Grosser Beleg

Konzeption eines Codegenerators zur Überwachung nicht-funktionaler Eigenschaften

bearbeitet von

Torvald Riegel

geboren am 1.3.1979

Technische Universität Dresden
Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Betreuer: Dipl.-Inf. Steffen Zschaler
Hochschullehrer: Dr.-Ing. Th. Santen

Eingereicht am 31. Januar 2004

Inhaltsverzeichnis

1 Einleitung **1**

2 Problemanalyse **4**
2.1 Problemumgebung 4
2.2 Anwendungsfälle 8
2.3 Problemgliederung 10

3 Konzeptionelle Lösung **13**
3.1 Verantwortlichkeiten im Entwicklungsprozess 13
3.2 Messverfahren 16
3.2.1 Stichprobenartiges Messen 18
3.2.2 Änderungsgesteuertes Messen 19
3.3 Mess-Kontextmodelle 21
3.3.1 OCL-`values`-Klausel und Mess-Kontextmodell 21
3.3.2 Abbildung zwischen Mess-Kontextmodell und Container 23
3.3.3 Anordnung von Zugriffen auf den Mess-Kontext . . . 27
3.4 Messfunktion 30
3.4.1 Umgebung von OCL-Ausdrücken in `values`-Klauseln . 30
3.4.2 Datentypen in CQML$^+$, OCL und im Kontextmodell . 33
3.4.3 Generieren von Mess-Code 35
3.5 Einbettung des Messsystems in den Container 38
3.6 Unterschiede zur CQML$^+$-Spezifikation 39
3.6.1 Statistische Aspekte 39
3.6.2 Invarianten 40

4 Prototyp **42**

5 Zusammenfassung und Ausblick **44**

Literatur **46**

Abbildungs- und Listingverzeichnis **48**

1 Einleitung

Folgend wird eine Lösung zum Messen von nicht-funktionalen Eigenschaften im Komponenten-Framework des COMQUAD-Projektes[1] vorgestellt. Dabei werden ausgehend vom zugrundeliegenden Komponentenmodell über ein frei wählbares Kontextmodell und der mittels OCL[2] beschriebenen Semantik der Eigenschaft Messwerte berechnet. Es wird die Implementation eines Prototypen erläutert.

Nicht-funktionale Eigenschaften von Komponenten können während der Laufzeit der Komponente unterschiedliche Werte annehmen, da sie abhängig vom Verhalten der Komponenten und Zuständen und Ereignissen in der Laufzeitumgebung der Komponenten sind. Die aktuellen Werte einer Eigenschaft sind Voraussetzung für viele Mechanismen wie zum Beispiel Adaption und natürlich für das Überprüfen von Zusagen einer Komponente zu einer bestimmten Leistung.

Um den Wert einer nicht-funktionalen Eigenschaft ermitteln zu können, muss diese zuerst spezifiziert werden. Bestimmte Vorstellungen von quantitativen Maßen, wie zum Beispiel Antwortzeit, werden dabei als Funktion über ein Systemmodell beschrieben. Im COMQUAD-Projekt wird CQML$^+$ [RZ03a] als Spezifikationssprache verwendet. Diese trennt zwischen der Definition eines Maßes und der Angabe einer konkreten Eigenschaft durch Zuordnen eines Maßes zu einem Teil der funktionalen Schnittstelle einer Komponente.

Es wird bei der Spezifikation der Eigenschaft nicht angegeben, wie der Wert der Eigenschaft tatsächlich berechnet wird, sondern wie er sich aus dem Zustand des Systems ableitet. Diese Abbildung wird mit einem OCL-Ausdruck beschrieben und im Folgenden Messfunktion genannt. OCL ist eine reine Abfragesprache und verändert das System nicht. Zusätzlich wird von der Sprache definiert, das Abfragen keine Zeit benötigen. Das Ermitteln des Wertes einer Eigenschaft ist somit atomar und Zustände des Systems ändern sich während der Berechnung nicht. Daraus folgt, dass Autoren und Anwender von nicht-funktionalen Eigenschaften unabhängig von der Laufzeitumgebung und den Vorgängen in dieser sind.

Da Berechnungen aber natürlich Zeit und andere Ressourcen benötigen, muss die Laufzeitumgebung dafür sorgen, dass der Messfunktion während einer Ausführung ein unveränderlichen Zustand des Systems zur Verfügung steht. Die Komponenten sind nicht dafür geeignet, Werte von Eigenschaften zu ermitteln, da zum Beispiel das Überprüfung von Zusagen durch eine neutrale Stelle vorgenommen werden sollte.

[1] COMponents with QUantitative properties and ADaptivity, siehe [ABF$^+$03] und `http://www.comquad.org`

[2] Object Constraint Language [OCL03]

Um also Werte von nicht-funktionalen Eigenschaften messen zu können, muss die Spezifikation der Eigenschaft in eine ausführbare Funktion umgewandelt werden (Codegenerator), und die Laufzeitumgebung muss eine Messung unterstützen. Zusätzlich muss aus dem Zweck der Messwerterhebung das Messverfahren abgeleitet werden, das heißt wie und wann welche Einzelmessungen durchzuführen sind.

Momentan gibt es innerhalb des COMQUAD-Projektes keine detaillierten Vorstellungen, wie die eben genannten drei Punkte umzusetzen sind. In dieser Arbeit soll deshalb das Aufgabengebiet erkundet und strukturiert werden. Dafür sind folgende **Teilprobleme** zu lösen oder detailliert darzustellen:

A.1 Anwendungsfälle für das Erheben von Messwerten und die daraus resultierenden Anforderungen an die Messwert-Ermittlung müssen beschrieben werden.

A.2 Es müssen Messverfahren definiert werden, die diese Anforderungen erfüllen.

B.1 Die Messverfahren stellen wiederum bestimmte Anforderungen an die Laufzeitumgebung. Diese Anforderungen sind zu beschreiben.

B.2 Auf den Systemzustand wird mittels zweier Modellschichten zugegriffen: Kontext- und Komponentenmodell. Es ist zu untersuchen, was in diesen wie zu spezifizieren ist und welchen Einschränkungen sie unterliegen.

B.3 Es müssen Mechanismen gefunden werden, mit denen die Anforderungen der Messverfahren in der Laufzeitumgebung unterstützt und Kontext- und Komponentenmodelle implementiert werden können.

C.1 Das Generieren von Code aus der spezifizierten Messfunktion ist zu erläutern und es muss beschrieben werden, über welche Schnittstelle der Code auf den Systemzustand zugreift.

C.2 Die Spezifikationsmöglichkeiten von nicht-funktionalen Eigenschaften stellen Anforderungen an das Messverfahren und die Laufzeitumgebung. Es sollten verschiedene Varianten bei der Gestaltung der Messfunktion auf ihre Anforderungen hin untersucht werden.

C.3 Der generierte Messcode muss genauer untersucht werden hinsichtlich Ressourcenverbrauch und möglicher Optimierungen.

C.4 Als Machbarkeitsnachweis ist ein Prototyp zu erstellen, der aus Spezifikationen von nicht-funktionalen Eigenschaften ausführbaren Code mit der Messfunktion erstellt.

In den folgenden Kapitel wird das Problem ähnlich zu den Phasen bei der Software-Entwicklung bearbeitet:

Problemanalyse: Kapitel 2 auf der nächsten Seite untersucht das Aufgabengebiet. Dazu wird die Umgebung des Problems detaillierter beschrieben und die Aufgabe abgegrenzt. Danach werden die Anwendungsfälle erläutert und das Problem strukturiert dargestellt. Abschließend werden die Änderungen und Verfeinerungen an der Aufgabenstellung genannt, die sich während der Bearbeitung als sinnvoll herausstellten.

Konzeptionelle Lösung: In Kapitel 3 auf Seite 13 werden die einzelnen Teilprobleme untersucht beziehungsweise gelöst. Unter anderem wird das Messverfahren definiert sowie Kontext- und Komponentenmodelle und die Messfunktion betreffendes behandelt.

Prototyp: Letztendlich wird in Kapitel 4 auf Seite 42 kurz die Implementierung des Prototyps beschrieben.

Anschließend werden die Ergebnisse der Arbeit zusammengefasst und es wird ein Ausblick gegeben auf weiterhin bestehende und neu aufgetretene Aufgaben.

Als Ergänzung zu dieser Einleitung empfiehlt sich Kapitel 2.3 auf Seite 10, Problemgliederung. Es enthält eine detailliertere Untersuchung und Strukturierung der Aufgabe, als es diese Einleitung bieten kann. Dabei wird auch die Beziehung zur konkreten Problemumgebung und den Anwendungsfällen hergestellt. Gegebenenfalls sollte man also vorher die davor liegenden Abschnitte des Kapitels 2 lesen, wenn man nicht näher mit dem Problem vertraut ist.

Im Text werden wichtige oder neu eingeführte Bezeichnungen *kursiv hervorgehoben.* Aus Spezifikations- oder anderen Sprachen entnommene Schlüsselwörter und Bezeichner sind in einer nicht-proportionalen Schrift (`Maschinenschrift`) gehalten. Das gleiche gilt auch für ganze Ausdrücke in diesen Sprachen, einige Schlüsselwörter sind dort zusätzlich fett gedruckt.

2 Problemanalyse

In diesem Kapitel wird das Aufgabengebiet abgegrenzt und erläutert und die für die Aufgabe relevanten Anwendungsfälle beschrieben. Bezugnehmend darauf wird das Problem anschließend strukturiert. Die sich daraus ergebenden Änderungen und Verfeinerungen der Aufgabenstellung werden genannt.

2.1 Problemumgebung

Wie schon gesagt wurde, ist der Kontext der Aufgabe das COMQUAD-Projekt. Somit gilt dies auch für alle Anwendungsfälle und weiteren Anforderungen. Es folgt ein zusammenfassender Überblick über für die Aufgabe relevante Teile des Projektes.

In [ABF+03] werden die Teilprojekte genannt, in die COMQUAD gegliedert ist. Das Messen von nicht-funktionalen Eigenschaften hat Bezug zu zwei von diesen:

Spezifikation von nicht-funktionalen Eigenschaften: Die Spezifikation dient als Definition der zu messenden Eigenschaften. Weitere Aufgaben wie die Komposition von Komponenten haben keinen direkten Einfluss, die Möglichkeiten zur Spezifikation sind aber auch auf diese Aufgaben zugeschnitten. Messungen nutzen diese Möglichkeiten, sie stellen aber gegebenenfalls auch Anforderungen an diese oder schränken sie ein. Dieses Teilprojekt wird damit nicht unbedingt durch das Messen beeinflusst, einzelne Änderungen im Rahmen dieser Arbeit könnten aber dennoch auch für weitere Aufgaben vorteilhaft sein.

Containerschnittstelle und -realisierung: Messungen müssen von der Laufzeitumgebung(*Container*) der Komponenten ausgeführt werden. Der Container beeinflusst aber auch die Art und Weise wie gemessen werden kann. Eine Abhängigkeit zwischen dem Teilprojekt und dieser Arbeit besteht somit in beiden Richtungen.

Die zur Spezifikation von nicht-funktionalen Eigenschaften verwendete Sprache CQML+ [RZ03a] ist eine Erweiterung von CQML [Aag01]. Von beiden werden drei parametrisierbare Konstrukte zur Definition von Zusagen verwendet:

Charakteristiken definieren eine nicht-funktionale Eigenschaft inklusive deren Semantik.

Statements sind boolsche Ausdrücke über Werte von Charakteristiken.

Profiles binden Statements an Teile der funktionalen Spezifikation von Komponenten.

CQML$^+$ definiert zusätzlich noch eine Möglichkeit zur Spezifikation des Ressourcenbedarfs. Dafür kann aber kein Messcode generiert werden, denn die genaue Semantik der einzelnen Ressourcen-Typen wird durch den verwendeten Resource-Manager bestimmt und nicht mittels einer Funktion spezifiziert. Außerdem sollten Zusagen immer durch eine neutrale Instanz überprüft werden. Im Falle der Zusagen von Komponenten vertrauen diese dem Container und seinen Messungen, beim Ressourcenbedarf würde aber der Container seine eigenen Zusagen „überprüfen“. Davon abgesehen kann die für normale Charakteristiken gefundene Lösung später auch auf Ressourcen übertragen werden, wenn die Semantik der Ressourcen über eine Funktion oder eine andere geeignete Schnittstelle zugänglich gemacht wird. Genaueres zur Angabe des Ressourcenbedarfs findet sich in [RZ03a].

Eine *Messfunktion* wird durch eine Charakteristik beschrieben. Statements und Profiles beschreiben eher, welche Messungen durchzuführen sind und welche Messfunktionen dazu mit welchen Parametern ausgeführt werden müssen. Deshalb werden Statements und Profiles im Rahmen dieser Arbeit nur kurz betrachtet. Listing 1 zeigt eine beispielhafte Definition einer Charakteristik mit dem Namen `delay`.

```
quality_characteristic delay(c:OperationCall) {
    domain: decreasing real [0..);
    maximum;
    values: c.returnTime - c.callTime;
}
```

Listing 1: Beispiel für eine Charakteristik-Definition in CQML

Die Domäne der Charakteristik sind die reellen Zahlen (`real`) größer als oder gleich null. Weiterhin stehen noch ganze (`integer`) und natürliche Zahlen (`natural`) zur Verfügung. Alternativ kann die Domäne eine Menge (`set`) oder eine Enumeration (`enum`) sein, bei beiden sind die Elemente als Menge von Bezeichnern zu definieren. CQML$^+$ bietet zusätzlich noch Tupel für strukturierte Charakteristiken an. Mittels `decreasing` wird ausgedrückt, dass der der Wert der Eigenschaft umso besser ist, desto niedriger das Ergebnis der Messfunktion ausfällt. In obigem Fall wird also eine geringe Verzögerung als wertvoller angesehen.

Es können *statistische Aspekte* zur Verfügung gestellt werden, die auf den Ergebnissen der Messfunktion aufbauen. Im Beispiel wird mit Hilfe von `maxmimum` das Maximum der bisherigen Messergebnisse ermittelt.

Die *`values`-Klausel* ist der OCL-Ausdruck (siehe [OCL03]), der auf das Schlüsselwort `values` folgt und die eigentliche Definition der Messfunktion darstellt. Der Ergebnis-Typ des Ausdrucks muss zur Domäne der Charakteristik, also dem Wertebereich der Messfunktion passen. Definitionsbereich sind die Parameter der Charakteristik, in unserem Fall ein Parameter mit dem Namen `c` vom Typ `OperationCall`. Im Beispiel enthalten Objekte dieses Typs zwei Attribute, die jeweils die Zeitpunkte des Aufrufs einer Funk-

tion und der Rückgabe des Ergebnisses enthalten. Nach jedem erfolgten Aufruf werden diese Attribute neu gesetzt.

Die Charakteristik in Listing 1 berechnet also die Zeit, die zum Ausführen einer Operation benötigt wurde.

Abbildung 1: Beispiel für ein Kontextmodell

Kontextmodelle bilden die Umgebung, auf die sich Charakteristiken beziehen. Abbildung 1 zeigt ein Kontextmodell, das als Basis für die obige Charakteristik dienen könnte. Die Klasse `Operation` steht für eine Operation und `OperationCall` für die tatsächlichen Aufrufe dieser Operation, `lastInvocation` zeigt auf den zeitlich letzten von diesen. Für den formalen Parameter `c` könnte dann eine Instanz von `OperationCall` verwendet[3] und damit der Wert der Charakteristik für diese Kontextmodell-Instanz berechnet werden.

