MODELING QOS PARAMETERS IN COMPONENT-BASED SYSTEMS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Praveen Gopalakrishna

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

August 2004

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
AUG 2004		**00-00-2004 to 00-00-2004**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Modeling QoS Parameters in Component-Based Systems	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Purdue University,Department of Computer and Information Sciences,Indianapolis,IN,46202**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**101**	

To amma

ACKNOWLEDGEMENTS

It's been a knowledge elevating experience during my course of graduate studies at the Department of Computer and Science, Indiana University Purdue University Indianapolis. I would like to take this opportunity to express my gratitude to all those who made my graduate study, a memorable one and making this thesis possible.

I would like to thank my advisor Dr Rajeev Raje for creating an opportunity, for me to be part of UniFrame project. I thank him for all his advice on the research front as well as guiding me in my graduate studies. I am grateful for his constant encouragement which made it possible for me to explore and learn new things.

I wish to thank Dr Andrew Olson for providing me with valuable advice and input in my research front and being on my thesis committee.

I would also like to thank Dr Stanley Chien for being on my thesis committee and for reviewing my thesis.

I am thankful to U.S. Department of Defense and the U.S. Office of Naval research for supporting this research with their grant.

I would like to thank all my teammates of UniFrame project, faculty and the staff of Computer Science Department for their co-operation and assistance towards this thesis.

Finally I would like to thank my mother, brother and sister for their kind support and encouragement.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Gopalakrishna, Praveen. M.S., Purdue University, August 2004. Modeling QoS Parameters in Component-based Systems. Major Professor: Rajeev Raje.

Current trends in the software development are focused on creating systems by integrating previously developed software components. This approach aids in the reusability of the code and helps to reduce the cost of software development. In addition to the functionality a component offers, it may contain the necessary code for measuring how well the functionality will be achieved during the execution. This gives rise to the notion of quality of service (QoS) offered by a component -- latency, throughput, capacity, precision, etc., are a few examples of QoS parameters. Many applications, such as multi-media, emphasize and require a certain level of the QoS offered by components. Thus, it is critical to model the QoS, at an appropriate level of an abstraction, during the modeling of component-based systems. Such a modeling will not only assist the component developers but also emphasize the need for integrating the QoS during the development and implementation phases of the software design. In this thesis, an approach based on a unified framework (UniFrame) is proposed to model the QoS parameters in component-based systems. The approach involves QoS concepts relevant for specifying QoS, a UML profile for representing the concepts, integrating the QoS specification with the functional specification and mapping the specification manually onto the code of the component.

1. INTRODUCTION

Component based software development (CBSD) is an emerging field which aims at developing software systems out of prefabricated Commercial off the shelf (COTS) software components. A software component is defined by [SZY99] as a unit of composition which has contracts specified in its interface and explicit context dependencies of the component. A component could be deployed independently and can be used as a composition unit for building applications by third parties. Technological advances in networking have led to high speed network connections between computers and thus aided in a new computing paradigm called distributed computing, where components involved in the application are remotely located and communication between these components happens using a network.

A number of distributed computing models are in existence for developing distributed computing applications. Some of them are J2EE™, .NET™, CORBA™, etc. These models do not provide facilities to interact with each other, thus hindering interoperability among components which adhere to various models. For example, J2EE focuses on Java programming language to create distributed systems and thus, can build systems composed of Java components. .NET framework supports multiple programming languages, provided that the language conforms to .NET Common Language Interface. Such a model enables the creation of distributed systems that work well within its domain, but when it comes to creating distributed systems involving components adhering to different models, it hinders, as the models do not consider the idiosyncrasies of the other models.

An approach that takes into consideration the existence of different models is required for achieving interoperability among models. One possible approach is to use a meta-model which encompasses the necessary aspects of components which adhere to

distributed computing models. Such a meta-model will assist in achieving interoperability among different heterogeneous distributed computing models. One such meta-model is incorporated in UniFrame.

The UniFrame and UniFrame Approach (UA) [RAJ01, RAJ02] provide a framework which allows a seamless interoperation of heterogeneous and distributed software components. The key concepts of the approach are: a) A meta-component model (the Unified Meta Model – UMM [RAJ00]), with an associated hierarchical setup for indicating the contracts and constraints of the components and associated queries for integrating a distributed system, b) distributed resource discovery service for discovering the components, c) the validation and assurance of Quality of Service (QoS) based on concepts of event grammars.

The UniFrame framework automates the process of integrating heterogeneous components to create a distributed system that conforms to QoS requirement. The QoS is an abstract term that is realized by how much a user of the system was satisfied by the functionality provided by the system. The framework incorporates a discovery service, UniFrame Resource Discovery Service (URDS) [NAY02], which discovers the components necessary to build a system. The discovered components may be heterogeneous in nature, so in order to make them interoperate, a glue and wrapper generation framework is provided as part of UniFrame. The glue and wrapper generation framework generates the necessary code that mediates between heterogeneous components to make them interoperate. Since glue and wrapper code forms part of the system, it must be quality aware so that it does not hinder the QoS provided by the system formed by composing heterogeneous components. In some cases, if the discovery service is not able to find some of the components necessary to build the system, it may be useful to generate the components by using existing code generation techniques. One such technique is provided in Generic Modeling Environment [GME02], which allows the system developer to model the functionality of a system and generate the code. The component generated must provide the necessary QoS as it forms the part of the composed system in UniFrame.

1.1. Problem Definition and Motivation

The CBSD is about developing a system with prefabricated components. The UniFrame provides a framework to create a distributed system that conforms to quality requirements. It has been indicated in [SUN02] that the QoS offered by the components in the component-based distributed system has a significant effect on the QoS for the entire system. The QoS provided by the system developed using CBSD will depend on the QoS provided by the constituent components, which are interacting with each other. The QoS provided by a component depends on whether it received necessary resources, which will ensure the satisfactory functioning of the component. The resources may be a computed value required by component A which is expected to be provided by component B. In a system, in which each component's QoS offer depends on the QoS provided by the interacting components, there is a need to ensure that the proper QoS is provided by all interacting components, such that, the system formed out of these components provides the necessary QoS to the user of the system. Some applications, such as multi-media, emphasize and require that a certain level of QoS be offered by the integrated system. This emphasizes that the constituent components must provide the necessary QoS. Thus, it is critical to model QoS, at an appropriate level of abstraction, during the design of any component-based system. Such a model will not only assist the component developer but also emphasize the need for integrating QoS during the design and implementation phases of software. Abstractions of QoS also assist in reuse of information during the design and implementation phases. The Object Management Group (OMG), a consortium of industries for creating standards in many domains has recognized the importance of emphasizing QoS during design and has issued a Request for Proposal (RFP) for modeling QoS and fault tolerance characteristics and mechanisms [OMG02].

Software development had taken a new approach with availability of Computer Aided Software Engineering (CASE) tools, which depict the abstracted information about the functionality of the system in a graphical format. This graphical model is used by a code generator to generate code templates. The process of creating a graphical model involves gathering of information about the functionality of the system and representing

it using an accepted standard. This process provides the notion of abstracting information which is not dependent on technology, and it also suggests that the abstracted information could be reused. OMG's Model Driven Architecture [MDA01] initiative is an example which aims at separating business or application logic from underlying technology. It also aims at standardizing these Platform Independent Models and transformations of these models to multiple Platform Specific Models. Abstracted QoS information can be represented in a graphical format which can be used to specify QoS during the design phase of components, which in turn would be used by the code generator to generate code templates which are quality aware.

The Unified Modeling Language (UML) [UML01] by OMG is a visual paradigm for describing the functional aspects of an object-oriented system. The visual paradigm for software development has been accepted as a way of describing the functionality of the system [UML01] and it has been supported by the availability of CASE tools. UML, though rich in modeling elements for functional aspects of the system, lacks the support for modeling or expressing constraints on the objects of the system [OCL01]. Many existing formal languages for describing these constraints could have been used but they require mathematical background for the system developer. The Object Constraint Language (OCL) is used to address the issues of expressing constraints on the objects of the system. The OCL was designed to be simple enough for developers, yet able to express the constraints without ambiguity. The OCL is a pure expression language and it does not have any side effects [OCL01]. Adding an OCL expression in a functional model will not cause the state of the system to change, even though OCL could be used to specify the state change. The OCL uses a lexical approach to specify constraints and to express the constraints using OCL requires the specification of context in which the constraint has to be applied and the context may involve all or some of the following, objects, methods, roles, attributes, association. These constraints are typically expressed in a file which has to be referenced while viewing the functional model. If a complex system is involved, significant effort is needed for viewing and comprehending the constraints of the system.

The QoS can be viewed as constraints expressed on the components of the system. The OCL could be used to express these QoS constraints, but its lexical nature makes the component developer, who is not an expert at programming, to put more effort in comprehending the constraints of the system. However, computers can process textual representation more easily than graphical one as graphics needs to be converted to an intermediate form so that the existing computer technology can be used to process the representation. It has been indicated in [NOS90] that the visual representation is more understandable and transparent than the textual representation. However, [GRE92] states that visual representation deemphasizes the issues of syntax and provides a higher level of abstraction. This thesis proposes an approach to represent QoS that makes use of the better of both worlds by representing the concepts for QoS (which will be explained in Chapter 4) essential for specifying QoS in visual paradigm and expressing the constraints in a QoS visual model using text. In addition to representation of QoS, the thesis also proposes an approach where the same QoS information can be used for another context in a system. The OCL currently does not provide a way of reusing the information and thereby not reducing the effort required for expressing the constraints of the system.

The UML [UML01] provides an extension mechanism for modeling issues specific to a domain. The UML provides two ways to extend the model, which are the Heavyweight extension and Lightweight extension mechanisms. The Heavyweight extension mechanism extends by modifying the existing UML meta-model. The Lightweight extension mechanism extends without modifying the meta-model. It extends by adding Stereotype, Tagged values and constraints to the meta-model. The Stereotype, Tagged values and constraints are derived from the existing model elements, attributes, methods, links, etc., in the UML meta-model. These extensions are grouped together to form a profile, which will enable description of particular modeling problems, specific to the domain and thus, provide constructs to express them.

In this thesis, a QoS profile is proposed that enhances the UML to provide a visual paradigm for expressing QoS. The OCL is used to augment the visual model in expressing precise constraints. The QoS model (visual representation of QoS), referred to as the non-functional model, depicts only the QoS aspect of the functional model. Thus, it

is a separate view of the functional model. This thesis also provides an approach to relate the non-functional and functional model.

1.2. Objectives: Statement of Goals

The objectives of this thesis are:

- Provide a representation mechanism for specifying non-functional attributes of the components and a system of components during its design and implementation phase.
- Provide a way of integrating the functional and non-functional model during design and development phases of the system.
- Provide a way of mapping the specification of QoS onto the system code.
- To study the effectiveness of the proposed non-functional model on various QoS parameters defined in QoS catalog [BRA02].

The approach used in this thesis to achieve the above mentioned objectives and goals are as follows:

- Extension of generic QoS concepts proposed in [AAG01] to incorporate the concepts related to QoS parameters that do not get affected during run time.
- Representation of the QoS concepts in UML.
- Integration of the functional model and non-functional model using interaction diagrams of UML.

1.3. Contribution of this Thesis

The contributions of the thesis are as follows:

- Proposes Quality of Service concepts relevant for modeling QoS of components.
- Creates a QoS profile in UML to represent the QoS concepts.
- Proposes an approach for integrating functional and non functional model.
- Manually mapping QoS specifications onto relevant code of the component.

- A case study from the document management domain to illustrate the applicability of the proposed approach in real world scenarios.

1.4. Organization of this Thesis

The thesis is organized into six chapters. Chapter 1 provides the introduction to thesis with problem definition and motivation. Chapter 2 presents the related work on modeling QoS in CBSD. Chapter 3 describes about the UniFrame approach, UMM and the glue wrapper generation. Chapter 4 presents an approach to model quality of service along with concepts and representation using UML. It also proposes an approach to integrate the functional and non functional model. Chapter 5 presents a case study to validate the approach. Chapter 6 presents the conclusion to the thesis by listing the features of the model, possible enhancements and future work.

2. BACKGROUND AND RELATED WORK

This chapter provides an overview of the existing QoS specification mechanisms for components in distributed systems.

2.1. Quality Objects

The Quality Object (QuO) is a framework [BBN01] for providing QoS in distributed applications that are composed of objects. The QuO is intended to help application developers to develop distributed applications with specific QoS requirements. It enables the specification, measurement and control of QoS of the application and the adaptation to changes in QoS. The QuO attempts to bridge the gap of network level guarantees and application level QoS requirements.

The QuO extends the functional IDL of CORBA to incorporate QoS. The QoS is specified using QoS description language (QDL) which describes the QoS contracts between the clients and the objects, the resources of the system, and the way of measuring and providing QoS.

The QDL consists of Contract Description Language (CDL) and Structure Description language (SDL). CDL is used to specify the QoS contract between the client and object in an application. The QoS contract indicates the QoS required by the client and the QoS that can be provided by the object. The QoS contract also specifies the possible levels of QoS the system provides, the behavior to invoke when the client desires, the object expectation or QoS conditions change.

The CDL contract consists of: a) a set of nested operating regions indicating the possible states of the QoS. Each state has a predicate associated with it to indicate

whether the state is active or not, b) a transition for each state to specify the behavior to invoke when the active region changes, c) references to system condition objects, which are passed as parameters to contracts in order to measure and control QoS, and d) callbacks to notify client or objects about the transition from one state to another.

A sample CDL contract which specifies the throughput for the system is shown in Figure 2.1.

```
contract Throughput( syscond valuesSC ValueSCimpl ClientExpectedThroughput,
callback AvailCB ClientCallback,
syscond ValueSC ValueSCImpl MeasuredThroughput,
syscond Thpr ThprSCImpl ThprMgr) is.
negotiated regions are
  region low_throughput: when ClientExpectedThroughput == 10 =>
        reality regions are
                region Low : when MeasuredThroughput < 10 =>
                region Normal: when MeasuredThroughput == 10 =>
                region High: when MeasureThroughput > 10 =>
                 transition are
                        transition any->Low :ClientCallback.throughput_degraded();
                        transition any->Normal: ClientCallback.throughput_normal();
                        transition any->High: ClientCallback.Throughput_being_wasted();
                end transition;
        end reality regions;
  region high_throughput : when ClientExpectedThroughput >=11 =>
        reality regions are
            region Low : when MeasuredThroughput < ClientExpectedThroughput =>
            region Normal: when MeasuredThroughput >= ClientExpectedThroughput =>
                 transition are
                        transition any->Low :ClientCallback.throughput_degraded();
                        transition any->Normal: ClientCallback.throughput_normal();
                end transition;
        end reality regions;
   transitions are
        transition low_throughput-> high_throughput
          Thprmgr.adjust_throughput(ClientExpectedThroughput)
        transition high_throughput->low_throughput
          Thprmgr.adjust_throughput(ClientExpectedThroughput)
   end transition;
  end negotiated region;
end Throughput contract;
```

Figure 2.1. A CDL Contract

The above sample CDL contract specifies the Throughput behavior of a QuO application. The client has two operating regions, which are low_throughput and high_throughput. These two regions have nested regions in which the system can operate. The client can request different Throughput based on the client's requirements. These requested throughputs are correspondingly indicated by low_throughput and high_throughput regions. The contract also specifies the method to be invoked whenever there is a transition from one region to another.

The Structural Definition Language (SDL) describes the structure of the remote object implementation, such as implementation alternatives and adaptive behavior of the object delegates. A SDL description has: a) a set of interfaces and contracts with adaptive behavior specified by the SDL specification, b) method calls for which the adaptive behavior is to be specified, c) a set of regions indicating the states that QoS can adapt to and, d) behavior specification of various alternative object bindings.

A Sample SDL specification is shown in Figure 2.2.

```
delegate behavior for Targeting and Throughput is
 obj : bind Targeting with name SingleTargetingobject;
 group: bind Targeting with characteristic { Througput = True };
 call distance_to_target:
 region Throughput.Normal:
 pass to group;
region low_throughput.Normal :
pass to obj;
region high_throughput.Low:
throw ThroughputDegraded;
default:
pass to obj;
return distance_to_target:
 pass through;
 default
 pass to obj;
end delegate behavior;
```

Figure 2.2. A SDL Specification

The QuO focuses only on the quality aware communication channel between the client and the servers. It supports the specification of QoS characteristics which are measurable and vary during the runtime, thereby forcing a restriction on the type of QoS characteristics that can be used for specifying QoS. Some of the characteristics which may not be specifiable using QuO are accuracy, dependability, security, etc., as these characteristics do not vary during runtime.

The QuO specifies the quality contracts the client and object will have. It does not specify on how this contract is reflected in the implementation of the object. It is left to the developer to provide the implementation for the contract during the development of the object.

The Quality specification provided by QuO is for the entire system and it does not provide a way to specify QoS of individual components involved in the system.

2.2. Quality of Service Modeling Language

The QoS Modeling Language (QML) proposed in [FRO98] is a QoS specification language which can be used to specify QoS in Object-Oriented systems. The QML provides a QoS specification mechanism that

a) is syntactically separate from the service specification, such as the Interface. This feature provides a way to specify different QoS properties for various implementations of the same interface,
b) allows the QoS of the client to be specified separately from the QoS of the server. This feature allows a developer to specify the QoS properties of a component separately for all the components which are involved in collaboration to form a system,
c) allows a way to determine whether the QoS specification for a service satisfies the QoS requirements by the client.

The QML is a general purpose QoS specification language for describing QoS parameters in any application domain. It provides three concepts for specifying QoS: contract type, contract and profile. A contract type relates dimensions to a QoS category

defined in [ISO99]. Performance, reliability, and security are few examples of QoS categories. A contract type has a dimension type for each of the dimensions in the category. For example, delay and throughput are dimensions of QoS category performance. A dimension type provides a way of indicating the value of the dimension. Three dimension types which have been defined in QML are Set, Enumeration and Numeric.

Contracts are instances of a contract type and their structure is defined by the contract type. A contract contains a list of constraints which is imposed on the values of the dimensions of a QoS category. A constraint consists of: a name of the dimension, an operator and a value. For example in the constraint, numberofFailure == 10, the constraint specifies the numberofFailure dimension to be equal to ten.

The profiles separate the QoS specification for a component from the interface definition of the component. A profile is related to an interface and it specifies the QoS contracts for the attributes and operations described in the interface. A profile is used to specify client QoS requirements or server QoS provisioning. A profile will be bound to an entity and the entity can be a client or a server. A sample specification for contract type, contract and profile are shown in Figure 2.3, 2.4, 2.5 respectively.

```
type Reliability = contract {

numberOfFailure: decreasing numeric per year;
meanTimeToRepair: decreasing numeric sec;
availability: increasing numeric;

};
```

Figure 2.3. A Sample Contract Type

A sample contract type for *Reliability* is shown in Figure 2.3. The contract type for *Reliability* has dimensions: numberOfFailure, meanTimeToRepair and availability. The associated values for the dimension are also shown in the Figure. According to the contract, the numberOfFailure suggests that decreasing failure rate is desirable. The

meanTimeToRepair should be of decreasing value and the system availability to be of increasing value.

```
systemReliability Reliability contract {
  numberoFFailure < 5 units/yr
};

serverReliability Reliability contract {
meanTimeToRepair < 10 sec;
};
```

Figure 2.4. A Sample Contract

A sample contract for reliability is shown in Figure 2.4. The contract *systemReliability* is an instance of contract type *Reliability*. It will have the structure of contract type *Reliability*. The above contract for *systemReliability* specifies the number of failures to be less than 5. If any of the dimensions in the contract type is omitted in contract, it is assumed to be not provided by the component. Similarly, the contract *serverReliability* requires the mean time to repair to be less than 10 sec..

```
interface server {
void init();
void register( string name);
 object lookup ( string name);
 }
serverProfile for server = profile {
 require serverReliability;
 };
```

Figure 2.5. A Sample Profile

The Figure 2.5 shows a profile for a *server* interface. The profile termed *serverProfile* associates contracts with the operations defined in the *server* interface. The *serverProfile* associates the *serverReliability* contract as the default contract.

