SYNCHRONIZATION AND QUALITY OF SERVICE SPECIFICATIONS AND MATCHING OF SOFTWARE COMPONENTS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Anjali Kumari

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

December 2004

1. INTRODUCTION

A Distributed Computing System (DCS) is a collection of autonomous computers that are networked over a Local Area Network (LAN) and cooperate in order to accomplish a given service or functionality, thereby sharing software and hardware resources. It brings together resources of different makes, models and performance levels and coordinates them. With the increase in the use of DCSs, there has been an increase in the demand for enhanced usability, robustness, reliability, flexibility, adaptability, and simpler installation and deployment in these systems. Consequently, a recent shift has been observed in the distributed computing paradigm from the traditional method of building software systems to Component-based Software Development (CBSD). CBSD is defined as the process of building large software systems by integrating independently developed and deployed software components, also know as commercial-off-the-shelf (COTS) software components. Component-based Software Engineering (CBSE) has been recognized as a sub-discipline of Software Engineering that focuses on the systematic approach for the development of Component-based Systems [CRN02]. It aims at realizing the concept of software reuse, which has become popular with the advent of Object-Oriented Programming and facilitates the development of software solutions for a DCS, by bringing together components running concurrently on different computers.

A DCS offers various advantages, such as fault-tolerance, reusability, scalability and an improved performance; still the development of reliable large-scale DCS is a major challenge in the field of software engineering. It introduces issues such as heterogeneity, reliability, openness, security, scalability, concurrency, and transparency [DIS01]. A brief description of these challenges is as follows:

Heterogeneity – It is defined as the difference in the networks, computer hardware, operating systems, programming languages, and the implementation of the components by different developers of the distributed applications. For the interoperation of the components of a DCS, it is necessary to mask these differences.

Openness – It is the characteristic that ascertains if a system can be extended and re-implemented in various ways. To achieve openness in DCS, there is a need for a comprehensive documentation of the software interfaces of the components that constitute the system.

Security – In a DCS, the components are remotely accessed and they communicate through message passing. Therefore, a DCS should provide a mechanism for maintaining the confidentiality, integrity and availability of the resources.

Scalability – A DCS should be scalable in order to effectively handle the increase in the number of resources and the number of users.

Concurrency – The components in a DCS should be able to access resources concurrently and should allow multiple/concurrent accesses to their interfaces.

Transparency – In a DCS, the separation of components should be concealed from the users and the application programmers, so that the system is perceived as a whole and not as a collection of independent components.

Thus, there is a need for a framework that allows an efficient interoperation of heterogeneous and distributed software components by addressing all or a subset of the above challenges. UniFrame [RAJ01] is one such approach to create an efficient and effective framework to develop a high confidence DCS. UniFrame addresses several of these above mentioned challenges.

1.1 UniFrame

UniFrame is a distributed computing research project that provides a unified framework, which will allow a seamless interoperation of (heterogeneous) distributed components for the development of reliable DCS. It involves:

a) Unified Meta-object Model (UMM), a meta-component model that provides the contracts of the software components belonging to different models. This helps in creating an open distributed system [RAJ00], b) Semi-automatic generation of glue and wrappers based on the specifications provided by the application developer, for achieving interoperability among software components, c) Incorporation of the Quality of Service (QoS) attributes, also know as non-functional attributes, at the component as well as DCS levels, d) The validation and assurance of the QoS, based on the concept of event grammars, and e) Specifying generative rules for creating DCSs by assembling available components along with their formal specifications [UNI01].

UniFrame follows the concept of CBSD of building the systems by integrating software components. A software component can be defined as a reusable unit of deployment and composition with the following characteristics: a) Contractually specified interfaces, b) Context-dependencies, c) Independently developed and deployed, and d) Subject to third party composition [SZY98]. The components provide two types of interfaces, namely the required interface and the provided interface. The required interface gives a description of how the component acquires services from other components and the provided interface describes how the component provides services to other components. Along with the advantages such as reusability and fault tolerance, the

CBSD also poses various challenges. The next section discusses a few of the challenges involved in creating DCS using CBSD.

1.2 Challenges in Component-based Software Development

As discussed above, the use of CBSE for the development of a reliable DCS poses several challenges. As the components in CBSD are independently developed and deployed, they are designed and implemented without a prior knowledge about the system that they may be a part of and other components with which they may interact. The developer of the component makes some assumptions about the environment that the component is going to interact with, which may result in different heterogeneity or incompatibility problems. Some of the problems have been summarized here [CRN02]:

1. Component specification – Characteristics of a component are usually specified by their interfaces. It consists of a precise definition of the component's operations and context dependencies. Though lot of work has been done in this direction [OMG01, MEN98], there is still no consensus about how it should be specified. This thesis aims at providing a mechanism to specify the component’s synchronization policies and the non-functional attributes.
2. Component Models – There exist different development models and technologies [MIC02, COR01, JAV01] with various characteristics and features, which may result in heterogeneity problems. Heterogeneity may also arise due to the difference in the implementation language, difference in the interfaces, difference in the communication patterns, and difference in the operating systems of the software components. Apart from these differences, the interoperation between the components, the relations between two component models and the relations between the component models and the system architecture may not be precisely defined.
3. Component-based software development process – Since the development of the components and that of the systems are two unsynchronized activities, the

development of component-based software systems has become more complex. A component well suited for a system may not function optimally when made to interact with other components in the system. Moreover, as a component-based system includes components with independent lifecycles, system evolution becomes a major concern.

4. Composition predictability – Once the system has been formed and all the relative attributes have been specified of a component, the corresponding attributes of the system of which they are composed has to be determined. The ideal approach to derive system attributes from component attributes is still a subject of research. UniFrame provides a mechanism to achieve the composition and decomposition of the values of the quality of service parameters from that of the individual components and of the system respectively [CHA03].
5. Trusted components and component certification – A component specification along with the attributes, should also provide parameters against which the component can be verified and validated. It is also known as contract between the component developer and the user [BEU99].
6. Non-functional Attributes – Apart from the functional attributes, some of the safety-critical domains and real-time systems requires assurance for non-functional attributes of the components. Ensuring quality and the non-functional attributes of the components is a major challenge in CBSE as well.

The problems of CBSD are overcome through software interoperability, which is defined as the ability of two or more software components to cooperate despite differences in language, interface, and execution platform [WEG96]. One way of accomplishing software interoperability is to categorize the problems into different levels and provide a mechanism for the software components to interoperate at each level. A software component specification/contract for its required and provided interfaces at each of these levels helps in achieving interoperability [BEU99]. This thesis focuses on the specification of software components for the development of high-confidence DCS.

In general the CBSD problems can be divided into following four levels:

1. Syntax/Signature Level – arises due to difference in the interface representation, and type mismatch of the operations of the cooperating components.
2. Semantic Level – arises due to the difference in the understanding of the messages exchanged between the requester and the provider of the services [HEI95] and the component's behavior.
3. Synchronization Level – arises due to the ordering and the blocking constraints that rule the availability of the component's services.
4. Quality of Service (QoS) Level – arises due to the difference in the Quality of Service levels of the required and provided services.

A formal specification or description of a component's interfaces and the non-functional attributes at each of the above mentioned levels helps identifying the heterogeneity problems at each of the levels. The specifications then can be matched in order to determine if the component can substitute for or cooperate with other components to form a DCS. It also aids in determining how one component can be modified to fit the requirements of the other components. A component specification is the formal description of the component's properties that reveals what services the component provides, how the component should be used and how it interacts with other components when brought together to form a distributed system.

This thesis contributes to the UniFrame research project by providing a mechanism for specifying a component's properties at the synchronization and the QoS levels and comparing two specifications to determine whether the components are substitutable for or compatible with each other.

1.3 Problem Definition and Motivation

Many researchers [ZAR96, HEI95, BEU99] have argued that specifying the component's behavior, syntactic and synchronization properties is a crucial part in CBSD. The specification makes the components less prone to error while integrating it with other components and more trust worthy. It helps in determining whether a component can be used within a certain context and what the component provides without actually delving into the details of how it has been implemented.

The specification should be such that it provides parameters against which the component can be verified and validated, thus providing a kind of contract between the component and its users [BEU99].

1.3.1 Problem Definition

The aim of this thesis is to provide a mechanism for the formal specification of the component's properties at the synchronization and the Quality of Service levels, identify types of matches applicable to different levels and provide a mechanism to compare/match the specifications of component in order to determine substitutability or compatibility relationships between them.

The first part of the thesis deals with specifying component's properties. As indicated earlier, component specifications can be divided into following four different levels:

1. Syntactic/Signature Level – At this level a component specification gives information about the syntax of the interfaces of the software components. It consists of the function name, the parameter list type and the return value type of the component's required and provided services. This helps in matching the type of the parameters and that of the return values of the provided services against required services. Zaremski [ZAR96] provides a comprehensive study on the

Signature Matching of the components in her thesis ‘Signature and Specification Matching.

2. Semantic Level – The Syntactic/Signature level specification describes the component’s type information, where as the semantic level specifications describe the component’s dynamic behavior. It characterizes the semantics of the interfaces of a software component. Apart from the signature matching, the thesis [ZAR96] also provides a mechanism to specify the semantic information of a software component and defines set of matches for both the levels.
3. Synchronization Level – The Specification at this level describes the synchronization policies used by the components to synchronize multiple clients accessing the component’s interfaces. It specifies the global behavior of objects in terms of synchronizations between method calls. This helps describing the dependencies between services provided by the component [BEU99].
4. Quality of Service (QoS) Level – At this level, the specification consists of describing the measures of the pre-specified QoS parameters, such as turn around time and throughput, etc. It also provides information about the dependency of the QoS parameters on the environment variables and usage patterns [BRA02].

The second part of the thesis constitutes defining the different types of matches with respect to substitutability and compatibility. Substitutability refers to the ability of two components to replace each other within a system. Compatibility is the ability of two components to interoperate, communicate and cooperate with each other when brought together to form a system. The matching of component’s specifications is done at all the four levels with respect to both the substitutability and the compatibility, and it can be divided into two or more categories for each of the criteria.

Zaremski’s work on Signature and Specification Matching addresses the specification and the matching issues at the first two levels i.e., the Syntax/Signature level and the Semantic level [ZAR96].

Many attempts have been made in the recent past to describe component's synchronization and QoS attributes, but not much success has been achieved. The concentration of the thesis is on specifying component's properties with respect to the third and the fourth levels i.e., the synchronization and the QoS levels. These specifications and the matching form a part of UniFrame Generative Domain Model (UGDM), a comprehensive knowledge base of UniFrame, and help the UniFrame Resource Discovery (URDS) in refining the component search process.

1.3.2 Motivation

The motivation for providing mechanisms for the specification of the component's synchronization and QoS properties and for matching the component specifications is as follows:

A formal specification at the syntax level indicates how a particular interface of a component can be accessed. It assists other components to understand the protocol for accessing the interfaces of the component. The formal specification at syntax level tells only about the parameter types, the return value types and the function name. It does not tell anything about the behavior of the operation. The components are independently developed and deployed, so it cannot be assumed that two components accomplishing the same task will have same interface names or the same order of the parameters. Hence, it becomes necessary to define the behavior of the components and the meanings of the messages exchanged between the components.

Components in a large-scale distributed system communicate with each other by exchanging services and data with one another. It is based on agreements between requesters and providers on, for example, message passing protocols, procedure names, error codes, and argument types. The specification at the semantic level ensures that these exchanges make sense and that the requester and the provider have a common understanding of the meanings of the requested services and data. It provides contracts on algorithms for computing requested values, the expected side effects of a requested procedure, or the source or accuracy of requested data elements [HEI95].

The interfaces of a component are accessed concurrently by more than one component and hence components must provide mechanisms to synchronize the concurrent accesses on the interfaces. A formal specification of these mechanisms helps other components understand the synchronization policies used by the component's interface and predict the behavior of the interfaces while the component is handling concurrent requests. It also helps in matching the synchronization requirements of the component requesting a service and the component providing the service.

Apart from the syntax, the behavior, and the synchronization aspects, there is a need to specify the non-functional attributes such as turn around time and throughput, of a component. It helps the service requester choose a component that fits these requirements pertinently.

This thesis focuses on the synchronization and the QoS levels. The syntactic and the semantic level contracts have been discussed in [ZAR95, ZAR96, ZAR97].

1.4 Objectives

The specific objectives of the thesis are:

1. Provide a mechanism for the formal specification of the components at the synchronization and the QoS level.
2. Define a set of matches for comparing two component specifications at these two levels.
3. Create a mechanism for the matching of the formal specification of the components at these two levels.

4. Develop a prototype to validate the formal specifications and the matching techniques developed.
5. Generalize the approach and make it a part of the UniFrame GDM.

1.5 Contributions

As discussed, the contracts play an important role in the creation of a high confidence DCS. The thesis provides guidelines for the creating software contracts at the synchronization and QoS levels. It borrows the guidelines for creating the contracts at the syntactic and semantic levels from [ZAR96]. The thesis also provides a set of matching criteria that enables users to match their requirements against the component specifications and to match two component specifications.

The contributions of the thesis are:

1. Definition of the synchronization policy catalog – The thesis provides a synchronization policy catalog that consists of commonly used synchronization policies employed in the development of the software components and the description of their behavior.
2. Provision of the mechanism to create the synchronization and the QoS level software component contracts – The thesis presents a generalized mechanism to create the synchronization and QoS contracts for a software component. The synchronization contract specifies the synchronization behavior of the component. The QoS contract provides the values of the QoS parameters for each of the components.
3. Definition of the matching criteria for the matching of the software components synchronization and QoS level contracts – The thesis provides a set of matching criteria that is used to match the contracts at the synchronization and QoS levels.
4. Validation of the proposed mechanism for the creation and matching of the contracts – The thesis validates the proposed mechanisms by creating the

contracts for a Document Management System and matching the specifications in order to determine if the components can interact or not.

1.6 Organization of the Thesis

The thesis consists of six chapters. Chapter one provides the introduction to the thesis and defines the problem. It also provides the motivation, objectives and the contributions of the thesis. The second chapter discusses the background of the thesis and few of the related works of the thesis. Chapter three provides an introduction to the Temporal Logic of Action that is one of the related works and provides a description of the constructs of Temporal Logic of Action+ specification language, used by the thesis to implement the contracts. Chapter four describes the proposed mechanism for creating the software contracts at the synchronization and the QoS levels in detail. Chapter five gives the matching criteria for the matching of the software contracts. The sixth chapter provides a case study and validates the proposed mechanism using an example of Document Management System. Chapter seven provides a discussion on the possible enhancements of the software contracts and a summary of this thesis.

2. BACKGROUND AND RELATED WORK

In the previous chapter, a brief introduction of the thesis is presented along with the problem definition, the thesis objectives and the contributions of this thesis. There has been a considerable amount of research work done [GUT93, LAM02, ZAR96] for specifying the component's properties. This chapter provides a discussion of the background and few of the related research works of the thesis. The next section talks about the 'Design by Contract' that forms the basis for this research. Section 2.2 takes the idea of the Design by Contract a bit further and discusses the topic 'making components contract aware'. Section 2.3 discusses the UniFrame, the research project that initiated this thesis. The chapter also talks about Larch family of specification languages and path expressions in sections 2.4 and 2.6 respectively, as related works. Section 2.5 talks about TLA that the thesis employs for its implementation.

2.1 Design by Contract

The main challenge in the development of a DCS is its high-confidence i.e., how to make a correct and a robust distributed system using heterogeneous software components. The correctness of a software system refers to the ability of the system to perform its job according to its specifications and the robustness of a software system refers to the ability of the system to handle abnormal situations. As discussed in chapter one, CBSD is a promising approach for the development of a DCS and reliability particularly becomes important in CBSD because of the concept of reusability of the software components. A component is a reliable reusable component, if its correctness can be trusted by the user/application developer to some extent. Hence, one of the major issues involved in CBSE is how an application developer trusts a particular software component.