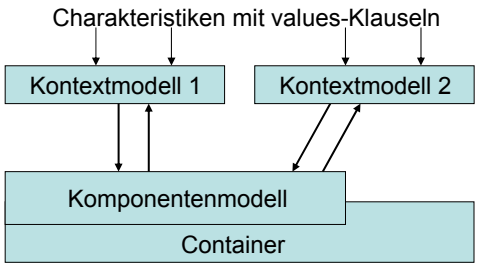

Abbildung 2: Beziehungen zwischen Charakteristiken, Kontext- und Komponentenmodellen

Wie in Abbildung 2 zu sehen ist, können mehrere Charakteristiken auf ein Kontextmodell zugreifen. Es gibt weiterhin verschiedene Kontextmodelle, die unterschiedliche Sichten auf die Umgebung der Komponenten und damit das System, in dem gemessen werden soll, darstellen. Zum Beispiel könnten nur bestimmte Teile für die dieses Kontextmodell nutzende Charakteristiken interessant sein oder die Modelle könnten unterschiedlich stark abstrahieren.

Vor allem aber stellen Kontextmodelle eine Indirektion zwischen Charakteristiken und dem zugrundeliegenden System dar (siehe [RZ03b]).

[3] Anders als CQML [Aag01] schreibt CQML$^+$ [RZ03a] explizit vor, dass tatsächliche Parameter Teile der Instanz des Kontextmodells sein müssen.

Ein *Komponentenmodell* ist eine hauptsächlich statische Sicht auf einen Container. In [GPRZ04] wird das COMQUAD-Komponentenmodell beschrieben. Wichtig für das Messen von Eigenschaften ist dabei, dass alle Schnittstellen einer Komponente durch den Container mit den Schnittstellen des Containers oder anderer Komponenten verbunden werden. Damit kann der Container die entsprechende Verbindungsstücke kontrollieren und ist über alle Interaktionen einer Komponente informiert. Diese Information stehen dann wiederum über das Komponentenmodell für Messungen zur Verfügung. Container-spezifische Eigenschaften können natürlich auch im Komponentenmodell enthalten sein.

Für jedes Kontextmodell existiert eine Abbildung auf das Komponentenmodell, also das zugrundeliegende System. Die tatsächliche Form und Richtung beziehungsweise Richtungen der Abbildung wird innerhalb dieser Arbeit untersucht werden, im Text wird aber vereinfachend immer von einer Abbildung gesprochen.

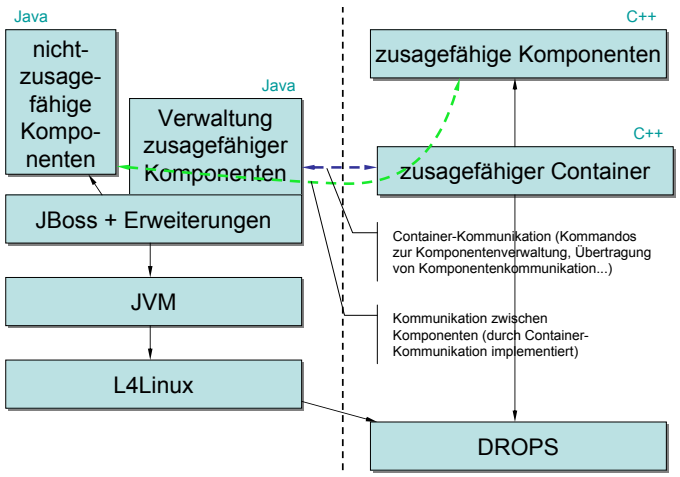

Abbildung 3: Architektur des Containers

Abbildung 3 zeigt die Architektur des Containers, also der Laufzeitumgebung für COMQUAD-Komponenten. Er besteht aus einem zusagefähigen[4] und einem nicht-zusagefähigen Teil (im Folgenden ZF- und NZF-Teil genannt). Der ZF-Teil beherbergt zusagefähige Komponenten und einen Container für diese. Als Betriebssystem ist DROPS (Dresden Realtime OPerating System [HBB+98]) vorgesehen. Im NZF-Teil laufen die nicht-zusagefähigen Komponenten auf Basis von L4Linux und einem modifizierten JBOSS-Applicationserver[5]. Komponenten sind im ZF-Teil in C++ und im NZF-Teil

[4]Zusagen im Sinne von Echtzeit-Eigenschaften, also zum Beispiel das Vollenden einer Aufgabe vor einem bestimmten Zeitpunkt oder Aussagen über den maximalen Ressourcenverbrauch

[5]`http://www.jboss.org`

in Java implementiert.

2.2 Anwendungsfälle

Im Folgenden sollen relevante Anwendungsfälle gefunden werden. Dazu werden zuerst einige mögliche Gründe für das Durchführen von Messungen genannt. Danach werden diese auf Unterschiede und Gemeinsamkeiten untersucht und daraus die umzusetzenden Anwendungsfälle ermittelt.

Wenn zuerst nur nach den Gründen gesucht werden soll, aufgrund derer zu Messen ist, dann müssen auch nur zwei Akteure beachtet werden.

Der *Auftraggeber der Messung* hat Interesse an dem Wert einer nichtfunktionalen Eigenschaft unter folgenden Gesichtspunkten:

- Das Interesse erstreckt sich auf einen bestimmten Zeitraum oder Zeitpunkt.
- Die Messergebnisse müssen dem Auftraggeber unterschiedlich schnell übermittelt werden, entweder in Bezug zum Auftrag oder durchgeführten Einzelmessungen.

Dem gegenüber steht das *Messsystem*, dass die Messungen durchführt (zum Beispiel der Container). Es nimmt Messaufträge entgegen und gibt Ergebnisse an den Auftraggeber zurück. In Bezug auf die Messergebnisse besteht also ein Produzent-Konsument-Verhältnis zwischen Messsystem und Auftraggeber. Die nicht-funktionalen Anforderungen des Auftraggebers an dieses Verhältnis sind vom Messsystem entsprechend umzusetzen.

Es folgt eine Auflistung von wichtigen Gründen für das Durchführen von Messungen:

Protokollierung: Bei der Protokollierung sollen frühere Systemzustände später nachvollzogen werden können. Messungen können zum einen in der Form von Stichproben durchgeführt werden, periodisch oder in unregelmäßigen Abständen. Zum anderen muss es möglich sein, alle Änderungen des Messwertes in einem bestimmten Zeitraum aufzeichnen zu können, damit der Verlauf lückenlos erfasst wird. Die Messergebnisse müssen nicht sofort verfügbar sein, die Messung muss aber zum angeforderten Zeitpunkt stattfinden.

Überprüfen von Zusagen: Zusagen werden gewöhnlich für einen bestimmten Zeitraum gegeben, der Messwert ist also lückenlos zu beobachten. Das die Zusage repräsentierende Kriterium wird entweder eingehalten oder nicht. Weniger strikte Kriterien sollten mit entsprechend weich formulierten nicht-funktionale Eigenschaften, zum Beispiel mittels statistischer Aspekte, ausgedrückt werden.

Wo die Zusage geprüft wird, also beim Messsystem oder beim Auftraggeber, macht keinen grundsätzlichen Unterschied. Entweder entsteht mehr Berechnungsaufwand beim Messsystem und es werden weniger umfangreiche Ergebnisse produziert, da das Ergebnis nur ein Eingehalten/Nicht-Eingehalten Wert ist. Oder das Messsystem wird entlastet, die Domäne des Messergebnisses ist entsprechend umfangreicher und der Auftraggeber hat den Aufwand der Überprüfung.

Verletzte Zusagen haben in der Regel Konsequenzen, auf die reagiert werden muss. Falls die Verletzung nur bemerkt werden muss, ist das äquivalent zur Protokollierung des Kriteriums. Andere Reaktionen könnten aber zum Beispiel bei kritischen Eigenschaften das Herunterfahren des Gesamtsystems oder das Umschalten auf eine funktionierende Alternative sein. Dafür müssen die Messergebnisse sofort an den Auftraggeber übermittelt werden, damit dieser schnell reagieren kann.

Adaption: Hierbei führt der Auftraggeber Messungen durch, um auf bestimmte Zustände im System reagieren zu können. Die Verzögerung bei der Übermittlung der Messergebnisse ist relevant, da gegebenenfalls reagiert werden muss. Es sollen Stichproben genommen werden können, wenn Adaptionsalgorithmen am aktuellen Zustand oder dem Zustand zu einem bestimmten Zeitpunkt interessiert sind und ihre Aktionen der Situation anpassen wollen. Andererseits soll auf unerwartete Änderungen des Messergebnisses reagiert werden können. Dafür muss das Messsystem die Eigenschaft kontinuierlich überwachen und alle Änderungen dem Auftraggeber mitteilen.

Beobachten des Systems: Während der Entwicklung des Containers oder von Komponenten kann es nützlich sein, dass die Entwickler über bestimmte Eigenschaften informiert sind. Das können sowohl Stichproben zu bestimmten Zeitpunkten sein, um sich einen Eindruck von der aktuellen Situation verschaffen zu können, als auch lückenlose Aufzeichnungen von Messergebnissen. Da ein Mensch letztendlich Konsument der Daten ist, können diese auch mit etwas Verzögerung übermittelt werden.

Die obigen Szenarien zeigen, dass jeweils zwei Punkte zu beachten und charakteristisch für eine Messung sind. Zum einen sind das die *Anforderungen an die Übermittlung* von Messergebnissen im Rahmen des Produzent-Konsument-Verhältnisses zwischen Messsystem und Auftraggeber. Diese schwanken zum Teil auch innerhalb eines Szenarios und können nicht von vornherein eingegrenzt werden.

Der zweite Punkt ist davon unabhängig und betrifft die *Art und Weise der Ermittlung* des Messwertes. Selbst wenn zum Beispiel das Ergebnis einer Stichprobe zu einem bestimmten Zeitpunkt verzögert übermittelt werden kann, so muss die Messung im Messsystem doch zu genau diesem Zeit-

punkt ausgeführt werden. In den Szenarien traten zwei Varianten auf, wie Messwerte ermittelt werden können:

Stichprobenartige Messung: Es wird der Wert einer Eigenschaft zu einen vom Auftraggeber angegebenen Zeitpunkt erfasst. Dies kann zum Beispiel periodisch wiederholt werden.

Änderungsgesteuerte Messung: Der Auftraggeber weist das Messsystem an, für ihn eine Eigenschaft zu überwachen und ihn zu informieren, sobald sich dieser Wert ändert. Für die Messung wird vom Auftraggeber ein Zeitraum angegeben, in dem die Eigenschaft zu überwachen ist.

Diese zwei Arten von Messungen bilden somit die tatsächlich umzusetzenden funktionalen Anwendungsfälle. Die nicht-funktionalen Anforderungen an die Übermittlung bilden eher eine Erweiterung, die Anforderungen an die Implementierung stellt. Die funktionalen Teile werden dadurch aber nicht grundlegend geändert.

2.3 Problemgliederung

Ziel dieses Kapitels ist es, das Problem zu strukturieren, Abhängigkeiten zwischen den einzelnen Teilen zu zeigen und einen Leitfaden für die folgenden Kapitel zu geben. In der Einleitung wurde die Aufgabe in Teilprobleme gegliedert. Dabei entstanden drei Bereiche:

A) Anwendungsfälle und Messverfahren

B) Anforderungen an das die Messung durchführende System

C) Generieren von Code aus der Messfunktion

Kapitel 2.2 auf Seite 8 beschäftigte sich schon mit den Anwendungsfällen (Teilproblem A.1). Dabei wurden die Akteure *Auftraggeber* und *Messsystem* beschrieben. Ziel dieser Arbeit ist das Erstellen des Messsystems, dass dann durch externe Auftraggeber genutzt werden kann.

Teilproblem A.2 war die Definition von *Messverfahren*. Die Semantik von nicht-funktionalen Eigenschaften ist durch die Messfunktion rein statisch beschrieben. Wie die Anwendungsfälle aber zeigten, ist das nicht die ganze Sicht auf das Problem, vielmehr sind Auftraggeber an bestimmten Arten von Messungen interessiert: *stichprobenartig* und *änderungsgesteuert*. Kapitel 3.2 auf Seite 16 untersucht, wie Messverfahren dafür prinzipiell umgesetzt werden können und wie die Beziehungen zwischen Zugriffen der Messfunktion auf ein System und Änderungen an diesem System geregelt werden können.

Die Messverfahren legen somit fest, was die genauen Aufgaben des Messsystems sind und welche Anforderungen es in den verbleibenden Problembereichen B und C gibt. Zusätzliche Anforderungen gibt es aber noch durch den für COMQUAD gültigen Entwicklungsprozess und dessen Rollenmodell, weshalb dieses Thema in Kapitel 3.1 auf Seite 13 betrachtet wird.

Die genaue Zuordnung von Verantwortlichkeiten zu Rollen ist wegen der verteilten Entwicklung besonders wichtig. Das Messsystem führt in Form der Messfunktion Code der Auftraggeber aus und ist deshalb auch vor diesem Code zu schützen. So können zum Beispiel bei der normalen Verwendung von fremden Bibliotheken diese getestet werden, bevor das fertige Produkt ausgeliefert wird. Bei Messfunktionen ist das nicht so einfach[6] möglich, denn diese sind dem Messsystem erst zur Laufzeit bekannt.

Zu verhindern sind zum Beispiel nicht erlaubte Veränderungen von Daten, das Ausspähen von Daten oder zu hoher Verbrauch von Ressourcen. Allerdings müssen nicht alle auftretenden Verletzungen absichtlich geschehen. Minimieren lassen sie sich dadurch, dass Fehlverhalten klare und bekannte Konsequenzen für den Verursacher hat, und nur für diesen. Damit werden auftretende Probleme an die Verursacher weitergeleitet und sie werden motiviert, diese zu beheben. Ähnlich kann auch bei Fehlverhalten beliebig anderer Art verfahren werden.

Der Sicherheitsaspekt kann nicht in einem eigenen Kapitel behandelt werden, sondern muss bei jedem Einzelproblem betrachtet werden[7].

Wenn man also die Architektur des Messsystems auf Implementierungs-Ebene betrachtet, gibt es einen eigenen, normal zu entwickelnden Teil und fremden Code, der erst zur Laufzeit generiert wird. Entsprechend lässt sich auch das restliche Problem strukturieren, denn die Schnittstelle zu dem eigenen Code muss schon vor der Laufzeit feststehen. Analog zu den Problembereichen B und C entstehen zwei Aufgaben (siehe auch Abbildung 2 auf Seite 6):

- Kontext- und Komponentenmodelle sind schon vor der Laufzeit bekannt und kommen damit als Schnittstelle zum eigenen Teil des Messsystems in Frage. Die Implementierung der Modelle und ihrer Beziehungen untereinander macht einen wesentlichen Teil des Messsystems aus. Dies wird in Kapitel 3.3 auf Seite 21 behandelt.

- Messfunktionen werden aus der Spezifikation der nicht-funktionalen Eigenschaften in Form der `values`-Klauseln generiert. Der Generator

[6] Eventuell kann die Semantik automatisch analysiert werden, das hängt aber von der konkreten Messfunktion und den Möglichkeiten zur Spezifikation dieser ab.

[7] Vergleiche auch [GPRZ04]: Sicherheit gehört zu den nicht-funktionalen Eigenschaften und ist somit unter Aspekt-orientierter Sicht ein „cross-cutting concern“, also orthogonal zur restlichen Struktur des Systems angelegt.

ist unabhängig vom Messsystem, muss aber Code erstellen, der zu diesem kompatibel ist. In Kapitel 3.4 auf Seite 30 werden deshalb die Umgebung von `values`-Klauseln und die unterschiedlichen Typsysteme in CQML$^+$ und OCL untersucht sowie das eigentliche Generieren von Code beschrieben.

Anschließend wird kurz auf die Einbettung des Messsystems in den Container und Unterschiede zur CQML$^+$-Spezifikation eingegangen und der Prototyp erläutert.

Bei der Bearbeitung der Aufgabe wurde die Prioritäten beziehungsweise der Fokus wie folgt gewählt:

1. **Exploration:** Die Erkundung des Aufgabengebiet hat höchste Priorität, weil das Wissen über das Gebiet entsprechend gering ist und somit die wirklichen Probleme nicht gut geschätzt werden können. Das zeigt sich auch durch die Unterschiede zwischen der Aufgabe und den sich letztendlich als wichtig herausstellenden Problemen.