The QML is a generic QoS specification language. It allows separation of QoS specification from the functional specification. It allows creation of profiles to associate the QoS with the functional specification, but it does not provide a way to specify the number of contracts which can be offered by the component developer for varying environmental conditions.

2.3. Component Quality Modeling Language

The Component Quality Modeling Language (CQML) [AAG01] is generic QoS specification language to specify QoS of components in a distributed system. The CQML provides the necessary concepts and constructs to specify QoS of components. The CQML concept model for QoS is shown in Figure 2.6. The figures depicted in this section are from the work [AAG01]. The concept model has elements which are independent entities and are indicated by rectangular blocks. The relations between these blocks are indicated using UML notation for Association and Generalization. A *QoSComponent* offers services with certain QoS specified in its *QoSProfile*. A *QoSProfile* is a relation that specifies the QoS provided by the *QoSComponent*. The QoS is specified by using statements that are indicated as *QoSStatement* in the model. A *QoSStatement* indicates the restraining values on the *QoSCharacteristic* and is therefore composed of constraints, which are termed *QoSConstraint*. A *QoSConstraint* indicates the restrictions on the *QoSCharateristic*.

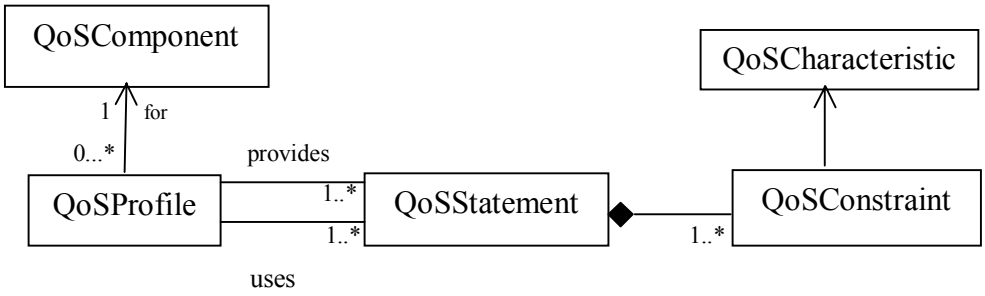

Figure 2.6. QoS Concept Model

A *QoSComponent* represents the role of a model element, which could be a component, actor, interface, use case, or object.

The *QoSProfile* concept shown in Figure 2.7 specifies the contract terms under which the component can have a QoS relation. It specifies the conditional contracts under which the component will provide the necessary QoS as long as the environment provides the QoS needed by the component. A *QoSProfile* may be simple or compound. A *SimpleProfile* specifies one QoS offer by the component and one QoS expectation from the environment by the component. A *CompoundProfile* specifies more than one offer and expectation pair. A *CompoundProfile* can adapt to changes during run time by transitioning from one profile to another. The transition occurs when an existing profile expectation gets invalidated due to environment changes and there exists another profile which assumes a weaker expectation. A *QoSProfile* uses a *ProfileTransition* function to make a call back operation on the component in order for the component to adapt, in accordance to the change.

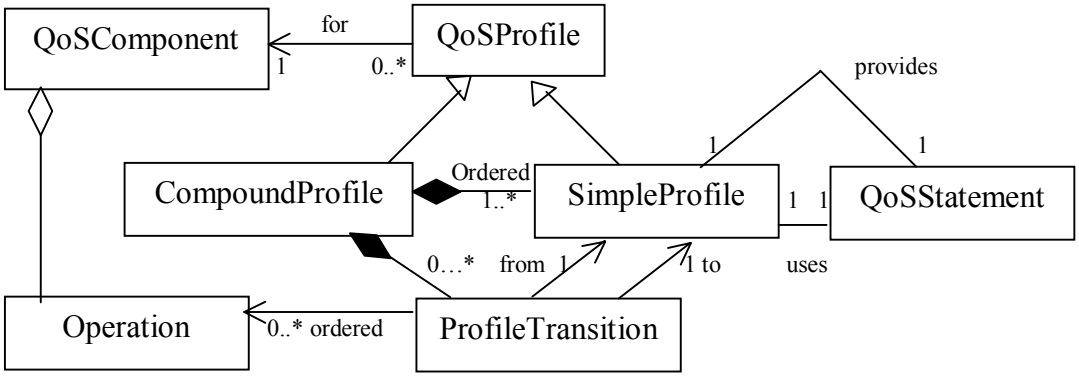

Figure 2.7. QoS Profile Concept

The conceptual model of *QoSStatement* is shown in Figure 2.8. A *QoSStatement* specifies the QoS offers and QoS expectations by the component. A *QoSStatement* can be a single or compound. A Compound statement is comprised of two or more *QoSStatement*. A Compound statement relates to its constituent *QoSStatement*'s by AND or OR relations. A single QoSStatement contains a *QoSConstraint*. A QoSStatement may

have parameters, which relate to properties of the component to which the *QoSStatement* pertains. The *QualificationKind* attribute type is an enumeration whose values are {guaranteed, best effort, threshold}.

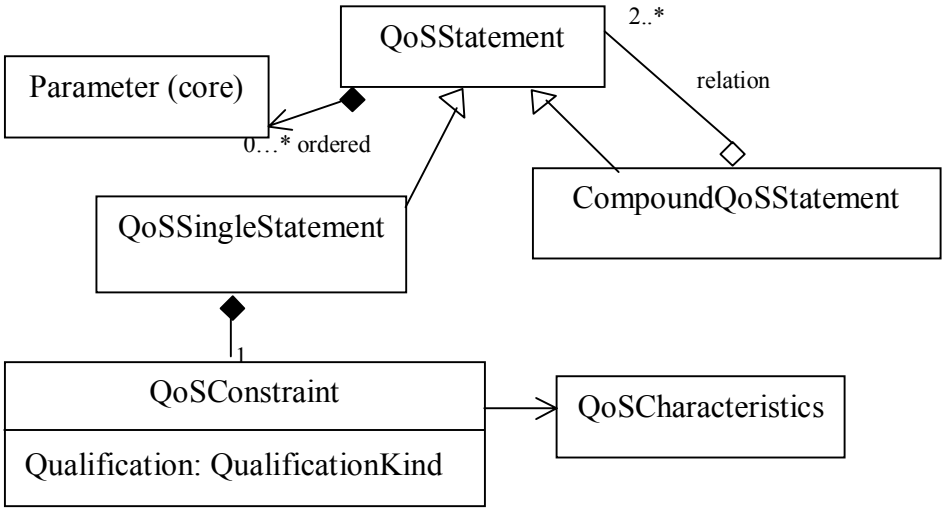

Figure 2.8. QoS Statement Concept

The *QoSCharacteristic* as defined in [ISO99] is some aspect of QoS which is identifiable and quantifiable. For example, delay, throughput, security, etc. The Figure 2.9 shows the conceptual model for the *QoSCharacteristic*. A Composition attribute denotes the effect with respect to composition of the *QoSCharacteristic* under consideration. Value attribute denotes how the values of the *QoSCharacteristic* can be derived at run time. *QoSCharacterisitc* has a domain associated which specifies the values the *QoSCharacteristic* can have, and it also indicates whether an increasing value or a decreasing value is good.

The *DirectionKind* attribute type is an enumeration whose values are undefined, increasing, decreasing. The *StatisticalAttributeKind* attribute type is an enumeration whose values are {undefined, maximum, minimum, range, mean}.

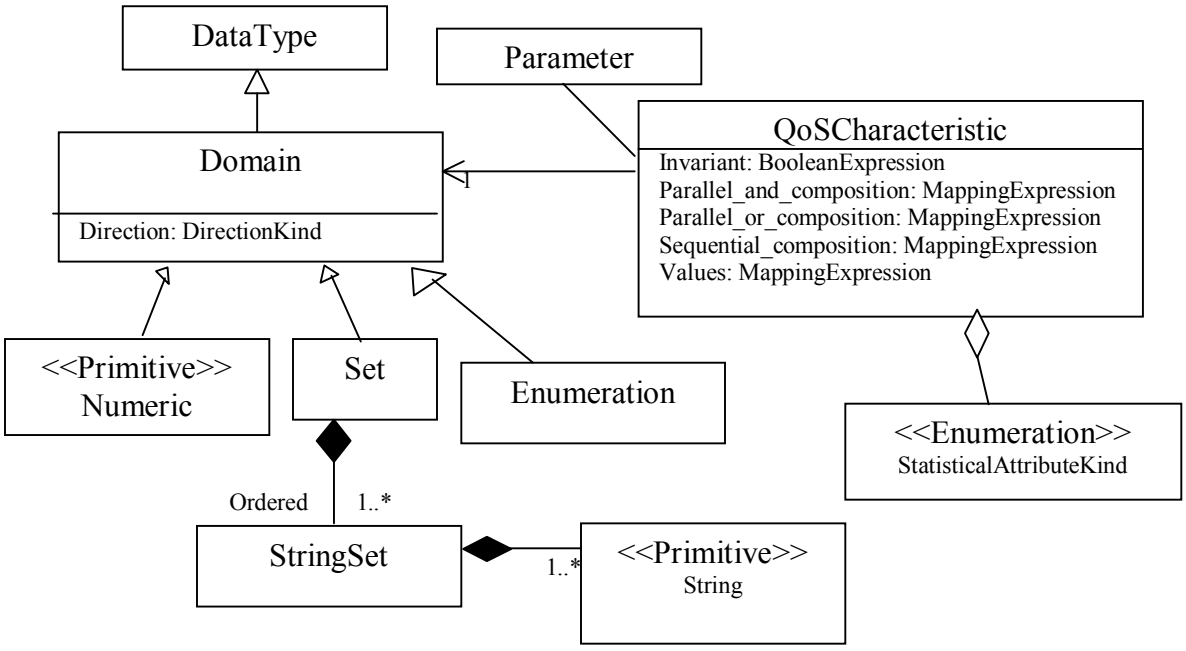

Figure 2.9. QoS Characteristic Concept

The Figure 2.10 shows the specification of a statistical_delay characteristic and the specification of a QoS statement for guaranteed_high characteristic. The domain in the statistical_delay indicates that decreasing numeric values are good and the characteristic value is expressed as the mean of the values. The Frame output statement in guaranteed_high characteristic specifies the frame rate of 25 or greater is required for guaranteeing high quality video.

```
QoSCharacteristic:
        Quality_Characteristic statisticalDelay {
        domain: decreasing numeric millisecond;
                mean;
                }
QoSStatement:
                Quality guaranteed_high {
                frameoutput >= 25
                }
```

Figure 2.10. QoS Specification

The CQML is able to express QoS specifications for components and is applicable to QoS parameters which can be specified and implemented as part of a component. The CQML does not provide ways to specify QoS characteristics, such as security or maintainability, that are realized by conducting external tests on the component. The CQML does not provide concepts for specifying some QoS Characteristics like security, maintainability, dependability, etc., such that the implemented code aids in the testing of the component.

2.4. ISO/IEC 9126

The ISO/IEC 9126 [ISO99] is a standard from the International Organization for Standardization. The ISO/IEC 9126 provides standards for evaluating quality of a software product. The standard lists a set of quality characteristics for evaluating software products. The standard defines the quality characteristics and provides a simple way to measure them. It defines six quality characteristics for a software product and these characteristics are further divided into sub-characteristics. The six quality characteristics and their associated sub-characteristics are listed and defined below.

a) Functionality

It refers to whether the software product has the desired functionality required by the user.

- Suitability: To evaluate whether the software has the functionality which is required to perform the specified task.
- Accurateness: To evaluate whether the software functionality provides the right or agreed results when used.
- Interoperability: To evaluate whether the software functionality can interoperate with other parts of the system.
- Security: To evaluate whether software functionality can be prevented from unauthorized access.

b) Reliability

It is used to evaluate the reliability of the software product.

- Maturity: To evaluate the frequency of software product failures due to software faults.
- Recoverability: To evaluate whether the data can be recovered in case of software product failure and the time and effort needed for it.
- Fault Tolerance: To evaluate whether the software product can maintain the level of performance in case of software faults.

c) Usability

It is used to evaluate the usability of software product.

- Understandability: To evaluate the effort needed to understand the logical concept of the software product
- Learnability: To evaluate the effort needed for learning the application of software products

d) Efficiency

It is used to evaluate the efficiency of the software product.

- Time Behavior: To evaluate the response and processing time for the use of functionality of the software product.

e) Maintainability

It is used to evaluate the complexity of effort needed to change the functionality of the software product.

- Changeability: To evaluate the effort needed to modify or remove faults in the software product.
- Stability: To evaluate the risk involved in the functionality of the software product if the functionality is modified.
- Testability: To evaluate the effort needed for validating the modified software.

f) Portability

It is used for evaluating the software products ability to operate in different environment.

- Adaptability: To evaluate the software product's ability to adapt to new environment without modifying the software product.
- Installability: To evaluate the effort needed to install the software product in a specified environment.
- Conformance: To evaluate the conformance of the software product for standards or conventions relating to software portability.

The ISO/IEC 9126 standard identifies the QoS characteristics which are intended for software in general, but it does not identify some of the QoS Characteristics which are unique to CBSD like capacity, ordering constraints, etc.

The ISO/IEC 9126 provides a way to evaluate the QoS of the software product which has already been designed and implemented. It does not provide a way to specify QoS during the design phase of the component so that the QoS factors can be taken into consideration during implementation of the component.

2.5. Object Constraint Language

The Object Constraint Language (OCL) [OCL01] provides a way of specifying the constraints on the model elements of the functional model of a software system. A constraint is a restriction on one or more values of an Object-Oriented model or system. The functional model depicts the abstracted functionality of a software system in a visual paradigm. The UML, a visual paradigm, by OMG is used to depict the abstracted functionality of the software system. The UML provides necessary modeling elements for modeling an Object-Oriented system. However it does not provide a way of representing constraints on the system.

The OCL was developed to address the issue of expressing constraints on the system. It is a simple, text-based formal language for expressing constraints on the system, and it is also used to define precise semantics to the modeling elements of UML, and thereby enabling creation of precise models of the system functionality. The constraints, when enforced and followed, will result in providing QoS by the system.

A simple OCL constraint example from [WAR03] is shown in Figure 2.11 for a system depicted in 2.12.

```
context Flight:
inv: passengers->size() <= plane.numberOf Seats
```

Figure 2.11. OCL Constraint

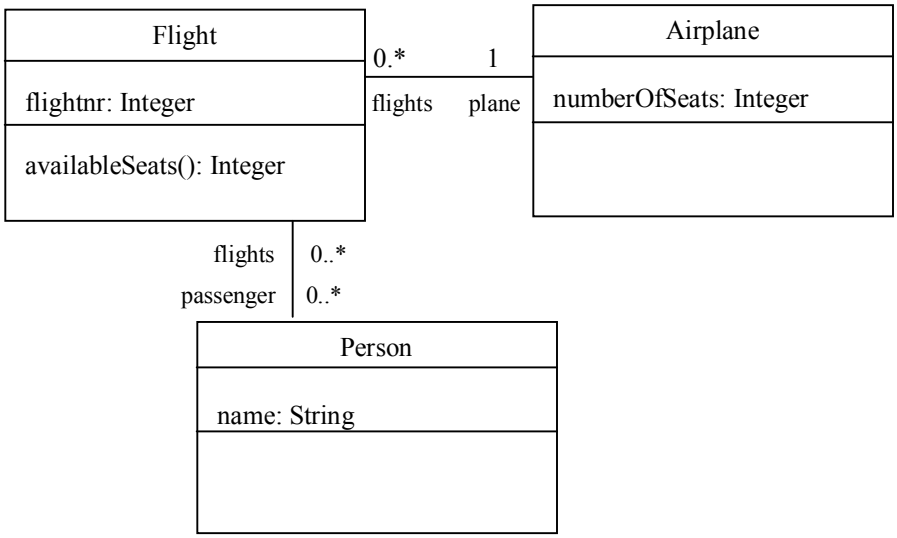

Figure 2.12. Functional Model of A System

The functional model shown in Figure 2.12 has three objects interacting with each other: flight, airplane and person. The constraint shown in Figure 2.11 puts a restriction on the role named passenger such that the number of passengers must be less than or equal to the number of seats in the plane. This constraint must be true at any instant of time; thus it is indicated by specifying it to be an invariant. This constraint when enforced and followed will result in QoS characteristic called capacity.

The OCL provides a way of specifying QoS during the design phases of the component, but currently does not provide a facility to reuse the constraint in some other context. Each constraint has a context associated with it. It does not provide a construct to specify multiple contexts with the same constraint.

The OCL does not provide constructs for expressing QoS characteristics like security, maintainability, etc.

2.6. Modeling QoS in Components

In the previous sections of this chapter a few of the related works for specifying QoS in CBSD were discussed along with their shortcomings. Some of the shortcomings are summarized here.

a) Unable to specify QoS during design phases of the component such that it gets reflected in the implementation of the component by using existing code generation techniques [GME02].
b) Unable to provide reusability of QoS specification.
c) Identifying and defining QoS parameters, which enhance the current list of parameters defined in ISO 9126, and are relevant for CBSD such as ordering constraint, parallelism, etc.
d) Specifying QoS parameters which do not change during runtime of the component like security, maintainability, etc.
e) Mapping of QoS specification onto the source code statements of the component, which does the necessary functionality to measure and ensure QoS.

UniFrame framework [RAJ01, RAJ02] provides the generation of QoS aware glue and wrappers and QoS aware components. Thus it needs a QoS specification mechanism which can

a) Specify the QoS parameters for individual components and for the whole system.
b) Provide reusability of QoS specifications to minimize the effort of specifying QoS.
c) Provide a mapping mechanism to reflect the QoS specification on the code that will ensure the QoS of the functionality.

The thesis provides a QoS specification mechanism that

a) Extends the concepts defined in CQML [AAG01] to take into account the parameters which do not vary during run time.
b) Represents the concepts in UML, thereby providing constructs to express QoS during the design phases of the component.

c) Uses the Quality catalog [BRA02], a work which is part of UniFrame framework, to identify the QoS parameters for CBSD.
d) Provides a mechanism to reuse the QoS specification.
e) Is manually mapped onto code.

In this chapter an overview of the existing QoS specification mechanism was presented. Some of the drawbacks of the existing QoS specification mechanisms were summarized, and the need for QoS specification mechanisms that can take into account the drawbacks of the existing approaches so that quality aware glues and wrappers and components can be generated in UniFrame was also indicated. The next chapter provides an overview of UniFrame approach.

