *layer* ist zwar inhaltlich nur für Unterklassen von *Composite* relevant, aber aus Gründen der Typkorrektheit als Klassenattribut von *Component* und nicht von *Composite* realisiert. Der formale Parameter *flavour* legt fest, ob es sich um eine strenge Hierarchisierung handelt.

```
schema isLayeredGraph(String flavour)
   precond:
      Component.attributes->includes(‘layer‘) and
      Component.allInstances->
                forAll(x | x.layer.oclIsKindOf(Integer))

   ocl: context  Composite inv:
      let subtypes = OclType.allInstances->select( c |
         c.allSupertypes->includes(Composite)) in
          subtypes->collect( c | c.layer ) =
               Bag { 1 .. subtypes->size} and
          if (flavour = ’strong’) then
           self.children->forAll( x |
                (x.layer = self.layer - 1 ) or
                (self.layer = 1 and x.oclIsKindOf(Leaf)))
          else
           self.children->forAll( x |
                x.layer <= self.layer or
                x.oclIsKindOf(Leaf))
          endif
```

Der Softwareentwickler muß das Composite Muster und das Constraint-Schema *isLayeredGraph* geeignet instanziieren. Wir verzichten an dieser Stelle aus Platzgründen auf die Beschreibung der Schritte 1.1 – 3 und präsentieren nur das Ergebnis der Anpassung:




```
context ABComposite inv:
let subtypes = OclType.allInstances->select( c |
   c.allSupertypes->includes(ABComposite)) in
    subtypes->collect( c | c.layer ) =
          Bag { 1 .. subtypes->size} and
    if ('strong' = 'strong') then
     self.entry->forAll( x |
          (x.layer = self.layer - 1 ) or
          (self.layer = 1 and x.oclIsKindOf(Contact)))
    else
     self.entry->forAll( x |
          x.layer <= self.layer or
          x.oclIsKindOf(Contact))
    endif
```

4.4 OCL als Schema-Sprache

Entwurfsmuster sind auf das Wesentliche reduziert und das ist ein wichtiger Grund für ihren Erfolg. Dies macht es allerdings notwendig, Entwurfsmuster durch strukturelle Veränderungen (z.B. Hinzufügen von Unterklassen) auf die aktuelle Situation anzupassen.

Ein überraschendes Ergebnis unserer Untersuchung ist, daß es einer solchen *strukturellen* Anpassung für Constraint-Schemata nicht bedarf. Der im **ocl** Abschnitt eines Constraint-Schemas aufgeführte Spezifikationstext kann strukturell unverändert, d.h. lediglich durch Anpassung von Signatur und Einsetzen aktueller Parameter modifiziert, in den resultierenden Entwurf übernommen werden. Es wäre nicht abwegig zu vermuten, daß zur Formulierung von Constraint-Schemata neben OCL auch metasprachliche Erweiterungen vonnöten sind, die den Mechanismus der Anpassung eines Musters reflektieren. In der Tat favorisierten wir zunächst einen solchen Ansatz [9].

Überraschenderweise ist dies jedoch überflüssig, da OCL schon durch den Typ *OclType* und den darauf definierten Attributen *attributes*, *operations*, *supertypes* und *allInstances* den Zugriff auf die Metaebene ermöglicht. Somit lassen sich für eine beliebige Klasse des UML-Modells ihre Attribute, Methoden, Subklassen und Instanzenmenge bestimmen. Dies erlaubt eine Formulierung von Constraints relativ unabhängig von der Anwendung in einem konkreten UML-Modell. Das Beispiel in Abschnitt 4.3 nutzt diesen Umstand aus.

4.5 Optimierungen

Ist ein UML-Modell strukturell fixiert, so können die generierten Constraints syntaktisch vereinfacht werden. Diese Vereinfachung dient lediglich dazu, die Constraints für den Softwareentwickler übersichtlicher aufzubereiten. Eine Änderung der Semantik ist damit nicht verbunden. Wir diskutieren zwei offensichtliche Vereinfachungen:

Tote Zweige in `if-then-else`. Ausdrücke wie `if X = X then E1 else E2` entstehen häufig durch Variantenbildung von Mustern. Der `if`-Ausdruck läßt sich durch `E1` ersetzen. Ein Beispiel dafür ist der Teilausdruck mit der Bedingung `'strong' = 'strong'` im letzten Constraint von Abschnitt 4.3.
Schwieriger ist die Vereinfachung von `if X = Y then E1 else E2` bei syntaktisch unterschiedlichen `X` und `Y`. Oftmals kann man aber `X <> Y` aufgrund ihrer Typisierungen folgern, z.B., wenn `X`, `Y` vom Typ *String* sind. In diesen Fällen kann man den `if`-Ausdruck durch `E2` ersetzen.

Auflösung von Meta-Konstrukten. Die generierten Constraints enthalten eine Reihe von OCL-Meta-Konstrukten. Typisch ist folgende Struktur[2]:

```
context ClassA inv:
   ClassA.allSubtypes()->
      forAll(type | type.allInstances->
        forAll(inst | inst.expr))
```

Wenn im endgültigen Entwurf alle Subklassen von `ClassA` feststehen, angenommen dies seien `SubClass1` und `SubClass2`, so kann man obige Struktur in zwei Schritten vereinfachen:

Schritt 1: Einsetzung der Untertypen:

```
context ClassA inv:
   SubClass1.allInstances->forAll(inst | inst.expr) and
   SubClass2.allInstances->forAll(inst | inst.expr)
```

Schritt 2: Auflösen von **context** `ClassA`

```
context SubClass1 inv:
   expr

context SubClass2 inv:
   expr
```

[2] Der Ausdruck `Class.allSubtypes()` ist eine Abkürzung für `OclType.allInstances->select(type| type.allSupertypes->includes(Class))`

5 Zusammenfassung und Ausblick

Wir haben in dieser Arbeit unser Konzept vorgestellt, den Entwickler von UML-Spezifikationen bei der Erstellung von OCL-Constraints zu unterstützen. Damit ist ein leichterer Zugang zu formalen Methoden und die Nutzung der damit verbundenen Vorteile wie größere Klarheit des Entwurfs, Anwendung von Verifikationstechniken, etc. möglich.

