QOS COMPOSITON AND DECOMPOSITON MODEL

IN UNIFRAME

A Thesis

Submitted to the Faculty

of

Purdue University

by

Changlin Sun

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

August 2003

To Mom and Dad

ACKNOWLEDGMENTS

I would like to thank all those people who made this thesis possible and an enjoyable experience for me.

First of all, I wish to express my sincere gratitude to my advisor, Dr. Rajeev Raje, without whose continuous support and patient guidance this thesis would not be possible.

I would like to thank Dr. Andrew Olson for providing invaluable suggestion and help whenever I was in need throughout this study.

I would like to thank Dr. Jiangyu Zheng for his valuable time and effort serving as member of my advisory committee.

I would like to thank my teammate in the UniFrame project. The discussions and cooperation with them were very important to this work.

I would like to thank the staff of CS department for their cooperation and assistance during this work.

I gratefully acknowledge the U.S. Department of Defense and the U.S. Office of Naval Research for supporting this research with their grant under the award number N00014-01-1-0746

Finally, I would like to express my deepest gratitude to my family for their endless love and support during my study at IUPUI.

TABLE OF CONTENTS

Page

Page

Page

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

ABSTRACT

Changlin Sun. M.S., Purdue University, August 2003. The QoS Composition and Decomposition Model in UniFrame. Major Professor: Rajeev Raje.

Software systems are increasingly large, complex, heterogeneous, distributed and pervasive. The component-based software development provides a promising methodology for developing large-scale, complex software systems. Component-based software development advocates developing software systems by selecting commercial-off-the-shelf or in-house software components and assembling them within appropriate software architectures. This software development methodology promises software reuse and thus increases development productivity. A major challenge of component-based software development is the prediction of the system quality attributes in the absence of implementation details of individual software components. In top-down development, it is critical to factor the system level quality attributes into individual components, and thus facilitate the selection of qualified components. In bottom-up development, it is important to predict the system wide quality attributes based on the quality attributes of individual components and the way they compose. This thesis proposes an approach for decomposing and composing quality of service parameters during development of component-based software systems. The inter-component communication patterns are identified and their effects on the composition and decomposition of QoS parameters are studied. The effect of the network and the execution environment on the composition and decomposition of QoS parameters are preliminarily investigated.

1. INTRODUCTION

1.1 Component-Based Distributed Systems

A shift towards distributed computing systems has occurred in recent years due to the cheap computing power and a better networking infrastructure. The proliferation of the World Wide Web and Internet-based technologies and services will make the future computing happen anywhere, anytime, for any data and on any device. A distributed system is a collection of autonomous computers linked by a computer network and supported by software that enables the collection to operate as an integrated facility. Well-established techniques such as the inter-process communication and the remote invocation, the naming services, the cryptographic security, the distributed file systems, the data replication, and the distributed transaction mechanisms provide the run-time infrastructure supporting today's distributed systems.

In developing large scale, complex distributed systems, especially critical systems, the traditional software development can result in a high development cost, a low productivity, an unmanageable software quality and create less reliable software systems. Component-based software development is one of the most promising approaches available today to overcome these drawbacks. This approach is based on the idea that software systems can be developed by selecting appropriate off-the-shelf components and then assembling them with well-defined software architecture. Thus component-based software development is different from the traditional software development approach, in which software systems can only be implemented from scratch. In the component-based software development approach, the commercial off-the-shelf components can be developed by different suppliers using different languages and different platforms. System developers browse catalogs of software components from multiple vendors and assemble complex systems from available building blocks. Many

benefits result from this scheme, including a decreased system development time and cost, a low component cost due to the amortization of component development costs over multiple users, availability of robust components due to greater maintenance resources supported by multiple users of each component, and a wide range of general-purpose and domain specific components.

Component-based development offers a vision of plug and play software development. It will likely shift the focus of software engineering from the “specify, design, and implement” concept toward “select, evaluate, and integrate” [1]. The traditional waterfall model of software life cycle is shown in Figure 1.1 [2]. The component-based life cycle is shown in Figure 1.2 [3].

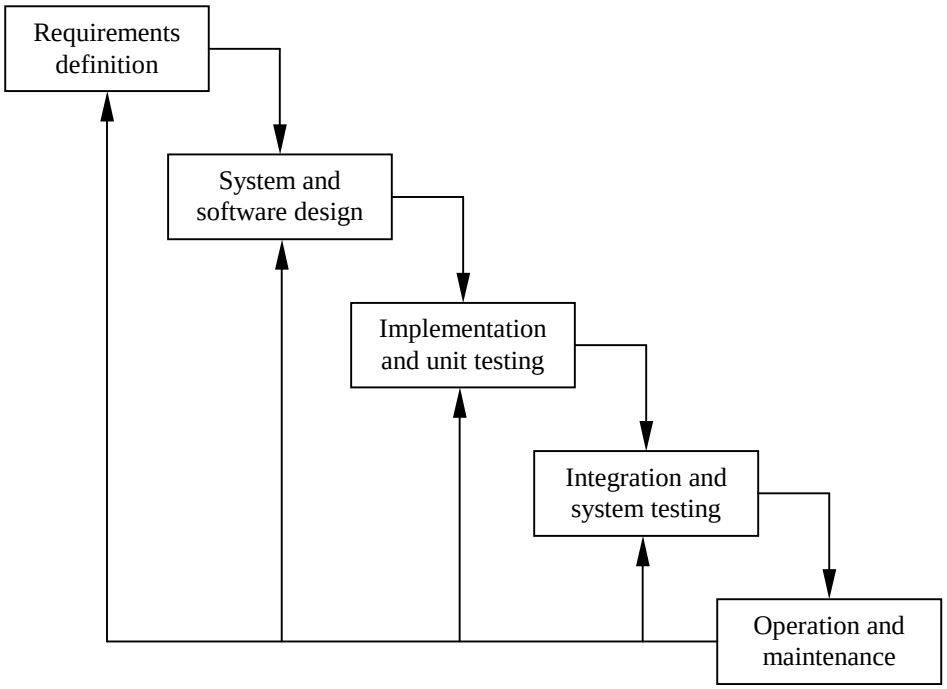

Figure 1.1 Traditional Software Life Cycle: Waterfall Model

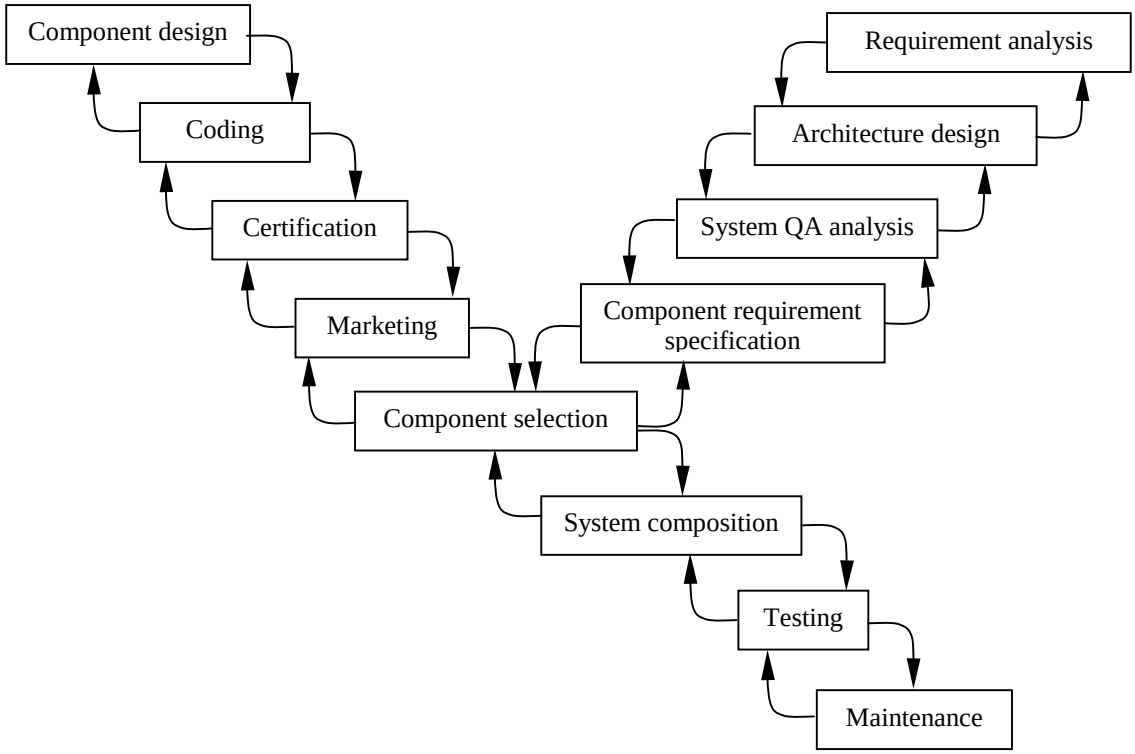

Figure1.2 Component-Based Software Life Cycle

1.1.1 Software Component & Component Model

A software component is a software element that conforms to a component model and can be independently deployed and is subject to composition by third parties without modification according to a composition standard [4]. In another words, a software component is a self-contained unit encapsulating data and logic, a composable unit with well-defined external interfaces, a configurable unit reused in various contexts and a packaged and deployable unit. A component can be as small as a single procedure or as large as an entire application.

The use of the notion of a software component has the following advantages:

1) Reduced time to market: use of software component can improve application productivity, reduce complexity, and increase reuse of existing code.

2) Programming by assembly (manufacturing) rather than development (engineering):

This can reduce skills requirements, and focus expertise on domain problems and on improving software quality.

There are many challenges emerging from the usage of components. These are:

1) Heterogeneity: the heterogeneity widely exists in networks, operating systems, runtime libraries, and programming languages.

2) Interoperability: components developed with different models (Java, CORBA, .NET) cannot easily interoperate. Presently, there are no commonly agreed vendor-neutral models for creating & integrating components.

3) Configurability: providing a flexible and consistent connection between provided interface with required interface is non trivial in component-based development.

4) Composability: how to reason about the system properties from component properties is still an unsolved problem.

5) Packaging, deployment, and configuration: this issue is associated with the distribution and selling of components and installation and configuration of component-based systems.

Typically, a component is based on a certain component model. Current and emerging component models include Java, CORBA, .NET and Web Services. A Java [5] component is pure Java. It utilizes all the features of Java in a native way. It uses Java language to describe component services. The communication between Java components is via Java RMI (Remote Method Invocation). Component discovery is achieved through the JNDI (Java Naming & Directory Interface). The advantage of a Java component is that it is platform independent. A Java component can run on any platform as long as there exists a Java virtual machine on that platform. A Java component is Java specific. So it is difficult to integrate a Java component with a component written in another language.

CORBA [6] is a specification for building and deploying distributed components. Different implementations of CORBA exist. In CORBA, component interfaces are described using a language called IDL (Interface Definition Language). This allows the interface definitions in a language independent manner. A mapping exists for compiling IDL to different languages such as C, C++, and Java. So, the CORBA component is

language independent. Components written in different languages can communicate with each other within the CORBA model. The communication between two CORBA components is through IIOP (Internet Inter-ORB Protocol). CORBA uses naming services or trading services to locate objects and services. CORBA is platform independent because it presumes a heterogeneous environment. Using IIOP, different CORBA implementations interoperate with each other to produce a worldwide object bus.

.Net [7] defines a full platform. It defines a virtual machine (common runtime environment), which can execute intermediate code called byte code. The byte code is called intermediate language (IL) that is independent of any platform. .Net components can be written in any language, such as C++, C#, VB.NET and others, that have tools to produce the IL byte code. In this sense, a .Net component is also language independent (at byte code level) and platform independent. .Net does not provide direct support for distributed objects but does support web services that provide support for distributed objects.

Web Services [8] is a recent component model that builds on the XML technology. Web Services are complementary to Java, CORBA, and .Net technologies. They do not replace them. In Web Services, any service can be described by WSDL (Web Service Description Language). The communication between components is via SOAP (Simple Object Access Protocol). In Web Service, the component services are located using UDDI (Universal Description, Discovery, and Integration). Thus, Web Services are independent of the programming languages used. Any model can support or will support Web Service to facilitate an integration. Thus, Web services can provide a simple way to integrate any model such as Java, CORBA, and .Net. Web Services can be used to integrate heterogeneous components in two ways: wrapping existing component as Web Services or use Web Services as “glue” to integrate multiple components.

1.1.2 Software Architecture

The software architecture is the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time [9]. The software architecture is another promising approach proposed in response to the growing complexity of software systems and the problems they attempt to solve. The purpose of software architecture is to reduce costs of developing applications and to increase the potential for commonality between different members of a closely related product family. Software architecture provides explicit, high-level system models and a support for capturing recurring properties of an application domain. Architectures enable developers to focus on the “big picture” while developing a system and to adopt a component-based development philosophy as opposed to always building a system from scratch. Architectures do this by making a software system’s structure explicit, separating the computations of components from their interactions in a system, and providing a high-level model of a system that can be manipulated and analyzed before any changes are effected in an actual implementation.

Software architecture is commonly described in terms of the following basic elements: components, connectors, configurations, and constraints. In software architectures, components are normally independently developed and deployed. Connectors are wires between components. Configurations describe a set of rules about how components are connected with each other. Constraints describe the functional or non-functional requirements imposed on components and connectors.

Existing component models are component-centric: they are primarily concerned with standardizing external component properties—interfaces, packaging, binding mechanism, inter-component communication protocols, and expectations regarding the runtime environment. They do not support reasoning about system quality attributes, e.g., performance, modifiability, reliability and safety. Instead, an engineer must wait until the components have been acquired, integrated, and the system as a whole benchmarked, to determine whether a system meet its quality attribute goals. Software architectures, in contrast, are system-centric: they focus on specifying systems of communicating black-box components, analyzing resulting system properties, and generating “glue” code that

binds system components. Component models and software architectures both are crucial aspects of component-based software development. There are still gaps between these two domains. Component models alone do not adequately address certain system-wide aspects of engineering large, complex, distributed software systems. The software architecture research, on the other hand, typically has not focused on component development, packaging, and interoperability. These different but complementary domains indicate an opportunity for an effective marriage of the two areas, where one can couple the benefits of explicit architectural models with those of component interoperability models. Such a unified approach would form a solid basis on which a successful software component marketplace can build.

The architectural issues are important in component-based development. For example, during component selection, it has to be ensured that the components fit into already known architectural properties of the application under development. This compatibility with the selected architecture is especially important because the system architecture has extreme impact on the non-functional properties of the system. By conforming to the selected system architecture, the foundation is laid for the composed system to adhere to the non-functional requirements to the application.

1.1.3 Networking Infrastructure

The systems in the future will be distributed, consists of many components, and services, and will be highly dependent on networking and information infrastructure [10]. The exponential growth of the Web and other Internet-based systems and services make it a challenge for distributed system technology to provide flexible and reliable infrastructures for large-scale systems that meet the demands of developers, users and service providers.

Today’s new communication networks are converging into a unique network infrastructure carrying data in the form of voice, video and mission critical applications. Mobile communication is moving towards multimedia contents. The outcome will be a

new network environment where data, voice and video, fixed and mobile applications will share the same infrastructure.

With today's distributed systems from application domains, such as data mining, e-commerce, and multimedia, which are bandwidth hungry, time sensitive, and mission critical, new and more demand requirements are continually imposed on the underlying network infrastructure. The traditional network handles network traffic in a best-effort way. If network resources are available, then the best effort traffic is delivered, otherwise the traffic is dropped. Moreover, it can only assure the data delivery without any guarantee for a minimum data rate or a timed delivery. The new voice, video and interactive multimedia applications will require the network infrastructure to provide QoS mechanisms, such as, avoiding network congestion, bandwidth reservation, and traffic shaping and policing.

In addition to the traditional best effort service, a network can provide differentiated and guaranteed services. A differentiated service is an intermediate quality of service level. Different types of traffic can share the same data path across the network. For example, time-critical applications can compete with less time-depended applications for the network resources. In the differentiated service framework, some traffic has higher privileges than the others. Packets entering the network are classified according to the application to which they belong to or to the service level agreements with the customer. Then the most important traffic is prioritized following such classification. Prioritizing some traffic means to give precedence to this traffic in the use of network resources.

Prioritizing network traffic is not an optimal solution for the quality of service requirements. It is usually a statistical treatment, not a strict guarantee. During a hard congestion phase, all traffic, even the most important, might undergo the network congestion state and reduce the quality of applications. In order to obtain a strict guarantee of quality of service, a guaranteed service is needed. It is the highest level of quality of service architecture where there is an absolute end-to-end reservation of network resources for a specific traffic type. In the guaranteed service framework, incoming traffic is classified and divided into different traffic flows with different priority

levels to meet the corresponding network application requirements. Some of these flows can receive guaranteed resources like a definite percentage of the available bandwidth on the transmitting channel. The guaranteed service makes it possible to provide a service that guarantees both delay and bandwidth of networks.

1.2 Functional and Non-Functional Requirements

Software requirements are descritpions of the services which the system should provide and the constriants under which the system must operate. Software requirements are partitioned into functional requirements and non-functional requirements. Functional requirements capture the intended behavior of the system. This behavior may be expressed as services, tasks or functions the system is required to perform. Non-functional requirements are constraints on various attributes of these service, tasks or functions. Functional requirements can be thought as verbs while non-functional requirements can be thought as adjectives or adverbs.

1.2.1 Functional Requirements

One of the first steps in developing component-based distributed systems is to identify the system's functional requirements (i.e., what is intended to be done). A clear system functional requirement will facilitate the system design.

For example, in a bank system, Automatic Teller Machines (ATMs) are used by banks to let customers withdraw money from their accounts without interaction with bank personnel and at any time of day. Since they operate on bank accounts, there is a need for a high security mechanism. The system to be built is a security device to be inserted between the ATM and the bank. It operates on the banks database, using an encrypted communication line, to serve requests from the ATM. For this bank system, the following functional requirements can be identified:

1) The DES encryption protocol shall be used for communication with the bank and the ATM with a key stored internally in the device.

2) The encryption key shall be set using a keypad mounted on the device.

3) There shall be a service check_pin (customerid, code) which checks that the customer with the id given by customerid has the PIN-code code.

4) There shall be a service get_balance (account) which returns the balance of the account with number account.

5) There shall be a service withdraw (account, amount) which withdraws amount from the account with number account.

The system functional requirements can be defined by use cases [11]. A use case defines a goal-oriented set of interactions between external actors and a system under consideration. A use case is initiated by a user with a particular goal in mind, and followed by a sequence of interactions between actors and the system that are necessary to deliver the service that satisfies the goal. A complete set of use cases specifies all the different ways to use the system, and therefore defines all behavior required (functional requirements) of the system. Table 1.1 shows a typical template of use cases.

Table 1.1 A Typical Use Case Template

Use Case ID:			
Use Case Name:			
Created By:		Last Updated By:	
Date Created:		Date Last Updated:	
Actors:	Outside entities		
Description:	Provide a brief description of the reason for and outcome of this use case.		
Preconditions:	List any activities that must take place, or any conditions that must be true, before the use case can be started.		
Postconditions:	Describe the state of the system at the conclusion of the use case execution.		
Normal Course:	Provide a detailed description of the user actions and system responses that will take place during execution of the use case under normal, expected conditions.		
Alternative Courses:	Document other, legitimate usage scenarios that can take place within this use case separately in this section.		
Exceptions:	Describe any anticipated error conditions that could occur during execution of the use case, and define how the system is to respond to those conditions.		
Includes:	List any other use cases that are included by this use case.		
Priority:	Indicate the relative priority of implementing the functionality required to allow this use case to be executed.		
Frequency of Use:	Estimate the number of times this use case will be performed by the actors per some appropriate unit of time.		
Business Rules:	List any business rules that influence this use case.		
Special Requirements:	Identify any additional requirements, such as nonfunctional requirements, for the use case that may need to be addressed during design or implementation.		
Assumptions:	List any assumptions that were made in the analysis that led to accepting this use case into the product description and writing the use case description.		
Notes and Issues:	List any additional comments about this use case or any remaining open issues or TBDs (To Be Determined) that must be resolved.		