3. THE UNIFRAME

The objective of the thesis is to provide a QoS specification mechanism during the design phase of the system composed of components. This QoS specification mechanism: a) aids in designing quality aware components which can perform a part of the functionality of the system to be built, b) enable the UniFrame glue wrapper generator, which is a part of UniFrame framework to generate quality aware components, to provide an interoperability mechanism for heterogeneous components,

In the previous chapter, an overview of the existing techniques for specifying QoS of the system was presented. This chapter presents an overview of The UniFrame approach, Unified Meta-Component model and the UniGGen glue generator which will provide a prelude for the objective of the thesis.

3.1. The UniFrame Approach

This section presents the UniFrame approach which implements the concepts identified in UMM (presented in next section) for achieving discovery, interoperability, and collaboration of components adhering to various distributed component models. The UniFrame approach is shown in figure 3.1. The approach has two phases, a) The component development and deployment phase, b) The automatic generation of system using components and validation of QoS of the system.

The UniFrame approach uses a generative programming [CAZ99] paradigm to generate the system from components. The UniFrame approach assumes that the generation of system will be built around a Generative Domain specific Model (GDM) which supports component assembly. This emphasizes that the component will be created for a specific application domain based on a standardized knowledgebase. The UniFrame

knowledgebase indicated in the Figure 3.1 is assumed to be created by domain experts, will contain the necessary application domain information such as the concepts related to the domain (like transaction, type of accounts, etc, for banking domain) usage of concepts (Use cases), etc. The component developer uses the information in knowledgebase to develop the component for a specific application domain. The developer will also include the specification of the component which details about the computational, cooperative, auxiliary aspects, QoS metrics of the component and general information like the application domain, author, etc. The component developer will use the QoS catalog [BRA02] (explained in the section 3.3) to obtain the necessary QoS parameters for the component and does an empirical evaluation of the QoS of the component. If the values of the QoS parameters are satisfied then the component is deployed on the network.

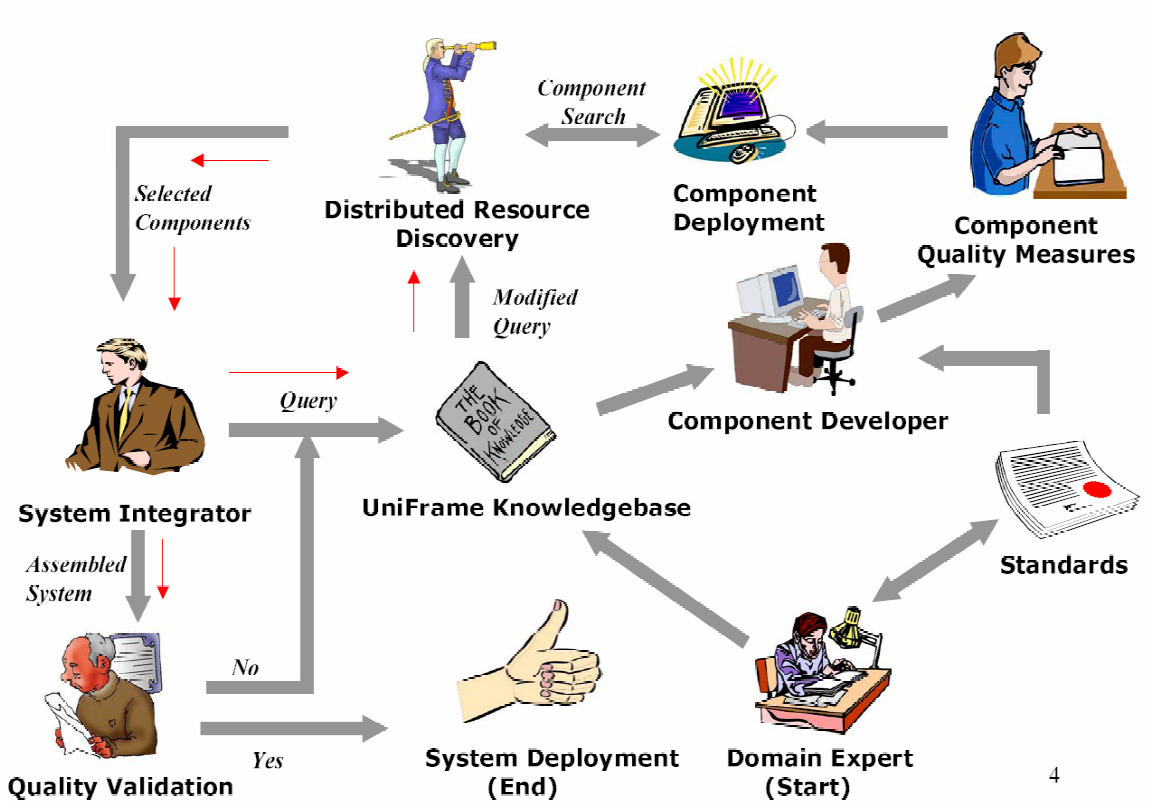

Figure 3.1. The UniFrame Approach

These components once deployed on the network are discovered by the *head-hunters*. The *head-hunters* are a part of the Distributed Resource Discovery process indicated in the Figure 3.1. The UniFrame Resource Discovery Service (URDS) framework which implements the Distributed Resource Discovery Process provides the functionality for discovering components and has the following constituent components:

- Internet Component Broker (ICB): It provides the communication infrastructure necessary to identify and locate services, enforce domain security and handle mediation between the heterogeneous components. The ICB component provides these services using a Domain Security manager (DSM), Query Manager (QM), Adaptor Manager (AM) and the Link Manager (LM).
- DSM: It serves as an authorized third party for enforcing access control of the users of the domain.
- QM: The QM translates the natural language like query from the system integrator to a structured query and passes it onto the appropriate domain head-hunters.
- LM: The LM establishes links with other ICBs for the purpose of federation and to propagate query received from the QM to other ICB.
- AM: The AM servers as a registry/lookup for clients seeking adaptor components.
- Head-Hunters (HH): The *head-hunter* performs the discovery of service providers and registers their functionality. It returns a list of service providers which matches the requirement of the system integrator to the ICB.
- Meta-Repository (MR): The MR serves as a data store which is used by the head-hunter to hold the UMM specification information of the service exporters adhering to different models.

The generation of integrated system from independently developed and deployed components, begins with the system developer, willing to build a system, by presenting a query to the system generator. The query describes about the characteristic of the system in a structured form of natural language. The query is processed with the help of domain

knowledge in the UniFrame Knowledgebase and a set of functional and QoS based search parameters are generated which are presented to the Distributed Resource Discovery which will use the information to aid the *head-hunters* in discovering the components, which meet the functional and QoS requirements. The discovered components are returned to the system integrator. If all the required components indicated in the knowledgebase are found, then the system is built using the system generator. If some of the components are not found then the system integrator can modify the system query by adding more information about the system to be built to get the components necessary to build the system or may supply proprietary components.

Once the system has been built by the generator, it is tested for desired functionality and QoS criteria by using event traces and a set of test cases. In UniFrame event grammars [AUG95, AUG97] are used measure and validate the QoS parameter which vary based on usage pattern and environmental condition. An event is a detectable action performed by the component, like method call or execution of a statement. An event may include another event or precede another event. A system is composed of events with the relation between them indicated by precede or include. These relations form an event trace which is used to validate QoS. If the system meets the QoS and functionality criteria, it is deployed or else another system is built from the component collection and tested.

The QoS specification mechanism enhances the UniFrame knowledgebase ability to specify the QoS chosen for the component.

3.2. Unified Meta-component Model

The Unified Meta-component Model (UMM)[RAJ00, RAJ01] provides the fundamental concept for UniFrame approach and it attempts to unify the existing and emerging distributed models under one common meta-model, thereby enabling discovery, interoperability and collaboration of heterogeneous components. The meta-model is composed of three parts and they are

a) Components: The UMM considers components as autonomous entities whose implementation are non-uniform and adhere to some distributed component model. Each component will have a state, identity, behavior, well defined interface and private implementation. In addition to these characteristics, the components in UMM are considered to have a computational aspect, cooperative aspect and an auxiliary aspect.

   i. Computational aspect: The computational aspect reflects the task performed by each component which in turn depends on the objectives of the task, techniques carried out to achieve the task and the specification of the functionality offered by the component. The UMM indicates the computational aspect using a mixed approach in which the informal text is used to provide book keeping information about the component and precise formal part for the description of computation, its associated contracts and level of service offered by the component.

   ii. Cooperative aspect: The UMM assumes that the components are in the process of cooperating with each other and this is indicated by the cooperative aspect which indicates the collaborators which can collaborate with the component under consideration. It also indicates the components on which the component under consideration depends upon for collaboration as well as the other components which depend on this component.

   iii. Auxiliary aspect: The auxiliary aspect indicates the other issues like mobility, security, and fault tolerance of the component.

A sample UMM description of the component indicated in [RAJ01] is shown in Figure 3.2.

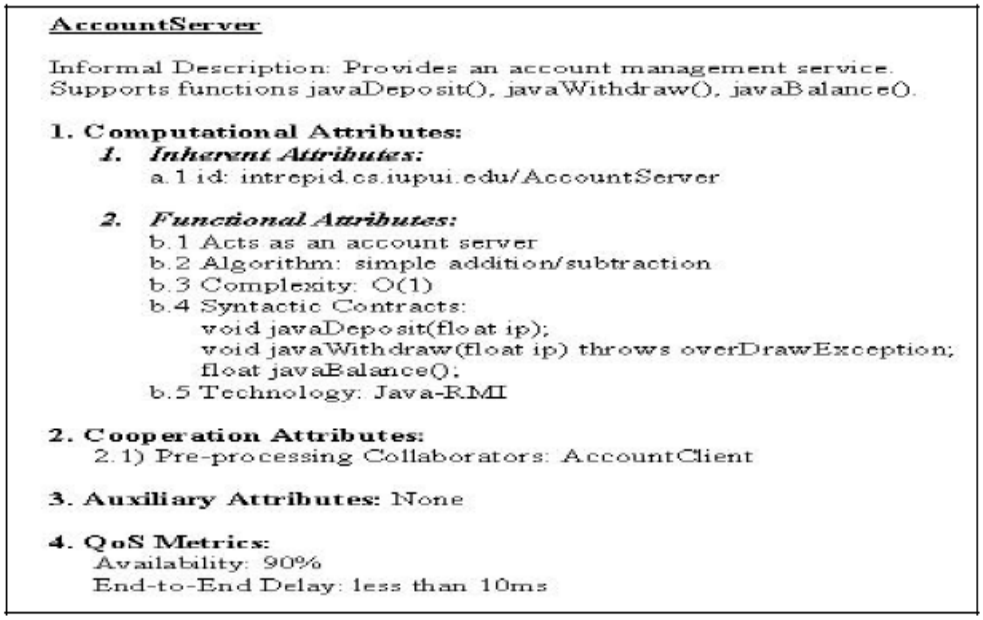

Figure 3.2. A UMM Component Description

b) Service and Service Guarantees: A UMM component may offers services in the form of an intensive computational effort or an access to underlying resources. There may be several components which can offer the same service and thereby offering a choice for the developer to choose the component, which he finds it useful. The indicators which can be used for selecting the component are the cost and the QoS offered by the component. In UMM it is necessary to specify the QoS that the component can offer in terms of the QoS parameters listed in the QoS catalog [BRA02].

c) Infrastructure: The UMM tackles the issues of non-uniformity in DCS due to local autonomy through the concept of *head-hunter* and *Internet-Component Broker*. These two concepts allow interoperability of different component models.

i. *head-hunter*: The *head-hunter* carries the task of detecting new components in the search space and registering their functionality. A *head-hunter* is analogous to a binder or a trader in other models, but differentiates itself from them by being active. The *head-*

hunter is active as it discovers the components and registers with itself, where as in trader, the components bear the responsibility of registering themselves with the trader. A UMM component will inform about its aspects to the *head-hunter* which is used for match making of service producers and consumers. The *head-hunter* may cooperate with each other for discovering larger number of components.

ii. *Internet-Component Broker*: The *Internet-Component Broker* is intended to mediate between two components which adhere to different component models. It uses the adaptor technology for providing translation capabilities for specific component architectures. The computational aspect of the adaptor component indicates the model for which it provides the interoperability mechanism. The adaptor technology achieves interoperability using the principles of *wrap* and *glue* technology [BER01]. The *Internet-Component Broker* is analogous to Object Request Broker (ORB) which provides interoperability among objects written in different programming languages. The *Internet-Component Broker* is intended to provide interoperability among components adhering to different models by generating glues and wrappers.

3.3. Quality of Service Catalog

In UniFrame, service and service guarantees is integral part of every component UMM specification. It is used during the system generation phase for ensuring service guarantees. The UniFrame Quality of Service (UQOS) [BRA02] framework provides the infrastructure for ensuring service and service guarantees. This framework provides an objective paradigm for quantifying and specifying the quality of software components. It also takes into account the effect of environment and usage pattern on the QoS of the software components. The objectives of the UQOS framework is to objectively quantify

the QoS of software components, standardize the notion of quality of components using QoS catalog, standard approach to incorporate the effect of environment on the QoS of the software components, an approach to incorporate effect of usage patterns on the QoS of components and to provide a specification mechanism to specify QoS of software components.

The QoS catalog lists and provides details of the QoS parameters for the software components. The component developer can use the catalog to identify the QoS parameters relevant for the component. Some of the details which are provided for the QoS parameters in the catalog are

a) The domain of usage, which will assist the component developer to select the QoS parameters relevant for the domain.

b) Whether the QoS parameter varies during runtime based on usage pattern and the operating environment and thereby assisting the component developer to know whether the QoS of the component can be improved by changing the environmental condition and usage pattern.

c) Nature of the parameter, which classifies the QoS parameters according to the characteristics. This classification assist the component developer in making out whether the QoS parameter is a time related (throughput, delay, etc), importance related (priority), capacity related (capacity), etc.

d) Composability of QoS parameter.

e) The methodology for the quantification of the QoS parameter.

f) Interrelationships of the QoS parameter with other parameters.

A part of Turn-Around-Time parameter details, which is depicted in [BRA02], is shown in Figure 3.3.

TURN-AROUND-TIME

Intent:	It is a measure of the time taken by the component to return the result.
Description:	It is defined as the time interval between the instant the component receives a request until the final result is generated.
Motivation:	1. It indicates the delay involved in getting results from a component. 2. It is one of the measures of component performance.
Applicability:	This attribute can be used in any system, which specifies bounds on the response times of its components.
Model Used:	Empirical approach.
Metrics Used:	Mean Turn-around-time.
Influencing Factors:	1. Implementation (algorithm used, multi-thread mechanism etc). 2. Speed of the CPU. 3. Available memory. 4. Process priority. 5. Usage Pattern. 6. Computer Organization. 7. Hardware resources like floating point processor, system bus, I/O devices,etc. 8. Operating System's access policy for resources like: CPU, I/O, memory, etc.

Figure 3.3. Turn-Around-Time Parameter Description

The UQOS takes into account the effect of the environment and the usage patterns on the QoS of the software component by providing the necessary steps to consider the environmental condition. The Figure 3.4 shows the steps indicated in [BRA02] to consider environmental conditions.

For each selected parameter P_i (i=1 to n),
a. If P_i is static,
    i. for each set of representative test cases, t_c (c=1 to n)
       Run the instrumentation code, record the values
    ii. Include the QoS metrics in the UniFrame description of the component.
b. Else, If P_i is dynamic,
   Vary the set of environment variables E_j (j=1 to m) as follows:
   Select a subset E_s (s=1 to k) of E_j
    i. Vary the environment variables in the subset E_s while keeping the variables in the set (E_j- E_s) constant.
    ii. Run the instrumentation code and record the value of the parameter
    P_i for each set of values of environment variables in E_s.
    iii. Plot a graph of P_i versus E_s.
    iv. Prepare a table with values of E_s and P_i.

    v. Include the prepared table in the UniFrame description of the component.

Figure 3.4. Effect of Environment on QoS

The QoS specification mechanism, which will be presented in the next chapter, will provide a way for the component developers to specify the QoS parameters mentioned in the catalog so that a quality aware component is generated.

3.4. UniFrame Glue Generator (UniGGen) Framework

The UniGGen provides the framework for generating glue code necessary to make the heterogeneous components discovered by the URDS, to communicate with each other using a template based approach. The templates provide a mechanism to generate classes and functions based on type parameters. Currently the glue generator framework provides interoperability mechanisms for Java RMI and CORBA components. The template contains the glue generic code required to interoperate between Java RMI and

CORBA components using RMI-IIOP for communication. The UniGGen architecture proposed in [TUM04] is shown in Figure 3.5.

The input to the glue generator consists of the system name along with the list of heterogeneous components, which needs glue for interoperation. The GlueGenerator gets the necessary UMM specification, the component interactions and the technology of the heterogeneous components from the knowledgebase and passes it onto GlueCodeGen and GlueConfigGen. The GlueCodeGen generates the necessary glue code to interoperate using the glue code templates and GlueConfigureGen generates the glue configure code to using glue configure templates. The glue configure code is used to configure the initiator component to the glue and glue to the responder component. An initiator component is one which request services and responder component is one which provides services.

The knowledgebase provides information to the GlueGenerator such as the URL of the UMM specification of the components and component interactions. The UMM specification of the component has a URL which gives the remote location of the component. The component interaction indicates the initiator and responder components. The knowledgebase also contains the glue code templates and the glue configure templates. The framework also provides a GUI for compiling and deploying the glue code.

