INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Fortgeschrittenenpraktikum

Neuimplementierung eines Compilers für eine Sprache zur Definition von Dienstgütemerkmalen mit JavaCC

Johannes Schaal

Aufgabensteller: Prof. Dr. Heinz-Gerd Hegering

Betreuer: Dr. Markus Garschhammer
Martin Sailer

Abgabetermin: 15. Juli 2006

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Fortgeschrittenenpraktikum

Neuimplementierung eines Compilers für eine Sprache zur Definition von Dienstgütemerkmalen mit JavaCC

Johannes Schaal

Aufgabensteller: Prof. Dr. Heinz-Gerd Hegering

Betreuer: Dr. Markus Garschhammer
Martin Sailer

Abgabetermin: 15. Juli 2006

Hiermit versichere ich, dass ich das vorliegende Fortgeschrittenenpraktikum selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. Juli 2006

. . . . . .

(Unterschrift des Kandidaten)

Die Qualität von IT-Diensten wird durch sogenannte Dienstgütemerkmale beschrieben. Nachdem diese für unterschiedliche Dienste jeweils spezifisch sind, entsteht bei Vertragsverhandlungen und Messungen das Problem der formalen Spezifikation der bereitzustellenden Dienstgüte. Dazu wurde an diesem Lehrstuhl eine formale Sprache namens QoSSL bzw. eQoSuL entwickelt, mit deren Hilfe sich solche Kriterien beschreiben lassen.

Für jene Spezifikationssprache wird in dieser Arbeit ein Übersetzer entwickelt, der aus einer solchen formalen Spezifikation ein ablauffähiges Meßsystem erzeugen kann. Dies geschieht auf Grundlage eines allgemeinen Meßprozesses für Dienstgütemerkmale auf Basis des MNM-Dienstmodells. Dabei wird aus jenem allgemeinen Modell ein für die gegebene Spezifikation von Dienstgütemerkmalen zugeschnittenes System abgeleitet.

Einhergehend mit dieser Entwicklung wird die Sprache QoSSL/eQoSuL syntaktisch formalisiert.

Der vorliegende Übersetzer unterstützt im Hinblick auf die geforderte Unabhängigkeit von der eingesetzten Technologie den Einsatz verschiedener Implementierungen von Dienstmodellen. Damit kann ein breites Spektrum von Anwendungsszenarien abgedeckt werden.

Inhaltsverzeichnis

1 Einleitung **1**

2 Problemstellung **2**
2.1 Meßprozeß für Dienstgüte 2
2.2 Rolle des Übersetzers 3

3 Anforderungen **4**

4 QoSSL-Syntax **6**

5 Implementierung des Übersetzers **9**
5.1 Programmiersprache 9
5.2 JavaCC 9
5.3 Zwei-Lauf-Verfahren 9
5.4 Aufbau des Programms 9
5.5 Symboltabelle 12

6 Funktionsweise, Ablauf der Übersetzung **14**
6.1 Quellprogramm 14
6.2 Zusammengesetztes Quellprogramm 15
6.3 Direktiven 17
6.4 Kommentare 17
6.5 Typsystem 17
6.6 Ermittlung der Korrelationsfunktionen und Aggregationsklassen 19
6.7 Parsen von Deklarationen 20
6.8 Ausdrücke 23
6.9 Algorithmen zur Vorwärtsreferenzierung 25
6.10 Zwischencode 26
6.11 Ausgabe 31

7 Zusammenfassung und Ausblick **37**
7.1 Spezifikationssprache 37
7.2 Übersetzer 37
7.3 Fehlerbehandlung 38

A QoSSL-Grammatik **41**

B Bestandteile des Compiler-Programms **43**
B.1 Paket `de.lmu.ifi.nm.qossl.aggregation` 45
B.2 Paket `de.lmu.ifi.nm.qossl.analyzer` 45
B.3 Paket `de.lmu.ifi.nm.qossl.base` 45
B.4 Paket `de.lmu.ifi.nm.qossl.generated` 46
B.5 Paket `de.lmu.ifi.nm.qossl.out` 46
B.6 Paket `de.lmu.ifi.nm.qossl.parser` 46
B.7 Paket `de.lmu.ifi.nm.qossl.semantics` 46
B.8 Paket `de.lmu.ifi.nm.qossl.types` 47
B.9 Beispielspezifikationen 47
B.10 Weitere Dateien und Verzeichnisse 47

C Handhabung des Übersetzers **49**
C.1 Voraussetzungen 49
C.2 Installation und Anwendung 49
C.3 Weiterführende Dokumentation 49

D Bereitstellung von Meßprozessen und Aggregationsfunktionalitäten **50**
D.1 Pakete 50
D.2 Aggregationsfunktionalitäten 51
D.3 Meßprozeß 52
D.4 Generator-Paket: Reflektor 52
D.5 Generator-Paket: Generator 54

Abbildungsverzeichnis

2.1 Darstellung des Dienstgüte-Meßprozesses als UML-Aktivitätsdiagramm; aus [Gars 04] 2
2.2 Darstellung des Automatisierungsprozesses aus [Gars 04] 3

5.1 Aufbau des Compiler-Programms 11

6.1 Das Typsystem der eingebauten Typen von QoSSL 19
6.2 Syntaxbaum des Berechnungsausdrucks `trans.count()` 24
6.3 Algorithmus zur Auswertung von Typisierungsausdrücken 25
6.4 Algorithmus zur Auswertung von Ausdrücken bei Verwendung 26
6.5 Grobe Struktur des Zwischencodes 27
6.6 Verfeinerte Darstellung des Zwischencodes 28
6.7 Aus der Spezifikation abgeleiteter Teil des Zwischencodes 29
6.8 Aus Direktiven und Bibliotheken abgeleiteter Teil des Zwischencodes 30

Tabellenverzeichnis

1 Einleitung

Die Umsetzung von Festlegungen von Dienstgütemerkmalen wurde bisher nicht in ausreichendem Maße durch Werkzeuge unterstützt. In [Gars 04] wird nun ein Verfahren eingeführt, das eine automatische Umsetzung solcher Festlegungen für die Bereitstellungs- und Nutzungsphase eines Dienstes erlaubt.
Zur Beschreibung dieser Formalisierung wurde dabei das Konzept einer formalen Sprache (*QoSSL, Quality of Service Specification Language*) entwickelt, die zur technologieunabhängigen Spezifikation von Dienstgütemerkmalen dient.
In dieser Arbeit wird ein Übersetzer für diese Spezifikationssprache aufgebaut, der im Gegensatz zu dem in [Gars 04] entwickelten Prototyp nicht als „Proof of Concept“, sondern zur tatsächlichen Anwendung vorgesehen und auch geeignet ist. Dies schließt eine Festlegung aller syntaktischen Elemente der Sprache QoSSL mit ein.

2 Problemstellung

2.1 Meßprozeß für Dienstgüte

Das vorliegende Konzept der Dienstgütespezifikation aus [Gars 04] geht von einem generischen, parametrisierbaren Meßprozeß aus. „Generisch“ bedeutet eine Unabhängigkeit von der verwendeten Technologie; unter den „Parametern“ sind z. B. die Berechnungsvorschriften für die Korrelation von Ereignissen und die statistische Nachverarbeitung von Meßwerten zu verstehen, die vom Entwickler frei gewählt werden können. Statt „generisch“ wird auch die Bezeichnung „allgemein“ verwendet.
Die Sprache QoSSL stellt einen Mechanismus zur Spezifikation eines solchen parametrisierten Meßprozesses

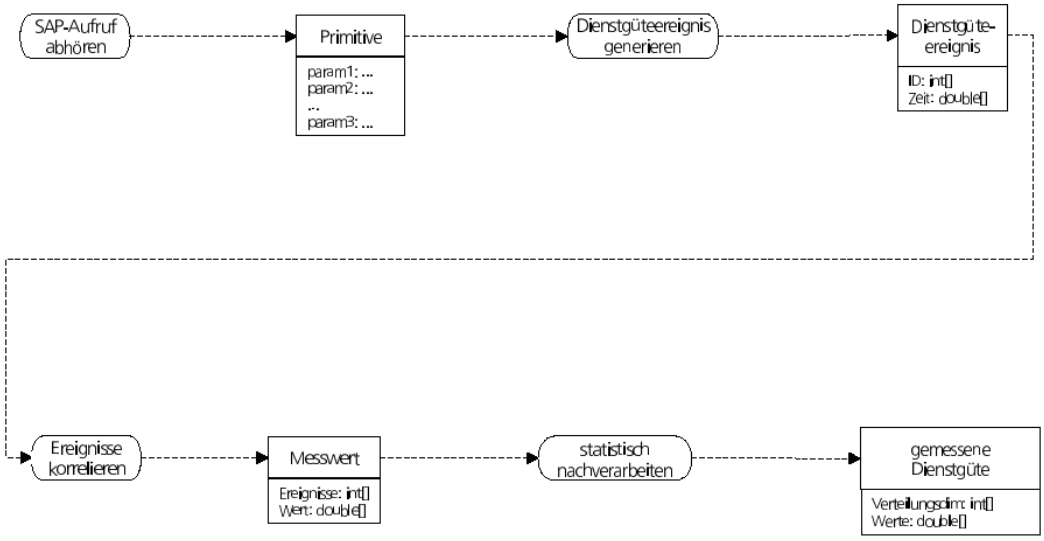

Abbildung 2.1: Darstellung des Dienstgüte-Meßprozesses als UML-Aktivitätsdiagramm; aus [Gars 04]

zur Bestimmung von Dienstgütemerkmalen zur Verfügung, dessen einzelne Komponenten mittels deklarativer Konstrukte angegeben werden. Diese sind hier in der Reihenfolge ihrer konzeptuellen Notwendigkeit nach Abbildung 2.1 angegeben:

1. Der Dienstzugangspunkt (Service Access Point, SAP) ist zunächst (symbolisch) anzugeben, da der Meßprozeß an eben dieser Stelle durch einen Methodenaufruf angestoßen wird.
2. Eine Deklaration von Primitiven, die die Dienstfunktionalität darstellen, dessen Parameter durch Attribute mit entsprechenden Namen und Typen modelliert werden.
3. Dienstgüte-Ereignisse werden generiert, wenn an einem bestimmten SAP ein Dienstaufruf mit bestimmten Parametern erfolgt. Diese Belegungen sind innerhalb einer Ereignisdeklaration anzugeben.
4. Meßwerte werden durch Korrelation von Ereignissen erzeugt, deren Berechnungsvorschrift innerhalb einer Meßwert-Deklaration frei definiert werden kann.
5. Zur Berechnung von Meßwerten soll auf eine Menge von Korrelationsfunktionen zurückgegriffen werden.

6. Die statistische Nachverarbeitung solcher Meßwerte geschieht über eine Aggregationsfunktion. Sie wird in einer gesonderten Deklaration spezifiziert.
7. Die Menge der Aggregationsklassen oder -funktionen ist nicht endgültig festgelegt.

2.2 Rolle des Übersetzers

Der Übersetzer erzeugt nun aus der mit o. g. Komponenten in der Sprache QoSSL spezifizierten Entwicklersicht die Meßsicht. Dabei verarbeitet er eine vom Anwender festgelegte Definition von Dienstgütemerkmalen zu einer Spezifikation der Definitionssicht in einer ausführbaren Programmiersprache. „Ausführbar“ bedeutet, daß sich aus dieser Spezifikation durch Kompilation und einige Anpassungen ein lauffähiges Meßsystem erzeugen läßt. Abbildung 2.2 zeigt das Verfahren auf. Voraussetzung für die Ablauffähigkeit dieses Meßsystems

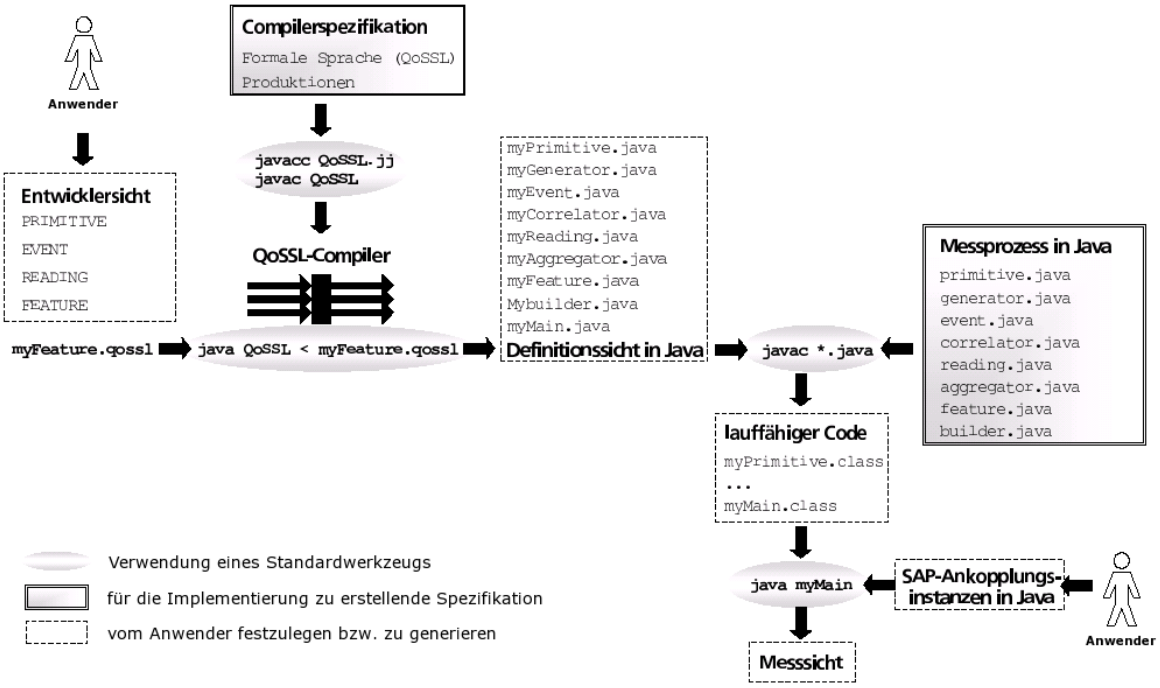

Abbildung 2.2: Darstellung des Automatisierungsprozesses aus [Gars 04]

ist das Vorhandensein einer Implementierung des generischen Meßprozesses, der derzeit ebenfalls die Korrelationsfunktionen beinhaltet, sowie rudimentär entwickelter Aggregationsklassen. Dieser allgemeine Meßprozeß wird mit allen Komponenten aus [Gars 04] übernommen, der in jener Arbeit prototypisch entwickelte Übersetzer hingegen von Grund auf neu geschrieben.

3 Anforderungen

Unabhängig von Vorgehensmodellen steht am Anfang der Entwicklung von Software der Erwerb und die Analyse von Anforderungen.[Wirs 03] Diese gliedern sich in einen konzeptionellen Teil, der sich aus der grundsätzlichen Problemstellung ergibt, und in einen technischen Teil, der sich aus den allgemeinen Anforderungen an einem Übersetzer ableiten läßt.
Aus der grundsätzlichen Problemstellung ergibt sich zunächst eine Reihe von konzeptionellen Anforderungen an das zu entwickelnde Programm:

- Die Syntax der Sprache QoSSL soll deklarative Konstrukte formulieren und von natürlichen Personen möglichst gut lesbar sein.
- Eine in dieser Sprache formulierte Spezifikation soll über mehrere Dateien verteilt werden können.
- Die Sprache soll auf ein erweiterbares Typsystem vorbereitet sein, um im Hinblick auf Attribute der Primitiven eine hohe Abstraktion zu ermöglichen.
- Das ursprüngliche Konzept aus [Gars 04] sieht eine (beispielhafte!) Implementierung in Java vor. Es erscheint zweckmäßig, die Programmiersprache beizubehalten.
- Fehler in der Spezifikation sollen dem Anwender der Sprache und des Compilers auf möglichst verständliche Art mitgeteilt werden. Dazu ist sicherzustellen, daß aus jeder durch das vorliegende Programm erzeugte Definitionssicht ohne Fehlermeldungen die Meßsicht generiert werden kann.
- Da derzeit keine endgültige Festlegung bezüglich der Zahl von Korrelationsfunktionen getroffen werden kann, erscheint eine flexible Handhabung unerläßlich.
- Gleiches gilt für die Anzahl der Aggregationsfunktionalitäten, da diese sogar von Anwendung zu Anwendung abweichen kann.
- Es ist wünschenswert, aus der Entwicklersicht heraus die Meßsicht nicht nur zu *spezifizieren*, sondern auch zu *dokumentieren*.

Daneben gibt es eine Reihe von technischen Anforderungen, die sich anschließend aus den grundsätzlichen Kriterien für den Aufbau von Übersetzern ableiten lassen:

- Die zu wählende Syntax muß für Maschinen möglichst einfach zu verarbeiten sein. Das schließt Eindeutigkeit und eine *linksrekursive Vorausschau von 1 von links* (`LL(1)`) mit ein.
- Eine Übersetzung durch einfache Textersetzungsregeln erscheint zu unflexibel und ist nur schwer wartbar. Daher ist ein Zwischenschritt bei der Kompilation zweckmäßig.
- Die Verteilung einer Spezifikation auf mehrere Dateien erfordern eine flexible Handhabung der Deklarationen. Insbesondere soll die Reihenfolge der Deklarationen keine Rolle spielen.
- Existieren in der gegebenen QoSSL-Spezifikation Fehler, so soll die Übersetzung abgebrochen werden, und zwar möglichst erst nach Ende des Parse-Vorgangs; dabei sollen alle gefundenen Fehler gemeldet werden.
- Eine dynamischen Suche nach Korrelationsfunktionen und Aggregationsklassen ist zu implementieren.
- Da in der Spezifikation nicht nur arithmetische Ausdrücke, sondern auch Anwendungen von Korrelationsfunktionen vorkommen dürfen, ist ihre Analyse mittels rekursivem Abstieg [Bry 04][USA 01] zur Übersetzungszeit erforderlich.

Schließlich geht es noch vor allem darum, Mängel des erwähnten Prototyps aus [Gars 04] zu beseitigen. An dieser Stelle seien einige genannt:

- Der prototypische Übersetzer hat eine *Vorausschau von 5 von links* und ist somit weit vom Ideal des LL(1)-Parsers entfernt, der eine Bearbeitungszeit von $\Theta(n)$ garantiert.
- Die bisherige Implementierung verfolgt den Ansatz eines „intelligenten Textersetzers“. Konkret wird beim Einlesen einer Deklaration nach und nach ein Zeichenkettenpuffer mit „Entsprechnungen“ gefüllt und am Ende in eine Datei ausgeschrieben. Dem zugrundeliegenden Ansatz, Sichten abzubilden, wird insofern nur wenig entsprochen.
- Ebenfalls notwendig ist eine stärkere Modularisierung. So ist es beispielsweise unzweckmäßig, das Layout des Zielcodes innerhalb der Parsing-Routinen für jede zu erzeugende Datei einzeln festzulegen.
- (Arithmetische) Ausdrücke sollten besser mittels rekursiven Abstiegs geparst werden, um ein hohes Maß an Skalierbarkeit zu erreichen.
- Bezeichner, wie etwa diejenigen von Korrelationsfunktionen, sind vielfach „hartkodiert“; der Parser beschränkt die Menge der zulässigen Werte innerhalb seines Programmcodes. Dies liegt im Widerspruch zu einer erweiterbaren Spezifikation, deren Sinn es nicht sein sollte, bei jeder Änderung den Übersetzer anpassen zu müssen.
- Generell ist eine gewisse Abstraktion des zugrundeliegenden Modells innerhalb des Programms erstrebenswert, um vom „Textersetzer“ zu einem Programm zu gelangen, das „in Zusammenhängen“ arbeitet. Dies erleichtert Wartung und Fehlerbehandlung, letztere speziell bei modellabhängigen semantischen Fehlern.

4 QoSSL-Syntax

An dieser Stelle wird die durch den Compiler festgelegte Syntax der Sprache QoSSL in ihren Grundzügen erläutert. Dazu werden die Produktionsregeln der formalen Grammatik kurz vorgestellt. Eine vollständige Auflistung aller Produktionsregeln befindet sich in der Dokumentation des Parsers [QoSP] und im Anhang A. Die Deklarationen der QoSSL-Komponenten sind an jene aus [Gars 04] angelehnt und werden im Hinblick auf die genannten Anforderungen verfeinert. Dazu gehört die Behebung kleinerer syntaktischer Mängel, aber auch die einheitliche Behandlung sämtlicher Vorkommen von Bezeichnern als `<IDENTIFIER>` und der Ansatz, Berechnungsausdrücke unter Einhaltung von Präzedenzregeln zu behandeln. Die Typprüfung geschieht nunmehr anhand des Kontexts. Weiterhin ist die Menge der gültigen Bezeichner definert durch die entsprechenden Regeln der Programmiersprache Java [GJSB 00] [Ulle 06].
Übernommen worden sind insbesondere die Schlüsselwörter und der Aufbau der Deklarationen.
An dieser Stelle folgt nun nochmals eine kurze Auflistung Produktionsregeln der formalen Grammatik für QoSSL-Deklarationen. Sie werden in EBNF [ISO 14977] [Scho 01] notiert.