To make sure that a distributed system performs correctly, there is a need for a systematic approach for specifying and implementing software components. For this purpose Meyer [MEY92] introduced a theory called Design by Contract. In this theory, a software system is viewed as a set of communicating components whose interaction is based on precisely defined specifications of the mutual obligations, called Contracts [ISE01].

The Design by Contract theory provides the following benefits [ISE01]:

1. A better understanding of the component-based software development method.
2. A systematic approach to building bug-free DCS.
3. An effective framework for debugging, testing and, more generally, quality assurance.
4. A method for documenting software components.
5. Better understanding and control of the inheritance mechanism.
6. A technique for dealing with abnormal cases, leading to a safe and effective language construct for exception handling.

In Design by Contract theory, the first goal is to define the functionality of each software element as precisely as possible, in order to ensure that it does what is indicated by the specification/contract. These contracts govern the interactions of the software components with its environment consisting of the execution environment and/or other software components.

2.2 Making Components Contract Aware

Based on the theory of Design by Contract [BEU99] defines a general model of software contracts and shows how existing mechanisms could be used to turn traditional components into contract aware ones. The authors agree that software components will bring about a bright future of large-scale software reuse, but argues that in mission-critical settings it raises many questions such as what if the component behaves unexpectedly, what if the components are faulty or what if it is misused by the application developer.

[BEU99] asserts that there is a need for a way of determining whether a given component can be used in a particular context. It indicates that these information would take the form of a specification that tells us what the component does, without entering into the details of how [BEU99]. According to the paper, the specifications should provide parameters against which the components can be verified and validated, thus, providing a contract between the component and its users.

The paper talks about four classes of contracts in the context of software components: syntax, behavioral, synchronization and quantitative. This classification is based on the increasing negotiable properties of the various levels of contract. The hierarchy based on the increasing negotiability of the different levels of contracts is as follows: 1) Syntactic, 2) Semantic, 3) Synchronization, and 4) Quantitative, with syntactic level contract at the bottom of the hierarchy.

2.2.1 Syntax Level Contract

The first level contract, i.e., the syntactic level contract takes the form of specifying the operations that a component perform, the operation’s input and output parameters and the possible exceptions that might be raised during the execution of the operation. The static type checking and the dynamic type checking forms an important part of syntax level contract. The paper provides a mechanism to verify that all clients use the component interface properly; static type checking performs the verification at the compile time whereas the dynamic type checking performs the verification at the runtime.

2.2.2 Semantic Level Contract

The syntactic specification of a component's operation provides information about how the operation may be invoked by other components but it does not define precisely the effect of operation's execution. For instance, the signature of adding two integers and subtracting two integers may be same, with two integer input parameters and an integer return value, but, they cannot be distinguished behaviorally with the information provided by the syntactic contract. Thus, there is a need for describing the behavior of the operations of the component. The behavioral level describes the mutual obligations and the benefits between the software components and the consistency conditions that could go wrong during the execution of the operations. It defines all the consistency conditions that could go wrong. The behavioral property of a component's property can be specified using Boolean assertions, pre- and post- conditions.

2.2.3 Synchronization Level Contract

The behavioral level contract assumes that the services are atomic, which may or may not be true in many application scenarios. The synchronization level contract provides the behavior of the component in terms of synchronization between the method calls. It gives information about the dependencies between the services provided by a component such as sequencing and parallelism. A contract at this level specifies the ways in which the component serves its clients. It is especially important in a single server and multiple client scenarios. In such an environment, the synchronization level contract guarantees to each client that whatever the other client's request may be, the requested service will be executed according to its specifications.

2.2.4 Quantitative Level Contract

The Quantitative level contract helps specifying the QoS attributes of the component's services such as turn-around time, throughput, maximum response delay, average response, and quality of the result, by enumerating the features that the component respects. The negotiability at this level of contract is the highest.

This thesis incorporates the idea of contracts at various levels and extends it to provide a matching of the contracts. It explores the different ways of specifying the component's the synchronization and the QoS properties and provides a mechanism to match two specifications in order to determine if they satisfy each others' requirements.

2.3 UniFrame

As indicated in chapter one, the UniFrame provides an effective and efficient framework for the development of a high confidence DCS. The framework follows the approach of CBSD and unifies existing and emerging distributed component models under Unified Meta Model (UMM), a common meta-model that encompasses various component models [RAJ00]. The UniFrame enables component developers to create, test, verify QoS and deploy the components and application developers to select components and generate a DCS in an automatic or semi-automatic fashion. Apart from the UMM, UniFrame also consists of the UniFrame Approach (UA), a UMM-based technique for the automatic or semi-automatic generation of a DCS.

2.3.1 Unified Meta-Component Model (UMM)

UMM tries to bridge the differences that exist among various component models. It consists of components, service and service guarantees, and infrastructure.

Components:

As discussed in chapter one, components are the building blocks of any distributed system. They are autonomous entities with non-uniform implementations i.e., components adhere to some distributed-computing model and there is no common implementation framework. Each component in UMM has a state, an identity, a behavior along with three aspects 1) Computational Aspect, 2) Co-operative Aspect, and 3) Auxiliary Aspect. The computational aspect indicates the objectives of the task performed by the component, the technique used to achieve these objectives and the specification of the functionality offered by the component. The co-operative aspect describes the interaction between components. It specifies the expected collaborators, other components with which this component may interact with. Pre-processing collaborators, and post-processing collaborators for two types of collaborators. Pre-processing collaborators refer to other components on which this component depends upon where as post-processing collaborators refer to those components that may depend on this component. The auxiliary aspect specifies additional features of the components, such as mobility, security and fault tolerance, in other words it describes the QoS attributes of the components.

Service and Service Guarantees:

The components offer services that are either a computational effort or access to an underlying resource, to other components. The quality of the service offered by a component depends on the computation performed by the component, the algorithm used to implement it, the resources required and the cost of each service. There exist different implementations for the same functionality and hence, there is a need to specify and guarantee the QoS offered by the components. Service Guarantee is an indication of the component’s confidence, i.e., how efficiently the component carries out its service with the changing environments. It helps in the selection of the component by an application developer.

Infrastructure:

The infrastructure of UniFrame consists of the headhunters and the Internet Component Broker (ICB). The ICB and the headhunters form a significant part of UniFrame Resource Discovery Service (URDS), which provides active distributed component management. The headhunter is responsible for searching heterogeneous, geographically distributed components. The headhunter actively discovers the components and tries to register them with it. It also co-operates with other headhunters in order to discover better quality components and large number of components. The ICB acts as a mediator between two components adhering to different component models.

2.3.2 The UniFrame Approach

UniFrame approach is a UMM-based technique for the development of a reliable DCS in a cost effective way. It consists of two levels: 1) the component level – where the components are developed with UMM specifications (explained in the next section), and 2) the system level – where a DCS is created by integrating the components, semi-automatically or automatically.

2.3.3 The UMM Specification

The UMM specifications are informally defined in a natural language-like style and hence, the component developers who wish to agree to and adopt the UniFrame should adhere to the UMM specification for a component. A sample UMM specification has been provided in the Figure 2.1. The example UMM specification describes the

UserValidationServer of a Document Management System, explained in the section 6.1 (Chapter six).

```
<?xml version="1.0" encoding='utf-8'?>

<!-- UMM description for DocumentValidationServer in the banking
domain example -->

<UMM_ConcreteComponent>
    <ComponentName> UserValidationServer </ComponentName>
    <ComponentSubcase> UserValidationServerCase1 </ComponentSubcase>
    <DomainName> Document </DomainName>
    <SystemName> DocumentManager </SystemName>
    <Description> Provide document validation service in document.
</Description>
    <ComputationalAttributes>
        <InherentAttributes>
            <id> magellan.cs.iupui.edu:1610/UserValidationServer</id>
            <Version> 1.0 </Version>
            <Author> Zhisheng Huang</Author>
            <Date> August 2002 </Date>
            <Validity> Yes </Validity>
            <Atomicity> Yes </Atomicity>
            <Registration> magellan.cs.iupui.edu:1310/HeadHunter
</Registration>
            <Model> Java RMI </Model>
        </InherentAttributes>
        <FunctionalAttributes>
            <Purpose> Act as validation server for users in document.
</Purpose>
            <Algorithms>
                <algorithm> JFC </algorithm>
            </Algorithms>
            <Complexity> O(1) </Complexity>
            <SyntacticContract>
                <ProvidedInterfaces>
                    <Interface> IValidationCase1 </Interface>
                </ProvidedInterfaces>
                <RequiredInterfaces>
                </RequiredInterfaces>
            </SyntacticContract>
```

Figure 2.1 UMM specification for the component UserValidationServer

```
            <Technologies>
                <technology> Java RMI </technology>
            </Technologies>
            <ExpectedResources>
                <resource> CPU: 500 mhz </resource>
                <resource> Memory: 1.0 Gb </resource>
            </ExpectedResources>
            <DesignPatterns>
                <pattern> </pattern>
            </DesignPatterns>
            <KnownUsage>
                <usage> </usage>
            </KnownUsage>
            <Aliases>
                <alias> </alias>
            </Aliases>
         </FunctionalAttributes>
    </ComputationalAttributes>
    <CooperationAttributes>
        <PreprocessingCollaborators>
            <Collaborator> DocumentTerminalCase1 </Collaborator>
        </PreprocessingCollaborators>
        <PostprocessingCollaborators>
        </PostprocessingCollaborators>
    </CooperationAttributes>
    <AuxiliaryAttributes>
        <Mobility> No </Mobility>
        <Security> L1 </Security>
        <FaultTolerance> L1 </FaultTolerance>
    </AuxiliaryAttributes>
    <QoS>
        <QoSMetrics>
            <Metric>
                <ParameterName> throughput </ParameterName>
                <FunctionName> validate </FunctionName>
                <Value> 675 </Value>
            </Metric>
            <Metric>
                <ParameterName> endToEndDelay </ParameterName>
                <FunctionName> validate </FunctionName>
                <Value> 1541 </Value>
            </Metric>
        </QoSMetrics>
        <QoSLevel> L1 </QoSLevel>
        <Cost> L1 </Cost>
        <QualityLevel> L1 </QualityLevel>
    </QoS>
</UMM_ConcreteComponent>
```

Figure 2.1 UMM specification for the component UserValidationServer (con't.)

The component developers should specify the parameters in the UMM specification during the development and deployment phase of the component. Thus, there is a need of a formal notation of these specifications such that component developers can refer them for specifying the computational, co-operative and auxiliary aspect of the software components. A formal representation of these specifications is also needed for enabling effective and efficient matching of the specifications against the application developer's requirements.

This thesis focuses on providing a formal specification of the component's synchronization and QoS properties. The UMM specification contains a brief description of these aspects and this thesis concentrates on elaborating these aspects for the component. In UniFrame, the components are matched against the application developer's requirements only at the syntactic level. One of the objectives of this thesis is also to provide a mechanism for matching the component specifications, not only at the syntactic level but also at the behavioral, the synchronization, and the QoS levels.

2.4 Larch Family of Specification Languages

The Larch family of specification languages [GUT93] is based on two-tiered, definitional style of specification. A Larch specification consists of 1) a trait written in Larch Shared Language (LSL), and 2) an interface written in Larch Interface Language (LIL). LSL helps defining properties of a component that are independent of the state of the component. It is an algebraic specification language and is independent of any programming language. LIL is used to describe the effect of the execution of an operation on the state of the component. It is a predicative language and uses pre- and post-conditions to specify the changes. It is designed for a specific programming language.

2.4.1 Larch Shared Language

LSL is used to describe the basic constructs and the programming details are left to the LIL. It has a simpler underlying semantics than most of the programming languages and hence, is less error prone. It is easier to make and to check assertions about the semantics properties of LSL than that of the interface language. Larch encourages a separation of concern, with the mathematical abstractions in the LSL tier, and the programming pragmatics in the LIL tier. The basic unit of an LSL specification is a trait. A trait may describe an abstract data type or may encapsulate a property shared by several data types. An LSL trait defines sorts and operators on sorts. A sort represents a set of terms, made up of operators that denote values. A trait introduces its sorts and operators, defines the operators with a set of equations that define which terms are equal to one another, and may assert additional properties about the sorts and operators. A trait can also include other traits.

2.4.2 Larch Interface Languages

LIL is used to specify the interfaces between program components. It provides the information needed by the clients to use a program's interface. It incorporates programming-language-specific notations for some of the features of the programming languages such as exception handling and aids in writing assertions about the state of a program. A few of the existing Larch Interface languages are Larch Interface Language for C (LCL), Larch Interface Language for Modula (Larch/ML), Generic Interface language (GIL), and Generic Concurrent Interface Language (GCIL).

As discussed earlier, the objectives of this thesis include specifying software components at the synchronization and the QoS levels. The Larch specification languages were designed for specifying component's properties. Many researchers [LAM89, GUT91, ZAR96] have been working on developing new Larch Interface Languages for specifying interfaces of the software components and providing specifications for the component's syntactic and semantic properties.

2.4.2.1 Specifying Objects of Concurrent Systems

The research [LAM89] aims at presenting a formal specification language for describing concurrent systems that is useful to designers and programmers. It proposes Generic Concurrent Interface Language (GCIL), a LIL used to describe concurrent systems. It is an extension of GIL, one of the LILs of the Larch family of specification languages. The GIL is used to specify the modules and the abstract data type for a sequential program. GIL specifications describe sequential programs in terms of objects, values, and states (object/value bindings). A specification in general describes an operation's behavior by describing the relation between the state in which the operation is invoked, and the state immediately following the operation. In a concurrent program it is necessary to specify when an operation may execute apart from the states before and after execution. A synchronization condition is needed to describe when an operation can start executing.

A GCIL specification not only describes the signatures of its operations and each operation's behavior in terms of state predicates like most sequential program specification languages, but also makes explicit statements about the operation's atomicity and synchronization. It focuses on processes and not on data as most of the interface languages. It concentrates on concurrent processes' interaction with other components rather than the processes themselves. It provides the specifier with the means to express and reason about the synchronization conditions for each operation. It supports modularization and abstraction in the specification of a class of concurrent systems and adds a general mechanism for specifying synchronization among component's operations, supporting a variety of concurrency models.

2.4.2.2 Specification and Signature of Components

The research described in [ZAR96] presents a way to retrieve components from a library, index a library and compare two components to help realize more of the potential of software libraries. It uses semantic information associated with the software components to match the requirements of the clients. It is based on the assumption that components are stored in a library and each component in a library has a signature (type information) and a specification (behavioral information). It presents a solution for the retrieval, indexing, navigation, substitution, sub-typing, and modification of the components. The authors claim that in all the cases the component's behavior plays an important role and matching of their specifications becomes an essential step in order to determine whether a particular relation holds among the components. The research gives a comprehensive set of matches for matching component specifications.

For matching the component's specifications, the research takes three parameters into consideration: 1) the kind of information used to describe the components – a component abstraction/description may vary from a textual description, a control- or Data- flow graphs to the signatures to the semantic information of the operations of the component, 2) the granularity of the components – components vary in size from the functions, the modular collections of functions to the stand-alone software systems, and 3) the degree of relaxation of the match – for each kind of match the research provides exact and relaxed sets of matches.

This research project uses ML as the component implementation language and Larch/ML as the specification language. Larch/ML is a Larch Interface Language for Modula and provides a mechanism to specifying the syntax of the component's operations and the behavior of the components.