2. **Messverfahren:** Die ursprüngliche Vorstellung von der Art und Weise des Messens war viel zu ungenau. Die demzufolge gewählten Messverfahren bestimmen die zu lösenden Problem genauer, insbesondere was Anforderungen an das Messsystem betrifft.

3. **Konzeptionelle Lösung:** Einer konzeptionellen Lösung wurde Vorrang vor einer konkreten Lösung, wie zum Beispiel einem Prototypen, gegeben. Dies passt besser zur Exploration und das Aufgabengebiet ist für eine konkrete Lösung im Rahmen dieser Arbeit zu groß.

3 Konzeptionelle Lösung

In diesem Kapitel wird nach konzeptionellen Lösungen der Aufgabe gesucht. Wie dafür vorzugehen ist, wurde schon in Kapitel 2.3 auf Seite 10 betrachtet.

3.1 Verantwortlichkeiten im Entwicklungsprozess

In diesem Kapitel wird untersucht, wie sich das Messen von nicht-funktionalen Eigenschaften in den COMQUAD-Entwicklungsprozess einordnet. Dazu wird dieser zuerst knapp vorgestellt und danach erläutert, welche Artefakte aus diesem wie in das Messen einfließen.

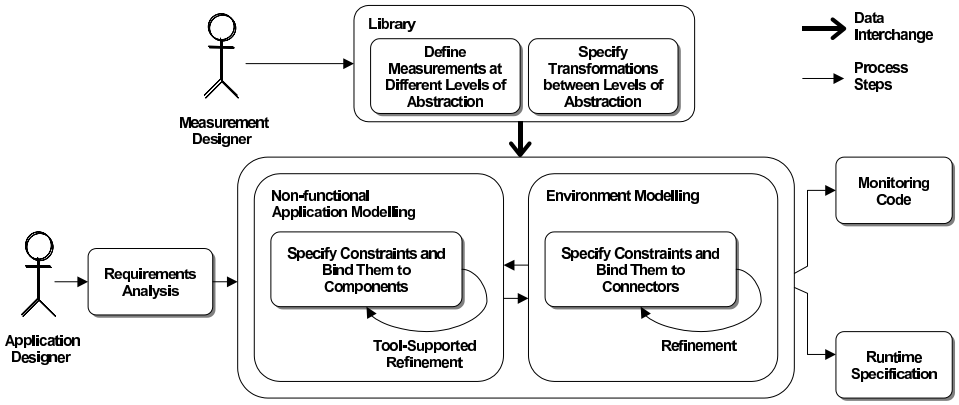

Abbildung 4: Entwicklungsprozess für nicht-funktionale Eigenschaften

Welche Arbeitsschritte bei der Entwicklung im Bereich der nicht-funktionalen Eigenschaften vorkommen, wird in der aus [RZ03b] übernommenen Abbildung 4 gezeigt.

Measurement Designer spezifizieren Charakteristiken[8] und Transformationen zwischen Kontext- und Komponentenmodellen sowie Kontextmodellen untereinander (siehe auch Abbildung 2 auf Seite 6). Diese Charakteristiken sind unabhängig von der Anwendung und können zum Beispiel zu Bibliotheken zusammengefasst werden.

Application Designer verwenden die vorgefertigten Charakteristiken und binden sie an Teile der Anwendung, beziehungsweise bei Komponentenbasierter Entwicklung an die Komponente. Sie können dabei unterschiedliche Kontextmodelle benutzen und mittels den Transformationen zwischen den Modellen die Charakteristiken verfeinert anwenden.

Die resultierenden Charakteristiken bilden dann die Spezifikation des zu generierenden Messcodes („Monitoring Code“). Sie beziehen sich laut [RZ03b] auf ein Komponentenmodell. Es spricht aber auch nichts dagegen,

[8] Measurements bzw. Maße sind äquivalent zu Charakteristiken, in dieser Arbeit wird aber weiterhin der CQML$^+$-spezifische Begriff Charakteristik verwendet.

sie nicht vom Kontextmodell in das Komponentenmodell zu transformieren, vielmehr kann sich der wahrscheinlich höhere Abstraktionsgrad von Kontextmodellen als günstig für das Generieren und Ausführen von Messcode erweisen. Deshalb werden im Folgenden auch beide Varianten betrachtet.

Wie aber auch in Abbildung 2 zu sehen ist, sind Komponentenmodelle, und damit auch Kontextmodelle, nur eine statische, konzeptionelle Sicht auf das System, in dem zu Messen ist. Im weiteren wird der Container stellvertretend für dieses System betrachtet.

Der konkrete Container spielt in obigem Entwicklungsprozess keine Rolle, und es gibt auch keine direkte Abbildung zwischen dem Modell und dem Container. An diesem Punkt ist somit die Schnittstelle zwischen Entwicklung und tatsächlichem Messen zu ziehen. Der Entwicklungsprozess liefert als Produkt ein Modell und Charakteristiken, die sich auf dieses Modell beziehen. Da die Wahl zwischen Kontext- und Komponentenmodell offen gelassen werden sollte, wird dieses Modell ab jetzt als *Mess-Kontextmodell* bezeichnet. Es stellt den Kontext für Charakteristiken dar, die dem Messsystem vom Auftraggeber einer Messung übergeben werden. Eine Instanz des Mess-Kontextmodells wird *Mess-Kontext* genannt.

Das Messsystem ist für das Ausführen von Messungen verantwortlich und muss somit folgende Aufgaben erledigen:

- Es muss einen Mess-Kontext verwalten und eine Abbildung vom Modell zum Container herstellen. Ereignisse und Zustände im Container müssen sich also im Modell wiederfinden. Wie das genau erledigt werden kann, wird erst in Kapitel 3.3.2 auf Seite 23 besprochen.

- Die auf das Mess-Kontextmodell zugreifenden Messfunktionen müssen ausgeführt werden.

In Abbildung 5 auf der nächsten Seite ist skizziert, wie aus einer Charakteristik beziehungsweise Messfunktion, die sich auf ein Kontextmodell bezieht, ausführbarer Messcode wird. Die linke Seite betrachtet die `values`-Klausel der Charakteristik, die rechte das Modell, auf das die `values`-Klausel zugreift.

Die oberste Stufe zeigt den Anfangszustand, vom Measurement Designer erstellte `values`-Klauseln beziehen sich auf ein Kontextmodellund werden vom Application Designer verwendet.

Die zweite Stufe enthält die Spezifikation, die dem Messsystem übergeben werden kann. Es wurde ein Mess-Kontextmodell ausgewählt, das dem Messsystem bekannt ist und auf das der Container abgebildet werden kann. Die `values`-Klausel wurde mittels der bekannten Beziehung zwischen dem Kontextmodell und dem Mess-Kontextmodell transformiert, ähnlich zur Transformation zwischen Kontextmodellen in [RZ03b].

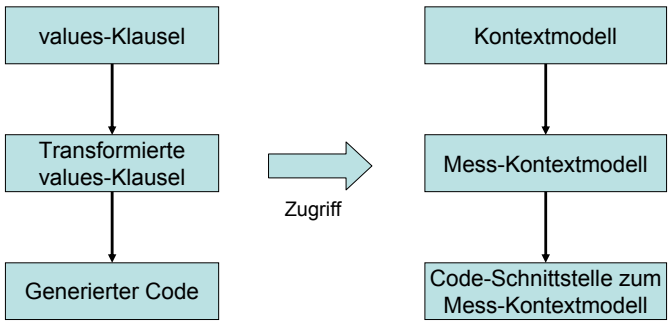

Abbildung 5: Transformationen von `values`-Klauseln und Modellen

Die letzte Transformation wird vom Messsystem vorgenommen. Die transformierte `values`-Klausel wird in ausführbaren Messcode umgewandelt. Es wird vom Messsystem eine Schnittstelle zur Verfügung gestellt, die das Mess-Kontextmodell repräsentiert und auf die der Messcode zugreift. Die Form dieser Schnittstelle wird in Kapitel 3.3.1 auf Seite 21 beschrieben.

Es bleibt noch zu klären, wann ein Messsystem einem Mess-Kontextmodell unterstützt beziehungsweise wie das erreicht werden kann. Dafür muss die oben offen gelassene Wahl wieder herangezogen und Kontext- und Komponentenmodellgetrennt untersucht werden:

- Falls das Mess-Kontextmodell ein Komponentenmodell ist, dann kann es genau dann unterstützt werden, wenn es durch den Container verwendet beziehungsweise implementiert wird und das Messsystem Teil des Containers ist. Alle Kontextmodelle mit Transformationen auf dieses Komponentenmodell können indirekt durch eine der Transformation der Charakteristiken unterstützt werden.

- Falls es sich bei dem Modell um ein Kontextmodell handelt, so kann dieses auch unterstützt werden, wenn das Messsystem eine Abbildung zwischen dem Container und diesem Kontextmodell beinhaltet. Das Kontextmodell spielt dann eine ähnliche Rolle wie das Komponentenmodell im anderen Fall.

Bei beiden Fällen muss das Messsystem auf Interna des Containers zugreifen. Es liegt also nahe, dass die Entwickler des Containers auch für die Entwicklung des Messsystem verantwortlich sind, oder wenigstens die Abbildungen zwischen Modellen und dem konkreten Container bereitstellen.

Kontext- und Komponentenmodelle müssen eindeutig identifizierbar sein, damit überprüft werden kann, ob das vom Messsystem unterstützte Modell auch dem tatsächlich angeforderten entspricht. Bei Komponentenmodellen gestaltet sich das wegen der kleinen Anzahl an Modell einfacher als bei Kontextmodellen.

Ein Container wird immer ein Komponentenmodell implementieren, insofern sollte das Messsystem dieses auch unterstützten. Ob allerdings auch einzelne Kontextmodelle angeboten werden sollten, ist eher eine Kosten-Nutzen-Frage.

Die Transformation von einem neuen Kontextmodell auf ein Komponentenmodell sollten in der Regel möglich sein. Das ist nur dann nicht der Fall, wenn im Container mehr gemessen werden könnte, als durch das Komponentenmodell zugänglich gemacht wird, was aber eher unwahrscheinlich ist. Die Transformation muss vom Measurement Designer erstellt werden, sollte diesem aber keinen zu großen Aufwand bereiten.

Wenn das Messsystem ein neues Kontextmodell unterstützen soll, so müssen die Entwickler des Containers eine Abbildung vom Container auf das Modell anbieten. Ob das einen größeren Aufwand als eine Transformation zwischen Modellen darstellt, hängt vom Umfang und der Komplexität des Modells ab. Modelle mit höherem Abstraktionsgrad und weniger Details werden sich in der Regel leichter auf Container abbilden lassen.

Weiterhin ist der Ressourcenbedarf für das Verwalten eines Komponentenmodells relevant. Wenn zum Beispiel die tatsächlich eingesetzten Kontextmodelle sehr klein sind und nur einen kleinen Teil des Komponentenmodells abdecken, so ist es während der Laufzeit effizienter, direkt das kleine Kontextmodell zu unterstützen, als ein großes Komponentenmodell mit vielen nicht genutzten Teilen zu verwalten.

Zusammengefasst entstehen also im Entwicklungsprozess über diverse Transformationen und unter Mithilfe von Measurement Designer und Application Designer Charakteristiken, die sich auf Mess-Kontextmodelle beziehen und aus denen Messcode generiert wird. Das Messsystem verwaltet zur Laufzeit dann einen Mess-Kontext, auf den der Messcode zugreifen kann.

3.2 Messverfahren

Bevor Messverfahren festgelegt werden können, muss zuerst das zugrundeliegende Modell für das System, auf das sich die Messungen beziehen, spezifiziert werden. Das Mess-Kontextmodell dient dafür als Modell des Zustands des Systems. Es enthält schon Konsistenzkriterien aus statischer Sicht, zum Beispiel durch die Beschreibung als ein UML-Modell. Im Folgenden wird deshalb die Dynamik des Systems betrachtet.

Innerhalb des Messsystems gibt es zwei Akteure, die auf das System zugreifen:

- Messfunktionen lesen Zustände, verändern sie aber nicht (OCL-Ausdrücke sind reine Abfragen).
- Die Laufzeitumgebung oder das System, das durch das Mess-Kontextmodell repräsentiert wird, verändern sich. Diese Änderungen sollen

sich natürlich auch im Abbild, dem Mess-Kontext, wiederfinden. So werden zum Beispiel tatsächliche Aufrufe einer Operation[9] in eine Zustandsänderung übersetzt, die eine neue `OperationCall`-Instanz erstellt, diese der `invocations`-Assoziation hinzufügt und `lastInvocation` neu setzt.

Zustandsänderungen führen somit in der Regel mehrere Zugriffe und Änderungen durch, um den Mess-Kontext von einem Zustand in einen anderen zu überführen, die statischen Konsistenzkriterien erfordern aber fertige Zustände. Deshalb sind Zustandsänderungen als Transaktionen zu betrachten, die das System von einem konsistenten Zustand in einer anderen überführen, aber aus mehreren Einzeloperationen bestehen. Für Messfunktionen gilt dies auch, denn sie erwarten konsistente Zustände und sind per Definition atomar[10].

Im Folgenden werden Operationsfolgen, die durch das Ausführen einer Messfunktionen entstehen, als *lesende Transaktionen* bezeichnet. Zustandsänderungen werden *schreibende Transaktionen genannt*, sie können natürlich auch lesend zugreifen.

Das Messsystem verwaltet den Mess-Kontext und ist damit auch für das Ausführen der Transaktionen verantwortlich. Es erhält die Transaktionen von den Auftraggebern von Messungen und von dem System, dass durch den Mess-Kontext repräsentiert wird. Diese Transaktionen sind partiell geordnet, zum Beispiel wenn eine schreibende Transaktion auf eine andere folgt und damit die zweite Transaktion den Zustand der ersten verändern muss. Lesende Transaktionen sind auch Teil dieser Ordnung, in welcher Art wird aber erst durch die konkreten Messverfahren geregelt und weiter unten behandelt.

Um die Transaktionen ausführen zu können, muss das Messsystem die Operationen der Transaktionen so anordnen, dass die ursprüngliche Ordnung ihrer Transaktionen gewahrt bleibt. Eine nicht überlappende Anordnung, bei der jeweils nur eine Transaktion aktiv ist und deren Operationen ohne Unterbrechung durch andere hintereinander ausgeführt werden, ist immer möglich. Die Operationen können dann nach Planung der Operationsanordnung durch das Messsystem durchgeführt werden. Kapitel 3.3.3 auf Seite 27 betrachtet, wie solche Anordnungen noch aussehen können, also zum Beispiel mit mehr Nebenläufigkeit. Zusätzlich werden dort die Anforderungen der Transaktionen näher untersucht, also bezüglich Isolationsgrad und ähnlichem.

Welche schreibenden Transaktionen es geben kann beziehungsweise gibt, und welche Operationen diese haben, hängt vom Mess-Kontextmodell und

[9] vergleiche das Beispiel-Kontextmodell in Abbildung 1

[10] auch als „instant evaluation“ bezeichnet, d.h. es wird keine Zeit zum Ausführen benötigt

dessen Konsistenzkriterien ab. Lesende Transaktionen entstehen durch `values`-Klauseln, sind also auch nicht im voraus bekannt.

Voraussetzungen für das Messen sind also ein Mess-Kontext, in dem gemessen wird, und ein Verfahren zur korrekten, das heißt im Kontext des Modells sinnvollen Anordnung von Operationen von schreibenden und lesenden Transaktionen. In Kapitel 2.2 wurden zwei Anwendungsfälle gefunden, aus denen sich unterschiedliche Messverfahren ergeben müssen, stichprobenartiges und änderungsgesteuertes Messen.