1. Die Angabe der Instanz der SAP-Ankopplung erfolgt durch das Schlüsselwort `TAP`. Im Gegensatz zum Prototypen geschieht dies mit einer eigenen Deklaration; damit können dieselben SAPs mehrfach referenziert werden. Die Abkürzung TAP leitet sich von Test Access Point ab, bedeutet also soviel wie Testzugangspunkt.

```
<TAP_DECL> ::= "TAP" <IDENTIFIER> ";" .
```

2. Das Konstrukt zur Festlegung von Primitiven wird wie bisher durch das Wort `PRIMITIVE` eingeleitet. Die Attribute werden durch eine Liste von Typisierungsausdrücken der Form `<TYP><BEZEICHNER>` angegeben.

```
<PRIMITIVE_DECL>    ::= "PRIMITIVE" <IDENTIFIER>
                        "(" (<TYPE_CONSTR_LIST>)?")" ";" .
<TYPE_CONSTR_LIST>  ::= <TYPE_CONSTRAINT>
                        (","<TYPE_CONSTRAINT>)* .
<TYPE_CONSTRAINT>   ::= <TYPE><IDENTIFIER> .
```

3. Die Deklaration von Ereignissen beginnt mit `EVENT`. Nach dem Wort `USES` wird die Primitive referenziert, aus der sich diese Art von Ereignissen ableiten lassen. Die zugehörige Instanz der SAP-Ankopplung folgt nach `ON`, die Filtervorschrift wird nach `ISOLATE` angegeben. Die Filterregeln müssen dabei mit den Typisierungsausdrücken der oben referenzierten Primitive in Anzahl, Reihenfolge und Typ korrespondieren. Nach dem Wort `ID` wird der sogenannte Identifikatormodus angegeben, auf welche Weise der nach diesem Entwurf notwendige eindeutige Identifikator in dieser Ereignisklasse gebildet werden soll. Dies kann entweder der Name eines Attributs, ein aus mehreren Attributen gebildeter Hash-Wert (Streuspeicherwert) sein oder „automatisch“ geschehen, was durch die Angabe von `AUTO` spezifiziert wird.

```
<EVENT_DECL>        ::= "EVENT" <IDENTIFIER>
                        "USES" <IDENTIFIER>
                        "ON" <IDENTIFIER>
                        "ISOLATE" <FILTER_EXPN_LIST>
                        "ID" <IDENTIFICATE_BY>
                        ";" .
```

```
<FILTER_EXPN_LIST>  ::= "("(<FILTER_EXPRESSION>
                        (","<FILTER_EXPRESSION>)*)?")" .
<FILTER_EXPRESSION> ::= ("*"|<GENERIC_EXPRESSION>) .
<IDENTIFICATE_BY>   ::= (<IDENTIFIER>|"AUTO"|"HASH" <ARGUMENTS>) .
```

4. Das Konstrukt zur Vereinbarung von Meßwerten wird mit dem Schlüsselwort `READING` eingeleitet. Nach `REQUIRES` erfolgt die Einordnung der referenzierten Ereignisklassen in Warteschlangen. Dabei wird ein Ereignistyp `ert` einer Warteschlange `ws0` mit dem Konstrukt `ert AS QUEUE ws0` zugeordnet; die Zuordungen werden mit Kommata separiert. Nach `COMPUTES` folgt der Berechnungsausdruck für den Meßwert. Hinter `UNIT` wird mit einer beliebigen Zeichenkette die zu verwendende Maßeinheit festgelegt.

```
<READING_DECL>      ::= "READING" <IDENTIFIER>
                        "REQUIRES" <LOCAL_DECL_LIST>
                        "COMPUTES" <EXPRESSION>
                        "UNIT" <STRING_LITERAL>
                        ";" .
<LOCAL_DECL_LIST>   ::= "("<LOCAL_DECLARATION>
                        (","<LOGICAL_OR_EXPN>)*")" .
<LOCAL_DECLARATION> ::= <IDENTIFIER> "AS" "QUEUE" <IDENTIFIER> .
```

5. Die gemessene Dienstgüte wird mit der `FEATURE`-Deklaration spezifiziert. Sie ergibt sich aus der Zusammenfassung von Meßwerten, deren Anzahl nach `JOIN` mit einem Berechnungsausdruck und deren Referenz nach `OF` angegeben wird. Hinter dem Wort `TO` wird der Bezeichner der gewählten Aggregationsklasse eingefügt.

```
<FEATURE_DECL> ::= "FEATURE" <IDENTIFIER>
                   "JOIN" <EXPRESSION>
                   "OF" <IDENTIFIER>
                   "TO" <IDENTIFIER>
                   ";" .
```

Neben den Deklarationen werden in dieser Arbeit noch zwei weitere Sprachkonstrukte eingeführt:

1. Direktiven dienen der Steuerung der Übersetzung und bestehen aus einem Schlüsselwort gefolgt von einem Bezeichner. Sie enden *nicht* mit dem „`;`“-Zeichen. Die Menge der gültigen Bezeichner hängt von den einzelnen Direktiven ab. Sie müssen immer ganz am Anfang der Spezifikation stehen.

```
<DIRECTIVES>           ::= (<LANG_DIRECTIVE>
                         | <NAMESPACE_DIRECTIVE>
                       (* | ... *)
                         ) .
<LANG_DIRECTIVE>       ::= "LANG" <STRING_LITERAL> .
<NAMESPACE_DIRECTIVE>  ::= "NAMESPACE" <STRING_LITERAL> .
(* ... *)
```

2. Import-Anweisungen dienen zur Angabe von in der Spezifikation verwendeten, aber nicht in QoSSL eingebauten Typen. Sie beginnen mit dem Schlüsselwort `import`, gefolgt von einem Klassentyp beziehungsweise einem höheren Datentyp der Implementierungssprache der Eingabebibliothek.

```
<IMPORT> ::= "import" <CLASS_TYPE> (".*")? ";" .
```

Gegenüber dem diskutierten Prototypen haben sich noch das Typkonzept und das Verfahren zur Analyse von Berechnungsausdrücken wie folgt verändert:

1. Das Typkonzept folgt *syntaktisch* der Programmiersprache Java [GJSB 00] [Ulle 06]. Demensprechend erfolgt eine Einteilung in primitive, Referenz- und Klassentypen nach folgendem Schema. Hier wird jedoch nur die Syntax angegeben; eine ausführliche Diskussion des Typsystems folgt weiter unten im Kapitel 6.5.

```
<TYPE>           ::= (<REFERENCE_TYPE>|<PRIMITIVE_TYPE>) .
<REFERENCE_TYPE> ::= (<PRIMITIVE_TYPE>("["<LOGICAL_OR_EXPN>"]")+ |
                      <CLASS_TYPE>("["<LOGICAL_OR_EXPN>"]")*) .
<CLASS_TYPE>     ::= <IDENTIFIER>("."<IDENTIFIER>)* .
<PRIMITIVE_TYPE> ::= ("boolean"|
                      "byte"|"short"|"int"|"long"|
                      "float"|"double") .
```

2. Ausdrücke, wie sie z. B. zur Berechnung von Meßwerten vorgesehen sind, unterstützen sämtliche gängigen arithmetischen und logischen Operationen und werden nach den Regeln des *rekursiven Abstiegs* geparst. Die sich daraus ergebende Präzedenzhierarchie entspricht jener der Programmiersprache Java [GJSB 00] [Ulle 06].

```
<GENERIC_EXPRESSION> ::= <LOGICAL_OR_EXPN> .
<LOGICAL_OR_EXPN>    ::= <LOGICAL_AND_EXPN>("||"<LOGICAL_AND_EXPN>)* .
<LOGICAL_AND_EXPN>   ::= <INCL_OR_EXPN>("&&"<INCL_OR_EXPN>)* .
(* ... *)
```

Diese Änderungen erleichtern dem Anwender die Arbeit, weil er so bezogen auf Typen, Ausdrücke und Auswertung ein hohes Maß an Transparenz (im politischen Sinne, wohlgemerkt) vorfindet.

5 Implementierung des Übersetzers

5.1 Programmiersprache

Der Übersetzer für die durch obige Grammatik festgelegte Sprache QoSSL ist in der Programmiersprache *Java* [GJSB 00] [Ulle 06] implementiert. Dies geschieht aus drei Gründen:

- Das in [Gars 04] vorgelegte Konzept aus Entwickler-, Definitions- und Meßsicht sieht eine Ableitung des spezifischen Meßprozesses aus der generischen Implementierung durch Vererbung vor. Dies führt zu einer Entscheidung zugunsten einer objektorientierten Programmiersprache.
- Die diskutierte Anforderung, den Übersetzer mit einem gewissen Niveau an Abstraktion arbeiten zu lassen, bestärkt diese Entscheidung zusätzlich.
- Der generische Meßprozeß ist in Java implementiert und kann mit geringfügigen Anpassungen übernommen werden. Die Verwendung der gleichen Programmiersprache für den Übersetzer erleichtert die Arbeit des Programmierers und sorgt für eine weniger schwierige Wartung des Codes.

5.2 JavaCC

Der Parser wird wie bereits in [Gars 04] mit dem Parser-Generator *JavaCC* (*Java Compiler Compiler* [JCC] [JCCQ 04]) erstellt, da eine solche Implementierung händisch nur schwer zu bewältigen und zudem sehr fehleranfällig und kompliziert zu warten wäre.
Im Gegensatz zu anderen Parser-Generatoren wie etwa [YaCC] erzeugt JavaCC als Ausgabe ein Java-Programm und bietet zudem die Möglichkeit, die Produktionsregeln nahezu beliebig mit Java-Code zu ergänzen und Subroutinen hinzuzufügen.

5.3 Zwei-Lauf-Verfahren

Der Übersetzer arbeitet im sogenannten *Zwei-Lauf-Verfahren (Two-Pass)*. In einem ersten Lauf erfolgt eine komplette *Analyse der gegebenen QoSSL-Spezifikation*, wobei alle daraus zu gewinnenden Informationen in einem *Zwischencode* (siehe Kapitel 6.10) gespeichert werden, der als Objektsystem realisiert ist. Damit wird ein sehr hohes Maß an Abstraktion erreicht. Erst wenn diese Analyse ohne Fehler terminiert hat, kann mit der *Erzeugung der Definitionssicht* fortgefahren werden, wobei die gespeicherten Informationen wieder verwertet werden. Das scheint auf den ersten Blick umständlich, ist aber sehr zweckmäßig, wenn, wie in unserem Fall, eine Spezifikation auch auf Korrektheit bezüglich Umsetzung von Richtlinien geprüft werden soll. Die Vorgehensweise ist dann folgende: Zuerst versetzt die eingelesene Spezifikation das System in einen bestimmten Zustand, der durch den Zwischencode repräsentiert wird. Anschließend prüft das System, ob dieser Zustand erlaubt ist, also mit diesen Richtlinien vereinbar ist. Falls dem so ist, wird mit der Code-Erzeugung begonnen; andernfalls kann das Programm an dieser Stelle Fehler melden und die Übersetzung abbrechen.

5.4 Aufbau des Programms

Das Übersetzer-Programm ist nach gewissen Grundregeln für Systemarchitektur zusammengesetzt. Ziele sind dabei einerseits hohe Kohäsion, also das Zusammenfassen zusammengehöriger Teile zu Komponenten; an-

dererseits verlangt [Henn 03] eine geringe Kopplung, so daß möglichst wenige Abhängigkeiten zwischen den Komponenten entstehen. Zieht man nun Problemstellung und Anforderungen hinzu, ergibt sich folgendes Bild:

- Auf der einen Seite steht der Parser zusammen mit seinen Hilfsklassen. Letztere werden nach Funktionalität in drei Gruppen aufgeteilt: Analyse-Hilfsklassen, das noch zu besprechende Typsystem von QoSSL, sowie Klassen zur Repräsentation des Zwischencodes, dessen Notwendigkeit sich aus dem Zwei-Lauf-Verfahren ergibt.
- Wie bereits in [Gars 04], so stellt auch hier die Implementierung des generischen Meßprozesses eine zentrale Komponente dar.
- Ebenso bildet das erweitertete Modell eine eigene Komponente. Es handelt sich hierbei um die Gesamtheit des zu erzeugenden Codes.
- Die vierte Hauptkomponente ist die Bibliothek der Aggregationsklassen. Die Korrelationsfunktionen kommen dagegen im generischen Meßprozeß zu liegen.
- Schließlich bildet noch der Generator eine eigene Komponente. Damit ist die Code-Erzeugung auch architektonisch vom ersten Lauf der Übersetzung getrennt.

Das UML-Klassendiagramm [UML 01] [RJB 98] (Abbildung 5.1) stellt das vorliegende Gesamtsystem in seinen Grundzügen dar. Die Schnittstellen zwischen den einzelnen Komponenten sind jeweils nur so detailliert wie nötig angegeben.[1] Dabei ist auf der linken Seite der eigentliche Übersetzer zu sehen, rechts stehen die anderen diskutierten Komponenten und in der Mitte eine Reihe von Klassen, die als Schnittstelle zwischen den einzelnen Komponenten fungieren. Der Übersetzer ist nur mit seinen wichtigsten Teilkomponenten dargestellt. Der Parser liest die gegebene Spezifikation ein, untersucht sie unter Zuhilfenahme des Analysator-Pakets und des Typsystems und erzeugt den Zwischencode.
Der Zwischencode ist hier nur mit seinen wichtigsten drei Klassen präsent; sie seien an dieser Stelle kurz erläutert. Die Klasse `QoSSLTranslationUnit` bildet die Wurzel der Architektur des Zwischencodes; genau eine Instanz dieser Klasse repräsentiert den gesamten Zwischencode, der sich wiederum aus mehreren `QoSSLProcessUnit`-Objekten zusammensetzt. Jede dieser Instanzen steht gemäß ihrer Klassenzugehörigkeit für eine Komponente der gegebenen Spezifikation, also beispielsweise ein deklariertes Ereignis oder eine Dienstprimitive. Die Klasse `TranslationUnitProperties` kapselt gewisse Parameter, mit denen die Code-Erzeugung gesteuert werden kann.
Die mittlere Spalte zeigt eine Reihe von Klassen, die allesamt eine Schnittstellenfunktion zwischen den einzelnen Komponenten innehaben. Dies sind im einzelnen:

- Das Interface `Reflector` definiert die Signaturen der Methoden, die zum Aufspüren von Korrelationsfunktionen und Aggregationsfunktionalitäten dienen sollen. Die Implementierung soll innerhalb des Generator-Pakets erfolgen.
- Das Interface `Generator` trifft Festlegungen für diejenige Klasse des Generator-Pakets, die zur Code-Erzeugung dienen soll. Die Methode `generateTargetCode` soll die Code-Erzeugung anstoßen.
- Die Schnittstelle `BaseProperties` definiert *unter anderem* die Namen der Reflektor- und Generator-Implementierungen. Die Konstante `AGG_FOLDER` hält außerdem den Bezeichner des Unterpakets der Aggregationsbibliotheken. Diese Festlegungen sind notwendig, damit der Parser während des ersten Laufs sozusagen „blind“ nach dem Reflektor suchen und diesen instanzieren kann. Er läßt ihn anschließend nach verfügbaren Korrelationsfunktionen und Aggregationsbibliotheken suchen.

Auf der rechten Seite sind die vier anderen Hauptbestandteile zu sehen. Ganz oben ist die Aggregationsbibliothek eingezeichnet. Dieses Paket besteht in unserem Fall einfach aus der Menge von Klassen, die Aggregationsfunktionalitäten bereitstellen sollen. Sie entstehen durch Implementierung der ebenfalls im Paket enthaltenen Schnittstellenklasse (Interface) `AggregationRepository`. Repräsentativ sind hier `Sum` und `Boxplot` angegeben. Den Anforderungen entsprechend verfolgt das Design das Ziel, daß solche Klassen möglichst einfach hinzugefügt oder entfernt werden können. Auch der Prozeß der Übersetzung unterstützt dieses Merkmal, wie wir noch sehen werden. Der Name des Pakets leitet sich unter anderem aus der Konstante `AGG_FOLDER` der Hilfsklasse `BaseProperties` ab.

[1] Nicht UML-konform ist die Notation in { } hinter den Bezeichnern der einzelnen Komponenten; dort steht zur Orientierung der Name des jeweiligen Unterpakets.

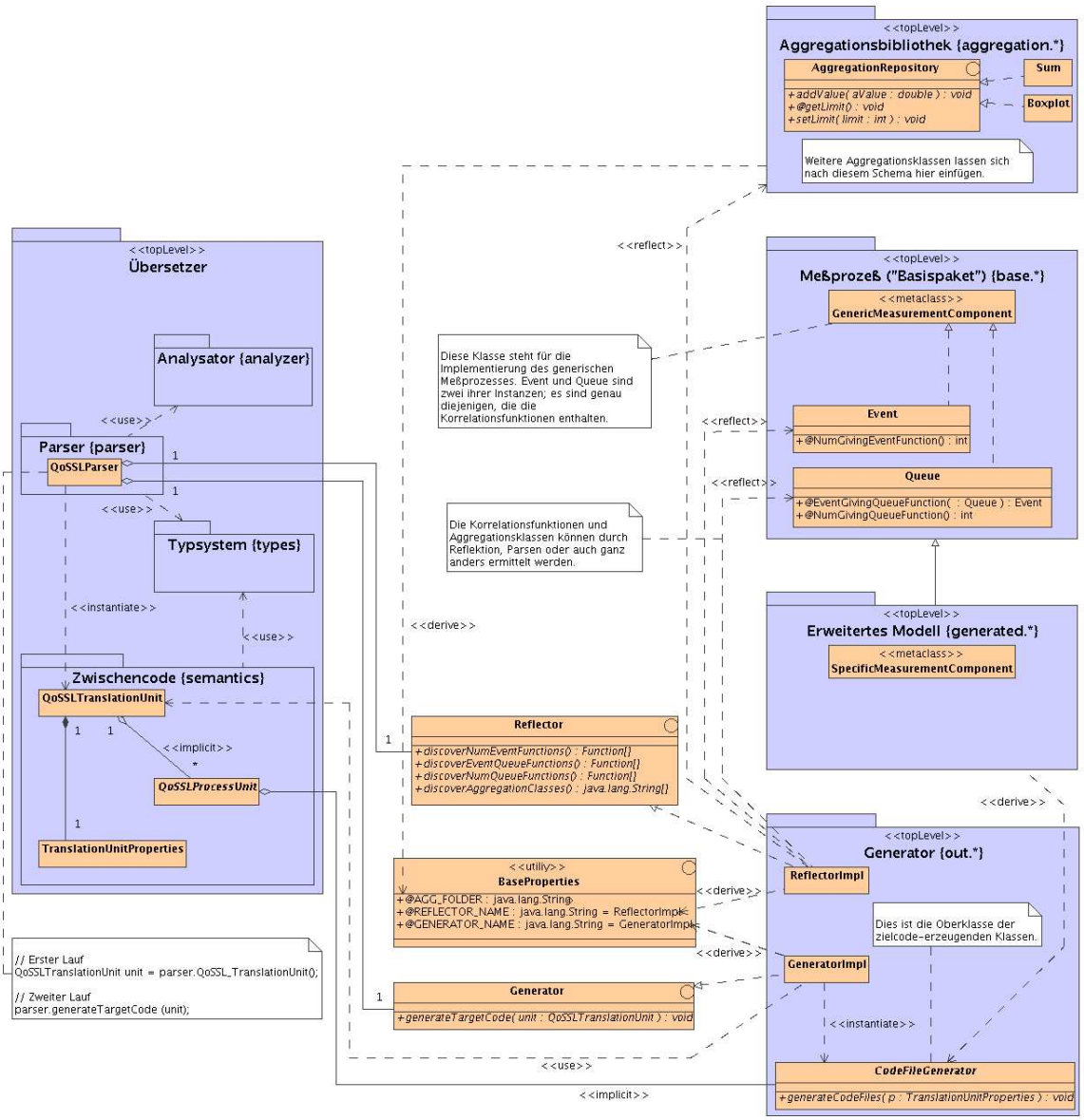

Abbildung 5.1: Aufbau des Compiler-Programms

Darunter ist der *Allgemeine Meßprozeß* nach [Gars 04] eingezeichnet. Die Implementierung ist hier auf die Metaklasse `GenericMeasurementComponent` reduziert worden; sie steht für einen Satz von hier nicht einzeln aufgeführten Klassen und ihren Beziehungen zueinander. Die beiden für diesen Abschnitt wichtigen Elemente sind `Event` und `Queue`, die wie aus [Gars 04] bekannt, die Korrelationsfunktionen enthalten. Auch sie sind insoweit austauschbar, als die beiden Klassen durch eine andere Implementierung ersetzt werden können.
Als drittes ist das *Erweiterte Modell des Allgemeinen Meßprozesses* eingezeichnet. Dieses ist zu Beginn noch gar nicht vorhanden; es entsteht ja erst während der Übersetzung. Analog zu oben ist seine Implementierung auf eine Metaklasse reduziert; wichtig an dieser Stelle ist die Generalisierungs-Assoziation zum allgemeinen Modell. Sie drückt aus, daß das spezifische Modell eines Meßprozesses eine Spezialisierung des Allgemeinen Meßprozesses darstellt. Der Pfeil nach unten weist auf die Herkunft dieser Komponente als Ergebnis einer Code-Erzeugung durch den Generator rechts unten hin.
Dieses Generator-Paket ist für Reflektion und Code-Erzeugung zuständig. Es beinhaltet dreierlei Klassen:

- `CodeFileGenerator` ist die abstrakte Superklasse aller Klassen, die ausgehend von einer Instanz der Klasse `QoSSLTranslationUnit` Code der Definitionssicht erzeugen sollen. Sie ist als Analogon zu `QoSSLProcessUnit` zu verstehen; jede ihrer Unterklassen gibt den zu einer bestimmten Komponente des Zwischencodes gehörenden Teil der Definitionssicht aus.
- Die Klasse `ReflectorImpl` implementiert das *Reflektor-Interface* derart, daß die geforderten Methoden in der Lage sind, die Aggregationsklassen in der Bibliothek zu finden und die Signaturen der Korrelationsfunktionen aus dem allgemeinen Meßprozeß zu extrahieren. Sie werden als Zeichenketten bzw. `Function`-Objekte zurückgegeben.
- Die Klasse `GeneratorImpl` implementiert das *Generator-Interface*. Die darin enthaltene Methode `generateTargetCode` erhält den Zwischencode als Aufrufparameter und erzeugt in Abhängigkeit von dessen konkreter Gestalt den Zielcode der Definitionssicht, der in eine Reihe von Dateien ausgeschrieben wird.

Die Unterteilung des Programms in mehrere Komponenten hat nicht nur den Zweck, die Systemarchitektur übersichtlich zu gestalten, sondern auch, um eine hohe Flexibilität zu erreichen. Das Design ist so ausgelegt, daß Meßprozeß, Aggregationsbibliothek und Code-Erzeuger ausgetauscht werden können, ohne den eigentlichen Übersetzer verändern zu müssen. Dabei unterliegen diese Komponenten gewissen Vorschriften:

- Die Implementierung des Meßprozesses darf beliebig gestaltet sein.
- Gleiches gilt für die Aggregationsbibliotheken.
- Das Generator-Modul ist in Java zu implementieren und hat mindestens die Klassen `ReflectorImpl` und `GeneratorImpl` bereitzustellen. Sie müssen mit den anderen bereitzustellenden Komponenten geeignet umgehen können, um ein Ergebnis im Sinne der Aufgabenstellung und der Spezifikationsrichtlinien zu erzielen.

Insgesamt kommt dieses Software-Design der erhobenen Forderung nach Unabhängigkeit von der im Meßprozeß eingesetzten Technologie nach.

5.5 Symboltabelle

Wie alle Übersetzer besitzt auch dieser eine Symboltabelle, um Namen und Typen verwalten zu können. Der Mechanismus setzt sich aus diesen drei Klassen zusammen, die allesamt in der Analysator-Komponente liegen:

- Instanzen der Java-Klasse `Symbol` repräsentieren jeweils einen Bezeichner. Sie speichern ihren Namen, den Typ und bei Spezialtypen das Referenzobjekt, also den jeweiligen semantischen Repräsentanten. Außerdem enthalten sie Informationen darüber, ob dieser Bezeichner bereits vereinbart worden ist oder nicht.
- Die Klasse `SymbolTable` im selben Paket realisiert die einschlägigen Anforderungen an eine Symboltabelle, die sich von Seiten der Literatur ergeben.[USA 01] Dies sind Operationen zum Einfügen, Auffinden und zur Rückgabe von Symbolen.

- Die Klasse `StaticSymbolTable` stellt eine Erweiterung der Klasse `SymbolTable` dar und ist die tatsächlich verwendete Implementierung einer Symboltabelle. Die zusätzlichen Operationen dienen unter anderem zur Markierung der Vereinbarung von Bezeichnern.

Da die vorliegende Implementierung Vorwärtsreferenzierungen unterstützt, wird zwischen dem Einfügen von Namen und ihrer Kennzeichnung als vereinbart unterschieden. Vereinbart bedeutet, daß die Eigenschaften dieser Instanz des Typs festgelegt worden sind; im Gegensatz dazu existieren Entitäten, bei denen nur Typ und Bezeichner bekannt sind. So bleibt erkennbar, ob ein Bezeichner nur verwendet oder tatsächlich auch vereinbart worden ist.