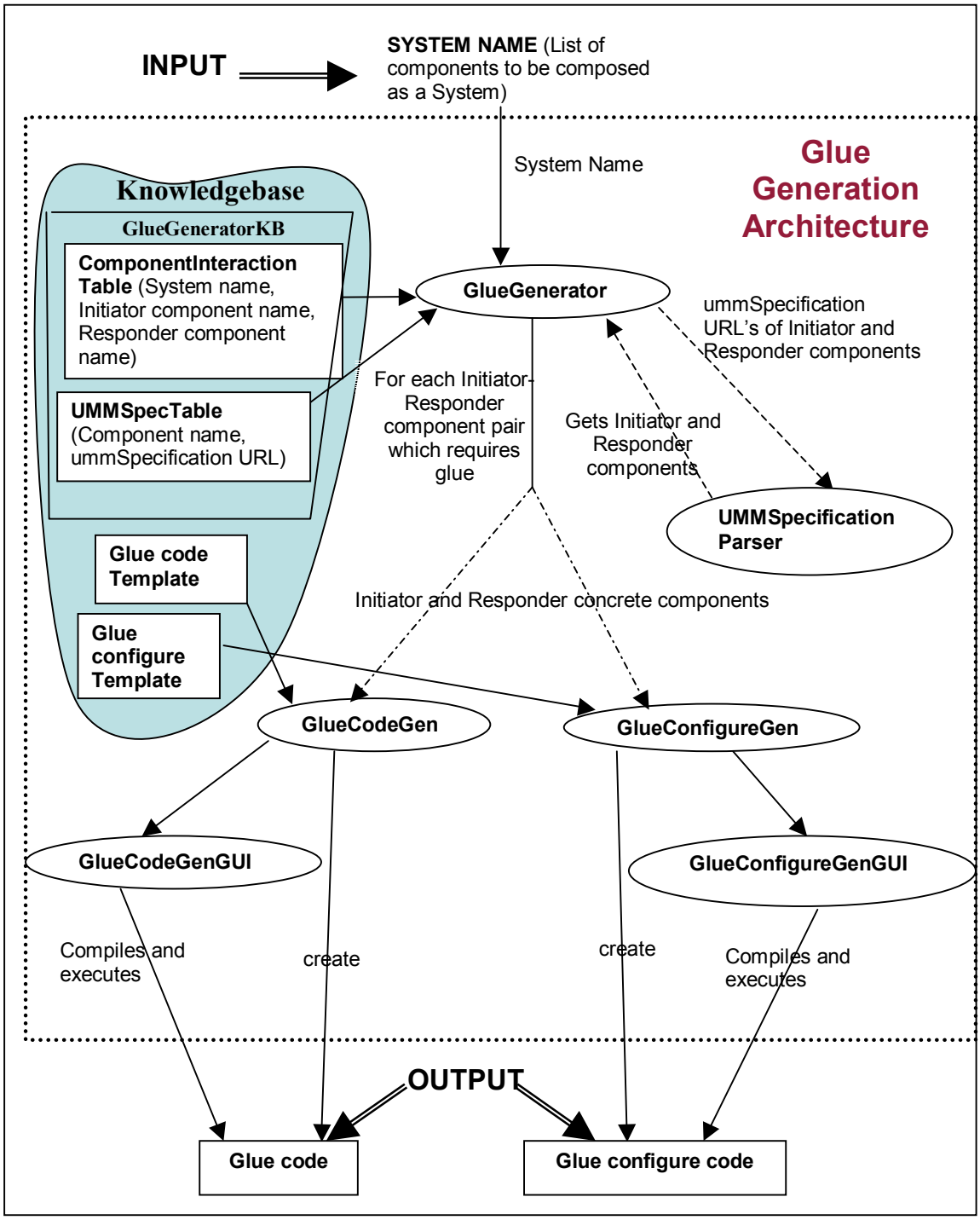

Figure 3.5. Glue Generation Architecture

In this section a brief overview of UniFrame was presented. The glue generator is provided as part of the UniFrame framework for providing interoperability mechanism for components adhering to different distributed component models. The glue generator currently uses templates which are not quality aware. There is a need for quality aware templates to create quality aware glue code. This will ensure that the QoS provided by the system composed of components (including the glue component), satisfies the system integrator QoS requirements.

This thesis provides a QoS specification mechanism for code generators like [GME02] to generate code templates which are quality aware. The following chapter gives the details about the QoS specification mechanism.

4. MODELING QOS PARAMETERS IN CBSD

In the previous chapter an overview of the UniFrame, UMM, QoS catalog and UniGGen was presented. The necessity for specifying QoS during design time of the component such that it aids in development of quality aware glues for interoperability of the heterogeneous components was also indicated. This chapter describes the QoS specification mechanism for generating QoS aware components. It has four parts, namely: The QoS concepts, the QoS profile for providing constructs for specifying QoS, integration of the QoS specification with the functional specification of the system and mapping of the QoS specification onto the relevant parts of the component code that form the system. The details of these parts are explained in the sections of the chapter.

4.1. The QoS Concept

The QoS concepts provide the necessary principles for specifying the QoS during the design phase of the component. The main difficulty in specifying QoS for components is the determination of what constitutes the QoS specification and the elements identified for QoS specification to be able to specify all the QoS parameters that are appropriate for a particular component. The QoS concepts proposed in this section provide a way of specifying all QoS parameters by

- using the classification of QoS parameters mentioned in the QoS catalog [BRA02]; the parameters are classified to be either static or dynamic based on the behavior of the parameters due to varying environmental conditions. The value of a static parameter value does not vary during runtime of the component while the value of the dynamic parameter can vary during run time of the component,

- using the concepts proposed by [AAG01] for identifying the elements which constitute the specification of the dynamic QoS parameters,
- extending the concepts proposed by [AAG01] for identifying the elements which constitute the specification of the static QoS parameters.

The QoS concept model is aimed at identifying the necessary elements which can aid in the QoS specification for the component. The component developer will be able to use the concepts for semantics of the elements that constitute the QoS specification, thereby aiding him to specify the QoS during design of the component.

The OMG has recognized the importance of QoS during design and development phase and has issued a Request for Proposal (RFP) for a UML profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms [OMG02]. The key mandatory requirements are shown in Figure 4.1.

A General Quality of Service Framework

To ensure consistency in modeling various qualities of service, submissions shall define a standard framework or, reference model, for QoS modeling in the context of the UML. This shall include:

- A general categorization of different kinds of QoS; including QoS that are fixed at design time as well as ones that are managed dynamically
- Integration of different categories of QoS for the purpose of QoS modeling of system aspects.
- Identification of the basic conceptual elements involved in QoS and their mutual relationships. This shall include the ability to associate QoS characteristics to model elements (specification), a generic model of the system aspects involved in QoS-associated collaboration and their functional interactions and use cases (usage model), and a generic model of how QoS allocation and decomposition is managed.
- A coherent set of stereotypes, tagged values, and constraints as necessary to represent the identified QoS properties constructing a UML Profile.

A Definition of Individual QoS Characteristics

Submissions shall define QoS characteristics, particularly those important to real-time and high confidence systems, which describe the fundamental aspects of the various specific kinds of QoS based on the QoS categorization identified in the framework. These shall include but are not limited to the following:

- time-related characteristics (delays, freshness)
- importance-related characteristics (priority, precedence)
- capacity-related characteristics (throughput, capacity)
- integrity related characteristics (accuracy)
- fault tolerance characteristics (mean-time between failures, mean-time to repair, number of replicas)

A coherent set of stereotypes, tagged values, and constraints as necessary to represent the identified QoS properties constructing a UML Profile.

Figure 4.1. OMG RFP for QoS Profile

This RFP calls for the classification of different kinds QoS, including the ones which are fixed during runtime and the others which are managed dynamically. It also makes the identification of concepts of QoS and the definition of different kinds of QoS characteristics for different categories as a part of the mandatory requirements. The proposed QoS concept model in this thesis incorporates the necessary concepts for specifying QoS.

4.1.1. QoS Concept Model

The proposed QoS concept model, presented in this thesis, extends the QoS model of [AAG01], which was described in the Chapter 2 (section 2.3). The specifications of throughput and maintainability QoS parameters for a component are used as examples to indicate the need for using and extending the model [AAG01]. Let the constraint imposed on the throughput of the component by a developer be 20 units/sec. By using the model proposed in [AAG01], the QoS statement concept enables QoS statement code to be incorporated in the component to measure and ensure the necessary throughput (20 units/sec). The constraint concept enables the QoS statement to ensure the required QoS (20 units/sec) be provided. The QoS characteristic concept enables identifying the QoS parameter (throughput) for the component. The profile concept will enable us to create multiple sets of QoS statements to take into account the effect of the environment on the QoS of the component. The required QoS and provided QoS relations, specify the QoS that a component expects from the environment and the QoS, a component can provide (20 units/sec). These relations get reflected as statements in the component. This model is able to provide concepts for parameters which vary based on the environmental condition. Now let us take an example to specify QoS parameters whose values do not vary due to environmental condition (like maintainability).

The determination of the static QoS parameter, maintainability, involves testing for lines of code, method coupling, cyclomatic complexity, etc., on the developed component. The semantics of the QoS concept identified in [AAG01] does not provide a way to incorporate static QoS specification that will enable generation of interfaces for

testing the component. For example, the cyclomatic complexity is the number of independent paths in the program. These paths will have statements that perform some functionality of the component and they are not QoS statements. Hence, there is a need to extend the model to take into account the specification of static QoS parameters. The proposed approach in this thesis uses the [AAG01] model to incorporate concepts for dynamic QoS parameters. The model of [AAG01] is presented in a different aspect to just indicate that dynamic QoS concepts can also be viewed in another way. The difference can be seen in provided and required QoS concepts. These are considered as QoS statements in the proposed approach for QoS specification, which reflect the required QoS and provided QoS of the component. The QoS statements relate to a QoS characteristic and the QoS characteristic relates to a component using a profile.

The figure 4.2 shows the extended concept model for QoS. The figure has rectangular boxes, which represents the independent concept elements and the relation between these elements is indicated by using the Generalization and Association representation mechanisms of UML.

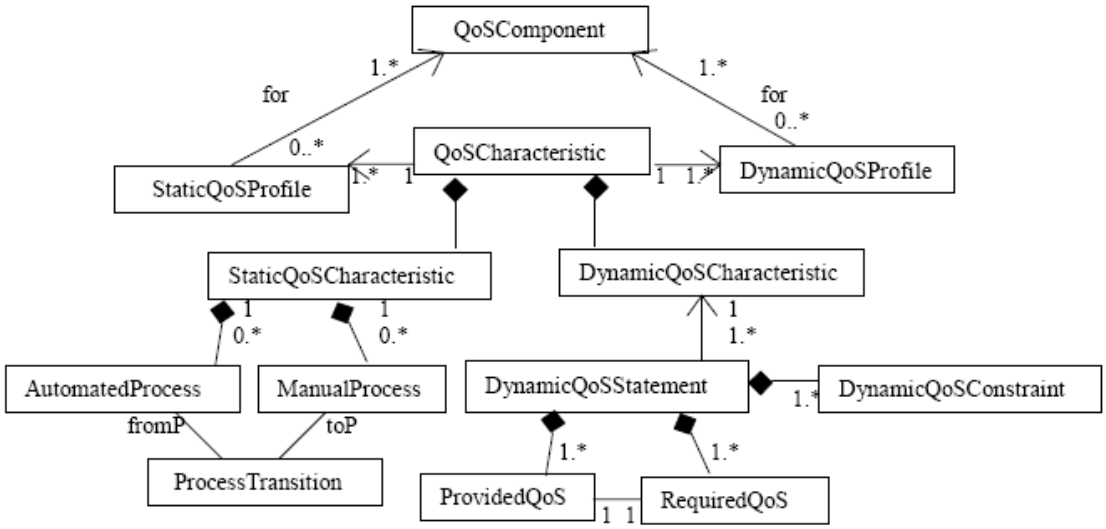

Figure 4.2. QoS Concept Model

A *QoSCharacteristic* is some aspect of QoS which can be identified and quantified. For example, delay, throughput, capacity, maintainability, etc. Some of these *QoSCharacteristics* can vary during runtime and some of them remain constant. This is

indicated by classifying the *QoSCharacteristic* to be dynamic and static respectively. A *QoSCharacteristic* is composed of *StaticQoSCharacteristic* (such as maintainability, dependability, etc) and *DynamicQoSCharacteristic* (such as delay, capacity, etc.). A *DynamicQoSStatement* is composed of *DynamicQoSConstraint.* A *DynamicQoSStatement* specifies the QoS statement that constrains a *DynamicQoSCharacteristic.* A constraint is a restriction on the values of a *DynamicQoSCharacteristic.* The *ProvidedQoS* and *RequiredQoS* are statements which express the QoS provided by the component and the QoS expectation from environment by the component. A *DynamicQoSProfile* relates all the *DynamicQoSStatement* to the *QoSComponent*. A *QoSComponent* concept represents a scenario where the *QoSCharacteristic* statements are applied, for example, an object, a use case, a method. A *DynamicQoSCharacteristic* can be measured by incorporating the *DynamicQoSStatement* in the component. A *StaticQoSCharacteristic* determination process involves conducting external tests on the component. There are different models proposed, that will enable the determination of different *StaticQoSCharacteristic*, for example, the Maintainability model [FRA94] based on software metrics for determination of maintainability of the component and the Dependability model [VOA00] for determination of the dependability of the software component. These models may have concepts that can be automated as well as the concepts that have to be carried out manually. For example, the determination of maintainability of the component using [FRA94] involves the determination of lines of code, that can be automated and the determination of method coupling, which has to be done manually. The *AutomatedProcess* concept represents the part of the *StaticQoSCharacteristic* determination process that can be automated and *ManualProcess* concept represents the part that has to be carried out manually. The *ProcessTransition* specifies the transition from automatic to manual process and vice versa.

The knowledge about the *StaticQoSCharacteristic* determination process will enable the component developer to know about the support (like interface for testing) that a component has to provide for *StaticQoSCharacteristic* determination process. The component developer may specify the information about the StaticQoSCharacteristic

model during the design phase of the component, thereby generating the necessary supporting mechanism to aid the StaticQoSCharacteristic determination process in testing the component. For example, knowledge about the *AutomatedProcess* will enable the component developer to know the part of the process that needs support from the component (such as providing an interface for testing), and the *ManualProcess* will enable him to know about the part of the source code necessary for determining the *StaticQoSCharacteristic* (such as classes needed for determining the method coupling). The component developer could also follow the *StaticQoSCharacteristic* determination model to incorporate the necessary concepts during the design phase of the component. An example of access-control *StaticQoSCharacteristic* is explained in section 4.4.2. The *StaticQoSProfile* relates the process needed for determining the static QoS to the *QoSComponent*.

4.1.2. QoS Statement

The *DynamicQoSStatement* concept shown in Figure 4.3 specifies that the QoS statement can be a *SingleDynamicQoSStatement* or *CompoundDynamicQoSStatement*. The CompoundDynamicQoSStatement is an aggregation of 2 or more *SingleDynamicQoSStatement*. The relations between the *SingleDynamicQoSStatement* in a *CompoundDynamicQoSStatement* are indicated by AND or OR logical operators. The *ProvidedQoS* statement and *RequiredQoS* statement specifies the QoS, provided by the component and the QoS, the component expects from the environment. The association between the *ProvidedQoS* and the *RequiredQoS* indicates that for each *ProvidedQoS* there is a *RequiredQoS*. The *DynamicQoSConstraint* has a *qualification* attribute whose values can be either guaranteed or best-effort. The *constraint* attribute specifies the constraint on the values of the Dynamic*QoSCharacteristic*.

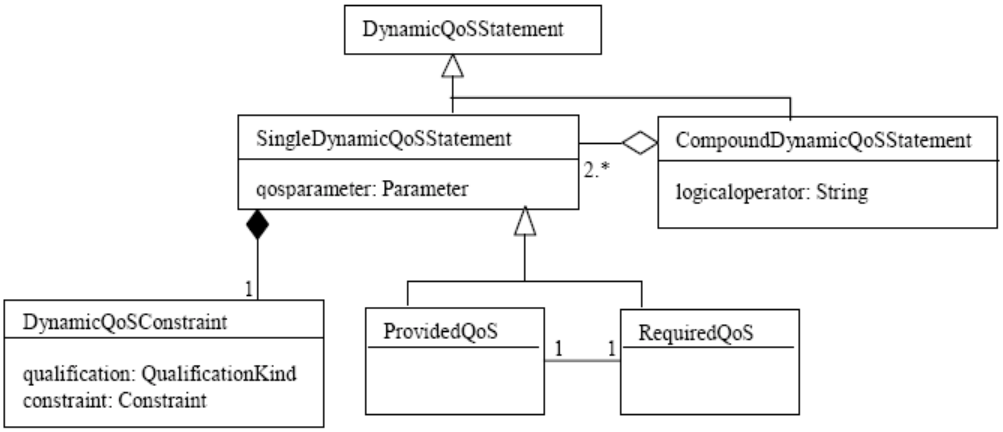

Figure 4.3. Dynamic QoS Statement Concept

4.1.3. Dynamic QoS Profile

The *DynamicQoSProfile* relates the *DynamicQoSStatment* to the *QoSComponent.* The *QoSComponent* expects certain QoS from the environment in order for it provide a certain QoS. The QoS expected from the environment by the component is measured by the component using a set of statements to determine the environment QoS If the environment cannot provide the necessary QoS, then the component can assume a weaker expectation of QoS from the environment. This is enabled by switching to another profile that makes weaker assumptions about the environment QoS. This concept is indicated in Figure 4.4. The *Transition* association has an operation attribute, which specifies the method to be invoked in the component to switch to another set of *DynamicQoSStatement*, that make a weaker assumption about the environment QoS.

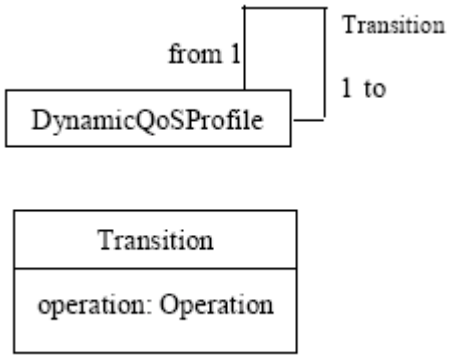

Figure 4.4. Dynamic QoS Profile

4.1.4. Dynamic and Static QoS Characteristic Models

A *DynamicQoSCharacteristic* shown in Figure 4.5 has a domain associated with it. The domain specifies the values the *DynamicQoSCharacteristic* can have. The *direction* attribute specifies whether the increasing or decreasing value is better for the *DynamicQoSCharacteristic*. The *statisticalattribute* attribute provides a way to specify the values using statistical concepts like mean, variance, etc. The *string* attribute indicates whether the QoS characteristic value is undefined or defined. The *QoSCategory* groups related QoS characteristics under one category (e.g., delay and throughput under the Time-related category). Similarly, the *StaticQoSCharacteristic* concept is shown in Figure 4.6.

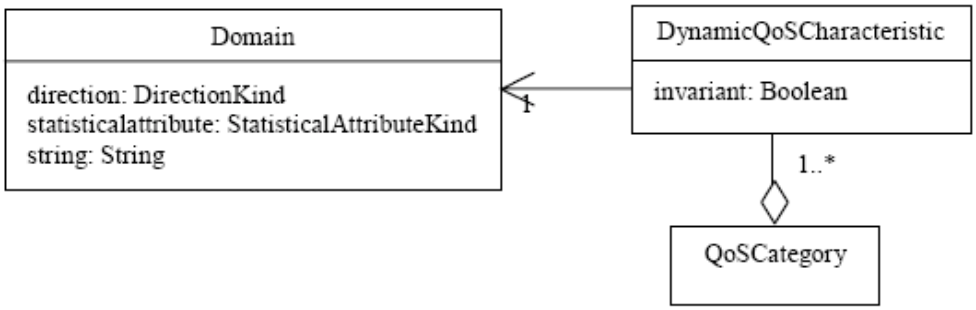

Figure 4.5. Dynamic QoS Characteristic

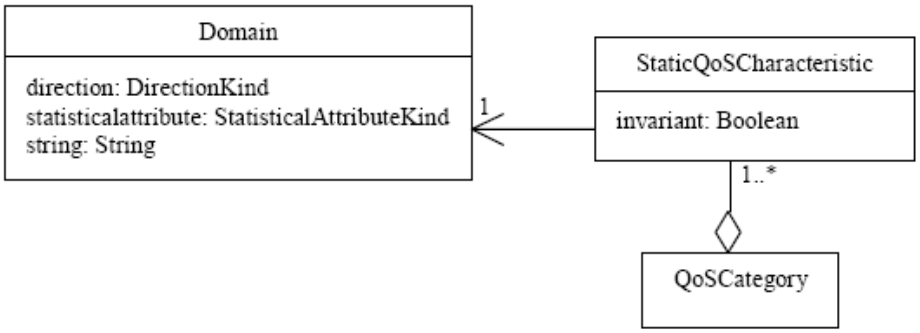

Figure 4.6. Static QoS Characteristic

4.1.5. Static QoS Determination Process

The *AutomatedProcess and ManualProcess,* shown in Figure 4.7, have attributes, which specify the model used for the determination of the QoS and the part, which can be automated or manually carried out. This information, when specified during design of the component, will enable generating the necessary support (e.g., interfaces, classes, etc.) for the QoS determination model. The *ProcessTransition* has attributes that specify the transition from automated to manual and vice versa.