The research work defines a set of matches for the functions and the modules components and for the signature and specification of the component's operations. For functions, the work provides separate set of matches for signature and specification, whereas for modules they define the matches independently of whether the abstracts are signatures or specifications. A brief overview of the matches is provided below:

Function Signature Matching:

It consists of set of matches that can be performed to match the syntax of the interfaces of the components. The degree of match is categorized further in two types, exact and relaxed, for both Signature and Specification matching. The relaxed match in the signature matching allows reordering of elements in a tuple, uncurrying of arguments to a function, renaming of type constructors and instantiation of type variables. The function signature match definitions are expressed in terms of whether a transformation can be applied to the functions signatures such that the results are equal. The relaxed match for the function signature matching is further divided into transformation relaxation and partial relaxation. The transformation relaxation, in matching component's signature, includes transforming a type expression for instance renaming type constructors, changing whether a function is curried or uncurried, changing the order of types in a tuple, and changing the order of the arguments, to achieve a match. The partial relaxation matching involves finding a partial order relationship between the types.

Function Specification Matching:

In the function specification matching, matches are defined in terms of a logical relationship, such as implication, between two specifications or between parts of the specifications. The matching is categorized based on the pre- and the post-conditions and specification predicates, implication between the pre- and the post-conditions. The pre/post matches and the predicate matches are further divided into exact and relaxed matches. The exact match is differentiated from the relaxed match by the equivalence relationship between the pre- and the post- condition of the component's specifications. For instance, the exact pre/post match can be expressed as the existence of an equivalence relationship between the pre-conditions and the post-conditions of the component's specifications; where as the relaxed match between the components can be defined as the existence of the implication relationship between the pre- and the post-conditions of the component's specifications.

The Signature and Specification matching of software components to a large extent tries to specify various aspects of a component and provides mechanisms for matching them against the requirements of an application developer. However, it does not provide mechanisms to specify the synchronization and the QoS properties of the components and to match them. This thesis extends the work of Amy [ZAR96] and defines a set of matches for the synchronization and the QoS levels. It utilizes the signature and the specification matching functions for the component's operations as it is.

2.5 Temporal Logic of Action (TLA)

The TLA is a logic-based mechanism for specifying and verifying concurrent systems. Algorithms in concurrent systems are usually specified with a program and correctness of the algorithm indicates that the program satisfies a desired property. The TLA provides a simple approach of specifying the algorithm and the property of the program using formulas in a single logical system. The idea is to reason about an abstract algorithm, and not about the actual concurrent program that is executed. It uses logics, which are the formalization of everyday mathematics and hence, simpler than programs [LAM02].

The TLA specification is based on formulae and each TLA formulae can be expressed in terms of familiar mathematical operators, with some additional operators. TLA is built on a logic of actions, which is a language for writing predicates, state functions, and actions, and a logic for reasoning about them. TLA combines two logics: logic of actions and a standard temporal logic. It provides a mathematical foundation for describing systems, and TLA+ is a complete specification language built on top of this foundation.

An execution of an algorithm is a sequence of steps resulting in a set of new states. Hence, the semantic meaning of an algorithm can be defined as the set of all its possible executions. Temporal logic can be defined as the reasoning about the algorithms or the reasoning about the sequences of the states. A temporal formula is built from elementary formulas using the Boolean operators and one unary operator '[]'. The semantics of temporal logic is based on behaviors, an infinite sequence of states. The meaning of a temporal formula is defined in terms of the meanings of the elementary formulas it contains. Temporal logic of action has been discussed in detail in the chapter three.

This thesis uses TLA+ language to specify the synchronization policies, and the synchronization aspects of a software component.

2.6 Predicate Path Expression

Path expressions (PE) [CAM74] are a high-level synchronization construct used to specify concurrent processes. They form an integral part of the data abstraction mechanism in a programming language and helps in specifying synchronization in terms of the allowable sequences of operations on a component of the abstract data type. PEs use a notation based on regular expressions. In their pure form, PEs do not allow convenient specification of many synchronization problems [AND79]. The research takes the path expression synchronization construct along three dimensions – specification, verification, and implementation, by introducing Predicate Path Expressions (PPE), i.e., PE with Predicates. PE provides a complete description of possible sequences of operations and PPE adds the power of synchronization with counters to it.

The researchers claims that adding predicates to PEs, and analyzing the need for parallel constructs will increase the expressive power of the PPE, and will allow direct specification of most commonly occurring synchronization problems. They also argue that by formally specifying the semantics of the path expressions in a tractable ways, they will allow the use of well-know verification techniques for sequential programs and will make it possible to verify consistency of a data abstraction, absence of deadlock and absence of starvation. For the implementation of PPE, they proposed to use Algol 68S, a subset of Algol 68 and a multiprocessor environment.

Predicate Path Expressions aim at specifying concurrent processes and their synchronization properties. But not much concrete work has been observed in this direction. This thesis also focuses on specifying the synchronization and the QoS properties of software components and also provides a mechanism for verifying the system with respect to the synchronization aspect. It also provides mechanism for matching of components specifications.

This chapter provided a discussion of the background of the thesis and brief descriptions about few of the related works. It also discusses the TLA+ as one of the related work. The thesis bases the implementation of the proposed mechanism [Chapters five and six], for specifying the software components synchronization behavior, on TLA+. The next chapter provides a detailed description of the constructs of TLA+.

3. TEMPORAL LOGIC OF ACTION

Chapter two discussed a few of the related works of this thesis. It also provided a brief introduction of Temporal Logic of Action (TLA). This chapter explains the constructs of TLA in detail. It also gives an introduction to the liveness and fairness properties of a specification and discusses their representation in TLA. These constructs and properties have been described in [LAM02] in detail.

TLA, as discussed in chapter two, is a logic for specifying and reasoning concurrent systems. TLA+ makes it practical to describe a system by a single formula. Most of a TLA specification consists of ordinary, non-temporal mathematics. It is a robust language for writing mathematics. It takes ideas from programming language for modularizing large specifications. TLA+ is good for specifying a wide class of systems, from program interfaces (APIs) to distributed systems. It's especially well suited for describing asynchronous systems, i.e., the systems with components that do not operate in strict lock-step.

3.1 Logic of Action

TLA, as mentioned earlier, combines two logics: logic of actions and a standard temporal logic. Logic of action consists of writing state functions, predicates and actions, and a logic for reasoning about them. The Temporal Logic is defined in the next section.

A state function f is an expression built from variables and values. [f] represents a mapping from the set of states to a set of values. s[f] is used to denote the value [f] assigned to state s. The semantic definition is given as:

$s[f] \equiv f\,(\text{forall } v\text{: } s[v]/v)$

where, f (forall v : s[v]/v) denotes the value obtained from f by substituting s[v] for v, for all variables v.

A predicate is a boolean-valued state function, i.e., a predicate P is a state function such that s[P] equals true or false for every state s. A state s satisfies a predicate P, if and only if s[P] equals true.

State functions correspond both to expressions in ordinary programming languages and to sub-expressions of the assertions used in ordinary program verification. Predicates correspond both to boolean-valued expressions in programming languages and to assertions.

An action is any boolean-valued expression formed from variables, primed variables, and values. An action represents a relation between old states and new states, where the unprimed variables refer to the old state and the primed variables refer to the new state. The meaning [A] of an action A is a relation between states, a function that assigns a boolean s[A]t to a pair of states s, t. s refers to the old state and t, refers to the new state in s[A]t. and s[A]t is obtained from A by replacing each unprimed variable v by s[v] and each primed variable v' by t[v].

$s[A]t \equiv A(\text{forall } v\text{:}s[v]/v,\ t[v]/v')$

The pair of states s; t is called an "A step" iff s[A]t equals true.

3.2 Simple Temporal Logic

An execution of an algorithm is often thought of as a sequence of steps, each producing a new state by changing the values of one or more variables. Execution can be considered as resulting sequence of states, and the semantic meaning of an algorithm as the set of all possible executions of the algorithm. Reasoning about algorithms will therefore require reasoning about sequences of states. Such reasoning is the province of temporal logic [LAM91].

3.2.1 Temporal Formulas

A temporal formula is built from elementary formulas using boolean operators and the unary operator '[]'. The meaning of temporal formulas is defined in terms of the meanings of the elementary formulas it contains. A simple temporal logic is defined for a class of elementary formulae. The semantics of temporal logic is based on behaviors, where a behavior is an infinite sequence of states. A temporal formula is interpreted as an assertion about behaviors.

[F] of a formula F is a boolean-values function on some behavior. $\sigma[F]$ denote the boolean value that formula F assigns to behavior σ. The behavior σ is said to satisfy F if and only if $\sigma[F]$ is true.

The conjunction and the negation rules are applied in the following manner:

$\sigma\ [F \wedge G] \equiv \sigma[F] \wedge \sigma[G]$,

i.e., a behavior satisfies $F \wedge G$ if and only if it satisfies both F and G.

$\sigma[\neg F] \equiv \neg\sigma[F]$, i.e., a behavior satisfies $\neg F$ if and only if it does not satisfy F.

If <<s_0, s_1, s_2, s_3,…>> represents a behavior whose first state is s_0, second state is s_1 and so on, then,

[[]F] ≡ \for all n Є Natural Numbers: << s_n, s_{n+1}, s_{n+2}, …>>[F], …………(Eq 3.1)

<<s_n, …>>[F] asserts that F is true n of this behavior,

The temporal formula (Eq 3.1) asserts that if F is true at all times during the behavior <<s_0,…>>. []F asserts that F is always true.

There are other temporal formulas described in [LAM91] and [LAM02]. These formulas will be explained in this thesis as and when needed.

3.2.2 Actions as Temporal Formulas

The Temporal Logic of Actions, or TLA, is obtained by letting the elementary temporal formulas be actions. In Logic of Action, if P is a predicate, then, s[P]t is a Boolean, which equals s[P], for any states s and t, i.e., a pair of states s; t is a P step if and only if s satisfies P, and if P' is an action, then, s[P']t equals t[P] for any states s and t.

The meaning [A] of an action A is a relation between states, which is a function that assigns a boolean s[A]t to a pair of states s, t. s[A]t is defined by considering s to be the "old state" and t the "new state", so s[A]t is obtained from A by replacing each unprimed variable v by s[v] and each primed variable v' by t[v]:

s[A]t ≡ A(forall v : s[v]/v; t[v]/v')

where, [A] is defined to be true for a behavior if and only if the first pair of states in the behavior is an A step.

TLA formulas are built up from actions using logical operators and the temporal operator '[]'. Thus, if A is an action, then []A is a TLA formula.

Its meaning is defined as follows.

$<<s_0, s_1, s_2, \ldots>>[[]A]$

$\equiv$ forall n Є Natural numbers: $<< s_n, s_{n+1}, s_{n+2}, \ldots>>[A]$

$\equiv$ forall n Є Natural numbers: $s_n [A] s_{n+1}$

In other words, a behavior satisfies []A if and only if every step of the behavior is an A step.

For any action A and a state function f, $[A]_f$ can be defined as,

$[A]_f \equiv A \vee (f' = f)$, where f is a state function and A is any action.

To illustrate the use of TLA, the following section gives a very simple specification of describing the behavior of mutual exclusion synchronization policy which is described in the next chapter in detail. The behavior of mutual exclusion can be defined in terms of three steps: 1) Request, 2) Acquire, and 3) Release. The behavior can be described as follows:

1. A process requests for a resource.
2. The resource is shared among several processes.
3. A process is allowed to acquire the process only if no other process has acquired the resource.
4. The process releases the resource after execution and the resource is ready to be acquired by other processes.

The TLA+ specification of the synchronization policy 'mutual exclusion' can be represented as follows:

```
(******************************************************************************************)
(* This module gives the behavior of the synchronization policy 'Mutual Exclusion'        *)
(* The CONSTANT Process indicates the set of process that are trying access the shared    *)
(* variable that is protected using the Mutual Exclusion                                  *)
(* The behavior has been informally defined in the Thesis, " Synchronization and Quality  *)
(* of Service Specifications and Matching of Software Components".                        *)
(******************************************************************************************)

-------------------------------------- MODULE MutualExclusion------------------------------

(******************************************************************************************)
(* Declaring the CONSTANTS and the VARIABLES                                              *)
(******************************************************************************************)

EXTENDS TLC
CONSTANT Process
VARIABLE stateP

(******************************************************************************************)
(* Defining the intial values of the variables                                            *)
(******************************************************************************************)

Invariant == /\ \A p1, p2 \in Process : (stateP[p1] = "executing")
                                        /\ (stateP[p2] = "executing") => (p1=p2)
             /\ stateP \in [Process -> {"idle", "waiting", "executing", "releasing"}]

Init == stateP = [p \in Process |-> "idle"]
-------------------------------------------------------------------------------------------
(******************************************************************************************)
```

Figure 3.1 Simple TLA+ specification for the synchronization policy Mutual Exclusion

```
(* Defining the methods supported by the synchronization policy Mutual Exclusion as       *)
(* described in    the thesis                                                             *)
(******************************************************************************************)

Request(p) == /\ stateP[p] = "idle"
              /\ stateP' = [stateP EXCEPT ![p] = "waiting"]

Acquire(p) == /\ stateP[p] = "waiting"
              /\ \A q \in Process : stateP[q] # "executing"
              /\ stateP' = [stateP EXCEPT ![p] = "executing"]

Release(p) == /\ stateP[p] = "executing"
              /\ stateP' = [stateP EXCEPT ![p] = "releasing"]

Exiting(p) == /\ stateP[p] = "releasing"
              /\ stateP' = [stateP EXCEPT ![p] = "idle"]
(******************************************************************************************)
(* Defining the possible next statements or the actions that could be taken.              *)
(******************************************************************************************)

Next == \E p \in Process : Request(p) \/ Acquire(p) \/ Release(p) \/ Exiting(p)

PNext == Next /\ Print(stateP, TRUE)
-------------------------------------------------------------------------------------------
(******************************************************************************************)
(* Defining the Specification                                                             *)
(******************************************************************************************)
Spec == /\ Init
        /\ [][PNext]_stateP

THEOREM Spec => []Invariant
=============================================================================================
```

Figure 3.1 Simple TLA+ specification for the synchronization policy Mutual Exclusion (con't.)

TLA+ specifications are partitioned into modules. The specification in the Figure 3.1 contains one module named MutualExclusion.

Every symbol that appears in a formula must either be a built-in operator of TLA+, or else it must be declared or defined. The statement,

```
EXTENDS TLC,
```

extends and includes the statements and declarations in the module TLC.

The keywords VARIABLES and CONSTANTS, declare a set of variables and constants, respectively. For instance, in the above example, the statements

```
CONSTANT Process, and
VARIABLE stateP
```

declare the constant Process and the variable stateP

The statement,

```
Init == stateP = [p \in Process |-> "idle"]
```

defines the intial predicate *Init* of the module. It assigns initial values to the variables of the current module. In the above case the statement assigns the variable *stateP* a value idle for every p in the constant process. The symbol '\in' is used for Є, i.e. belongs to and the symbol '= =' represents 'is defined to equal'.

The next-state action *Next* consists of either a process requesting a resource, acquiring a resource, releasing a resource or exiting. Hence, the statement

```
Next == \E p \in Process: Request(p) \/ Acquire(p) \/ Release(p) \/
Exiting(p)
```

defines the possible next-state actions. The actions requesting, acquiring, releasing and exiting a resource has been defined as the following state predicates: *Request(p)*, *Acquire(p)*, *Release(p)*, and *Exiting(p)*. The next step is one of these predicates and hence the predicate *Next* combines them using a disjunction. The symbol \E stands for there exists. The statement *PNext* defines a predicate that is *Next* and a Print statement that prints the current state of the variable *stateP*.

The *Next* predicate consists of the predicates *Request(p)*, *Acquire(p)*, *Release(p)*, and *Exiting(p)*. The predicates include a primed value and an unprimed value. For instance, in the example given below,

```
Acquire(p) == /\ stateP[p] = "waiting"
              /\ \A q \in Process : stateP[q] # "executing"
              /\ stateP' = [stateP EXCEPT ![p] = "executing"]
```

The first statement 'stateP[p] ="waiting"' checks if the current value of the variable stateP[p] is waiting or not. In the second statement '*\A q \in Process: stateP[p] # "executing"*', the symbol '\A' stands for 'for all', and '#' stands for 'not equal to'. The second statement checks if the current value of *stateP[p]* is not equal to *"executing"*. The last statement consists of primed values, where it states that if first and second statements are true, then the new value of the variable *stateP*, indicated by *stateP'* is the same as the old value of the variable *stateP* except the value *stateP[p]*, which changes to executing. Here keyword EXCEPT stands for except in the previous line. All these statements have been combined using a conjunction, i.e., each of statements should be true for the step Acquire(p) to take place.