1.2.2 Non-functional Requirements

Non-functional requirements (i.e., Quality of Service) define how well the system operates or how well the functionality is exhibited. Each non-functional requirement defines the quality of a system from one specific aspect. Some examples of non-functional requirements are: timeliness, performance, reliability, availability, safety, security, scalability, flexibility, usability, maintainability, and reusability.

Non-functional requirements are consequences of the design decisions taken to implement the system's functional requirements. They are rarely considered when system is built, especially in the early stages of the system development phase. The following are reasons for not considering these requirements explicitly: a) non-functional requirements

are usually very abstract and can be stated only informally, b) non-functional requirements are rarely supported by languages, methodologies and tools, c) non-functional requirements are more complex to deal with, and d) non-functional requirements are difficult to effectively satisfy during the system development [12].

It is not trivial to verify whether a specific non-functional requirement is satisfied by the final product or not. Very often non-functional requirements conflict and compete with each other, e.g., security and performance. Non-functional requirements commonly concern environment builders instead of the application programmers. The separation of functional and non-functional requirements cannot be easily defined.

There are no good approaches for tackling the non-functional requirements. These non-functional aspects should be treated as independent design dimensions, specified in an implementation-independent way and implemented in such a way that the resulting system behavior becomes predictable. Non-functional requirements should be stated in the requirements and taken into consideration from the beginning, i.e., during the architectural phase. Unfortunately, current software engineering practices often consider non-functional requirements only during the implementation phase. Obviously, the explicit treatment of the non-functional requirements from the beginning of the development is necessary for building a predictable system out of components.

1.3 System Composition and Decomposition

In developing component-based distributed systems, with components as the building blocks, it is important to decompose the system's functional and non-functional requirements into the individual components and predict/observe the system functional and non-functional requirements by composing the functional and non-functional requirements of the individual components.

1.3.1 Composition and Decomposition of Functional Requirements

The System decomposition is a divide-and-conquer approach to deal with complexity of the system and improve its reusability. A system can be decomposed into interacting subsystems. Each subsystem may have a similar internal decomposition. Components are the elements at the lowest level in a system hierarchy. The system hierarchy allows a system to be understood at different levels of granularity. A subsystem is more specialized and implements simpler functionality. The decomposition of system functionality may maximize the cohesion within subsystem and minimize the coupling between subsystems. Other factors may also affect the system decomposition. For example, assume that in a component-based distributed system, there are two components: the client component, and the server component. The decomposition of application functionality between the client component and the server component depends on whether the goal is to minimize the network bandwidth or to maximize the client computation. For instance, thin client-based systems move all computations to the server component and just send updates to the client component. As a result, clients do not have much computing power but networks need to have more bandwidth and less latency for the system to run in a satisfactory manner

In the composition of system functionality, subsystems are selected and integrated into a system under specific system architecture. In some scenarios, glues or wrappers are needed to make subsystems interoperate with each other. Some potential problems may arise in system composition. For example, an architectural mismatch, which occurs when subsystems fail to meet the architectural constraints; functional deficiencies, which arise when subsystems do not satisfy all the functional requirements; and combining quality attributes, which means how the system inherits properties that are associated with the individual components. In the system composition, emerging functionality may arise from the interactions of subsystems. In a sound composition of system functionality, subsystems are integrated together and cooperate properly to achieve the desired system functionality.

1.3.2 Composition and Decomposition of Non-functional Requirements

Future systems will need to satisfy not only the functional requirements but also the non-functional requirements. This is especially true for real-time systems, multimedia systems, e-commerce systems, mission critical systems, and safety critical systems. Therefore, while developing component-based distributed systems, the composition and decomposition of non-functional requirements also need a proper attention.

In the decomposition of non-functional requirements, a system-wide property is decomposed into the properties of its subsystems. By taking different design decisions, the system-wide property can be built into subsystems. The property decomposed into subsystems can be used to select appropriate subsystems during the system integration phase.

In the composition of non-functional requirements, system quality attributes are predicted based on the properties of the individual components. The assembly of the system from subsystems needs to be in a fashion that preserves the predictions. How can systems be composed in a way that certain system-wide quality requirements will be met is still an open research problem.

1.4 Objectives of This Thesis

The objectives of this thesis are to study the issues of incorporating non-functional requirements (Quality of Service) into the development of component-based software systems and propose an approach for decomposing system-level non-functional requirements into individual components and composing non-functional properties of individual components to predict the system-wide non-functional properties. This approach will assist the system developers in selecting appropriate components, predicting the properties of the integrated system and empirically validating the predicted values. This work is a significant part of a research effort called UniFrame. The proposed composition and decomposition approach is used in generating (semi-automatically) component-based distributed systems using UniFrame.

1.5 Contributions of This Thesis

The contributions of this thesis are:

1) Propose an approach to engineer non-functional requirements into the development of component-based software systems. The proposed approach provides the composition and decomposition rules for factoring system-wide non-functional requirements into individual components and composing non-functional properties of individual components to predict the system-wide non-functional properties.

2) Identify inter-component communication patterns and investigate their effect on the composition and decomposition of non-functional properties.

3) Preliminarily incorporate the network component into the proposed QoS composition and decomposition approach.

4) Identify the key environment factors and preliminarily analyze their effects on system composition and decomposition.

1.6 Organization of This Thesis

The thesis is organized as:

The second chapter introduces the background and the related work. The third chapter presents an overview of the UniFrame. The fourth chapter presents the system composition and decomposition rules. The fifth chapter investigates the inter-component communication patterns and their effects on system composition and decomposition. The sixth chapter investigates the effect of the network on the composition and decomposition of QoS parameters. The seventh chapter presents the effects of the environment on the composition and decomposition of non-functional properties. The eighth chapter makes several conclusions and indicates future directions.

2. BACKGROUND AND RELATED WORKS

2.1 Current Approaches in System Composition and Decomposition

System composition and decomposition with reasoning about system properties are important in component-based system design. This compositional design makes it possible that the component developers can develop and verify components independently and system designers can build a system and verify the system properties with given properties of components. Some of the ongoing research works related to system composition and decomposition with property reasoning are briefly discussed in the following sections.

2.1.1 Formal Approaches

System composition and decomposition is a broad concept and the problem can be defined and dealt with in different contexts, for example, formal methods, software engineering, programming languages. In the following subsections, the current formal approaches are discussed.

2.1.1.1 Assumption-Guarantee Approach

Abadi and Lamport [13] investigated the composition of an assumption-guarantee property. An assumption-guarantee property states that a component can guarantee a desired property in composition as long as certain assumptions hold true for its environment. In the assumption-guarantee composition reasoning approach, a component is differentiated from its environment. The assumptions about the environment can be constrained by the safety property (assert that something bad does not happen) or the

progress property (assert that something good eventually does happen). The safety property composed well, but the progress property does not compose well, even though the progress property is widely used in system specifications. It is not completely clear what kind of constraints can be put on the environment if a component needs to guarantee a certain property. The constraints on the environment cannot be too strong because it would make the component less reusable, nor be too weak, because it would not realize the guaranteed property in composition.

2.1.1.2 Composition Based on Existential and Universal Properties

Chandy, Sanders, and Charpentier [14-17] study the system composition by starting from those compositional properties. The compositional property is the property that allows the deduction of the system property from the component properties using simple rules. For example, mass is a compositional property because the mass of a system is the sum of the masses of its components. Two types of the compositional property were proposed: an existential property and a universal property. A property is of an existential type when it holds in any system in which at least one component has that property. A property is of a universal type when it holds in any system in which all components have that property. Consider the example of putting pieces together in a jigsaw puzzle. An example of a universal property is ”the component is entirely dark colored.” If the entirely dark-colored components are put together, an entirely dark-colored (larger) component will be got. An example of an existential property is: ”the component has a light-colored region.” A component has a light-colored region if it has a subcomponent with a light-colored region.

They proposed a theory of composition based on existential and universal properties. However, some properties are neither universal nor existential. Therefore, the problem is how to compute the property of composed systems given properties of components, whether the properties are compositional or not. The composition of non-compositional properties can be dealt with by first specifying components as conjunctions of universal and existential properties so that universal and existential

system properties can be readily derived from component properties. Then the non-compositional system properties can be derived from these universal and existential system properties.

2.1.2 Software Architecture-Based Approach

It is believed that predicting and ensuring system-level quality attributes and controlling component feature interactions are closely related [18]. In an architecture-based approach, a set of components and connectors along with their topology and pattern of interaction is used to characterize the necessary context in which the component will be deployed.

2.1.2.1 Scenario-Based Architectural Analysis Method

In [19], a scenario-based architectural analysis method (SAAM) is presented, using scenarios to analyze architectures to determine their fitness with respect to certain qualities of the resulting systems, for example, performance, reliability, security, maintainability, portability, and etc. The method follows the following steps: describe candidate architecture, develop scenarios, perform scenario evaluations, reveal scenario interactions, and the overall evaluation.

2.1.2.2 Attribute-Based Architecture Styles

In [20], the architectural style for composition is used to characterize the necessary context in which the component will be deployed. An attribute-reasoning framework is associated with an architectural style. These reasoning frameworks are based on quality attribute-specific models, which exist in the various quality attribute communities. For example, if the goal is to reason about reliability, then the salient features of the architecture (such as redundancy) need to be mapped onto reliability models (such as Markov models). This approach makes it possible that software

architecture can be designed and analyzed based on reusing known patterns of software components with predictable properties.

2.1.2.3 Architecture Tradeoff Analysis Method

In [21], the architecture tradeoff analysis method (ATAM) is proposed to consider interactions among quality attributes and evaluate software architecture's fitness with respect to multiple competing quality attributes. The steps of the method are: collect scenarios, collect requirements/constraints/environment, describe architectural views, realize scenarios, perform attribute specific analysis, identify sensitivities, and identify tradeoffs.

2.2 Analysis of Non-Functional Requirements at the Early Design Phase

In traditional software development, the non-functional requirements are rarely considered when software is built, especially in the early stages of the software development process. The failure to meet the non-functional requirements is mostly due to a lack of consideration of non-functional requirement issues early in the development process. The non-functional requirement issues are ignored until system testing or later. The non-functional requirement issues discovered until late in the development process are more difficult and more expensive to fix. Therefore, incorporating non-functional requirements into the early design phase of software development process is important to build quality software with reduced time-to-market. Four approaches dealing with non-functional requirements at the early design phase are discussed in the following sections.

2.2.1 Parmenides Framework

In [22], an architecture-based framework is proposed for dealing with non-functional requirements during both development and execution time of dynamic distributed systems. The framework consists of a process-oriented language for

describing non-functional requirements at the begin of the development, a guideline on how to incorporate non-functional requirements into the software architecture, a set of rules for refinement of non-functional architectures, a strategy for mapping non-functional architectures into actual implementation elements, a product-oriented language suitable for describing non-functional requirements at the final product, and a set of change operations that incorporates conditions for preserving the integrity of non-functional properties.

In this framework, non-functional requirements are fundamentally viewed as constraints on possible design decisions for implementing the functional parts of software. Hence, every decision taken for implementing a functionality of the software must respect the constraints imposed by the non-functional part of the requirements. The non-functional requirement is modeled by three abstractions: NF-Attribute, NF-Realization and NF-Requirement. A NF-Attribute models both any non-functional characteristic of the software that can be precisely pointed out and any non-functional feature that cannot be quantified, but may be defined as present in the software in a certain level (security). NF-Requirements are constraints over the NF-Attributes. NF-Realizations act as design constraints; if they are adopted the NF-Requirement can be achieved.

The non-functional requirements are specified by a process non-functional language that contains the definition of many NF-Attributes, many NF-Realizations and one NF-Requirement. The non-functional requirements then can be integrated with the software architecture elements. The refinement rules define how a concrete non-functional architecture is obtained from an abstract non-functional one. In the last step of the development process, the mapping strategy guides how the abstract elements may be mapped into actual implementation elements.

2.2.2 SPE

The performance of a system at early development stage (concept, requirement, and design) can be validated by constructing and evaluating a system performance model. In using the model-based approach, the following problems must be addressed:

1) In pre-implementation stages factual information is limited: final software plans have not been formulated, actual resource usage can only be estimated, and workload characteristics must be anticipated.

2) The large number of uncertainties introduces the risk of model omissions: models only reflect what you know to model, and the omissions may have serious performance consequences.

3) Thorough modeling studies may require extensive effort to study the many variations of operational scenarios possible in the final system.

4) Models are not universal: different types of system assessments require particular types of models. For example the models of typical response time are different from models to assess reliability, fault tolerance, performance, or safety.

The SPE approach [23] incorporates the model-based techniques with the performance engineering process for mitigating these problems. There are five steps in using SPE approach to validate a system performance:

1) Capture performance requirements, and understand the system functions and rates of operation.

In most systems there are several types of response, with different requirements. Users can be asked to describe performance requirements, to identify the types and the acceptable delays and capacity limitations. An experienced analyst must review their responses with the users. Practices for obtaining performance requirements are poorly developed. Research is needed on questions such as tests for realism, testability, completeness and consistency of performance requirements, on methodology for capturing them, preferably in the context of a standard software notation such as UML, and on the construction of performance tests from the requirements.

2) Understand the structure of the system and develop a model which is a performance abstraction of the system.

In this step a software execution model is created, representing the functions the software must perform. It traces scenarios through typical execution paths. Software execution models that represent the sequence of operations, including precedence, looping, choices, and forking/joining of flows, is used to capturing scenarios which represent different types of response. Then a performance evaluation model (system execution model, a queuing model) is created. Demands captured in an execution graph can be converted into parameters of a queuing model.

3) Capture the resource requirements and insert them as model parameters.

There seem to be four different sources of actual values for execution demands:

a) Measurements on parts of the software which are already implemented, such as basic services, existing components, a design prototype or an earlier version [15],

b) Compiler output from existing code,

c) Demand estimates (CPU, I/O, etc.) based on designer judgment and reviews,

d) “Budget” figures, estimated from experience and the performance requirements, may be used as demand *goals* for designers to meet (rather than as estimates for the code they will produce).

The actual demand parameters for CPU time, disk operations, network services and so forth are estimated and inserted into the queuing model.

4) Solve the model and compare the results to the requirements.

Queuing network models are relatively lightweight and give basic analytic models which solve quickly. However the basic forms of queuing network models are limited to systems that use one resource at a time.

5) Follow-up: interpret the predictions to suggest changes to aspects that fail to meet performance requirements.

If the requirements are not met, the analysis will often point to changes that make it satisfactory. These may be changes to the execution platform, to the software design, or to the requirements. Changes to the software design may be to reduce the cost of individual operations, or to reduce the number of repetitions of an operation. Larger scale changes may be to change the process architecture or the object architecture, to reduce overhead costs or to simplify the control path. If the bottleneck is a processing device

(processor, network card, I/O channel, disk, or attached device of some kind) then the analysis can be modified to consider more powerful devices. If the cost of adapting the software or the environment is too high, one should finally consider the possibility that the requirements are unrealistic and should be relaxed.

2.2.3 PASA

PASA [24] is a method for the performance assessment of software architects. It uses the principles and techniques of SPE to identify potential areas of risk within the architecture with respect to performance and other quality objectives. If a problem is found, PASA also identifies strategies for reducing or eliminating those risks. PASA approach is scenario-based. These scenarios provide a means of reasoning about the performance of the software as well as other qualities. The PASA approach consists of the nine steps summarized below:

1) Process overview --- The assessment process begins with a presentation designed to familiarize both managers and developers with the reasons for an architectural assessment, the assessment process, and the outcomes.

2) Architecture overview --- in this step, the development team presents the current or planned architecture.

3) Identification of critical use case --- the externally visible behaviors of the software that are important to responsiveness or scalability is identified.

4) Selection of key performance scenarios --- for each critical use case, the scenarios that are important to performance are identified.

5) Identification of performance objectives --- Precise, quantitative, measurable performance objectives are identified for each key scenario.

6) Architecture clarification and discussion --- Participants conduct a more detailed discussion of the architecture and the specific features that support the key performance scenarios. Problem area is explored in more depth.

7) Architectural analysis --- the architecture is analyzed to determine whether it will support the performance objectives.

8) Identification of alternatives --- if a problem is found, alternatives for meeting performance objectives are identified.
9) Presentation of results --- results and recommendations are presented to managers and developers.

2.2.4 UCM2LQN

In [25], a performance aware software development approach was proposed. In this approach, a software performance model, LQN, is created from scenarios, which is described with UCM.

Use Case Maps (UCMs) are used as a visual notation for describing causal relationships between responsibilities of one or more use cases. A Use Case Map is a collection of elements that describe one or more scenarios in a system. A scenario is represented by a path, shown as a line from a start point to an end point, and traversed by a token from start to end. Paths can be overlaid on components representing functional or logical entities, which may represent hardware or software resources. Responsibilities represent functions to be accomplished. The generation of performance models assumes that computational workload is associated with responsibilities, or is overhead implied by crossings between components. Responsibilities are annotated by service demands (number of CPU or disk operations, or calls to other services) and data store operations.

Compared to the UML, UCMs fit in between Use Cases and UML behavioral diagrams. UCMs provide a behavioral framework for evaluating and making architectural decisions at a high level of design. UCMs bridge the gap between requirements and design by combining behavior and structure in one view and by flexibly allocating scenario responsibilities to architectural components.

Layered Queuing Networks (LQN) models contention for both software and hardware resources, based on request for services. Entities in the role of clients make service requests and queue at the server. In LQN, servers may make requests to other servers, with any number of layers. An LQN can thus model the performance impact of

the software structure and interaction, and be used to detect software bottlenecks as well as hardware performance bottlenecks.

In an LQN the software resources are called tasks, and the hardware resources are called devices. Tasks can have priority on their CPU. An LQN can be represented by a graph with nodes for tasks and devices, and arrows for service requests.

A program called UCM2LQN automatically extracts a LQN performance model from a UCM scenario model. The LQN model is completed by adding details of the hardware and software aspects of the execution environment.

In this chapter, the composition and decomposition of system properties using formal methods and the architecture approach, as well as the incorporation of system non-functional requirements into early design phase based on software architecture are discussed. Due to the lack of mathematical foundation, support of programming languages, and corresponding tools, much more research work is needed.

In the next chapter, a framework, UniFrame, for integrating heterogeneous software components is presented.

3. OVERVIEW OF UNIFRAME

UniFrame [26] is a framework for building distributed systems by integrating geographically distributed software components with an emphasis on the QoS and heterogeneity. The salient features of UniFrame are: presence of a Meta component model, a seamless interoperation of heterogeneous software components, an active component resource discovery mechanism, a QoS-aware system development, and a generative system production.

3.1 Meta Component Model

In UniFrame, a meta component model is presented. The meta component model consists of the following parts:

- Components: The UniFrame approach is component-based. Components are considered to be autonomous entities with non-uniform implementations. This means that the components may adhere to diverse distributed computing models. Every component has three aspects:
  - Computational Aspect: This refers to the task carried out by the component. It is a form of introspection by which every component describes its services to other components. UMM uses two categories of parameters namely:
    - Inherent parameters: This consists of simple textual information containing the book-keeping information of a component.
    - Functional parameters: This consists of a formal and precise description of the computation, its associated contracts and the levels of service that the component offers.
  - Cooperative Aspect: This consists of,
    - Pre-processing collaborators: other components on which this component depends.

    - Post-processing collaborators: other components that may depend on this component.
  - Auxiliary Aspect: This aspect addresses issues like mobility, security and fault tolerance of a component.
- Service and Service Guarantees: A UMM component offers services that may be in the form of an intensive computational effort or an access to underlying resources. The quality of the service offered by a component plays an important role in whether or not the component is selected for a given system. The quality of service of a component is an indication of the component developer's confidence in the ability of that component to carry out a specified service. In UMM, every component must specify the quality of service that it can offer in terms of the QoS Parameters.
- Infrastructure: UMM utilizes the headhunters and the Internet Component Brokers (ICB) as infrastructure to address the issue of interoperability between heterogeneous DCS models. A detailed description of the infrastructure is given in section 3.3.

3.2 Seamless Interoperation of Heterogeneous Software Components

Software components can be developed using different component models, such as, CORBA, Java, and .Net. Currently, there is no universally accepted component model. It can be expected that in the near future, the heterogeneity of components will still exist. The glue and wrapper technique allows the composition of heterogeneous components seamlessly. In UniFrame, the glues and wrappers can be generated automatically based on Two-Level-Grammar [28].