6 Funktionsweise, Ablauf der Übersetzung

6.1 Quellprogramm

Wir beginnen nun mit einer detaillierten Beschreibung des Übersetzungsvorgangs. Dies geschieht sowohl ganz allgemein als auch anhand einer konkreten Beispielspezifikation. Wir beginnen mit einer in QoSSL niedergeschriebenen Entwicklersicht und erhalten am Ende den gesamten Code der Definitionssicht.
Zu Beginn des Übersetzungsvorgangs steht eine in QoSSL formulierte Dienstgüte-Spezifikation. Das vorliegende Programm erlaubt es, diese auf mehreren Dateien zu verteilen. Der Anwender kann seine gesamte Spezifikation in einer Datei ablegen, oder einzelne Teile davon in andere Dateien auslagern und über `#include`-Makros referenzieren. Auch kann die Hauptdatei ausschließlich aus solchen Zeilen bestehen; auch darf eine so referenzierte Datei wiederum `#include`-Makros enthalten.
Als Beispiel sei eine Spezifikation für die Messung des Datendurchsatzes an einem TCP-Socket gegeben. Diese ist dabei auf zwei Dateien `throughput.qossl` und `throughput_body.qossl` derart aufgeteilt, daß erstere nur die Direktiven und eine einzige Vereinbarung enthält und mittels `#include`-Makro die zweite Datei referenziert:

```
/* Anfang der Datei throughput.qossl */

LANG "java"

TAP socket_test;

#include "throughput_body.qossl"

/**
    Unser Messwert ergibt sich einfach aus der Anzahl an Socket-Aufrufen pro
    Zeiteinheit.
*/
READING data_transfer
    REQUIRES (
            any_data AS QUEUE trans
    )
    COMPUTES trans.getSize()
    UNIT "Aufrufe";

/* Ende der Datei throughput.qossl */
```

Die zweite Datei enthält in diesem Fall fast die gesamte Spezifikation. Der Übersetzer wird sie vor dem Parsen an eben die Stelle der `#include`-Zeile von `throughput.qossl` einfügen.
Das Schlüsselwort `#define` kennzeichnet einen weiteren Typ von Makros, die zur *Makro-Expansion* dienen. Damit kann der Anwender Zeichenkettenkonstanten definieren, deren Name von dieser Stelle an überall im Quelltext vor der Übersetzung durch die gegebene Folge von Zeichen ersetzt wird. Dies ist insbesondere dann nützlich, wenn eine Datei für mehrere Spezifikationen benutzt und mittels `#include` eingebunden werden soll, jedoch an einigen Stellen jeweils verschiedene Bezeichner zu verwenden sind.

```
/* Anfang der Datei throughput_body.qossl */

#define MY_TAP socket_test
```

```
#define MY_READING data_transfer

/**
    Stellt den Aufruf eines Sockets dar.
    Ein Socket kennt Quell- und Ziel-IP (src und dest),
    Quell- und Zielport (src_port und dest_port) sowie Nutzlast (data).
*/
PRIMITIVE socket_data (
    byte[4] src,
    byte[4] dest,
    short src_port,
    short dest_port,
    byte[] data
);

/**
    Uns interessieren dabei sämtliche Daten, die über diesen Socket laufen.
    Wir geben unter ISOLATE also nur *s an.
*/
EVENT any_data
    USES socket_data
    ON MY_TAP
    ISOLATE (*,*,*,*,*)
    ID      AUTO;

/**
    Die Übertragungsrate ergibt sich direkt aus dem Messwert data_transfer.
*/
FEATURE data_transfer_rate
    JOIN 1 OF MY_READING TO Sum;

/* Ende der Datei throughput_body.qossl */
```

Das Konzept der Makros rührt von der Programmiersprache *C* her. Dieser Mechanismus wird von dort übernommen, weil die Aufgabe, vor der eigentlichen Übersetzung mehrere Quelldateien zu einer zusammenzufassen oder Zeichenkettenkonstanten zu definieren, praktisch die gleiche ist. Ebenso läßt sich dieser Mechanismus zum *Bedingten Kompilieren* verwenden. Eine anschauliche Diskussion über solche Makros gibt es in [KeRi 90].
Zur Bearbeitung der Makros wird in der vorliegenden Implementierung der *GCC-Präprozessor* [GCC 03] eingesetzt. Er wird vom Übersetzer selbsttätig mit folgenden Optionen aufgerufen:

```
-P          #xxx-Zeilenmarkierungen nicht erzeugen
-ftabstop=4 Tabulatorbreite auf 4 einstellen
-C          Kommentare durchreichen ohne Makros
-E          nur Praeprozessor benutzen, nicht kompilieren
-x c        Eingabe als C-Code behandeln, obwohl Dateiname nicht *.c ist
-o <datei>  Ausgabedatei
```

Im Grunde genommen verhalten sich diese Makros so, wie man es vom C-Präprozessor her gewohnt ist. Das genaue Verhalten der Makros hängt jedoch vom lokalen System ab.

6.2 Zusammengesetztes Quellprogramm

Bevor der Übersetzer mit dem Einlesen der Spezifikation beginnt, werden alle diskutierten Makros abgearbeitet. Dabei entsteht als Ergebnis im selben Verzeichnis eine neue Datei `throughput.qossl.ppc`, wobei

der GCC-Präprozessor die referenzierte Datei an Stelle der `#include`-Zeile eingefügt und die mit `#define` spezifizierten Makros expandiert hat.

```
/* Anfang der Datei throughput.qossl */

LANG "java"

TAP socket_test;

/* Anfang der Datei throughput_body.qossl */

/**
    Stellt den Aufruf eines Sockets dar.
    Ein Socket kennt Quell- und Ziel-IP (src und dest),
    Quell- und Zielport (src_port und dest_port) sowie Nutzlast (data).
*/
PRIMITIVE socket_data (
    byte[4] src,
    byte[4] dest,
    short src_port,
    short dest_port,
    byte[] data
);

/**
    Uns interessieren dabei sämtliche Daten, die über diesen Socket laufen.
    Wir geben unter ISOLATE also nur *s an.
*/
EVENT any_data
    USES socket_data
    ON socket_test
    ISOLATE (*,*,*,*,*)
    ID AUTO;

/**
    Die Übertragungsrate ergibt sich direkt aus dem Messwert data_transfer.
*/
FEATURE data_transfer_rate
    JOIN 1 OF data_transfer TO Sum;

/* Ende der Datei throughput_body.qossl */

/**
    Unser Messwert ergibt sich einfach aus der Anzahl an Socket-Aufrufen pro
    Zeiteinheit.
*/
READING data_transfer
    REQUIRES (
            any_data AS QUEUE trans
    )
    COMPUTES trans.getSize()
    UNIT "Aufrufe";

/* Ende der Datei throughput.qossl */
```

Es ist zu bemerken, daß die letzten beiden Deklarationen nicht in der Reihenfolge ihres konzeptionellen Zusammenhangs in der endgültigen Spezifikation auftauchen. Um den Anwender nicht dazu zu zwingen, seine Vereinbarungen im Vorfeld umsortieren zu müssen, stellt der Übersetzer für diesen Fall einen speziellen Mechanismus zur Verfügung, der weiter unten noch ausführlich besprochen wird.

6.3 Direktiven

Direktiven dienen im Gegensatz zu den Vereinbarungen *nicht* dazu, die Definitionssicht aus der formal spezifizierten Entwicklersicht zu generieren, sondern den Übersetzungsprozeß zu steuern. Sie sind also keine Parameter des Meßprozesses.
Ein weiterer Unterschied zu Deklarationen ist die Tatsache, daß jede Art von Direktive *genau einmal* vorkommen soll. Fehlt eine Direktive, so wird der jeweilige Standardwert angenommen und gesetzt; kommt eine mehrfach vor, so wird der im *ersten* Vorkommen spezifizierte beibehalten.
Bisher sind zwei Direktiven implementiert:

- Die *Namensraum-Direktive* wird durch das Schlüsselwort `NAMESPACE` eingeleitet. Sie gibt den relativen Bezeichner (sogenannter *Basisname* oder engl. *Basename*) desjenigen Unterordner des Verzeichnisses `./de/lmu/ifi/nm/qossl/` an, in welchem die erzeugte Definitionssicht abgelegt werden soll. Der Standardwert für diese Direktive ist „`generated`“. Die Verzeichnisstruktur wird in Anhang B erläutert.
- Ebenso wird die *Language-Direktive* durch das Schlüsselwort `LANG` eingeleitet. Sie gibt den Basisnamen desjenigen Unterordner des Verzeichnisses `./de/lmu/ifi/nm/qossl/base/` an, in welchem der zugrundeliegende generischer Meßprozeß abgelegt ist. Dieser Bezeichner muß mit dem einen Basisnamen des Ordners für Code-Erzeuger unterhalb von `./de/lmu/ifi/nm/qossl/out/` und Aggregationsbibliotheken unterhalb von `./de/lmu/ifi/nm/qossl/aggregation/` korrespondieren. Der Standardwert für diese Direktive ist „`java`“.

In unserem Beispiel ist also nicht nur explizit `LANG` „`java`“ gesetzt, sondern auch implizit `NAMESPACE` „`generated`“.

6.4 Kommentare

QoSSL stellt zweierlei Kommentare bereit:

- *Formale Kommentare* werden mit der Zeichenfolge `/**` eingeleitet und enden mit `*/`. Sie werden vom Parser eingelesen, automatisch der darauffolgenden Vereinbarung zugeordnet und an entsprechender Stelle im Zwischencode abgespeichert. Der Kommentartext kann demzufolge bei der Erzeugung des Zielcodes verwendet werden.
- *Normale Kommentare* beginnen mit `/*`, enden mit `*/` und sind zudem *schachtelbar*. Sie werden vom Parser ignoriert.

Die vorliegende Java-Implementierung fügt die eingelesenen formalen Kommentare in die zu erzeugenden Verfeinerungen der Datenübertragungsklassen als *formale Java-Kommentare* („Dokumentationskommentare“, `/** ... */`) ein. Diese Kommentare können so als Grundlage für eine mit dem Werkzeug `Javadoc` erstellte Programmdokumentation dienen. Damit kann innerhalb einer QoSSL-Spezifikation nicht nur die Spezifikation, sondern auch die *Dokumentation der Meßsicht* aus der Entwicklersicht heraus geschehen.
Formale Kommentare verhalten sich standardmäßig wie folgt:

- Einer Deklaration `D` wird immer derjenige formale Kommentar `F` zugeordnet, der unmittelbar vor `D` erscheint. Liegt zwischen `F` und `D` ein weiterer formaler Kommentar `F'`, so wird `F'` diesem `D` zugeordnet und `F` verworfen.
- Existiert zu einer Deklaration `D` kein formaler Kommentar `F` in diesem Sinne, so wird er als *leer* angesehen; es gilt dann also `$F == /** */`.

6.5 Typsystem

Soll eine Programmier- oder Spezifikationssprache mit verschiedenen Sorten von Daten umgehen, so bietet sich die Einführung von Datentypen an. Damit lassen sich solche Daten sinnvoll gruppieren und erlaubte Ope-

rationen zu ihrer Bearbeitung anbieten.[GuSo 00] [Bry 02] Bildet man die zuvor diskutierten Komponenten auf Typen und die einzelnen Deklarationen auf Operationen ab, so erhält man eine solide Basis zur Erkennung von Fehlern und zur Beherrschung der Materie überhaupt.
In der ursprüglichen Ausführung fungiert der Übersetzer lediglich als Textersetzer; Datentypen werden dort auf QoSSL-Ebene gar nicht erst eingeführt. Somit kann bei der Erzeugung der Definitionssicht aus der Entwicklersicht nur schwer festgestellt werden, ob die einzelnen Teile der vorliegenden Spezifikation überhaupt zueinander passen. Selbst die Einführung von Hilfsvariablen hätte immer noch eine sehr schlechte Skalierbarkeit zur Folge; der Übersetzer könnte immer nur eine Spezifikation auf einmal bearbeiten.
Betrachten wir zur Motivation nochmals folgende Deklaration:

```
FEATURE data_transfer_rate
JOIN 1 OF data_transfer TO Sum;
```

In diesen Zeilen lassen sich zwischen den Schlüsselwörtern intuitiv vier Ausdrücke mit entsprechend verschiedenen Typen erkennen:

- `data_transfer_rate` wird als Bezeichner für gemessene Dienstgüte eingeführt,
- der Berechnungsausdruck `1` ergibt trivialerweise eine Zahl,
- `data_transfer` muß zwingend einen Meßwert darstellen,
- und ebenso ist `Sum` der Name einer Aggregationsklasse.

Eine Zuordnung von Ausdrücken und insbesondere von Bezeichnern auf Mengen von Werten vereinfacht ihre Handhabung, da sie jeweils einfach in die passende „Schublade“ eingeordnet werden müssen, um bei Bedarf wiederum aus ihr hervorgeholt zu werden. Hier würde etwa der Name `data_transfer_rate` der Menge der Bezeichner für gemessenen Dienstgüte hinzugefügt. Ebenso kann der Übersetzer die möglicherweise von Meßprozeß zu Meßprozeß unterschiedliche Menge der verfügbaren Aggregationsklassen vorhalten und beim Einlesen dieser Deklaration mit dem Bezeichner `Sum` abgleichen. Genauso haben wir im Fall von `data_transfer` den Vorteil, daß der Parser nur diesen Abgleich machen muß, sich aber nicht zu „erinnern“ braucht, *woher* dieser Bezeichner kommt. Wir vereinfachen also den Programmcode des Übersetzers.
Weiterhin kann mit einem Typsystem neben dem Syntaxfehler eine weitere Art der Übersetzungszeitfehler zuverlässig erkannt werden, nämlich der Typfehler. Dies bedeutet, daß der für einen vorgefundenen Bezeichner ermittelte Typ nicht mit dem geforderten übereinstimmt bzw. in unerlaubtem Kontext vorkommt. Der in [Gars 04] entwickelte Prototyp würde solche Fehler in die Definitionssicht passieren lassen; sie würden dann entweder bei der Übersetzung in die Meßsicht oder gar erst bei deren Einsatz entdeckt. Im ersten Fall wären die Fehlermeldungen des (Java-)Compilers für die Behebung durch den Anwender wenig hilfreich, da diese sich ja auf die hier erzeugte Definitionssicht beziehen. Im zweiten Fall kann gerade bei komplexeren Spezifikationen die Fehlersuche extrem zeit- und arbeitsaufwendig werden.
Im Zusammenhang mit der Neuimplementierung des Übersetzers wird QoSSL also mit einem Typsystem ausgestattet, das für die einzelnen *Komponenten* Typen bereitstellt. Da außerdem Berechnungsausdrücke auf Korrektheit hin untersuchen werden sollen, sind *Zahlen* und mit ihnen *Verbunde* und *Wahrheitswerte* aufgenommen worden. Die Realisierung erfolgt auf Basis von Java-Klassen, wobei jedem Typ eine bestimmte Klasse zugeordnet wird.
Das Typsystem der Sprache QoSSL ist hierarchisch strukturiert. Die Struktur ergibt sich implizit aus der Hierarchie der Klassen im Paket `de.lmu.ifi.nm.qossl.types` [QoSH]. Sie ist in Abbildung 6.1 dargestellt. Die abstrakte Klasse `QoSSLType` bildet zunächst einen *abstrakten Obertyp*. Er ist selbst aber nicht als Typ im eigentlichen Sinne zu verstehen, sondern legt nur das *Verhalten* der QoSSL-Datentypen fest. Hier sind vor allem Operationen zur Typprüfung definiert.
Der größte ‚echte“ Typ heißt `QAny`; er ist der generische Obertyp. Darunter liegen einerseits die Spezialtypen, angeführt durch deren abstrakten Obertyp `QSpecial` und andererseits die „generischen“ Typen. Die Spezialtypen repräsentieren die Typen mit einer semantischen Repräsentation, also diejenigen Typen, die den konzeptionellen Elementen von QoSSL entsprechen und auf die Klassen des Objektsystems im Zwischencode abgebildet werden können. Deren gemeinsames Verhalten ist in der Klasse `QSpecial` festgelegt.
Die generischen Typen gliedern sich in die Verbundtypen, die deklarierten und die einfachen Typen. Die zugrundeliegende Typstruktur orientiert sich an jenen moderner Programmiersprachen.[ECMA-334] Die Zahlentypen sind anhand ihrer Genauigkeit hierarchisiert, wobei der Typ, der für die höchste Genauigkeit steht,

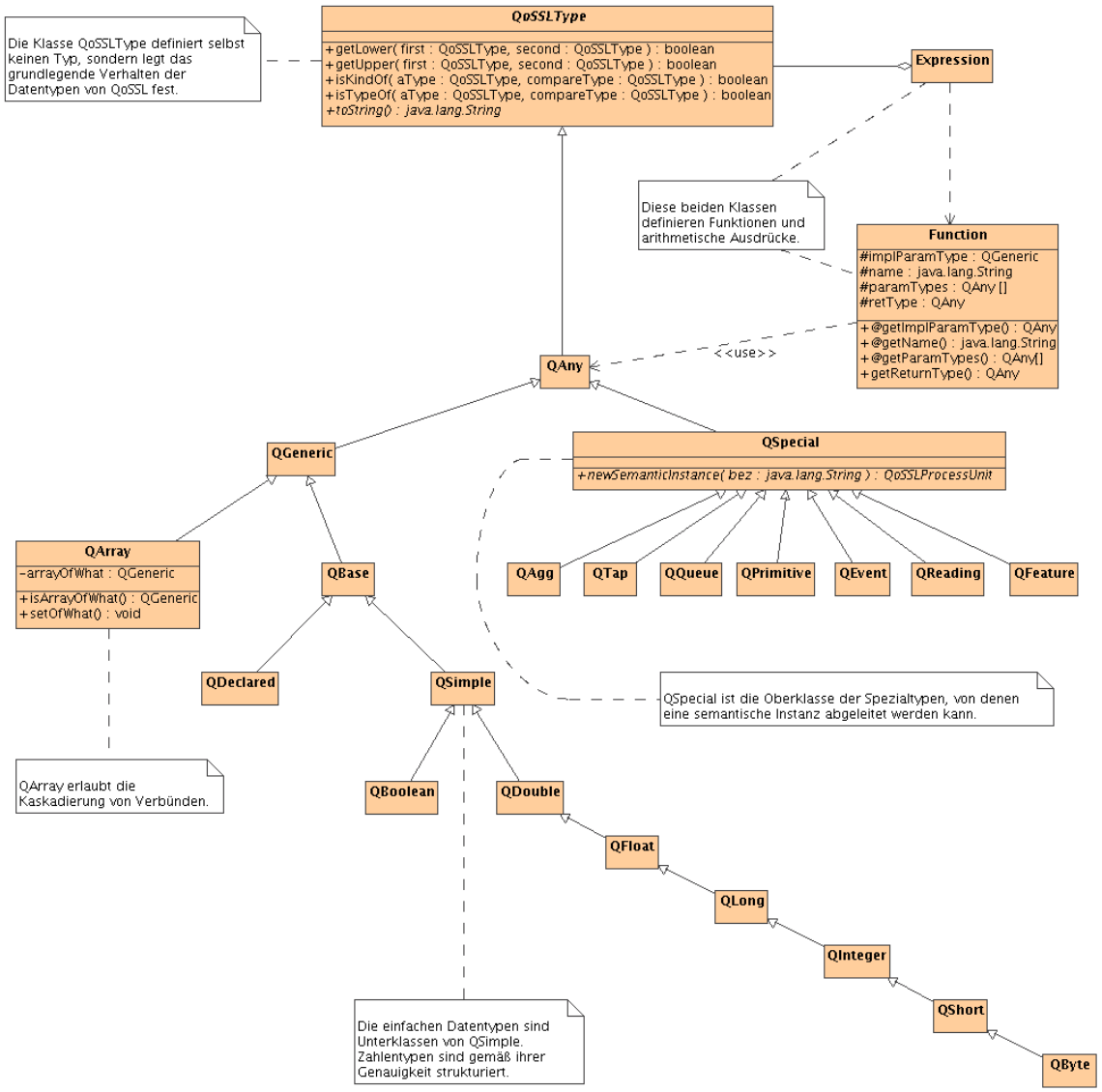

Abbildung 6.1: Das Typsystem der eingebauten Typen von QoSSL

der größte dieser Typen ist. Die Verbundtypen sind schließlich so angeordnet, daß sie kaskadiert („mehrdimensionale Verbunde“) gebildet werden können.
Die bisher diskutierten Typen stellen die sogenannten eingebauten Typen dar. Die Einordnung weiterer, nicht eingebauter Typen ist unterhalb des Typs `QDeclared` vorgesehen, welche somit die Schnittstelle zur Erweiterung des Typsystems bildet. Die Bekanntmachung solcher Typen geschieht wie bereits erwähnt mittels `import`-Anweisungen.

6.6 Ermittlung der Korrelationsfunktionen und Aggregationsklassen

Im nächsten Schritt der Übersetzung werden im durch die *Language-Direktive* angegebenen Unterpaket die Korrelationsfunktionen und Aggregationsklassen gesucht. Dazu wird zunächst die dort enthaltene Implementierung des Interfaces `de.lmu.ifi.nm.qossl.base.Reflector` mittels *Reflektion* (zuständige Me-

thode `Class.forName(java.lang.String)`) instanziert.[Ulle 06] Der Name dieser Klasse ist in der Konstante `REFLECTOR_NAME` in `BaseProperties` für alle Basispakete verbindlich als `ReflectorImpl` festgelegt. Durch den anschließenden Aufruf der in der *Reflektor-Schnittstelle* spezifizierten Methoden erhält der Übersetzer zweierlei Informationen.
Sei $L der Wert der Language-Direktive, dann gilt:

- Die Korrelationsfunktionen, in [Gars 04] anhand ihrer Signatur unterschieden und daher namentlich als *NumGivingQueueFunctions*, *EventGivingQueueFunctions* und *NumGivingEventFunctions* bezeichnet, werden im Paket `de.lmu.ifi.nm.qossl.base.$L` gesucht und als Instanzen der Klasse `de.lmu.ifi.nm.qossl.types.Function` abstrahiert; die jeweilige Signatur wird in den zugehörigen Attributen gespeichert.
- Die Aggregationsklassen liegen im Paket `de.lmu.ifi.nm.qossl.aggregation.$L`. Da Aggregationen nur anhand ihres Namens identifiziert werden, findet (bisher) keine weitere Abstraktion statt. Sie werden demzufolge als Objekte vom Typ `java.lang.String` gespeichert.

Diese Abstraktionen werden in der zu dieser Übersetzungseinheit (`QoSSLTranslationUnit`) gehörenden Instanz der Klasse `TranslationUnitProperties` abgespeichert. Damit stehen diese Informationen für den weiteren Übersetzungsprozeß zur Verfügung.
Die Bezeichner der Funktionen werden zudem in der Symboltabelle eingetragen; ihr Typ ist der jeweilige Ergebnistyp. Die Bezeichner der Aggregationsklassen werden ebenfalls eingetragen; sie erhalten den QoSSL-Typ `QAgg`. An dieser Stelle sei daran erinnert, daß ein Typ eine Menge von Werten ist.[Bry 02]
In der vorliegenden Beispielimplementierung aus Diagramm 5.1 werden durch den beschriebenen Vorgang `Sum` und `Boxplot` als Aggregationsklassen identifiziert. Die Korrelationsfunktionen sind vollständig aus [Gars 04] übernommen; ihre Signaturen lauten:

- `<QQueue>::getSize() => <QInteger>`
- `<QQueue>::myMatch(<QQueue>) => <QEvent>`
- `<QQueue>::hisMatch(<QQueue>) => <QEvent>`
- `<QEvent>::getId() => <QInteger>`
- `<QEvent>::getTime() => <QInteger>`

Dabei sind `QEvent`, `QQueue` und `QInteger` gemäß 6.5 die Typen von QoSSL für Ereignisse, deren Warteschlangen und für natürliche Zahlen.
Die tatsächliche Implementierung des Reflektor-Interface für die Suche nach diesen Informationen ist für jeden Satz von Basis- und Aggregationspaketen spezifisch vorzunehmen. Daher ist wie unter Kapitel 5.4 besprochen eine individuelle Ausgestaltung dieser Pakete möglich, ohne daß eine Änderung am eigentlichen Übersetzer notwendig wäre.