Using the statement,

```
Spec == /\ Init
        /\ [][PNext]_stateP
```

the specification/behavior of the synchronization policy can be expressed in a single formula. The statement asserts that

1. The initial state of the behavior satisfies *Init*.
2. And each of the steps of the behavior satisfies *Next*.

The first assertion is expressed as the formula *Init* and the second assertion is expressed using a temporal formula [][PNext]_stateP. The temporal formula []F, as discussed earlier asserts that the formula F is always true. Hence, the statement [][PNext]_stateP asserts that the PNext is true for every step in the behavior.

The statement,

```
Invariant == /\ \A p1, p2 \in Process:
                (stateP[p1] = "executing")
             /\ (stateP[p2] = "executing") => (p1=p2)
             /\ stateP \in [Process -> {"idle", "waiting", "executing",
            "releasing"}]
```

is a state predicate that does not appear in the specification. The specification implies that the predicate *Invariant* is an invariant that does not change throughout the sequence of steps. The statement,

```
THEOREM Spec => []Invariant
```

provides an assertion that the specification implies the invariant *Invariant*. Theorem specifies a temporal formula that is satisfied by every behavior, and since the invariant should be satisfied by every behavior, the implication '*spec => []invariant*' is a theorem.

3.3 TLA+ Language

Apart from the operators described in the previous specification, the thesis uses other operators as well. This section gives a brief description of those operators based on the categories they belong to.

Logic operators:

1. \/, /\, ¬, and =>	Stand for or, and, not, and equivalence respectively.
2. \A, and \E	Stand the universal and existential quantifiers respectively.

Set Operators:

1. \, \CUP, and \in	Set difference, union, and belong to operators respectively.

Functions:

1. f [e] stand for	Function application
2. [x \in S |-> e]	Function f such that f [x] = e for x belongs to S].

Records

1. e.h	The h-field of record e.
2. [h_1 |-> e_1, …, h_n |-> e_n]	The record whose h_i field is e_i.
3. [r except !.h = e]	Record r' equal to r except r'.h = e].

Tuples

1. e[i]	The ith component of tuple e.
2. <<e_1,…, e_n>>	The n-tuple whose ith component is e_i.

Miscellaneous constructs

1. IF p THEN e_1 ELSE e_2	e_1 if p is true, else e_2.

Action operators

1. e'	The value of e in the final state of a step.
2. $[A]\_{e}$	$A \vee (e' = e)$
3. UNCHANGED e	e' = e

Temporal operators

1. []F	F is always true.
2. ◊F	F is eventually true.
3. WF_{e} (A)	Weak fairness for action A.
4. SF_{e}	Strong fairness for action A.

The next two examples show the use of most of the constructs explained above. The first example is the implementation of the Readers Writers synchronization policy and utilizes the most of the record and the set based operators. The second example is an implementation of the FirstComeFirstServed Priority policy that makes use of IF-THEN-ELSE construct and the tuple based operators.

```
-------------------------- MODULE ReadersWriters -----------------------------------------
EXTENDS TLC
CONSTANT Client
VARIABLE stateC

TypeCheck == stateC \in [readers: SUBSET Client, writers: SUBSET Client]

Init == /\ TypeCheck
        /\ stateC = [readers |-> {}, writers |-> {}]
-------------------------------------------------------------------------------------------
enter_read(p) == /\ stateC.writers = {}
                 /\ stateC' = [stateC EXCEPT !.readers = @ \cup {p}]

exit_read(p) == /\ p \in stateC.readers
                /\ stateC' = [stateC EXCEPT !.readers = @ \ {p}]

enter_write(p) == /\ stateC.writers = {}
                  /\ stateC.readers = {}
                  /\ stateC' = [stateC EXCEPT !.writers = @ \cup {p}]

exit_write(p) == /\ p \in stateC.writers
                 /\ stateC' = [stateC EXCEPT !.writers = @ \ {p}]

Next == \E p \in Client: enter_read(p) \/ exit_read(p) \/ enter_write(p) \/ exit_write(p)

PNext == Next /\ Print(stateC, TRUE)
-------------------------------------------------------------------------------------------
Spec == Init /\ [][PNext]_stateC
===========================================================================================
```

Figure 3.2 Simple TLA+ specification for the synchronization policy Readerswriters

```
-------------------------------- MODULE FCFSPriority---------------------------------

EXTENDS TLC, Naturals
CONSTANT Process
VARIABLE stateP

Priority == <<5,4,3>>
Invariant == /\ \A p1, p2 \in Process : (stateP[p1] = "executing")
                                        /\ (stateP[p2] = "executing") => (p1=p2)
             /\ stateP \in [Process -> {"idle", "waiting", "executing", "releasing"}]

Init == /\ stateP = [p \in Process |-> "idle"]
-------------------------------------------------------------------------------------
Request(p) == /\ stateP[p] = "idle"
              /\ stateP' = [stateP EXCEPT ![p] = "waiting"]

Acquire(p) == /\ stateP[p] = "waiting"
              /\ IF \A q \in Process:
                                      /\ \/ (Priority[p] > Priority[q]/\
stateP[q]="waiting")
                                         \/ (Priority[p] < Priority[q] /\ stateP[q] =
"releasing")
                                         \/ (Priority[p] = Priority[q] /\ stateP[q] =
"waiting")
                 THEN /\ stateP' = [stateP EXCEPT ![p] = "executing"]
                 ELSE /\ UNCHANGED <<stateP>>
```

Figure 3.3 Simple TLA+ specification for the synchronization policy FCFSPriority

```
Release(p) == /\ stateP[p] = "executing"
              /\ stateP' = [stateP EXCEPT ![p] = "releasing"]


Exiting(p) == /\ stateP[p] = "releasing"
              /\ stateP' = [stateP EXCEPT ![p] = "idle"]


Next == \E p \in Process: Request(p)\/ Acquire(p) \/ Release(p)\/Exiting(p)
PNext == Next /\ Print(stateP, TRUE)
-----------------------------------------------------------------------------------------------
------------

Spec == /\ Init
        /\ [][PNext]_stateP
===============================================================================================
============
```

Figure 3.3 Simple TLA+ specification for the synchronization policy FCFSPriority (con't.)

3.4 Liveness and Fairness

The specifications in the above examples indicate 'what a system must not do'. For instance the specification of the mutual exclusion synchronization policy indicates that a process cannot acquire a resource if some other process has already acquired it. These properties are called safety properties. This section talks about the liveness properties, they are the ones that cannot be violated at any point of time. It is expressed as temporal formulas.

As discussed earlier, a state assigns a value to every variable, and a behavior is an infinite sequence of states. A temporal formula is true or false of a behavior. The temporal formulas used till now are all a combination of three simple kinds of formulas, with the following meanings:

1. A state predicate, viewed as a temporal formula, is true of a behavior if and only of it is true in the first state of the behavior.
2. A formula []P, where P is a state predicate, is true of a behavior if and only if P is true in every state of the behavior.
3. A formula [][N]v , where N is an action and v is a state function, is true of a behavior if and only if every successive pair of steps in the behavior is a [N]v step.

The section 3.2 describes the []F temporal formula. This section describes the other temporal formulas. The section redefines following operators to explain the liveness and fairness properties:

1. ◊F, which is equal to ¬[]¬F, asserts that F is not always false, which means that F is true at some time. ◊ is read as eventually.
2. ◊<A>$_v$, is equal to ¬[][¬F]$_v$, where A is an action and v a state function. It asserts that not every step is a (¬A) \/ (v' = v) step, so some stem is a ¬((¬A) \/ (v' = v)) step. It is equivalent to A /\ (v' ≠ v). It asserts that eventually an <A>$_v$ step occurs.
3. []◊F, asserts that at all times, F is true then or at some time, i.e., for time n_0, it implies that F is true at some time $n_1 \geq n_0 + 1$. Continuing the process, results in

the implication that F is true at an infinite many instants. Hence, []◊<A> assert that infinitely many $<A>_v$ steps occur.

4. ◊[]F asserts that eventually at some time, F becomes true and remains true thereafter, i.e., F is eventually always true.

```
-------------------------------------- MODULE Mutex ---------------------------------------

EXTENDS TLC
CONSTANT Process
VARIABLE stateP

Invariant == /\ \A p1, p2 \in Process : (stateP[p1] = "executing")
                              /\ (stateP[p2] = "executing") => (p1=p2)
           /\ stateP \in [Process -> {"idle", "waiting", "executing", "releasing"}]

Init == stateP = [p \in Process |-> "idle"]
-------------------------------------------------------------------------------------------
Request(p) == /\ stateP[p] = "idle"
              /\ stateP' = [stateP EXCEPT ![p] = "waiting"]

Acquire(p) == /\ stateP[p] = "waiting"
              /\ \A q \in Process : stateP[q] # "executing"
              /\ stateP' = [stateP EXCEPT ![p] = "executing"]

Release(p) == /\ stateP[p] = "executing"
              /\ stateP' = [stateP EXCEPT ![p] = "releasing"]

Exiting(p) == /\ stateP[p] = "releasing"
              /\ stateP' = [stateP EXCEPT ![p] = "idle"]
-------------------------------------------------------------------------------------------
(*******************************************************************************************)
(* Describing one of the liveness properties.                                             *)
eventually == \A p \in Process: /\ (stateP[p] = "waiting") ~> (stateP[p] = "executing")
(*******************************************************************************************)
```

Figure 3.4 Example for liveness property Muex.tla

```
Next == \E p \in Process : Request(p) \/ Acquire(p) \/ Release(p) \/ Exiting(p)

PNext == Next /\ Print(stateP, TRUE)
-----------------------------------------------------------------------------------------------


Spec == /\ Init
\*      /\ [][Next]_stateP
        /\ [][PNext]_stateP
        /\ eventually
```

Figure 3.4 Example for liveness property, Mutex.tla (con't.)

```
CONSTANT
      Process = {1,2}

SPECIFICATION Spec
INVARIANT Invariant
PROPERTY eventually
```

Figure 3.5 Mutex.cfg

In the above example (Figure 3.4), the statement,

```
eventually == \A p \in Process:(stateP[p] = "waiting") ~>
                               (stateP[p] = "executing")
```

states that for all p in Process if *stateP[p]= "waiting"*, then eventually *stateP[p]="executing"*. The formula eventually is included as a PROPERTY in the configuration file(discussed in the next section), as shown in the Figure 3.5.

The next section provides an overview of the model checker used to check the specifications written in TLA+.

3.5 TLC Model Checker

TLC [LAM02] is a tool written in Java for finding errors in TLA+ specifications. It handles specification of the following standard form

Init /\ [][Next]$_{vars}$ /\ Temporal

where, *Init* is the initial predicate, *Next* is the next-state action, *vars* is the tuple of all variables, and *Temporal* is a temporal formula that specifies liveness conditions if they exist. TLC finds errors in a specification by verifying if it satisfies properties that it should.

The input to the TLC consists of a TLA_ module and a configuration file. The configuration file tells TLC the names of the specification and of the properties to be checked. For instance, the configuration file for the specification in the Figure 3.1 looks as given below:

```
CONSTANT
        Process = {1,2}

SPECIFICATION Spec
INVARIANT Invariant
```

Figure 3.6 MutualExclusion.cfg

The TLA+ module is the specification file and the configuration file tells the TLC the names of the specification and of the properties that are to be checked. For instance, in the configuration file, MutualExclusion.cfg, for the TLA+ specification MutualExclusion.tla, tells TLC that Spec is the specification for MutualExclusion.tla. It does that using SPECIFICATION keyword and the properties are indicated using PROPERTY keyword as shown in the Figure 3.4.

TLC works by generating behaviors that satisfy the specification. To do this, it must be given what we call a model of the specification. To define a model we must assign values to the specification's constant parameters. The invariance *Invariant* in the TLA+ specification file, MutualExclusion.tla, can also be represented as a property and included in the configuration as follows:

In the TLA+ specification a statement

```
InvProperty == []Invariance
```

can be added and in the configuration file a statement,

```
PROPERTY InvProperty
```

can be added. The TLC then checks the if the specification implies the property InvProperty.

TLC also provides a mechanism to check the invariance using the keyword INVARIANT. For instance, for the MutualExclusion.tla (Figure 3.1) example, the configuration file can be modified by adding the statement,

```
INVARIANT Invariance
```

for TLC to perform an invariance check.

The statement,

```
CONSTANT Process = {1,2}
```

assigns a value to the constant Process, defined in MutualExclusion.tla. The configuration file must specify values for all the constant parameters of the specification.

There are two ways of using TLC. The default method is model checking, in which it tries to find all reachable states, i.e., all states that can occur in behaviors satisfying the formula specified by the specification. TLC can also be run in simulation mode, in which it randomly generates behavior, without trying to check all reachable states. The thesis [LAM02] provides the details of both the methods.

3.6 Limitations of TLA+

The following are the limitations of TLA+:

1. TLA+ allows describing a wide variety of values, for example, the set of all sequences of prime numbers, but TLC can compute only a restricted class of values, called TLC values. These values are built from the following four types of primitive values: Booleans, Integers, Strings, Model Values. Booleans, Integers, Strings have the same meaning as in any other programming language. Model values are the values introduced in the CONSTANT statement of the configuration file. Model Values with different names are assumed to be different. TLC value can be defined as either of the following: 1) A primitive value. 2) A finite set of comparable TLC values. Two TLC values are said to be comparable if and only if the semantics of TLA+ determines that they are equal.
2. Checking of specification requires evaluating expressions. For example, TLC does invariance checking by evaluating the invariant in each reachable state, i.e., computing its TLC value, which should be TRUE. TLC evaluates expressions by evaluating sub-expressions from left to right. TLC cannot evaluate unbounded quantifiers or unbounded CHOOSE expressions. It cannot evaluate any expression whose value is not a TLC value.

TLC can evaluate a TLA temporal formula if and only if the following holds good for the formula:

1. The formula is a conjunction of formulas that belong to one of the following classes: a) state predicate, b) invariance formula – a formula of the form []P, where P is a state predicate, c) box-action formula – a formula of the form $[][A]_v$, where A is an action and v is a state function, and d) simple temporal formula – a

formula constructed from temporal state formula and simple action formulas by applying simple Boolean operators.

2. TLC can evaluate all the ordinary expressions of which the formula is composed.

TLC evaluates states as follows: It evaluates an invariant by calculating the invariant's value, which is either TRUE or FALSE. It evaluates the initial predicate or the next-state action by computing a set of states, for the initial predicate, i.e., the set of all initial states, and for the next-state action, i.e,., the set of possible successor states.

3.7 Running TLC Model Checker

The command to run TLC to check a particular specification is as follows:

Program_name options spec_file ……………………………………….(Eq 3.2)

where,

Program_name is specific to the operating system being used. It is usually java tlc.TLC.

spec_file – it is the file containing the TLA+ specification. Each TLA+ module is stored on a disk has a name with an extension .tla. The extension .tla may be omitted from spec_file.

options is a sequence consisting of zero or more of the following:

1. -deadlock – it tells TLC not to check for deadlock.
2. -simulate – tells TLC to run in simulation mode, generating randomly chosen behaviors, instead of generating all reachable states.
3. -config config_file – Specifies that the configuration file is named config_file, which must be a file with extension .cfg. The extension .cfg may be omitted from

config_file. If this option is omitted, the configuration file is assumed to have the same name as spec_file, except with the extension .cfg.

The above mentioned set of options is not a complete set of options. The full set of options can be found in [LAM02].