3.2.1 Stichprobenartiges Messen

Beim stichprobenartigen Messen soll zu einem bestimmten Zeitpunkt eine einzelne Messung durchgeführt werden. Dafür muss eine lesende Transaktion durch das Messsystem eingeplant werden, die die Operationen der Messfunktion ausführt. Es wird angenommen, dass die Anordnung der Operationen so möglich ist, dass Transaktionen jeweils konsistente Zustände sehen. Was aber hier hinzukommt, ist die Beziehung zwischen dem gewünschten Zeitpunkt der Messung und der Position der Messung bezüglich der anderen Transaktionen. Der Messzeitpunkt fließt somit als weitere Bedingung in die Planung der Anordnung ein und muss vom Messsystem beachtet werden.

Folgend wird deshalb die Beziehung zwischen Zeit und Transaktionen bzw. Operationen genauer untersucht, damit sich daraus konkrete Bedingungen für die Anordnung schlussfolgern lassen.

Es muss auf jeden Fall eine Abbildung der Zeit des Auftraggebers der Messung auf die Zeit des Messsystems geben. Das muss aber vom Auftraggeber erreicht werden, und wird deshalb hier nicht behandelt. Der somit vorliegende Messzeitpunkt im Rahmen des Messsystems muss sich auch auf das Mess-Kontextmodell übertragen lassen. Deshalb sollte es auch eine Definition von Zeit im Rahmen des Mess-Kontextmodells geben.

Da Änderungen oder Berechnungen der Zeit auf jeden Fall einzelne Abläufe im Mess-Kontext oder letztendlich in der Laufzeitumgebung erfordern, ist die Domäne der Zeit diskret[11]. Ist die Zeit beispielsweise durch einen Wert modelliert, der Teil des Mess-Kontexts ist, so erfordert jede Änderung der Zeit eine schreibende Transaktion, die eben diesen Wert verändert.

Damit entspricht ein bestimmter Messzeitpunkt einer Zeitspanne, die begrenzt wird durch zwei die Zeit verändernde Transaktionen. Soll nun eine lesende Transaktion in dieser Zeitspanne stattfinden, so ist sie zwischen die zwei schreibenden Transaktionen einzuordnen, und wird damit Teil der partiellen Ordnung der Transaktionen untereinander.

Schreibende Transaktionen müssen auch in Bezug auf Zeitänderungen angeordnet werden, falls sie in einer bestimmten Zeitspanne erfolgen sollen.

[11] Vergleiche auch das Zeitmodell in [Aag01]

Sie werden damit gegenüber den lesenden Transaktionen geordnet, die sich auch auf diese Zeitspanne beziehen. Ob und wie schreibende Transaktionen Bezug zur Zeit haben, hängt ganz von den verursachenden Ereignissen ab. Wenn zum Beispiel ein Ereignis nur irgendwann innerhalb von zehn Sekunden bearbeitet wird, dann müssen lesende Transaktionen das Ereignis beziehungsweise dessen Transaktion auch erst beachten, wenn sie nach den zehn Sekunden messen wollen. Die schreibende Transaktion könnte schon nach einer Sekunde stattfinden und die lesende Transaktion nach zwei Sekunden messen wollen, und trotzdem würde sie das Ereignis noch nicht lesen müssen. Anders sähe es wiederum dann aus, wenn das Ereignis genau zur zehnten Sekunde stattfinden müsste, dann dürfte die Transaktion das Ereignis noch gar nicht lesen.

Transaktion bzw. deren Operationen benötigen Ressourcen des Messsystem, und die sind beschränkt. Während also eine korrekte Anordnung der Operationen immer möglich sein sollte, so kann es doch sein, dass diese nicht vom Messsystemausgeführt werden kann.

Wenn zum Beispiel das Mess-Kontextmodell den Container repräsentiert, und das Zeitverständnis des Containers an die Menge verstrichener Prozessorzeit gebunden ist, dann richtet sich die Zeit im Mess-Kontextmodellauch an dieser aus. Somit steht dem Messsystem in einer Zeitspanne nur eine begrenzte Menge von Prozessorzeit zur Verfügung, und damit kann in dieser Zeitspanne auch nur eine begrenzte Menge von Operationen ausgeführt werden.

Es muss also überprüft werden, ob genügend Ressourcen für das Ausführen der eigentlich eingeplanten Operationen vorhanden sind[12]. Falls es keine Möglichkeit gibt, die Operationen auszuführen, müssen die entsprechenden Transaktionen vom Messsystem abgelehnt werden. Eine Messung ist zu diesem Zeitpunkt dann auch nicht möglich.

Das stichprobenartige Messen ist also durch die zusätzliche Beziehung zwischen der lesenden Transaktion der Messfunktion und den schreibenden Transaktionen, die Zeitveränderungen darstellen, charakterisiert. Weiterhin muss das Messsystem beachten, dass in einer Zeitspanne nur begrenzte Ressourcen zur Verfügung stehen.

3.2.2 Änderungsgesteuertes Messen

Das Ermitteln jeder Veränderung des Ergebnisses der Messfunktion ist Ziel beim änderungsgesteuerten Messen. Das Ergebnis der Messfunktion hängt

[12]Es wird hier und im Folgenden vorausgesetzt, dass entweder die obere Grenze für die zum Ausführen einer Transaktion nötigen Ressourcen automatisch ermittelt werden kann, oder jemand im Rahmen des Entwicklungsprozesses diese bereitstellt und es für das Verletzen dieser dann auch sinnvolle Reaktionen gibt. Im Kapitel **??** auf Seite ?? wird der Ressourcenbedarf von Messfunktionen näher betrachtet.

von den Daten ihres Wertebereichs[13] ab, somit kann sich das Ergebnis genau dann ändern, wenn sich die von der Funktion verwendeten Daten im Mess-Kontext ändern.

Diese Änderungen durch schreibende Transaktionen geschehen nur zu bestimmten Zeitpunkten und sind durch Ereignisse in dem System, das durch das Modell repräsentiert wird, begründet. Aussagen über diese Zeitpunkte machen eine Ermittlung der nötigen Messzeitpunkte möglich und sind somit Voraussetzung für das änderungsgesteuerte Messen.

Die Informationen über die Zeitpunkte der Änderungen stehen dem Messsystem entweder direkt zur Verfügung, oder lassen sich aus der Verwaltung des Mess-Kontexts ableiten.

Der erste Fall tritt beispielsweise ein, wenn das Messsystem Teil des Containers ist und über die Vorgänge in diesem Bescheid weiß. So kann ein Datenpaket zu einem nicht kontrollierbaren Zeitpunkt auf einem Socket eintreffen, aber der Programmcode, der Sockets verwaltet, ist nur zu bestimmten Zeitscheiben aktiv. Daraus und aus dem Minimalaufwand für die Bearbeitung eines Datenpaketes lässt sich die Frequenz von möglichen Ereignissen für neue Datenpakete ableiten. Es muss also die Implementierung des Containers auf solche Abhängigkeiten untersucht werden. Oft wird es aber auch günstiger sein, solche Festlegungen extra für eine Container-Implementierung oder ein Modell zu treffen, um die Frequenz zu verringern.

Falls dem Messsystem solche Daten nicht bekannt sind, so weiß es aber mittels einer analogen Schlussfolgerung, mit welcher Frequenz es schreibende Transaktionen aufnehmen kann. Das Ergebnis ist allerdings wesentlich schlechter, da wahrscheinlich die Art der Transaktion nicht im voraus bekannt ist, und somit potentiell alle Messungen neu auszuführen sind.

Für jede Neuberechnung einer Messfunktion wird dann wie beim stichprobenartigen Messen eine lesende Transaktion eingeplant. Diese muss nach der auslösenden schreibenden Transaktion liegen und vor anderen schreibenden, die den zu lesenden Zustand ändern könnten. Lesende Transaktionen behindern sich nicht gegenseitig.

Bezüglich der Beziehung einer Zustandsänderung zur Zeit gibt es drei Punkte, die zu beachten sind:

- Falls Messfunktionen auf die Zeit zugreifen, dann sind die entsprechenden lesenden Transaktionen in die gleiche Zeitspanne einzuordnen wie die der Änderung zugeordnete schreibende Transaktion.

- Der Auftraggeber kann an der Zeit interessiert sein, zu der eine Änderung stattfand. Die Zeit der Änderung muss also vermerkt und mit

[13] Der Wertebereich der Messfunktion beschränkt sich dabei nicht nur auf die Parameter, sondern auf alle eventuell referenzierten Teile des Kontextmodells.

übergeben werden, die lesende Transaktion muss aber nicht in der gleichen Zeitspanne ausgeführt werden.

- Wie bei den Anwendungsfällen in Kapitel 2.2 beschrieben, existieren nicht-funktionale Anforderungen an die Übermittlung von Messergebnissen, für die der zeitliche Abstand zwischen Änderung und Messergebnis relevant sein kann.

Am Anfang einer änderungsgesteuerten Messung muss noch eine stichprobenartige Messung durchgeführt werden, da die änderungsgesteuerte Messung nur Änderungen erfasst, und damit ansonsten der Anfangswert nicht ermittelt werden könnte.

Wichtig beim änderungsgesteuerten Messen sind also die Änderungsraten im Mess-Kontextmodell, da abhängig von diesen Messwerte neu berechnet werden müssen. Die dem Messsystem zur Verfügung stehenden Ressourcen sind auch hier relevant, wenn zum Beispiel Änderungsraten so hoch sind, dass in den kurzen Perioden nicht alle Messungen ausgeführt werden können.

3.3 Mess-Kontextmodelle

In diesem Kapitel wird untersucht, wie Mess-Kontexte verwaltet werden können.

3.3.1 OCL-`values`-Klausel und Mess-Kontextmodell

Im Kapitel 3.1 wurden die Verantwortlichkeiten im Entwicklungsprozess betrachtet. Dabei stellte sich heraus, dass das Messsystem einen Mess-Kontext verwalten und eine Schnittstelle zu diesem anbieten muss. In Abbildung 5 auf Seite 15 ist die Schnittstelle als Code-Schnittstelle bezeichnet. Der Name zeigt an, dass die Syntax der Schnittstelle zum Messcode passen muss, der auf die Schnittstelle zugreift. Der Inhalt der Schnittstelle wird dagegen durch das Mess-Kontextmodell bestimmt. Ziel dieses Kapitels ist es nun, Festlegungen für die Syntax, also die Form dieser Code-Schnittstelle zu treffen.

Dabei ist zu beachten, dass das Messsystem für die Schnittstelle und die Code-Generierung aus den transformierten `values`-Klauseln verantwortlich ist. Somit ist der gesamte Bereich unter Kontrolle des Messsystems, und die Art und Weise, wie Code generiert wird, kann auch an die Schnittstelle angepasst werden. Insbesondere muss sich der Mess-Kontext und die Abbildung des Modells auf den Container auch in der Implementierungsebene befinden. Da aber die Abbildung durch Container und Modell bestimmt ist, sollte sie auch auf der Container-Seite der Schnittstelle liegen (auf der anderen Seite befindet sich der Messcode). Die Schnittstelle würde sich dann nur

an den Elementen des Modells orientieren und der Code-Generator könnte wiederverwendet werden.

Die Schnittstelle muss inhaltlich gesehen den Zugriff von OCL-Ausdrücken auf das UML-Modell enthalten, das die statische Sicht auf das Mess-Kontextmodell darstellt. Zum einen sind damit Zugriffe auf Instanzen aus dem Modell nötig. Weiterhin muss zum Beispiel für den `oclIsKindOf()` Operator, der zurückgibt, ob ein Objekt Zuweisungs-kompatibel zu einem bestimmten Typ ist, die Typ-Hierarchie und der Typ jedes Objektes bekannt sein. Wenn das schon während der Code-Generierung festgestellt werden soll, und nicht erst während der Ausführung des OCL-Ausdrucks, dann dürfen Objekte des Modells nur mit ihrer eigentlichen Klasse und nicht unter einer Oberklasse laufend an den OCL-Ausdruck übergeben werden. Damit kennt der OCL-Ausdruck den Typ jedes nicht selbst konstruierten Objektes. Innerhalb des OCL-Ausdrucks beziehungsweise durch den generierten Code können dann Typinformationen mitgeführt werden, mit denen Operationen wie `oclIsKindOf()` oder `oclAsType()` arbeiten. Andererseits sind solche Operationen eher für das Testen von Programmen als für das reine Ausrechnen von Werten nützlich, und müssen somit wahrscheinlich nicht unterstützt werden. Falls diese Operationen nicht angeboten werden, oder der Typ jedes Objekts aus dem Modell bekannt ist, müssen Objekte ihren eigenen Typ nicht kennen.

Zum einen muss also das Konzept von Objekten beziehungsweise Instanzen von Klassen umgesetzt werden, und das geschieht natürlich am Besten eben mit Objekten. Es wird davon ausgegangen, dass die bei der Code-Generierung verwendete Sprache dies schon unterstützt, also Objekt-orientiert ist. Alternativ kann das Konzept von Objekten aber auch relativ einfach in nicht Objekt-orientierten Sprachen umgesetzt werden, zum Beispiel durch das Ersetzen von Objekten durch Tupel, die um ein Feld mit einem eindeutigen Identitätsattribut erweitert sind.

Da die Schnittstelle das Modell repräsentieren soll, muss die Schnittstelle die Klassen-Hierarchie des Modells enthalten. Das UML-Modell kann somit bezüglich der Klassen direkt als Schnittstelle verwendet werden.

OCL unterscheidet zwischen drei Elementen („Properties“) von Objekten, auf die von OCL-Ausdrücken aus zugegriffen werden kann: Attribute, Assoziationen und Operationen. Auf die ersten beiden wird nur lesend zugegriffen. Die Werte dieser sollten aber nicht direkt als Attribute in den Klassen der Schnittstelle zur Verfügung stehen, sondern über Getter-Methoden der Klassen ausgelesen werden. Dadurch hat das Messsystem bei der Verwaltung einer Modell-Instanz freie Wahl bezüglich der Art und Weise der Speicherung der Objektdaten.

Am einfachsten geschieht die Zuordnung von Attributen und Assoziationen zu Methoden der Schnittstellen-Klassen durch Benennungsregeln. Die Regeln sind relativ beliebig, es dürfen nur keine Namens-Kollisionen auf-

treten. Zum Beispiel könnte einem Attribut `a` die Methode `getAttr_a()` zugeordnet werden, der Ergebnistyp der Methode wäre der Typ des Attributs. Alternativ könnten schon im Modell nur Methoden vorhanden und Attribute und Assoziationen nicht mehr direkt zugänglich sein, dass macht das Schreiben von `values`-Klauseln aber unbequemer und reduziert bei Assoziationen die Lesbarkeit des Modells.

Um Operationen in OCL-Ausdrücken verwenden zu können, dürfen diese keine Seiteneffekte haben, also das Modell nicht verändern. Deshalb muss dies für alle Methoden des Modells und damit auch der Schnittstelle gelten. Am günstigsten ist es sicherlich, im Rahmen des Mess-Kontextmodells eine Variante des UML-Modells bereitzustellen, die nur die Seiteneffekt-freien Operationen enthält. Danach können Operationen direkt vom Modell in die Schnittstelle übernommen werden, da sie schon genügend Freiraum für Indirektion oder ähnliches bieten. Namens-Kollisionen mit den Methoden für Attribute und Assoziationen müssen vermieden werden, deshalb sollte vor die eigentlichen Methodennamen noch ein Präfix gehängt werden, beispielsweise `operation_`.

Neben den im Modell definierten Typen müssen auch noch primitive Typen zur Verfügung stehen. Welche das sein sollten wird aber erst in Kapitel 3.4.2 auf Seite 33 untersucht, da dieses Thema eng mit der Messfunktion verbunden ist.