3.3 Active Component Resource Discovery

As components are deployed on network, a discovery service is needed to search those components. UniFrame provides a resource discovery service, URDS [29], as shown in Figure 3.1, which consists of the following components: Headhunters, Active Registries (AR) and Internet Component Brokers (ICB).

The responsibilities of the headhunter include: detection of presence of service providers (service discovery), registration of functionality of the service providers and returning to the Query Manager a list of discovered service providers that match the requirements.

Active-Registries listen and respond to multicast messages from headhunters. Each also has introspection capabilities to discover not only the instances, but also the specifications of the components registered with them.

The ICB consists of Query Manager (QM), the Domain Security Manager (DSM), Link Manager (LM) and Adapter Manager (AM).

The QM is used to translate a system integrator's requirements specification for a component into a Structured Query Language (SQL) statement and dispatch this query to the appropriate headhunters. The headhunters, in turn, return lists of service provider components that match the search criteria contained in the query. The QM and the Link Manager together are responsible for propagating the queries to other linked ICBs.

The URDS discovery protocol is based on periodic multicast announcements. The multicasting exposes the URDS to a number of security threats. The DSM is responsible for ensuring that the security and integrity of the URDS are maintained. The security scheme implemented by the DSM involves the generation and distribution of secret keys for the ICB. It also enforces multicast group memberships and controls access to multicast addresses allocated for a particular domain, through authentication and use of Access Control Lists. Access Control Lists allow a sender or an authorized third party to maintain an inclusion or an exclusion list of hosts on the Internet corresponding to a multicast group. Each time a host requests to join the multicast group, the sender or the third party checks with the access control list to determine whether the host is authorized to join the group.

Link Manager establishes links between ICBs to form a federation and propagate the queries received from the QM to the linked ICBs. The ICB

administrator configures the LM with the location information of LMs of other ICBs with which links are to be established.

Adapter Manager: It acts as registry or lookup service for clients seeking adapter components. The adapter components register with the AM and at the same time indicates which component models they can bridge efficiently. The AM is contacted by the clients to locate the adapter components matching their requirements.

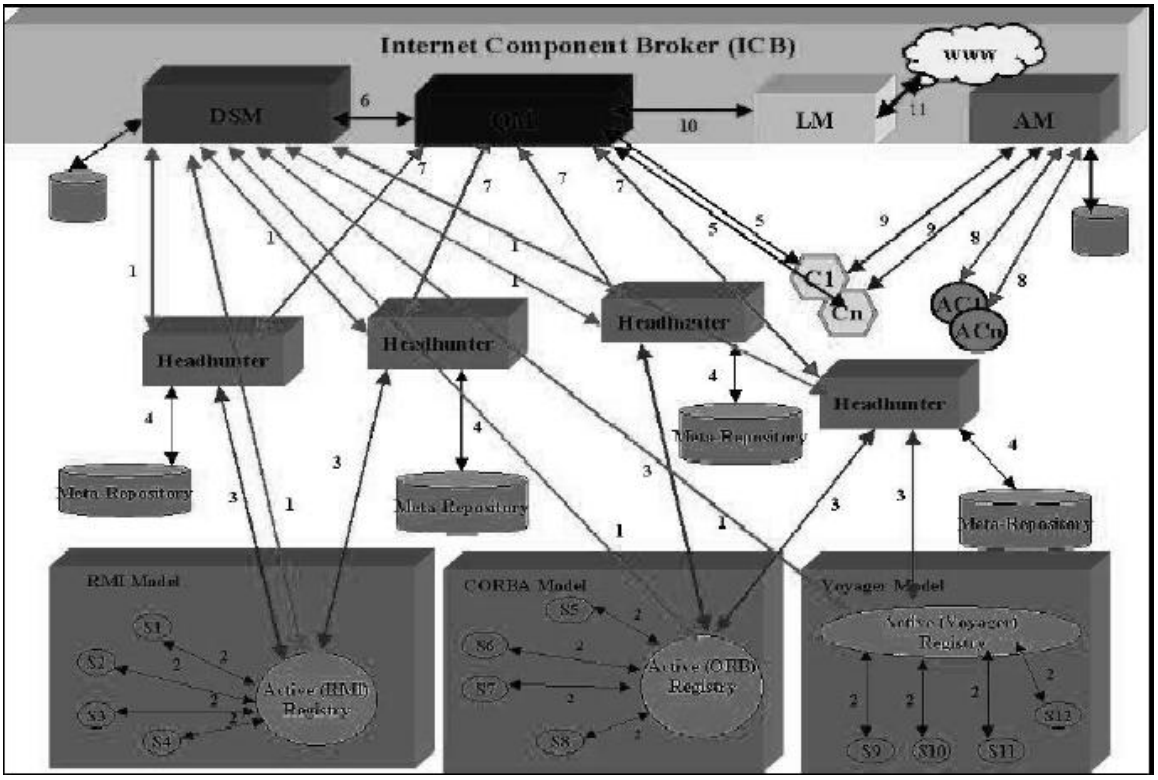

Figure 3.1 URDS Architecture (from [29])

3.4 QoS Aware System Development

With more and more applications requiring QoS guarantees, especially in multimedia, real-time and mission-critical applications, software developers need to consider QoS during the software developing process. In UniFrame, a QoS framework

[UQOS] is created to deal with QoS issues during the system development. The QoS framework consists of two levels: component level and system level. At the component level, it provides the QoS catalog [30] for certifying component QoS and formal specification of the component QoS. At the system level, UQOS provides the system QoS decomposition and composition rules, which decompose the system-level QoS into the QoS of individual component and compose the component-level QoS to reason about the system-level QoS. The dynamic testing of the individual component QoS and the system QoS is provided by use of event grammar [31].

The creation of a QoS catalog is the first step in an effort to build the UniFrame QoS framework. The QoS Catalog is intended to act as a tool for the standardization of the notion of quality of software components. The catalog contains detailed descriptions about the QoS parameters of software components including the metrics, evaluation methodologies, the factors influencing the parameters and the interrelationships among the parameters. The QoS Catalog, used in conjunction with the UniFrame approach, would force the component developer to consider and validate the QoS of a component before advertising its quality. The motivation for creating the QoS Catalog is two fold. It would prove to be a valuable tool for:

- The component developer, by:
  - Acting as a reference manual for incorporating QoS parameters into the components being developed.
  - Allowing him to enhance the performance of his components in an iterative fashion by being able to quantify their QoS parameters.
  - Enabling him to advertise the Quality of his components, after validation, by utilizing the QoS metrics.
- The System Developer, by:
  - Enabling him to specify the QoS requirements for the components that are incorporated into his system.
  - Allowing him to verify and validate the claims made by a component developer regarding the quality of component before incorporating it into the system.

- Allowing him to make objective comparisons of the Quality of Components having the same functionality.
- Empowering him with the means to choose the best-suited components for his system.

The general format used to describe each parameter in the catalog is outlined below:

- Name: Indicates the name of the parameter.
- Intent: Indicates the purpose of the parameter.
- Description: Provides a brief description of the parameter.
- Motivation: States the motivation behind the inclusion of the parameter and the importance of the parameter.
- Applicability: Indicates the type of systems where the parameter can be used.
- Model Used: Indicates the model used for Quantification of the parameter.
- Influencing Factors: Indicates the factors on which the parameter depends.
- Measure: Indicates the unit used to measure the parameter.
- Evaluation Procedure: Outlines the steps involved in the quantification procedure.
- Evaluation Formulae: Indicates the formulae used in the evaluation procedure.
- Result Type: Indicates the type of the result returned by the evaluation procedure.
- Nature: Indicates the nature of the parameter as suggested in [OMG02].
- Static/Dynamic: Indicates whether the value of the parameter is constant or varies during run-time.
- Increasing/Decreasing: Indicates whether higher values of the parameter correspond to better QoS (Increasing) or lower values correspond to better QoS (Decreasing).
- Consequences: Indicates the possible effects of using the chosen model to quantify the parameter.
- Related Parameters: Indicates the other related QoS parameters.
- Domain of Usage: Indicates the domains where the parameter is widely used.
- User Caution: It warns the user about the consequences of choosing a component with a lower level of a QoS parameter over another component (having the same functionality) with a higher level of the QoS parameter.
- Aliases: Indicates other prevalent equivalent names for a parameter, if any.

In UniFrame, the component QoS specification includes not only the QoS values but also the variables of component execution environment and component usage patterns [27].

Some of the environment variables are CPU speed, the memory, the disk bandwidth, network bandwidth, and the process priority assigned to the component. The fact that the environment variables can affect the QoS of a software component implies that any QoS associated with a software component would not necessarily hold true in foreign environments. Hence, it becomes critical to account for the effect of the execution environment on the QoS of software components. On the other hand, it is possible to enhance the QoS of a software component by suitably varying its execution environment. A component user might desire to improve a component's QoS (depending on the component's semantics) by suitably altering its execution environment (like providing a faster processor, increasing the memory etc).

Once a component is deployed on the network by the component user, it may be subjected to varying usage patterns, for example, the number of concurrent users for the component may be different at different time. The variations in the component usage patterns can have a profound impact on the QoS of a component. This in effect implies that it is crucial to be able to deduce the effect of usage patterns on the QoS of software components.

3.5 Generative System Production

UniFrame is component-based and thus it improves the system creation productivity by reusing components. In UniFrame, a generative domain model is used to further reuse the system architecture and facilitate the product generation from a product family by customization. The generative domain model (GDM) [32] consists of a problem space, a solution space, and the configuration knowledge mapping between them. The problem space consists of the application-oriented concepts and features that application developers can use to express their needs. The GDM contains a design space

model to represent the common and variable properties of a software architecture and a set of abstract components as specifications for creating reusable concrete components. The solution space consists of combinations of concrete components developed during the component engineering phase. The configuration knowledge includes illegal feature combinations, default settings, default dependencies, construction rules and optimization rules, etc. It also includes additional important knowledge, such as, QoS composition and decomposition rules, which help ensure the assembled distributed system meets not only the functional requirements but also the non-function requirements.

In UniFrame system development process, the system requirements are decomposed and abstract components with certain non-functional constraints are determined based on the generative domain model. The qualified concrete components can be found from network by use of the search activity in UniFrame. The final system can be generated by selecting proper components and integrating them. Finally the integrated system can be statically verified and dynamic tested.

In summary, UniFrame provides an approach, which has the potential to improve the productivity of software development and facilitate the development of software product with guaranteed quality of services. In the next chapter, the system composition and decomposition rules are introduced.

4. SYSTEM COMPOSITION AND DECOMPOSITION RULES

4.1 Introduction

In this chapter, system composition and decomposition rules are proposed to address the problem of how to factor the non-functional requirements of the entire system into the corresponding properties of individual components (top-down) and how to predict the non-functional properties of the integrated system based on the individual component properties (bottom-up).

4.2 Classification of System Non-Functional Properties

The non-functional properties of a system cover a wide range of the aspects of the system, and may have different attributes. The aim of this section is to investigate these non-functional properties from the angles of the system composition and the decomposition, and classify them into different categories. The classification of the system non-functional properties provides knowledge on how to treat these properties during system composition and decomposition.

4.2.1 Static/Dynamic Non-Functional Properties

Static non-functional properties can be evaluated by examining the internal structure of a software component. These properties are stable in different environments provided the internal structure of component is unchanged. The examples of static non-functional properties are *reliability, maintainability, portability, scalability, reusability, presentation, usability, security, priority, and parallelism constraints,* etc. Dynamic non-functional properties, on the other hand, can be measured by observing the system

behavior at run-time. These system properties are tightly associated with the deployment environment. Examples of dynamic properties are *throughput, turnaround time, capacity, availability, result,* etc.

From the point of the system composition and decomposition, static non-functional properties may compose well as they tend not to change during the system execution. The dynamic non-functional properties are influenced by the execution environment, which includes computational resources such as the CPU time, the memory, the disk bandwidth; communication resources such as the network bandwidth; the software resources such as the lock, the pool, the buffer, the semaphores, and the interactions with other components. Most of these factors are not known in advance, thereby the composition of these properties becomes difficult.

4.2.2 Domain Dependent / Independent Non-Functional Properties

Different non-functional properties are emphasized in different application domains. For example, security is most important in the banking domain, while safety and reliability are highly demanded in health care systems. In different application domains, the same non-functional properties may (domain independent) or may not (domain dependent) have the same decomposition and composition rules. For example, the reusability is an example of a domain independent property, while the throughput is an example of a domain dependent property. The system reusability depends on the component with the minimum value of reusability. For a system with two components, if the two components are in a sequence, then the system throughput depends on the component with the minimum throughput; if the two components are in parallel, then the system throughput is the sum of the throughputs of the two components. Reliability is another example of domain dependent property. For a system with two components, if the two components are in serial configuration, the system is reliable if all of these two components are reliable. On the other hand, if the two components are in parallel or redundant configuration, then the system is reliable if at least one component is reliable. Obviously, the domain independent system properties are more convenient to deal with

than the domain dependent system properties from the angle of the composition and the decomposition, because the latter need further information from the specific application domains.

4.2.3 Compositional/Non-Compositional Non-Functional Properties

As defined in [17], the compositional property allows deducing the system property from the individual component properties using simple rules. If a property is not compositional at all, then the composed system property cannot be easily predicted based on the component properties using simple rules. For example, mass (M) and energy (E) are compositional properties because they are conservative. Therefore it is valid to write: $M_1+M_2=M_{1+2}$, $E_1+E_2=E_{1+2}$. However, temperature (T) is not compositional because there is no simple relation between the temperature of a system and the temperature of its subsystems. In this sense, $T_1+T_2=T_{1+2}$ is not valid. To obtain the temperature of a composed system, a relation of the temperature with the mass and the energy can be established as: $E=\kappa\times M\times T$, where κ is a constant. Based on this relation, the temperature in the composed system can be deduced in the following way:

$$E_{1+2}=\kappa\times M_{1+2}\times T_{1+2} \quad (4.1)$$

$$E_1=\kappa\times M_1\times T_1 \quad (4.2)$$

$$E_2=\kappa\times M_2\times T_2 \quad (4.3)$$

Since energy is compositional:

$$E_{1+2}=E_1+E_2 \quad (4.4)$$

$$\kappa\times M_{1+2}\times T_{1+2}=\kappa\times M_1\times T_1+\kappa\times M_2\times T_2 \quad (4.5)$$

$$T_{1+2}=\frac{M_1\times T_1+M_2\times T_2}{M_{1+2}} \quad (4.6)$$

Since mass is compositional:

$$T_{1+2} = \frac{M_1 \times T_1 + M_2 \times T_2}{M_1 + M_2} \qquad (4.7)$$

So

$$T_{1+2} = c_1 T_1 + c_2 T_2 \qquad (4.8)$$

Where $c_1 = \frac{M_1}{M_1 + M_2}$ and $c_2 = \frac{M_2}{M_1 + M_2}$. The temperature of a composed system is non-compositional because the values of the coefficients c_1 and c_2 in equation (4.8) depend on the properties (mass) of individual components.

The maintainability of a software system that is built from individual components is also non-compositional. If the maintainability is evaluated as the time unit per line of code [30], then the system maintainability can be written as:

$$M = \frac{time}{LOC} = \frac{\sum_{i=1}^{n} time_i}{\sum_{i=1}^{n} LOC_i} = \frac{\sum_{i=1}^{n} M_i \times LOC_i}{\sum_{i=1}^{n} LOC_i} = \sum_{i=1}^{n} \frac{LOC_i}{\sum_{j=1}^{n} LOC_j} \times M = \sum_{i=1}^{n} c_i \times M_i \qquad (4.9)$$

where $c_i = \frac{LOC_i}{\sum_{j=1}^{n} LOC_j}$ (i=1, 2, …, n), LOC=Lines of Code.

The response time of a software system is a compositional property because the system response time is the sum of the response time of individual components.

4.2.4 User-Oriented /System-Oriented Non-Functional Properties

Some non-functional properties have meaning to users while some other non-functional properties have meaning to system maintainers. The user-oriented properties are perceptible to users and usually associated with individual user cases. Examples of user-oriented properties are response time, accuracy of results, availability, usability, reliability, etc. The system-oriented properties are concerns of system maintainers. These properties are associated with the entire system. Examples of system-oriented properties are throughput, resource utilization, security, maintainability, reusability, portability, adaptability, etc.

In composition and decomposition of non-functional properties, the user-oriented properties are applicable in the use case level while system-oriented properties are applicable in the whole system.

4.3 The Decomposition Rules

In the UniFrame approach, the decomposition rules are used to factor the system non-functional requirements (QoS) into the individual abstract components (components determined based on the Generative Domain Model, GDM, of the application domain) of the target system. The non-functional requirements decomposed into individual abstract components are used as one of the criteria to search for those concrete components (components implement the functionalities of the corresponding abstract components) from the Internet.

Two levels of decomposition rules are proposed in this research: the domain independent rules and the domain specific rules. The domain independent decomposition rules (DIDR) are abstract rules over different domains and have no specific domain information. The domain independent rules can be reused in different domains. However, they are usually weak in decomposing system non-functional requirements. Therefore, the corresponding domain dependent rules are needed.

4.3.1 Domain Independent Decomposition Rules (DIDR)

The decomposition of a system non-functional property could be homogeneous or heterogeneous. For a homogeneous decomposition, the system non-functional property (X) is decomposed to the same non-functional property (X) of individual components. For a heterogeneous decomposition, the system non-functional property X is decomposed to a different non-functional property (Y) of individual components.

For a homogeneous decomposition, the following de-compositional properties are identified: a*n at-least-one property, a k-out-of-n property*, *a universal property*, and *a component-specific property*. A property X is an at-least-one property when any system that has property X contains at least one component having the property X. An example of at least one property is: “platform dependent”. A system is platform dependent means at least one component in the system is platform dependent. For example, a system consists of four components, of which three are pure Java components and the fourth component is a Java component but calls a C procedure via JNI (Java Native Interface). The three pure Java components are platform independent while the fourth component is platform dependent. Therefore, the system is platform dependent. A property X is a k-out-of-n property when any system that has the property X contains at least k out of n components having the property X (n is the number of components in the system). Consider the reliability in a fault tolerant system in which users can access documents over replicated Web servers. The fault tolerant algorithm breaks the user request for a Web document, which is assumed to be replicated among n different servers, into n requests such that any k replies are sufficient to reconstruct the whole page. Hence, the system will work properly if at least k servers work properly. Hence the reliability of this fault tolerant system is a k-out-of-n property. A property X is a universal property when any system that has the property X contains components all having the property X. An example of a universal property is the safety property. A system is safe only if all components in the system are safe. The turn-around time is another example of a universal decomposition property. “The system turn-around time must be less than 1500 msec” is universally decomposed into each component in the system. Hence, each component in the system has property: “turn-around time must be less than 1500 msec”.

A property X is a component-specific property when any system that has the property X contains a particular component having the property X. For example, usability and presentation of a system are decomposed into the usability and the presentation of a user-interface component in the system.

For a heterogeneous decomposition, an *at-least-one-X property* and a *universal-X property* are identified. A property Y is an at-least-one-X property when any system that has the property Y contains at least one component having the property X. A property Y is a universal-X property when any system that has the property Y contains components all having the property X. For example, in scientific visualization, the requirement for the frame rate is mapped into the requirements for the throughputs of all the individual components in the system: the data source, the filter, the mapper, and the renderer. In this case, frame rate is a universal-X property, where X is property throughput.

4.3.2 Domain Specific Decomposition Rules (DSDR)

Because domain independent decomposition rules are abstract rules over different domains, they may not strong enough for a specific domain in decomposing system non-functional requirements. It is necessary to built domain specific decomposition rules for individual domains and to make them one part of the domain model. The domain specific rules can be created manually by a domain expert or automatically by use of tools. Therefore, the domain model contains information for decomposing system functional requirements as well as system non-functional requirements. The following two examples show the domain specific decomposition rules in a Bank ATM system.