6.7 Parsen von Deklarationen

Zu Beginn des Parse-Vorgangs wird *genau eine* Instanz der Klasse `de.lmu.ifi.nm.qossl.semantics.QoSSLTranslationUnit` angelegt. Sie repräsentiert die Wurzel der zu übersetzenden Informationen. Dieses Objekt hält als Attribut zunächst eine Instanz der Klasse `TranslationUnitProperties`, die in ihren Attributen wiederum u. a. über Direktiven gesetzten Werte hält.
Die Syntax von QoSSL definiert sich durch die in der Grammatik spezifizierten Produktionsregeln. Der Parser geht nun den „umgekehrten“ Weg, indem er aus den angegebenen Deklarationen die notwendigen Informationen gewinnt. Das Programm erkennt die Art der Deklaration am jeweils einleitenden Schlüsselwort; erfolgreich eingelesen löst sie eine Reihe von Aktionen aus, die in diesem Kapitel besprochen werden.
Der Parser abstrahiert jede Deklaration aus dem QoSSL-Quellcode als solche, indem er eine Abbildung auf ihren jeweiligen semantischen Repräsentanten vornimmt. Dies entspricht jeweils genau einer Instanzierung der passenden Unterklasse von `QoSSLProcessUnit`, die von genanntem `QoSSLTranslationUnit`-Objekt direkt erreichbar ist. Assoziationen unter den referenzierten `QoSSLProcessUnit`-Objekten werden nach Bedarf gesetzt, d. h. wenn die eingelesene Spezifikation einen entsprechenden Zusammenhang festlegt.

Heißt es innerhalb einer Ereignis-Deklaration etwa `USES socket_data` wie in unserem Beispiel, so bedeutet dies eine Verbindung vom semantischen Repräsentanten der hier deklarierten Ereignisklasse zur Primitive `socket_data`.
In diesem Abschnitt gehen wir die oben angegebene Beispiel-Spezifikation Vereinbarung für Vereinbarung durch; das beschriebene Verhalten ergibt sich sinngemäß für eine beliebige *fehlerfreie* QoSSL-Spezifikation.

1. Deklaration der Instanz der SAP-Ankopplung:

```
TAP socket_test;
```

   Der Parser erkennt diese Vereinbarung am Schlüsselwort `TAP` und erhält als Information den (symbolischen) Bezeichner `socket_test`. Er sorgt dafür, daß nach Abarbeitung dieser Vereinbarung

   - ein Symbol mit dem Bezeichner `socket_test` vom Typ `QTap` in der Symboltabelle existiert,
   - ein semantischer Repräsentant in Form einer Instanz der Klasse `TapUnit` existiert,
   - dieser Repräsentant den Bezeichner `socket_test` als Wert des Namens-Attributs und
   - das erzeugte Symbol diesen Repräsentanten als Referenzobjekt erhalten hat.

2. Deklaration der Primitive:

```
/**
    Stellt den Aufruf eines Sockets dar.
    Ein Socket kennt Quell- und Ziel-IP (src und dest),
    Quell- und Zielport (src_port und dest_port) sowie Nutzlast (data).
*/
PRIMITIVE socket_data (
    byte[4] src,
    byte[4] dest,
    short src_port,
    short dest_port,
    byte[] data
);
```

   Der Parser erkennt diese Vereinbarung am Schlüsselwort `PRIMITIVE` und erhält als Information den Bezeichner `socket_data` und die Attribute in Form mehrerer Typisiserungsausdrücke. Außerdem wird der formale Kommentar zwischengespeichert und dieser Deklaration zugeordnet. Die Behandlung der Typisierungsausdrücke wird unter 6.8 beschrieben. Nach Abarbeitung dieser Vereinbarung

   - existiert ein Symbol mit dem Bezeichner `socket_data` vom Typ `QPrimitive` in der Symboltabelle,
   - hat das erzeugte Symbol diesen Repräsentanten als Referenzobjekt erhalten,
   - existiert ein semantischer Repräsentant in Form einer Instanz der Klasse `PrimitiveUnit`,
   - der den Bezeichner `socket_data` im Namens-Attribut und den formalen Kommentar speichert und die Typisierungsausdrücke als `Expression`-Objekte *in der angegebenen Reihenfolge und vom angegebenen Typ* hält.

3. Deklaration der Ereignisklasse:

```
/**
    Uns interessieren dabei sämtliche Daten, die über diesen Socket
    laufen.
    Wir geben unter ISOLATE also nur *s an.
*/
EVENT any_data
    USES socket_data
    ON socket_test
```

```
        ISOLATE (*,*,*,*,*)
        ID      AUTO;
```

Der Parser erkennt diese Vereinbarung am Schlüsselwort `EVENT` und erhält den Bezeichner `any_data`. Die Elemente `socket_data` und `socket_test` werden aufgrund der vorangehenden Schlüsselwörter `USES` und `ON` als Bezeichner für die referenzierte Primitive und die Instanz der SAP-Ankopplung interpretiert. Auf gleiche Weise wird der spezifizierte Identifikatormodus erkannt. Die Filtervorschrift hinter `ISOLATE` gibt an, welche Bedingungen die Attribute der referenzierten Primitive erfüllen müssen, damit ein Ereignis genau dieses Typs generiert wird. Das Zeichen `*` weist auf eine beliebige Belegung des jeweiligen Attributs hin — unser Beispiel ist sehr stark vereinfacht, weil die Funktionsweise des Übersetzers im Vordergrund steht. Der Parser sorgt dafür, daß nach Abarbeitung dieser Vereinbarung

- ein Symbol mit dem Bezeichner `any_data` vom Typ `QEvent` in der Symboltabelle existiert,
- ein semantischer Repräsentant in Form einer Instanz der Klasse `EventUnit` existiert,
- dieser Repräsentant den Bezeichner `any_data` als Wert des Names-Attributs und den Identifikatormodus „automatisch“ erhalten hat,
- das erzeugte Symbol diesen Repräsentanten als Referenzobjekt erhalten hat,
- der Repräsentant jeweils eine Referenz auf die semantischen Repräsentanten der angegebenen Primitive und der SAP-Ankopplung hält,
- den gegebenen formalen Kommentar speichert und
- die Liste von Filterregeln erhalten hat, die von Typ und Reihenfolge mit den Attributen der referenzierten Primitive korrespondieren sollen.

Im diesem Abschnitt bleibt die Tatsache, daß die Vereinbarung der Dienstgüte Vorwärtsreferenzierung verwendet, zunächst unberücksichtigt.

4. Deklaration der gemessenen Dienstgüte:

```
/**
Die Übertragungsrate ergibt sich direkt aus dem Messwert data_transfer.
*/
FEATURE data_transfer_rate
JOIN 1 OF data_transfer TO Sum;
```

Der Parser erkennt diese Vereinbarung am Schlüsselwort `FEATURE` und erhält als Information den Bezeichner `date_transfer_rate`. Die Elemente `data_transfer` und `Sum` werden aufgrund der vorangehenden Schlüsselwörter `OF` und `TO` als Bezeichner für den referenzierten Meßwert und die Aggregationsklasse interpretiert; der Berechnungsausdruck nach `JOIN` wird wiederum gesondert behandelt; dazu siehe Kapitel 6.8. Der Parser sorgt dafür, daß nach Abarbeitung dieser Vereinbarung

- ein Symbol mit dem Bezeichner `data_transfer_rate` vom Typ `QFeature` in der Symboltabelle existiert,
- ein semantischer Repräsentant in Form einer Instanz der Klasse `FeatureUnit` existiert,
- dieser Repräsentant den Bezeichner `data_transfer_rate` als Wert des Names-Attributs und den Berechnungsausdruck als `Expression`-Objekt erhalten hat,
- das erzeugte Symbol diesen Repräsentanten als Referenzobjekt erhalten hat,
- der Repräsentant eine Referenz auf denjenigen des angegebenen Meßwerts hält,
- wiederum den gegebenen formalen Kommentar speichert und
- ein semantischer Repräsentant der Aggregationsklasse in Form einer `AggregationUnit`-Instanz mit dem Namen `Sum` existiert, der von der Instanz von `FeatureUnit` aus referenziert wird.

5. Deklaration des Meßwerts:

```
/**
    Unser Messwert ergibt sich einfach aus der Anzahl an Socket-Aufrufen
    pro Zeiteinheit.
*/
READING data_transfer
    REQUIRES (
            any_data AS QUEUE trans
    )
    COMPUTES trans.getSize()
    UNIT "Aufrufe";
```

Der Parser erkennt diese Vereinbarung am Schlüsselwort `READING` und erhält den angegebenen Bezeichner `data_transfer`. Die Elemente `any_data` und `trans` werden aufgrund ihrer Position innerhalb der `REQUIRES`-Klammerung und den Schlüsselwörtern `AS QUEUE` als Bezeichner für die referenzierte Ereignisklasse und die ihnen zugeordnete Warteschlange interpretiert. Auf gleiche Weise wird die Maßeinheit durch vorangehendes `UNIT` erkannt. Der Berechnungsausdruck nach `COMPUTES` gehört wiederum zu den unter 6.8 beschriebenen Ausdrücken und wird als solcher behandelt. Der Parser sorgt dafür, daß nach Abarbeitung dieser Vereinbarung

- ein Symbol mit dem Bezeichner `data_transfer` vom Typ `QReading` in der Symboltabelle existiert,
- ein semantischer Repräsentant in Form einer Instanz der Klasse `ReadingUnit` existiert,
- dieser Repräsentant den Bezeichner `data_transfer` als Namen, den Berechnungsausdruck als `Expression`-Objekt und die Zeichenkette „Aufrufe“ als Maßeinheit erhalten hat,
- das erzeugte Symbol diesen Repräsentanten als Referenzobjekt erhalten hat,
- ein semantischer Repräsentant der Warteschlange in Form einer Instanz der Klasse `QueueUnit` existiert,
- das `ReadingUnit`-Objekt eine Referenz auf dieses `QueueUnit`-Objekt erhalten hat und damit Zugriff auf die Ereignisse (Instanzen von `EventUnit`) möglich ist und
- der formale Kommentar auf bekannte Weise gespeichert wird.

6.8 Ausdrücke

Typisierungsausdrücke und Berechnungsausdrücke, wie sie etwa zur Ermittlung des eigentlichen Meßwerts verwendet werden, werden nach dem Verfahren des *rekursiven Abstiegs* [Bry 04] behandelt. Dabei wird beim Parsen dieser *arithmetischen* oder *booleschen Ausdrücke* eine aus Knoten (abstrahiert als Instanzen der Klasse `Node`) bestehenden Baumstruktur (*Syntaxbaum* [Bry 04]) nach folgenden schematischen Regeln aus [USA 01] aufgebaut:

```
Seien E, E1, E2 (Teil-)ausdrücke,
T ein QoSSL-Typ,
id ein Bezeichner und
c eine Konstante.
Dann gilt sinngemäß:

E -> T E1           ist ein binärer Teilausdruck/Typisierungs-A. und wird zu
Node n = Node.makeBinNode (" ", NodeT, NodeE1);

E -> E1 op E2       ist ein binärer Teilausdruck und wird zu
NodeE = Node.makeBinNode ("op", NodeE1, NodeE2);
```

```
E -> -E1            ist ein unärer Teilausdruck und wird zu
NodeE = Node.makeUnaryNode ("-", NodeE1);

E -> (E1)           ist ein geklammerter Teilausdruck und wird zu
NodeE = Node.makeBracketNode ("(", NodeE1, ")");

E -> id             ist ein Blatt-Teilausdruck und wird zu
NodeE = Node.makeLeaf ("id");

E -> c              ist ein Blatt-Teilausdruck und wird zu
NodeE = Node.makeLeaf ("c");
```

Sind Parse-Vorgang und sämtliche kontextabhängigen Prüfungen erfolgreich abgeschlossen, so wird der Wurzelknoten in ein `Expression`-Objekt gekapselt, das dabei genau diejenige Instanz der Klasse `Node` referenziert, die die Wurzel des zuvor aufgebauten Syntaxbaumes darstellt. *Der (QoSSL-)Typ des Ausdrucks entspricht immer dem (QoSSL-)Typ des Wurzelknotens.*

```
Node rootNode = Node.make...;   /* Wurzelknoten ist rootNode vom Typ Node */
Expression e = new Expression (rootNode);
```

Dieses `Expression`-Objekt wird dann je nach Kontext an die richtige Stelle des Zwischencodes eingefügt. Ein Aufruf der Methode `print()` dieser Klasse sorgt dafür, daß die syntaktische Darstellung dieses Ausdrucks an entsprechender Stelle des *Zielcodes* eingesetzt werden kann. Dies geschieht mittels rekursiver Aufrufe der `print()`-Methode der Klasse `Node` gemäß dem sogenannten *INORDER-TREE-WALK-Algorithmus* [Corm 01]. Abbildung 6.2 zeigt den Syntaxbaum, der zum Berechnungsausdruck der in Kapitel 6.1 angegebenen Meßwert-Deklaration gehört.

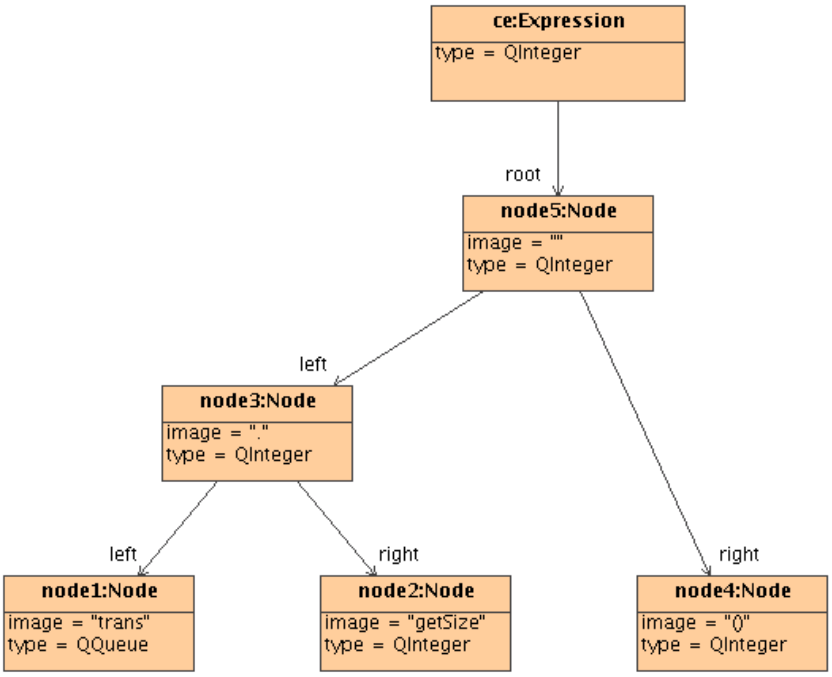

Abbildung 6.2: Syntaxbaum des Berechnungsausdrucks `trans.count()`

6.9 Algorithmen zur Vorwärtsreferenzierung

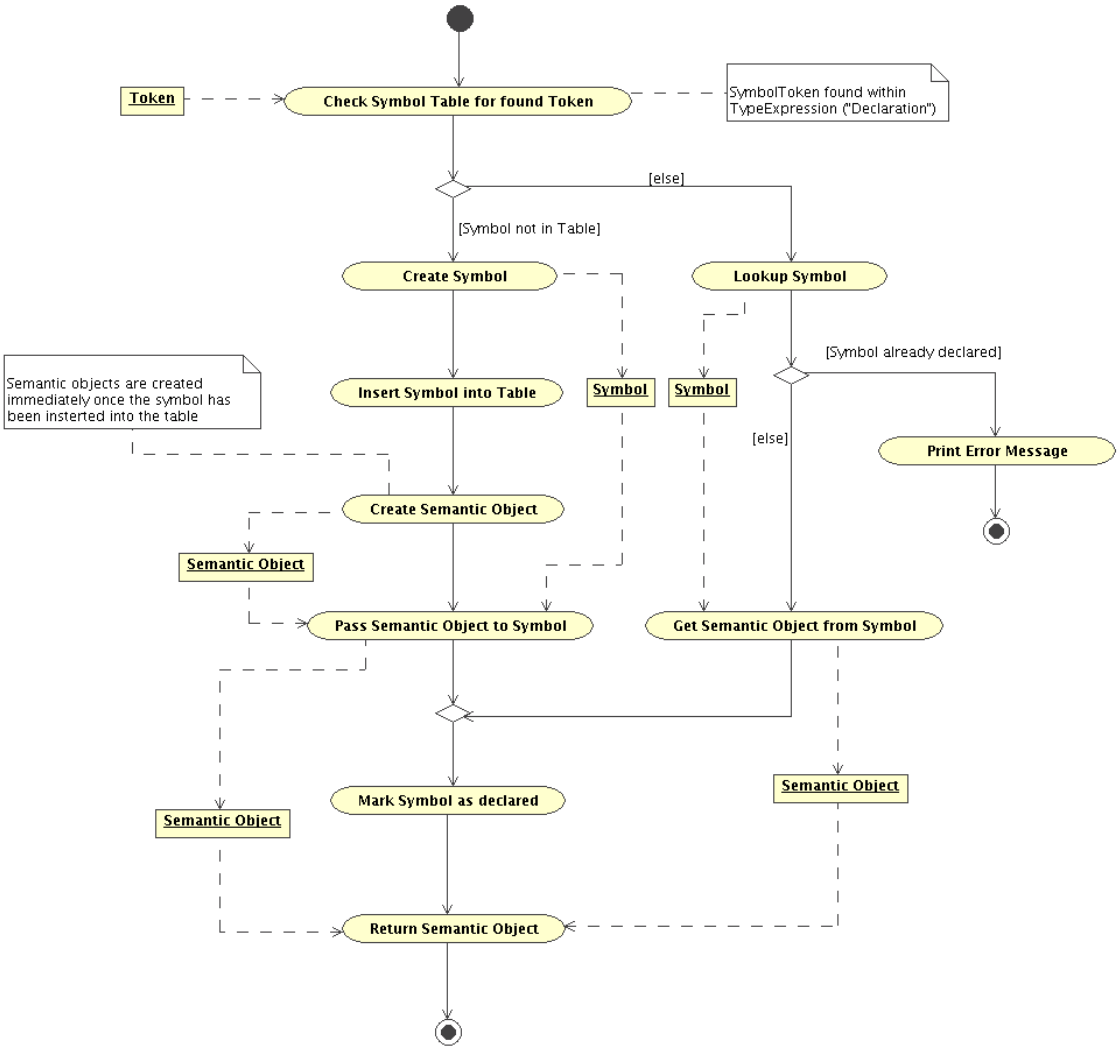

Abbildung 6.3: Algorithmus zur Auswertung von Typisierungsausdrücken

Um dem Anwender von QoSSL die Arbeit zu erleichtern, erlaubt dieser Übersetzer die Vorwärtsreferenzierung. So können bei sequentieller Betrachtung der Deklarationen Bezeichner vor ihrer Vereinbarung verwendet werden, was insbesondere im Hinblick auf das Zusammenfügen mehrerer Dateien zu einer Spezifikation eine große Hilfe ist. Dem Übersetzer stehen zur Bearbeitung von Vereinbarungen und Bezeichnern zwei Algorithmen zur Verfügung, mit denen die Entscheidung getroffen werden kann, ob die jeweiligen semantischen Repräsentanten erzeugt werden müssen oder aus den bereits vorhandenen zu ermitteln sind.
Der erste Algorithmus dient zur Behandlung eines Typisierungsausdrucks, in dem der Bezeichner für eine Komponente der QoSSL-Spezifikation (beispielsweise einer Ereignisklasse) *vereinbart* wird. Der Ablauf ist im UML-Aktivitätsdiagramm 6.3 [UML 01] [RJB 98] zu erkennen.
Zuerst wird geprüft, ob der erkannte Bezeichner in der Symboltabelle enthalten ist. Ist dies der Fall, so ist dieser Bezeichner schon mindestens einmal vor dieser Stelle im Quellcode aufgetaucht, und zwar entweder in einer Vereinbarung oder in fremdem Kontext — er wurde also lediglich *verwendet*. In diesem letzten Fall ist jedoch sein Typ aufgrund der zu verwendenden *Schlüsselwörter* (z. B. `USES` vor einem Bezeichner vom Typ `QPrimitive` innerhalb einer *Ereignis-Vereinbarung*) bereits eindeutig zu ermitteln gewesen. Ist die daran anschließende Typprüfung erfolgreich, so wird der (notwendigerweise bereits existierende) *semantische Re-*

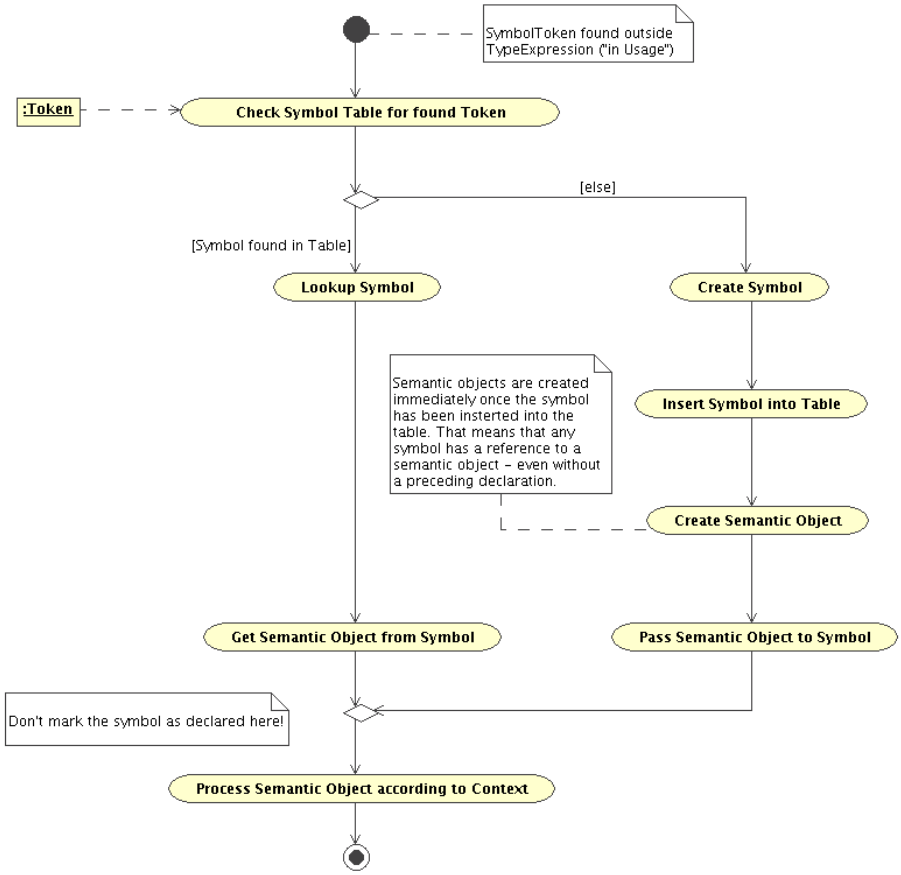

Abbildung 6.4: Algorithmus zur Auswertung von Ausdrücken bei Verwendung

präsentant ermittelt. Andernfalls wird der Bezeichner als `Symbol`-Objekt in der Symboltabelle unter dem übergebenen Typ gespeichert und ein semantischer Repräsentant erzeugt. Der Algorithmus terminiert aber nur dann erfolgreich, wenn das Symbol bisher noch in keiner anderen Deklaration vereinbart worden ist. Stellt das Programm an dieser Stelle fest, daß jenes Symbol bereits als „deklariert“ markiert worden ist, bricht es mit einer Fehlermeldung ab.
Der zweite Algorithmus dient zur Behandlung von Ausdrücken, in denen eine Komponente der QoSSL-Spezifikation *verwendet* wird, d. h. der Bezeichner nicht im Kontext seiner Vereinbarung im Quelltext auftaucht. Auch für diesen Algorithmus existiert mit Abbildung 6.4 ein UML-Aktivitätsdiagramm. Zuerst wird wiederum geprüft, ob der erkannte Bezeichner in der Symboltabelle enthalten ist. Ist dies der Fall, so ist dieser Bezeichner wiederum bereits vorher im Quellcode aufgetaucht, und sein Typ war zu ermitteln. Ist die daran anschließende Typprüfung erfolgreich, so wird der (notwendigerweise bereits existierende) semantische Repräsentant ermittelt. Andernfalls wird der Bezeichner als `Symbol`-Objekt in der Symboltabelle unter dem übergebenen Typ gespeichert und ein semantischer Repräsentant erzeugt.