The chapter provided an overview of the constructs of TLA+, used in this thesis, for specifying and matching the software contracts at the synchronization level. The next chapter discusses in detail the proposed mechanism of specifying software components at the Synchronization and QoS levels. It also provides a brief description of the research work done by [ZAR96] for creating contracts at the semantic and syntactic levels.

4. THE SYNCHRONIZATION AND QOS LEVEL CONTRACTS

As discussed in previous chapters, there is a need for providing contracts for the interface of a software component, and mechanisms to match these contracts, for the development of a high-confidence DCS using the components. This chapter provides a mechanism for creating the synchronization level and QoS level contracts, for the interface of the components developed under the UMM. The mechanism for creating a contract at the syntactic and semantic levels for the interface of the components follows the work of [ZAR96], discussed in chapter two, section 2.4.2.2.

4.1 Synchronization and Quality of Service Specifications

The specification or a contract is a written description of what a component expects of its clients and what the clients can expect of it. Contracts in the component-based software development consist of precisely defined checkable interface specifications. Precisely defined checkable specifications are defined as the specifications that indicate all the aspects of the component's interface necessary for the clients to access them. These include the acceptable inputs, the return values and their meanings, the behavioral, and the synchronization pre-, post-conditions, and invariants and the performance guarantees. The contracts are used to specify the behavioral compositions and the obligations on the participating components. They aim at formalizing the collaboration and behavioral relationships between the components. They are also used to verify the implementation of the interfaces.

As discussed in section 1.3.1., various aspects of a component's interface that needs to be defined in the contract of a component include the syntactic, the semantic, the synchronization, and the QoS aspects. Hence, a component's contract is divided into four levels, namely, the syntactic level, the semantic level, the synchronization level, and the QoS level. The syntactic, semantic, and the synchronization contracts of a component are specified in terms of the contract of its methods. Contracts at these three levels usually take the form of explicit pre- and post-conditions. The pre-condition describes the constraints that the caller must satisfy before actually making a call on the component's interface. The post-condition describes the conditions that the component's implementation will guarantee to be true after the method has been executed. The contract at the QoS level specifies the value of the pre-defined QoS parameters.

Once the contracts of the components have been created, the contracts of the client components are matched with that of the component's contracts in order to determine whether the semantics allows them to interoperate with each other. The matching of the specifications is also required while searching the software components from the component library that satisfy a given query and while integrating the software components to form a DCS in order to understand how a component's interface can be used and to ensure that the components work properly when integrated. The matching of the specifications is performed based on certain criteria and rules, discussed in chapter five.

While selecting components from the component library for a given query system, the specifications of interfaces of the components are matched and combined to predict the behavior of the resulting system. This helps in verifying the behaviors of the resulting system against the required system specifications and detecting problems before actually integrating the components.

Hence, the process of specifying the components and matching their specifications consists of the following three steps, which are the objectives of the thesis:

1. Creating contracts at the four levels.
2. Defining matching criteria for each of the levels.
3. Matching the contracts and verifying and validating the resulting system.

The next few sections elucidate the first step in detail, thus, proposing a solution to the first objective of the thesis.

4.2 Creating Contracts at Different Levels

This section describes the contract at four different levels, i.e., the syntactic level, the semantic level, the synchronization level and the QoS level. It provides an overview of the first two levels as, these have been addressed in [ZAR96], and a detailed study of the last two levels. The thesis contributes towards specifying the component's interface, discussed in this section, and defining the matching criteria, discussed in the next chapter at the last two levels.

4.2.1 Syntactic Level Contract

The objective of the syntactic level contract is to provide the syntactic information about a component's interface that includes the signature of the method, the method name, the parameters types and the return types. The contract at this level helps the users in determining if the components will interoperate when integrated and facilitates the integration process to form the end system by providing information about how to use the interface of the components.

The syntactic contract uses the following predicates:

1. Name of the component – It indicates the name of the component and can be represented as,

Component: <*ComponentName*>

where, ComponentName is the name of the component.

2. Name of the Interface – It indicates the name of the interface of the component. It can be represented in the following general predicate form:

   Interface: <*InterfaceName*>

   where, InterfaceName is the name of the interface of the component.

3. Name of the method – It indicates the name of the method of the component and can be represented as,

   MethodName: <*MethodName*>

   where, MethodName is the name of the method of the component.

4. Signature of the Method – It gives the signature of the method of the component. It has the following general format:

   Signature: <*Signature*>

   where, Signature is the signature giving the return types, the parameter types and the parameter names.

5. Parameter types of the Method – It gives type information about the parameters of the method and can be represented as,

   ParameterType: <*[Type1 , Type2, Type3…]*>

   where, Type1, Type2, and Type3 are the type of the parameters. There can be zero or more parameters. [] indicate that the types are optional, since the parameters are optional.

6. Return Types of the Method – It gives the return type of the methods of the component. It can be represented as,

   ReturnType: <*Type*>

where, Type is the return type. It can also be void if the component does not return any value.

The contract takes the following general form:

Component: <*ComponentName*>
  Interface: <*InterfaceName1*>
    MethodName: <*MethodName1*>
    Signature: <*Signature1*>
    ParameterType: <*[Type11, Type12, Type13…]*>
    ReturnType: <*Type1*>

    MethodName: <*MethodName2*>
    Signature: <*Signature2*>
    ParameterType: <*[Type21, Type22, Type23…]*>
    ReturnType: <*Type2*>]

where,

<*InterfaceName*> is the name of the Interface
<*MethodName1*> and <*MethodName1*> are the names of the Methods
<*Signature1*> and <*Signature2*> are the signatures of the methods MethodName1 and MethodName2 respectively.
<*Type11, Type12, Type13*> and < *Type21, Type22, Type23*> - are the types of the parameters of the methods MethodName1 and MethodName2 respectively.
<*Type1*> and <*Type2*> are the return types of the methods MethodName1 and MethodName2 respectively.

The section 6.2 shows an example of the syntactic contract of a component in the predicate form. The thesis utilizes the work of [ZAR96] for implementing the syntactic level contract. [ZAR96] uses ML as the component implementation language and relies on the ML type system for the parameter and the return types. It uses the Larch/ML, a Larch interface language for ML, as the specification language to specify a component's syntactic contract.

4.2.2 Semantic Level Contract

The semantic level contract provides the information such as the behavioral pre-, post- conditions, and the invariants about the each of the methods of the component's interface.

The contract uses the following predicates:

1. Name of the component – It indicates the name of the component and can be represented as,

   Component: <*ComponentName*>

   where, ComponentName is the name of the component.
2. Name of the Interface – It indicates the name of the interface of the component. It can be represented in the following general predicate form:

   Interface: <*InterfaceName*>

   where, InterfaceName is the name of the interface of the component.
3. Name of the method – It indicates the name of the method of the component and can be represented as,

   MethodName: <*MethodName*>

   where, MethodName is the name of the method of the component.
4. Behavioral Precondition – It gives the behavioral precondition of the method, i.e., the constraint that the caller must satisfy before making a call on the method. It can be represented as,

   BehavioralPrecondition: <*precondition*>

   where, precondition is the behavioral precondition.

5. Behavioral Invariant – It gives the behavioral invariant of the method, i.e., the conditions of a component that remain unchanged during the execution of the method. It has the following predicate format:

   BehavioralInvariant: <*Invariant*>

   where, Invariant is the behavioral precondition.

6. Behavioral Post-condition – It gives the behavioral post-condition of the method, i.e., the condition that the component guarantees, after the execution of the method, to the calling methods. It can be represented as,

   BehavioralPostcondition: <*postcondition*>

   where, postcondition is the post-condition of the method.

The contract takes the following general form:

Component: <ComponentName>

Interface: <*InterfaceName*>

MethodName: <*MethodName1*>

Signature: <*Signature1*>

BehavioralPrecondition: <*precondition1*>

BehavioralInvariant: <*invariant1*>)

BehavioralPostcondition: <*postcondition1*>

[MethodName: <*MethodName2*>

Signature: <*Signature2*>

BehavioralPrecondition: <*precondition2*>

BehavioralInvariant: <*invariant2*>)

BehavioralPostcondition: <*postcondition2*>

where,

<*ComponentName*> is the name of the component

<*InterfaceName*> is the name of the Interface

<*MethodName1*> and <*MethodName2*> are the name of the Methods of the component ComponentName.

<*Signature1*> and <*Signature2*> are the signatures of the methods MethodName1 and MethodName2 respectively.

<*precondition1*> and <*precondition2*> are the behavioral preconditions that must be satisfied by the clients calling the methods MethodName1 and MethodName2, respectively.

<*invariant1*> and <*invariant2*> are the invariants that must be satisfied by the clients before calling the methods MethodName1 and MethodName2, respectively, during the execution and after the execution of the method.

<*postcondition1*> and <*postcondition2*> are the behavioral post-conditions that the component ensures that it will satisfy if the respective methods MethodName1 and MethodName2 are executed and terminated.

The section 6.3 shows an example of the behavioral contract in the predicate form. [ZAR96] provides the mechanism to specify a software component behavior, using the Larch/ML specification languages.

4.2.3 Synchronization Level Contract

This section gives the synchronization contract for the software component's interface. The section first discusses why synchronization is needed for a component's interface, and what information the contract should reveal. Then, it provides a mechanism to formally represent the contract.

4.2.3.1 Synchronization between Software Components

In a uni-processor system, synchronization is needed if the processes access shared memory locations or shared variables. The processes may need to block as they may be using variables that they share with other processes. They may need to take turns or wait for some operation to complete in some other process before they may reasonably do their own work.

In component-based DCS, components run concurrently, independent of each other and they communicate through synchronous remote procedure calls. In synchronous remote procedure calls an invocation on a server component's method is handled by an ordinary exported procedure. The server component refers to the component that provides a service and the component that requires the service is referred to as the client component. The client component sends a message, to the server component and blocks itself. The method on the server component receives the message, performs the task and sends the result back to the blocked client component causing the client's method to be awakened and the server component's method terminates. Communications between the server and the client component's methods take place through parameters. The parameters can be transmitted in both the directions in a machine-independent data format. Since the processes related to component's method are active in different address spaces, there are no shared variables. But, synchronization, in a remote procedure call, is needed to control the access to the variables of the server component that are accessed by multiple processes in the address space of the server component. It helps in guarding the methods of the server component that are accessed by multiple client components at the same time and the resources provided by the server component.

4.2.3.2 <u>The Synchronization Contract Requirements</u>

Defining contracts for a software component raises questions such as, what should the contract contain? How the contract should be represented formally? etc. This section tries to answer these questions for the synchronization level contract.

The synchronization level contract for a software component provides the following information:

1. Synchronization policy – the contract indicates the synchronization policy used by the component's interface to handle multiple client accesses/requests on a method. The policies include a few basic synchronization policies, such as mutual exclusion, readers-writers policy, producer-consumer policy [Section 4.2.3.3].
2. Synchronization policy's behavior – This describes the behavior of the synchronization policy utilized by the components to protect their interfaces. It is provided by indicating the change in the state of the variables of the software components, before and after the execution of the methods.
3. Synchronization policy's implementation methods – The Synchronization policy's implementation method indicates the method used to implement the synchronization policy for the component's interface. For instance, the policy mutual exclusion can be implemented using semaphores or mutexes.
4. The synchronization pre-condition of the methods – The synchronization pre-condition of a method specifies the synchronization conditions/constraints that the caller and the other methods of the server component must satisfy in order to start the execution of the method. It specifies the obligations that a client component must meet before it may invoke the server method.
5. The synchronization action for the methods – the synchronization action specifies the action with respect to synchronization that will be taken by the component's method if the pre-condition is satisfied.
6. The synchronization invariants for the methods – the synchronization invariants specifies the conditions that must hold throughout the process of a client making

invocation on the server component's method, server component executing the method and the termination of the method.

7. The post-conditions of the methods – the synchronization post-condition of a method specifies the conditions that must hold after the server component's method is terminated. It specifies the guarantees that the server component makes to its clients.

4.2.3.3 The Synchronization Policy Catalog

There exist a number of basic synchronization policies that could be used to protect the shared resources. The component developers can use these basic synchronization policies to protect their interfaces or use a variation of these synchronization policies. There is a need for a catalog that provides a list of existing basic synchronization policies. This thesis defines a set of basic synchronization policies and also provides a mechanism to formally represent their behaviors. This section discusses the basic synchronization policies that form the catalog and a brief description about their behavior.

The synchronization policy catalog provides a standard list of the synchronization policies that could be utilized by the software components to protect their interfaces. It contains detailed descriptions of the various synchronization policies. It acts as a reference manual for the component developers to incorporate the synchronization policies in the components.

The catalog provides the following information about the synchronization policies:

1. Name: Indicates the name of the synchronization policy.
2. Processes: Indicates the minimum number of processes involved in the policy.

3. Region: Indicates the number of critical regions that the processes are going to be in while trying to access the shared resource. It also indicates the number of the shared resources that are being protected from the concurrent access of the processes, by the policy.
4. Behavior: It indicates the behavior of the synchronization policy in terms of the processes and the regions.

The following are the list of basic synchronization policies and the description of their behavior.

I. Mutual Exclusion Policy –
    1. Name: MutualExclusion
       Processes: n Processes
       Region: n Critical Regions; 1 Shared Resource
       Behavior:
        i. If a process Pi is executing in its critical region, i.e., the region where the component accesses the shared resource, then no other processes can be executing in their critical regions.
        ii. If no process is executing in its critical region and there exist some processes that wish to enter their critical regions, then the selection of the process that will enter the critical region next cannot be postponed indefinitely.

It can be implemented using the following methods:

a. Request(p) – It provides a method for the process p to request the shared resource.
    i. Pre-condition: state of the process p is idle.
    ii. Invariant:
        - If the state of a process p1 is executing and the state of another process p2 is also executing then. p1 = p2.

- State of any process p1 is idle, waiting, executing, or released.

iii. Action: state of the process p changes to waiting.

iv. Post-condition: state of the process p is waiting.

b. Acquire(p) – It provides a method for the process p to acquire the shared resource.

i. Pre-condition: state of the process p is waiting.

ii. Invariant:

- If the state of a process p1 is executing and the state of another process p2 is also executing then. p1 = p2.
- State of any process p1 is idle, waiting, executing, or released.

iii. Action: state of the process p changes to executing.

iv. Post-condition: state of the process p is executing.

c. Release(p) – It provides a method for the process p to release an already acquired resource.

i. Pre-condition: state of the process p is executing.

ii. Invariant:

- If the state of a process p1 is executing and the state of another process p2 is also executing then. p1 = p2.
- State of any process p1 is idle, waiting, executing, or released.

iii. Action: state of the process p changes to released.

iv. Post-condition: state of the process p is released.

II. Conditional Synchronization –

1. Name: Barrier

   Processes: n Processes

Region: n Critical Regions

Behavior:

i. The processes that have entered a critical region and have finished executing the region must wait until all other processes have finished their respective critical regions and then, they leave together.

It can be implemented using the following methods:

a. EnterRegion(p,r) – It provides a method for the process p to enter the critical region r.
    i. Pre-condition: state of the process p is idle.
    ii. Invariant: State of any process p1 is idle, executing, terminated or proceed.
    iii. Action: state of the process p changes to executing.
    iv. Post-condition: state of the process p is executing.
b. LeaveRegion(p,r) – It provides a method for the process p to leave the critical region it has already entered.
    i. Pre-condition: state of process p is executing.
    ii. Invariant: State of any process p1 is idle, executing, terminated or proceed.
    iii. Action: state of the process p changes to terminated.
    iv. Post-condition: state of process p is terminated.
c. Proceed(p) – It provides a method for the process p to check if it can proceed with the next statement.
    i. Pre-condition: state of all the n processes is either terminated or proceed.
    ii. Invariant: State of any process p1 is idle, executing, terminated or proceed.
    iii. Action: state of the process p changes to proceed.
    iv. Post-condition: state of process p is proceed.