Die Code-Schnittstelle ist somit im wesentlichen eine Übertragung des Klassendiagramms aus dem Mess-Kontextmodell in die bei der Code-Generierung verwendete Sprache. Methoden werden als Indirektion für den Zugriff auf Attribute und Operationen eingeführt.

3.3.2 Abbildung zwischen Mess-Kontextmodell und Container

Im Kapitel 3.3.1 wurde die Schnittstelle zum Mess-Kontext beschrieben. Hier soll es nun darum gehen, wie Zustände im Container auf Zustände im Mess-Kontext abgebildet werden können und dann über diese Schnittstelle zugänglich sind. Dabei geht es nur um die statische Sicht, auf die Umsetzung im dynamischen Sinne wird näher in Kapitel 3.3.3 eingegangen.

Bisher wurde angenommen, dass der Mess-Kontext, also eine Instanz des Mess-Kontextmodells, immer existiert und die Instanz genau in der durch das Mess-Kontextmodell vorgegebenen Form gespeichert wird. Da der Mess-Kontext aber durch die Abbildung zwischen Mess-Kontextmodell und Container und die Zustände im Container bestimmt wird, muss nicht unbedingt ein Mess-Kontext mit Datenhaltung aufgebaut werden, sondern Zugriffe auf die Code-Schnittstelle können direkt auf Zustände im Container abgebildet werden. Für die Gestaltung der Abbildung gibt es also folgende Möglichkeiten:

1. Verwalten eines Mess-Kontexts
2. Direktes Abbilden auf den Container, keine Mess-Kontext
3. Erstellen von Teilen einer Mess-Kontexts bei Bedarf, also Zugriffen auf die Code-Schnittstelle

Bei der ersten Variante wird eine *Mess-Kontext* als Daten-Schicht zwischen `values`-Klausel und Container aufgebaut. Der Mess-Kontext implementiert die Code-Schnittstelle und ist um Funktionen zum Verändern von Objekten innerhalb des Mess-Kontexts erweitert. Jede dieser Änderungen hat als Ursache ein Ereignis oder eine Zustandsänderung im Container, Grundlage dafür ist die Inverse der dem Messsystem bekannten Spezifikation der Abbildung von Mess-Kontextmodell auf den Container. Der Mess-Kontext ist vollständig, das heißt das sich alle durch das Mess-Kontextmodell abgedeckten Zustände und Ereignisse im Container auch im Mess-Kontext wiederfinden.

Eine Untervariante ist das bedarfsgesteuerte Aufbauen des Mess-Kontexts auf Basis des Mess-Kontextmodells. Dabei werden nur die Teile des Modells verwaltet, die von Messfunktionen genutzt werden können. Welche Teile das sind, wird genau wie beim änderungsgesteuerten Messen ermittelt. Allerdings ist die Menge der möglichen Messfunktionen nicht von vornherein bekannt, unerwartete Stichproben-Messungen könnten ein Aufbauen von großen Teilen der Modell-Instanz erfordern. Die dafür nötigen Ressourcen müssten dann vor der Zusage zum Messen aufgebracht werden. Es wird wohl auch nur bei sehr großen Mess-Kontextmodell vorkommen, das wesentliche Teile des Mess-Kontexts nicht verwendet werden.

Wenn *kein Mess-Kontext* aufgebaut werden soll, muss bei jedem Zugriff auf die Code-Schnittstelle der Rückgabewert direkt aus dem Zustand des Containers gebildet werden, dem Messsystem ist die dafür genutzte Abbildung vom Mess-Kontextmodell auf den Container bekannt. Dieses Funktionsorientierte Vorgehen passt aber schlecht zu der Objekt-orientierten Code-Schnittstelle. Zum einen sind sowohl Definitions- als auch Wertebereich (Methoden und deren Rückgabewerte) in Objekt-orientierter Form gehalten. Es müssten also auf jeden Fall Objekte für die Rückgabewerte erzeugt werden, zum Beispiel als Adapter auf Elemente im Container.

Das andere Problem ist die benötigte Objekt-Identität, die zum Beispiel beim Vergleich von Objekten verwendet wird. Die OCL-Ausdrücke in den `values`-Klauseln beziehungsweise die Messfunktionen müssen zum Beispiel beim wiederholten Abfragen von Assoziationen immer identische Ergebnismengen erhalten, das heißt schon einmal zurückgegebene Ergebnisse müssen wiederverwendet werden. Die erstellten Objekte müssen mindestens bis zum Ende des Ausführens der Messfunktion gültig bleiben. Bei Objekten, die Parameter von Messfunktionen sind und damit sozusagen deren Einstiegspunkt

bilden, ist eine den gesamten Messzeitraum umfassende Lebensdauer sinnvoller.

Zu beachten ist, dass dieses Problem unabhängig von der Darstellung der Objekt-Identität auftritt. Wenn zum Beispiel die Objekt-Identität durch einen nur einmalig vergebenen Wert repräsentiert wird, so muss beim wiederholten Abfragen der Assoziation rekonstruiert werden, welcher Wert beim ersten Mal vergeben wurde. Für das eigentliche Ergebnis trifft dies ebenso zu.

Bei der dritten Variante, dem *Erstellen von Teilen des Mess-Kontexts je nach Bedarf*, werden aber Objekte verwendet, womit die obigen Probleme nicht auftreten. Dabei existiert eine Implementierung der Schnittstelle, die Teile des Mess-Kontexts nur berechnet, wenn auf diese zugegriffen wird. Gespeichert werden diese mindestens solange wie die eben beschriebene nötige Gültigkeitsdauer, entweder dürfen währenddessen entsprechende Teile des Containers nicht verändert werden oder die Konsistenzkriterien des Mess-Kontextmodells müssen die Verwendung von alten oder teilweise veränderten Daten erlauben. Das Berechnen der Zustände geschieht genau wie bei der direkten Abbildung.

Der Unterschied zur bedarfsgesteuerten ersten Variante liegt darin, das bei dieser die Partitionierung zum Beispiel über Klassen, das heißt verwendeten Teilen des Modells, vorgenommen wird. Hier aber wird nach Objekten, also Teilen der Instanz des Modells, unterteilt. Bei der ersten Variante sind die Klassen, auf die zugegriffen werden könnte, bekannt. Für Instanzen trifft das aber nur zu, wenn die Messfunktionen entsprechend genau analysiert werden können, und dann tritt die Frage auf, ob die Verwaltung der Menge der möglicherweise verwendeten Instanzen so aufwändig ist, dass sich die genaue Analyse nicht mehr lohnt. Es steht somit eine relativ grobe, dafür aber vorher bekannte und garantierte Partitionierung einer feineren, aber nicht planbaren gegenüber.

Relevant erscheinen also nur die Varianten eins und drei. Zusätzlich zu den schon genannten Punkten unterscheiden sie sich wesentlich in der Abhängigkeit zwischen dem dynamischen Verhalten des Containers und den Messaktivitäten. Bei der ersten Variante bezieht sich die Indirektion nicht nur auf die Darstellung und Speicherung der Daten, sondern auch auf den Aufbau und die Verwaltung dieser. Der Mess-Kontext wird dabei von Vorgängen im Container weitestgehend entkoppelt, indem Änderungen im Container zuerst nur gesammelt und dann im Mess-Kontext isoliert als schreibende Transaktionen eingeplant werden. Hingegen kann es bei den anderen Varianten dazu kommen, das Vorgänge im Container von Messungen abhängig sind, zum Beispiel wenn Änderungsoperationen warten müssen, weil die Messung eine konstante Sicht auf die Daten benötigt[14].

[14]Das folgt aus der Instant-Evaluation-Semantik von OCL, vergleiche auch die Beschrei-

Allgemein ausgedrückt sind die unterschiedlichen Konsistenzkriterien der Modelle wesentlich einfacher durchzusetzen, wenn Änderungen in einer Modell-Instanz vorbereitet und abgeschlossen an die nächste weitergegeben werden, als wenn versucht wird, zu jedem möglichen Zeitpunkt einen momentanen Zustand in einen korrekten Zustand der anderen Modell-Instanz zu übersetzen. Die eventuell entstehende zusätzliche Verzögerung zwischen der ursächlichen und der daraus resultierenden Zustandsänderung beeinflusst natürlich die nicht-funktionalen Eigenschaften der Messwertübermittlung (siehe Kapitel 2.2).

Der Ressourcenbedarf bei den verschiedenen Varianten, ohne Betrachtung des Ressourcenbedarfs für die eigentlichen Messungen, hängt ab vom Verhältnis zwischen der Anzahl der Änderungen und den ausgeführten Messfunktion. Bei der ersten Variante sind Ressourcen proportional zur Menge der Änderungen zu reservieren, da unterschiedliche Messungen die gleichen Daten verwenden. Bei der dritten Variante ist es proportional zur Anzahl der Messungen, da für jede wahrscheinlich neue Daten aufgebaut werden müssen. Durch den gegebenenfalls hohen Aufwand beim Aufbau von Teilen der Modellinstanz sind bei der dritten Variante allerdings wahrscheinlich mehr Ressourcen zu reservieren, als tatsächlich gebraucht werden, falls die Menge der verwendeten Instanzen nicht perfekt vorhergesagt werden kann. Das obige sind aber nur Schätzungen, die konkreten Werte hängen ganz von den bei der Implementierung genutzten Mechanismen ab, zum Beispiel könnten sich Messungen bei der dritten Variante aufgebaute Daten teilen.

Die oben genannten Gründe sprechen, außer bei einer hohen Übereinstimmung zwischen Mess-Kontext und Container, für die erste Variante. Vor allem die Entkoppelung der Mess-Vorgänge im Mess-Kontext von dem dynamischen Verhalten des Containers entlasten diesen deutlich.

Bei dieser Lösung muss dann noch entschieden werden, wie viele Instanzen des Kontextmodells existieren sollen. Wird nur eine Instanz von allen Komponenten verwendet, so muss sichergestellt werden, dass Daten einer Komponente vor dem Zugriff fremder Komponenten geschützt werden. Nicht zugängliche Daten sollten zum Beispiel beim Navigieren über Assoziationen einfach aus den Ergebnismengen entfernt werden. Gibt es dagegen eine Instanz pro Komponente oder Gruppe von zusammengehörenden Komponenten, braucht diese auch nur die für die Komponente relevanten und zugänglichen Daten zu enthalten. Weiterhin können dann Kosten, zum Beispiel Speicherplatzbedarf, einfacher den Verursachern zugeordnet werden und sind direkt an deren Lebensdauer gebunden. Der Ressourcenverbrauch sollte in der Regel nicht steigen, da viele Kontextmodelle schon nach Komponenten unterteilen werden und die Redundanz damit gering ist.

---

bung der lesenden Transaktionen in Kapitel 3.2.

3.3.3 Anordnung von Zugriffen auf den Mess-Kontext

Bei der Beschreibung des Messverfahrens wurde schon gezeigt, dass Änderungen am Mess-Kontext und durchgeführte Messungen Transaktionen sind. Der einfachste Weg zur Serialisierbarkeit ist die Anordnung der Transaktionen als Sequenz, ohne Überlappungen. Im folgenden wird nun betrachtet, mit welchen Mitteln die Nebenläufigkeit erhöht werden kann.

Die ACID-Eigenschaften, die normalerweise mit Transaktionen verbunden werden, haben im Kontext des Messverfahrens folgende Bedeutung:

Atomicity ist natürlich relevant. Es darf aber nicht passieren, dass eine Transaktion nicht ausgeführt werden kann. Bei Änderungen des Mess-Kontexts würde das zu einem falschen Zustand führen. Bei ausgeführten Messfunktionen wäre ein nicht erfolgreiches Beenden des Messens möglich, ob das aber sinnvoll ist, hängt von der Verwendung des Messwertes ab. Viele Messungen werden nicht optional sein, zum Beispiel beim Überwachen von Zusagen einer Komponente. Wäre diese Messung nicht möglich, sollte wegen der Unsicherheit die Komponente beendet werden, obwohl es keinen Nachweis über das Verletzten der Zusage gibt. Entsprechende Regeln für das Reagieren auf fehlgeschlagene Messungen existieren außerdem noch nicht, einmal angenommene Messung sollten somit garantiert ausgeführt werden.

Consistency Änderungen sollten den Mess-Kontext immer von einem konsistenten Zustand in einen anderen überführen.

Isolation Bei Messungen kann es plausibel sein, dass diese beispielsweise alte Zustände oder Zwischenergebnisse lesen. Das sollte dann aber über weniger strikte Konsistenzkriterien im Mess-Kontextmodell realisiert werden. Damit würden auch Messungen nur auf konsistente Zustände zugreifen, das Verständnis von Konsistenz wär aber dem Verwendungszweck des Modells angepasst. Änderungen müssen wiederum isoliert erfolgen.

Durability Einmal durchgeführte Änderungen dürfen nicht verloren gehen. Der Mess-Kontext muss aber nur die gleiche Lebensdauer wie der Container haben, weiterreichende Persistenz ist also nicht nötig. Falls Zustände im Mess-Kontext eine kürzere Lebensdauer haben, zum Beispiel weil sie für jede Messung neu berechnet werden, so ist die Lebensdauer des Zustands relevant.

Wie schon bei der Atomarität gesagt, sollten Transaktionen garantiert erfolgreich ausgeführt werden. Dafür muss die Art ihrer Ausführung, also die Anordnung ihrer Operationen im Vergleich zu denen anderer Transaktionen vorher geplant werden.

Ist nichts über die Operationen einer Transaktion bekannt, also welcher Funktion diese ausführen und auf welche Daten sie zugreifen, so muss der schlechteste Fall angenommen werden (die Transaktion verändert oder liest den gesamten Mess-Kontext). Das hat zur Folge, dass die Transaktion während ihrer Ausführung als einzige auf den Mess-Kontext zugreifen darf und entspricht der oben genannten Anordnung als Sequenz. Ziel muss also sein, so viel wie möglich von vornherein über eine Transaktion zu wissen, damit entsprechend ressourcenschonend geplant werden kann. Auf die dafür verwendeten Algorithmen wird hier nicht eingegangen, sie müssen aber zusätzlich noch aus dem änderungsgesteuerten Messen resultierende Abhängigkeiten (neue Messungen folgen direkt auf Veränderungen) und den Ressourcenbedarf zum Ausführen der Transaktionen berücksichtigen.

Die Semantik von Änderungs-Transaktionen ist den Entwickler des Containers bekannt, denn sie sind die Autoren. Da sie auch die Verantwortung für die Abbildung zwischen Mess-Kontextmodell und Container haben sollen (siehe Kapitel 3.1), steht dieses Wissen also auch bei der Planung der Transaktionen zur Verfügung.

Bei Messfunktionen und deren durch die OCL-`values`-Klauseln repräsentierten Transaktionen gestaltet sich das schwieriger. Deren Autoren sind die Measurement Designer, und es kann nicht so einfach im Rahmen des Entwicklungsprozesses Wissen über die Messfunktionen übertragen werden, weil Messfunktionen auch nach dem Entwickeln des Containers und von anderen Autoren erstellt werden. Deshalb kann nur versucht werden, aus den `values`-Klauseln bestimmte Eigenschaften zu schlussfolgern. Dies ist aber je nach Mächtigkeit der Spezifikations-Möglichkeiten bei `values`-Klauseln nicht immer möglich. Zumindest kann aber immer festgestellt werden, auf welche Teile des Mess-Kontextmodells die `values`-Klausel zugreifen kann, eine weitere Beschränkung auf bestimmte Instanzen ist aber nicht möglich. Dass Messfunktionen nur Lese-Operationen haben, macht es wiederum einfacher, denn diese können zueinander nebenläufig erfolgen.