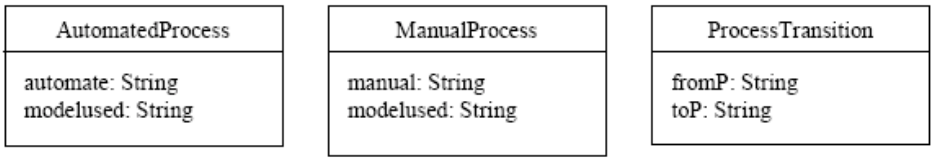

Figure 4.7. Static QoS Process

4.2. UML Profile for QoS

The concepts identified for QoS in the previous section need a representation mechanism so that they can be used for specifying QoS during the design phase of the

component. This representation will enable the component developer to specify QoS for the component during the design phase.

The UML provides a way of representing the functionality of the system in a visual paradigm. It does not provide a way to specify QoS. However UML provides a way to extend its capability to represent a problem domain by its extensibility mechanism. It provides two ways to extend the model: Heavyweight extension and Lightweight extension [UML01]. The Heavyweight extension mechanism involves modifying the existing UML meta-model elements. The Lightweight extension mechanism allows using the existing UML meta-model elements to add new elements that extend the existing model elements. The Light weight extension mechanism is used in our approach so that the extension to the meta-model will not affect other domains which use existing UML meta-model elements for providing constructs. The Light weight extension mechanism involves identifying the stereotypes and tagged values from the concepts.

The Table 4.1 show, the stereotype and tagged values which have been identified for *DynamicQoSProfile* and *Transition* representation.

Table 4.1. Stereotype and Tags for DynamicQoSProfile and Transition

Stereotype	Extends	Tags
<<DynamicQoSProfile>>	Classifier	for
<<Transition>>	Classifier	from to operation

The <<*DynamicQoSProfile*>> stereotype extends the Classifier of the UML meta-model and it has a tag named *for*. The target of the <<DynamicQoSProfile>> is specified by *for* tagged value. The target can be an object, use-case, or component. The tag types are shown in Table 4.2. The classifier is a model element provided in UML meta-model to express the structural (attributes) and behavioral (methods) features.

Table 4.2. Tag Types for DynamicQoSProfile and Transition

Tags	Type
for	Association
from	<<DynamicQoSProfile>>
to	<<DynamicQoSProfile>>
operation	Operation

The tag *for* is of type association. One end of the association is a *DynamicQoSProfile* and the other end of the association is a *QoSComponent.* There may be many *DynamicQoSProfile*'s for a *QoSComponent* and many *QoSComponent*'s for a single profile. This will enable reusability of the QoS specification for different contexts. *from* and *to* tags are of type *<<DynamicQoSProfile>>* and they specify the source and target profile, during a transition. The operation tag is of type operation in UML, and it specifies the behavior to invoke to inform about the change in profile for the component.

The stereotypes and tags for the *DynamicQoSStatement* are shown in Table 4.3. The *<<DynamicQoSConstraint>>* and *<<DynamicQoSStatement>>* extends the classifier of the UML meta-model. The *<<DynamicQoSConstraint>>* has *constraint* and *qualification* as tags. The *constraint* specifies the constraint on *QoScharacteristic* and *qualification* specifies whether the constraint has to be guaranteed or best-effort. The *<<ProvidedQoS>>* and the *<<RequiredQoS>>* extends *<<SingleDynamicQoSStatement>>* and the tags indicate the provided and required QoS of the *QoSComponent*. The *<<SingleDynamicQoSStatement>>* and *<<CompoundDynamicQoSStatement>>* extends *<<DynamicQoSStatement>>*. The QoS statement may have a *qosparameter,* which relates to properties of the *QoSComponent*. The *logicaloperator* specifies the relation between two or more *<<DynamicQoSStatement>>* using an AND or OR logical operator. The tag types are shown in Table 4.4.

Table 4.3. Stereotype and Tags for DynamicQoSStatement

Stereotype	Extends	Tags
<<DynamicQoSStatement>>	Classifier	
<<SingleDynamicQoSStatement>>	<<DynamicQoSStatement>>	qosparameter
<<CompoundDynamicQoSStatement>>	<<DynamicQoSStatement>>	logicaloperator
<<ProvidedQoS>>	<<SingleDynamicQoSStatement>>	provide
<<RequiredQoS>>	<<SingleDynamicQoSStatement>>	require
<<DynamicQoSConstraint>>	Classifier	constraint qualification

Table 4.4. Tag Type for DynamicQoSStatement

Tags	Type
qosparameter	Parameter
logicaloperator	String
provide	<<SingleDynamicQoSStatement>>
require	<< SingleDynamicQoSStatement>>
constraint	Constraint

The stereotype and tags for *DynamicQoSCharacteristic* is shown in Table 4.5. The *<<DynamicQoSCharacteristic>>* extends the classifier of UML meta-model. The *invariant* tag specifies whether the characteristic value must be constant or vary during the runtime. The *<<Domain>>* indicates the values, the QoS characteristic can have and the tags provide a way of specifying the value in terms of statistical concepts (like mean, variance, etc), the string (specifies whether the value is undefined), and direction (specifies whether increasing or decreasing value of the characteristic is better). The tag type is shown in Table 4.6. The Stereotype and tags for *StaticQoSCharacteristic* are shown in Table 4.7. The tag types for *StaticQoSCharacteristic* are same as tag type of *DynamicQoSCharacteristic*. The semantics of stereotype and tags of *StaticQoSCharacteristic* are similar to those of stereotype and tags of *DynamicQoSCharacteristic,* except that the *StaticQoSCharacteristic* remains constant during runtime.

Table 4.5. Stereotype and Tags for DynamicQoSCharacteristic

Stereotype	Extends	Tags
<<DynamicQoSCharacteristic>>	classifier	invariant
<<Domain>>	Datatype	direction statisticalattribute string

Table 4.6. Tag Type for DynamicQoSCharacteristic

Tags	Type
invariant	Boolean
direction	DirectionKind
statisticalattribute	StatisticalAttributeKind
string	String

Table 4.7. Stereotype and Tags for StaticQoSCharacteristic

Stereotype	Extends	Tags
<<StaticQoSCharacteristic>>	classifier	invariant
<<Domain>>	Datatype	direction statisticalattribute string

The Stereotype and tags for the *AutomatedProcess*, *ManualProcess* and *ProcessTransition* are shown in Table 4.8. They extend the classifier of the UML meta-model. The tags indicate the model used for automated or manual processes and the method, that needs to be automated or to be done manually. The tag types are shown in Table 4.9.

Table 4.8. Stereotype and Tags for Process

Stereotype	Extends	Tags
<<AutomatedProcess>>	Classifier	modelused automate
<<ManualProcess>>	Classifier	modelused manual
<<ProcessTransition>>	Classifier	fromp Top

Table 4.9. Tag Type for Process

Tags	Type
modelused	String
automate	String
manual	String
fromP	String
toP	String

The stereotype and tag for *StaticQoSProfile* is shown in Table 4.10. The <<*StaticQoSProfile*>> extends the classifier and it has the tag *fo*r, which relates the profile to the QoSComponent. The tag is of type association. The *StaticQoSCharactersitic* does not vary during runtime, so the concept of profile transition does not exit.

Table 4.10. Stereotype and Tags for StaticQoSProfile

Stereotype	Extends	Tags
<<StaticQoSProfile>>	Classifier	for

4.2.1. Graphical Representation of Stereotypes and Tags

This section shows a graphical representation of the stereotypes that enable the component developer to specify QoS in a visual paradigm such that generation of QoS code using techniques such as [GME02] can be automated.

DynamicQoSProfile:

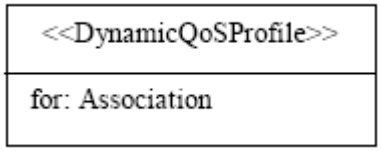

Figure 4.8. Visual Representation of DynamicQoSProfile

Transition:

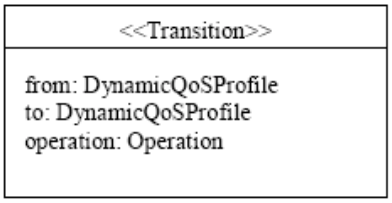

Figure 4.9. Visual Representation of Transition

DynamicQoSStatement:

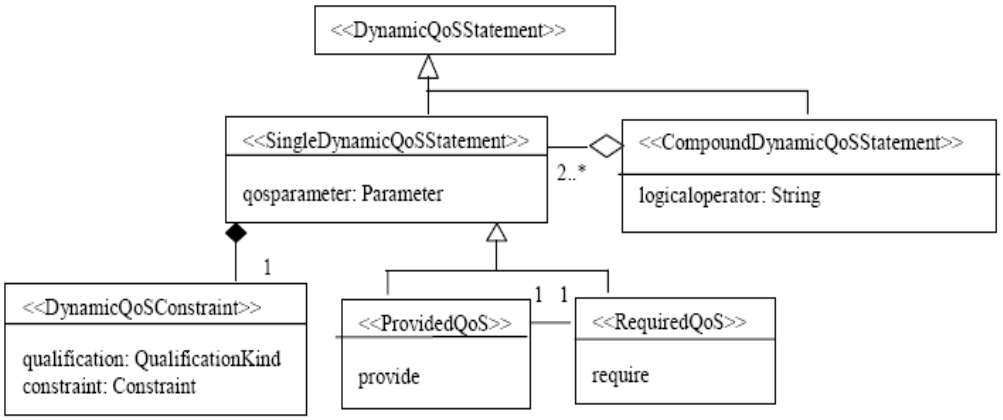

Figure 4.10. Visual Representation of DynamicQoSStatement

DynamicQoSCharacteristic:

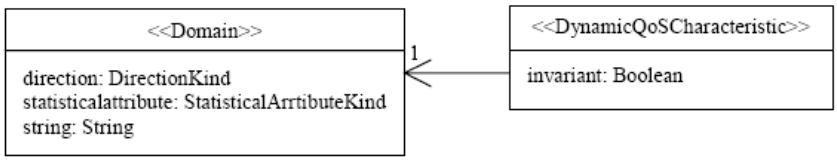

Figure 4.11. Visual Representation of DynamicQoSCharacteristic

StaticQoSCharacteristic:

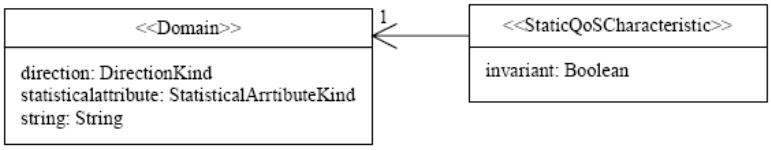

Figure 4.12. Visual Representation of StaticQoSCharacteristic

StaticQoSProfile:

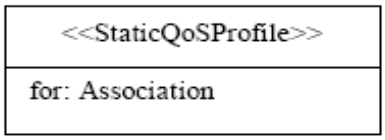

Figure 4.13. Visual Representation of StaticQoSProfile

AutomatedProcess:

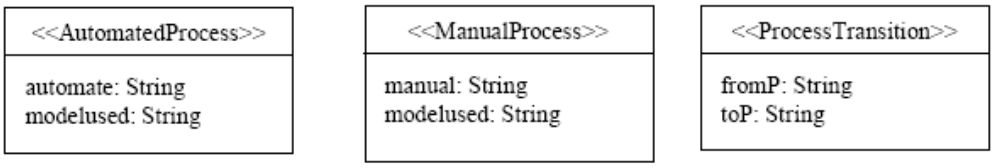

Figure 4.14. Visual Representation of Automated Process, Manual Process and Process Transition

4.2.2. OCL Expression for Precise Constraint

The *DynamicQoSStatement* expresses the constraints using an OCL expression. As indicated in Chapter 2 (section 2.5), the OCL expresses constraints on the functional model and has essential constructs to specify them. The *DynamicQoSStatement* model element, which was shown in Figure 4.10, has, a *SingleDynamicQoSStatement,* which is composed of *DynamicQoSConstraint*. The *constraint* attribute of the *DynamicQoSConstraint* specifies the precise constraint using an OCL expression. The OCL expression used for specifying a constraint in the visual model has only the constraint expression part. The context for the constraint is provided by the *for* attribute of the *DynamicQoSProfile* modeling element. For example, assume a component that has a method named getData, which performs the functionality of retrieving the data from the database. The constraint the component developer wants to have on this method is that the delay for retrieval should be less than 50 ms. The QoS specification for the method getData is as shown in Figure 4.15. The OCL expression used for the constraint is "delay < 50 ms". The *delayProfile* specifies the context for which the constraint has to be applied. This QoS specification mechanism uses both visual and text for specifying QoS.

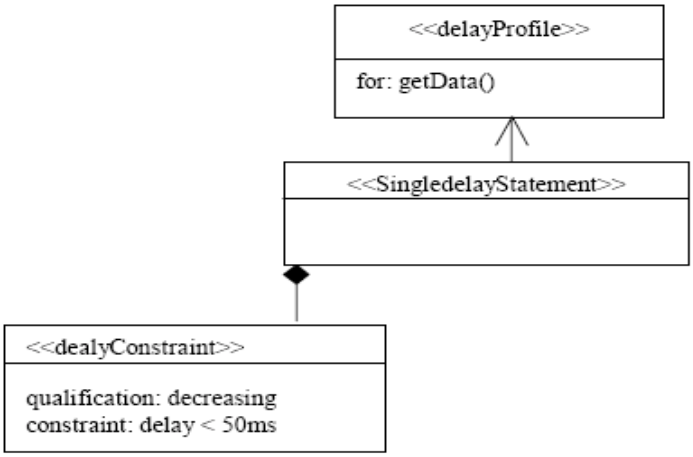

Figure 4.15. Delay Constraint Using OCL in DynamicQoSStatement

4.3. Relating Functional and Non-functional Models

The UML provides a graphical paradigm to express the functionality of the system at a high level of abstraction than implementation level. It provides graphical elements that can be combined to form a diagram. For example, a class diagram in UML has the following visual elements:

- Class representation
- Association
- Inheritance and Generalization
- Aggregation
- Interface and realization
- Visibility

The UML also provides rules for combining the visual elements. The component developer may use all or some of these visual elements to represent the class diagram for the system during the design phase. The other, diagrams that are provided in the UML are

- Object diagram
- Use case diagram
- State diagram
- Activity diagram
- Collaboration diagram
- Component diagram
- Deployment diagram

These diagrams express multiple views of the system and a set of these multiple views constitute a model. The diagrams are used to represent the functional view of the system and hence it is called the functional model of the system.

The UML capability for specifying QoS was enabled by providing visual elements (stereotypes and tagged values), which were presented in the section 4.2.1 of this chapter. These visual elements enable a component developer to provide a QoS view of the functional model of the system. The developer community has considered the QoS as non-functional attributes of the systems; hence, the QoS view is termed the non-functional model in this thesis. Since the non-functional model specifies the QoS view of

the corresponding functional model, a way of relating both of the models is needed. The thesis proposes an approach to relate the two models using the Collaboration diagram.

The Collaboration diagram depicts [TUT02]

- The Objects involved in interactions.
- The messages sent between the objects during interactions.
- Sequencing of the messages involved in the interactions.
- Associations of the objects involved in the interactions.

The messages sent between objects in a Collaboration diagram are represented by arrows, which point to the receiving object near the end of the association line between two objects. A label near the arrow shows the content of the message. The Sequence diagram of UML is considered to depict the same information as collaboration diagram, but the emphasis is on the time and not on the content of the message being sent between the two objects. An arrow from one object to another object in a Sequence diagram depicts only that a message was sent at that specific time. Thus, a sequence diagram may, or may not, depict the message content. However, the semantics of collaboration diagram require that the message content and other information be included in the diagram, so it provides a sure way to depict the QoS specification information.

The Collaboration diagram emphasizes on the content of the message, so a context for the applicability of the QoS model can be indicated. The Figure 4.16 shows one such diagram where an endToEndDelay non-functional model is related to the functional model depicted in the collaboration diagram. The functional model depicts the two collaborating objects (*DocumentTerminal* and *DocumentServer*). The message getDocument (name) is being sent from *DocumentTerminal* to *DocumentServer*. This message is annotated with QoS specification. The **endToEndDelay* specifies the QoS for the method and **capacity* specifies the number of threads or requests the DocumentServer object should handle.

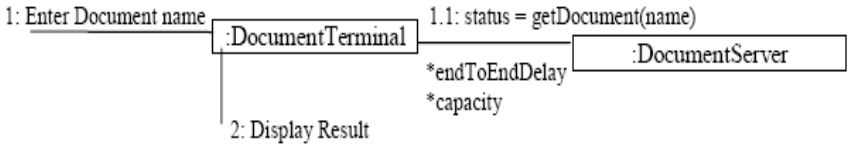

Figure 4.16. Collaboration Diagram Indicating the Relation Between Functional and Non-functional Model

A general algorithm to interpret the annotated collaboration diagram is described below.

In a collaboration diagram
For objects depicted in the collaboration diagram
Check if messages are sent between objects or within an object
if a message is sent check for QoS specification annotation
If QoS annotation exists then
Apply the relevant QoS specification for the object /method in the object to which the message is directed.

4.3.1. Reusability of Non-functional Model

Specifying QoS involves annotating the collaboration diagram to indicate the context for applying the QoS specification. As QoS specification is a separate view of the corresponding functional model, the *for* attribute of the *DynamicQoSprofile* specifies the context where the QoS specification has to be applied. This attribute may specify many contexts for which a QoS specification can be applied. An example of multiple contexts is shown in Figure 4.17.