A bank ATM system, shown in the Figure 4.1, consists of one ATM, one customer validation server, one transaction server manager, two transaction servers, and one account database. The domain specific decomposition rules of throughput for this system could be: the throughput of the customer validation server should be equal or greater than the system throughput; the throughput of the transaction server manager should be equal or greater than the system throughput; the throughput of the transaction server should be equal or greater than one half of the system throughput assuming the

load is equally balanced between two Transaction Servers; the throughput of the account database should be equal or greater than the system throughput. Compared to the domain independent decomposition rules of throughput, which is: the throughput of the individual component is greater or equal to the system throughput, the domain specific decomposition rules are stronger in decomposing the system throughput into individual components.

In creating the domain dependent decomposition rules of turn-around time for the bank ATM system, the critical use cases are used to decompose the system requirement on turn-around time. Critical use cases are those that are important to the operation of the system, or that are important to a system property observed by the user [12]. For each critical use case, one can focus on the scenarios that are executed frequently and on those that are critical to the user's perception of performance for some systems.

In the bank ATM system, there is several use cases: "customer validation", "deposit money", "withdraw money", "transfer money", and "check balance". Assume in this application domain, the critical use cases are "deposit money", "withdraw money", and "transfer money".

The critical use cases are described by sequence or collaboration diagrams. Given the system turn-around time, the budget of the turn-around time assigned to individual interfaces of the components is determined based on the sequence of message flow. The assignment of turn-around time budget to the individual interfaces of the components could be equal or unequal. If it is unequal, the weight of the turn-around time for each interface needs to be determined based on the domain knowledge.

Figure 4.2 shows the sequence diagram of the deposit money use case. There are in total six operations on the component interface: *deposit money* on ATM, *login account* on the transaction server manager, *deposit money* on the transaction server, *get account* on the account database, *save account* on the account database, and *exit account* on the transaction server manager. The decomposition of the system turn-around time for this use case is: the turn-around time of the component should be less than 1/6 of system turn-around time, here assuming the turn-around time is equally assigned into components interfaces.

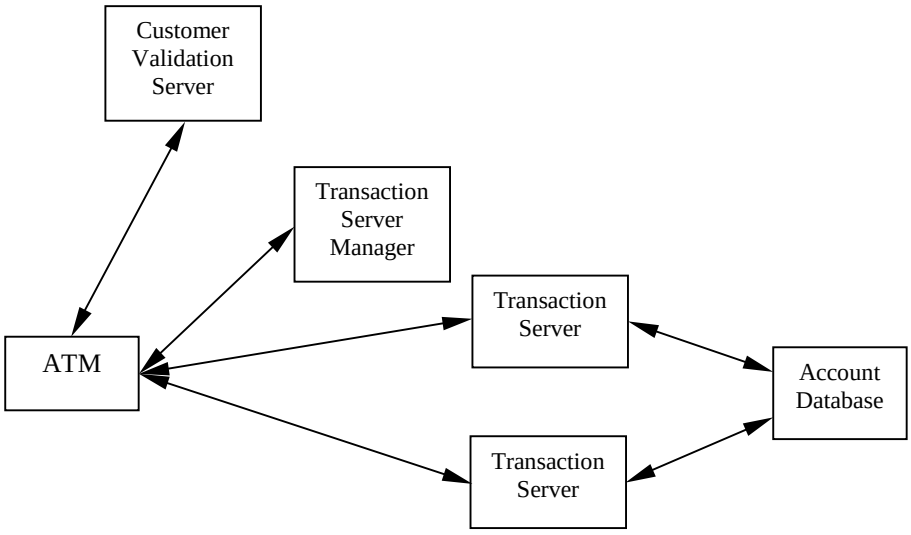

Figure 4.1 The Bank ATM System

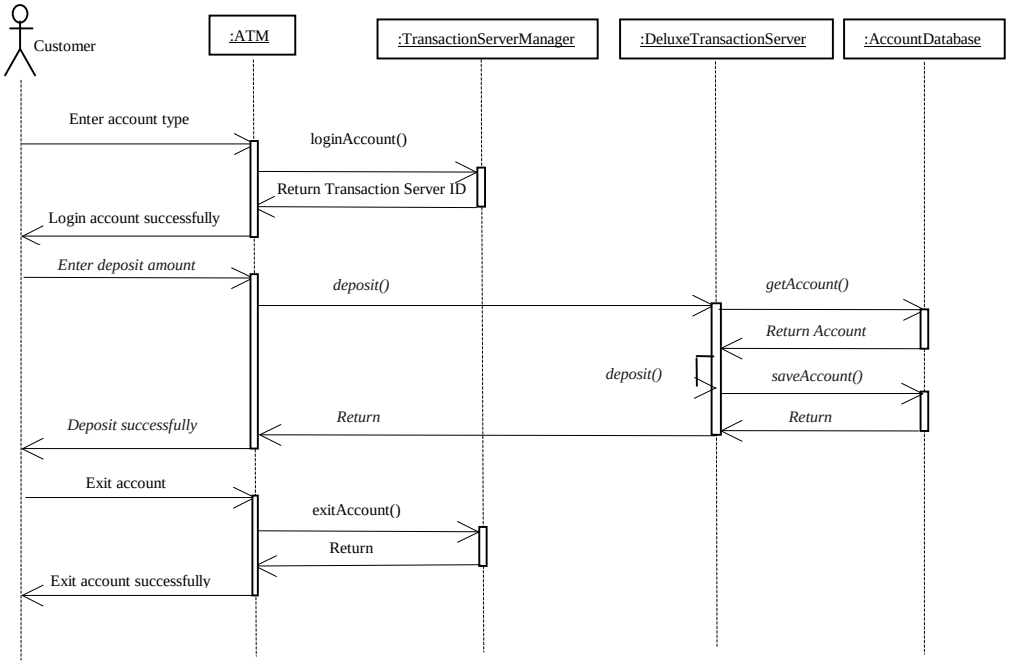

Figure 4.2 The Sequence Diagram of Deposit Money in the ATM System

4.4 The Category of Composition

A composition can be classified into three categories based on its effects on the non-functional properties of individual components: property-preserved composition, property-non-preserved composition, and property-emerging composition.

4.4.1 Property-Preserved Composition

In the property preserved composition, any property that holds at a component level also holds at the system level. In a property-preserved composition, when a component C_0 with a property p_0 integrates with a component C_1 with a property p_1, properties p_0 and p_1 hold in the compound component CC_1. When a component C_{i+1} integrates with a compound component CC_i (that is the composition of component 0 to i), the properties p_0, p_1, …, p_i and p_{i+1} hold in the compound component CC_{i+1}. The design task is, given a component C_i and a property p_i, find a component C_{i+1} and a mode of the composition such that the composition of C_i and C_{i+1} satisfy p_i. The static non-functional properties are preserved in composition. For example, in embedded systems, memory consumption is an important issue. As a component A with memory consumption of x bytes, which including the code and static data, composes with a component B with a memory consumption of y bytes, the property “component A consumes x bytes memory” still holds in the composition. The property-preserved composition is desired in system composition.

4.4.2 Property-Non-Preserved Composition

For property-non-preserved composition, the property of a component may not be preserved in composition due to the interactions with other components or its environment. For example, as a component composes with another component with different security policy domains or the security policies are inconsistent, the security level of either of the components can be compromised. As a component composes with

an un-trusted or malicious component, it may lose it security property. The property-non-preserved composition is not desired in system compositional reasoning.

4.4.3 Property-Emerging Composition

In the property-emerging composition, a new property may emerge in the composition of components. The new emerging property is not a property of any component present in the composition. The emerging property is due to the interactions among components. For example, as a deadlock-free component A composes with a deadlock-free component B, it is possible that the composed system has deadlock if the two components share some resources and access the shared resources in an improper order.

4.5 The Composition Rules

The predictability of the component-based software system is critical to the success of the component-based software development approach. In addition to the reuse of implementation code, the properties of individual components need to be reused in reasoning about the system properties. Therefore, a prediction model of the system non-functional properties based on the properties of individual components is required.

In the UniFrame approach, the composition rules are used to predict or statically validate the system non-functional properties based on the non-functional properties of individual components. The composition rules are divided into domain independent rules and domain specific rules.

4.5.1 Domain Independent Composition Rules (DICR)

Due to the causal link between the property of the system and the properties of components in the system, it is assumed the property of the composed system depends on the properties of components in the system. Different properties may follow different

aggregation rules. For example, mass and energy follow the sum rule; temperature follows the average rule; strength follows the minimum rule; the composition rules of electrical resistance, electrical current and electrical voltage depend on the configuration of components. Some of the domain independent composition rules are described in the following sections.

4.5.1.1 The Minimum Rule

The minimum rule describes the system property as the minimum of the component properties:

$$P = min\ (p_1, p_2, \ldots, p_n), \qquad (4.10)$$

where P is the system property, and p_i (i=1, 2, …, n) is the property of the i^{th} component in the system. For example, the composition of security follows the minimum rule. The system security depends on the components with the minimum security.

4.5.1.2 The Maximum Rule

The maximum rule describes the system property as the maximum of the component properties:

$$P = max\ (p_1, p_2, \ldots, p_n), \qquad (4.11)$$

where P is the system property, and p_i (i=1, 2, …, n) is the property of the i^{th} component in the system. For example, the composition of packet loss follows the maximum rule. The packet loss of a network system depends on the network component with the maximum packet loss.

4.5.1.3 The Sum Rule

The sum rule describes the system property as the sum of the component properties:

$$P = sum\ (p_1, p_2, \ldots, p_n), \qquad (4.12)$$

where P is the system property, and p_i (i=1, 2, …, n) is the property of the i^{th} component in the system. For example, the composition of the response time follows the sum rule. The response time of a system equals the sum of the response time of the individual components.

4.5.1.4 The Weighted Sum Rule

The weighted sum rule describes the system property as the weighted sum of the component properties:

$$P = w_1p_1 + w_2p_2 + \ldots + w_np_n\ , \qquad (4.13)$$

where w_i (i=1, 2, …, n) is a constant coefficient within the range of [0, 1]. For example, the composition of maintainability follows the weighted sum rule. The maintainability of a system equals to:

$$c_1M_1 + c_2M_2 + \ldots + c_nM_n \qquad (4.14)$$

where $c_i = \frac{LOC_i}{\sum_{j=1}^{n} LOC_j}$ (i=1, 2, …, n), LOC=Lines of Code, and M_i is the maintainability score of the i^{th} component.

4.5.1.5 The Product Rule

The product rule describes the system property as the product of the component properties:

$$P = p_1 \times p_2 \times \ldots \times p_n \,, \qquad (4.15)$$

where P is the system property, and p_i (i=1, 2, …, n) is the property of the i^{th} component in the system. For example, the availability (or reliability) follows the product rule if it is assumed the availabilities (or reliabilities) of the individual components are statistically independent.

4.5.2 Domain Specific Composition Rules (DSCR)

The domain independent rules are abstract rules and cannot be applied directly in a specific domain. The domain specific composition rules are derived or instantiated from the domain independent rules and incorporate the domain specific knowledge. The domain specific rules can be generated manually by domain experts or automatically by use of tools. The system non-functional properties are statically validated using the domain specific composition rules during the system validation phase. The following example shows how to create the domain specific composition rules for turn-around time in a bank ATM system.

The turn-around time is a user-oriented property. The composition of turn-around time is based on use cases. In this example, the composition rule of turn-around time for the withdraw money use case is illustrated. As shown in the Figure 4.3, the sequence of operations on component interfaces is: *withdraw money* on ATM, *login account* on the transaction server manager, *withdraw money* on the transaction server, *get account* on the account database, *save account* on the account database, and *exit account* on the transaction server manager. Therefore, the composition rule of turn-around time for the withdraw money use case is: TAT_{sys} (withdraw money) = TAT_{atm} (withdraw money) +

TAT_{tsm} (login Account) + TAT_{ts} (withdraw money) + TAT_{adb} (get account) + TAT_{adb} (save account) + TAT_{tsm} (exitAccount).

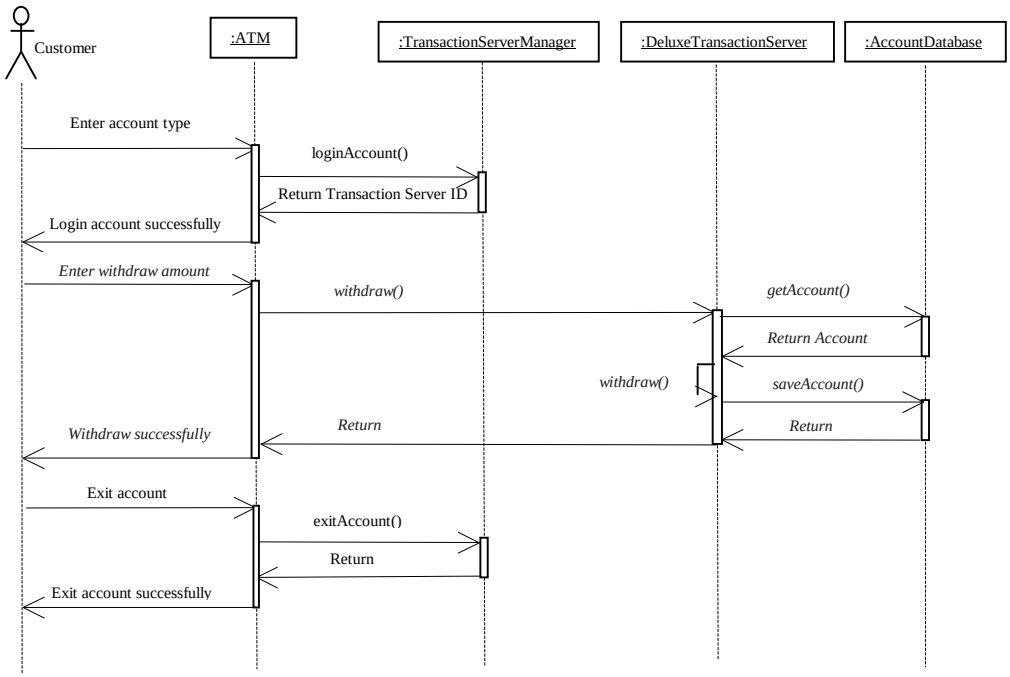

Figure 4.3 The Sequence Diagram of Withdraw Money in the ATM system

4.6 Summary

In this chapter, decomposition rules and composition rules are proposed. They are divided into domain independent rules and domain specific rules. The decomposition rules are used in factoring the system non-functional requirements into properties of individual components, while the composition rules are used in predicting the properties of a whole system based on the properties of individual components. In the next chapter, the inter-component communication patterns will be discussed and their effect on the composition and decomposition of non-functional properties will be investigated.

5. EFFECT OF INTER-COMPONENT COMMUNICATION PATTERNS ON SYSTEM COMPOSITION AND DECOMPOSITION

5.1 Introduction

In the previous chapter, the composition and decomposition rules of non-functional properties are discussed. However, these composition and decomposition rules did not consider the interactions between components.

A component may use different communication patterns to interact with different components or the same component at different times. The communication patterns between components can be invocation-based, event-based, or stream-based. The reason to study the inter-component communication patterns is to find the composition and decomposition rules associated with the individual communication patterns.

5.2 Invocation-Based Communication

The attributes of an invocation-based communication pattern are that the two components be aware of each other's identity and the communication be in a unicast mode. The invocation-based component interaction mode is widely supported by the current distributed component models, such as Java [5], CORBA [6], and .Net [7]. There are three subcategories of invocation-based communication: asynchronous one-way invocation, synchronous two-way invocation, and asynchronous two-way invocation. They are described in the following sections.

5.2.1 Asynchronous One-Way Invocation

In the one-way invocation, as shown in Figure 5.1, the client component returns immediately after it has sent a request to the server. The sever component returns no response to the client component. This invocation pattern could achieve a better throughput than other patterns. However, this one-way mode is inadequate for systems demanding a high degree of reliability because there is no way for the sender to know whether the call was successful or not.

5.2.2 Synchronous Two-Way Invocation

In a two-way synchronous invocation, as shown in Figure 5.2, when the client component sends a request to a server component, the client component blocks and waits for a response from the server component. If single-threaded, the client component is unable to perform any other work while it waits for a response.

5.2.3 Asynchronous Two-Way Invocation

In a two-way asynchronous invocation, the calling client can return immediately and serve subsequent calls after it invokes the server. Therefore, an asynchronous call can lead to a better throughput than the corresponding synchronous call. There are two variations of the asynchronous two-way invocation pattern: polling and call back. In the polling mode, as shown in Figure 5.3, the client component is not blocked after sending a request. It can continue to processes subsequent requests. The client component will poll the result from the server at a later time. In the call back mode, as shown in Figure 5.4, after sending a request, the client component can continue to process the subsequent requests. The server component will send the result back to the client when the server finishes the service of the request. For example, CORBA Asynchronous Method Invocation (AMI) provides polling and callback. In the polling mode, the invocation returns an object reference that can be queried at any time to obtain the status of the outstanding request. In the callback mode, a client passes a callback object reference as

part of the invocation. When the reply is available, that callback object is invoked with the data of the reply. In both cases, the client will not block in making requests.

An asynchronous invocation can lead to a better throughput than the corresponding synchronous invocation. The synchronous invocation is more suitable when an application needs reliable request processing and error handling. An asynchronous invocation contains a more complex programming model than the synchronous invocation model. The asynchronous model provides more services, such as message routing, reliable message delivery, message priority and ordering, etc., at the cost of a greater application complexity and more work on the part of developers.

An invocation-based communication pattern only allows a one-to-one communication model and forces a tight coupling between the involved parties. It does not scale well for large Internet-wide systems because of the large number of potential communication partners, and the dynamic nature of all interactions with new clients joining the system and servers failing. The event-based communication pattern is designed for the large-scale distributed systems.

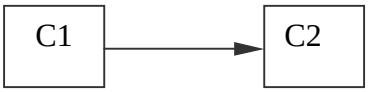

Figure 5.1 The Asynchronous One-Way Invocation-Based Communication Pattern (Pattern No. 1)

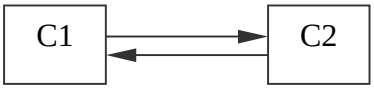

Figure 5.2 The Synchronous Two-Way Invocation-Based Communication Pattern (Pattern No. 2)

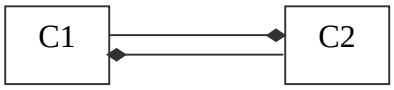

Figure 5.3 The Asynchronous (Polling) Two-Way Invocation-Based Communication Pattern (Pattern No. 3)

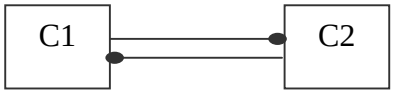

Figure 5.4 The Asynchronous (Callback) Two-Way Invocation-Based Communication Pattern (Pattern No. 4)

5.3 Event-Based Communication

The event-based communication model represents an emerging paradigm for asynchronously interconnecting components that comprise an application in a potentially distributed and heterogeneous environment, such as large-scale Internet services and mobile programming environments.

The attributes of an event-based communication pattern are that the two components not be aware of each other's identity. An event-based communication is essentially asynchronous, which results in a less tightly coupled communication relationship between the application components compared to the traditional request/response communication model. Event-based communication provides a one-to-many or many-to-many communication pattern. The CORBA Event Service and Java Messaging Service (JMS) are examples of the event-based communication model. Event-based communication includes the following styles:

♦ Push style: as shown in Figure 5.5, suppliers are initiators of events and consumers passively wait to receive events. An event broker plays the role of a notifier in this approach.

♦ Pull style: as shown in Figure 5.6, consumers are initiators of events and suppliers passively wait to get events pulled from them. An event broker plays the role of a producer in this case.

♦ Push-and-pull style: as shown in Figure 5.7, suppliers push the events to the event brokers and the consumers pull events from the event brokers. Both suppliers and consumers are active. An event broker plays the role of a queue in this approach.

♦ Pull-and-push style: as shown in Figure 5.8, event brokers pull events from the suppliers and then push them to the consumers. Both suppliers and consumers are passive. An event broker functions as an active agent in this method.