2. Name: BoundedBuffer

   Processes: 2 processes; a producer thread and a consumer thread

   Region: 2 Critical Regions; 1 Shared Resource called Buffer

   Behavior:

   i. The buffer is a region that can be filled by the producer thread and emptied by the consumer thread.
   ii. The threads communicate through shared memory. The buffer is the shared data between the two processes/threads.
   iii. The buffer is of finite size. The problem is known as the Bounded-buffer producer-consumer problem. It assumes a fixed buffer size and the consumer must wait for a new item when buffer is empty and the producer must wait when the buffer is full.

If e is an element that is put in the buffer, then it can be implemented using the following methods:

a. SProduce(p, e) – It provides a method for the producer thread p to start producing an element and put it in the buffer.
   i. Pre-condition: state of the process p is idle, size of buffer is less than its maximum size.
   ii. Invariant: The number of elements in the buffer is no more than the size of the buffer.
   iii. Action: state of the process p changes to producing.
   iv. Post-condition: state of the process p is producing.
b. EProduce(p,e) – It provides a method for the producer thread p to stop producing.
   i. Pre-condition: state of the process p is producing.
   ii. Invariant: The number of elements in the buffer is no more than the size of the buffer.
   iii. Action: the element e is added to the tail of the buffer and state of the process p changes to idle.

iv. Post-condition: state of the process p is idle and element e is added in the buffer and the size of the buffer is increases by one.

c. SConsume() – It provides a method for a consumer thread c to leave the critical region it has already entered.

   i. Pre-condition: state of the process c is idle and the size of the buffer is greater than zero.
   ii. Invariant: The number of elements in the buffer is no more than the maximum size of the buffer.
   iii. Action: state of the process c changes to consuming.
   iv. Post-condition: state of the process c is consuming.

d. EConsume() – It provides a method for the consumer thread to stop consuming.

   i. Pre-condition: state of the process c is consuming.
   ii. Invariant: The number of elements in the buffer is no more than the size of the buffer.
   iii. Action: state of the process c changes to consuming and the first element from the buffer is removed.
   iv. Post-condition: state of the process c is consuming and size of the buffer decreases by one.

3. Name: ReadersWriters

   Processes: 2 Types of Processes; n Readers and m Writers

   Region: n X m Regions; 1 Shared Resource

   Behavior:

   i. The resource is shared among concurrent processes. Some of the processes, known as Readers are interested in reading the values from the shared resource and some of them are interested in updating the values of the shared resource and are known as Writers.

ii. If two readers access the shared resource simultaneously, there will not be any inconsistency. They will be able to access it simultaneously and will get the same value.

iii. More than one writer cannot access the shared resource as they may do a write operation, which changes the value of the resource and if two writers do the write operation simultaneously, it might result in an inconsistent state of the resource.

iv. Writers should have exclusive access to the shared resource, i.e., if there are any readers accessing data, writer must wait till all the readers have finished reading.

It can be implemented using the following methods:

a. Enter_read(p, r) – It provides a method for a reader thread p to enter the critical region r to read the value of the shared resource.
   i. Pre-condition: the thread p is a reader thread, the state of the process p is idle and there are no writers currently accessing the the resource.
   ii. Invariant: NONE
   iii. Action: state of the process p changes to executing.
   iv. Post-condition: state of process p is executing.

b. Exit_read(p, r) – It provides a method for a reader thread p to leave the critical region it has already entered.
   i. Pre-condition: the thread p is a reader thread and state of p is executing.
   ii. Invariant: NONE
   iii. Action: state of the process p changes to idle.
   iv. Post-condition: state of process p is idle.

c. Enter_write(p, r) – It provides a method for a process p to enter the critical region r to update or modify the value of the shared resource.

i. Pre-condition: the thread p is a writer thread, state of p is idle and there is no reader or writer thread currently accessing the resource.
ii. Invariant: NONE
iii. Action: state of the process p changes to executing.
iv. Post-condition: state of process p is executing.

d. Exit_write(p, r) – It provides a method for a writer thread p to leave the critical region it has already entered

i. Pre-condition: state of thread p is executing.
ii. Invariant: NONE
iii. Action: state of the process p changes to idle.
iv. Post-condition: state of process p is idle.

III. Priority Synchronization

1. Name: FCFSPriority
Processes: n Processes
Region: n Regions; 1 Shared Resource
Behavior:

i. Each process is associated with a priority assigned according to the request arrival time, i.e., on a first come first server basis.
ii. A process with a higher priority is allowed to access the resource first.

The synchronization policy can be implemented using the following methods:

a. Request(p) – It provides a method for a process p to request for entering the critical region.

i. Pre-condition: the state of the process p is idle.
ii. Invariant: NONE
iii. Action: the state of the process p changes to waiting.
iv. Post-condition: the state of process p is waiting.

b. Acquire(p) – It provides a method for a process p to enter the critical region according to the priority assigned. In this case the priority is assigned on a first come first served basis.
    i. Pre-condition: P_1 is the set of all the processes with a priority higher than the process p and the state of the all the processes in P_1 is released; and P_2 is the set of all the processes with a priority lower than the process p and the state of all the processes in P_2 is waiting; and the state of the process p is idle.
    ii. Invariant: NONE
    iii. Action: state of the process p changes to executing.
    iv. Post-condition: state of process p is executing.

c. Release(p) – It provides a method for a process p to leave the critical region.
    i. Pre-condition: state of the process p is executing.
    ii. Invariant: NONE
    iii. Action: state of the process p changes to releasing.
    iv. Post-condition: state of process p is releasing.

2. Name: SJFPriority

   Processes: n Processes

   Region: n Regions; 1 Shared Resource

   Behavior:
    i. Each process is associated with a priority assigned according to the time taken to execute the critical region, the lesser the time, higher the priority.
    ii. A process with higher priority is allowed to access the resource first.

The synchronization policy can be implemented using the following methods:

a. Request(p) – It provides a method for a process p to request for entering the critical region.
    i. Pre-condition: the state of the process p is idle.
    ii. Invariant: NONE
    iii. Action: the state of the process p is waiting.
    iv. Post-condition: the state of process p is waiting.
b. Acquire(p) – It provides a method for a process p to enter the critical region according to the priority assigned. In this case the priority is assigned on shortest job first basis.
    i. Pre-condition: if the set of processes P_1 with priority higher than the process p and the state of the all the processes in P_1 is released; if the set of processes P_2 with priority lower than the process p and the state of all the processes in P_2 is waiting; and the state of the process p is idle.
    ii. Invariant: NONE
    iii. Action: state of the process p is executing.
    iv. Post-condition: state of process p is executing.
c. Release(p) – It provides a method for a process p to leave the critical region.
    i. Pre-condition: state of the process p is executing.
    ii. Invariant: NONE
    iii. Action: state of the process p is releasing.
    iv. Post-condition: state of process p is releasing.

IV. Other Synchronization Policies

1. Name: Bound

   Processes: n Processes

   Region: 1 Critical Region, maximum size of Region = N

   Behavior:

i. At most N processes can be in the critical region at any point of time. Sleeping Barber problem explained in the next section uses this policy.

The synchronization policy can be implemented using the following methods:

a. EnterRegion(p) – It provides a method for the process p to enter a critical region.
    i. Pre-condition: state of the process p is idle and the number of process currently executing the critical region is less than N.
    ii. Invariant: State of any process p1 is idle, executing, or terminated and the size of the Region is no more than N.
    iii. Action: state of the process p changes to executing.
    iv. Post-condition: state of the process p is executing and the number of process executing the region is incremented by one.
b. LeaveRegion(p,r) – It provides a method for the process p to leave the critical region it has already entered.
    i. Pre-condition: state of the process p is executing.
    ii. Invariant: State of any process p1 is idle, executing, or terminated and the size of the Region is no more than N.
    iii. Action: state of the process p changes to terminated.
    iv. Post-condition: state of process p is terminated and the number of process executing the region is decremented by one.

2. Name: relay

   Processes: 2 processes

   Region: 2 Regions

   Behavior:

i. The process P1 entering Region 1 can leave immediately after executing; however the process P2 entering Region 2 is blocked and cannot leave Region 2 until P1 arrives at Region 1. The Sleeping Barber Problem explained in the next section uses this policy.

The synchronization policy can be implemented using the following methods:

a. EnterRegion(p) – It provides a method for the process p to enter any of the critical regions.
    i. Pre-condition: state of the process p is idle.
    ii. Invariant: NONE
    iii. Action: state of the process p changes to executing.
    iv. Post-condition: state of the process p is executing.
b. LeaveDRegion(p,r) – It provides a method for the process p entering the dependent region Region2 to leave the critical region.
    i. Pre-condition: state of the process p is executing and all the processes that wish to enter Region 1 is either executing or terminated from the critical region.
    ii. Invariant: NONE
    iii. Action: state of the process p changes to terminated.
    iv. Post-condition: state of process p is terminated.
c. LeaveDRegion(p,r) – It provides a method for the process p entering the dependent region Region1 to leave the critical region.
    i. Pre-condition: state of the process p is executing.
    ii. Invariant: NONE
    iii. Action: state of the process p changes to terminated.
    iv. Post-condition: state of process p is terminated.

4.2.3.4 Some Famous Synchronization Problems

This section describes few of the famous synchronization problems that can be used for synchronizing the interfaces of the software components. The problems include the Dining Philosopher Problem, and the Sleeping Barber Problem.

The following are a description of the behaviors of the problems:

1. Name: DiningPhilosopher
   Processes: n Processes
   Region: 2 Regions; eating and thinking; n Resources
   Behavior:
   i. The philosophers are seated around a circular table. They eat or think at a particular time.
   ii. There are n resources (forks) placed one each in between two philosophers. Each philosopher has a plate of spaghetti in front of them and eating spaghetti needs 2 forks.
   iii. When a philosopher is hungry he/she tries to pick up one fork at a time (left and right) in either order.
   iv. If successful in acquiring two forks a philosopher starts eating and puts down the forks once he/she is done and continues to think.

2. Name: SleepingBarber
   Processes: 2 Processes; the barber process/thread and the customer process/thread
   Region: 4 Barber Regions and 5 Customer Regions; 1 Barber Chair, n Customer Chairs
   Behavior:
   i. If there are no customers in the barber room, the barber sits on the barber chair and falls asleep.
   ii. When a customer arrives, he wakes up the sleeping barber and the barber starts cutting the customer's hair.

iii. If additional customers arrive, while the barber is cutting a customer's hair, they either sit down in the waiting room, if there are empty chairs, and wait until the barber becomes free or leave the shop if all the chairs are occupied.
iv. The customer waiting in the waiting room enters the barber room once the barber is free and the waits for the barber to finish the hair cut.
v. The barber finishes the hair cut and informs the customer. The barber waits until the customer leaves the barber room and the customer leaves the barber room.

There are several ways of implementing these policies. For example, mutual exclusion can be implemented using semaphores, mutexes or monitors. Similarly, producer consumer problem can be implemented either using monitors or semaphores. The behavior of the policies remains the same for all the implementations. The next section gives an example TLA+ specification describing the behaviors of one of the policy. It also presents TLA+ specification for a policy implemented using semaphores.

4.2.3.5 Formal Representation of the Synchronization Policies

The thesis uses TLA+ to formally specify the behavior of the synchronization policies. It also provides specifications for the synchronization policies for different implementation methods. An example TLA+ specification for the policy 'MutualExclusion' has been provided in the Figure 4.1. The Figure 4.2 provides the TLA+ specification of 'MutualExclusion' that uses semaphores as the implementation method.

```
(******************************************************************************************)
(* This module gives the behavior of the synchronization policy 'Mutual Exclusion'        *)
(* The CONSTANT Process indicates the set of process that are trying access the shared    *)
(* variable that is protected using the Mutual Exclusion                                  *)
(* The behavior has been informally defined in the Thesis, " Synchronization and Quality  *)
(* of Service Specifications and Matching of Software Components".                        *)
(******************************************************************************************)

----------------------------------- MODULE MutualExclusion---------------------------------

(******************************************************************************************)
(* Declaring the CONSTANTS and the VARIABLES                                              *)
(******************************************************************************************)

EXTENDS TLC
CONSTANT Process
VARIABLE stateP

(******************************************************************************************)
(* Defining the Invariant and the initial values of the variables                         *)
(******************************************************************************************)

Invariant == /\ \A p1, p2 \in Process : (stateP[p1] = "executing")
                                        /\ (stateP[p2] = "executing") => (p1=p2)
             /\ stateP \in [Process -> {"idle", "waiting", "executing", "releasing"}]

Init == stateP = [p \in Process |-> "idle"]
-------------------------------------------------------------------------------------------
(******************************************************************************************)
(* Defining the methods supported by the synchronization policy Mutual Exclusion as       *)
(* described in the thesis                                                                *)
(******************************************************************************************)
```

Figure 4.1 TLA+ specification for the synchronization policy mutual exclusion (MutualExclusion.tla)

```
Request(p) == /\ stateP[p] = "idle"
              /\ stateP' = [stateP EXCEPT ![p] = "waiting"]

Acquire(p) == /\ stateP[p] = "waiting"
              /\ \A q \in Process : stateP[q] # "executing"
              /\ stateP' = [stateP EXCEPT ![p] = "executing"]

Release(p) == /\ stateP[p] = "executing"
              /\ stateP' = [stateP EXCEPT ![p] = "releasing"]

Exiting(p) == /\ stateP[p] = "releasing"
              /\ stateP' = [stateP EXCEPT ![p] = "idle"]

(******************************************************************************************)
(* Defining the possible next statements or the actions that could be taken.              *)
(******************************************************************************************)

Next == \E p \in Process : Request(p) \/ Acquire(p) \/ Release(p) \/ Exiting(p)

PNext == Next /\ Print(stateP, TRUE)
-------------------------------------------------------------------------------------------
(******************************************************************************************)
(* Defining the Specification that is deadlock and starvation free.                       *)
(******************************************************************************************)
Spec == /\ Init
        /\ [][PNext]_stateP
        /\ (\A p \in Process : WF_stateP(Exiting(p)))
        /\ WF_stateP(\E p \in Process : Acquire(p))  \* Deadlock free
        /\ (\A t \in Process : WF_stateP(Exiting(t)))
        /\ (\A t \in Process : SF_stateP(Acquire(t))) \* Starvation free

THEOREM Spec => []Invariant
```

Figure 4.1 TLA+ specification for the synchronization policy mutual exclusion (MutualExclusion.tla) (con't.)

```
(******************************************************************************************)
(* This module gives the behavior of the synchronization policy 'Mutual Exclusion' that   *)
is implemented using semaphores                                                           *)
(******************************************************************************************)

------------------------------------------MODULE MutexSem ---------------------------------
(******************************************************************************************)
(* Declaring the CONSTANTS and the VARIABLES                                              *)
(******************************************************************************************)

EXTENDS Semaphore, TLC
CONSTANT Process
VARIABLES stateP, rSem
-------------------------------------------------------------------------------------------

(******************************************************************************************)
(* Defining the Invariant and the intial values of the variables                          *)
(******************************************************************************************)

Invariant == /\ \A p1, p2 \in Process : (stateP[p1] = "executing")
                                        /\ (stateP[p2] = "executing") => (p1=p2)
             /\ stateP \in [Process -> {"idle", "waiting", "executing", "releasing"}]

Init == /\ rSem = 1
        /\ stateP = [p \in Process |-> "idle"]
-------------------------------------------------------------------------------------------
(******************************************************************************************)
(* Defining the methods supported by the synchronization policy Mutual Exclusion as       *)
described in the thesis                                                                   *)
(******************************************************************************************)
```

Figure 4.2 TLA+ specification for the policy MutualExclusion using Semaphore (MutexSem.tla)

```
Request(p) ==  /\ stateP[p] = "idle"
               /\ stateP' = [stateP EXCEPT ![p] = "waiting"]
               /\ UNCHANGED rSem

Acquire(p) == /\ stateP[p] = "waiting"
              /\ P(rSem)
              /\ stateP' = [stateP EXCEPT ![p] = "executing"]

Release(p) == /\ stateP[p] = "executing"
              /\ stateP' = [stateP EXCEPT ![p] = "released"]
              /\ V(rSem)

exiting(p) == /\ stateP[p] = "released"
              /\ stateP' = [stateP EXCEPT ![p] = "idle"]
              /\ UNCHANGED rSem

(*******************************************************************************************)
(* Defining the possible next statements or the actions that could be taken.             *)
(*******************************************************************************************)

Next == \E p \in Process: Request(p) \/ Acquire(p) \/ Release(p)\/ exiting(p)

PNext == Next /\ Print(stateP, TRUE)
-------------------------------------------------------------------------------------------
(*******************************************************************************************)
(* Defining the Specification that is deadlock and starvation free.                      *)
(*******************************************************************************************)

Spec == Init /\ [][Next]_<<stateP>>
THEOREM Spec => []Invariant
===========================================================================================
```

Figure 4.2 TLA+ specification for the policy MutualExclusion using Semaphore (MutexSem.tla) (con't..)