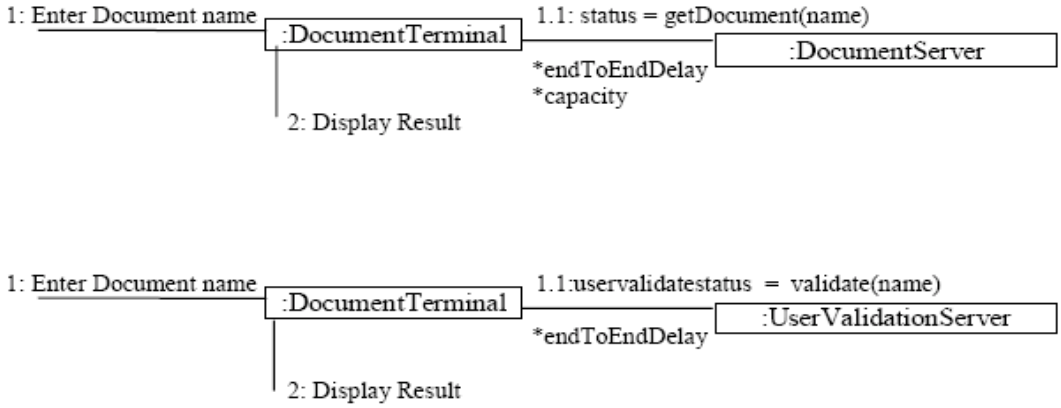

Figure 4.17. Collaboration Diagrams Indicating Reusability

The collaboration diagrams shown in Figure 4.17 depict the interaction of *DocumentTerminal* object with the *DocumentServer* and *UserValidationServer*. The message sent from *DocumentTerminal* to *UserValidationServer* is annotated with *endToEndDelay and the message sent from *DocumentTerminal* to *DocumentServer* is annotated with *endToEndDelay. It implies the same QoS specification to be applied for both the methods. The *for* attribute in *DynamicQoSProfile* element of the QoS specification will indicate these methods, thereby providing the association of QoS specification to both the methods. The *endToEndDelayProfile* is shown in Figure 4.18.

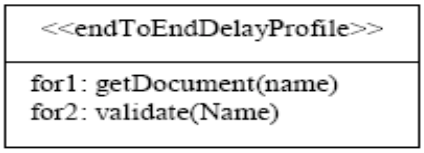

Figure 4.18. Multiple Context Specification using an endToEndDelay Profile

4.4. Mapping of Non-functional Model onto the Code

The previous sections of this chapter proposed a specification mechanism for the QoS, during design of the system. This section provides an algorithm for transforming the QoS parameter specification onto the necessary code for a component implementation.

4.4.1. Mapping of Dynamic QoS Specification onto the Code

A general algorithm for model transformation is presented first and then an example is illustrated by taking a parameter into consideration and applying the transformation algorithm. The transformation process chosen here involves model-to-model transformation and then, transformation of the refined model to code. However, a model can be transformed into code without refinement. The model refinement helps in generating minimum code (e.g., minimum number of classes) and provides the same functionality as the one which was generated without refining the model.

A general algorithm for model-to-model transformation for a dynamic QoS specification is given below:

For a dynamic QoS parameter specification,

i. Merge the functionality of the Domain with the QoS characteristics.

ii. Merge the functionality of CompoundQoSStatement, ProvidedQoS, RequiredQoS and DynamicQoSConstraint into SingleQoSStatement.

ii. Merge the functionality of Transition into QoSProfile.

A general algorithm for transforming the refined model of dynamic QoS parameters into code is given below.

For a dynamic QoS parameter specification

i. Create a class named DynamicQoSProfile with relevant attributes, which are indicated in the UML representation.

In the DynamicQoSProfile class,

i. Create a reference to the object or the method for which the DynamicQoSProfile has to be applied. The object or method is obtained from the collaboration diagram.

ii. Create a reference to another DynamicQoSProfile which will be called when the current profile needs to be changed. Obtain the target DynamicQoSProfile from the *to* attribute of the current DynamicQoSProfile.

iii. Create a method which will invoke a method in the object for which the QoS specification is depicted. The method in the object will let the object know about the change in profile due to environmental conditions.

iv. Create a reference to the DynamicQoSStatement class that passes the object to the DynamicQoSStatement class.

ii. Create a DynamicQoSStatement class with the attributes indicated in its UML representation.

In the DynamicQoSStatement class,

i. Create the necessary statements for expressing the constraints on the object or method of the object.

ii. if multiple constraints are specified then, use the logical operators, for indicating the relation between the statements specifying constraint.

iii. Create the RequiredQoS statement, which specifies the QoS required by the object or method.

iv. Create the ProvidedQoS statement which specifies the QoS provided by the component if the required QoS is met.

v. Create a reference to QoSCharacteristic class to obtain information regarding the values of the QoSCharacteristic.

iii. Create the DynamicQoSCharacteristic class with necessary attributes for providing information about the values of the QoS parameters.

Figure 4.19 shows the dynamic QoS parameter specification elements for throughput. Each element could be transformed into classes and references created for class interactions. But some of the elements provide little functionality and can be merged with other elements. For example in Figure 4.19, the QoS characteristic throughput, has a domain that specifies the values, it can have. A reference to the Domain class can be instantiated in the Throughput class, which enables, the Throughput class to know the values, a throughput can have. The Domain class does not provide any other functionality, so its existing functionality (attributes) can be merged into the throughput class.

The UML provides the elements to emphasize on the individual concepts. While transforming them onto code, the model can be refined and another model can be generated that merges some of the concepts. The approach followed here for transformation uses merging of some concepts.. One such refined model, where the functionality of some of the elements is merged together is shown in Figure 4.20. This model will enable generation of fewer classes compared to the original model.

In the Figure 4.19, the concept of CompoundThrougputStatement, which specifies the relational operator between the two SingleThrougputStatement can be merged into SingleThrougputStatement with a logicaloperator attribute added to it. The logicaloperater in SingleThroughputStatement still relates the statements. Similarly, the ProvidedQoS and RequiredQoS can be merged into the SingleThroughputStatement. The attributes of the ThroughputConstraint concept can be added to the SingleThroughputStatement.

The Domain attributes can be added to the Throughput characteristic model element as they specify only the values the Throughput characteristic can have. The

Transition model element attributes can be added to the ThrougputProfile. The transformed QoS specification model is shown in Figure 4.20.

Throughput parameter: It has been defined in [BRA02] as the response time of the component. The specification for Throughput QoS parameter is shown in figure 4.19.

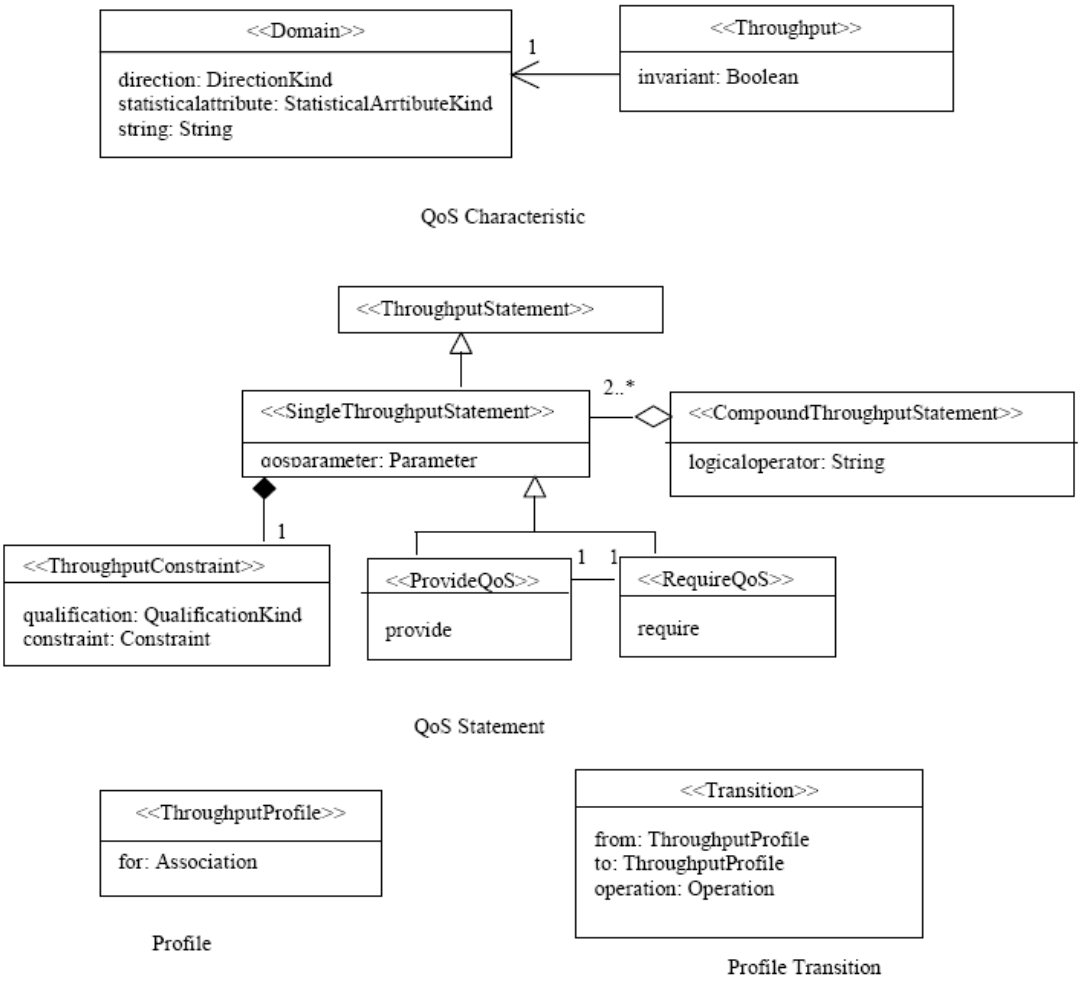

Figure 4.19. Throughput QoS Specification

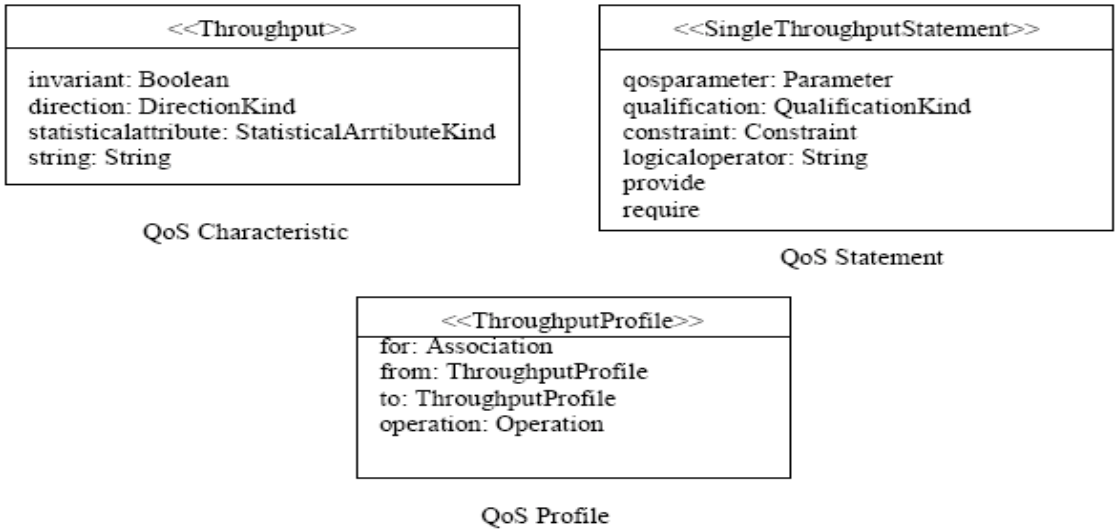

Figure 4.20. Transformed Throughput Model

Transformation of the model depicted in Figure 4.20 onto the component code is explained below. This transformation specifies the creation of necessary classes and methods for measuring and enforcing QoS.

1. Create a class named ThroughputProfile with attributes as indicated in the UML representation. In the ThroughputProfile,

Create a reference to the object or the method for which the Throughput profile has to be applied. The object or method is obtained from the collaboration diagram.

Create a reference to another Throughput profile which will be called when the current profile needs to be changed. The target Throughput profile is indicated by *to* attribute.

Create a method which will invoke a method in the object for which the QoS specification is depicted. The method in the object will let the object know about the change in profile due to environmental conditions.

Create a reference to SingleThroughputStatement class and pass the object to the SingleThroughputStatement class.

2. Create a SingleThroughputStatement class with attributes indicated in the UML representation. The SingleThroughputStatement expresses the constraint. In the SingleThrougputStatement,

Create the necessary statements for expressing the constraints on the object or method of the object. For example, the Throughput constraint of 20 executions/ms on a method will incorporate the line of code as shown in Figure 4.21. The statements measure the time required to execute the method and average the time for 50 method calls. These statements vary based on the QoS parameter. For capacity parameter, the statements will provide a multithreading capability for the component. A tool can be programmed to add the corresponding statements related to the QoS parameter.

```
accumulate = 0;
While ( I <=50 )
Time1 = startTimer()
objectName.Method()
Time2 = stopTimer()
accumulate += (Time2 – Time1)
end while
throughput =  50/ accumulate
if(throughput >= 20) //Single constraint
output result
else
output low throughput
```

Figure 4.21. Throughput Constraint Realization Code

For multiple constraints, use the logical operator for indicating the relation between the statements specifying constraint.

Create the RequireQoS statement, which specifies the QoS required by the object or method. (throughput >=20) in Figure 4.19 reflects a statement which is required QoS for outputting the result.

Create the ProvideQoS statement that specifies the QoS provided by the component if the required QoS is met.

Create a reference to the Throughput class to obtain information regarding the values of the Throughput. The information can be how the value of the QoS parameter is expressed and whether increasing or decreasing value is good.

3. Create a Throughput class with necessary attributes for providing the information about the values of the QoS parameter.

This algorithm will enable instrumentation of the necessary dynamic QoS parameter code that will measure and ensure the QoS of the component.

4.4.2. Mapping of Static QoS Specification onto the Code

Determination of static QoS will involve external tests on the component as static QoS involves judging the characteristics of a developed component. Some of the characteristics are: how well the component has been designed, how easy it is to change the functionality of the component, etc. There are many models that provide concepts for determining different static QoS. These concepts vary from approach to approach. So, considering incorporation of static QoS parameters during design time will involve the incorporation of the concepts of an approach as well as providing the necessary interface to test the static QoS.

Consider the model for access control [BUR03] shown in Figure 4.22. The access-control model depicts the necessary concepts that can be used for incorporating guards and security policies into the component code. It also can act as a standard for

creating the test cases to test the access-control QoS characteristics. The stereotypes and tagged values can be identified from the concepts and a profile for access-control can be created, which will enable modeling of access-control QoS, thereby providing a way to specify access–control QoS during the design phase of the system as well as to create test cases to test the access-control QoS. This access-control profile will help in automating the process of placing necessary guards for the system and generating interfaces for testing. The test cases generated based on this model will enable the determination of access-control QoS for the component.

The model shown in Figure 4.22 specifies that a guard consults with the login manager, which in turn consults with the Authentication server to verify the authenticity of the user of the system. Once the user is authenticated, the guard consults with the Access manager to find the access policy for the user. The access policy varies based on the security context. The context can be a group, accessId, role, or dynamic property. The Access manager consults with the Authorization server for providing the access privileges. The Authorization server consults with the appropriate Decision authority for authorizing the user. The Decision authority consults with the Access policy evaluator. The Access policy evaluator evaluates the access policy for the user and returns the result, which is sent down the hierarchy to let the guard know about the policy for the user.

The collaboration diagram is used to determine where the static QoS model has to be applied. The access-control model shown in Figure 4.22 does not involve any manual intervention concepts. The entire process can be automated, so this is specified using the AutomatedProcess element of the UML profile for QoS. The AutomatedProcess element in Figure 4.23 specifies the model used and the part that needs to be automated. In this case the model used is the access-control model shown in Figure 4.22 and the process that needs to be automated is the placement of guards and appropriate security policies for the user. The stereotypes identified for this purpose from the [BUR03] model is shown in Table 4.11. These stereotypes will enable incorporation of access-control mechanism in the component code and provide an interface to test the access-control. It

also acts as a standard to create test cases to test the access control. A sample test case could be to find out whether the access manager checks for the proper policy for the user.

Transformation rules for mapping static QoS parameters varies for different static QoS parameters as the approach for determination processes are different for different parameters. Therefore, generalized mapping rules cannot be determined. It is left to the tool provider to incorporate the necessary transformation rules for automating the incorporation of static QoS parameters for an approach. However, the process will still involve refining the model by transforming the model to another model. The transformation rules for the transformed model will provide the necessary mechanism for generating the necessary QoS code in the component. The transformation of model to code will still involve generation of classes and reference and required statements and interfaces in the component

Table 4.11. Stereotype and Tags for Access Control

Stereotype	Tags	Tag type
Guard	consultsLM consultsAM	String String
AccessManager	consultsAS consultsAttibuteServer	String String
AuthorisationServer	consultsDA	String
DecisionAuthority	consultsAPE	String
AccessPolicyEvaluator	evaluatesAP	String
AccessPolicy	composeAR	String
SecurityContext	name	String
Role	has	User

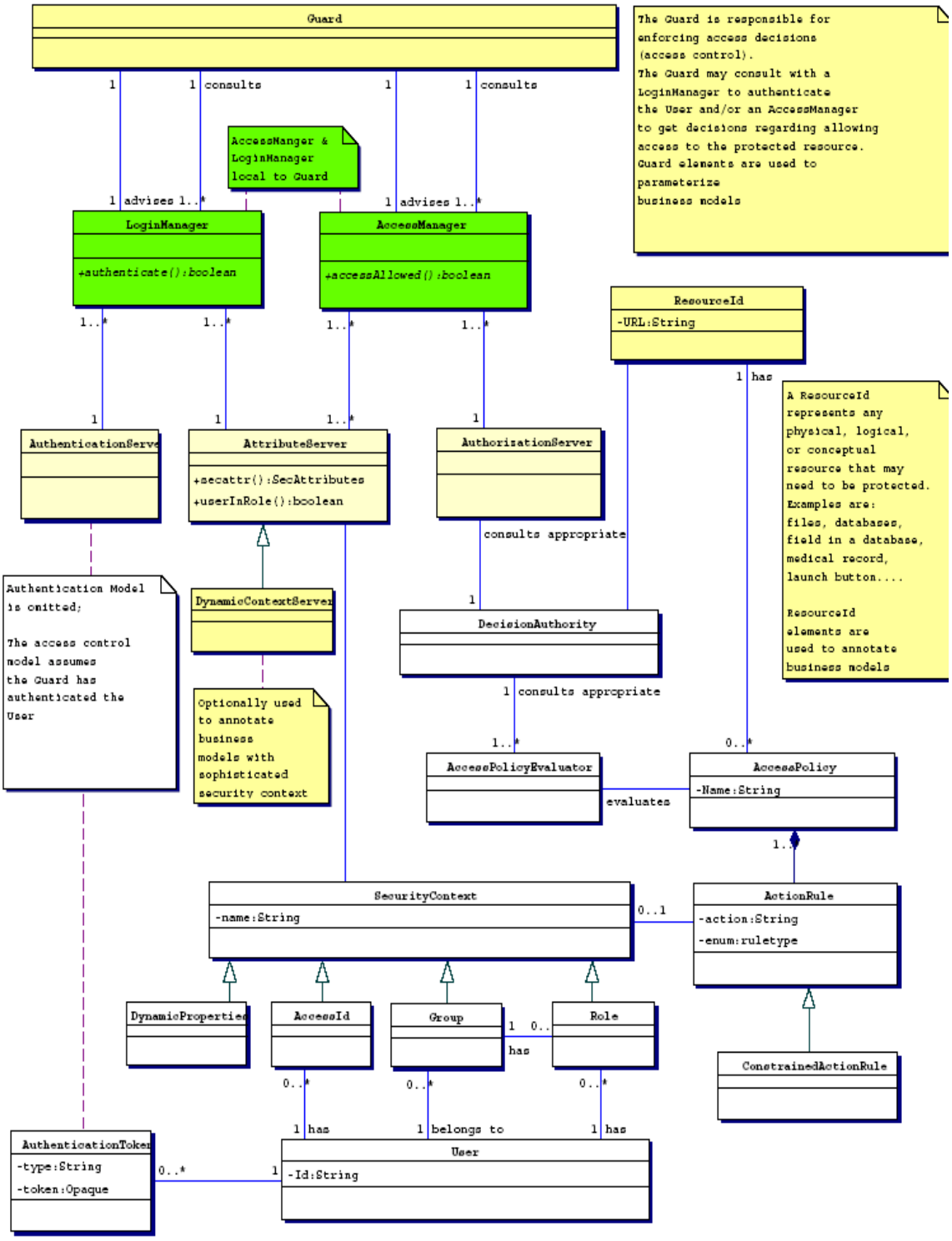

Figure 4.22. Access Control Model

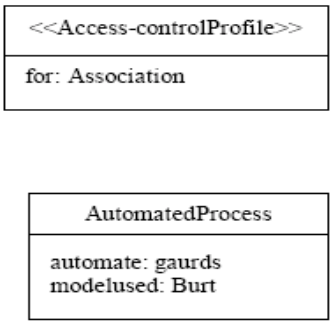

Figure 4.23. Access Control Specification

This chapter presented an approach for specifying QoS which involved determination of concepts of QoS, creation of UML profile for QoS, relating the functional and non-functional models and model-to-code transformation using manual approach. It also showed how to specify a parameter using the QoS model. In the next chapter a case study from the document management system domain is presented to illustrate the applicability of the QoS specification mechanism in real world scenarios.