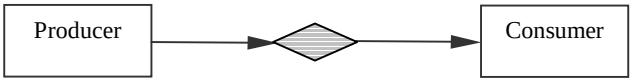

Figure 5.5 The Push Style Event-Based Communication Pattern (Pattern No. 5)

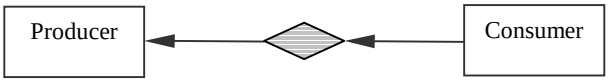

Figure 5.6 The Pull Style Event-Based Communication Pattern (Pattern No. 6)

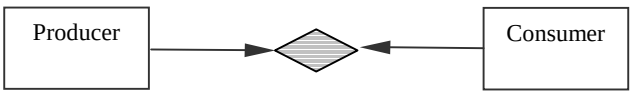

Figure 5.7 The Push-and-Pull Style Event-Based Communication Pattern (Pattern No. 7)

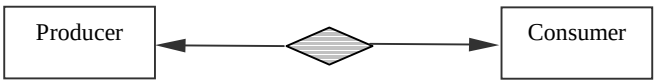

Figure 5.8 The Pull-and-Push Style Event-Based Communication Pattern (Pattern No. 8)

5.4 Stream-Based Communication

The attributes of a stream-based communication pattern are that it deals with the continuous media data and the dataflow is unidirectional. There is generally a single source and one or more sinks (unicast or multicast). Simple stream type consists of a single flow of data, e.g., audio or video. Complex stream type consists of multiple data flows, e.g., stereo audio or combination of audio/video. Typically, it uses the UDP transport protocol.

Message-oriented communication may occasionally happen and usually has high-level semantics. It is sensitive to the loss of messages. However, its requirement on the delivery latency is moderate, as long as it is within reasonable bounds. In contrast, the stream-based communication may constantly occur. Its semantic level usually is relatively low and the drop of data units up to several is usually tolerable. But it is sensitive to the delivery latency and variation of the delivery latency.

Stream-based communication consumes more network bandwidth compared to bursty data communication. Therefore, the network component would be the potential bottleneck of the system. The QoS of the network component would play an important role in the system QoS, so it cannot be neglected.

There is a need to specify the QoS of stream-based communication and the translation to resource reservations in the underlying communication system. There is no standard way of specifying QoS, describing resources, and mapping QoS specifications to resource reservations.

Figure 5.9 The Stream-Based Communication Pattern (Pattern No. 9)

5.5 The Factors Associated with Individual Communication Patterns

In addition to the four types of communication patterns discussed in the previous sections, the factors that are associated with individual inter-component interaction are also important and discussed in the following subsections.

5.5.1 Transport Protocols

The components may choose different transport protocols, such as TCP and UDP to deliver its services. TCP uses three-way handshake to establish a connection and provides a sequence number for their interactions, an acknowledgement (ACK) and a congestion control to make the data transfer reliable. UDP has no connection establishment overhead and eliminates the TCP ACK packets and the retransmission, so UDP is faster and has better throughput. UDP does not provide a sequence number, congestion control or error control, so it is unreliable. Wireless middleware, real-time middleware, and multimedia middleware normally use the UDP/IP protocol for performance reasons. The protocol chosen for a particular data type is, in large part, determined by the trade off between the speed and the accuracy of data communication. Due to the fact that TCP is a reliable service, delays will be introduced whenever a bit error or packet loss occurs. This delay is caused by retransmission of the broken packet, along with any successive packets that may have already been sent. This can be a large source of jitter. The TCP transport protocol is suitable for RPC and object invocation, but not for some other inter-component interactions, such as streaming media. On the contrary, the UDP transport protocol is suitable for a streaming communication pattern.

5.5.2 Component Access Patterns

A component can be accessed by an individual request or multiple simultaneous requests. As seen from the Figure 5.1, in the light load zone, as the number of concurrent requests increases, the throughput grows almost linearly and the response time remains relatively constant. In the heavy load zone, the throughput remains relatively constant, but the response time increases proportionally with the number of client requests. At some point in the buckle zone, one of the resources such as the disk bandwidth becomes exhausted, causing the throughput to degrade and the response time to increase drastically.

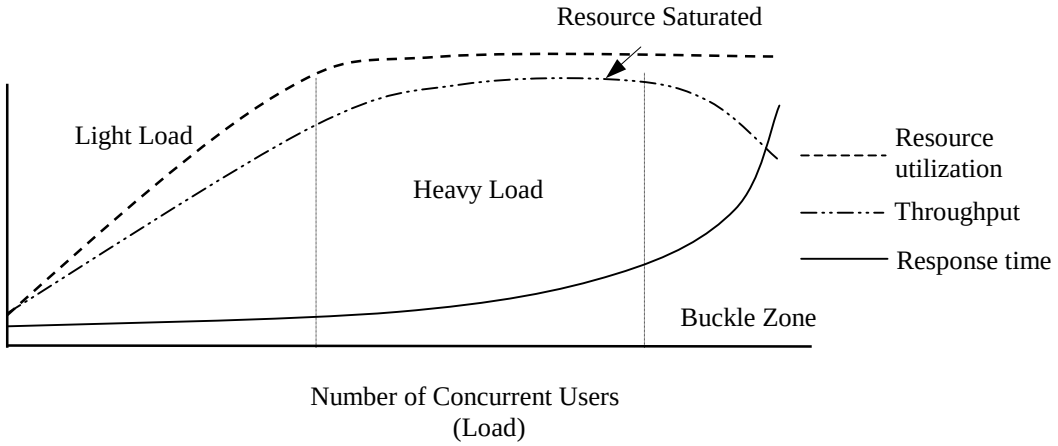

Figure 5.10 A Typical Relationship Between the Load, the Response Time and the Throughput (from [21])

5.5.3 Sequence of Component Interactions

Some system properties are not symmetrical with respect to the sequence of component interaction. For example, the turnaround time from A $\rightarrow$ B $\rightarrow$ C may differ from C $\rightarrow$ B $\rightarrow$ A, based on the actual path in each direction. The priority scheme of a composed system is not symmetrical either. For example, suppose component A and B service incoming requests based on priority and they use different priority schemes: component A assign higher priority for subscribed users and lower priority for public

users; component B assigns higher priority for small sized data files and lower priority for large sized data files. If clients call component A and then component A calls component B, then the overall priority scheme is determined by component B. That means a request for a small data file get fast response and a request for a large data file gets slow response no matter whether the user is subscribed or public. On the other hand, if clients call component B and then component B calls component A, the overall priority scheme is determined by component A.

5.5.4 Data Type and Data Size

Different types of data can be sent to a component, such as short, long, struct, object, etc. A complex data type may need more time for marshaling or un-marshaling resulting in additional overhead.

5.6 Composition of Communication Patterns

This section describes complex communication patterns composed from basic communication patterns and the composition of communication patterns in real applications.

5.6.1 Composition of the Basic Communication Patterns

The basic communication patterns described in 5.2, 5.3 and 5.4 are elements in building complex communication patterns. A complex communication pattern can be the result of a certain kind of composition of the basic communication patterns.

Figure 5.11 shows the sequential composition of two basic one-way invocation-based communication patterns. Figure 5.12 shows the sequential composition of two synchronous two-way invocation-based communication patterns. In this composite communication pattern, the receiver of the first communication pattern, C_2, is the initiator of the second communication pattern. Figure 5.13 shows the sequential composition of

two asynchronous invocation-based communication patterns. In this composite communication pattern, the component C_1 and C_2 do not block themselves as they invoke the subsequent component (C_2 and C_3). Hence, the performance of this pattern is better than the performance of the pattern No. 11. Figure 5.14 shows the sequential composition of synchronous two-way invocation-based communication pattern and asynchronous two-way invocation-based communication pattern. In this composite communication pattern, the component C_2 does not block itself as it invokes the component C_3. Therefore, compared to the pattern No. 11, the performance of this pattern is better. Figure 5.15 shows the sequential composition of asynchronous two-way invocation-based communication pattern and synchronous two-way invocation-based communication pattern. In this composite communication pattern, the component C_1 does not block itself as it invokes the component C_2. Therefore, its performance is better than the performance of the pattern No. 11. Figure 5.16 shows the filter-style composition of two synchronous two-way invocation-based communication patterns. In this composite communication pattern, the first communication pattern is activated more frequently than the other one. In another words, the first communication pattern can filter out some requests, which cannot reach the second communication pattern. Figure 5.17 shows the forward sequential composition of synchronous two-way invocation-based communication pattern and asynchronous one-way invocation-based communication pattern. In this composite communication pattern, the response is directly forwarded to the component C_1 instead of the component C_2 from the component C_3. Figure 5.18 shows the partial sequential composition of two synchronous two-way invocation-based communication patterns. In this composite communication pattern, the two communication patterns have the same initiator of the communication, which is the component C_1. Figure 5.19 shows the partial sequential composition of asynchronous two-way invocation-based communication patterns. This composite communication pattern can potentially improve the performance of the component C_1. Figure 5.20 shows the partial sequential composition of synchronous two-way invocation-based communication pattern and asynchronous two-way invocation-based communication pattern. This composite communication pattern has the potential to improve the performance of the component C_1. Figure 5.21 shows the

partial sequential composition of asynchronous two-way invocation-based communication pattern and synchronous two-way invocation-based communication pattern. This composite communication pattern has the potential to improve the component C_1's performance. Figure 5.22 shows the parallel composition of two synchronous two-way invocation-based communication patterns. In this composite communication pattern, the two communication patterns are activated simultaneously. Figure 5.23 shows the partial sequential composition of two synchronous two-way invocation-based communication patterns. In this composite communication pattern, the two communication patterns have the same receiver, which is the component C_3. Figure 5.24 shows the partial sequential composition of two synchronous two-way invocation-based communication patterns between the same two components. In this composite communication pattern, the two communication patterns have the same initiator and receiver and are activated at different times. Figure 5.25 shows the fault tolerant composition of two synchronous two-way invocation-based communication patterns. In this composite communication pattern, if one of the communication patterns is broken, the other communication pattern is activated. Figure 5.26 shows the alternative composition of two synchronous two-way invocation-based communication patterns. In this composite communication pattern, one of the two communication patterns is activated, which depends on the request.

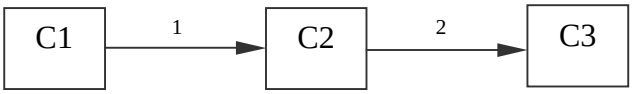

Figure 5.11 The Sequential Composition of Two One-Way Invocation-Based Communication Patterns (Pattern No. 10)

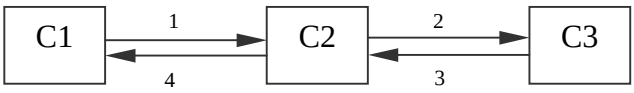

Figure 5.12 The Sequential Composition of Two Synchronous Two-Way Invocation-Based Communication Patterns (Pattern No. 11)

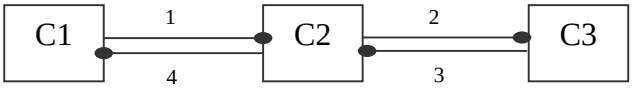

Figure 5.13 The Sequential Composition of Two Asynchronous (Callback) Two-Way Invocation-Based Communication Patterns (Pattern No. 12)

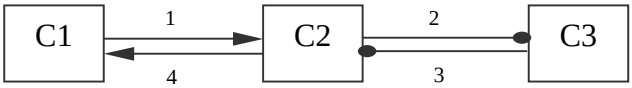

Figure 5.14 The Sequential Composition of Synchronous Two-Way Invocation-Based Communication Pattern and Asynchronous (Callback) Two-Way Invocation-Based Communication Pattern (Pattern No. 13)

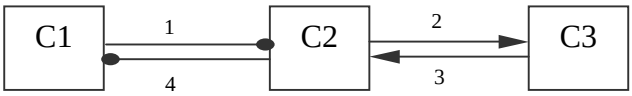

Figure 5.15 The Sequential Composition of Asynchronous (Callback) Two-Way Invocation-Based Communication Pattern and Synchronous Two-Way Invocation-Based Communication Pattern (Pattern No. 14)

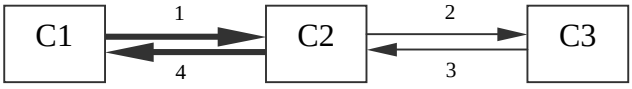

Figure 5.16 The Filter-Style Composition of Two Synchronous Two-Way Invocation-Based Communication Patterns (Pattern No. 15)

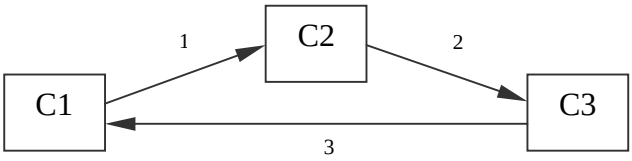

Figure 5.17 The Forward Composition of Synchronous Two-Way Invocation-Based Communication Pattern and Asynchronous One-Way Invocation-Based Communication Pattern (Pattern No. 16)

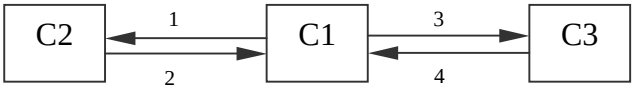

Figure 5.18 The Partial Sequential Composition of Two Synchronous Two-Way Invocation-Based Communication Patterns (Pattern No. 17)

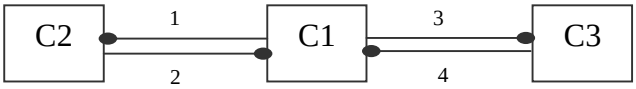

Figure 5.19 The Partial Sequential Composition of Two Asynchronous (Callback) Two-Way Invocation-Based Communication Patterns (Pattern No. 18)

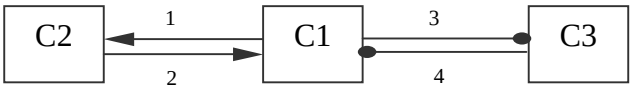

Figure 5.20 The Partial Sequential Composition of Synchronous Two-Way Invocation-Based Communication Pattern and Asynchronous (Callback) Two-Way Invocation-Based Communication Pattern (Pattern No. 19)

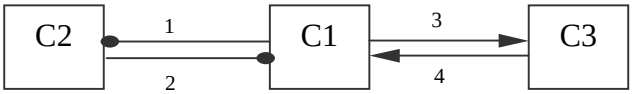

Figure 5.21 The Partial Sequential Composition of Asynchronous (Callback) Two-Way Invocation-Based Communication Pattern and Synchronous Two-Way Invocation-Based Communication Pattern (Pattern No. 20)

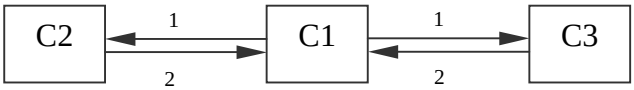

Figure 5.22 The Parallel Composition of Two Synchronous Two-Way Invocation-Based Communication Patterns (Pattern No. 21)

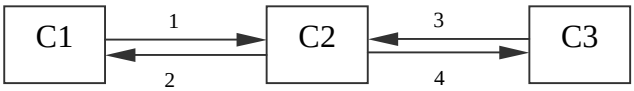

Figure 5.23 The Partial Sequential Composition of Two Synchronous Two-Way Invocation-Based Communication Patterns (Pattern No. 22)

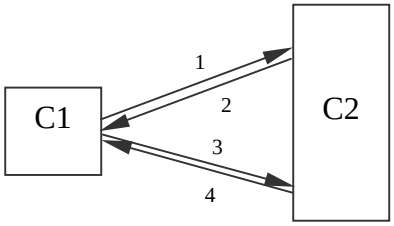

Figure 5.24 The Partial Sequential Composition of Two Synchronous Two-Way Invocation-Based Communication Patterns (Pattern No. 23)

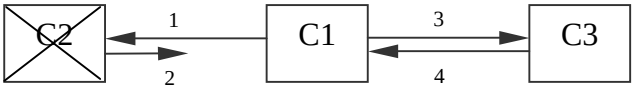

Figure 5.25 The Fault-Tolerant Composition of Two Synchronous Two-Way Invocation-Based Communication Patterns (Pattern No. 24)

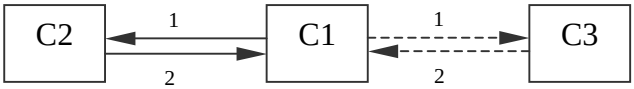

Figure 5.26 The Alternative Composition of Two Synchronous Two-Way Invocation-Based Communication Patterns (Pattern No. 25)

5.6.2 Composition of Communication Patterns in Real Applications

In the previous section, the compositions of the basic communication patterns are described. Based on the basic and composite communication patterns, a system can be built by composing the basic and composite communication patterns in a certain manner.

Figure 5.27 shows an example of the communication pattern 17. The client first sends a request with the server name to the DNS server. After getting the IP address of the Web server from the DNS server, the client then sends the request for the web page to the Web server. Eventually, the Web server responds to the client with the requested web page. In this example, the client initially contacts the DNS server using the two-way synchronous communication pattern, and then contacts with the Web server using the two-way synchronous communication pattern.

Figure 5.28 shows the example of the composition of communication pattern 17 and communication pattern 2. In this example, the ATM initially sends user's account number and pin number to the customer validation server to validate the user. If the validation is successful, the ATM then sends the account number to the transaction server manager. After getting the transaction server information from the transaction server manager, the ATM finally sends the user transaction request to the transaction server and

gets the result from the transaction server. The communication patterns between the ATM and the customer validation server, the transaction server manager, the transaction server are synchronous two-way invocation and have the same initiator, which is the ATM.

Figure 5.29 shows the example of communication pattern No. 15. In this example, the Web client sends a request to the Web proxy server. If the requested Web object is cached in the Web proxy server, the Web proxy server returns the Web object to the Web client. Otherwise, the Web proxy server sends the request for the Web object to the remote Web server on behalf of the Web client. When the Web proxy server receives the response from the Web server, it returns it to the Web client. A request for an object that is cached in the Web proxy server is filtered and is not forwarded to the Web server.

Figure 5.30 shows an example of an online shop selling music over the network [34]. In such a scenario, multiple service providers maintain databases of digital music tracks. A client wanting to buy music browses the tracks available at a particular on-line record service provider and can listen to streamed samples of tracks before paying for and downloading high-quality versions of the files onto their local computers.

From the above description, some of the high-level components of a record service system can be identified. The on-line record store is accessed through a component that maintains the database of information about the music in the store. The music itself is stored in one or more media servers. Components can be instantiated on these media servers to stream a low-resolution preview of the music or to download the file to a client’s local machine. The client program is made up of components that allow the user to visually browse the contents in the store, receive and play streams of audio, and download purchased files onto the local machine. Other components may be used to stream audio data to and from disk or audio devices and to process audio streams, to convert between formats for example.

These components interact in different ways, and each separate usage involving the same interaction type may need different qualities of service and levels of security. The client browses the contents of the music store by invoking request/reply operations over a reliable connection. When requesting a preview of a track, the client will receive a

stream of continuous media, which may not have to be reliable but requires guarantee about the bandwidth and the jitter. When requesting a purchase of one or more files, the client uses a request/ reply transaction over a reliable connection; however, unlike the connection used for browsing, the connection used for requesting a purchase must also be secure. Finally, the music files are transferred to the client over a network that efficiently transfers large amounts of data; this interaction also requires a reliable and a secure connection.

Figure 5.31 shows an example of a real-time content-based media access [35]. At the lowest level of the hierarchy, the media streams are filtered and transformed, e.g., transforming a video stream from color to black and white only, reducing the spatial or the temporal resolution. The transformed media streams are then fed to feature extraction algorithms. Feature extraction algorithms operate on samples or segments from the transformed media streams and calculate features such as the texture coarseness, the center of gravity, and the color histogram. Results from feature extraction algorithms are generally reported to classification algorithms that are responsible for detecting higher level domain concepts such as a “person” occurring in a media stream.

The interaction patterns in this example use the event-based model. Event based systems rely on the presence of an event broker. The responsibility of the event broker is to propagate events from the event producers to event consumers, generally in a many-to-many manner. An event produced by components at the bottom of the hierarchy is likely to be interesting to a large number of other components. A push style interaction fits one-many communication well, and the need for an explicit request message, introducing delay, is eliminated. From a resource consumption viewpoint it is important to execute complex and time-consuming algorithms only when absolutely necessary. This reasoning suggests that such algorithms should be demand driven, pulled by other components. For example, the classification component is allowed to pull the feature extraction component.