6.10 Zwischencode

Terminiert soeben diskutierte erste Lauf erfolgreich, d. h. ohne Fehler in der gegebenen Entwicklersicht, so liegt die Spezifikation an dieser Stelle vollständig im Zwischencode vor.
Der Zwischencode besteht aus einem wohldefinierten Objektsystem, das hauptsächlich aus Instanzen der Klassen aus dem Paket `de.lmu.ifi.nm.qossl.semantics` besteht. Ausgehend vom Aufbau des Übersetzer-Programms sei zunächst dessen Struktur anhand von UML-Klassendiagrammen [UML 01] [RJB 98] erläutert. Die Fassadenklasse `QoSSLTranslationUnit` hält genau eine `TranslationUnitProperties`-Instanz

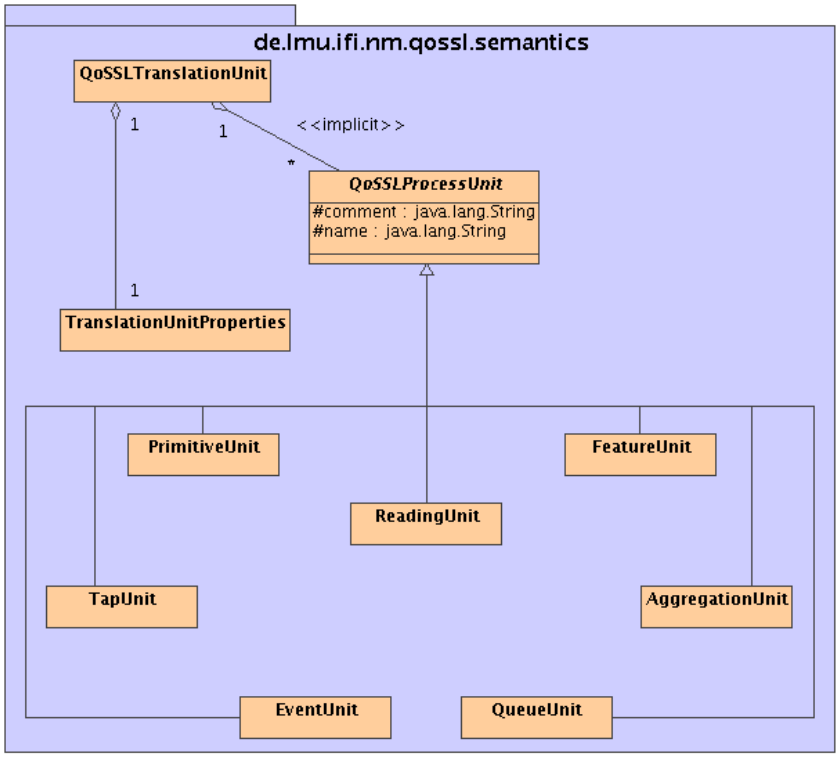

Abbildung 6.5: Grobe Struktur des Zwischencodes

vor, die die wesentlichen von der vorliegenden Spezifikation unabhängigen *Eigenschaften der zu bearbeitenden Übersetzungseinheit* definiert, also beispielsweise die durch Direktiven angegebenen Werte für Namensräume. Die eingezeichneten Unterklassen von `QoSSLProcessUnit` repräsentieren die Komponenten der gegebenen Spezifikation, indem sie beispielsweise Attribute mit deren Bezeichnern halten und Assoziationen untereinander aufbauen.

Konkret bedeutet die Abbildung einer gegebenen QoSSL-Spezifikation auf den Zwischencode eine geeignete Instanzierung der Klassen dieser Hierarchie und die Bildung von Verknüpfungen. Die vom Anwender gegebenen Bezeichner der Entitäten werden im jeweiligen `name`-Attribut gespeichert, ein optionaler formaler Kommentar wird im `comment`-Attribut abgelegt. Nachfolgende UML-Objektdiagramme [UML 01] [RJB 98] stellen den aus unserem Beispiel hervorgegangenen Zwischencode dar. Beide Abbildungen ergeben zusammengenonmmen die Gesamtheit des Zwischencodes, wie er sich nach erfolgreicher Beendigung des Parse-Vorgangs zu Beginn der Code-Erzeugung darstellt.

Das Objektdiagramm 6.7 zeigt jeweils mit kräftiger Farbe unterlegt die Übersetzungseinheit mit den semantischen Repräsentanten und ihre Zusammenhänge an. Mit schwächerer Farbe sind andere vom Anwender spezifizierte Informationen markiert; weiße Elemente dienen u. a. als Hilfsmittel zur Speicherung der Daten. Bezeichner und formale Kommentare der Komponenten sind in den beschriebenen Attributen abgelegt.
Links unten ist die Instanz von `PrimitiveUnit` zu erkennen, die die angegebene Primitive `socket_data` mit den Typisierungsausdrücken repräsentiert; der zugehörige formale Kommentar taucht im `comment`-Attribut wieder auf.
Rechts davon stehen die dazu korrespondierenden Filterregeln, die die zugehörige Ereignisklasse `any_data` definieren. Als Attribute des `EventUnit`-Objekts tauchen auch noch die Anzahl der Attribute und der Identifikatormodus auf.
Die (symbolische) Instanz der SAP-Ankopplung steht als `TapUnit` darüber.
Die Spezifikation des Meßwerts wird auf die rechts dargestellte Instanz von `ReadingUnit` abgebildet. Neben dem Namen `data_transfer` und dem Kommentar sind der Berechnungsausdruck als `Expression`-Objekt, ein `String` zur Speicherung der Maßeinheit und eine `Hashtable` für die Abbildung der genannten Ereignisklasse `any_data` auf die Warteschlange mit Namen `trans` als Attribute gespeichert.

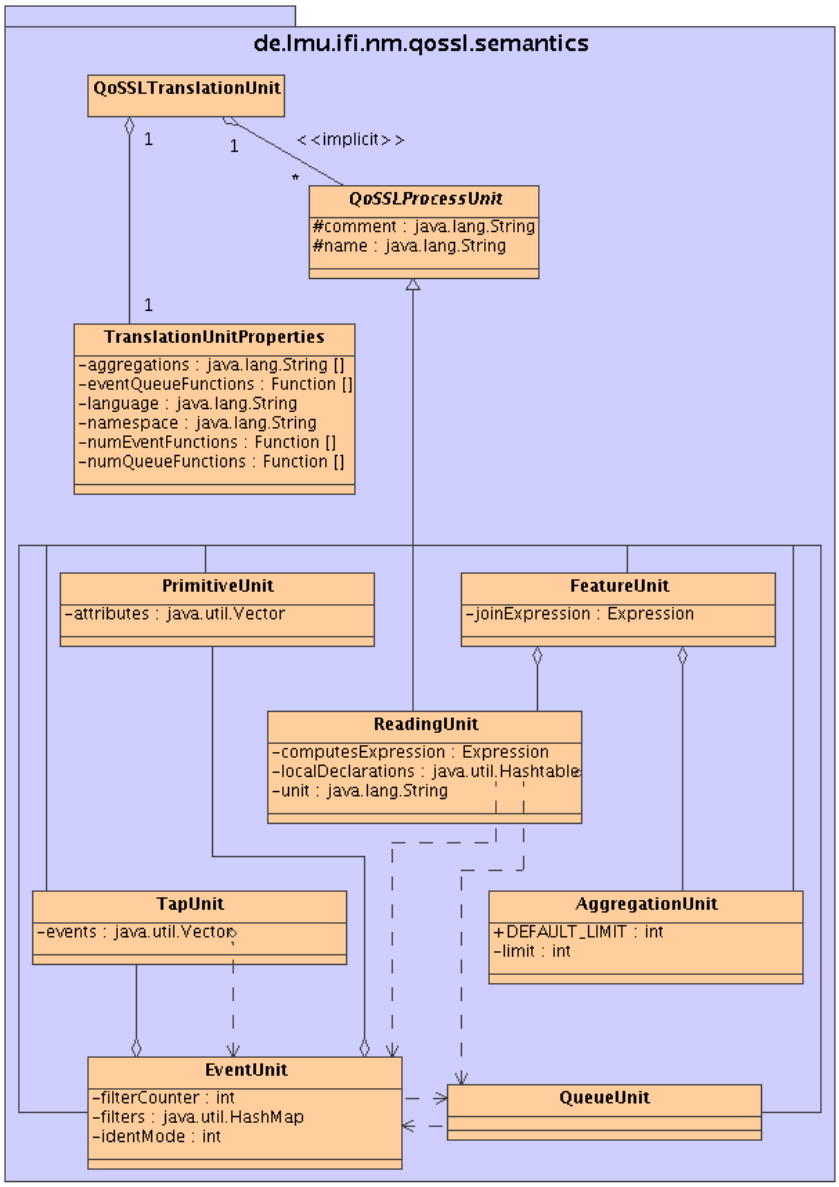

Abbildung 6.6: Verfeinerte Darstellung des Zwischencodes

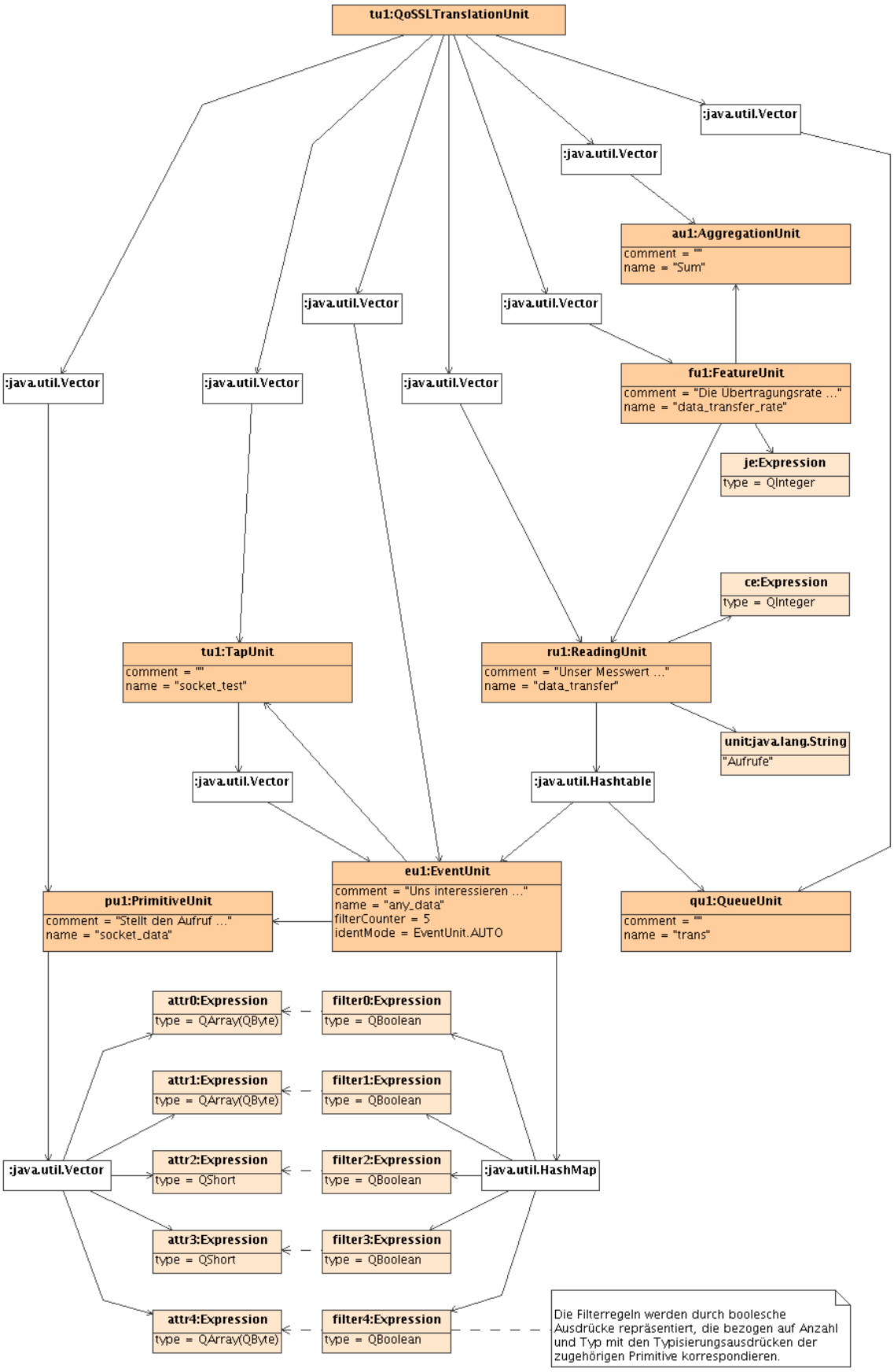

Abbildung 6.7: Aus der Spezifikation abgeleiteter Teil des Zwischencodes

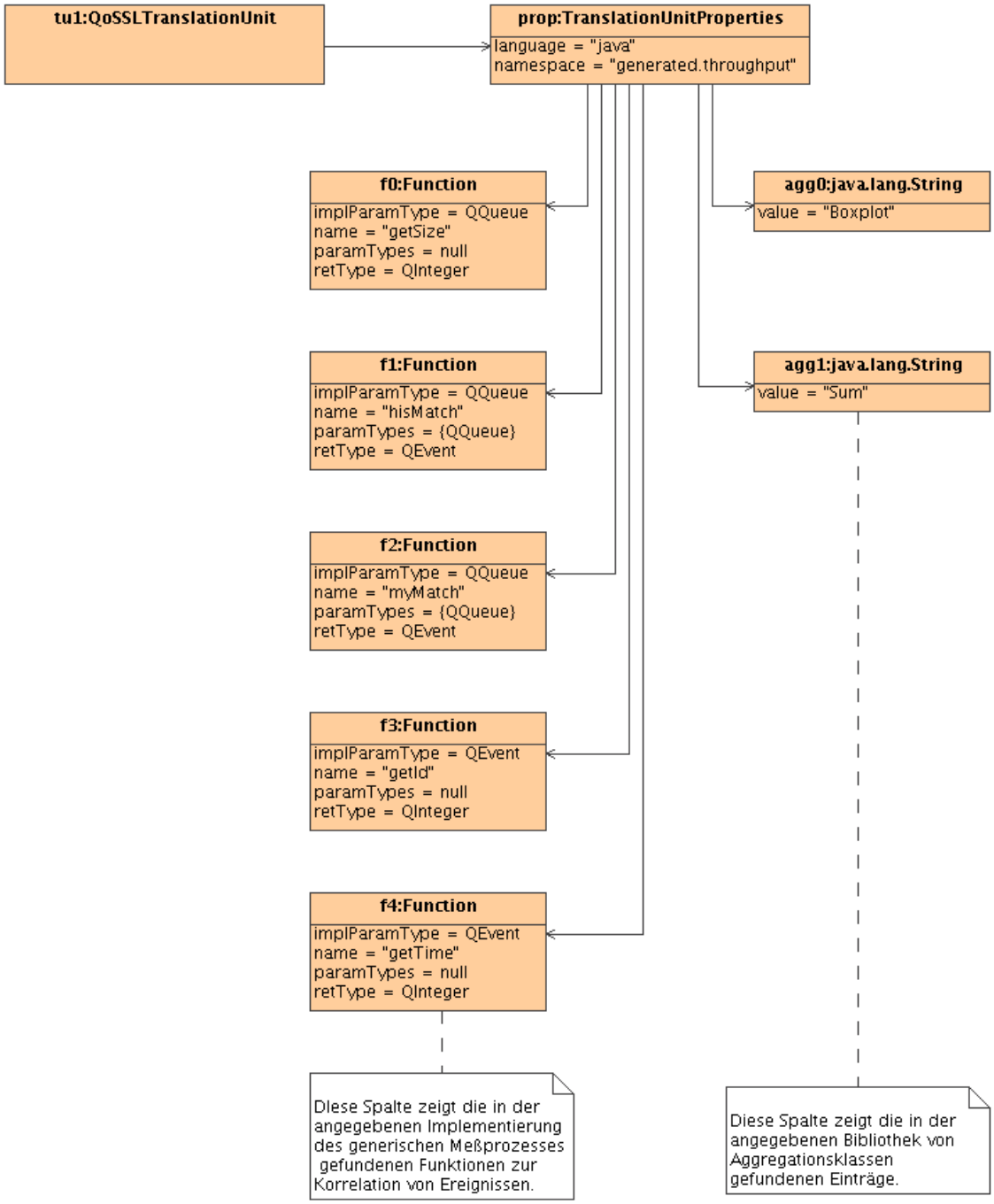

Abbildung 6.8: Aus Direktiven und Bibliotheken abgeleiteter Teil des Zwischencodes

Die darüberliegende Instanz der Klasse `FeatureUnit` hält neben einer Referenz auf den Repräsentanten des zugehörigen Meßwerts (`ReadingUnit`) noch den Berechnungsausdruck aus der Meßwert-Deklaration als `Expression`-Objekt. Direkt darüber ist eine `AggregationUnit`-Instanz zu erkennen, die mit dem Wert `Sum` im `name`-Attribut die gleichnamige Aggregationsklasse repräsentiert. Da hierzu keine weiteren Angaben existieren, ist der Kommentar an dieser Stelle leer.
Das UML-Objektdiagramm 6.8 zeigt oben links nochmal dasselbe `QoSSLTranslationUnit`-Objekt. Es hält immer genau eine Instanz der Klasse `TranslationUnitProperties`, deren Attribute mit den Werten aus den Direktiven belegt worden sind. Weiterhin sind darin die fünf durch die spezifische Reflektor-Implementierung auf die in Kapitel 6.6 beschriebene Weise gefundenen Korrelationsfunktionen als `Function`-Objekte abgelegt; deren Attributbelegungen lassen ihre jeweilige Signatur erkennen.
Rechts davon ist die auf gleiche Weise ermittelte Liste der Aggregationsklassen zu erkennen. Sie werden nur dem Namen nach unterschieden und sind als Objekte vom Typ `java.lang.String` abgespeichert.
Das vorliegende Implementierungsmuster erlaubt das Hinzufügen weiterer Korrelationsfunktionen und Aggregationsklassen; es müssen einfach weitere Instanzen nach diesem Schema hinzugefügt werden. Diese Erweiterbarkeit und die dynamische Suche solcher Einheiten zur Laufzeit stellt die notwendige Flexibilität des Programms an dieser Stelle sicher.

6.11 Ausgabe

Für die Erzeugung des Zielcodes ist die jeweils eingesetzte Generator-Implementierung verantwortlich. In unserem Beispiel wird jedem semantischen Repräsentanten eine Klasse zugeordnet, die genau den Teil der Definitionssicht erzeugt, der von diesem ableitbar ist. Das so erstellte System von Java-Klassen entspricht dem in [Gars 04] vorgestellten Ergebnis der Implementierung des automatisierten Ableitungsprozesses.
In diesem Abschnitt wird nun die aus unserem Beispiel erzeugte Definitionssicht besprochen. Dabei wird insbesondere dokumentiert, was über das in [Gars 04] eingeführte, ursprüngliche Konzept hinausgeht oder davon abweicht. Der erzeugte Code ist bis auf programminterne Dokumentation in voller Länge angegeben.
Jede vom Übersetzer angelegte Datei erhält eine Kopfzeile, die einen Hinweis auf das erzeugende Programm und dessen Autor sowie einen Zeitstempel beinhaltet.

```
/************************************************************************
          Datei Tap_socket_data.java
------------------------------------------------------------------------
          Erzeugt von QoSSL-Compiler, Version 0.17
          am Sat Apr 15 21:04:15 CEST 2006
          E-Mail: schaal (bei) cip.ifi.lmu.de
          Bitte diese Zeilen nicht verändern!
************************************************************************/
```

Ebenso wird jeweils eine Fußzeile eingefügt. Sie markiert einfach das Ende der Datei.

```
/************************************************************************
          Ende der Datei Tap_socket_data.java
          Bitte diese Zeilen nicht verändern!
************************************************************************/
```

Die Klasse `Tap_socket_data` in der Datei `Tap_socket_data.java` wird als Verfeinerung der Klasse `Tap` aufgebaut. Entgegen der ursprünglichen Implementierung entfällt die Testmethode `doIt()`, welche die Anbindung an einen realen SAP simuliert hätte, indem sie Instanzen der Klasse `Primitive_socket_data` versendet hätte. Stattdessen wird ein deutlicher Hinweis eingefügt, daß die Attribute von Hand zu belegen sind. Die notwendigen Typisierungsausdrücke sind als Hilfestellung in Form von Zeilenkommentaren angegeben.

```
package de.lmu.ifi.nm.qossl.generated;
```

```
import de.lmu.ifi.nm.qossl.base.java.*;

/**
    Rohling der SAP-Ankopplung für Primitive vom Typ Primitive_socket_data.
*/

public class Tap_socket_data
extends Tap {

    public void doIt () {
        Primitive_socket_data prim = new Primitive_socket_data ();
        /************************************
            ATTRIBUTE BELEGEN
        ************************************/

        // byte[4] src;

        // byte[4] dest;

        // short src_port;

        // short dest_port;

        // byte[] data;

        /************************************/
        send (prim);
    }

}
```

Die Datei `Primitive_socket_data.java` enthält die Klasse `Primitive_socket_data`. Entgegen der ursprünglichen Implementierung erfolgt die Bereitstellung der Attribute lediglich mit *komponenten-privatem* statt mit *komponenten-öffentlichem* Zugriff. Der im QoSSL-Quellcode spezifizierte Kommentar ist vom Übersetzer als *Dokumentationskommentar* eingefügt worden.

```
package de.lmu.ifi.nm.qossl.generated;

import de.lmu.ifi.nm.qossl.base.java.*;

/**
    Stellt den Aufruf eines Sockets dar.
    Ein Socket kennt Quell- und Ziel-IP (src und dest),
    Quell- und Zielport (src_port und dest_port) sowie Nutzlast (data).
*/

public class Primitive_socket_data
extends Primitive {

    byte[] src = new byte[4];
    byte[] dest = new byte[4];
    short src_port;
    short dest_port;
    byte[] data;

}
```

Die Klasse `EventGen_any_data` in der Datei `EventGen_any_data.java` erweitert `EventGenerator`. Die Filtervorschrift wird bereits durch eine `if`-Anweisung ausformuliert, wobei die einzelnen Filterregeln

durch *Logisches Und* miteinander verknüpft werden. Da zur Vereinfachung der Darstellung nur die Regel `(*,*,*,*,*)` angegeben ist, die keine Selektion darstellt, erscheinen im folgenden Abschnitt lediglich die Wahrheitswerte `true`. Solche augenscheinlich sinnlosen Tests werden der Einfachheit halber stehengelassen und nicht wegoptimiert; diesen Schritt kann der jeweilige Compiler übernehmen. Der formale Kommentar entstammt in diesem Fall der Generator-Implementierung.

```
package de.lmu.ifi.nm.qossl.generated;

import de.lmu.ifi.nm.qossl.base.java.*;

/**
    Generator-Klasse für Ereignisse vom Typ Event_any_data.
*/

public class EventGen_any_data
extends EventGenerator {

    private static long eventID = 0;

    public void receive (Primitive aPrimitive) {
        Primitive_socket_data primitive1 =
            (Primitive_socket_data) aPrimitive;
        if (
            true
        &&
            true
        &&
            true
        &&
            true
        &&
            true
        ) {
            Event_any_data event1 = new Event_any_data (
                "Event_any_data", eventID, System.currentTimeMillis ()
            );
            eventID++;
            send (event1);
        }
    }

}
```