Die Methode, OCL-Schemata an Entwurfsmuster zu koppeln, hat sich in einer Reihe von Beispielen bewährt. OCL-Schemata müssen mögliche Erweiterungen des Musters berücksichtigen. Deshalb ist es entscheidend, daß die Sprache OCL nicht nur die Formulierung von Constraints für eine konkrete Modellierung, sondern, dank der Metakonstrukte in OCL, auch für eine Klasse von Modellierungen erlaubt.

Uns sind keine anderen Arbeiten bekannt, die eine mustergesteuerte Erzeugung von OCL-Constraints im Softwareentwurf vorschlagen. Am nächsten verwandte Arbeiten sind *Spezifikations*-Muster [2] zur Formalisierung von Eigenschaften endlicher Zustandssysteme. Hier werden formale Spezifikationen selber zum Bestandteil von Mustern und der Kontext ist System- nicht Softwareverifikation. Der bekannte Design-by-Contract Ansatz [6] umfaßt auch die Erzeugung formaler Constraints, ist jedoch auf der Implementierungsebene angesiedelt und es gibt keine systematischen Hinweise, wie man zu Constraints kommt.

Wir haben uns in dieser Arbeit konzentriert auf Entwurfsmuster, die in der Klassifikation von [3] erzeugende oder strukturelle Muster (creational or structural patterns) heißen. In [9] werden auch Verhaltensmuster (behavioural patterns) betrachtet. Die Erweiterung dieser Kategorie von Mustern durch Constraint-Schemata stellt neue Anforderungen, die noch nicht ausreichend untersucht sind.

Viele CASE-Werkzeuge unterstützen den Anwender bei der Einbindung von Entwurfsmustern in eine UML-Modellierung. Der hier vorgestellte Ansatz der Constraint-Schemata geht über diese Möglichkeiten hinaus und wurde im Rahmen des KeY-Projekts[3] bereits prototypisch als Erweiterung des Entwicklungswerkzeugs TogetherJ implementiert.

Eine langfristige Aufgabe wird sein, die Erfassung weiterer OCL-Schemata zu Entwurfsmustern in zusätzlichen Anwendungsstudien voranzutreiben. Entwurfsmuster fassen die im Laufe der Zeit von kompetenten Spezialisten gemachten Erfahrungen bei der Lösung wiederkehrender Entwurfsprobleme zusammen, wir hoffen, daß man von OCL-Schemata, als Ergänzung zu Entwurfsmustern, einmal dasselbe wird sagen können.

Danksagung

Wir danken der Firma *fun communications* GmbH, Karlsruhe für ihre Zusammenarbeit bei der Durchführung der hier vorgestellten Untersuchungen sowie für die freundliche Erlaubnis zur Veröffentlichung von Teilentwürfen des Projekts *keyMail/S*. Unser persönlicher Dank gilt den Mitarbeitern Thomas Fuchß

[3] `http://i12www.ira.uka.de/~key/`

und Dirk Arnold, bei denen wir für unsere Fragen immer ein offenes Ohr und großzügige Unterstützung gefunden haben.

Die Arbeiten, über die hier berichtet wurde, wurden von der Deutschen Forschungsgemeinschaft im Rahmen des Projektes *Integrierter Deduktiver Softwareentwurf*, Kennzeichen Ha 261/2-1, gefördert.

Literatur

1. Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad und M. Stal: *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, New York, 1996.
2. Dwyer, M. B., G. S. Avrunin und J. C. Corbett: *Patterns in Property Specifications for Finite-State Verification*. In: *Proc. 21st International Conference on Software Engineering*, S. 411–420. IEEE Computer Society Press, ACM Press, 1999.
3. Gamma, E., R. Helm, R. Johnson und J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading/MA, 1995.
4. Holloway, C. M. und K. J. Hayhurst (Hrsg.): *Fourth NASA Langley Formal Methods Workshop*, Nr. 3356 in *NASA Conference Publication*, Hampton, Viginia, 1997.
5. Knight, J. C., C. L. DeJong, M. S. Gibble und L. G. Nakano: *Why Are Formal Methods Not USED More Widely?*. In: Holloway, C. M. und Hayhurst [4], S. 1–12.
6. Meyer, B.: *Applying "Design by Contract"*. IEEE Computer, 25(10):40–51, Okt. 1992.
7. Object Modeling Group: *Unified Modelling Language Specification, version 1.3*, Juni 1999. URL: `uml.shl.com:80/docs/UML1.3/99-06-08.pdf`.
8. Rumbaugh, J., I. Jacobson und G. Booch: *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.
9. Sattler, T.: *Einbindung formaler Constraints in UML Spezifikationen*. Interner Bericht 2000-16, Fakultät für Informatik, Universität Karlsruhe, Juni 2000.
10. Unisys Corp. et al.: *XML Metadata Interchange (XMI)*, Okt. 1998. URL: `ftp://ftp.omg.org/pub/docs/ad/98-10-05.pdf`.
11. Warmer, J. und A. Kleppe: *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.