A classification component consumes events produced by other feature extraction components and/or classification components and generates events that again may be

consumed by other classification components, or reported directly to a user, or stored in a database as indexing information.

The effect of the inter-component communication patterns on the composition is discussed in the next section.

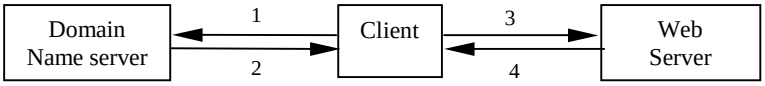

Figure 5.27 The Web Server Example

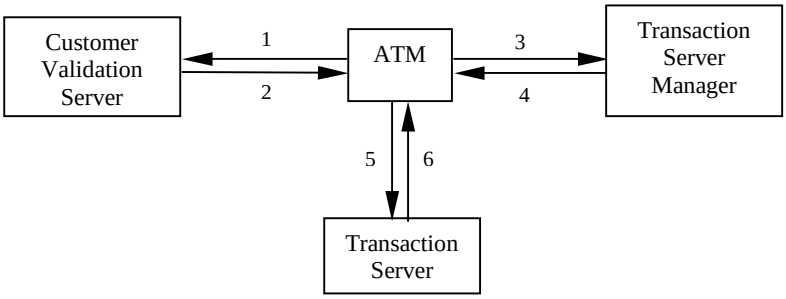

Figure 5.28 The Bank ATM Example

Figure 5.29 The Web Proxy Server Example

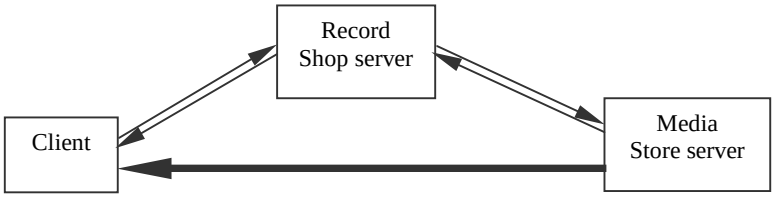

Figure 5.30 The On-Line Music Shop

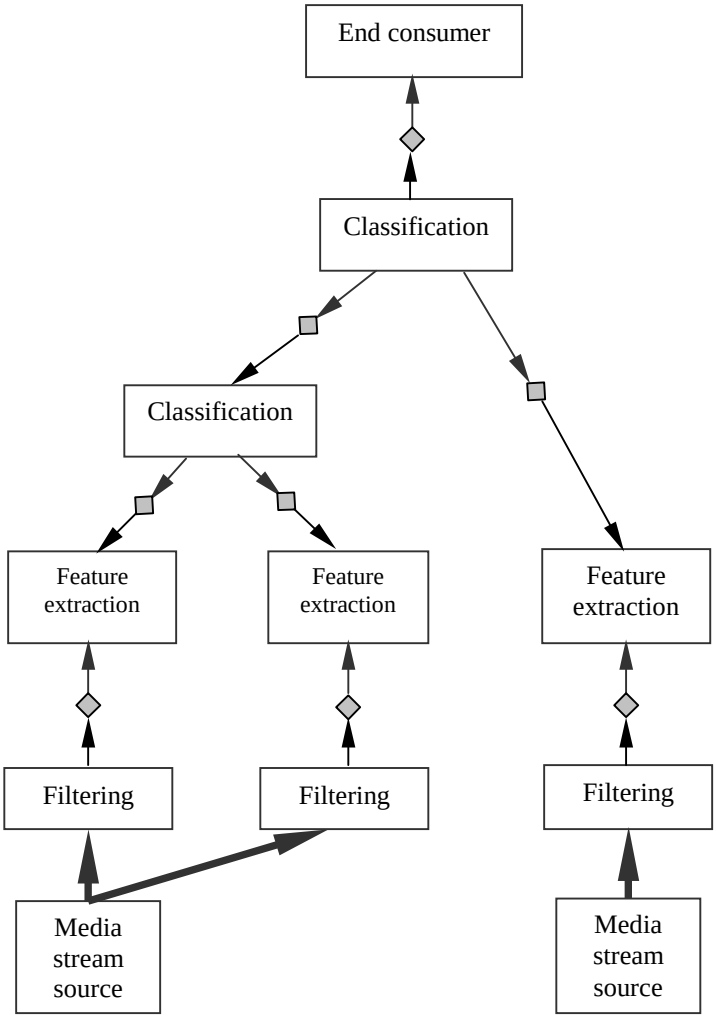

Figure 5.31 The Real-Time Content-Based Media Access

5.7 The Composition Rules of Response Time and Throughput under Different Communication Patterns

In this section, the composition rules of response time and throughput under different communication patterns and computation models are proposed. The communication patterns studied are patterns No. 2, 3, 11, 12, 13, 14, 17, 18, 19, and 20. The computation models considered are single threaded, multi-threaded, thread-per-request and thread pool.

In order to empirically validate the proposed composition rules, experiments were conducted on four Sun SPARC 5 workstations. These four workstations are connected via a 10 Mbit Ethernet. The components are implemented with C and BSD sockets. Each component performs some computation after receiving any request from another component. The throughput of individual components, or the system, is measured by sending a certain amount of requests to the components, or the system, and recording the time it takes to complete all these requests. The average response time of a request for individual components, or the system, is measured by sending requests one by one every certain amount of time, and recording the start and the end times of each request.

5.7.1 The Composition Rules of the Communication Pattern No. 2

The communication pattern No. 2 is a synchronous two-way invocation. Two types of components are considered: single threaded and multi-threaded.

5.7.1.1 Single-Threaded Components

The queuing model for a single-threaded component consists of a waiting queue and a thread in execution. Let the length of queue be m and the demand (CPU) time for a request be T_d. Further assume the request is served using the first-come-first-served approach and the component is in the steady state, i.e., the rate of an arrival of a request is equal to the rate of the completion of a request. When a new request comes, assume there are m requests in the waiting queue and no request being serviced. The new request has to

wait for mT_d seconds before being served. Thus, the mean response time for the new request is:

$$T_r = mT_d + T_d = (m+1)\, T_d \quad (1)$$

For a system with a synchronous invocation, let, m_1, m_2 denote the queue size of component 1 and component 2 respectively and T_{d1}, T_{d2} denote the demand times at component 1 and component 2, respectively. Let T_{r1}, T_{r2} denote the response times for a request in component 1 and component 2, respectively. If $m_1=m_2$, a new request to the system has to wait for $m_1 (T_{d1}+T_{d2})$ seconds before being served and the service time for the new request is $T_{d1}+T_{d2}$. Hence, the mean response time for the new request is:

$$T_r = m_1 (T_{d1} + T_{d2}) + (T_{d1}+T_{d2}) = T_{r1}+ T_{r2} \quad (if\ m_1=m_2) \quad (2)$$

Based on equation (2), the response time of the composed system equals the sum of the response times of the two individual components when the two components interact synchronously.

For the composed system, let λ, λ_1, λ_2 denote the throughputs of the composed system, component 1, and component 2, respectively. Based on Little's law [36], we have:

$$m_1+1 = T_r\lambda = (T_{r1}+ T_{r2})\, \lambda = (\frac{m_1+1}{\lambda_1} + \frac{m_2+1}{\lambda_2})\lambda \quad (3)$$

$$\frac{1}{\lambda} = \frac{1}{\lambda_1} + \frac{m_2+1}{m_1+1}\frac{1}{\lambda_2} \quad (4)$$

$$\frac{1}{\lambda} = \frac{1}{\lambda_1} + \frac{1}{\lambda_2} \quad (If\ m_1=m_2) \quad (5)$$

Equation (5) indicates that the reciprocal of the throughput of the composed system equals the sum of the reciprocals of the throughputs of individual components when the two components communicate using synchronous mode.

Table 5.1 shows the experimental results of a synchronous two-way communication pattern with single threaded components as the participants. It can be seen that the proposed composition rules accurately predict the throughput and response time of a composed system of two single threaded components communicating using synchronous two-way invocation.

Table 5.1 Experiment Results of the Communication Pattern No. 2 with Single Threaded Components as the Participants

	Component 1	Component 2	Synchronous
Measured throughput (reqs/sec)	0.206	0.406	0.137
Predicted throughput (reqs/sec)			0.137
Demand time (sec)	4.846	2.463	
Measured response time (sec)	52.134	26.219	81.272
Predicted response time (sec)			78.353
Measured throughput (reqs/sec)	0.406	0.206	0.136
Predicted throughput (reqs/sec)			0.137
Demand time (sec)	2.464	4.845	
Measured response time (sec)	26.179	52.442	79.511
Predicted response time (sec)			78.621

5.7.1.2 Multi-Threaded Components

In this section, the components participating the synchronous two-way communication patterns are assumed to be multi-threaded.

The queuing model for a multi-threaded component with a thread pool consists of a waiting queue and a service center with a fixed number of threads. Let the queue length be *m* and the number of threads in the pool be *n*. Let T_d denote the CPU demand time of a request, assuming no parallel operations, such as I/O. When a new request arrives, let there be *m (m=k* n)* requests in the queue, and no request being served. Then the response time for the new request is the time spent at the component by the preceding *m* requests in the queue plus the time spent at the component (shared by possibly n active

threads) by the new request:

$$T_r = mT_d + nT_d = (m+n)T_d \quad (6)$$

Multi-threaded components can overlap the processing of requests in the synchronous communication pattern. When some threads are blocked in waiting for response, the other unblocked threads can continue to serve the incoming requests. Therefore, multi-threaded components in the synchronous two-way communication pattern can achieve a certain degree of asynchronous effect. To quantify the degree of the asynchronous effect of multithreaded components in synchronous communication patterns, the degree of asynchronous effect are proposed in this study and defined as:

$$\phi = \frac{T_{put} - T_{put}(totally\_syn)}{T_{put}(totally\_asyn) - T_{put}(totally\_syn)}, \quad (7)$$

where T_{put} denotes the throughput of the synchronous two-way communication pattern with multi-threaded components as the participants. T_{put}(totally_syn) denotes the throughput of the synchronous two-way communication pattern with no asynchronous effect and can be calculated based on equation (5). T_{put}(totally_asyn) denotes the throughput of the asynchronous two-way communication pattern and equals to the minimum throughput of the two participating components.

Table 5.2 and Table 5.3 show the experimental results of synchronous two-way communication patterns with multi-threaded components as the participants. Table 5.2 shows the experimental results of synchronous two-way communication pattern between two multithreaded (4 threads) components. The two cases of the communication pattern shown in Table 5.2 have asynchronous degree of 0.23 and 0.10 respectively. Table 5.3 shows the experimental results of synchronous two-way communication pattern between two multithreaded (10 threads) components. The two cases of the communication pattern shown in Table 5.3 have asynchronous degree of 0.75 and 0.41 respectively. It can be seen that the number of threads provided by a component and the invocation order

between components can affect the asynchronous degree of multi-threaded components in synchronous two-way communication pattern. It is challenging to obtain an empirical formula to calculate the asynchronous degree of a communication pattern based on the factors such as the number of threads provided by the participating components, the invocation order, and so on. With the calculated degree of asynchronous effect, the throughput and the response time of the communication pattern can be predicted.

Table 5.2 Experiment Results of the Communication Pattern No. 2 with Multi-Threaded (4 Threads) Components as the Participants

	Component 1	Component 2	Synchronous
Measured throughput (reqs/sec)	0.206	0.406	0.153
Predicted throughput (reqs/sec)			0.137
Demand time (sec)	4.851	2.463	
Measured response time (sec)	60.713	29.323	87.719
Predicted response time (sec)			90.036
Measured throughput (reqs/sec)	0.406	0.206	0.144
Predicted throughput (reqs/sec)			0.137
Demand time (sec)	2.466	4.845	
Measured response time (sec)	30.032	60.660	85.282
Predicted response time (sec)			90.692

Table 5.3 Experiment Results of the Communication Pattern No. 2 with Multi-Threaded (10 Threads) Components as the Participants

	Component 1	Component 2	Synchronous
Measured throughput (reqs/sec)	0.206	0.406	0.189
Predicted throughput (reqs/sec)			0.206
Demand time (sec)	4.846	2.463	
Measured response time (sec)	72.837	37.311	94.591
Predicted response time (sec)			97.468
Measured throughput (reqs/sec)	0.406	0.206	0.165
Predicted throughput (reqs/sec)			0.206
Demand time (sec)	2.464	4.845	
Measured response time (sec)	37.142	74.466	86.855
Predicted response time (sec)			86.785

5.7.2 The Composition Rules of the Communication Pattern No. 3

The communication pattern No. 3 is an asynchronous (callback) two-way invocation. Two types of components are considered: single threaded and multi-threaded.

5.7.2.1 Single Threaded Components

If component 1 is the bottleneck, i.e., two throughputs satisfy $\lambda_1<\lambda_2$, then in the steady state, the queue of component 1 is full and the queue of component 2 is empty. When a new request arrives, let there be *m* requests in the queue of component 1; no request being served by component 1; no request in the queue of component 2; and one request being served by component 2. The response time for the new request is the summation of the time it spends in component 1, including the time required by the *m* preceding requests in the queue and the service time needed in component 1, and the time spent in component 2. This time is given by the following equation:

$$T_r = m_1 T_{d1} + T_{d1} + T_{d2} = T_{r1} + T_{d2} \quad (8)$$

Similarly, if component 2 is the bottleneck, i.e., $\lambda_1>\lambda_2$, then at the steady state, the queue of component 1 is empty and the queue of component 2 is full. When a new request arrives, let there be no request at the queue of component 1; no request being served by component 1; m_2 requests in the queue of component 2; and no request being served by component 2. The response time of the new request is the summation of the time it spends in component 2 plus the time required by the preceding m_2 requests and the time spent in component 1 in phase-2 (a state when the thread in component 1 gets a reply from component 2) by the new request. This time is given by the following equation:

$$T_r = m_2 T_{d2} + T_{d2} + T_{d1\_phase2} = T_{r2} + T_{d1\_phase2} \quad (9)$$

Equations (8) and (9) indicate that the response time of the composed system is less than the sum of the response time of the two individual components when the two components interact asynchronously.

For the asynchronous communication pattern, the two components are fully decoupled. At steady state, the arrival rate of a new request to component 1 (λ_{11}) equals the departure rate of completed requests from component 1 (λ_{12}). The arrival rate of requests forwarded from component 1 to component 2 (λ_{21}) equals the request departure rate of component 2 (λ_{22}). It is obvious that $\lambda_{11} = \lambda_{21}$ and $\lambda_{12} = \lambda_{22}$. This indicates that the throughput of the composed system with callback asynchronous communication patterns equals to the minimum throughput of the two components.

$$\lambda = min\ (\lambda_1, \lambda_2) \tag{10}$$

Table 5.4 shows the experimental results of an asynchronous two-way communication pattern with single threaded components as the participants. It can be seen that the proposed rules (8)-(10) provide good accuracy in predicting the throughput and the response time of the composed system.

Table 5.4 Experiment Results of the Communication Pattern No. 3 with Single Threaded Components as the Participants

	Component 1	Component 2	Asynchronous
Measured throughput (reqs/sec)	0.206	0.406	0.206
Predicted throughput (reqs/sec)			0.206
Demand time (sec)	4.846	2.463	
Measured response time (sec)	52.134	26.219	56.838
Predicted response time (sec)			54.597
Measured throughput (reqs/sec)	0.406	0.206	0.204
Predicted throughput (reqs/sec)			0.206
Demand time (sec)	2.464	4.845	
Measured response time (sec)	26.179	52.442	53.522
Predicted response time (sec)			53.673

5.7.2.2 Multi-Threaded Components

For multi-threaded components, let T_{r1} and T_{r2} be the response times of component 1 and component 2 respectively, and T_{d1} and T_{d2} be the demand times of request at component 1 and component 2, and n_1 and n_2 be the numbers of threads in the pools of component 1 and component 2, respectively. Following the same analysis as in the case of single-threaded components, when component 1 is the bottleneck ($\lambda_1<\lambda_2$), the average system response time is:

$$T_r = T_{r1} + n_2 T_{d2} \tag{11}$$

If component 2 is the bottleneck ($\lambda_1>\lambda_2$), the system response time of a request can be written as:

$$T_r = T_{r2} + n_1 T_{d1\_phase2} \tag{12}$$

For the asynchronous communication pattern, the system throughput for multi-threaded components has the same composition rule as the single-threaded components, as indicated in equation (10).

Table 5.5 and Table 5.6 show the experimental results of an asynchronous two-way communication pattern with multi-threaded components as the participants. It can be seen from Table 5.5 and Table 5.6 that the predicted throughput and response time based on the proposed composition rules are close to the measured throughput and response time. The errors in predicting the throughput and the response time are within 6.2% and 8.1% respectively.

Table 5.5 Experiment Results of the Communication Pattern No. 3 with Multi-Threaded (4 Threads) Components as the Participants

	Component 1	Component 2	Asynchronous
Measured throughput (reqs/sec)	0.206	0.406	0.206
Predicted throughput (reqs/sec)			0.206
Demand time (sec)	4.851	2.463	
Measured response time (sec)	60.713	29.323	69.422
Predicted response time (sec)			70.566
Measured throughput (reqs/sec)	0.406	0.206	0.200
Predicted throughput (reqs/sec)			0.206
Demand time (sec)	2.466	4.845	
Measured response time (sec)	30.032	60.660	68.032
Predicted response time (sec)			65.591

Table 5.6 Experiment Results of the Communication Pattern No. 3 with Multi-Threaded (10 Threads) Components as the Participants

	Component 1	Component 2	Asynchronous
Measured throughput (reqs/sec)	0.206	0.406	0.206
Predicted throughput (reqs/sec)			0.206
Demand time (sec)	4.846	2.463	
Measured response time (sec)	72.837	37.311	90.179
Predicted response time (sec)			97.468
Measured throughput (reqs/sec)	0.406	0.206	0.194
Predicted throughput (reqs/sec)			0.206
Demand time (sec)	2.464	4.845	
Measured response time (sec)	37.142	74.466	87.570
Predicted response time (sec)			86.785

5.7.3 The Composition Rules of the Communication Pattern No. 11

The communication pattern No. 11 is the sequential composition of two synchronous two-way invocation-based communication patterns

The composition rule for communication pattern No. 11 can be constructed based on the basic communication patterns as: Component 1 ⊗ (Component 2 ⊗ Component 3), where ⊗ denotes the synchronous two-way communication pattern.

Table 5.7 shows the experimental results of the communication pattern No. 11 with multi-threaded (4 threads) components as the participants. It can be seen from Table 5.7 that the errors in predicting the throughput and the response time based on the composition rule are within 37.5% and 30.9% respectively. Because the composition does not consider the asynchronous degree of the communication pattern, the predicted results are not as good as expected.

Table 5.7 Experiment Results of Communication Pattern No. 11 with Multi-Threaded (4 Threads) Components as the Participants

	Component 1	Component 2	Component 3	Syn Syn
Measured throughput	0.406	0.306	0.206	0.152
Predicted system throughput				0.095
Demand time	2.464	3.271	4.845	
Measured response time	30.032	40.078	60.783	147.136
Predicted system response time				130.893
Measured throughput	0.206	0.306	0.406	0.126
Predicted system throughput				0.095
Demand time	4.846	3.271	2.464	
Measured response time	60.713	40.078	30.652	121.597
Predicted system response time				131.443
Measured throughput	0.306	0.206	0.406	0.120
Predicted system throughput				0.095
Demand time	3.271	4.846	2.464	
Measured response time	40.501	60.660	30.652	100.725
Predicted system response time				131.813

5.7.4 The Composition Rules of the Communication Pattern No. 12

The communication pattern No. 12 is the sequential composition of two asynchronous two-way invocation-based communication patterns.

The composition rule for communication pattern No. 12 can be constructed based on the basic communication patterns as: Component 1 ⊕ (Component 2 ⊕ Component 3), where ⊕ denotes the asynchronous two-way communication pattern.