Die Klasse `Event_any_data` ist eine Erweiterung von `Event` gemäß der ursprünglichen Konzeption. Wiederum hat der Übersetzer den angegebenen formalen Kommentar an entsprechender Stelle eingefügt.

```
package de.lmu.ifi.nm.qossl.generated;

import de.lmu.ifi.nm.qossl.base.java.*;

/**
    Uns interessieren dabei sämtliche Daten, die ueber diesen Socket laufen.
    Wir geben unter ISOLATE also nur *s an.
*/

public class Event_any_data
extends Event {

    public Event_any_data (String nameToSet,
```

```
            long idToSet, long timeStampToSet) {
            super (nameToSet, idToSet, timeStampToSet);
        }

    }
```

Die `EventCorrelator` erweiternde Klasse `EventCorr_data_transfer` wird entsprechend in der Datei `EventCorr_any_data.java` abgelegt. Innerhalb der Methode `correlate()` finden wir unseren Korrelationsausdruck wieder, dem wir unter `COMPUTES` angegeben hatten. Auch hier ist der Dokumentationskommentar nicht aus der Spezifikation abgeleitet.

```
    package de.lmu.ifi.nm.qossl.generated;

    import de.lmu.ifi.nm.qossl.base.java.*;

    /**
        Korrelator-Klasse.
    */

    public class EventCorr_data_transfer
    extends EventCorrelator {

        Queue trans;

        public EventCorr_data_transfer () {
            trans = new Queue ();
        }

        public void receive (Event_any_data receivedEvent) {
            trans.add (receivedEvent);
            correlate ();
        }

        public void correlate () {
            send (new Reading_data_transfer (trans.getSize(), 0, 0));
        }

    }
```

Die Meßwerte repräsentierende Klasse `Reading` wird durch `Reading_data_transfer` in der Datei `Reading_data_transfer.java` gemäß der Konzeption erweitert. Wiederum erscheint der formale Kommentar an gewohnter Stelle.

```
    package de.lmu.ifi.nm.qossl.generated;

    import de.lmu.ifi.nm.qossl.base.java.*;

    /**
        Unser Messwert ergibt sich einfach aus der Anzahl an Socket-Aufrufen
        pro Zeiteinheit.
    */

    public class Reading_data_transfer
    extends Reading {

        public Reading_data_transfer (double valueToSet,
            long processedEventsToSet, double relativeErrorToSet) {
            super (valueToSet, processedEventsToSet, relativeErrorToSet);
```

```
        }

    }
```

Ebenso implementiert die Klasse `PostProcessor_data_transfer_rate` in der entsprechenden Datei `PostProcessor_data_transfer_rate.java` das Interface `PostProcessor`. Der Berechnungsausdruck aus der `FEATURE`-Deklaration ist innerhalb der `receive()`-Methode zu sehen. Die Testausgabe in der `send()`-Methode wird entgegen der ursprünglichen Fassung weggelassen.

```
    package de.lmu.ifi.nm.qossl.generated;

    import de.lmu.ifi.nm.qossl.base.java.*;

    /**
        Diese Klasse dient zur statistischen Nachverarbeitung von Messwerten vom
        Typ data_transfer.
    */

    public class PostProcessor_data_transfer_rate
    implements PostProcessor {

        FeatureReceiver theReceiver;
        int readingCounter = 0;
        Feature_data_transfer_rate featureToBuild;

        public void bind (FeatureReceiver receiverToRegister) {
            theReceiver = receiverToRegister;
        }

        public void receive (Reading aReading) {
            if (readingCounter == 0) {
                featureToBuild = new Feature_data_transfer_rate ();
            }
            readingCounter++;
            featureToBuild.addValue (aReading.getValue ());
            if (readingCounter >= 1) {
                send (featureToBuild);
                readingCounter = 0;
            }
        }

        public void send (Feature featureToSend) {
            /*************************************************
             KOMMENTARZEICHEN ENTFERNEN, FALLS EMPFÄNGER VORHANDEN
             *************************************************/
            // theReceiver.receive (featureToSend);
        }

    }
```

Die für Dienstgütemerkmale zuständige Klasse `Feature` wird durch `Feature_data_transfer_rate` in der Datei `Feature_data_transfer_rate.java` gemäß der Konzeption als Erweiterung der gefundenen Aggregationsklasse implemetiert. Auch hier ist wieder der im Quellcode angegebene formale Kommentar zu sehen.

```
    package de.lmu.ifi.nm.qossl.generated;

    import de.lmu.ifi.nm.qossl.base.java.*;
```

```
import de.lmu.ifi.nm.qossl.aggregation.java.*;

/**
    Die Übertragungsrate ergibt sich direkt aus dem Messwert data_transfer.
*/

public class Feature_data_transfer_rate
extends Sum
implements Feature {

}
```

Die Klasse `CustomizedBuilder` erweitert als spezifische Instanzierungshilfestellung die Klasse `Builder` aus dem Basispaket. Dabei werden im Konstruktor alle für den Meßprozeß notwendigen Komponenten aufgebaut. Die Methode `buildInstances()` sorgt für eine korrekte Bindung der Komponenten aneinander.

```
package de.lmu.ifi.nm.qossl.generated;

import de.lmu.ifi.nm.qossl.base.java.*;

/**
    Diese Klasse dient als spezifische Instanzierungshilfestellung.
    Dabei werden alle notwendigen Instanzen des Mess-Systems aufgebaut.
*/

public class CustomizedBuilder
extends Builder {

    Tap socket_test;
    EventGen_any_data the_EventGen_any_data;
    EventCorr_data_transfer the_EventCorr_data_transfer;
    PostProcessor_data_transfer_rate the_PostProcessor_data_transfer_rate;

    public CustomizedBuilder (Tap _socket_test) {
        socket_test = _socket_test;
        the_EventGen_any_data = new EventGen_any_data ();
        the_EventCorr_data_transfer = new EventCorr_data_transfer ();
        the_PostProcessor_data_transfer_rate =
            new PostProcessor_data_transfer_rate ();
    }

    public void buildInstances () {
        socket_test.bind (the_EventGen_any_data);
        the_EventGen_any_data.bind (the_EventCorr_data_transfer);
        the_EventCorr_data_transfer.bind (
            the_PostProcessor_data_transfer_rate
        );
    }

}
```

Im Gegensatz zur Implementierung aus [Gars 04] baut die vorliegende Implementierung *keine* Testklasse auf.

7 Zusammenfassung und Ausblick

In dieser Arbeit wird das in [Gars 04] eingeführte Konzept zur formalen Spezifikation von Dienstgütemerkmalen und deren rechnergestützte Umsetzung um ein brauchbares Werkzeug erweitert. Im Gegensatz zur bisherigen prototypischen Implementierung ist dieses Programm bereits für den Arbeitseinsatz geeignet.

7.1 Spezifikationssprache

Als Grundlage für den Übersetzer wird in dieser Arbeit die in [Gars 04] vorgestellte formale Spezifikationssprache QoSSL im Hinblick auf die bisherige Diskussion der Problemstellung syntaktisch in allen Einzelheiten festgelegt. Im Vergleich zur ursprünglichen Fassung haben sich nur insofern Änderungen ergeben, als im Rahmen neuer Anforderungen Erweiterungen notwendig waren. Beispiele hierfür sind die Makros, die eine Zusammensetzung der QoSSL-Spezifikation aus mehreren Dateien und bedingtes Kompilieren ermöglichen und die Direktiven zur Steuerung der Übersetzung.
Das eingebaute Typsystem von QoSSL verfügt über Verbunde, Zahlen und Wahrheitswerte einerseits und die Spezialtypen zur Repräsentation der Spezifikationskomponenten andererseits. Weiterhin ist es durch Bereitstellung der `import`-Anweisung auf Erweiterbarkeit vorbereitet. Allerdings sind noch Mechanismen zur Verwaltung der auf diese Weise bekanntgemachten Typen zu realisieren. Diese sind unerläßlich, um bei den Attributen von Primitiven ein höheres Abstraktionsniveau zu erreichen.

7.2 Übersetzer

Der Übersetzer selbst ist objektorientiert programmiert; er wird mittels Parser-Generator erzeugt und mit geeigneten Behandlungs- und Ausgaberoutinen ergänzt. In einem ersten Lauf wird die vom Anwender vorgegebene Spezifikation eingelesen und in einen abstrakten, objektbasierten Zwischencode abgelegt. Findet der Übersetzer bis zu dieser Stelle Fehler, so kann er den Vorgang hier abbrechen. Die zweite Phase der Übersetzung, die Code-Erzeugung, findet erst nach Beendigung des Parse-Vorgangs in einem zweiten Lauf statt. Korrelationsfunktionen und Aggregationsklassen werden zur Übersetzungszeit dynamisch gesucht; der Anwender entscheidet, welche Bibliotheken einzubeziehen sind. LL(1) ist realisiert bis auf einige Ausnahmen bei arithmetischen Ausdrücken.
Der Zwischencode abstrahiert so stark, daß das Programm eine beliebige Ausgabe erzeugen kann. Der modulare Aufbau des Übersetzers ermöglicht sogar das Einbeziehen unterschiedlicher Eingabebibliotheken. Der Programmierer einer solchen Erweiterung ist sogar in der Wahl seiner Programmiersprache frei; er hat nur eine geeignete Realisierung der Schnittstellen in der Implementierungssprache des Übersetzers bereitzustellen. Die Implementierung des Parsers selbst bleibt davon unberührt.
Die Bereitstellung verschiedener Implementierungen des allgemeinen Meßprozesses mit seinen Korrelationsfunktionen und Aggregationsklassen selbst ist nicht Bestandteil dieser Arbeit und erfolgt dementsprechend bisher nur ansatzweise. Das vorliegende Programm legt jedoch durch seinen modularen Aufbau die Grundlage für eine beliebige Zahl solcher Erweiterungen, die sich leicht integrieren lassen. Seien es nun andere Aggregationsfunktionalitäten, eine Verbesserung des Meßsystems etwa hinsichtlich der Fehlerabschätzung oder dessen Implementierung in einer anderen Programmiersprache zur Anpassung an technische Gegebenheiten. Auch eine physische Trennung der Komponenten, etwa in einem Netzwerk, sind wegen ihrer geringen Kopplung denkbar.

7.3 Fehlerbehandlung

Während des gesamten Parse-Vorgangs wird die QoSSL-Spezifikation auf mögliche Fehler hin untersucht. Zur Erkennung syntaktischer Fehler wird vom Parser-Generator bereits die Klasse `ParseException` in das erzeugte Programm eingebaut. Diese Art der Fehlerbehandlung deckt bisher nur Syntaxfehler ab; zudem sind die Fehlermeldungen schwer verständlich.
Für den Anwender sind daneben jedoch vor allem *semantische Fehler* interessant, z. B.:

- Typfehler, d. h. erwarteter und vorhandener Typ eines (Teil-)ausdrucks stimmen nicht überein,
- Bezeichner, die zwar verwendet, aber nirgends deklariert werden,
- Bezeichner, die zwar deklariert, aber nirgends verwendet werden,
- unvollständige Spezifikationen,
- Verwendung nicht vorhandener Korrelationsfunktionen oder Aggregationsklassen,
- Angabe einer falschen Signatur bei der Verwendung von Methoden in Berechnungsausdrücken,
- Widersprüchliche Direktiven,

Derartige Fehler verursachen zwar meistens einen Abbruch der Übersetzung, können aber nicht anhand der Meldungen innerhalb der Spezifikation lokalisiert werden.
Um eine für den Anwender nutzbringende Fehlerbehandlung zu gewährleisten, bedarf es also noch folgender Erweiterungen:

- Ein Fehler muß *auf jeden Fall* zu einem Abbruch des Übersetzungsvorgangs in dem Sinne führen, daß die Code-Erzeugung auf keinen Fall angestoßen wird, und der Anwender eine Nachricht erhält. Ein sofortiger Abbruch des Parsens ist jedoch nicht in jedem Fall sinnvoll.
- Das Programm muß ausnahmslos alle Fehler finden und melden.
- Fehlermeldungen müssen für den Anwender verständlich sein. Dazu sollen nicht nur „gefundene“ und „erwartete“ syntaktische Tokens gemeldet, sondern auch so weit wie möglich, der Kontext erklärt werden.
- Da vor dem Parsen mittels GCC-Präprozessor mehrere Dateien zu einer Spezifikation zusammengefügt werden können, ist eine Zuordung der Fehler auf Quelldateien und Zeilennummern erforderlich. Es reicht nicht, die tatsächliche Fundstelle anzuzeigen, denn der Anwender arbeitet ja nicht mit der eigentlich geparsten Datei, in der sich, wie aus Kapitel 6.2 bekannt, die bereits zusammengesetzte Spezifikation befindet.
- Weiterhin sollte das Programm bei „kleineren Ungereimtheiten“ Warnmeldungen ausgeben können. Im unserem Beispiel aus 6.2 wäre gemäß den Erläuterungen unter 6.3 die weggelassene `NAMESPACE`-Direktive zu monieren.

Das vorliegende Programm bietet bereits eine gute Grundlage für die Integration einer Fehlerbehandlung. Die Parser-Komponente kann nicht nur syntaktische Fehler erkennen, sondern auch beispielsweise Typfehler oder fehlende Reflektor-Implementierungen. Zur Behandlung weiterer Fehler bedarf es an dieser Stelle lediglich geringfügiger Erweiterungen, etwa eine Prüfung auf die Menge der gefundenen Aggregationsklassen hin.
Schließlich bleibt noch zu untersuchen, welche Fehler beim Parsen definitiv nicht gefunden werden können. In solchen Fällen kann der erstellte Zwischencode auf Grundlage der Spezifikationsrichtlinien hin untersucht werden. Dazu bietet es sich an, diese Richtlinien auf Basis von *Einschränkungen* (*Constraints*) innerhalb des `semantics`-Pakets zu formulieren; dies könnte beispielsweise mit OCL [Warm 99] [OCL 03] [Henn 04] geschehen.
Im Hinblick auf den objektorientierten Ansatz der bisherigen Implementierung erscheint eine Hierarchie von Ausnahmeklassen auf Basis von `java.lang.Exception` sinnvoll. So könnten einige von ihnen zur Behandlung von Parse-Fehlern `ParseException`-Objekte abfangen, andere wiederum Fehler im Zwischencode repräsentieren.

Die Fehlerbehandlung muß das Ziel verfolgen, daß die Ausgabe dieses Programms in jedem Falle frei von jeglichen Fehlern ist. Eine Umformung der erzeugten Definitionssicht in die Meßsicht soll in jedem Falle erfolgreich terminieren, sofern generischer Meßprozeß, Korrelations- und Aggregationsbibliotheken keine Fehler enthalten und sämtliche gemeldeten Fehler innerhalb der Entwicklersicht korrigiert worden sind. Der Grund dafür ist, daß Fehler, die erst bei Erzeugung der Meßsicht aus der Definitionssicht erkannt werden, für den die Entwicklersicht bereitstellenden Anwender nicht nachvollziehbar sind. Der Sinn des QoSSL-Formalismus ist ja gerade der, sich nicht mit der Definitionssicht auseinandersetzen zu müssen.

A QoSSL-Grammatik

In diesem Abschnitt ist die Grammatik der Sprache QoSSL in EBNF [ISO 14977] angegeben. Ausgenommen bleiben die regulären Ausdrücke zur Bildung von Bezeichnern und nicht-trivialen Literalen. Entsprechende Stellen sind mit `(* ... *)` gekennzeichnet.

```
<QOSSL_TRANSL_UNIT> ::= <DIRECTIVES>
                        (<IMPORT>)*
                        (<DECLARATION>)*.

<DIRECTIVES>           ::= (<LANG_DIRECTIVE>
                           | <NAMESPACE_DIRECTIVE>
                        (* | ... *)
                           ).
<LANG_DIRECTIVE>      ::= "LANG" <STRING_LITERAL>.
<NAMESPACE_DIRECTIVE> ::= "NAMESPACE" <STRING_LITERAL>.
(* ... *)

<IMPORT> ::= "import" <CLASS_TYPE> (".*")? ";".

<DECLARATION> ::= <EMPTY_DECL> |
                  <TAP_DECL> |
                  <PRIMITIVE_DECL> |
                  <EVENT_DECL> |
                  <READING_DECL> |
                  <FEATURE_DECL> .

<EMPTY_DECL> ::= ";".

<TAP_DECL> ::= "TAP" <IDENTIFIER> ";".

<PRIMITIVE_DECL>   ::= "PRIMITIVE" <IDENTIFIER>
                       "(" (<TYPE_CONSTR_LIST>)?")" ";".
<TYPE_CONSTR_LIST> ::= <TYPE_CONSTRAINT>
                       (","<TYPE_CONSTRAINT>)*.
<TYPE_CONSTRAINT>  ::= <TYPE><IDENTIFIER>.

<EVENT_DECL>        ::= "EVENT" <IDENTIFIER>
                        "USES" <IDENTIFIER>
                        "ON" <IDENTIFIER>
                        "ISOLATE" <FILTER_EXPN_LIST>
                        "ID" <IDENTIFICATE_BY>
                        ";".
<FILTER_EXPN_LIST>  ::= "("(<FILTER_EXPRESSION>
                        (","<FILTER_EXPRESSION>)*)?")".
<FILTER_EXPRESSION> ::= ("*"|<GENERIC_EXPRESSION>).
<IDENTIFICATE_BY>   ::= (<IDENTIFIER>|"AUTO"|"HASH" <ARGUMENTS>).

<READING_DECL>      ::= "READING" <IDENTIFIER>
                        "REQUIRES" <LOCAL_DECL_LIST>
                        "COMPUTES" <EXPRESSION>
                        "UNIT" <STRING_LITERAL>
                        ";".
```

```
<LOCAL_DECL_LIST>   ::= "("<LOCAL_DECLARATION>
                        (","<LOGICAL_OR_EXPN>)*")".
<LOCAL_DECLARATION> ::= <IDENTIFIER> "AS" "QUEUE" <IDENTIFIER>.

<FEATURE_DECL> ::= "FEATURE" <IDENTIFIER>
                   "JOIN" <EXPRESSION>
                   "OF" <IDENTIFIER>
                   "TO" <IDENTIFIER>
                   ";".

<TYPE>           ::= (<REFERENCE_TYPE>|<PRIMITIVE_TYPE>).
<REFERENCE_TYPE> ::= (<PRIMITIVE_TYPE>("["<LOGICAL_OR_EXPN>"]")+ |
                      <CLASS_TYPE>("["<LOGICAL_OR_EXPN>"]")*).
<CLASS_TYPE>     ::= <IDENTIFIER>("."<IDENTIFIER>)*.
<PRIMITIVE_TYPE> ::= ("boolean"|"byte"|"short"|"int"|"long"|
                      "float"|"double").

<GENERIC_EXPRESSION> ::= <LOGICAL_OR_EXPN>.
<LOGICAL_OR_EXPN>    ::= <LOGICAL_AND_EXPN>("||"<LOGICAL_AND_EXPN>)*.
<LOGICAL_AND_EXPN>   ::= <INCL_OR_EXPN>("&&"<INCL_OR_EXPN>)*.
<INCL_OR_EXPN>       ::= <EXCL_OR_EXPN>("|"<EXCL_OR_EXPN>)*.
<EXCL_OR_EXPN>       ::= <AND_EXPN>("^"<AND_EXPN>)*.
<AND_EXPN>           ::= <EQUALITY_EXPN>("&"<EXCL_OR_EXPN>)*.
<EQUALITY_EXPN>      ::= <RELATIONAL_EXPN>(("=="|"!=")<RELATIONAL_EXPN>)*.
<RELATIONAL_EXPN>    ::= <SHIFT_EXPN>
                         ((">"|">="|"<"|"<=")<SHIFT_EXPN>)*.
<SHIFT_EXPN>         ::= <ADDITIVE_EXPN>
                         ((">>"|">>>"|"<<)<ADDITIVE_EXPN>)*.
<ADDITIVE_EXPN>      ::= <MULT_EXPN>(("+"|"-")<MULT_EXPN>)*.
<MULT_EXPN>          ::= <UNARY_EXPN>(("*"|"/"|"%")<UNARY_EXPN>)*.
<UNARY_EXPN>         ::= (("+"|"-"|"~"|"!")<UNARY_EXPN> |
                         <CAST_EXPN> |
                         <PRIMARY_EXPN>).
<CAST_EXPN>          ::= "("<TYPE>")"<UNARY_EXPN>.
<PRIMARY_EXPN>       ::= <PRIMARY_PREFIX>(<PRIMARY_SUFFIX>)*.
<PRIMARY_PREFIX>     ::= (<LITERAL> |
                         <CLASS_TYPE> |
                         "("<LOGICAL_OR_EXPN>")").
<PRIMARY_SUFFIX>     ::= ("["<LOGICAL_OR_EXPN>"]"|"."<IDENTIFIER>|
                         <ARGUMENTS>).

<LITERAL>                ::= (<INTEGER_LITERAL> |
                             <FLOATING_POINT_LITERAL> |
                             <BOOL_LITERAL> |
                             <NULL_LITERAL> |
                             <IDDEM_LITERAL>).
<INTEGER_LITERAL>        ::= (* ... *) .
<FLOATING_POINT_LITERAL> ::= (* ... *) .
<STRING_LITERAL>         ::= (* ... *) .
<BOOL_LITERAL>           ::= ("true"|"false").
<NULL_LITERAL>           ::= "null".
<IDDEM_LITERAL>          ::= "IDDEM".

<ARGUMENTS>     ::= "("(<ARGUMENT_LIST>)?")".
<ARGUMENT_LIST> ::= <LOGICAL_OR_EXPN>
                    (","<LOGICAL_OR_EXPN>)*.

<IDENTIFIER> ::= ... (* ... *) .
```