The Figure 4.1 provides a specification describing the behavior of the synchronization policy, mutual exclusion. A brief description of the constructs of the TLA+ specification language has been provided in the third chapter. The specification specifies the behavior of the MutualExclusion synchronization policy using the methods described in the section 4.2.3.3. The methods include the request method, the acquire method and the release method.

The predicate,

```
Request(p) == /\ stateP[p] = "idle"
              /\ stateP' = [stateP EXCEPT ![p] = "waiting"]
```

specifies the request method of the synchronization policy. *stateP* is a variable that represents the state of the processes. The statements containing unprimed variables represent the preconditions and the statements containing primed variables represent the change/action or the post-conditions. For instance, in the above predicate, the statement

```
stateP[p] = "idle"
```

indicates the precondition of the method, i.e., the unprimed *stateP* represents the precondition.

And the statement,

```
stateP' = [stateP EXCEPT ![p] = "waiting"]
```

indicates the action and the post condition, i.e., the primed *stateP* represents the postcondition.

Invariant is specified by the predicate,

```
Invariant == /\ \A p1, p2 \in Process: (stateP[p1] = "executing")
                                          /\ (stateP[p2] = "executing")
                                          => (p1=p2)
             /\ stateP \in [Process -> {"idle", "waiting", "executing",
            "releasing"}]
```

Other predicates of the specification can be explained in a similar manner.

Once the synchronization policies have been described, the component developers need to specify what synchronization policies have been used by the components to protect their methods against multiple accesses by the clients and the synchronization behavior of the components method. This is achieved by providing the component's synchronization contracts. The next section provides a discussion about creating the synchronization contracts for software components.

4.2.3.6 The Synchronization Contract

The component's synchronization contract should provide information about the synchronization policy utilized by each of the methods of the component, the method used to implement the synchronization policy, and the synchronization behavior's pre- and post- conditions, and invariants.

The synchronization contract consists of the following predicates:

1. Name of the component – It indicates the name of the component and can be represented as,

   Component: <*ComponentName*>

   where, ComponentName is the name of the component.

2. Name of the Interface – It indicates the name of the interface of the component. It can be represented in the following general predicate form:

   Interface: <*InterfaceName*>

   where, InterfaceName is the name of the interface of the component.

3. Name of the Synchronization Policy – It indicates the name of the synchronization policy from the synchronization policy catalog. It can be represented as,

   SynchronizationPolicy: <*SynchronizationPolicyName*>

   where, *synchronizationPolicyName* is the name of the synchronization policy from the synchronization policy catalog.

4. Synchronization Policy implementation method – It indicates the name of the method used to implement the synchronization policy. It can be represented as,

   SynchPolicyImplementation: <*SynchronizationPolicyImplementation*>

   where, *synchroniztionPolicyImplementation* represents the name of the implementation method.

5. Synchronization Precondition – It gives the pre-condition with respect to the synchronization behavior of the method of the component. It can be represented using the following predicate:

   SynchronizationPrecondition: <*precondition*>

   where, *precondition* represents the pre-condition of the synchronization behavior of the method.

6. Synchronization Invariant - It gives the invariant with respect to the synchronization behavior of the method of the component. It can be represented using the following predicate:

   SynchronizationInvariant (<*MethodName*>, <*Invariant*>)

   where, *MethodName* is the name of the method of the component and *Invariant* represents the synchronization behavior's invariant of the method.
7. Synchronization Action - It gives the action, with respect to the synchronization behavior, that is taken by the method of the component. It can be represented using the following predicate:

   SynchronizationAction: <*Action*>

   where, *Action* represents the synchronization action to be taken by the method.
8. Synchronization Postcondition - It gives the post-condition with respect to the synchronization behavior of the method of the component. It can be represented using the following predicate:

   SynchronizationPrecondition: <*postcondition*>

   where, *postcondition* represents the post-condition of the synchronization behavior of the method.

The synchronization contract takes the following general form:

Component: <*ComponentName*>
    Interface: <*InterfaceName*>
        MethodName: <*MethodName*>
        SynchronizationPolicy: <*SynchronizationPolicyName*>
        SynchPolicyImplementation:
                <*SynchronizationPolicyImplementation*>)
        SynchronizationPrecondition: <*precondition*>

SynchronizationInvariant: <*Invariant*>

SynchronizationAction: <*Action*>

SynchronizationPostcondition: <*postcondition*>

[MethodName: <*MethodName1*>

SynchronizationPolicy: <*SynchronizationPolicyName1*>

SynchPolicyImplementation:

<*SynchronizationPolicyImplementation1*>)

SynchronizationPrecondition: <*precondition1*>

SynchronizationInvariant: <*Invariant1*>

SyncrhonizationAction: <*Action1*>

SynchronizationPostcondition: <*postcondition1*>]

where,

<*ComponentName*> is the name of the component

<*InterfaceName*> is the name of the Interface

<*MethodName*> and <*MethodName1*> are the name of the Methods of the

<*SynchronizationPolicyName*> and <*SynchronizationPolicyName1*>

- are the name of the synchronization amongst the policies listed in the Synchronization Policy catalog.

<*SynchronizationPolicyImplementation*> and

<*SynchronizationPolicyImplementation*> - are the implementation methods used to implement the synchronization policies <SynchronizationPolicyName> and <SynchronizationPolicyName1>

<*precondition*> and <*precondition1*> - are the synchronization preconditions that must be satisfied by the clients before calling the methods <MethodName1> and <MethodName2>

<*invariant*> and <*invariant1*> - are the synchronization invariants for the methods <MethodName1> and <MethodName2>

<*Action*> and <*Action1*> - are the synchronization actions for the methods <MethodName1> and <MethodName2>

<*postcondition*> and <*postcondition1*> - are the synchronization post-conditions that the component ensures that it will satisfy if the methods <MethodName> and <MethodName1> are executed and they terminate.

4.2.3.7 Formal Representation of the Synchronization Contract

As indicated earlier, the thesis defines the behavior of the basic synchronization policies. It also defines the synchronization policies of the representative synchronization problem described earlier. The thesis uses TLA+ specification language [LAM02] for the purpose. It also provides the specification in TLA+ for the component's interface that formally defines the synchronization behavior of the software components. An example TLA+ specification for the component DocumentServer, with two methods 'createDocument' and 'DeleteDocument' of the Document Management System has been provided in the Figures 4.3 and 4.4.

```
------------------------------------------- MODULE DocumentServer ---------------------------
--------------------
EXTENDS TLC
CONSTANT cdclientSet, ddclientSet, Method, Document
VARIABLE stateC

cdpre(d) == \A c \in cdclientSet: stateC["createDocument", d, c] # "executing"
ddpre(d) == \A c \in cdclientSet: stateC["deleteDocument", d, c] # "executing"
                                  /\ stateC["createDocument", d, c] # "executing"

Init == /\ stateC = [m \in Method, d \in Document, c \in cdclientSet |-> "idle"]
---------------------------------------------------------------------------------------------
---------
createDocumentStart(d, c) == /\ IF /\ stateC["createDocument", d, c] = "idle"
                                   /\ cdpre(d)
                                THEN stateC' = [stateC EXCEPT !["createDocument", d, c] =
                                    "executing"]
                                ELSE UNCHANGED stateC

createDocumentEnd(d, c) == /\ IF stateC[<<"createDocument", d, c>>] = "executing"
                              THEN /\ stateC' = [stateC EXCEPT ![<<"createDocument", d, c>>]
                                     = "finished"]
                              ELSE /\ UNCHANGED <<stateC>>
deleteDocumentStart(d, c) == /\ IF /\ stateC["deleteDocument", d, c] = "idle"
                                    /\ ddpre(d)
                                THEN stateC' = [stateC EXCEPT !["deleteDocument", d, c] =
                                    "executing"]
                                ELSE UNCHANGED stateC
```

Figure 4.3 TLA+ specification for the component DocumentServer

```
deleteDocumentEnd(d, c) == /\ IF stateC[<<"deleteDocument", d, c>>] = "executing"
                              THEN /\ stateC' = [stateC EXCEPT ![<<"deleteDocument", d, c>>]
                                     = "finished"]
                              ELSE /\ UNCHANGED <<stateC>>

createDocument(d, c) == \/ createDocumentStart(d, c) \/ createDocumentEnd(d, c)
deleteDocument(d, c) == \/ deleteDocumentStart(d, c) \/ deleteDocumentEnd(d, c)

Next == \E d \in Document, c \in cdclientSet: \/ createDocument(d, c)
                                              \/ deleteDocument(d, c)

PNext == Next /\ Print(stateC, TRUE)
-------------------------------------------------------------------------------------------
Spec1 == Init /\ [][Next]_<<stateC>>
===========================================================================================
```

Figure 4.3 TLA+ specification for the component DocumentServer (con't.)

The section 4.2.3 provides a mechanism for formally specifying the synchronization level contract. The section contributes to one of the objectives of the thesis that deals with the creation of contracts of the software components at each of the four levels mentioned in section 1.3.1. In the next section, the thesis tries to address the other part of the objective, i.e., creating QoS level contracts for the software components.

4.2.4 The QoS Level Contract

The QoS level contract explicitly specifies the measure of the non-functional/quality of service properties of the component's interface. These specifications are used by the component users to make a selection decision. Among components with similar functional properties, the knowledge of quality attributes often could be the most significant information. It also helps in predicting the value of the quality of service properties of the overall system using some composition and decomposition rules discussed in [CHA03].

4.2.4.1 The QoS of a Software Component

As described earlier, the components in a component-based DCS are associated with the functional attributes as well as few non-functional attributes known as the QoS parameters. The syntactic and the semantic level contracts tell the component's users about the functionality it provides to the other components and the services it needs from other components in order to accomplish the services provided by the component. The synchronization behavior of the component is provided by the synchronization contract, describing the synchronization behavior of its methods. Providing non-functional attributes along with the functional attributes adds to the confidence level with which application developers can use the component. It helps the application developers to objectively compare the performance characteristics of multiple components with the same functionality, especially in quality-critical applications. Moreover, a given component may be used under diverse environments. The definition of environment here includes features, called environment variables, of the execution platform such as the CPU speed, the memory, the usage pattern, the process priority assigned to the execution of a component and the operating system used. These variables might have a significant impact on the QoS parameters of a software component, i.e., the QoS parameter's values associated with the component in an environment would not necessarily hold true in other environments. Hence, it becomes necessary for the component developer to specify the QoS parameters of a component as a function of the execution environment. The specification of the QoS parameter associated with the independently developed software components also aids in predicting the quality of the software system composed of these components.

For specifying and matching the value of various QoS parameters associated with software components, there is a need for a common semantics of these parameters. The UniFrame QoS framework [BRA02] provides a QoS catalog that standardizes the notion of the quality of the software components. It contains detailed descriptions of QoS parameters associated with the software components, including the metrics, the evaluation methodologies, the factors influencing these parameters and the interrelationships among these parameters. The value of these parameters form a part of the component's interface specifications.

A summary of the QoS parameters has been provided in this section. A detailed description of these parameters can be found in [BRA02].

The QoS catalog as per [BRA02] consists of:

1. Dependability – It is a static, application independent, QoS parameter associated with a component, which measures the components confidence that it is free from errors. Static parameters are the parameters whose value remains constant during run-time over different execution environments. Dependability is defined in terms

of the probability that the component is defect free. It allows comparison of different components and modifications to a component to increase its dependability. The value returned is a floating point value between [0, 1].

2. Security – It is a static, application independent QoS parameter associated with a component that gives a measure of the ability of the component to resist intrusions. It is measured in terms of the Minimum-Time -To-Intrusion (MinTTI), the shortest predicted period of time before any intrusion, and the Mean-Time-To-Intrusion (MTTI), the average time interval before an intrusion will occur [BRA02]. It allows security critical applications to decide whether a component can be used for the required functionality. The return value is a floating point value in the desired unit of time.
3. Adaptability – It is a measure of the ability of the component to tolerate changes in resources and user requirements. It is an application independent, static QoS factor associated with the component that measures the component's extent to which it adapts to change in its execution environment without any external intervention and hence, makes it useful for an application running in a dynamic environment. The measured value is a floating point value between [0, 1].
4. Maintainability – It is a measure of the ease with which a software system can be maintained, i.e., modified to correct faults, improve performance, or other attributes, or adapt to a changing environment. It is a static, application independent QoS parameter associated with the component. The evaluation of maintainability depends on the measure of the functionality of the component (FNC) [BRA02]. If the FNC is calculated in terms of function point, maintainability of the component is calculated in terms of function points per man month, if FNC is measure in terms of the lines of code, it is calculated in terms of lines of codes per man month, and if FNC is measured by the intrinsic effort required to develop the software component, the maintainability of the component is calculated as the ration of man months over man months.
5. Portability – It is a static, application independent QoS parameter associated with a software component that measures the ease with which a component can be

migrated to a new environment. It is the act of producing an executable version of a software unit or system in a new environment, based on an existing version. A component is said to be portable across a class of environments to the degree that the cost to transport and adapt it to a new environment to the new environment is less than the cost of redeveloping it. It returns a Boolean value that is true if it supports asynchronous calls and false if it does not.

6. Parallelism constraints – It is a static, application independent QoS parameter that is implemented at the method level, i.e., each method has a value for the parameter, and is used to determine whether a method can accept asynchronous calls from other components. The synchronous/asynchronous behavior of a component's method severely affects the response time of that component. It returns a Boolean value that is true if the method supports asynchronous calls.
7. Ordering constraints – It is a dynamic, i.e., its value vary during run-time according to the execution environment, application independent parameter that indicates whether the results returned by a component's method are in the proper order. It refers to the ordering of the results returned by a method of the component. It returns a Boolean value that is true if the method returns ordered results and is false if returns unordered results.
8. Priority – It is a static, application independent QoS parameter of the methods of a software component that measures the ability of the method to utilize priorities to serve clients in a multiple client scenario. It gives a Boolean value that is true if the method supports priority and false if it does not.
9. Throughput – It is a dynamic, application dependent QoS parameter associated with the methods of a software component that measures the number of requests a method can serve per unit time. It indicates the speed or the efficiency of the method of the component. It returns a floating point value in terms of number of responses per second.
10. Capacity – It is a dynamic, application independent QoS parameter associated with the methods of a software component that gives a measure of the number of

concurrent requests the component can serve at a given time instant. It gives a floating point value in terms of number if concurrent requests per unit time.

11. Turn-around Time – It is an application independent, dynamic QoS parameter associated with the methods of a software component and it measures the time taken by the method to return the result. It is defined as the time interval between the instant the method receives a request until the final result is generated. It returns a floating point value in milliseconds.
12. Availability – It is an application independent, dynamic QoS parameter that indicates the duration when a method of a software component is available to offer a particular service. It is calculated in terms of the percentage of time the method of a component is available to offer its services. It returns a floating point percentage value.