Table 5.8 shows the experimental results of the communication pattern No. 12 with multi-threaded (4 threads) components as the participants. It can be seen from Table 5.8 that the predicted throughputs and response times based on the proposed composition rules approximate the measured throughputs and response times with a good accuracy. The errors in predicting the throughput and the response time are within 6.7% and 6.6% respectively.

Table 5.8 Experiment Results of Communication Pattern No. 12 with Multi-Threaded (4 Threads) Components as the Participants

	Component 1	Component 2	Component 3	Asyn Asyn
Measured throughput	0.406	0.306	0.206	0.193
Predicted system throughput				0.206
Demand time	2.464	3.271	4.845	
Measured response time	30.032	40.078	60.783	76.613
Predicted system response time				72.253
Measured throughput	0.206	0.306	0.406	0.205
Predicted system throughput				0.206
Demand time	4.846	3.271	2.464	
Measured response time	60.713	40.078	30.652	78.500
Predicted system response time				83.651
Measured throughput	0.306	0.206	0.406	0.197
Predicted system throughput				0.206
Demand time	3.271	4.846	2.464	
Measured response time	40.501	60.660	30.652	77.652
Predicted system response time				77.056

5.7.5 The Composition Rules of the Communication Pattern No. 13

The communication pattern No. 13 is the sequential composition of a synchronous two-way invocation and an asynchronous two-way invocation.

The composition rule of the communication pattern No. 13 can be constructed based on the basic communication pattern as: Component 1 ⊗ (Component 2 ⊕ Component 3), where ⊗ denotes the synchronous two-way communication pattern and ⊕ denotes the asynchronous two-way communication pattern.

Table 5.9 shows the experimental results of the communication pattern No. 13 with multi-threaded (4 threads) components as the participants. It can be seen that the throughputs and the response times predicted based on the proposed composition rules are close to the measured throughputs and response times. The error in predicting the throughput and the response time is within 11.5% and 11.7% respectively.

Table 5.9 Experiment Results of the Communication Pattern No. 13 with Multi-Threaded (4 Threads) Components as the Participants

	Component 1	Component 2	Component 3	Syn Asyn
Measured throughput	0.406	0.306	0.206	0.127
Predicted system throughput				0.137
Demand time	2.464	3.271	4.845	
Measured response time	30.032	40.078	60.783	110.273
Predicted system response time				97.357
Measured throughput	0.206	0.306	0.406	0.124
Predicted system throughput				0.123
Demand time	4.846	3.271	2.464	
Measured response time	60.713	40.078	30.652	105.031
Predicted system response time				110.645
Measured throughput	0.306	0.206	0.406	0.139
Predicted system throughput				0.123
Demand time	3.271	4.846	2.464	
Measured response time	40.501	60.660	30.652	112.017
Predicted system response time				111.015

5.7.6 The Composition Rules of the Communication Pattern No. 14

The communication pattern No. 14 is the sequential composition of an asynchronous two-way invocation-based communication pattern and a synchronous two-way invocation-based communication pattern.

The composition rule for the communication pattern No. 14 can be constructed based on the basic communication patterns as: Component 1 ⊕ (Component 2 ⊗ Component 3), where ⊕ denotes the asynchronous two-way communication pattern and ⊗ denotes the synchronous two-way communication pattern.

Table 5.10 shows the experimental results of the communication pattern No. 14 with multi-threaded (4 threads) components as the participants. Based on the results from Table 5.10, it can be seen that the predicted throughputs and response times using the proposed composition rules are close to the measured throughputs and response times. The errors in predicting the throughput and the response time are within 12.8% and 15.6% respectively.

Table 5.10 Experiment Results of the Communication Pattern No. 14 with Multi-Threaded (4 Threads) Components as the Participants

	Component 1	Component 2	Component 3	Asyn Syn
Measured throughput	0.406	0.306	0.206	0.141
Predicted system throughput				0.123
Demand time	2.464	3.271	4.845	
Measured response time	30.032	40.078	60.783	91.548
Predicted system response time				105.789
Measured throughput	0.206	0.306	0.406	0.190
Predicted system throughput				0.174
Demand time	4.846	3.271	2.464	
Measured response time	60.713	40.078	30.652	94.212
Predicted system response time				80.422
Measured throughput	0.306	0.206	0.406	0.156
Predicted system throughput				0.137
Demand time	3.271	4.846	2.464	
Measured response time	40.501	60.660	30.652	96.756
Predicted system response time				97.855

5.7.7 The Composition Rules of the Communication Pattern No. 17

The communication pattern No. 17 is the partial sequential composition of two synchronous two-way invocation-based communication patterns.

The composition rules for the communication pattern No. 17 can be constructed based on the basic communication patterns as: (Component 1 ⊗ Component 2) ⊗ Component 3, where ⊗ denotes the synchronous two-way communication pattern.

Table 5.11 shows the experimental results of the communication pattern No. 17 with multi-threaded (4 threads) components as the participants. The error in predicting the throughput and response time using the proposed composition rules are with in 26.3% and 41.4% respectively. Because the composition rules do not consider the asynchronous degree of a communication pattern, the predicted throughputs and response times are not as good as expected.

Table 5.11 Experiment Results of the Communication Pattern No. 17 with Multi-Threaded (4 Threads) Components as the Participants

	Component 1	Component 2	Component 3	Syn Syn
Measured throughput	0.406	0.306	0.206	0.114
Predicted system throughput				0.095
Demand time	2.464	3.271	4.845	
Measured response time	30.032	40.078	60.783	114.278
Predicted system response time				130.893
Measured throughput	0.206	0.306	0.406	0.111
Predicted system throughput				0.095
Demand time	4.846	3.271	2.464	
Measured response time	60.713	40.078	30.652	127.151
Predicted system response time				131.443
Measured throughput	0.306	0.206	0.406	0.129
Predicted system throughput				0.095
Demand time	3.271	4.846	2.464	
Measured response time	40.501	60.660	30.652	93.202
Predicted system response time				131.813

5.7.8 The Composition Rules of the Communication Pattern No. 18

The communication pattern No. 18 is the partial sequential composition of two asynchronous two-way invocation-based communication patterns.

The composition rule for the communication pattern No. 18 can be constructed based on the basic communication patterns as: (Component 1 ⊕ Component 2) ⊕ Component 3, where ⊕ denotes the asynchronous two-way communication pattern.

Table 5.12 shows the experimental results of the communication pattern No. 18 with multi-threaded (4 threads) components as the participants. It can be seen that the predicted throughputs and response times based on the proposed composition rules provide a good accuracy in approximating the measured throughputs and response times. The errors in predicting the throughput and the response time are within 11.4% and 12.9% respectively.

Table 5.12 Experiment Results of the Communication Pattern No. 18 with Multi-Threaded (4 Threads) Components as the Participants

	Component 1	Component 2	Component 3	Asyn Asyn
Measured throughput	0.406	0.306	0.206	0.185
Predicted system throughput				0.206
Demand time	2.464	3.271	4.845	
Measured response time	30.032	40.078	60.783	80.057
Predicted system response time				72.253
Measured throughput	0.206	0.306	0.406	0.205
Predicted system throughput				0.206
Demand time	4.846	3.271	2.464	
Measured response time	60.713	40.078	30.652	74.061
Predicted system response time				83.651
Measured throughput	0.306	0.206	0.406	0.191
Predicted system throughput				0.206
Demand time	3.271	4.846	2.464	
Measured response time	40.501	60.660	30.652	69.651
Predicted system response time				77.056

5.7.9 The Composition Rules of the Communication Pattern No. 19

The communication pattern No. 19 is the partial sequential composition of a synchronous two-way invocation-based communication pattern and an asynchronous two-way invocation-based communication pattern.

The composition rule for the communication pattern No. 19 can be constructed based on the basic communication patterns as: (Component 1 ⊗ Component 2) ⊕ Component 3, where ⊗ denotes the synchronous two-way communication pattern and ⊕ denotes the asynchronous two-way communication pattern.

Table 5.13 shows the experimental results of the communication pattern No. 19 with multi-threaded (4 threads) components as the participants. It can be seen from Table 5.13 that the predicted throughputs and response times based on the proposed composition rules are close to the measured throughputs and response times for most of the cases. The errors in predicting the throughput and the response time are within 18.5% and 16.4%.

Table 5.13 Experiment Results of the Communication Pattern No.19 with Multi-Threaded (4 Threads) Components as the Participants

	Component 1	Component 2	Component 3	Syn Asyn
Measured throughput	0.406	0.306	0.206	0.190
Predicted system throughput				0.174
Demand time	2.464	3.271	4.845	
Measured response time	30.032	40.078	60.783	92.967
Predicted system response time				89.490
Measured throughput	0.206	0.306	0.406	0.146
Predicted system throughput				0.123
Demand time	4.846	3.271	2.464	
Measured response time	60.713	40.078	30.652	115.816
Predicted system response time				110.647
Measured throughput	0.306	0.206	0.406	0.151
Predicted system throughput				0.123
Demand time	3.271	4.846	2.464	
Measured response time	40.501	60.660	30.652	132.805
Predicted system response time				111.017

5.7.10 The Composition Rules of the Communication Pattern No. 20

The communication pattern No. 20 is the partial sequential composition of an asynchronous two-way invocation-based communication pattern and a synchronous two-way invocation-based communication pattern.

The composition rule for the communication pattern No. 20 can be constructed based on the basic communication patterns as: Component 2 ⊕ (Component 1 ⊗ Component 3) for throughput, Component 1 ⊗ Component 3 for response time, where ⊕ denotes the asynchronous two-way communication pattern and ⊗ denotes the synchronous two-way communication pattern.

Table 5.13 shows the experimental results of the communication pattern No. 20 with multi-threaded (4 threads) components as the participants. It can be seen from Table 5.13 that the predicted throughput and response time approximate the measured throughput and response time with errors within 22.1% and 8.1% respectively.

Table 5.14 Experiment Results of the Communication Pattern No. 20 with Multi-Threaded (4 Threads) Components as the Participants

	Component 1	Component 2	Component 3	Asyn Syn
Measured throughput	0.406	0.306	0.206	0.176
Predicted system throughput				0.137
Demand time	2.464	3.271	4.845	
Measured response time	30.032	40.078	60.783	90.174
Predicted system response time				90.815
Measured throughput	0.206	0.306	0.406	0.171
Predicted system throughput				0.137
Demand time	4.846	3.271	2.464	
Measured response time	60.713	40.078	30.652	84.516
Predicted system response time				91.365
Measured throughput	0.306	0.206	0.406	0.192
Predicted system throughput				0.174
Demand time	3.271	4.846	2.464	
Measured response time	40.501	60.660	30.652	70.359
Predicted system response time				71.153

5.8 Summary

In this chapter, the effect of the inter-component communication patterns on the system composition and decomposition are studied. The major communication patterns are identified and the composition rules of the throughput and the turn-around time for some of these communication patterns are proposed. The communication patterns and the invocation order of components affect the throughput and the response time of a composed system. The proposed composition rules can provide a rough prediction of the throughput and the response time of a composed system. In the next chapter, the network effect on the system composition and decomposition are investigated.

6. EFFECT OF NETWORKS ON SYSTEM COMPOSITION AND DECOMPOSITION

6.1 Introduction

Today's distributed systems are highly dependent on networking and information infrastructure. The network plays an important role in distributed systems, especially from application domains, such as data mining, e-commerce, and multimedia, which are bandwidth hungry, time sensitive, and mission critical. In such applications, the network could be the bottleneck of the system. In distributed real-time systems, due to the tight real-time constraint, the network performance need to be highly predictable in order to meet the hard/software time deadline.

The proposed composition and decomposition rules did not incorporate the network components. In this chapter, a preliminary approach is proposed to consider the network effect in system composition and decomposition.

6.2 Network Component and Its QoS Parameters

In a distributed system, two distributed components are connected by a network component as shown in the Figure 6.1. In general, a network component consists of the following subcomponents along the communication path: the network stack at the end hosts, the switches/routers, and the links.

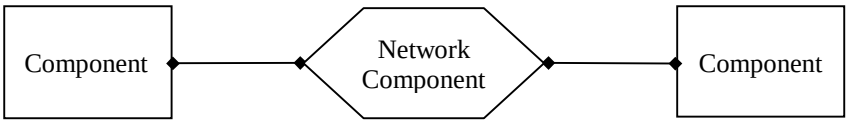

Figure 6.1 The Network Component in a Distributed System

Different parameters can be used to gauge the performance of networks. The well-known network QoS parameters consist of bandwidth, end-to-end delay, packet loss, and jitter.

1) End-to-end delay (ms):

End-to-end delay is a function of the number of hops a packet makes. The network end-to end delay consists of the following components: end system delay and network delay. The end system delay includes: processing delay at the sending end (fixed), packetization delay at the sending end (fixed, negligible), depacketization at the receiving end (fixed, negligible), processing delay at the receiving end (fixed), and jitter buffer delay at the receiving end (fixed). Network delay includes propagation delay at switch/router (fixed, = distance / speed of light), queuing delay at switch/router (variable), and transmit delay at switch/router (fixed, = packet size / bandwidth). The network end-to-end delay can be written as:

$$D(\text{network}) = D_{\text{sender}}(\text{processing}) + D(\text{queuing}) + D(\text{transmit}) + D(\text{propagation}) + D(\text{jitter buffer}) + D_{\text{receiver}}(\text{processing}) \quad (6.1)$$

For enterprise terrestrial networks, queuing and transmission delays are often the dominant components of the network delay. In satellite networks, the propagation delay can dominate.

2) Packet loss (%):

There are two reasons for the packet loss: 1) Queue overflows in routers or end-user machines, and 2) Bit errors. The probability of bit errors is very low on most networks. Therefore it is assumed that loss is induced by congestion rather than by bit errors. Buffer overflow can happen on a congested link or at the network interface of a workstation. When too many packets are simultaneously sent to a router, it simply discards some packets. The network end-to-end probability of packet loss can be determined based on the nodes in the end-to-end path [38]:

$$(1 - PL_{\text{network}}) = (1 - PL_1) *\ldots*(1 - PL_i)* (1 - PL_n), \quad (6.2)$$

where $PL_{network}$ denotes the packet loss of the network, PL_i (i=1, 2, …, n) denotes the packet loss at the node i.

3) Jitter (ms):

If the network is congested, arrival times for the packets will vary. The easiest way to remove jitter is to hold the packets in a buffer, the "jitter buffer", until the slowest packet arrives and then to transmit them in the correct sequence with an equal inter-packet timing. This method solves the problem of jitter but also introduces additional delays into the end-to-end connection. It is therefore critical to perform a balancing act between jitter and delay. The end-to-end network jitter can be determined based on the nodes in the end-to-end path [39]:

$$J_{network} = \sqrt{\frac{J_1^{\,2} + \ldots + J_i^{\,2} + \ldots J_n^{\,2}}{n}}\ , \tag{6.3}$$

where $J_{network}$ denotes the network end-to-end jitter, J_i (i=1,2,…,n) denotes the jitter at the node i.

6.3 Mapping Application QoS to Network QoS

The system QoS mapping is to translate the application level QoS requirements into constraints at the network level. The application level QoS parameters are meaningful to applications, while the network level QoS parameters are meaningful to the network. The question is how to map the application level QoS to the network level QoS, or vice versa. For example, in a streaming video application, the frame rate and the resolution are important QoS parameters at the application level, which are determined by the bandwidth at the network level. How to map between them is a challenging research problem.

6.4 Class of Services

The current network infrastructure can provide different classes of services, such as best effort, differentiated service, and guaranteed service. On the other hand, the applications from different domains impose different service requirements on the network.

6.4.1 QoS Levels Provided by Networks

Current network QoS services can be divided into three levels: best effort service, differentiated service, and guaranteed service. The best effort model does not guarantee a traffic delivery. A differentiated service groups the traffic into classes and a relative service priority exists among these classes. A differentiated service can provide a qualitative or statistical guaranteed service. A guaranteed service allocates the network resources to ensure specific service requirements. A deterministic bound can be specified for QoS with guaranteed service.

6.4.2 Network QoS Requirements Based on Class of Services

The network QoS parameters can be specified based on the class of service, as shown in Table 6.1 [37]. Four different QoS classes are defined and basic characteristics for each class are described. Each class of service corresponds to a certain range of QoS values and different applications can be mapped to different classes of services. In Table 6.1, IPTD denotes IP packet transfer delay; IPDV denotes IP packet delay variation; IPLR denotes IP packet loss ratio; IPER denotes IP packet error ratio; SPR denotes spurious IP packet rate; “U” means “unspecified” or “unbound”.

Table 6.1 Provisional IP QoS Class Definitions and Network Performance Objectives

			QoS Classes			
	Nature of the network performance objective	Default objectives	Class 0	Class 1 (Interactive)	Class 2 (Non-Interactive)	Class 3 (U class)
IPTD	Upper bound on the mean IPTD	No default	150 ms	400ms	1 sec	U
IPDV	Upper bound on the 1-10^{-3} quantile of IPTD minus the minimum IPTD	No default	50 ms	50ms	1 sec	U
IPLR	Upper bound on the packet loss ratio	No default	$1*10^{-3}$	$1*10^{-3}$	$1*10^{-3}$	U
IPER	Upper bound	$1*10^{-4}$	Default	Default	Default	U
SPR	Upper Bound	Default TBD	Default	Default	default	U

6.4.3 Network QoS Requirements Based on Application Domains

Network QoS requirements can also be specified based on the application domains. Each application domain requires a different level of the network QoS services. Table 6.2 shows the expected network QoS for conversational/real-time services. Table 6.3 shows the expected network QoS for streaming service. Table 6.4 shows the expected network QoS for interactive service [37]. In Table 6.2, 6.3 and 6.4, FER means frame erasure rates.

Table 6.2 End-User Performance Expectations - Conversational/Real-time Services

Medium	Application	Degree of symmetry	Data rate	Key performance parameters and target values		
				One-way Delay	Delay Variation	Information loss
Audio	Conversation at Narrow-band speech	Two-way	[4-13] kbit/s	<150 msec preferred <400 msec limit	< 1 msec	< 3% FER
Audio	Conversation at Wideband speech	Two-way	[4-13] kbit/s [10-64] kbit/s	<150 msec preferred <400 msec limit	< 1 msec	< 3% FER
Video	Videophone	Two-way	[32-384] kbit/s	< 150 msec preferred <400 msec limit Lip-synch: < 100 msec		< 1% FER
Data	Telemetry - two-way control	Two-way	[<28.8] kbit/s	< 250 msec	N.A	Zero
Data	Interactive games	Two-way	[< 1] KB	< 250 msec	N.A	Zero
Data	Telnet	Two-way (asymmetric)	[< 1] KB	< 250 msec	N.A	Zero

Table 6.3 End-User Performance Expectations - Streaming Services

Medium	Application	Degree of symmetry	Data rate	Key performance parameters and target values		
				One-way Delay	Delay Variation	Information loss
Audio	High quality streaming audio	Primarily one-way	[32-128] kbit/s	< 10 sec	< 1 msec	< 1% FER
Video	One-way	One-way	[32-384] kbit/s	< 10 sec		< 1% FER
Data	Bulk data transfer/retrieval	Primarily one-way		< 10 sec	N.A	Zero
Data	Still image	One-way		< 10 sec	N.A	Zero
Data	Telemetry - monitoring	One-way	[<28.8] kbit/s	< 10 sec	N.A	Zero

Table 6.4 End-User Performance Expectations - Interactive Services

Medium	Application	Degree of symmetry	Data rate	Key performance parameters and target values		
				One-way Delay	Delay Variation	Information loss
Audio	Voice messaging	Primarily one-way	[4-13] kbit/s	< 1 sec for playback < 2 sec for record	< 1 msec	< 3% FER
Data	Web browsing - HTML	Primarily one-way		< 4 sec/page	N.A	Zero
Data	Transaction services - high priority e.g. e-commerce, ATM	Two-way		< 4 sec	N.A	Zero
Data	E-mail (server access)	Primarily One-way		< 4 sec	N.A	Zero

6.5 Specification of Network Component

To incorporate the network effect into the UniFrame approach, the network component is considered as a special component in a distributed system. It is the connector between two distributed components in the system. The following is the preliminary format of the specification of the network component:

- *Component type: connector*
- *Application domain:*
- *Functionality: transmit media/bulk data*
- *Service level: best effort/differentiated server/guaranteed service*
- *QoS*
    - *Bandwidth (KB/s):*
    - *Delay (ms):*
    - *Jitter (ms):*
    - *Packet loss (%):*

6.6 Incorporation of Network Component into System Composition and Decomposition

In system composition and decomposition, the network component between two distributed components cannot be determined until the two distributed components are determined. In the system decomposition, the properties of the network component are initially assumed based on the application domains. When the system components are determined after the search and selection process, the network component between each two distributed components is determined. System composition then validates the system as a whole including network components.