5. CASE STUDY

The Chapter 4 of this thesis presented a QoS specification mechanism which will enable generation of QoS aware components. Chapter 3 of this thesis presented the UniFrame approach for automating the process of integration of heterogeneous components to create distributed systems that conform to quality requirements. The QoS catalog, a part of the UniFrame framework, was also presented, which will enable the component developer to choose the QoS parameters relevant for the component being developed. The QoS specification mechanism presented in Chapter 4 will enhance the QoS catalog by providing information about specification of the QoS parameters. This will enable the component developer to specify the QoS parameter chosen from the QoS catalog during the design phase of the component for incorporating the QoS in the component. The QoS specification mechanism will also assist in generating quality aware templates and thereby enabling generation of quality aware glues using the UniGGen framework in UniFrame.

This chapter presents a case study that will show the applicability of the QoS specification mechanism. A case study from the document management system is chosen for the purpose. The document management system handles management of documents, and, if the system is associated with a domain such as defense, it has to provide proper QoS, like access-control, quick access of documents, etc. A document management system for a defense domain provides scenarios where the QoS specification can help in generating a quality aware system.

The document management system involves management of documents, such as storing a document, retrieving a document, deleting a document, listing documents, creating a document, authenticating the user and providing a user interface to the system.

The system integrator who wants to build a document management system using the UniFrame approach will present a query, that will detail about the system and the QoS required from the system. Based on the information present in the knowledgebase, the components which are needed for a document management system and the QoS needed for each component (based on composition and decomposition rules [SUN02] for the system) are determined and the search process for the components is initiated. Suppose for example the components needed for building the system were a DocumentTerminal, a UserValidationServer and a DocumentServer, the search process is initiated to discover these components on the network. The component developers would have advertised the QoS provided by the functionality of their components in the corresponding UMM specification format.

The DocumentTerminal component will provide the user interface for the system for storing, retrieving, deleting, creating and listing documents. The component will also provide an interface for user authentication. The DocumentServer component will provide the actual functionality of storing, retrieving, deleting and accessing, and creating the documents. The UserValidationServer component will authenticate the user of the system. The Figure 5.1 shows the class diagram for the document management system.

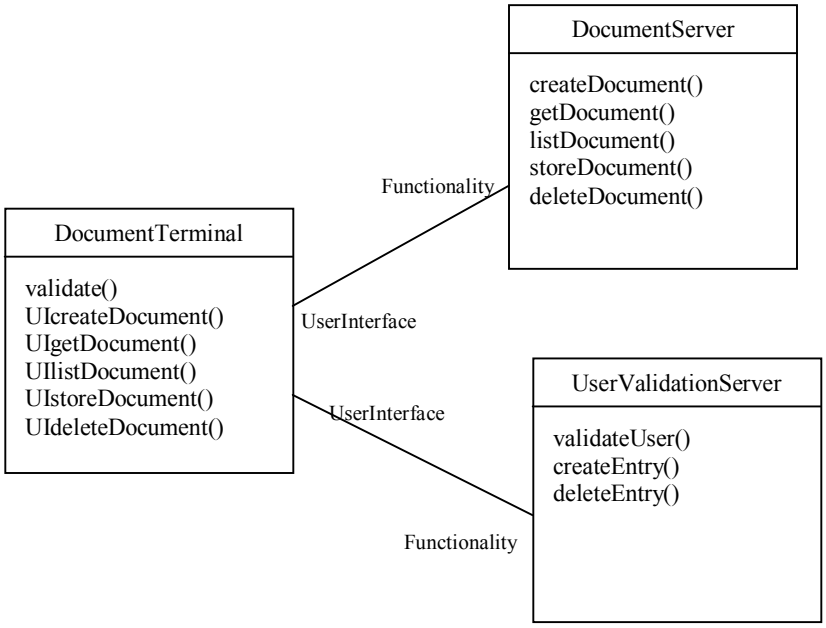

Figure 5.1. Class Diagram for Document Management System

The UniFrame knowledgebase, also contain information about functionality of the system. The information about the functionality could be represented by using the features provided by UML. (e.g., class diagram, sequence diagram, collaboration diagram, etc). Figure 5.2, depicts one such piece of information, which gives details about the timing and sequence of messages sent between interacting components and the life of each object in the system. This information is represented using the sequence diagram of UML.

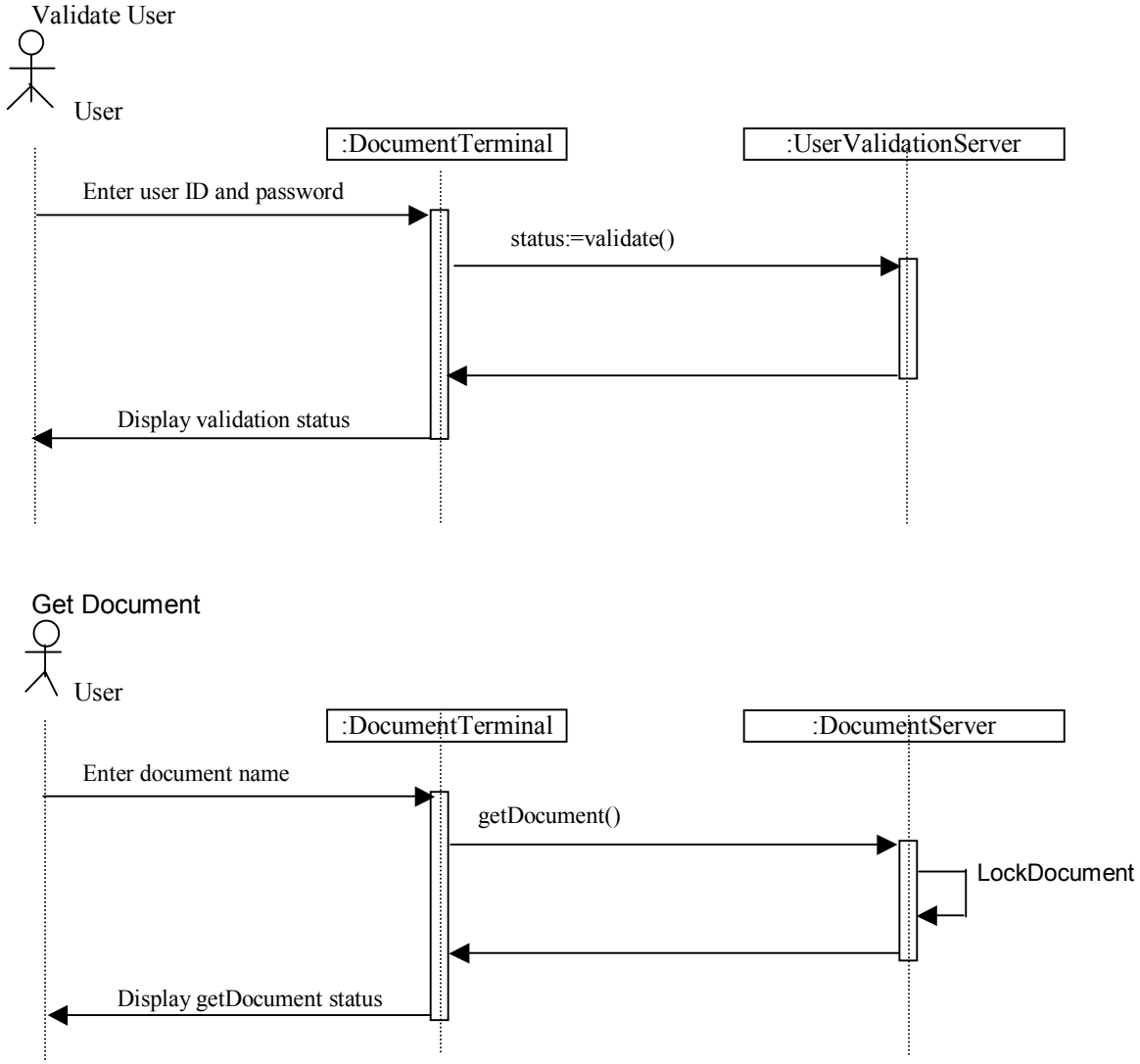

Figure 5.2. Sequence Diagram for Validating User and Retrieving Document

The sequence diagram, shown in Figure 5.2 indicates the interaction of components for validating the user and retrieving the document. The vertical rectangular box below each object specifies the life time of the objects required for successful completion of the interaction and the ordering of arrow indicates the timing of the messages.

The collaboration diagram indicates the interaction of components along with the content of the message being sent during the interaction. Figure 5.3 shows the collaboration diagram for DocumentTerminal, DocumentServer and UserValidationServer.

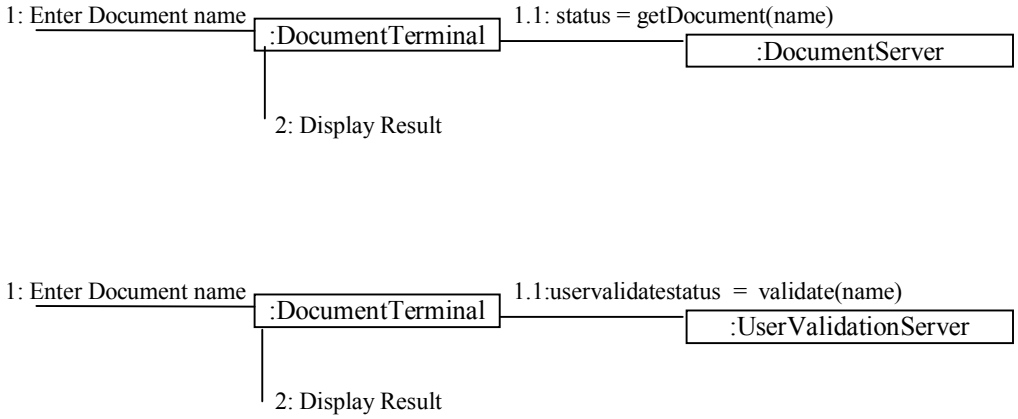

Figure 5.3. Collaboration Diagrams for Document Management System.

The other diagrams that convey information about the functionality of the system are not depicted in the case study. The QoS specification mechanism proposed in this thesis assumes that the information about the functionality of the system will be provided by the knowledgebase.

When the search process is finished, it returns the list of discovered components. There may be multiple components discovered that provide the same functionality. For example the discovery process may return two DocumentServer components that satisfy the QoS. The system integrator may choose the components that can provide the best QoS and use it to build the system. In some cases the discovery process may not be able to find the necessary components, in which case there is a need for developing the component so that a system can still be built. The component developed needs to provide the required QoS. The component can be developed by writing the entire code or use the existing technologies like [GME02], which provides a framework to generate code automatically based on the information about the functionality of the system represented in the visual paradigm. The QoS specification mechanism presented in Chapter 4 can be used to represent the QoS specification in [GME02] and the incorporation of rules for relating the non-functional and functional model and the rules for generating the code from the related model (the non-functional model applied to the functional model) will enable [GME02] to generate code that is quality aware.

Assume, for example, that the discovery process is not able to find the DocumentServer component and an automatic code generation mechanism is used to generate the code, for that component. The process followed for it involves representing the functionality of the component along with its interaction with the other components using a visual paradigm.

The class diagram shown in Figure 5.1 depicts the relationship of components. It also indicates the methods that are part of the components. The sequence and collaboration diagrams for the DocumentServer component, which were shown in Figures 5.2 and 5.3 respectively, indicate the interactions of the components in the system with one emphasizing time and the other emphasizing the messages being sent.

To generate a quality aware code for DocumentServer, the QoS, which has to be provided by the component, has to be determined. Assume that the system integrator had requested a document management system that can retrieve the documents within 120 ms. Based on QoS information presented by the system integrator, the knowledgebase determined that the QoS provided by DocumentTerminal (interface to the user) to be 120

ms. Since DocumentTerminal uses the functionality provided by DocumentServer, the DocumentServer has to provide the document retrieving functionality within 80 ms.

The class diagram shown in Figure 5.1 indicates that the DocumentServer component provides the document retrieving functionality using the getDocument method. The DocumentServer has to impose a constraint on the time to execute (here 80ms) for the method getDocument. The QoS parameter chosen from the QoS catalog for this QoS is endToEndDelay (the time to execute the method). The QoS specification for the method using the proposed approach in this thesis is shown in Figure 5.4. The endToEndDelay for the method should not be more than 80ms. The specification indicates that a decreasing value of the endToEndDealy is good (meaning desirable). To take into account the possibility of providing response (based on environmental conditions) in less than 80ms, a profile is created that has statements which can measure and ensure QoS of less than 80ms. For this example, an endToEndDelayprofile1, to ensure response within 60ms is created along with the endToEndDelayprofile2, which will ensure a response within 80ms. The endToEndDealyProfile1 says that the profile is for the method named getDocument () and this profile will ensure the endToEndDelay of 60ms. The transition indicates the endToEnddelay profile to switch to in case the environment cannot provide the QoS needed by the component (in this case it is endToEndDelayProfile2). ProvideQoS statement specifies that the method will execute in 60ms provided that it gets the result from its database access within 50ms. The database is assumed to be embedded in the DocumentServer for this example. The second profile (endToEndDelayProfile2) specifies the second set of statements which assume a weaker QoS from the environment. The second profile states that the method will execute in 80ms provided that the database access happens in 70ms.

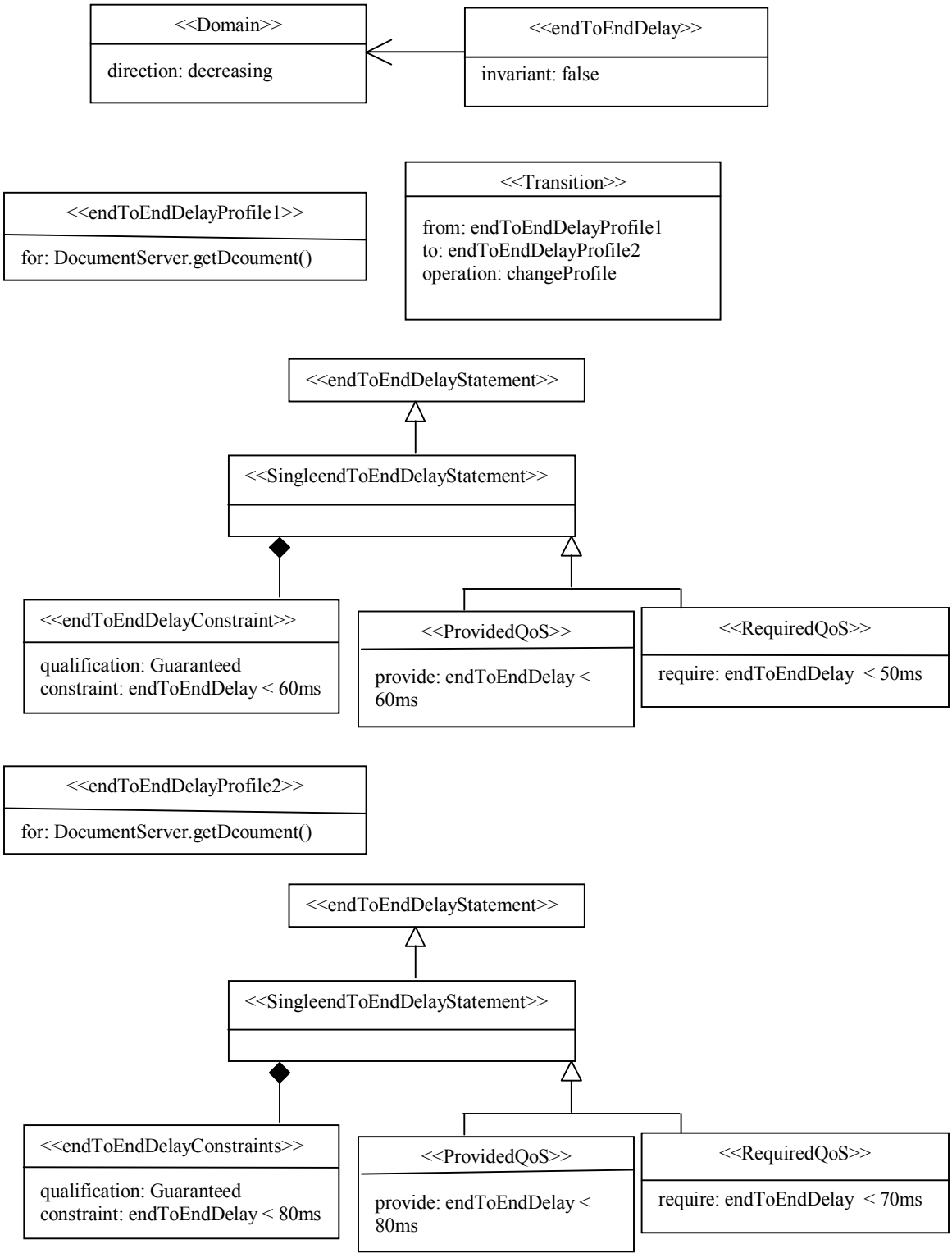

Figure 5.4. endToEndDelay QoS Specification

The collaboration diagram for the DocumentServer interacting with the DocumentTerminal that is annotated with the QoS specification is shown in Figure 5.5. The diagram indicates collaborating components and the message being passed. The message indicates the execution of the getDocument () method in the DocumentServer and the *endToEndDelay specifies that endToEndDelay (Dynamic QoS parameter) QoS specification for the method has to be applied.