Wenn nichts konkretes über eine Transaktion ausgesagt werden kann, so hilft es dennoch, bestimmte Fälle auszuschließen, so dass die Menge der verbleibenden Möglichkeiten und damit die Unsicherheit verringert wird. Zum einen geht dies über Festlegungen im Systemmodell, die die Entwickler an bestimmte Regeln für Transaktionen binden, zum Beispiel bestimmte Operationsfolgen nicht auszuführen. Im Fall des Containers und schreibenden Transaktionen ist das unbequem für die Entwickler, aber durchsetzbar, denn die Verantwortlichen tragen auch die Konsequenzen bei Verletzung dieser Regeln. Andererseits ist es bei Messfunktionen unpraktisch, weil es nicht automatisch überprüft werden kann. Wenn doch, wären diese Regeln gar nicht nötig, die Überprüfung würde das ansonsten nötige Vertrauen in eine Zusage überflüssig machen. Zusätzlich wird man diese Verletzung auch nur schwer feststellen können, denn man müsste zum Beispiel überprüfen,

ob ein falscher oder veralteter Zustand gemessen wurde. Selbst wenn sowas festgestellt werden kann, ist erfolgreiches Messen nicht mehr garantiert und die Regeln würden ihren Zweck nicht erreichen.

Die Transaktionen von Messfunktionen schränkt man besser mit Hilfe der Schnittstelle ein. Das Mess-Kontextmodell muss dazu so modifiziert werden, dass es nur noch bekannte oder vom Planungs-Algorithmus behandelbare Fälle ermöglicht. Ungünstige Operationen können entfernt und oft auftretende Operationsfolgen als einzelne Operationen angeboten werden. Beispielsweise könnte der Zugriff auf eine gesamte Kollektion verboten und durch eine Methode zum Ermitteln eines einzelnen Elements ersetzt werden, `values`-Klauseln würden dann nicht mehr über alle Elemente iterieren, sondern nur die eine Methode aufrufen. Die Semantik dieser vorgefertigten Sub-Transaktionen ist dann dem Planungs-Algorithmus bekannt und vereinfacht die Planung damit. Nachteil dieser Lösung ist, dass das Mess-Kontextmodell verändert werden muss und möglicherweise an Ausdrucksstärke verliert oder durch Anbieten vieler Sub-Transaktionen zu groß wird. Messcode kann trotzdem immer noch generiert werden, es wird nur Semantik aus den `values`-Klauseln in das Modell verlagert. Zusätzlich steigt der Aufwand für die Entwickler des Messsystems beziehungsweise des Containers, die dieses Modell anbieten wollen.

Ein weiterer Weg ist das Vereinfachen der Konsistenzkriterien des Kontextmodells. Dazu müssen Abhängigkeiten zwischen Operationen verringert werden, indem zum Beispiel alte oder unvollständige Werte gelesen werden dürfen. Das sollte sich aber ganz am Kontextmodell, dessen Einsatzzweck und den modellierten Aspekten orientieren, für Optimierungen eingeführte einfachere Kriterien dürfen nicht im Konflikt zum Sinn dieses Modells stehen.

3.4 Messfunktion

3.4.1 Umgebung von OCL-Ausdrücken in `values`-Klauseln

Bevor Code aus dem OCL-Ausdruck der `values`-Klausel generiert werden kann, muss die Umgebung modelliert werden, in die sie eingebettet ist.

OCL-Ausdrücke stehen immer im Kontext einer bestimmten Objekt-Instanz, die mit dem Schlüsselwort `self` angesprochen werden kann. Das UML-Modell, in dem sie operieren, ist in unserem Fall durch das Mess-Kontextmodell bestimmt. Die Code-Schnittstelle wurden schon in Kapitel 3.3.1 auf Seite 21 untersucht, und wird auch vom OCL-Ausdruck für den Zugriff auf den Mess-Kontext verwendet. Außerdem gibt es noch eine Liste mit Variablen, die aus `let`-Ausdrücken oder Parametern herrühren.

CQML$^+$ hat aus der CQML-Spezifikation das Konzept der Charakteristiken, Statements und Profiles übernommen. Damit existiert eine Grammatik für diese Elemente, die auch an den entsprechenden Stellen auf Teile der OCL-Grammatik verweist. Was aber in CQML nicht definiert wird, ist der Kontext, in dem die OCL-Ausdrücke, insbesondere in der `values`-Klausel, stehen. Der beschreibende Text lässt eher nur zwischen den Zeilen erkennen, was erwünscht ist. CQML$^+$ behebt einen wesentlichen Teil dieses Problems, in dem es Charakteristiken an ein Kontextmodell bindet und damit unter anderem das UML-Modell für die OCL-Ausdrücke bietet. Der fehlende Teil, also die Objekt-Instanz, auf die sich der Ausdruck bezieht, wird aber weiterhin offen gelassen. Deshalb wird im Folgenden versucht, gegebene und mögliche Bedingungen für den OCL-Ausdruck zu finden und diese in Form eines OCL-Kontexts auszudrücken. Weil es keine Abhängigkeit des Kontextmodells zu Charakteristiken geben sollte und diese damit nicht Teil des ursprünglichen Modells sind, bleibt nur noch eine entsprechend konstruierte Objekt-Instanz als Mittel dafür übrig.

Eine Charakteristik enthält folgende Teile, die für einen Zugriff von Seiten der `values`-Klausel in Frage kämen, und damit Teil des Kontexts sein müssten:

Parameter in CQML$^+$ sind im OCL-Ausdruck wie Parameter einer Funktion oder Methode zu behandeln.

Basis-Charakteristik: Eine Charakteristik kann eine Spezialisierung einer anderen sein. Von jener ist wiederum nur die `values`-Klausel interessant, Parameter sind lokal und die restlichen Eigenschaften werden übernommen. Ein Zugriff auf deren `values`-Klausel, beziehungsweise eine Spezialisierung dieser, ist in CQML$^+$ nicht vorgesehen. Die Ausdrucksstärke wird dadurch nicht verringert, aber das Warten der `values`-Klauseln wird durch die Redundanz aufwändiger.

Als Grundlage dienende Charakteristiken: Abgeleitete Charakteristiken sind abhängig von anderen, als Grundlage dienenden. In CQML$^+$

wird nur die Existenz einer Abhängigkeit angegeben. Es ist zwar die Rede davon, dass dies auch genauer mit Hilfe der `values`-Klausel spezifiziert werden könnte, eine Grammatik dafür oder Regeln für einen entsprechenden OCL-Kontext fehlen aber. Bei der Spezifikation der CQML$^+$-Semantik ist der Definitionsbereich der `values`-Klausel auch nur die Parameter-Menge.

Die eigene `values`-Klausel: Diese Möglichkeit wird in CQML$^+$ nicht erwähnt.

Statistische Aspekte: Der Zugriff auf diese ist nicht definiert. Es wird aber im Text genannt und tritt, zumindest bei Invarianten, in Beispielen auf. Auch statistische Aspekte sind laut Semantik-Spezifikation nicht Teil des Definitionsbereiches von `values`-Klauseln. Sie werden auch deshalb nicht unterstützt, die weiteren Gründe sind in Kapitel 3.6.1 auf Seite 39 zu finden.

Am einfachsten umzusetzen sind die *Parameter* und die durch die `values`-Klausel gegebene *Messfunktion*. Wie schon gesagt steht ein OCL-Ausdruck immer in Bezug zu einer Objekt-Instanz und es ist möglich, Hilfsmethoden zu definieren. Die Messfunktion ist Charakteristik-spezifisch, somit muss für jede Charakteristik eine neue Klasse eingeführt werden, innerhalb jener dann eine Methode mit der Signatur der Messfunktion definiert wird. In OCL-Syntax sähe das für die Charakteristik `delay` aus Listing 1 auf Seite 5 zum Beispiel so aus[15]:

```
context delay_Charakteristik::delay(c:OperationCall):Real
body: c.returnTime - c.callTime
```

Die Methode und die neue Klasse wurden entsprechend dem Name der Charakteristik benannt.

Der so entstehende Kontext enthält neben der benötigten Signatur der Messfunktion noch eine Instanz der Charakteristik-Klasse. Eigentlich wäre einen Klassen-Methode günstiger, aber für den Zugriff auf andere Charakteristiken und statistische Aspekte wird ein Ausgangspunkt benötigt. Außerdem ist die in CQML$^+$-Beispielen für so einen Zugriff vorkommende Syntax ähnlich zu einem Zugriff mit implizitem `self`.

Die entstehenden Zusatzinformationen sind allerdings harmlos. Es werden keine Konstruktoren für die neuen Klassen definiert und es gibt keine Assoziationen. Für `oclIsKindOf()`, `oclIsTypeOf()` und `oclAsType()` kommt, auch wenn der Name erraten wird, nur die neue Klasse in Frage. `oclInState()` und `oclIsNew()` liefern jeweils `false`. Falls `allInstances()` unterstützt werden soll, darf es aber nur die auch über `self` erreichbare Instanz zurückgeben. Kollisionen zwischen Namen von neu erstellten Klassen

[15] Damit wird ein Constraint über das Ergebnis einer Methode ausgedrückt, definiert werden können Methoden mittels `def`

und Klassen des UML-Modells müssen vermieden und die neuen Klassen nicht innerhalb von Parameterdefinitionen verwendet werden.

Ein weiterer Grund für die Lösung mittels einer echten Instanz im OCL-Kontext ist, dass der verwendete OCL-Ausdruck sich möglichst wenig von anderen, im ursprünglichen Umfeld eingesetzten unterscheiden soll. Während der Entwicklung von Charakteristiken könnten auch andere Werkzeuge zum Einsatz kommen, die die OCL-Ausdrücke bearbeiten oder auswerten. Mit einer veränderten Syntax oder einem nicht üblichen Kontext müssten diese Werkzeuge auch angepasst werden. Daher ist es günstiger, dem auf jeden Fall anzugebenden UML-Modell die Charakteristik-Klassen hinzuzufügen. Damit wird es wahrscheinlicher, dass das Werkzeug nicht verändert werden muss, zum Beispiel wenn es den oben gezeigten Weg über eine `body`-Klausel unterstützt oder der Ausdruck vorher einfach zu einer Nachbedingung über das Ergebnis umgeformt wird.

Es verbleiben jetzt noch die *Zugriffe auf `values`-Klauseln.* Im obigen Beispiel wurde die der `values`-Klausel zugeordnete Methode schon genauso genannt wie die Charakteristik. Bei den anderen sollte auch so verfahren werden: der Charakteristik-Klasse werden Methoden mit den Namen der Charakteristiken hinzugefügt. Die Methoden haben die gleiche Signatur wie die Charakteristikn und sind dann einfach über `self` erreichbar. Kollisionen zwischen den einzelnen Kategorien gibt es nicht, die eigene oder eine Basis-Charakteristik kann nicht gleichzeitig als Grundlage dienen.

Zu beachten ist, dass hier nur die Umgebung einer einzigen `values`-Klausel im Rahmen des Mess-Kontextmodells aufzubauen ist. Auch wenn der Aufruf einer anderen Methode in der gleichen Instanz bleibt, kann die Implementierung der Charakteristik-Klasse im Messsystem dann den Aufruf an eine andere Instanz der entsprechenden anderen Charakteristik-Klasse weiterleiten. Das stellt sicher, dass es für die `values`-Klausel nur die Sicht auf die lokale Charakteristik gibt. Andere Lösungen, zum Beispiel zuerst zu einer anderen Instanz zu navigieren und dann eine Methode dieser aufzurufen, füllen die Umgebung mit deutlich mehr CQML$^+$-fremden Informationen.

Als nächstes ist zu entscheiden, auf welche von den oben aufgeführten Arten von `values`-Klauseln zugegriffen werden kann. Dafür muss festgelegt werden, ob Rekursion erlaubt werden soll oder nicht. Mit Rekursion bieten sich den Autoren von Charakteristiken mehr Möglichkeiten. Rekursion ist dann möglich, wenn sich eine `values`-Klausel irgendwann selbst aufrufen kann, sich also Zyklen im von der Charakteristik ausgehend gebildeten Abhängigkeitsgraphen befinden. Beim Ermitteln dieses Graphen müssen alle Arten von `values`-Klauseln beachtet werden.

Ohne Rekursion beeinflusst die Entscheidung, auf welche Arten von `values`-Klauseln zugegriffen werden kann, die Ausdrucksstärke nicht. Allerdings kann Redundanz beim Entwickeln von Charakteristiken vermieden

werden. Falls Rekursion erlaubt ist und als Grundlage dienende `values`-Klauseln zugänglich sind, ist der Zugriff auf die eigene `values`-Klausel über den Umweg einer anderen Charakteristik möglich und kann somit auch erlaubt werden.

3.4.2 Datentypen in CQML$^+$, OCL und im Kontextmodell

Kontextmodelle und die Domänen von Charakteristiken werden durch `values`-Klauselen und deren OCL-Ausdrücke miteinander verbunden. Deshalb müssen die dabei unterschiedlichen Typsysteme aufeinander abgebildet werden. Abbildung 6 zeigt als Spalten die einzelnen Typsysteme, im oberen Teil sind die nötigen Abbildungen und im unteren Teil die abzubildenden Typen zu sehen.

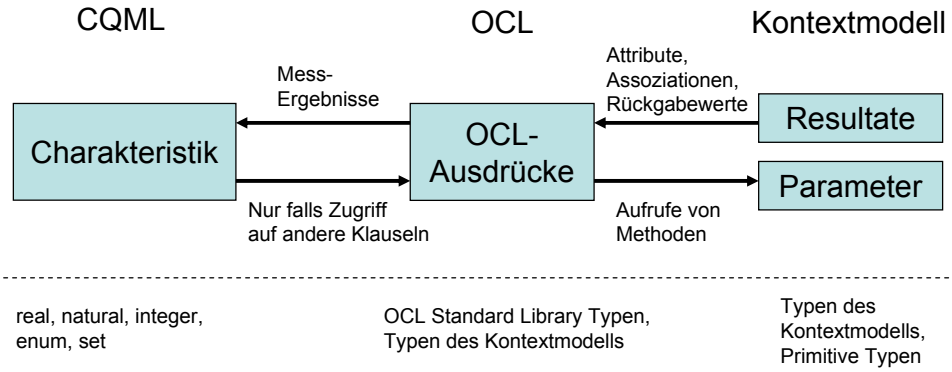

Abbildung 6: Datentypen in CQML$^+$, OCL und im Kontextmodell

In OCL-Ausdrücken stehen Typen aus der OCL Standard Library [OCL03] und dem UML-Modell, in unserem Fall also dem Kontextmodell, zur Verfügung. Im Kontextmodell gibt es nur die durch das Modell definierten Klassen und primitive Typen.

Zuerst sollen die Beziehungen zwischen OCL- und CQML-Typen untersucht werden. Die Domänen von Charakteristiken können entweder Zahlen, Enumerationen oder Mengen sein. Dafür gibt es direkte Entsprechungen aus der OCL Standard Library:

- Zahlen können direkt übernommen werden, `real` entspricht `Real` aus OCL und für `integer` und `natural` wird `Integer` verwendet. Es werden jeweils unbeschränkt kleine oder große Zahlen repräsentiert und die Konvertierung ist in beide Richtungen möglich.

- Enumeration in CQML$^+$ können zum einen durch OCL-Enumerationen ausgedrückt werden, dafür muss der Enumerations-Typ aber Teil des UML-Modells sein, also ähnlich wie die Charakteristik-Klasse in dieses nachträglich eingefügt werden. Die CQML$^+$-Spezifikation sagt nichts

darüber aus, welchen Namen der Typ tragen soll, obwohl der zur Konstruktion von Mess-Ergebnissen nötig wäre. Alternativ lässt sich eine Enumeration wie ein OCL-String behandeln, indem der Bezeichner in die Zeichenkette umgewandelt werden kann. Ebenso lassen sich mit wenig Aufwand Bezeichner in Form von Zeichenketten in Enumerationen verwandeln.

- Mengen in CQML$^+$ haben Enumerationen als Elemente. In OCL gibt es für Mengen den Typ `Set`, dessen Element-Typ dann auf den Typ für Enumerationen gesetzt werden muss. Ansonsten sind OCL- und CQML$^+$-Menge äquivalent.