B Bestandteile des Compiler-Programms

In diesem Abschnitt sind die einzelnen Programmpakete des Übersetzers aufgelistet. Eine detaillierte Beschreibung liegt dem Programm in Form von formalen Kommentaren in den einzelnen Klassen bei, aus denen mit dem Dokumentationsgenerator *Javadoc* der zugehörige Satz HTML-Dateien [HTML] erzeugt worden ist. Sie sind unter [QoSH] zu erreichen.
Zur Orientierung sei an dieser Stelle der Verzeichnisbaum des mitgelieferten Programms mit allen Dateien angegeben:

```
.
|-- README
|-- bin
|   |-- ...
|-- doc
|   |-- ...
|-- generate.sh
|-- qossl
|   |-- anhang_a4.qossl
|   |-- eth_faulty_frame_rate.qossl
|   |-- http.qossl
|   |-- icmp_echo.qossl
|   |-- socket_data.qossl
|   |-- socket.qossl
|   |-- telnet.qossl
|   |-- throughput_body.qossl
|   |-- throughput.qossl
|   |-- url_time.qossl
|   `-- ...
|-- src
|   `-- de
|       `-- lmu
|           `-- ifi
|               `-- nm
|                   `-- qossl
|                       |-- *.png
|                       |-- aggregation
|                       |   |-- java
|                       |   |   |-- AggregationRepository.java
|                       |   |   |-- Boxplot.java
|                       |   |   `-- Sum.java
|                       |-- analyzer
|                       |   |-- BinaryNode.java
|                       |   |-- BracketNode.java
|                       |   |-- Expression.java
|                       |   |-- LeafNode.java
|                       |   |-- Node.java
|                       |   |-- SemanticException.java
|                       |   |-- StaticSymbolTable.java
|                       |   |-- Symbol.java
|                       |   |-- SymbolAlreadyDeclaredException.java
|                       |   |-- SymbolAlreadyInTableException.java
|                       |   |-- SymbolNotDeclaredException.java
|                       |   |-- SymbolNotInTableException.java
```

```
|   |                       |   |-- SymbolTable.java
|   |                       |   |-- TypeErrorException.java
|   |                       |   `-- UnaryNode.java
|   |                       |-- base
|   |                       |   |-- BaseProperties.java
|   |                       |   |-- Reflector.java
|   |                       |   |-- java
|   |                       |   |   |-- Builder.java
|   |                       |   |   |-- Event.java
|   |                       |   |   |-- EventCorrelator.java
|   |                       |   |   |-- EventGenerator.java
|   |                       |   |   |-- EventReceiver.java
|   |                       |   |   |-- EventSender.java
|   |                       |   |   |-- Feature.java
|   |                       |   |   |-- FeatureReceiver.java
|   |                       |   |   |-- FeatureSender.java
|   |                       |   |   |-- JavaProperties.java
|   |                       |   |   |-- PostProcessor.java
|   |                       |   |   |-- Primitive.java
|   |                       |   |   |-- PrimitiveReceiver.java
|   |                       |   |   |-- PrimitiveSender.java
|   |                       |   |   |-- Queue.java
|   |                       |   |   |-- Reading.java
|   |                       |   |   |-- ReadingReceiver.java
|   |                       |   |   |-- ReadingSender.java
|   |                       |   |   `-- Tap.java
|   |                       |-- generated
|   |                       |-- out
|   |                       |   |-- Generator.java
|   |                       |   |-- generic
|   |                       |   |   `-- CodeWriter.java
|   |                       |   |-- java
|   |                       |   |   |-- CodeFileGenerator.java
|   |                       |   |   |-- Event_JavaFileGen.java
|   |                       |   |   |-- Feature_JavaFileGen.java
|   |                       |   |   |-- GeneratorImpl.java
|   |                       |   |   |-- JavaFileProperties.java
|   |                       |   |   |-- Primitive_JavaFileGen.java
|   |                       |   |   |-- Reading_JavaFileGen.java
|   |                       |   |   |-- ReflectorImpl.java
|   |                       |   |   `-- TranslUnit_JavaFileGen.java
|   |                       |-- overview.html
|   |                       |-- parser
|   |                       |-- qossl.jj
|   |                       |-- qossl_stylesheet.css
|   |                       |-- semantics
|   |                       |   |-- AggregationUnit.java
|   |                       |   |-- EventUnit.java
|   |                       |   |-- FeatureUnit.java
|   |                       |   |-- PrimitiveUnit.java
|   |                       |   |-- QoSSLProcessUnit.java
|   |                       |   |-- QoSSLTranslationUnit.java
|   |                       |   |-- QueueUnit.java
|   |                       |   |-- ReadingUnit.java
|   |                       |   |-- TapUnit.java
|   |                       |   `-- TranslationUnitProperties.java
|   |                       `-- types
|   |                           |-- Function.java
|   |                           |-- QAgg.java
|   |                           |-- QAny.java
|   |                           |-- QArray.java
```

```
|                          |-- QBase.java
|                          |-- QBoolean.java
|                          |-- QByte.java
|                          |-- QDeclared.java
|                          |-- QDouble.java
|                          |-- QEvent.java
|                          |-- QFeature.java
|                          |-- QFloat.java
|                          |-- QGeneric.java
|                          |-- QInteger.java
|                          |-- QLong.java
|                          |-- QPrimitive.java
|                          |-- QQueue.java
|                          |-- QReading.java
|                          |-- QShort.java
|                          |-- QSimple.java
|                          |-- QSpecial.java
|                          |-- QTap.java
|                          `-- QoSSLType.java
```

Der Quellcode ist entsprechend der Konvention zur Organisation von Java-Paketen in einer *Reverse-URL*-Verzeichnisstruktur organisiert, und zwar im Hinblick auf die WWW-Adresse dieses Lehrstuhls in Unterpaketen von `de.lmu.ifi.nm.qossl` in Unterverzeichnissen von `src/de/lmu/ifi/mn/qossl/`. Jeder Unterordner davon entspricht einem Java-Paket bzw. einer Programm-Komponente. Sie werden im folgenden aufgelistet; statt Verzeichnisnotation finden an dieser Stelle Selektionsoperatoren Verwendung:

B.1 Paket `de.lmu.ifi.nm.qossl.aggregation`

Dieses Paket soll die Aggregationsbibliotheken vorhalten. Das Unterpaket mit dem Namen `X` enthält die Aggregationsklassen der Eingabebibliothek `X`. Für unser Beispiel `java` sind bisher nur zwei rudimentäre Klassen vorhanden, die zu Demonstrationszwecken eingesetzt werden können.

B.2 Paket `de.lmu.ifi.nm.qossl.analyzer`

In diesem Paket befinden sich dreierlei Dinge:

- Die Klassen `Symbol`, `SymbolTable` und `StaticSymbolTable` zur Beschreibung der Symboltabelle des Übersetzers,
- die Klassen `Expression` und die `Node`-Klassen für die Nachbildung von *Berechnungsausdrücken* durch Syntaxbäume im *Zwischencode* und
- mehrere `Exception`-Klassen für Parse-Fehler und semantische Fehler.

B.3 Paket `de.lmu.ifi.nm.qossl.base`

In den Unterpaketen dieses Pakets befinden sich die Implementierungen für das Modell des allgemeinen Meßprozesses. Wiederum läßt der Name des Verzeichnisses auf die Eingabebibliothek schließen. Hier werden auch die Schnittstellen für die Basispakete und die Reflektion, also die Ermittlung von Aggregationsfunktionalitäten und Korrelationsfunktionen, festgelegt.

B.4 Paket de.lmu.ifi.nm.qossl.generated

Die Ausgabe der Übersetzung (*Zielcode*) wird in einem Unterpaket des Pakets `de.lmu.ifi.nm.qossl` abgelegt, dessen Name durch die `Namespace`-Direktive festgelegt wird. Weil in unserem Beispiel der Standardwert für diese Direktive, „`generated`“, gesetzt wird, werden die erzeugten Klassen in diesem Paket abgelegt.

B.5 Paket de.lmu.ifi.nm.qossl.out

In diesem Paket befinden sich diejenigen Programm-Komponenten, die zur Reflektion und Code-Erzeugung dienen und entsprechend Reflektor- und Generator-Interface zu implementieren haben. Letzteres ist hier in Gestalt der Klasse `Generator` abgelegt. Das Unterpaket `generic` beinhaltet die Hilfsklasse `CodeWriter`; sie bietet einen einfachen Mechanismus zum Öffnen und Schließen von Ausgabedateien und erzeugt deren Kopf- und Fußzeilen.
Die Reflektor- und Generator-Implementierung zu einer Eingabebibiliothek mit dem Namen `X` sollen in einem Unterpaket mit eben diesem Bezeichner `X` abgelegt werden. Die mitgelieferte Beispielimplementierung befindet sich entsprechend im Unterverzeichnis `java`.

B.6 Paket de.lmu.ifi.nm.qossl.parser

Dieses Paket enthält das mit JavaCC zu erzeugende Parser-Programm. Der Code dafür wird in der Klasse `QoSSLParser` abgelegt. Die übrigen Klassen dieses Pakets beinhalten den Token-Manager und die generische Ausnahme `ParseException` für Syntaxfehler. Die Liste der durch den Generator erzeugten Dateien ist:

```
|-- src
|   `-- de
|       `-- lmu
|           `-- ifi
|               `-- nm
|                   `-- qossl
|                       |-- ...
|                       |-- parser
|                       |   |-- JavaCharStream.java
|                       |   |-- ParseException.java
|                       |   |-- QoSSLParser.java
|                       |   |-- QoSSLParserConstants.java
|                       |   |-- QoSSLParserTokenManager.java
|                       |   |-- Token.java
|                       |   `-- TokenMgrError.java
|                       |-- ...
```

B.7 Paket de.lmu.ifi.nm.qossl.semantics

Instanzen dieser Klassen dienen zur Modellierung des Zwischencodes als Abstraktion der QoSSL-Spezifikation auf ein Objektsystem. Dies sind im einzelnen:

- Die Klasse `QoSSLTranslationUnit`, die die Gesamtheit des Zwischencodes repräsentiert und für jede Spezifikation genau einmal instanziert wird,

- die Klasse `TranslationUnitProperties`, die eine Reihe von Eigenschaften und Parametern des vorliegenden Zwischencodes kapselt und
- die Unterklassen von `QoSSLProcessUnit`, welche die einzelnen Komponenten der gegebenen Spezifikation abstrahieren.

Eine ausführliche Diskussion gibt es im Kapitel 6.10.

B.8 Paket `de.lmu.ifi.nm.qossl.types`

Dieses Paket definiert die Typen der Sprache QoSSL, deren Hierarchie sich aus der der zugehörigen Klassen ableitet. Die Klasse `QoSSLType` ist dabei kein Typ, sondern dient zur Spezifikation deren Verhaltens. Mit Instanzen der Klasse `Function` kann die Signatur von (Korrelations-)Funktionen spezifiziert werden.
Das Kapitel 6.5 bietet eine ausführliche Dokumentation.

B.9 Beispielspezifikationen

Dem Anwender werden von seiten dieser Arbeit eine Reihe von QoSSL-Beispielspezifikationen mitgegeben, die die unterschiedlichen Möglichkeiten dieses Formalismus beleuchten. Sie sind allesamt im Verzeichnis `./qossl/` abgelegt.

- Die Spezifikation aus der Datei `anhang_a4.qossl` entspricht dem Beispiel aus Anhang A.4 aus [Gars 04], wobei die Syntax an die in dieser Arbeit vorgestellten Regeln angepaßt worden ist.
- Das Beispiel `eth_faulty_frame_rate.qossl` ist ebenfalls aus [Gars 04] übernommen und soll die Rahmenfehlerrate bei Ethernet messen. Dabei kommen zwei verschiedene Warteschlangen zum Einsatz, die miteinander korreliert werden.
- Aus der gleichen Arbeit stammt auch die Datei `icmp_echo.qossl`. Hier wird die *ICMP-Laufzeit-Verzögerung (ICMP Round Trip Delay)* gemessen, wobei an dieser Stelle die Einbeziehung des Faktors „Zeit“ zu sehen ist.
- Das Beispiel `url_time.qossl` besitzt einen ähnlichen Hintergrund, doch dieses Mal geht es um die Anfrage an einen `URL`.
- Das fünfte Beispiel von [Gars 04], `response_time.qossl`, dient ganz allgemein zur Messung der Antwortzeit eines Dienstes.
- Die Dateien `throughput.qossl` und `throughput_body.qossl` enthalten das Beispiel, das im Rahmen dieser Arbeit besprochen wird.
- Aus dem Inhalt von `socket.qossl`, `socket_data.qossl`, `http.qossl` und `telnet.qossl` setzt sich eine sehr umfangreiche Spezifikation zusammen, die exzessiven Gebrauch von den vielfältigen Möglichkeiten von QoSSL macht. Die Hauptdatei ist dabei `socket.qossl`.

B.10 Weitere Dateien und Verzeichnisse

Der übersetzte und ablauffähige Compiler wird derselben Struktur folgend im Verzeichnis `./bin/` abgelegt. Die erwähnte HTML-Dokumentation [HTML] befindet sich ebenso angeordnet im Verzeichnis `./doc/`; Einstiegspunkt ist die Datei `index.html`. Weiterhin sind folgende Dateien zu erkennen:

- `README`: Schnelldokumentation zur Anwendung, siehe auch Anhang C
- `generate.sh`: Installations- und Anwendungsskript; Hilfe gibt es mit der Option `-h`
- `src/de/lmu/ifi/nm/qossl/overview.html`: diese Dokumentation in HTML, wie sie durch einen Aufruf von `javadoc` in [QoSH] eingebettet werden kann

- `src/de/lmu/ifi/nm/qossl/qossl_stylesheet.css` CSS-Stylesheet zur HTML-Dokumentation
- `src/de/lmu/ifi/nm/qossl/qossl.jj`: QoSSL-Grammatik und Parser-Spezifikation

C Handhabung des Übersetzers

Dieser Abschnitt stellt die „Bedienungsanleitung“ des vorliegenden Programms dar. Er liegt in abgewandelter Form als `README`-Datei im Quellverzeichnis vor.

C.1 Voraussetzungen

Voraussetzungen für den Betrieb dieses Programm sind:

- Der GCC-Compiler, bisher sind 3.2 und 3.3.5 erfolgreich getestet worden.
- Ein Java-SDK mit Java-Laufzeitumgebung; erfolgreich getestet sind 1.4.0_00, 1.4.2_xx und 1.5.0_0x, jeweils von *Sun Microsystems*
- Ein Web-Browser nach Wahl zur Betrachtung der HTML-Dokumentation [QoSH].
- Linux mit GNU Bourne-Again Shell und dem Coreutils-Paket; erfolgreich getestet ist 5.3.0-10.2
- Wenn der Parser aus der QoSSL-Grammatikdatei erzeugt werden soll, zusätzlich der Parser-Generator JavaCC, Version 3.2 oder höher. Sein Pfad sollte in der `PATH`-Variable eingetragen sein.

C.2 Installation und Anwendung

Die Installation des Übersetzers und die Anwendung kann mit dem mitgelieferten Skript `generate.sh` erfolgen. Anhand der zur Verfügung stehenden Schalter hat der Anwender die Möglichkeit, verschiedene Routinen auszulösen. Durch einen Aufruf mit der Option `--help` erhält man detaillierte Informationen.
Beginnend mit dem Auslieferungszustand ruft man dieses Skript zunächst mit den Optionen `--build-all` auf; damit sind Übersetzer und Dokumentation einsatzbereit. Möchte man nun eine QoSSL-Spezifikation `S` verarbeiten lassen, so ist zuerst sicherzustellen, daß sie in einer Datei namens `S.qossl` im Verzeichnis `./qossl/` vorliegt. Zur Erzeugung der Definitionssicht aus der Entwicklersicht wird `generate.sh` mit der Option `--parse-spec=S` aufgerufen; soll das Meßsystem erzeugt werden, so ist `--build-spec=S` zu wählen.
Mit der Option `--reset` kann das Programm in den Auslieferungszustand zurückversetzt werden. Außerdem ist die isolierte Abarbeitung von Einzelschritten der Installation und die Behandlung eigener Erweiterungen möglich.

C.3 Weiterführende Dokumentation

Nach einem Aufruf von `./generate.sh --build-doc` steht im Verzeichnis `./doc/` die mittels `javadoc`-Dokumentationsgenerator erstellte Dokumentation im HTML-Format [HTML] unter [QoSH] zur Verfügung. Der Einstiegspunkt für den Web-Browser ist die Datei `index.html`.
Es existiert *keine GNU-Handbuchseite*.
Die Bereitstellung von Erweiterungen ist in Anhang D beschrieben.

D Bereitstellung von Meßprozessen und Aggregationsfunktionalitäten

In dieser Arbeit wird ausgiebig diskutiert, daß Meßprozeß, Code-Erzeuger und Aggregationsbibliothek, also die sogenannte *Erweiterung*, austauschbar sind. An dieser Stelle sei ein Leitfaden zur Bereitstellung alternativer Implementierungen gegeben, wobei dies anhand der in dieser Arbeit mitgelieferten Komponenten beschrieben wird.

D.1 Pakete

Konkret geht es darum, die im Diagramm 5.1 auf der rechten Seite dargestellten Komponenten bereitzustellen bzw. die in dieser Arbeit mitgelieferten Versionen zu ersetzen. Dabei werden der Übersetzer und seine Schnittstellenklassen als vorhanden vorausgesetzt. Das Design verfolgt das Ziel, daß eventuell schon vorhandenene Erweiterungen nicht entfernt zu werden brauchen.
Steht man nun vor der Aufgabe, eine Erweiterung bereitzustellen, so bedarf es zu ihrer Unterscheidung zunächst eines eindeutigen Bezeichners. Damit werden dann die für Meßprozeß, Code-Erzeuger und Aggregationsbibliothek zu erstellenden Unterordner benannt. Weiterhin muß in der jeweiligen Implementierung auf eine geeignete Verwendung dieses Bezeichners geachtet werden; in der vorliegenden Erweiterung bezeichnet er etwa das entsprechende Unterpaket innerhalb der Implementierung.

```
.
|-- README
|-- generate.sh
|-- src
|   `-- de
|       `-- lmu
|           `-- ifi
|               `-- nm
|                   `-- qossl
|                       |-- aggregation
|                       |   |-- java
|                       |   |   |-- AggregationRepository.java
|                       |   |   |-- Boxplot.java
|                       |   |   `-- Sum.java
|                       |   `-- muster
|                       |       `-- [XXX]
|                       |-- analyzer
|                       |   |-- BinaryNode.java
|                       |   |-- ...
|                       |   `-- UnaryNode.java
|                       |-- base
|                       |   |-- BaseProperties.java
|                       |   |-- Reflector.java
|                       |   |-- java
|                       |   |   |-- Builder.java
|                       |   |   |-- ...
|                       |   |   `-- Tap.java
|                       |   `-- muster
|                       |       `-- [XXX]
```

```
|                       |-- generated
|                       |-- out
|                       |   |-- Generator.java
|                       |   |-- generic
|                       |   |   `-- CodeWriter.java
|                       |   |-- java
|                       |   |   |-- GeneratorImpl.java
|                       |   |   |-- ReflectorImpl.java
|                       |   |   `-- ...
|                       |   `-- muster
|                       |   |   |-- GeneratorImpl.java
|                       |   |   |-- ReflectorImpl.java
|                       |       `-- [XXX]
|                       |-- parser
|                       |   |-- JavaCharStream.java
|                       |   |-- ...
|                       |   `-- TokenMgrError.java
|                       |-- qossl.jj
|                       |-- semantics
|                       |   |-- AggregationUnit.java
|                       |   |-- ...
|                       |   `-- TranslationUnitProperties.java
|                       `-- types
|                           |-- Function.java
|                           |-- QAgg.java
|                           |-- ...
|                           |-- QTap.java
|                           `-- QoSSLType.java
```

In unserer Erweiterung wird der Bezeichner `java` verwendet. Zur besseren Übersichtlichkeit ist im Verzeichnisbaum auf gleiche Weise ein weiterer Bezeichner `muster` eingefügt worden. Die Stellen, an denen die Komponenten eingefügt werden sollen, sind zudem mit `[XXX]` markiert worden.
Grundsätzlich gilt: Die Generator-Komponente mit den Klassen `ReflectorImpl` und `GeneratorImpl` muß in der Implementierungssprache des Übersetzers bereitgestellt werden. Es muß also Klassen mit diesem Namen im entsprechenden Unterpaket geben. Alle anderen Komponenten sind von seiten dieser Arbeit keinen weiteren Reglementierungen unterworfen.

D.2 Aggregationsfunktionalitäten

Die Bereitstellung von Aggregationsfunktionalitäten geschieht denkbar einfach. Im zuvor vereinbarten Unterpaket `de.lmu.ifi.nm.qossl.aggregation` sollen sie so abgelegt werden, daß sie die zugehörige Reflektor-Implementierung vom Namen her unterscheiden kann. Diese Namen haben den Konventionen zu Bezeichnern in Programmiersprachen zu folgen, weil sie innerhalb der Spezifikation als solche auftauchen. Es gibt jedoch keinen Zwang, sich auf eine bestimmte Implementierung festzulegen.
In unserem Fall werden Aggregationen durch Java-Klassen realisiert. Die Schnittstellenklasse `AggregationRepository` dient zur Definition der Mindestanforderungen; als Beispiel für eine Aggregationsklasse kann in unserem Fall jede Klasse dienen, die obiges Interface implementiert; die Bereitstellung der tatsächlichen Funktionalität ist ohnehin nicht Bestandteil dieser Arbeit.
Alternativ zu solchen Klassen wäre eine Integration in den allgemeinen Meßprozeß oder eine Bereitstellung durch XML-Dokumente [XML] oder gar ASCII-Dateien denkbar. Ausschlaggebend ist einzig und allein die Tatsache, daß sich die Bezeichner der Aggregationsklassen mit der Reflektor-Implementierung jener Erweiterung zuverlässig erkennen lassen.

D.3 Meßprozeß

Das Modell des allgemeinen Meßprozesses wird aus [Gars 04] übernommen. Es kann jedoch grundsätzlich unabhängig vom Übersetzer entwickelt werden. Dabei ist auch die Wahl der Programmiersprache im Grunde frei; nur der Ablageordner soll unterhalb von `./de/lmu/ifi/nm/qossl/base/` liegen und vom Bezeichner her korrespondieren; in unserem Fall ist also `./de/lmu/ifi/nm/qossl/base/java` die richtige Wahl. Ein weiteres wichtiges Kriterium zur Entwicklung des Meßprozesses ist wie schon bei den Aggregationen, daß die zugehörige Reflektor-Implementierung damit umgehen können muß; das Modell soll ja beispielsweise auf Korrelationsfunktionen hin untersucht werden. Außerdem hat der Übersetzer ein funktionierendes, davon abgeleitetes, spezifisches Modell zu erzeugen.

D.4 Generator-Paket: Reflektor

Das bereitzustellende Generator-Paket hat der Wahl des Bezeichners der Erweiterung in Anhang D.1 zufolge `de.lmu.ifi.nm.qossl.out.java` zu heißen. Die Implementierung der Klassen `ReflectorImpl` und `GeneratorImpl` hat im Gegensatz zum Meßprozeß in der Programmiersprache des Compilers — Java [GJSB 00] — zu erfolgen, damit sie zu gegebener Zeit instanziert und damit überhaupt aktiviert werden können.
Die Reflektor-Klasse implementiert das Interface `de.lmu.ifi.nm.qossl.base.Reflector`. Die Korrelationsfunktionen werden mittels *Java-Reflektion* [Ulle 06] innerhalb des Basis-Pakets ermittelt; die Menge der Aggregationsklassen ist in unserem Beispiel als die Menge der `*.class`-Dateien im entsprechenen Binary-Paket definiert; mit Ausnahme des Interface `AggregationRepository`, versteht sich. Der Code ist stark gekürzt dargestellt, um das Schema der Implementierung deutlich zu machen.

```
package de.lmu.ifi.nm.qossl.out.java;

import de.lmu.ifi.nm.qossl.base.java.JavaProperties;
import de.lmu.ifi.nm.qossl.base.Reflector;
import de.lmu.ifi.nm.qossl.types.Function;

import java.io.File;
import java.io.FileFilter;
import java.util.Vector;
import java.lang.reflect.Method;
/* [...] */

public class ReflectorImpl
implements Reflector {

    public Function[] discoverNumQueueFunctions () {
        Function[] ret = null;
        try {
            Class queueClass = Class.forName (
              JavaProperties.BASE_PACKAGE + JavaProperties.NUM_QUEUE_CLASS);
              /* ("de.lmu.ifi.nm.qossl.base." + "java.") + "Queue" */

            Method[] methods = queueClass.getMethods ();

            Vector functions = new Vector (0, 1);

            for (int i = 0; i < methods.length; i++) {
                Method m = methods[i];
                int mod = m.getModifiers();
                if (/* Signatur passend*/) {
                    functions.add (new Function (...));
```

```
                }
            }

            /* Function-Objekte von functions nach ret kopieren */
            /* [...] */
        } catch (ClassNotFoundException cnfe0) {
        }
        return ret;
    }

    public Function[] discoverEventQueueFunctions () {
    }   // Implementierung analog zu oben

    public Function[] discoverNumEventFunctions () {
    }   // Implementierung analog zu oben

    public String[] discoverAggregationClasses () {

        File aggDir = new File (/* Verzeichnis von "aggregation.java" */);
        File[] aggFileList = aggDir.listFiles(/* nur *.class-Dateien */);

        String[] aggClassNames = new String[aggFileList.length];

        for (int r = 0; r < aggFileList.length; r++) {
            String s = /* Klassenname von aggFileList[r] */ ;
            aggClassNames[r] = s;
        }
        return aggClassNames;
    }
}
```

Der Parser wird zu Beginn des ersten Laufs diese vier Methoden aufrufen, um die notwendigen Informationen zu erhalten. Von der Klasse `ReflectorImpl` hängt es also ab, welche Korrelationsfunktionen und Aggregationsklassen dem Parser bekannt und somit innerhalb einer Spezifikation gültig sind. Es wird deutlich, daß die Implementierung von `ReflectorImpl` im Prinzip frei ist, sofern die Methoden sinnvolle Werte im Hinblick auf Meßprozeß und Spezifikationsrichtlinien liefern. Das zugehörige Code-Fragment in der Klasse `QoSSLParser` sieht so aus:

```
private void discoverFunctions (TranslationUnitProperties prop) {
    String language = prop.getLanguage ();
    try {
        /* Suchen */
        Class reflectorClass = Class.forName (
          BaseProperties.OUT_FOLDER +       /* de.lmu.ifi.nm.qossl.out. */
          language.toLowerCase () +         /* java */
          "." +
          BaseProperties.REFLECTOR_NAME);   /* ReflectorImpl */

        /* Instanzierung */
        Reflector refl = (Reflector)(reflectorClass.newInstance());
        Function[] functions = null;

        functions = refl.discoverNumQueueFunctions ();
        /* [...] */

    }  catch (Exception ex) {
    }
}
```

Es wird also versucht, eine Klasse zu finden und zu instanzieren, die folgende Eigenschaften hat:

- Sie befindet sich im Ordner `BaseProperties.OUT_FOLDER`.
- Der Unterordner heißt wie in der *Language-Direktive* angegeben.
- Der Basisname ist `BaseProperties.REFLECTOR_NAME`.