6.7 Summary

The network component is an essential part in component-based distributed systems. The difficulties to incorporate the network component into the system composition and decomposition approach are: 1) a mapping from the application QoS to network QoS is needed, which is non-trivial, 2) the network component sometimes is not under control of the system integrators. A preliminary study on how to incorporate the network component into the composition and decomposition approach is addressed in this chapter. In the next chapter, effect of the system execution environment on system composition and decomposition is studied.

7. EFFECT OF SYSTEM EXECUTION ENVIRONMENT ON SYSTEM COMPOSITION AND DECOMPOSITION

7.1 Introduction

A component-based distributed system executes in an environment, which has the following meanings:

- The hardware platforms: A component-based distributed system executes on a hardware platform such as handheld, desktop.
- The system resources: These resources perform the computation and communication.
- The security policy: The underlying execution environment may have its own security policy.
- The location of device: A component-based distributed system may sense the location of its components and provide the user mobility.
- Entities/actors a system interacts with: A component-based distributed system may need to collaborate with outside entities/actors to achieve a certain task.

The execution environment of component-based systems may experience a major change, for example, the change of hardware platform from desktop to handheld. It may undergo fine-grained run-time variations, for example, drops of network bandwidth, decreasing battery power, change of access control for security policy, and change of location when the user moves from one place to another place. The following describes three applications where the execution environment is important.

In distributed mobile computing systems, the changes of their execution environment are particularly substantially and frequently because of the intrinsic characteristics that make their execution environment susceptible to great variations such as high bandwidth variability, variable error rates, eventual disconnections, hardware and software heterogeneity, variable resource availability and user mobility.

In distributed mobile agent-based systems, the mobile agent can move from one execution environment to another execution environment. The new execution environment may be totally different in terms of the security policy, availability of resources.

The grid applications are characterized by the dynamic nature of their execution environment: the joining and leaving of resources. Applications that execute on computational grids can be susceptible to large changes in performance due to the inherently dynamic nature of grid resources.

In [27], the effect of the environment on the QoS of software components is studied. The studied environment variables include the CPU speed, the size of memory, and the execution priority of the components. It stated that the fact that the environment variables can affect the QoS of a software component implies that any QoS associated with a software component would not necessarily hold true in foreign environments. Hence, it becomes critical to account for the effect of the execution environment on the QoS of software components.

In extending the work in [27], the effect of the execution environment of component-based distributed systems on the QoS of the entire integrated system is investigated in this chapter.

7.2 The Hardware Platforms of the Execution Environment

Driven by the system evolution and software reuse, a component-based distributed system may be deployed across different hardware platforms: such as set top boxes, PDAs, cell phones, Web pads, and desktops. Different hardware platforms have different capabilities in terms of the computation power, the available memory, the bandwidth of connections and the power.

7.3 The Security Policy of the Execution Environment

The underlying execution environment of a component-based distributed system has its own security consideration and settings. Different execution environments may have different levels of security. For example, the Windows 9X and Windows NT/2000 have different security policies and Windows 9X is more vulnerable than Windows NT/2000. The change of the security policy of the underlying execution environment can affect the security of the application running on it. In another words, the security of a component-based distributed system is dependent on the security of the underlying execution environment.

7.4 The User Mobility of the Execution Environment

Today mobile devices are an integrated part of the execution environment of many distributed applications. This fact has brought up a new application domain called mobile computing, which offers information access anywhere and anytime but also introduces new problems special to this application area: the execution environment is susceptible to great variations as hand-held devices move about in the physical world, such as frequent disconnections and low bandwidth. The available hardware resources such as bandwidth, memory, and battery life can fluctuate quickly on such devices.

7.5 The System Resources of the Execution Environment

A system resource is an entity that is shared by different applications and might cause contention. System resources are classified into software/hardware resources, time shared/space shared resources, and active/passive resources.

The hardware resources include the CPU time, the memory, the I/O, the network I/O, and the power. The software resources include blocks, shared pools, locks, semaphores, buffers, and file descriptors. The software resources also consume hardware resources.

Time-shared resources can only be used by one client exclusively at each instant. An example of time-shared resources is the CPU time. Space-shared resources, in contrast, are structured aggregations of a set of identical elements. They are subject to sharing mainly on the basis of the simultaneous access to possibly overlapping subsets. The physical memory is an example of space-shared resources.

Active resources represent an entity that performs the work to accomplish a job. The essential attribute of the entity is the rate at which it does the work. CPU is an example of an active resource. Passive resources represent something needed to do a job, but do not accomplish the work. The essential attribute of the entity is capacity or amount of the resource. Physical memory is an example of a passive resource.

Different standalone components/applications compete for the available system resources in a machine. A resource contention occurs when the demand exceeds or become close to the capacity of that resource. The response time of a request can be broken into three parts. The first part is the time spent to access software resources that are mutually exclusive, such as a critical section. The second part is the time spent in waiting to use the hardware resources. The third part is the time the hardware resources spend in serving the request. In the case of resource contention, the first two parts of the response time of a request would increase. Excessive resource contention can give rise to an increased response time.

The performance of an application is dominated by the effects of the resource contention. Since the response time equals the service time plus the wait time, the performance can be improved by reducing either of these. The service time for a task may stay the same, but the wait time increases as contention increases. Resources such as CPUs, memory, I/O capacity, and network bandwidth are key to reducing the service time. Adding extra resources make a higher throughput possible and facilitate a faster response time.

7.6 Environment Failure

A distributed system cannot work properly if its underlying environment encounters a failure. Analysis of the effect of the environment failure on a distributed application is important in domains such as real-time and embedded domains.

The environment failure can be the failure of the system resources or failure of the outside entities/actors. The failures of system resources include processor failure, memory failure, disk failure and link failure. The failures of the outside entities/actors include the failure of the external entities and invalid behavior of external actors.

Each constituent of the execution environment of a distributed system could be a potential point of failure. To analyze the system failure model, the failure probability and failure duration of each environment constituent needs to be known. To calculate the system reliability, the environment constituents become nodes in a system fault tree

7.7 Environment-Sensitive Component

The response of a component may or may not be sensitive to the change of the system execution environment. The environment sensitive components are those that can be significantly affected by the change of the environment. For example, a component running on a mobile device is sensitive to the user movement in the environment. The network bound streaming video component is sensitive to the variation of the network bandwidth. The environment sensitive components need more precautions during system integration. Some environment-sensitive components are described in the following subsections.

7.7.1 Components in Resource-Constrained Environment

In the area of real-time/embedded systems, such as smart cards, hand-held devices, Internet appliance, set top boxes, sensor devices, cell phones, Web pads, etc., the execution environment of the system is resource-restricted due to their inherent scarcity

of computing power, memory space and hostile conditions, and the constraints on the battery capacity, the size and cost of the device, and the bandwidth consumptions.

The components in a resource-constrained environment are sensitive to the change of the environment. For example, mobile devices may experience network disconnection as the user moves from one place to another place. In real-time and embedded systems, there is a tight constraint on the time taken to respond to the external interactions. Any resource contentions can cause unpredictable delay, which is undesirable.

7.7.2 Computation Bound Components

The performance of a computation-bound component is mainly restricted by the available computational resources. CPU-bound and memory-bound are two examples of computation-bound components.

A CPU-bound component performs CPU-intensive computation and uses up most of the available processor cycles to perform a computation. For example, components sorting an array, multiplying matrices, or doing cryptographic computation are CPU-bound.

A memory-bounded component performs memory-intensive computation, making a significant number of memory accesses. Examples are a component applies a data mining algorithm on large dataset, or a component does video or audio compression.

7.7.3 I/O Bound Components

An I/O bound component is normally classified as a disk I/O bound or a network I/O bound component.

Disk I/O bound components performs a lot of reading and writing to the hard disk. The performance of the component is significantly limited by the speed of the disk. For example, components that perform OLTP (On-Line Transaction Processing) or video processing are disk I/O bound.

The performance of a network-bound component is mainly restricted by the available network resource, the network bandwidth. For example, components perform video streaming are network I/O bound.

7.8 Analysis of the Environment Effect on System Non-Functional Properties

The variation of the execution environment of distributed systems has an effect on the properties of the individual components. For example, as the computation resource decreases, the computation of a component in that environment slows down. To analyze the overall effect of the environment at the system level, the following procedures are proposed:

1) Determine the nature of the execution environment. Applications running on different execution environments may experience different kind of environment changes. This includes the types of the environment changes, the significance of the environment changes, and the frequency of the environment changes.

2) Identify the environment sensitive components. The variation of the environment of a distributed application may affect the non-functional properties of each component in the system, but the significance of this effect can differ greatly from one component to another. With this in mind, it is possible to analyze the environment's effect on the system non-functional properties by taking account of the environment sensitive components. The environment sensitive components can be identified based on its role in the system. For example, some components are computation-oriented, some components are communication-oriented, some components are mobile-oriented and some components are authentication-oriented. A component in a certain role may be sensitive to a certain change of the execution environment.

3) Based on the information collected in procedures 1) and 2), evaluate the environment effect on the system non-functional properties and provide recommendations.

7.9 Summary

In summary, the execution environment of a component-based distributed system undergoes various changes during its lifetime. These environment changes can affect the system non-functional properties of the distributed application, such as performance, reliability, security, and so on. In this chapter, a preliminary approach is proposed to consider the effect of the execution environment of a distributed system on system composition and decomposition. This approach is based on the nature of the execution environment and the environment sensitive components.

8. CONCLUSIONS AND FUTURE WORKS

This thesis addresses the composition and decomposition of non-functional properties in component-based distributed systems. The major conclusions are made in the section 8.1 and some future works are indicated in the section 8.2.

8.1 Conclusions

A sound composition and decomposition model of non-functional properties is important for the success of component-based software development. Based on this study, the following conclusion can be made:

1) Non-functional properties are abstract, informal and interact with functional properties. Even though a rigorous mathematical model for composition and decomposition is crucially needed, the understanding of the composition and decomposition mechanisms of the non-functional properties in component-based development is still in its early stage.

2) Due to the diversity of the non-functional properties and the distinct features of individual non-functional properties, it is difficult to create a general composition and decomposition rule for those properties. The composition and decomposition rules are specific to individual properties.

3) A non-functional property may be applicable to different domains. The domain independent composition and decomposition rules can be reused in different domains, but they are normally weak. The domain specific composition and decomposition rules are strong, but specific to an individual domain. There is a tradeoff between reusability and accuracy.

4) Different inter-component communication patterns are supported by current component models. A component may use different communication patterns to interact with different components or the same component at different times. The inter-component communication patterns can affect the composition and decomposition of non-functional properties. There is a need of the composition and decomposition rules of non-functional properties for individual communication patterns.

5) The network is an essential part of a distributed system. It is needed to incorporate the network into the composition and decomposition of non-functional properties.

6) In order to develop a dependable software system, the system execution environment has to be considered during the system development. Due to its dynamic characteristics, capturing the environment's effects on the system properties is difficult.

The contributions of this study are:

1) Propose the composition and decomposition rules of non-functional properties and divide them into domain independent rules and domain specific rules.

2) Identify the inter-component communication patterns and propose the composition rules of throughput and response time for some of these communication patterns, and validate these rules via experimental data.

3) Consider the network as a special component in UniFrame approach. The specification of network component, the incorporation of network component into the UniFrame approach is preliminary studied.

4) Identify the key factors of system execution environment. Propose the approach to address the effect of system execution environment on non-functional properties. The approach focuses on the nature of the execution environment and the environment sensitive analysis.

8.2 Future Works

The future works of this study are:

1) Formalize the composition and decomposition rules. The formal approach is rigorous, unambiguous and easy to be automated by use of tools.

2) Non-function properties are constraints on functional properties. Non-functional properties and functional properties cannot be handled separately. In this study, only the non-functional properties are addressed. It is important to extend this approach to address not only non-functional properties, but also the corresponding functional properties.

3) This approach considers only the product quality. It is necessary to define the process quality of the UniFrame approach and incorporate it into the composition and decomposition of non-function properties.

4) Further investigate the integration of network component with the system composition and decomposition approach. This includes: map the application QoS to network QoS and solve the problems such as the network component is unknown in system decomposition and the network component maybe not under control in system composition.

5) Experiment with the system execution environment and quantify its effect on system composition and decomposition.

REFERENCES

[1] P. Brereton and D. Budgen. “Component-Based Systems: A Classification of Issues,” IEEE Computer, Vol. 33, No. 11, November 2000, pp. 54-62.

[2] I. Sommerville. “Software Engineering,” Addison-Wesley, 2000, ISBN: 020139815X.

[3] J. A. Stafford. “PACC – Bridging the Gap Between Software Architecture and Software Components,” http://www-2.cs.cmu.edu/afs/cs/academic/class/17655-s02/www/lectures/25.c-b.systems.pdf.

[4] G. T. Heineman and W. T. Councill. “Component-Based Software Engineering: Putting the Pieces Together,” Addison Wesley, 2001, ISBN: 0201704854.

[5] Java™ 2 Platform Enterprise Edition: http://java.sun.com/j2ee.

[6] CORBA, OMG: http://www.corba.org.

[7] .NET, http://www.microsoft.com/net/.

[8] Web Services, http://www.w3.org/2002/ws/.

[9] D. Garlan and D. Perry. “Introduction to the Special Issue on Software Architecture. Guest Editorial,” IEEE Transaction on Software Engineering, Vol. 21, No. 4, April 1995, pp. 269-274.

[10] Inter-Agency Working Group on Information Technology Research and Development. “High Confidence Software and Systems Research Needs,” January 2001.

[11] UML, http://www.omg.org.

[12] N. S. Rosa, P. R. R. Cunha and G. R. R. Justo. “Process ^{NFL}: A Language for Describing Non-Functional Properties,” Proceedings of the 35th Annual Hawaii International Conference on System Sciences, 2002.

[13] M. Abadi and L. Lamport. “Composing Specifications,” ACM Transactions on Programming Languages and Systems, Vol. 15, No. 1, January 1993, pp. 73-132.

[14] K. M. Chandy and B. Sanders. “Reasoning About Program Composition,” Technical Report 96-035, Department of Computer and Information Science and Engineering, University of Florida, 1996.

[15] M. Charpentier and K. M. Chandy. “Examples of Program Composition Illustrating the Use of Universal Properties,” International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA’99), Springer-Verlag Lecture Notes in Computer Science, Vol. 1586, April 1999, pp. 1215-1227.

[16] M. Charpentier and K. M. Chandy. “Towards a Compositional Approach to the Design and Verification of Distributed Systems,” World Congress on Formal Methods in the Development of Computing Systems (FM’99), Springer-Verlag Lecture Notes in Computer Science, Vol. 1708, September 1999, pp. 570-589.

[17] M. Charpentier and K. M. Chandy. “Theorem About Composition,” International Conference on Mathematics of Program Construction, Springer-Verlag Lecture Notes in Computer Science, Vol. 1837, July 2000, pp. 167-186.

[18] J. Stafford and K. Wallnau. “Predicting Feature Interactions in Component-based Systems,” In proceedings of the Workshop on Feather Interaction of Composed Systems, in Conjunction with the 15th European Conference on Object-Oriented Programming, Budapest, Hungary, June 2001.

[19] R. Kazman, G. Abowd, L. Bass, and P. Clements. “Scenario-Based Analysis of Software Architecture,” IEEE Software, Vol. 13, No. 6, November 1996, pp. 47-55.

[20] M. Klein and R. Kazman. “Attribute-Based Architectural Styles,” CMU/SEI-99-TR-022, 1999.

[21] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. “The Architecture Tradeoff Analysis Method,” Fourth International Conference on Engineering Complex Computer Systems, August 1998, pp. 68-78.

[22] N. S. Rosa, G. R. R. Justo and P. R. F. Cunha. “A Framework for Building Non-Functional Software Architectures,” 16th ACM Symposium on Applied Computing, March 2001, pp. 141-147.

[23] C. U. Smith and L. G. Williams. “Performance and Scalability of Distributed Software Architectures: An SPE Approach,” Parallel and Distributed Computing Practices, Vol. 3, No. 4, 2002.

[24] L. G. Williams, and C. U. Smith. “PASA: A Method for the Performance Assessment of Software Architectures,” Proc. 3rd Int. Workshop on Software and Performance, Rome, 2002.

[25] D. Petriu and M. Woodside. “Software Performance Models from System Scenarios in Use Case Maps,” Proc. 12 Int. Conf. on Modeling Tools and Techniques for Computer and Communication System Performance Evaluation (Performance TOOLS 2002), London, April 2002.

[26] R. Raje, M. Auguston, B. Bryant, A. Olson, and C. Burt. “A Unified Approach for the Integration of Distributed Heterogeneous Software Components,” Proceedings of the 2001 Monterey Workshop (Sponsored by DARPA, ONR, ARO and AFOSR), Monterey, California, 2001, pp. 109-119.

[27] G. J. Brahnmath. “The Uniframe Quality of Service Framework,” M. S. Thesis, Department of Computer & Information Science, Indiana University Purdue University Indianapolis, December 2002.

[28] B. R. Bryant, B. S. Lee. “Two-Level Grammar as an Object-Oriented Requirements Specification Language,” Proceedings (on CD-ROM -- 10 Pages) of the 35th Hawaii International Conference on System Sciences, Hawaii, 2002.

[29] N. N. Siram. “An Architecture for Discovery of Heterogeneous Software Components,” M. S. Thesis, Department of Computer & Information Science, Indiana University Purdue University Indianapolis, March, 2002.

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

[31] M. Auguston. “A Program Behavior Model Based On Event Grammar and It’s Application for Debugging Automation,” In the Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging, AADEBUG'95, Saint-Malo, France, May 1995, pp. 277-291.

[32] Z. Huang, R. Raje, A. Olson, B. Bryant, M. Auguston, C. Burt, and C. Sun. “System-Level Generative Programming of Unified Approach Based on UMM for the Integration of Distributed Software Components,” Proceedings of the IEEE Fifth International Conference on Algorithms and Architectures for Parallel Processing, Beijing, China, October 2002, pp. 136-142.

[33] E. Lycklama. “Detecting, Diagnosing, and Overcoming the Five Most Common J2EE Application Performance Obstacles,” Web Services Edge, June 2002, http://www.sitraka.com/software/media/WebServicesEdge_June2002.ppt.

[34] N. G. Pryce. "Component Interaction in Distributed Systems," PhD dissertation, Department of Computing, Imperial College of Science, Technology, and Medicine, University of London, January 2000.

[35] V. S. W. Eide, F. Eliassen, and O. Lysne. "Supporting Distributed Processing of Time-Based Media Streams," The 3rd International Symposium on Distributed Objects and Applications (DOA'01), September 2001, pp. 281-288.

[36] D. Gross, and C. M. Haris. "Fundamentals of Queuing Theory," Wiley-Interscience, 1998, ISBN: 0471170836.

[37] The Telecommunication Technology Committee (TTC), Working Group 6-5 SWG4, "The Investigation Report of Major Standardization Activities About QoS of IP Services, Revision1," http://www.aptsec.org/astap/meetings/forum2001/fifth-astap/documents/ASTAP01-FR05-PL-24_FWA_PR-attachment3.doc, October 2001.

[38] I. Grgic and C. Hatch. "Inter-Operator IP QoS Framework - ToIP and UMTS Case Studies, Review of Existing IP QoS Activities and Extension of P1008 Findings," http://www.eurescom.de/~pub/deliverables/documents/P1100-series/P1103/TI1/p1103-ti1.pdf, August 2001.

[39] G. V. Bochmann and A. Hafid. "Some Principles for Quality of Service Management. Distributed Systems Engineering," Special Issue on Quality of Service, Vol. 4, No. 1, March 1997, pp. 16-27.