Beachtet werden müssen noch die möglichen Einschränkungen einer Charakteristik-Domäne. Bei den Zahlen ist das die Angabe von oberen und unteren Schranken für den Wertebereich, `natural` kann als `integer` angesehen werden, der größer oder gleich null sein muss. Falls für Enumerationen Strings verwendet werden, so ist die Definition der möglichen Werte einer Enumeration auch eine Einschränkung.

Diese Einschränkungen gelten für die CQML$^+$-Typen und sind deshalb nur bei der Umwandlung von OCL-Typen in CQML$^+$-Typen interessant. Die einzigen Konstruktoren für CQML$^+$-Typen sind `values`-Klauseln. Da der Code für diese generiert wird, kann auch Code für die Überprüfung der Einschränkungen erstellt werden. Dieser ist dann jeweils nach einer `values`-Klausel auszuführen und entscheidet, ob die Charakteristik einen gültigen Wert hat. Im Gegensatz dazu können CQML$^+$-Typen, beispielsweise beim Aufruf anderer `values`-Klauseln, immer problemlos in OCL-Typen umgewandelt werden.

Die CQML$^+$-Typen müssen somit auch nicht extra implementiert werden, man kann stattdessen einfach die OCL-Typen verwenden.

Die Typen des Kontextmodells kommen sowohl im Kontextmodell als auch in den OCL-Ausdrücken vor, und müssen somit nicht umgewandelt werden. Das trifft aber nicht auf die primitiven Typen des Kontextmodells und die Typen der OCL Standard Library zu.

OCL-Typen können sowohl in primitive als auch in passende Typen des Kontextmodells, beispielsweise im Fall von Kollektionen, übersetzt werden. Diese Abbildung müsste dann durch das Kontextmodell definiert und bei der Generierung von Messcode entsprechend beachtet werden. So eine Abbildung und insbesondere das Generieren von Code für diese ist aber alles andere als trivial, und damit als Lösung ungeeignet.

Günstiger ist es, die OCL-Typen in jedem Kontextmodell zugänglich zu machen und anstatt von primitiven Typen als Parameter zu verwenden. Ähnlich kann auch bei in OCL-Typen zu konvertierenden Resultaten argumentiert werden. Letztendlich verbleiben dann nur noch die Typen der OCL Standard Library und die Kontextmodell-Typen. Damit werden Entwickler

entlastet und es gibt weniger Abbildungsprobleme, zum Beispiel aufgrund unterschiedlicher Wertebereiche bei den diversen Integer-Typen.

Die Code-Schnittstelle muss auch nicht mit vollständigen OCL-Typen umgehen können. Es reicht, wenn in dieser schlanke Typen verwendet werden, die nur Daten speichern und einfache Zugriffsoperationen besitzen. Ein OCL-`Integer` muss für die Schnittstelle zum Beispiel nur eine Methode zum Ermitteln des Wertes enthalten. Der generierte Messcode ist dann bei Zugriffen auf die Schnittstelle für die Konvertierung zwischen vollständigen und schlanken Typen verantwortlich.

Bei der Umwandlung von Resultaten in OCL-Typen sollte analog verfahren werden.

Als letztes sollen noch nicht-definierte Werte betrachtet werden. Alle OCL-Typen können den Wert `undef` annehmen. Werden Operationen ausgeführt, bei denen mindestens ein Parameter nicht definiert ist, ist auch das Ergebnis der Operation nicht definiert. Ausnahmen sind Ausdrücke wie `true or undef`. Dieses Fortpflanzen eines nicht definierten Wertes bis an die Spitze eines Berechnungsbaums gleicht den Konsequenzen von Ausnahmen in Programmiersprachen wie zum Beispiel Java. Damit können Fehlersituationen nach außen signalisiert werden, ohne das jeder Teilausdruck explizit dafür Sorge zu tragen hat.

Aus diesem Grund sollten nicht-definierte Werte zum Anzeigen von Fehlersituationen verwendet werden. Methoden des Kontextmodells werden nie nicht-definierte Werte erhalten, da das Ergebnis durch obige Regel schon vorher feststeht. Elemente des Kontextmodells können aber nicht-definierte Werte zum signalisieren von Fehlern zurückgeben, die schlanken Typen wären dann entsprechend zu erweitern. Alternativ können auch andere Mittel wie Ausnahmen dazu verwendet werden, die dann wiederum vom Messcode als nicht-definierte Werte behandelt werden müssen. Charakteristiken nehmen nicht-definierte Werte an, wenn diese in der `values`-Klausel entstanden oder Einschränkungen der Charakteristik-Domäne nicht eingehalten wurden. Beim Zugriff auf andere `values`-Klauseln wird der Fehler somit auch an den aufrufenden Ausdruck weitergereicht.

3.4.3 Generieren von Mess-Code

Die Umgebung der `values`-Klausel und die Untersuchung der Typsysteme bilden die Grundlage für den Code-Generator. Im Folgenden werden die Arbeitsschritte für das Generieren von Code dargestellt.

Damit für eine Charakteristik Messcode generiert werden kann, muss diese eine `values`-Klausel haben und die Domäne muss definiert sein. Weiterhin muss die `values`-Klausel einen syntaktisch und semantisch korrekten OCL-Ausdruck enthalten und der Typ des Ausdrucks muss zur Domäne der Charakteristik passen.

Als Basis für den Code-Generator soll das Dresden OCL-Toolkit [DOT] verwendet werden. Es enthält unter anderem folgende für diese Arbeit relevante Komponenten:

Java-Implementierung der OCL-Basisbibliothek [Fin99]: Die Basisbibliothek enthält Implementierungen der Typen der OCL Standard Library.

OCL-Compiler [Fin00]: Der OCL-Compiler besteht im wesentlichen aus einem Parser und einem Code-Generator. Unterstützt wird OCL in der Version 1.3. Als Eingabe werden vollständige OCL-Constraints erwartet. Der generierte Java-Code verwendet die Implementierung der OCL-Basisbibliothek.

Es existiert noch ein metamodellbasierter OCL-Compiler [Ock03]. Dieser unterstützt OCL 2.0 und verwendet die gleiche Implementierung der OCL-Basisbibliothek, ein passender Parser ist aber Thema einer anderen Arbeit und momentan noch nicht fertig. Außerdem konzentriert sich dieser Compiler bisher auf das Erstellen von Code, der OCL-Constraints in Modellen ausführt. Die OCL-Ausdrücke der Charakteristiken beziehen sich aber auf den Mess-Kontext, also Instanzen des Mess-Kontextmodells.

Längerfristig ist trotzdem [Ock03] vorzuziehen, da dieser Compiler eine neuere OCL-Version unterstützt und die Beziehungen zu den Meta-Modellen von OCL beziehungsweise UML eine einfacher zu wartende und vor allem zu aktualisierende Architektur ermöglichen. Weiterhin ist ein Wechsel zu einer anderen Sprache (momentan wird nur Java-Code generiert) mit dieser Architektur einfacher umzusetzen. Da aber beide Compiler die gleiche Java-Implementierung der OCL-Basisbibliothek verwenden, müsste diese dafür auch umgestellt werden.

Kurzfristig können aber mit dem älteren Compiler [Fin00] bessere Resultate erzielt werden. Die kurzfristigen, praxisnahen Ergebnisse sind zum Beispiel für den Prototypen in dieser Arbeit nötig. Der mit diesem generierte Messcode ist nicht bei weitem nicht für einen echten Einsatz geeignet, kann aber für das Testen von Charakteristik-Definitionen und ähnlichem verwendet werden.

Unabhängig vom gewählten Compiler sind folgende Arbeitsschritte für das Generieren von Code für eine Charakteristik erforderlich:

1. Die Charakteristik-Definition muss geladen werden. Wichtig sind dabei die Domäne, die `values`-Klausel, die formalen Parameter der Charakteristik und deren Typen, die Invarianten und die Liste der als Grundlage dienenden Charakteristiken.

2. Das Mess-Kontextmodell und muss geladen und im Compiler als UML-Modell für den OCL-Ausdruck gesetzt werden.

3. Basierend auf der Definition der Charakteristik wird dem UML-Modell eine Klasse hinzugefügt, die den Kontext der `values`-Klausel darstellt (siehe Kapitel 3.4.1).

4. Aus der `values`-Klausel wird ein OCL-Ausdruck gebildet, für den der konkrete Compiler Code generieren kann. Bei der Verwendung von [Fin00] sind das zum Beispiel nur OCL-Constraints, so dass mit Hilfe der Parameter und der Domäne der Bedingung eine OCL-Vorbedingung erstellt werden muss:

   ```
   context delay_Charakteristik::delay(c:OperationCall):Real
   pre: c.returnTime - c.callTime
   ```

   In diesem Fall müsste im Compiler noch die Überprüfung des Typs des OCL-Ausdrucks abgeschaltet werden, da eine Vorbedingung ein boolsches Ergebnis erwartet.

5. Der Ausdruck ist zu parsen, zu normalisieren und mit Typ-Informationen zu versehen. Weiterhin muss er vom Compiler auf semantische Korrektheit überprüft werden. Das Ergebnis ist ein abstrakter Syntax-Baum oder eine ähnliche abstrakte Repräsentation des OCL-Ausdrucks.

6. Es muss überprüft werden, wie auf die Klasse, die den Kontext der `values`-Klausel darstellt, zugegriffen wird. Alle Eigenschaften, die dieser Klasse hinzugefügt wurden, aber nicht vom OCL-Ausdruck gesehen werden sollen, dürfen nicht referenziert werden. Das ist zum Beispiel der Fall bei der Methode für die eigene `values`-Klausel, wenn keine Rekursion erlaubt ist. Soll auf gar keine Eigenschaft zugegriffen werden können, so kann auch der Zugriff auf die gesamte Klasse verboten werden.

7. Der Typ des OCL-Ausdrucks muss zur Domäne der Charakteristik passen (siehe Kapitel 3.4.2).

8. Der Code ist aus diesem Ausdruck zu generieren und geeignet zu verpacken. Zum Beispiel kann er durch eine Methode umschlossen sein, die die gleiche Signatur wie die Messfunktion, also Domäne und Parameter der Charakteristik, hat.

9. Es ist eine Liste mit durch den OCL-Ausdruck referenzierten Typen zu bilden. Diese grenzt den Wertebereich der Messfunktion ein und kann für das änderungsgesteuerte Messen verwendet werden. Für ein detaillierteres Ergebnis kann auch noch nach Eigenschaften von Typen unterschieden werden.

10. Für die Invarianten ist mit ähnlichem Vorgehen Code zu generieren. Dabei besteht der Kontext für die Ausdrücke in nur einer Klasse mit

einer Methode. Die Methode erhält einen Parameter, der die Domäne der Charakteristik als Typ hat. Als Ergebnis liefert die Methode einen boolschen Wert.

Der auf diesem Weg erstellte Code kann anschließend in ausführbaren Code übersetzt und verwendet werden. Dieser Teil des Messsystem, also das Verwalten von Messaufträgen zur Laufzeit, ist aber nicht Teil dieser Arbeit und wird deshalb auch nicht näher behandelt.

3.5 Einbettung des Messsystems in den Container

Das Messsystem muss in den Container integriert werden, damit es einen Mess-Kontext basierend auf den Ereignissen und Zuständen im Container verwalten kann. In Abbildung 3 auf Seite 7 ist die Architektur des Containers abgebildet, mit nicht-zusagefähigem Teil (NZF-Teil) auf der linken Seite und zusagefähigem Teil (ZF-Teil) auf der rechten Seite. Im Folgenden wird nur die Positionierung des Messsystems bezüglich dieser Architektur betrachtet.

Als ein Grund für das Ausführen von Messungen wurde Adaption genannt. Diese braucht Messergebnisse ohne große Verzögerungen, damit entsprechend schnell reagiert werden kann. Das Messsystem kann solche Zusagen zu einer bestimmten Dienstgüte bei der Übermittlung von Messergebnissen aber nur geben, wenn seine Laufzeitumgebung selbst Zusagen abgeben kann. Aus diesem Grund müsste sich das Messsystem also im ZF-Teil befinden. Bei Messungen, deren Ergebnisse mit Verzögerungen eintreffen können, könnte sich das Messsystem auch im NZF-Teil befinden. Die nichtfunktionalen Anforderungen an die Übermittlung sind also ein Aspekt für die Wahl der Position.

Weiterhin muss betrachtet werden, wo sich die zu messenden Daten überhaupt befinden, beziehungsweise wo sie erzeugt werden. Zusagefähige Komponenten werden im ZF-Teil und nicht-zusagefähige im NZF-Teil verwaltet. Damit entstehen die Ereignisse und Zustandsänderungen auch in diesen Teilen. Um dann auf Daten auf der jeweils anderen Seite zuzugreifen, müssten die lesenden Transaktionen, die die Messfunktionen repräsentieren, sich auch auf diese Daten beziehen.

Wenn vom ZF-Teil auf den NZF-Teil zugegriffen werden soll, ist das nicht möglich. Die Transaktion muss auch im ZF-Teil eingeplant werden, das geht aber nicht, weil der NZF-Teil keine entsprechenden Zusagen über das Abliefern von Ergebnissen geben kann. Eine nicht garantierte Messung wäre möglich, aber diese nützt bei vielen Messgründen nicht (zum Beispiel dem Überprüfen von Zusagen).

In die andere Richtung ist es auch nur schwer möglich, da die Teile der der Transaktion, die sich auf den ZF-Teil beziehen, auf einmal übergeben werden

müssen (der NZF-Teil kann nicht garantieren, dass er die Operationen in einer bestimmten Zeit an den ZF-Teil schickt).

Wenn Messungen außerdem noch einen Bezug zur Zeit haben, und die Zeit zum Beispiel von verstrichener Prozessorzeit abhängt, dann kann keine Messung im NZF-Teil garantiert werden, weil dafür der Ressourcenverbrauch vorher zu planen wäre.

Es muss somit zwischen Messungen und Mess-Kontexten im ZF- und im NZF-Teil unterschieden werden. Innerhalb des ZF-Teils kann immer gemessen werden, und diese Ergebnisse können dann auch an den NZF-Teil übermittelt werden. Im NZF-Teil kann keine Garantie über den Erfolg einer Messung zu einem bestimmten Zeitpunkt und die schnelle Übermittlung des Ergebnisses gegeben werden.

3.6 Unterschiede zur CQML$^+$-Spezifikation

3.6.1 Statistische Aspekte

Statistische Aspekte werden in CQML$^+$ als Werte beschrieben, die durch Messung über einen Zeitraum entstehen und auf Messwerten basieren. Es bleiben aber einige Fragen unbeantwortet:

- Es wird für die statistischen Aspekte definiert, wie aus Werten das Ergebnis gebildet wird, zum Beispiel bei `mean` das arithmetische Mittel. Es wird aber wenig darüber ausgesagt, wie diese Werte entstehen und in die Berechnung eingehen. Einfach gestaltet es sich bei Minimum und Maximum, wo mittels änderungsgesteuertem Messen alle Veränderungen des Messwertes ermittelt werden müssen. Aber zum Beispiel bei Mittelwert oder Frequenz ist nicht klar, ob Veränderungen jeweils einen neuen Wert für die Berechnung begründen oder die Messwerte in ihrem zeitlichen Verlauf betrachtet und nach Dauer gewichtet werden. Letzteres ist wohl die öfter gewünschte Variante, trotzdem muss es vorher festgelegt werden.

- Der Zeitraum, in dem Messwerte für die statistischen Aspekte ermittelt werden, ist nicht bekannt. Wenn zum Beispiel eine stichprobenartige Messung auf einen statistischen Aspekt zugreift, wird dann mit dem Beobachten des Messwertes erst begonnen oder muss die Framework-Implementierung alle potentiellen Messwerte von vornherein betrachten?