Anschließend werden die oben besprochenen Methoden aufgerufen und die Ergebnisse in der Symboltabelle des Übersetzers und im Zwischencode gespeichert.

D.5 Generator-Paket: Generator

Der zweite Lauf der Übersetzung wird dadurch initiiert, daß auf gleiche Weise wie oben versucht wird, die Klasse `GeneratorImpl` zu instanzieren.

```
private void generateTargetCode (QoSSLTranslationUnit unit) {
    TranslationUnitProperties prop = unit.getProperties ();
    String language = prop.getLanguage ();
    try {
        /* Suchen */
        Class generatorClass = Class.forName (
          BaseProperties.OUT_FOLDER +       /* de.lmu.ifi.nm.qossl.out. */
          language.toLowerCase () +         /* java */
          "." +
          BaseProperties.GENERATOR_NAME);   /* GeneratorImpl */

        /* Instanzierung */
        Generator gen = (Generator)(generatorClass.newInstance());

        /* Code-Erzeugung */
        gen.generateTargetCode (unit);
    } catch (Exception ex) {
    }
}
```

Die Generator-Klasse implementiert das Interface `de.lmu.ifi.nm.qossl.base.Generator`. Beim Aufruf der Methode `generateTargetCode` wird das `QoSSLTranslationUnit`-Objekt übergeben, das den Zwischencode repräsentiert. Dieses kann nun mittels entsprechender Methoden systematisch zur Erzeugung des Zielcodes herangezogen werden. Wiederum ist die Art der Implementierung freigestellt; das Programm darf, drastisch formuliert, auch „Hallo Welt!“ liefern, was jedoch im Hinblick auf die Aufgabenstellung nicht gerade zweckgemäß ist.

```
package de.lmu.ifi.nm.qossl.out.java;

import de.lmu.ifi.nm.qossl.out.Generator;
import de.lmu.ifi.nm.qossl.semantics.*;

import java.util.Enumeration;
/* [...] */

public class GeneratorImpl implements Generator {

    public void generateTargetCode (QoSSLTranslationUnit unit){
        TranslationUnitProperties prop = unit.getProperties ();

        /* Zielverzeichnis erzeugen */
```

```
            /* [...] */

            Enumeration enu;

            /* fuer jede Primitve entsprechende Dateien anlegen */
            enu = unit.getPrimitives ();
            while (enu.hasMoreElements () ) {
                PrimitiveUnit pu = (PrimitiveUnit)(enu.nextElement ());
                Primitive_JavaFileGen fgen = new Primitive_JavaFileGen (pu);
                fgen.generateCodeFiles (prop);
            }

            /* fuer jedes Ereignis entsprechende Dateien anlegen */
            enu = unit.getEvents ();
            while (enu.hasMoreElements () ) /* ... */

            /* usw. */
        }

    }
```

In der vorliegenden Implementierung wird das übergebene `QoSSLTranslationUnit`-Objekt nach Unterinstanzen von `QoSSLProcessUnit` abgefragt. Dann wird für jede dieser Komponenten dieses Zwischencodes einmal die zugehörige `CodeFileGenerator`-Unterklasse instanziert und die Ausgabe des Zielcodes angestoßen.
Sind nun alle notwendigen Komponenten bereitgestellt, so kann mit dem Aufruf von `generate.sh` mit der Option `--build-extension=java` usw. die Erweiterung betriebsbereit gemacht werden.

Glossar

Aggregationsbibliothek Sammlung der zur Verfügung stehenden →Aggregationsklassen.

Aggregationsfunktionalität Eine (dem vorliegenden Programm zur Verfügung stehende) Art und Weise der Verarbeitung von →Meßwerten zu gemessener →Dienstgüte und ihrer Darstellung.

Aggregationsklasse Realisierung einer →Aggregationsfunktionalität, die der Übersetzer mit einer geeigneten Implementierung der →Reflektor-Schnittstelle erkennen kann.

Aggregation Bezeichnet die statistische Nachverarbeitung von ermittelten →Meßwerten zu gemessener →Dienstgüte. Wird auch als Synonym für →Aggregationsklasse benutzt.

Anwender Personengruppe, die das vorliegende Programm zur Erzeugung von →Meßsystemen aus →QoSSL-Spezifikationen benutzt.

Attribute (einer Primitive) Parameter einer modellierten →Dienstprimitive.

Basispaket Programmkomponente, die die Implementierung des generischen →Meßprozesses enthält, wovon ein erweitertes Modell abgeleitet werden soll.

Berechnungsausdruck Ein Ausdruck innerhalb der →QoSSL-Spezifikation, der kein Typisierungsausdruck ist und dessen →Typ insbesondere kein →Spezialtyp ist.

Code-Erzeuger Synonym für →Generator.

Code-Erzeugung Der auf das →Parsen der →QoSSL-Spezifikation folgende Schritt der Ausgabe des →Zielcodes als Ergebnis der Umformung; gleichzusetzen mit dem zweiten Lauf der →Übersetzung.

Definitionssicht Bezeichnet den in der Bereitstellungsphase eines Dienstes vorliegenden Implementierungsleitfaden.

Deklaration Element der →QoSSL-Spezifikation, mit denen die →Komponenten des parametrisierten →Meßprozesses vereinbart werden. Sie können als Typisierungsausdrücke für →Spezialtypen mit spezieller Syntax betrachtet werden.

deklarieren (eines Bezeichners) Definition dieses Bezeichners im Rahmen einer →Deklarartion, wobei gleichzeitig die Eigenschaften dieser Entität festgelegt werden.

Dienstgüte, gemessene Steht am Ende des (generischen oder parametrisierten) →Meßprozesses als dessen Ergebnis, welches durch →Aggregation der ermittelten →Meßwerte entsteht; entspricht ‘Quality of Service’.

Dienstgütemerkmal Synonym für gemessene →Dienstgüte.

Dienstprimitive Bezeichnet allgemein Operationen, mit denen ein Benutzerprozeß auf einen Dienst zugreifen kann (Dienstschnitt im Sinne des OSI-Schichtenmodells). Steht im Rahmen dieser Arbeit auch für deren Modellierung bezogen auf den in [Gars 04] vorgestellten →Meßprozeß.

Dienstzugangspunkt Bezeichnet die explizite Schnittstelle eines Dienstes (Dienstschnitt) im Sinne des OSI-Schichtenmodells.

Direktive Sprachelement von QoSSL, das zur Steuerung der →Übersetzung dient.

Entwicklersicht Bezeichnet die in der Verhandlungsphase des Dienstlebenszyklus vorliegende Dienstspezifikation.

Ereignis vom Typ x Dies ist als '→Ereignis der →Ereignisklasse x' zu verstehen.

Ereignisklasse Eine durch spezifische Belegung der →Attribute der zugehörigen →Dienstprimitive vom jeweiligen →Anwender definierte Art von →Ereignissen, die an einem bestimmten →Dienstzugangspunkt auftreten können.

Ereignis Bezeichnet entweder den tatsächlich registrierten Aufruf eines Dienstes an seinem →Dienstzugangspunkt oder stellvertretend eine Instanz einer →Ereignisklasse.

Erweiterung Bezeichnet einen zusammengehörigen Satz von Programmkomponenten, jeweils bestehend aus →Basispaket, →Korrelations- und →Aggregationsbibliothek sowie →Generator-Paket.

EventGivingQueueFunctions Bezeichnung für →Korrelationsfunktionen, die auf →Warteschlangen angewendet werden können und ein →Ereignis zurückgeben.

Event Die in diesem Zusammenhang verwendete englische Bezeichnung für →Ereignis.

Feature Die in diesem Zusammenhang verwendete englische Bezeichnung für die gemessene →Dienstgüte.

Filterregel Boolescher Ausdruck innerhalb einer →Ereignis-Deklaration, mit dem der korrespondierende Attributwert (→Attribut) der zugehörigen →Dienstprimitive auf Übereinstimmung hin überprüft wird, um dieses Ereignis auszulösen.

Filtervorschrift Sequenz von →Filterregeln, wie sie innerhalb einer →Ereignis-Deklaration vorkommt; spezifiziert die Gesamtheit der Bedingungen für die Generierung eines derartigen Ereignisses.

Funktion Bezeichnet in entsprechendem Kontext eine →Korrelationsfunktion.

Generator-Implementierung Realisierung der →Generator-Schnittstelle im Rahmen einer →Erweiterung.

Generator-Paket Programm-Komponente, die im Rahmen einer →Erweiterung bereitgestellt wird; besteht aus →Reflektor-Implementierung und →Generator-Implementierung.

Generator-Schnittstelle Beschreibt die Schnittstelle des Programms für eine →Erweiterung, die für die →Code-Erzeugung verantwortlich ist.

Identifikatormodus Kennzeichnet die Art und Weise, wie der eindeutige Identifikator eines →Ereignisses bezogen auf seine →Ereignisklasse festgelegt wird.

Komponente Kontextabhängig bezeichnet dieser Begriff entweder eine →Programmkomponente im Sinne der →Systemarchitektur oder einen Bestandteil des →Meßprozesses bzw. der zugehörigen →QoSSL-Spezifikation.

Korrelationsfunktion Bezeichnet in entsprechendem Kontext eine vom allgemeinen →Meßprozeß bereitgestellte Funktion zur →Korrelation von →Ereignissen; Einteilung in →NumGivingEventFunctions, →NumGivingQueueFunctions und →EventGivingQueueFunctions.

Korrelation Statische Berechnung eines →Meßwerts aus →Ereignissen; dazu stehen spezielle →Korrelationsfunktionen zur Verfügung.

Laufzeit Bezeichnet normalerweise die Laufzeit des vorliegenden Programms, was der Übersetzungszeit einer →QoSSL-Spezifikation in die materialisierte →Definitionssicht gleichkommt.

Meßprozeß, allgemeiner oder generischer Auf Basis eines generischen Dienstmodells entwickelter Prozeß für die Messung (Bestimmung) von →Dienstgüte, der durch Parametrisierung für bestimmte Anwendungen spezifiziert werden kann.

Meßprozeß, parametrisierter Bezeichnet den aus dem generischen →Meßprozeß mittels Bereitstellung einer →QoSSL-Spezifikation abgeleiteten, für die darin festgelegten →Dienstgütemerkmale spezifischen Meßprozeß.

Meßsicht Bezeichnet die Festlegung eines ablauffähigen →Meßsystems in der Nutzungsphase des Dienstes.

Meßsystem Bezeichnet das gewünschte Ergebnis der Übersetzung einer →QoSSL-Spezifikation; dies ist ein i. a. kompilierbare und anschließend ablauffähiges Programm zur Messung der →Dienstgüte an einem →SAP, die die →Meßsicht realisiert.

Meßwert Ergebnis von Berechnungen aus den Eigenschaften korrelierter →Ereignisse. (→Korrelation)

Nachverarbeitung, statistische Synonym für →Aggregation.

NumGivingEventFunctions Bezeichnung für →Korrelationsfunktionen, die auf →Ereignisse angewendet werden können und einen integralen Wert zurückgeben.

NumGivingQueueFunctions Bezeichnung für →Korrelationsfunktionen, die auf →Warteschlangen angewendet werden können und einen integralen Wert zurückgeben.

Parsen Parsen der gegebenen →QoSSL-Spezifikation im herkömmlichen Sinne; gleichzusetzen mit dem ersten Lauf des Übersetzers; synonym auch 'Parse-Vorgang' oder 'Parsing'.

Primitive Synonym für →Dienstprimitive.

Programmierer (einer Erweiterung) Personengruppe, die das vorliegende Programm z. B. durch weitere →Aggregationsfunktionalitäten erweitert.

QoSSL-Spezifikation Materialisierte →Entwicklersicht, niedergeschrieben als Sequenz von →Deklarationen in der Sprache QoSSL.

QoSSL-Typ →Typ.

Quellcode Die Eingabe des Übersetzers in Form der materialisierten →Entwicklersicht.

Reading Die in diesem Zusammenhang verwendete englische Bezeichnung für →Meßwert.

Referenzobjekt (eines Symbols) Der semantische →Repräsentant aus Sicht der Symboltabelle.

Reflektor-Implementierung Realisierung der →Reflektor-Schnittstelle im Rahmen einer →Erweiterung.

Reflektor-Schnittstelle Beschreibt die Schnittstelle des Programms für eine →Erweiterung, die für die Bestimmung von →Korrelationsfunktionen und →Aggregationsfunktionalitäten verantwortlich ist.

Repräsentant, semantischer Bezeichnet diejenige →Komponente der semantischen →Repräsentation, die mit einem bestimmten →Spezialtyp oder mit einer Instanz dieses →Typs korrespondiert.

Repräsentation, semantische Bezeichnet die Darstellung des Inhalts von →Deklarationen als Komponenten des →Meßprozesses im Zwischencode in Form eines Objektsystems.

SAP Abkürzung für Service Access Point (→Dienstzugangspunkt).

Spezialtyp Bezeichnet diejenigen Typen von QoSSL, die Komponenten des →Meßprozesses darstellen und daher eine semantische →Repräsentation haben.

Spezifikation, gegebene oder vorliegende →QoSSL-Spezifikation.

Typ, bekanntgemachter Dies ist ein Bezeichner, der durch eine `import`-Anweisung gegenüber dem Übersetzer als →Typ vereinbart wird.

Typ, einfacher/simpler Bezeichnet einen Zahlentyp oder den booleschen →Typ von QoSSL.

Typ, eingebauter Bezeichnet einen →Typ, der in QoSSL fest enthalten (d. h. bereits implementiert) ist; eingebaute T. sind die einfachen →Typen, die Verbund- und die →Spezialtypen.

Typ Bezeichnet einen Typ der Sprache QoSSL im herkömmlichen Sinne, sofern nicht anders vermerkt.

Vereinbarung Synonym für →Deklaration.

verwenden (eines Bezeichners) Bezeichnet das Vorkommen dieses Bezeichners im →Quellcode außerhalb einer →Vereinbarung, die seine Eigenschaften festlegt.

Warteschlange Bezeichnet die FIFO-Warteschlange zur Pufferung von →Ereignissen zu ihrer statistischen →Nachverarbeitung.

Zielcode Bezeichnet die materialisierte →Definitionssicht als Ergebnis der →Code-Erzeugung.

Übersetzungseinheit Abstraktum für die Gesamtheit der Informationen, die der Übersetzer zur →Laufzeit als Eingabe erhält und verarbeitet. (→Quellcode, →Spezifikation)

Übersetzung Bezeichnet die durch das vorliegenden Programm wohldefinierte Umformung einer →QoSSL-Spezifikation in ein Programm bzw. die Generierung der →Definitionssicht aus der →Entwicklersicht.

Literaturverzeichnis

[Bry 02] BRY, FRANÇOIS: *Einführung in Algorithmen und in die Programmierung. Skriptum der Vorlesung Informatik I, 2001, 2002*, 2002, http://www.pms.ifi.lmu.de/publikationen/lecture-notes/info1/www-Info1-Skriptum-2002.ps.gz .

[Bry 04] BRY, FRANÇOIS: *Übersetzerbau – Abstrakte Maschinen*, 2004, http://www.pms.ifi.lmu.de/publikationen/lecture-notes/uebersetzerbau/www-skriptum-2004.pdf .

[Corm 01] CORMEN, THOMAS H., CHARLES E. LEISERSON, RONALD L. RIVEST und CLIFFORD STEIN: *Introdction to Algorithms*. MIT Press London, England, ISBN 0-262-03293-7, Zweite Auflage, 2001.

[ECMA-334] *C# Language Specification. Standard ECMA-334, Third Edition*. European Computer Manufacturers Association, 2005, http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf .

[Gars 04] GARSCHHAMMER, MARKUS: *Dienstgütebehandlung im Dienstlebenszyklus – von der formalen Spezifikation zur rechnergestützten Umsetzung*. Dissertation, Ludwig-Maximilians-Universität München, Juli 2004, http://www.nm.ifi.lmu.de/pub/Dissertationen/gars04a/PDF-Version/gars04a.pdf .

[GCC 03] STALLMAN, RICHARD M.: *Using the GNU Compiler Collection (for GCC Version 3.3.6)*. Free Software Foundation Inc., 2003, http://gcc.gnu.org/onlinedocs/gcc-3.3.6/gcc.ps.gz .

[GJSB 00] GOSLING, J., B. JOY, G. STEELE und G. BRANCHA: *The Java Language Specification*. Addison-Wesley Professional, Reading, Mass., ISBN 0-201-31008-2, Zweite Auflage, 2000.

[GuSo 00] GUMM, HEINZ-PETER und MANFRED SOMMER: *Einführung in die Informatik*. Oldenbourg Verlag, München, ISBN 3-486-245050-7, Vierte Auflage, 2000.

[HaMe 02] HAROLD, ELLIOTTE RUSTY und W. SCOTT MEANS: *XML n a Nutshell*. O'Reilly & Associates, Sebastopol, Kalif. ISBN 0-596-00292-0, Zweite Auflage, 2002.

[HAN 99] HEGERING, H.-G., S. ABECK und B. NEUMAIR: *Integrated Management of Networked Systems – Concepts, Architectures and their Operational Application*. Morgan Kaufmann Publishers, ISBN 1-55860-571-1, 1999. 651 p.

[Henn 03] HENNICKER, ROLF: *Objektorientierte Software-Entwicklung*, 2003, http://www.pst.ifi.lmu.de/lehre/WS0304/oose/ .

[Henn 04] HENNICKER, ROLF: *Formale objektorientierte Software-Entwicklung*, 2004, http://www.pst.ifi.lmu.de/lehre/SS04/formale-oo/ .

[HiKa 03] HITZ, MARTIN und GERTI KAPPEL: *UML@Work*. dpunkt.verlag, Heidelberg, ISBN 3-89864-194-5, Zweite Auflage, 2003.

[HTML] RAGGETT, D., A. LE HORS und I. JACOBS: *HTML 4.01 Specification*, 1999, http://www.w3.org/TR/html401 .

[ISO 14977] *Extended BNF. ISO/IEC 14977 (final draft version SC22/N2249)*. International Organiszation for Standardization and International Electrotechnical Committee, 1996, http://www.cl.cam.ac.uk/ mgk25/iso-14977.pdf .

[JCC] JAVA TECHNOLOGY COLLABORATION JAVA.NET THE SOURCE FOR: *Java Compiler Compiler*, https://javacc.dev.java.net .

[JCCQ 04] NORVELL, THEODOR S.: *The JavaCC FAQ*. Technischer Bericht, Computer and Electrical Engineering Memorial University of Newfoundland, 2004, http://www.engr.mun.ca/ theo/JavaCC-FAQ/ .

[KeRi 90] KERNIGHAN, BRIAN W. und DENNIS M. RITCHIE: *Programmieren in C*. Carl Hanser Verlag (sic!), München, ISBN 3-446-15497-3, Zweite Ausgabe, ANSI C Auflage, 1990.

[MNM 02] GARSCHHAMMER, MARKUS, RAINER HAUCK, HEINZ-GERD HEGERING, BERNHARD KEMPTER, IGOR RADISIC, HARALD RÖLLE und HOLGER SCHMIDT: *A Case-Driven Methology for Applying the MNM Service model*. In: STADLER und ULEMA (Herausgeber): *Proceeding of the 8th International IFIP/IEEE Network Operations and Management Symposium (NOMS 2002)*, Seiten 697–710. IFIP/IEEE, IEEE Publishing, 2002, http://wwwmnmteam.informatik.uni-muenchen.de/php-bin/pub/show_pub.php?key=ghkr01 .

[OCL 03] (OMG), THE OBJECT MANAGEMENT GROUP: *UML 2.0 OCL Specification*. Adopted Specification, 2003, http://www.omg.org/cgi-bin/doc?ptc/2003-10-14 .

[QoSH] SCHAAL, JOHANNES: *QoSSL-Compiler – Programmdokumentation*. Java API documentation generator, 2006, ./doc/de/lmu/ifi/nm/qossl/index.html .

[QoSP] SCHAAL, JOHANNES: *QoSSL-Compiler – Programmdokumentation – QoSSLParser*. Java API documentation generator, 2006, ./doc/de/lmu/ifi/nm/qossl/parser/QoSSLParser.html .

[RJB 98] RUMBAUGH, JAMES, IVAR JACOBSON und GRADY BOOCH: *Unified Modeling Language – Reference Manual*. Addison-Wesley Professional, Reading, Mass., ISBN 0-201-30998-X, Erste Auflage, 1998.

[Scho 01] SCHÖNING, UWE: *Theoretische Informatik – kurzgefasst*. Spektrum Akademischer Verlag, Heidelberg, ISBN 3-8274-1099-1, Vierte Auflage, 2001.

[Tane 03] TANENBAUM, ANDREW S.: *Computernetzwerke*. Pearson Studium, München, ISBN 3-8273-7046-9, Vierte Auflage, 2003.

[Ulle 06] ULLENBOOM, CHRISTIAN: *Java ist auch eine Insel*. Galileo Computing, Bonn, ISBN 3-89842-747-1, Fünfte Auflage, 2006.

[UML 01] (OMG), THE OBJECT MANAGEMENT GROUP: *Unified Modeling Language Specification, Version 1.4*. Draft, 2001, http://www.omg.org/cgi-bin/doc?formal/05-04-01.pdf .

[USA 01] AHO, ALFRED V., RAVI SETHI und JEFFREY D. ULLMAN: *Compilers – Principles, Techniques, and Tools*. Prentice-Hall, New Jersey, ISBN 0-201-10194-7, Pearson International Auflage, 2003.

[Warm 99] WARMER, JOS und ANNEKE KLEPPE: *The Object Constraint Language*. Addison-Wesley Professional, Reading, Mass., ISBN 0-321-17936-6, Zweite Auflage, 1999.

[Wirs 03] WIRSING, MARTIN, HARALD STÖRRLE und NORA KOCH: *Methoden des Software-Engineering*, 2003, http://www.pst.ifi.lmu.de/lehre/WS0304/mse/ .

[X.641] ITU-T: *OSI Networking and System Aspects – Quality of Service*. Recommendation X.641, 1997.

[XHTML] *XHTML 1.0 – The Extensible Hypertext Markup Language – A Reformulation of HTML 4 in XML 1.0*. W3C Recommendation, 2000, http://www.w3.org/TR/xhtml1 .

[XML] BRAY, T., J. PAOLI, C. M. SPERBERG-MCQUEEN und E. MALER: *Extensible Markup Language (XML) 1.0*. W3C Recommendation, 2004, http://www.w3.org/TR/REC-xml .

[YaCC] JOHNSON, STEPHEN C.: *Yacc: Yet another Compiler-Compiler*, http://dinosaur.compilertools.net/yacc/ .