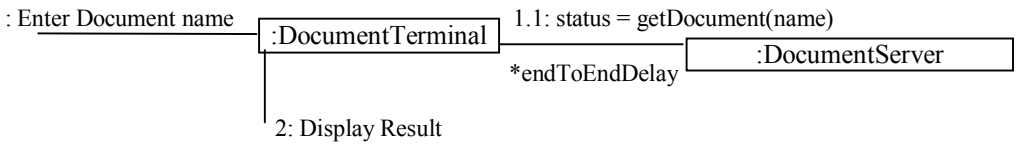

Figure 5.5. Collaboration Diagram Specifying the QoS Specification

Following the algorithm proposed in Chapter 4 (section 4.3) for relating the functional and the non-functional models, the message sent from DocumentTerminal to DocumentServer is found to be annotated with the endToEndDelay specification in collaboration. Now the endToEndDelay specification that was shown in Figure 5.4 is applied.

The algorithm for model transformation to code, which was indicated in Chapter 4 (section 4.4.1) is applied on the model to generate code. The class endToEndDelayProfile1 and singleendToEndStatement1 are the resultant QoS classes after transformation which will measure and ensure the end to end delay of 60ms. The endToEndDelayProfile1 gets the reference of the object which has the method for which the QoS profile is applied. The singleendToEndDelayStatement file provides the necessary mechanism (like using timer) to measure the end to end delay for the method. The QoS specification mechanisms aided in generating some of the necessary code to measure and provide the QoS of the component. This code will be part of the component which is being developed, thereby making the component to be quality aware. The class endToEndDelayProfile2 and singleendToEndDelayStatement2 ensure end to end delay of

80ms for the method getDocument. The sample code of the classes endToendDelayProfile1 and singleendToEndDelayStatement is shown below.

File: endToEndDelayProfile1

```
import java.util.*;
import java.rmi.*;
import java.lang.reflect.*;
import java.io.*;

public class endToEndDelayProfile1
{
   private String for;
   private String from;
   private String to;
   privtate String operation;

   /**
    * This method is to provide dynamic component QoS testing.
    */
   public ComponentQoS component_dynamic_test(String name) throws
RemoteException
   {

      String componentName = name;
      System.out.println(" The component name is " + componentName );
      ComponentQoS componentQoS = new
ComponentQoS("Document",componentName); // to store the values of the QoS

      try
      {
           IDocumentManagement
documentServer(IDocumentManagement)Naming.lookup("//magellan.cs.iupui.edu:9876/
DocumentServer");
           endToEndDelayProfile endToendDelayProfile2  = new
endToEndDelayProfile()
           singleendToEndDelayStatement endToEndDelayStatement = new
singleendToEndDelayStatement();
           /* A reference is created to the DocumentServer, another profile and the
statement class */

      }
      catch(Exception e)
      {
```

```
            System.out.println(e);
        }

          endToEndDelayStatement.statementon(documentServer);

    }

    public profileChange(documentServer)
                                        {

          documentserver.operation() //inform about the change in profile
                                        }


      public static void main(String args[]){
          try {


        endToEndDelayProfile EndToEndDelay = new endToEndDelayProfile();

        String docname = args[0];

EndToEndDelay.component_dynamic_test(DocumentServer);
        }
                                        catch ( Exception e){
        System.out.println( e);
        }
      }
}
```

File: singleendToEndDelayStatement1

```
import java.util.*;
import java.rmi.*;
import java.lang.reflect.*;
import java.io.*;

public class singleendToEndDelayStatement
{
   private String Qosparameter;
   private String Constraint;
   private String qualification;
   private String LogicalOperator;//Attributes from the UML diagram
   Private String provide;
   Private String Require;

   private Hashtable startingTimeTable; //keys: function name; values: Time objects
   private Hashtable stoppingTimeTable; //keys: function name; values: Time objects
   private long accumulatedDelay = 0;
   private int accumulatedCalls; //number of accumulated calls
   private double ETEDgetDocumentDS;

   public String docName ="";
   public String doc1 = "";
    /**
    * Constructor.
    */
   public SingleendToEndDealyStatement()
   {
      reset();

   }

   /**
    * Reset/initialize the private members.
    */
   public void reset()
   {
      startingTimeTable = new Hashtable();
      stoppingTimeTable = new Hashtable();
      accumulatedDelay = 0;
      accumulatedCalls = 0;

   }
   /**
```

```
 * This method records the starting time.
 */
public void startTimer(String functionName)
{
   if(functionName != null && !functionName.trim().equals(""))
   {

      Time startingTime = new Time();
      startingTime.getTime();
      startingTimeTable.put(functionName, startingTime);
   }
}

/**
 * This method records the stopping time.
 */
public void stopTimer(String functionName)
{
   if(functionName != null && !functionName.trim().equals(""))
   {
      Time stoppingTime = new Time();
      stoppingTime.getTime();
      stoppingTimeTable.put(functionName, stoppingTime);
   }
}

/**
 * This method returns end to end delay in usecond.
 */
public long getEndToEndDelay(String functionName)
{
   if(functionName != null && !functionName.trim().equals(""))
   {
      Time startingTime = (Time)startingTimeTable.get(functionName);
      Time stoppingTime = (Time)stoppingTimeTable.get(functionName);

      if(startingTime == null || stoppingTime == null)
      {
         return -1;
      }
      else
      {
         long second = stoppingTime.getSecond() - startingTime.getSecond();
         long uSecond = stoppingTime.getUsecond() - startingTime.getUsecond();
```

```
            startingTimeTable.remove(functionName);
            stoppingTimeTable.remove(functionName);

            return second * 1000000 + uSecond;
        }
    }
    else
        return -1;
}

/**
 * This method accumulates delay.
 * */
public void accumulateCallDelay(long delay)
{
    accumulatedDelay += delay;
    accumulatedCalls++;
}

/**
 * This method gets end to end delay (usecond).
 */
public double getEndToEndDelay()
{
    //sec/call
    double endToEndDelay = -1;

    if(accumulatedCalls != 0)
    {
        endToEndDel = (accumulatedDelay + 0.0)/accumulatedCalls;
    }

    return endToEndDelay;
}

      public void giveTime(String field){

          Time t1 = (Time)startingTimeTable.get(field);
          Time t2 = (Time)stoppingTimeTable.get(field);
          System.out.println("Starting time is " + t1.getUsecond());
          System.out.println("Ending time is " + t1.getUsecond());
          System.out.println("Elapsed time is " + (t2.getUsecond() - t1.getUsecond()));
  }

public void statementon(DocumentServer)
```

```
                                    {
        DocumentServer documentServer;
                                    }

   /**
    * This method is to provide dynamic component QoS testing for the method
getDocument
    */
   public QoSstatements(String name) throws RemoteException
   {


                                        System.out.println("Calling getDocument ");
        docName = doc1;
        for(int i = 0; i < 50; i++)
        {
          docName = docName + (i+700);
          docendToendDelay.startTimer("getDocument");
                                          documentServer.getDocument(docName);
          docendToEndDelay.stopTimer("getDocument");

docendToEndDelay.accumulateCallDelay(docendToEndDelay.getEndToEndDelay("get
Document"));
        }
                                      ETED = docendToEndDelay.getEndToEndDelay();
                                            ETEDgetDocumentDS = ETED;
              System.out.println("The end to end delay is " + ETED);


   }
}
```

The determination of Static QoS parameter involves external tests on the component. However, the component developer may incorporate the concepts used for testing to provide an interface as well as the quality code. One such model [BUR03], which was depicted in Figure 4.22, can be used to incorporate the access control into a component. The model also presents concepts that can be used for creating test cases to test the access-control QoS of the component. Figure 5.6 shows the collaboration diagram of the UserValidationServer and DocumentTerminal. It indicates that the access-control specification should be used for placing the guard's policy for the user specified

in the name parameter. The specification shown in Figure 5.7 specifies that the access control model does not involve any manual concept. The stereotype and tagged values which were identified from the access-control model (Chapter 4, Table 4.11) can be used to automate the process of inserting guards and policies for the component Since the approach used for static QoS, varies from model to model, a general transformation on to code cannot be incorporated. It is left to the developer to provide transformation rules for the approach being used for static QoS.

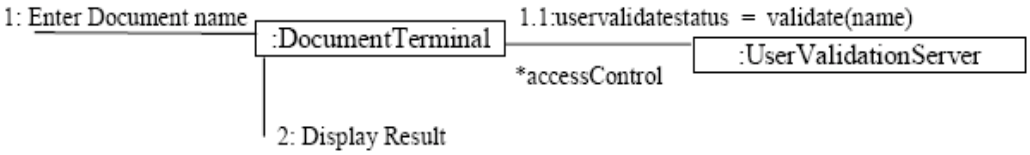

Figure 5.6. Collaboration Diagram for DocumentTerminal and UserValidationServer

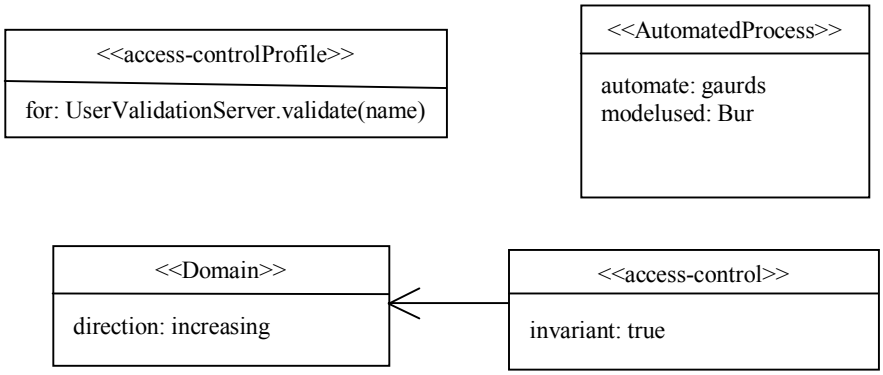

Figure 5.7. Access-Control Specification

In this chapter a case study was presented where the QoS specification mechanism was applied to a component being developed to make it quality aware. The QoS specification mechanism is general and can be applied to develop quality aware complex systems composed of many components.

6. CONCLUSION

This thesis presented an approach for specifying the QoS during the design and development phases of the component. This chapter concludes the thesis by presenting the features of the QoS specification mechanism, drawbacks of the mechanism and possible future enhancements.

6.1. Features of QoS Specification

The features of the specification mechanism are:

- It provides the QoS concept which enhances the UniFrame knowledgebase's ability for specifying QoS parameters. The concepts are very generic and can be applied to any QoS parameter.

- It provides necessary constructs to express the QoS concepts such that it can be used to specify QoS requirements during design and development of the software component.

- It provides a mechanism to relate the QoS specification to the specification of functional requirements and to reuse the QoS specification.

- It provides simple transformation rules for dynamic QoS parameter that will enable the transformation of QoS specifications onto the code that will enforce the QoS for the component.

- It provides a specification mechanism for static QoS parameters that will enable the developer to specify the model used for static QoS.

6.2. Issues Not Addressed

The following issues could not be addressed:

- The specification of the static QoS parameter for the component to provide the necessary support for determining the static QoS depends on the model used. The models do not provide a common approach and hence it could not be addressed. However the model element for the static QoS emphasizes the need for taking static QoS parameters into consideration during the design of the component

.

- Transformation rules for static QoS parameter.

- The QoS statements that perform the necessary dynamic QoS functionality depend on the QoS parameter. The QoS specification mechanism does not specify the actual statements that get inserted into the component code. The QoS statement model element specifies the constraint and how the constraint is enforced by the QoS statements varies for different parameters.

6.3. Future Work

Some of the possible future enhancements are:

- Formal transformation rules for mapping the QoS specification onto the component code. The transformation rules should be represented in formal

fashion to enable the code generator to apply rules for generating necessary QoS code.

- Enhancing the QoS specification for taking into account the composition and decomposition of QoS of the system composed of components. Currently, the decomposition and composition rules of [SUN02] are used to find out the QoS for components and then the QoS specification mechanism is used to specify QoS for each component.

- A tool that can incorporate the QoS specification onto the component code.

- Representing the heterogeneity issues related to QoS in glues and wrappers. The glues and wrapper mediate between heterogeneous components and involve mechanisms to handle heterogeneity.

6.4. Summation

This thesis presented an approach for QoS specification which will enable the component developer to specify the QoS during design of the component. It also presented an approach which will indicate the QoS requirement for the functional model. Simple transformation rules were also presented that will enable generation of QoS code for the component. A case study was presented to validate the approach. The QoS specification mechanism will enable UniFrame to generate QoS aware glues.

REFERENCES

REFERENCES

[AAG01] Aagedal J. Ø., "Quality of Service Support in Development of Distributed Systems", PhD thesis, Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo, 2001.

[AUG95] Auguston M., "Program Behavior Model based on Event Grammar and its application for debugging automation", Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging, 1995.

[AUG97] Auguston M., Gates A., Lujan M., "Defining a program behavior model for dynamic analyzers", Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering, pages 257-262, 1997.

[BBN01] BBN Corporation, Quality Objects Project Url: http://www.dist-systems.bbn.com/ tech/QuO, 2001.

[BRA01] Brahnmath G., Raje R., Olson A., Sun C., "Quality of Service Catalog for Software Components", Technical Report (TR-CIS-0219-01), Department of Computer and Information Science, Indiana University Purdue University Indianapolis, 2001.

[BRA02] Brahnmath G., Raje R., Olson A., Bryant B., Auguston M., Burt C., "A Quality of Service Catalog for Software Components", The Proceedings of the Southeastern Software Engineering Conference, Huntsville, Alabama, pages 513-520, April 2002.

[BER01] Berzins V., Shing M., Auguston M., Bryant B., Kin B., "DCAPS-Architecture for Distributed Computer Aided Prototyping System", Proceedings of RSP 2001, the 12th international workshop on rapid system prototyping, 2001.

[BUR02] Burt C., Raje R., Olson A., Bryant B., Auguston M., "Quality of Service Issues Related to Transforming Platform Independent Models to Platform Specific Models," Proceedings of the 6th IEEE International Enterprise Distributed Object Computing Conference, Lausanne, Switzerland, September 2002.

[BUR03] Burt C., Raje R., Olson A., Bryant B., Auguston M., "Model Driven Security: Unification of Authorization Models for Fine-Grain Access Control", Proceedings of EDOC 2003, The 7th IEEE International Enterprise Distributed Object Computing Conference, Brisbane, Australia, September 2003.

[CAZ99] Cazzola W. et al., "Rule-based Strategic Reflection: Observing and Modifying Behavior at the Architectural Level", Proceedings of 14th IEEE International Conference, Automated Software Engineering (ASE 99), IEEE Press, Piscataway, New Jersey, October 1999.

[CID02] OMG CORBA IDL Specification, Url: http://www.omg.org/gettingstarted/omg_idl.htm, 2002.

[CSA95] Frappier M., Matwin S., Mili A., "Software metrics for predicting maintainability", Canadian Space Agency, Software metrics study – Technical Memorandum 2, 1994.

[FRO98] Frolund S., Koistinen J., "Quality of Service specification in distributed object systems", Distributed System Engineering Journal, Vol. 5, Issue 4, December 1998.

[GRE92] Green T., Petre M., "When visual programs are harder to textual programs", Human-computer interaction: tasks and organization, ECCE-6, CUD:Rome, 1992.

[GME02] The "Generic Modeling Environment", Url: http://www.isis.vanderbilt.edu/projects/gme/, 2002.

[ISO, 1986], Quality Vocabulary, ISO, Report: ISO 8402, page 8.

[ISO99] ISO/IEC JTC1/SC7, "Information Technology - Software product quality: Quality model," ISO/IEC, 9126, 1999.

[LOY98] Loyall J., Bakken D., Schantz R., Zinky J., "QoS Aspect Languages and Their Runtime Integration", Lecture Notes in Computer Science, Vol. 1511, Springer-Verlag, Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98), Pittsburgh, Pennsylvania, pages 28-30, May 1998.

[MDA01] The "Model driven architecture", Url: http://www.omg.org/mda/, 2001.

[NAY02] Nayani N., Raje R., Olson A., Bryant B., Burt C., Auguston, M, "An Architecture for the UniFrame Resource Discovery Service," Proceedings of SEM 2002, 3rd International Workshop on Software Engineering and Middleware, May 20-21, 2002, Orlando, Florida Springer-Verlag Lecture Notes in Computer Science, Vol. 2596, pages 20-35, 2003.

[NOS90] Nosek J., Roth I., "Comparison of Formal Knowledge Representation schemes as communication tool: predicate logic vs semantic network", In: International Journal of Man Machine studies, Vol. 33, pages 227-239, 1990.

[OCL01] The Object Constaraint Language, http://www.omg.org/cgi-bin/ocl, 2001.

[OMG02] Object Management Group. 2002. "UML™ Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms". Request for Proposal, OMG document ad/02-01-07, Framington, MA, 2002.

[RAJ01] Raje R., Auguston M., Bryant B., Olson A., Burt C., "A Quality of Service – based framework for creating Distributed Heterogeneous Software Components", Submitted to Informatica, 2001.

[RAJ00] Raje R., "UMM: Unified Meta-object Model for Open Distributed Systems", Proceedings of the fourth IEEE International Conference on Algorithms and Architecture for Parallel Processing (ICA3PP'2000).

[RAJM01] Raje R., Auguston M., Bryant B., Olson A., Burt C., "A Unified Approach forthe Integration of Distributed Heterogeneous Software Components", Proceedings of the 2001 Monterey Workshop, Monterey, California, 2001.

[SUN02] Sun C., Raje R., Olson A., Bryant B., Auguston M., Burt C., Huang Z., "Composition and Decomposition of Quality of Service Parameters in Distributed Component-Based Systems", Proceedings of the IEEE Fifth International Conference on Algorithms and Architectures for Parallel Processing, Beijing, China, October 2002.

[SZY99] Szyperski C., "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, ISBN 0-201-17888-5, 1999.

[TUT04] The UML tutorial, http:// odl-skopje.etf.ukim.edu.mk/uml-help, 2004.

[TUM04] Tummala K., "Glue generation framework in UniFrame for the CORBA-JAVA/RMI interoperability", Technical Report (TR-CIS-0302-03), Department of Computer and Information Science, Indiana University Purdue University Indianapolis, 2004.

[UML01] The Unified Modeling Language, http://www.omg.org/technology/documents/formal/uml.htm, 2001.

[VOA98] Voas J., "An Approach to Certifying Off-the-Shelf Software Components", IEEE Computer, June 1998.

[VOA95] Voas J., "Software Testability Measurement for Assertion Injection and Fault Localization", Proceedings of 2nd Int'l. Workshop on Automated and Algorithmic Debugging (AADEBUG'95), St. Malo, France, May 1995.

[VOA96] Voas J., Ghosh A., McGraw G., Charron F., Miller K, Defining an adaptive software security metric from a dynamic software failure tolerance measure, Proceedings of the 1lth Annual Conference on Computer Assurance, pages 250-263, June 1996.

[VOA98] Voas J., “An Approach to Certifying Off-the-Shelf Software Components”, IEEE Computer, June 1998.

[VOA00] Voas J., Payne J., “Dependability Certification of Software Components”, Journal of Components and Software, 2000.

[WAR03] Warmer J., Kleppe A., “The Object Constraint Language”, Addison-Wesly, ISBN 0321179366, 2003.