- Bei Zugriffen auf statistische Aspekte innerhalb von `values`-Klauseln oder entsprechenden Ausdrücken in CQML$^+$-Statements muss noch spezifiziert werden, welche Lebensdauer die Aspekte haben, und wie viele Instanzen zum Speichern des Zustands existieren. Es könnte zum Beispiel wie folgt argumentiert werden:

1. Die statistischen Aspekte werden von einer bestimmten Messung verwendet und könnten sich somit an deren Lebensdauer ausrichten und dieser zugeordnet sein (aber vergleiche oben, Zeitraum der Messwert-Erhebung).
2. Entweder werden in einer Messung direkt bestimmte Eigenschaften abgefragt und damit eine Charakteristik-Instanz begründet, oder in CQML$^+$-Statements werden Bedingungen über parametrisierte Charakteristiken angegeben. Im zweiten Fall könnte jeweils pro Paar aus Charakteristik und Parameter-Belegung eine Instanz erstellt werden (falls Charakteristik und Parameter gleich sind, ist auch der Messwert immer gleich.

- Es ist zu klären, ob von `values`-Klauseln aus auf statistische Aspekte zugegriffen werden kann. Da die Messfunktionen auf die Aspekte zugreifen, diese aber wiederum von den Ergebnissen der Messfunktionen abhängen, kann es schnell zu nicht konvergierenden Funktionen kommen. So muss z.B. beim arithmetischen Mittel genau der Mittelwert zurückgegeben werden, damit die Berechnung terminiert. Es liegt die Vermutung nahe, das statistische Aspekte erst ab der Stufe der CQML$^+$-Statements sinnvoll einsetzbar sind.

Aus diesen Gründen werden statistische Aspekte erst unterstützt werden können, wenn die Spezifikation vollständig ist und es Regelungen für obige Punkte gibt. In der aktuellen Form gibt es zu viele Unklarheiten und das Beheben dieser ist zu weit vom Thema dieser Arbeit entfernt.

3.6.2 Invarianten

Die Spezifikation von CQML$^+$ geht auch auf Invarianten nur eher oberflächlich ein. Als Motivation für diese wird angegeben, dass sie die Domäne von Charakteristiken eingrenzen sollen. Invalide Werte als Ergebnis von Messungen sollen auf erlaubte Werte zurückgesetzt werden, für diese Berichtigung gibt es aber keine Regeln. In einem Beispiel wird das Verfahren angedeutet. Dabei wird untersucht, welcher Teil des boolschen Ausdrucks der Invariante verletzt wird und der Messwert wird so berichtigt, dass diese Verletzung nicht mehr auftritt. Das könnte zwar automatisch möglich sein, da für die Domänen jeweils eine Ordnungsrelation definiert ist, das schlecht erratene Resultat rechtfertigt den Aufwand aber keinesfalls. Stattdessen sollte zu jeder Invariante ein Wert angegeben werden, auf diesen der Messwert bei Verletzung berichtigt wird.

In der Beschreibung der Semantik von CQML$^+$ sind Invarianten als Prädikate modelliert. Das findet sich aber nicht in der Syntax der Beispiele wieder. Ähnlich zum Problem der Definition des OCL-Kontextes bei der `values`-Klausel muss auch hier verfahren werden, es reicht aber, wenn der Messwert über einen einzelnen, standardisierten Parameter zugänglich ist.

Eine mögliche Syntax für das Angeben von Invarianten könnte zum Beispiel wie folgt aussehen. Hinter `invariant` steht der Wert, auf den berichtigt wird und `value` ist der Parameter. Mit den folgenden Angaben würde der Wert auf einen Bereich von 10 bis 50 beschränkt werden:

```
invariant 10: value < 10
invariant 50: value > 50
```

Das gleiche könnte zwar durch `if-then-else` Konstrukte in der `values`-Klausel erreicht werden, diese würden dann aber nicht getrennt von der `values`-Klausel an Sub-Charakteristiken vererbt werden.

Falls Invarianten den Messwert nicht einfach berichtigen können oder dies nicht erwünscht ist, muss die Messung als ungültig erklärt werden. Dafür ist in CQML$^+$ noch kein Mechanismus festgelegt. Da aber die Domänen von Charakteristiken kein Element enthalten, das dem `undef`-Element bei OCL-Typen entspricht, bietet sich dieses zum Markieren von ungültigen Messungen an:

```
invariant undef: value < 20
```

Einschränkungen bei der Domäne einer Charakteristik könnten damit auch ausgedrückt werden, zum Beispiel wäre obige Invariante äquivalent zu der Angabe `numeric [20..)`.

4 Prototyp

Ziel des Prototyps ist es, einen Machbarkeitsnachweis in Bezug auf die Generierung von Messcode abzuliefern. Lösungen oder Code sollen soweit möglich wiederverwendet werden können.

Die geforderte prototypische Umsetzung einer Laufzeitunterstützung ist beim momentanen Entwicklungsstand des COMQUAD-Projektes eher nicht möglich. Eine wirklich sinnvolle und praxisrelevante Laufzeitunterstützung müsste sich im zusagefähigen Teil befinden (siehe Kapitel 3.5). Dieser ist aber noch nicht ausgereift, und es kann auch kein Java-Messcode in diesem ausgeführt werden, die zu verwendenden OCL-Compiler erzeugen aber zum Beispiel Java-Code. Auch eine teilweise, auf den NZF-Teil abgestimmte Umsetzung der in den Kapiteln 3.2, 3.3.2 und 3.3.3 genannten Verfahren würde den Rahmen dieser Arbeit sprengen. Außerdem sind diese Verfahren immer im Kontext eines konkreten Containers zu lösen, ohne die Existenz einer ausgereiften Container-Version wird sich die entstandene Implementierung dann auch nur schlecht weiterverwenden lassen.

Deshalb beschränkt sich der Prototyp auf das Generieren von Code. Auch dabei musste abgewogen werden zwischen der langfristig gesehen besseren Lösung und einer aktuell nützlichen. Die Entscheidung fiel auf letztere, da diese einen fertigen Parser beinhaltet und nur damit der Prototyp auch in der Praxis zum Testen von Charakteristiken eingesetzt werden kann. Das prinzipielle Vorgehen beim Generieren von Code ist unabhängig vom eingesetzten Compiler.

Der Prototyp setzt die in Kapitel 3.4.3 auf Seite 35 genannten Schritte in der Methode `generateCode` der Klasse `CodeGenerator` um:

- Das Mess-Kontextmodell wird als XMI-Dokument[16] geladen.
- Die Klasse für den Kontext des OCL-Ausdrucks wird dem Modell über den Adapter `VcModelFacade` hinzugefügt. Dieser beantwortet nur Anfragen für diese Klasse und leitet andere Anfragen an das ursprüngliche Modell weiter.
- Die in den Compiler eingebaute Bedingung, dass OCL-Ausdrücke den Ergebnistyp `Boolean` haben müssen, wird durch die Klasse `VcTypeChecker` gelöst, die diese Bedingung im Original durch einen eigenen Test ersetzt. Dieser Test überprüft, ob der Typ des OCL-Ausdrucks zur Domäne der Charakteristik passt.

[16] Ein Metadaten-Austauschformat, `http://www.omg.org/cgi-bin/doc?formal/2002-01-01`

- Falls die Klasse für den Kontext im OCL-Ausdruck auftritt, wird kein Code generiert. Dafür wird die Liste der referenzierten Typen durch die Klasse `VcAnalysis` ermittelt.

Die Ausgabe für die Charakteristik in Listing 1 sieht wie folgt aus:

```
//generating code for:
//context CqmlChr0::valuesClause(c:OperationCall):Real
//pre:c.returnTime - c.callTime
//generated method for values clause:
java.lang.Float myMethodName(OperationCall c){
final tudresden.ocl.lib.OclAnyImpl tudOclNode0=
  tudresden.ocl.lib.Ocl.toOclAnyImpl( tudresden.ocl.lib.Ocl.getFor(new Object()) );
final tudresden.ocl.lib.OclAnyImpl tudOclOpPar0=
  tudresden.ocl.lib.Ocl.toOclAnyImpl( tudresden.ocl.lib.Ocl.getFor(c) );
final tudresden.ocl.lib.OclReal tudOclNode1=
  tudresden.ocl.lib.Ocl.toOclReal(tudOclOpPar0.getFeature("returnTime"));
final tudresden.ocl.lib.OclReal tudOclNode2=
  tudresden.ocl.lib.Ocl.toOclReal(tudOclOpPar0.getFeature("callTime"));
final tudresden.ocl.lib.OclReal tudOclNode3=
  tudOclNode1.subtract(tudOclNode2);
return tudresden.ocl.lib.Ocl.reconvert(java.lang.Float.class, tudOclNode3);
}
//types used by values clause:
//[OperationCall, Real, OclType]
```

Die Indirektion mittels Methoden für Attribute und Assoziation des Mess-Kontextmodells(siehe Kapitel 3.3.1) wurde nicht umgesetzt. Dafür müsste aber auch eine angepasste Variante der Klasse `OclAnyImpl` aus der Basisbibliothek verwendet werden. Diese setzt die oben zu sehenden `getFeature()`-Aufrufe in Zugriffe auf die Objekte mittels Java Reflection um, die dabei verwendeten Namen müssen sich also nach den Benennungsregeln für die Indirektionsmethoden richten. Mit alleiniger Hilfe durch `NameAdapter` kann das nicht gelöst werden, denn der beachtet keine Operationen.

5 Zusammenfassung und Ausblick

In dieser Arbeit wurde das Messen von nicht-funktionalen Eigenschaften untersucht. Es wurden aus den Anwendungsfällen zwei Messverfahren abgeleitet:

- Stichprobenartiges Messen erfasst die Werte von Eigenschaften zu bestimmten Zeitpunkten
- Änderungsgesteuertes Messen erfasst alle Änderungen des Wertes einer Eigenschaft

Anschließend wurde der Entwicklungsprozess und die in diesem vorkommenden Rollen und Artefakte untersucht, das Mess-Kontextmodell wurde dabei als Schnittstelle zwischen Messsystem und generiertem Messcode festgelegt.

Für das Messsystem wurden die Beziehungen zwischen Mess-Kontextmodell und dem Container sowohl aus statischer als auch aus dynamischer Sicht betrachtet und konzeptionelle Lösungen für die dabei auftretenden Probleme gefunden. Die Schnittstelle auf Implementierungsebene zu einer Instanz eines Mess-Kontextmodells wurde beschrieben.

Es wurde untersucht wie die Umgebung von `values`-Klauselen genau aussieht und wie solche Umgebungen für die OCL-Ausdrücke der `values`-Klauseln umgesetzt werden können, ohne dabei einen neuen OCL-Dialekt zu erschaffen. Die Typsysteme von CQML$^+$, OCL und den Kontextmodellen wurden zueinander in Beziehung gesetzt und Abbildungen zwischen ihnen gefunden.

Abschließend wurden Änderungen an der CQML$^+$-Spezifikation vorgeschlagen und ein einfacher Prototyp vorgestellt.

Das Aufgabengebiet wurde nicht vollständig bearbeitet. Deshalb folgt eine Liste mit wichtigen offenen Problemen und möglichen Aufgaben:

Nicht-funktionale Anforderungen an die Übermittlung von Messergebnissen: Es wurde schon gesagt, dass Messsystem und Auftraggeber in einem Produzent-Konsument-Verhältnis stehen. Die Anforderungen, die einzelne Auftraggeber an die Übermittlung stellen, sollten bei der Ermittlung von Messwerten beachtet und garantiert werden können, insbesondere bei der Planung der Anordnung von Transaktionen.

Möglichkeiten zur Analyse von `values`-Klauselen: Die Angaben in der Literatur zur Mächtigkeit von OCL-Ausdrücken beziehen sich immer nur auf pure OCL-Ausdrücke, das heißt sie haben ein leeres UML-Modell. Es sollten die Grenzen der Analysemöglichkeiten im Fall von `values`-Klauselen und Kontextmodellen untersucht werden.

Eingrenzung des Wertebereichs von Messfunktionen: Die hier beschriebene Eingrenzung des Wertebereichs mittels der Klassen des Kontextmodells, auf die eventuell zugegriffen werden kann, ist zu grob.

Reservierung von Ressourcen für Mess-Kontexte: Mess-Kontexte wachsen, wenn sie eine vollständige Geschichte der Ereignisse speichern sollen. Es ist zu untersuchen, wie die Größe bzw. Anzahl der Ereignisse beschränkt werden kann. Dabei sollten die Messfunktionen betrachtet werden, und auf welchen Teil dieser Geschichte sie wirklich zugreifen.

Ermitteln des Ressourcenbedarfs von Messfunktionen: Der Ressourcenbedarf von Messfunktionen beziehungsweise dem generierten Messcode sollte automatisch ermittelt werden können, damit die entsprechenden lesenden Transaktionen eingeplant werden können. Alternativ ist im Entwicklungsprozess festzulegen, wer für die Angabe des Ressourcenbedarfs verantwortlich ist.

Literatur

[Aag01] Jan Øyvind Aagedal. *Quality of Service Support in Development of Distributed Systems*. Dissertation, University of Oslo, 2001.

[ABF+03] Ronald Aigner, Henrike Berthold, Elke Franz, Steffen Göbel, Herrmann Härtig, Heinrich Hußmann, Klaus Meißner, Klaus Meyer-Wegener, Marcus Meyerhöfer, Andreas Pfitzmann, Simone Röttger, Alexander Schill, Thomas Springer und Frank Wehner. COMQUAD – Komponentenbasierte Softwaresysteme mit zusagbaren quantitativen Eigenschaften und Adaptionsfähigkeit. *Informatik Forschung und Entwicklung*, 18:39–40, 2003.

[DOT] Dresden OCL-Toolkit, `http://dresden-ocl.sourceforge.net`.

[Fin99] Frank Finger. Java-Implementierung der OCL-Basisbibliothek. Technical report, 1999.

[Fin00] Frank Finger. Entwurf und Implementation eines modularen OCL-Compilers. Diplomarbeit, Dresden University of Technology, Department of Computer Science, 2000.

[GPRZ04] Steffen Göbel, Christoph Pohl, Simone Röttger und Steffen Zschaler. The COMQUAD Component Model – Enabling Dynamic Selection of Implementations by Weaving Non-functional Aspects. In *International Conference on Aspect-Oriented Software Development (AOSD'04)*, Lancaster, UK, 22–26 Marz 2004. ACM. To appear.

[HBB+98] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg und J. Wolter. DROPS: OS Support for Distributed Multimedia Applications. In *Proc. 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications (Sintra, Portugal, Sept. 1998)*, Sintra, Portugal, September 1998.

[Ock03] Stefan Ocke. Entwurf und Implementation eines metamodellbasierten OCL-Compilers. Diplomarbeit, Dresden University of Technology, Department of Computer Science, 2003.

[OCL03] UML 2.0 OCL 2nd revised submission. OMG Document, `http://www.omg.org/cgi-bin/doc?ad/2003-01-07`, Januar 2003.

[RZ03a] Simone Röttger und Steffen Zschaler. CQML+: Enhancements to CQML. In Jean-Michel Bruel, Hrsg., *Proc. 1st Intl. Workshop on Quality of Service in Component-Based Software Engineering, Toulouse, France*, Seiten 43–56. Cépaduès-Éditions, Juni 2003.

[RZ03b] Simone Röttger und Steffen Zschaler. A Software Development Process Supporting Non-funtional Properties. 2003.

Abbildungsverzeichnis

1 Beispiel für ein Kontextmodell 6
2 Beziehungen zwischen Charakteristiken, Kontext- und Komponentenmodellen 6
3 Architektur des Containers 7
4 Entwicklungsprozess für nicht-funktionale Eigenschaften . . . 13
5 Transformationen von `values`-Klauseln und Modellen 15
6 Datentypen in CQML$^+$, OCL und im Kontextmodell 33

Listings

1 Beispiel für eine Charakteristik-Definition in CQML 5

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Dresden, den 31. Januar 2004