For the system developers to effectively match the QoS requirements of the desired system with that of a system formed from a set of software components from the component library, it is necessary for the component developers to formally represent the properties of the QoS parameter's and their values for each of the components. This helps in predicting the QoS properties of the end system. The formal representation or the contract of the QoS parameters should provide information about the parameter that is necessary for the system developers or the component users to make a decision whether to use the component or not. The next section discusses the mechanism for formally specifying the properties and the values of the QoS parameters.

The mechanism for formally describing the properties and the values of the QoS parameters can be provided in two main parts depending on if the property is component specific or it is independent of the component. They are as follows:

1. Mechanism to provide a formal description of the component independent properties of the QoS parameters.
2. Mechanism to specify the value of the component dependent properties of the QoS parameters for each of the components, also known as the QoS Contract.

4.2.4.2 Formal Specification of the Component-independent Properties of the QoS Parameters

The formal specification of the properties of the QoS parameters consists of describing the component independent properties and behaviors of the parameters, such as the name of the QoS parameters, domain dependency, rules for composition and decomposition, and the nature of the parameters. This helps the system developers to predict the properties and the behavior of the parameters of the resultant system.

The specification provides the formal representation of the following properties of the QoS parameters:

1. Name of the QoS parameter – Gives the name of the QoS parameter.
2. Domain Dependency/Domain of Usages – Indicates if the QoS parameter is domain dependent or not. A domain dependent QoS parameter is associated with the domain of usage that is a set of all the domains where the parameter is widely used, and a domain independent QoS parameter is applicable to all the domains. The domains considered in UniFrame are based on the OMG domain task force groups [DTF00], as described in [BRA02]. These domains include C4I (Command, Control, Communication and Intelligence), Finance, Healthcare, Life Sciences, Manufacturing, Space, Telecom, Transportation, E-Commerce, Real-time, and Utilities (gas, electric). For the domain dependent parameters the specification should indicate the list of domains of usage for the parameter.
3. Composability – Indicates whether the QoS parameter can be used, during the component composition process, to predict the QoS value of the end system using the QoS values of the individual components. If the parameter is composable then it is associated with a composition rule that gives a mechanism to calculate the value of the QoS parameter of the end system by composing the values of the parameter of the individual components, and a decomposition rule that gives a mechanism to calculate the values of the QoS parameter of the individual components by decomposing the value of the parameter of the end system. For instance, the composition rule for Dependability can be given by the formula:

$$\mathbf{D = Minimum(D_i)}$$

where, D is the Dependability measure of end system, D_i is the Dependability measure of component i (i=1,2,…n), and n is the number of components brought together to form the system.

Similarly, the decomposition rule for Dependability can be given by the formula:

$$\mathbf{D_i \geq D}$$

where, D_i is the Dependability measure of the components i (i = 1, 2, …, n), D is the Dependability measure of the end system, and n is the number of components brought together to form the system.

For the composable QoS parameters, the specification also needs to indicate the composition and decomposition rules.

4. Value Type – Indicates the type of the result returned by the evaluation procedure of the QoS parameter, which could be a floating point number, an integer or a Boolean value.
5. Static/Dynamic – Indicates whether the value of the parameter is constant or it varies during run-time depending on the operating environment such as CPU speed, Memory, process priority, etc., and the usage patterns such as the pattern of users and the user requests received by components.
6. Aliases – Indicates other prevalent equivalent names of the parameter, if any.
7. Nature of the QoS parameter – Indicates the category of the QoS parameters to which it belongs. The categories are decided based on the parameter's characteristics. The categories could be one of the following: Time-related parameters such as Turn-around-time, Importance–related parameters such as priority, Capacity-related parameters such as throughput and capacity, Integrity-related parameters such as accuracy, Safety-related parameters such as security and Auxiliary parameters such as portability, maintainability.

8. Component/Method Level – A QoS parameter is associated either with a component as a whole or with each methods of the component. This property indicates if the parameter is associated with the component as a whole or it is associated with each of the methods of the component. For a parameter associated with each of the methods of the component the QoS contract for this parameter consists of one or more subcontracts that gives the value of the parameters for each of the methods.

The formal specification of the component independent QoS parameter's properties consists of the following predicates:

1. Name of the QoS Parameters – A predicate for representing the name of the QoS parameter can be given as

   qosP_Name(<*QoS_parameter_name*>)

   where, QoS_parameter_name is of type String that indicates the name of the parameter, such as dependability, adaptability, security, etc.

2. Domain Dependency/Domain of Usages – This property of the QoS parameters can be represented as follows:

   qosP_Domain_Dependency(<*QoS_parameter_name*>, <*Boolean_value*>)

   where, QoS_parameter_name is of type String indicating the name of the parameter, and Boolean_value is of type Boolean which is true if the QoS parameter name is domain dependent else it is false.

   If the parameter is domain dependent, then the specification consists of a set of domain names where the parameter is widely used. It can be represented using the following predicate:

qosP_Domain_of_Usage(<*QoS_parameter_name*>, <*List_of_domains*>)

where, QoS_parameter_name is of type String indicating the name of the parameter, and List_of_domains represents a list of values of type String, giving the name of the domains where the parameter is widely used.

3. Composability – A predicate representation of the property can be given as:

qosP_Composability(<*QoS parameter name*>, <*Boolean value*>)

where, QoS parameter name is of type String indicating the name of the parameter, and Boolean value is of type Boolean which is true if the QoS parameter name is composable, else it is false.

The composable parameters specifications include another predicate that indicates the composition and decomposition rules as described below:

qosP_Composition_Rule(<*QoS_parameter_name*>, PC_1, PC_2) = <*composition_rule*>
qosP_Decomposition_Rule(<*QoS_parameter_name*>, PS) = <de*composition_rule*>

where, QoS_parameter_name is of type String indicating the name of the parameter, PC_1 and PC_2 are the values of the parameters for the components C_1 and C_2, PS is the value of the parameter for the end system S, composition_rule gives the rule that governs the composition of the parameter values for C_1 and C_2, and decomposition_rule gives the rule that governs the decomposition of the parameter value of the system S.

4. Value Type – It can be represented as follows:.

   qosP_ValueType(<*QoS parameter name*>, <*valueType*>)

   where, QoS parameter name is of type String indicating the name of the parameter, and valueType can be a floating point, an integer or a Boolean depending on the return type of the QoS parameter.

5. Static/Dynamic – It can be represented as follows:

   qosP_StaticDynamic(<*QoS_parameter_name*>, <*qos_Type*>)

   where, QoS_parameter_name is of type String indicating the name of the parameter, and qos_Type takes one of the following String values, static or dynamic, depending on the nature of the QoS parameter.

6. Aliases – It can be represented as follows:

   qosP_Aliases(<*QoS_parameter_name*>, <*list_of_alias_names*>)

   where, QoS_parameter_name is of type String indicating the name of the parameter, and list_of_alias_names is a list of values of type String that indicates the different names of the parameter.

7. Nature of the QoS parameter – It can be represented in the following predicate form:

   qosP_Nature(<*QoS_Parameter_Name*>, <*categories*>)

   where, QoS_parameter_name is of type String indicating the name of the parameter, and categories take one of the following String values: Time-related, Importance-related, Capacity-related, Integrity-related, or Safety-related, depending on the nature of the QoS parameter.

8. Component/Method – It takes the following predicate form:

   qosP_ComponentMethod(<*QoS Parameter Name*>, <*value*>)

   where, QoS parameter name is of type String indicating the name of the parameter, and value takes one of the following String values: Component or Method depending on whether the QoS parameter is associated with the component as a whole or is associated with each of the methods of the component.

The formal specification for two of the QoS Parameters has been provided in the Figures 4.5, and 4.6.

```
qosP_Name(Dependability)
qosP_Domain_Dependency(Dependability, false )
qosP_Composability(Dependability, true)
      qosP_Composition_Rule(Dependability, D1, D2)= Min(D1, D2)
      qosP_Decomposition_Rule(Dependability, D) = ≥ D
qosP_ValueType(Dependability, Float)
qosP_StaticDynamic(Dependability, Static)
qosP_Aliases(Dependability, [Maturity, Fault Hiding Ability, Degree
of Testing])
qosP_Nature(Dependability, Auxiliary )
qosP_ComponentMethod(Dependability, component)
```

Figure 4.4 Formal specification of the QoS Parameter Dependability

```
qosP_Name(Security)
qosP_Domain_Dependency(Security, true )
      qosP_Domain_of_Usage(Security, [E-commerce, C4I])
qosP_Composability(Security, true)
      qosP_Composition_Rule(Security, S1, S2)= Min(S1, S2)
      qosP_Decomposition_Rule(Security, S) = ≥ S
qosP_ValueType(Security, Float)
qosP_StaticDynamic(Security, Static)
qosP_Aliases(Security, [Maturity, Fault Hiding Ability, Degree of
Testing])
qosP_Nature(Security, Auxiliary )
qosP_ComponentMethod(Security, component)
```

Figure 4.5 Formal specification of the QoS Parameter Dependability

4.2.4.3 Formal Specification of the Component-dependent Properties of the QoS Parameters, The QoS Contract

The QoS contract describes the component specific properties of the QoS parameters such as, the effect of the execution environment and the usage pattern on the value of the parameters and the value of the parameter for each of the components or the methods of each of the components. The component dependent property that indicates the effect of the execution environment and usage pattern depends on whether the component is a static parameter or a dynamic parameter, and the property that gives the value of the parameter depends on whether the parameter is applicable at the component level or the method level, i.e. whether the parameter value is associated with the component as a whole or it is associated with each of the methods of the component. Specifying the effect of environment on the values of the parameters helps in determining the possible change is the values due to the executing environment and the value of the QoS parameter for a component or its methods helps the users of the component in comparing the values of the parameters with that of the requirements of the desired system. The contract at the two levels has the following predicate form:

1. Static/dynamic Property level

    qosP_EnvironmentVariables(<*QoS_parameter_name*>, <*Component_name*>, <*list_of_environment_variables*>)

    where, Component_name is of type String that indicates the name of the component whose component dependent properties are being specified, QoS_parameter_name is of type String indicating the name of the parameter, and list_of_environment_variables takes a list of values of type String that indicate the list of environment variables that effect the value of the parameter.

2. The Component/method level

   If the parameter is applicable at the component level then,

   qosP_Value(<*QoS_parameter_name*>, <*Component_name*>, <*value*>)

   If the parameter is applicable at the method level then,

   qosP_Value(<*QoS_parameter_name*>, <*Component_name*>, <*method_name*>, <*value*>)

   where, QoS_parameter_name is of type String indicating the name of the parameter, Component_name is of type String that indicates the name of the component whose component dependent properties are being specified, method name is of type String that represents the name of the method of the component, and value is the value of the parameter of type indicated in the return type of the parameter.

Depending on whether the parameter is a static or a dynamic parameter, the QoS contract for a component at this level may consist of zero or more subcontracts, and depending on whether the parameter is applicable at the component or the method level, the QoS contract for a component at this level may consist of one or more subcontracts. The next section describes the subcontracts at both the levels.

4.2.4.3.1 Subcontract at the Static/Dynamic Level

As discussed, the value of the QoS parameters may remain constant or may vary with the change in the environment. If the value remains constant, it is considered to be a static parameter and if it varies, it is a dynamic parameter. A subcontract revealing this aspect of the QoS parameters is needed to determine whether the value of the parameter can be improved by changes to the operating environment. The subcontracts provide the links to the graphs indicating the effect of change in the execution environment and the usage patterns. It takes the following predicate form,

> qosP_environmentVariables_effect(<*QoS_parameter_name*>, <*Environment_Variable_name*>, <*url_change_EV*>)
> [qosP_environmentVariables_effect(<*QoS_parameter_name*>, <*Environment_Variable_name1*>, <*url_change_EV1*>)]
> qosP_usagePattern(<*QoS_paramete_ name*>, <*url_change_UP*>)

where, QoS_parameter_name is of type String indicating the name of the parameter, Environment_Variable name is the name of the environment variable, url_change_EV is a value of type String that gives the url for the graphs that indicate the effect of change in the environments variables and url_change_UP is a value of type String that gives the url for the graphs that indicate the effect of change in the usage pattern.

4.2.4.3.2 Subcontract at the Component/Method Level

Each QoS parameter has a value of the type indicated by the return value and the value, as discussed in the previous sections, is either associated with the component as a whole or with each of the methods of the component. For example, the QoS parameter Dependability gives a measure of the confidence that the component is free from errors, whereas the QoS parameter turn around time gives the time taken by a component's method to return the results. The subcontract at this level consists of either one subcontract indicating the values of the component level QoS parameters for a component or more than one subcontract giving the values of the method level QoS parameters for each of the methods in the component.

qosP_value(<*QoS_parameter_name*>, <*component_name*>, <*value*>)

or,

qosP_value(<*QoS_parameter_name*>, <*component_name*>, <*method_name1*>, <*value1*>)
[qosP_value(<*QoS_parameter_name*>, <*component_name*>, <*method_name2*>, <*value2*>)]

where, QoS_parameter_name is of type String indicating the name of the parameter, component_name is of type String that indicates the name of the component whose component dependent properties are being specified, method_name1 and method_name2 are of types String that represents the name of the methods of the component, and value1 and value2 are the values of the parameter of type indicated in the return type of the parameter.

4.2.4.4 Formal Representation of the QoS Contract

The QoS contract, explained in the previous section, can be formally specified in any form of logic programming such as Prolog. The knowledge thus provided forms a part of the database in the UniFrame knowledge base, which can then be queried by the system developers for searching the desired software components.

The Figure 4.7 gives an example of the QoS Contract of a component C_1 in the predicate form. Component C_1 provides the methods M_{p1}, and M_{p2} and requires methods M_{r1} and M_{r2}.

```
/* QoS Contract for the Component C1. It follows the UQoS standards.

qosP_value(C1, dependability, 0.62) /* value should be in between [0,
1]
qosP_value(C1, security, 1000) /* Unit of time: milliseconds
qosP_value(C1, adaptability, 0.75) /* value should be in between [0,
1]
qosP_value(C1, maintainability, 10) /* ratio of man months over man
months
qosP_value(C1, portability, 1)

qosP_value(C1, parallelismConstraints, Mp1, 1)
qosP_value(C1, parallelismConstraints, Mp2, 0)
qosP_value(C1, parallelismConstraints, Mr1, 1)
qosP_value(C1, parallelismConstraints, Mr2, 0)

qosP_environmentVariables(C1, orderingConstraints,
[networkTransmission, networkProtocol])
qosP_environmentVariables_effect(C1, orderingConstraints,
networkTransmission, <url:graph for effect of networkTransmission>)
qosP_environmentVariables_effect(C1, orderingConstraints,
networkProtocol, <url:graph for effect of networkProtocol>)

qosP_value(C1, orderingConstraints, Mp1, 0)
qosP_value(C1, orderingConstraints, Mp2, 1)
qosP_value(C1, orderingConstraints, Mr1, 0)
qosP_value(C1, orderingConstraints, Mr2, 1)

qosP_value(C1, priority, Mp1, 0)
qosP_value(C1, priority, Mp2, 0)
qosP_value(C1, priority, Mr1, 0)
qosP_value(C1, priority, Mr2, 0)
```

Figure 4.6 QoS Contract for the component C_1

```
/* QoS Contract for component C1 continued…

qosP_environmentVariables(C1, throughput, [algorithm, cpuSpeed,
memory, hardware])
qosP_environmentVariables_effect(C1, throughput, algorithm,
<url:graph for effect of algorithm>)
qosP_environmentVariables_effect(C1, throughput, cpuSpeed, <url:graph
for effect of cpuSpeed>)
qosP_environmentVariables_effect(C1, throughput, memory, <url:graph
for effect of memory>)
qosP_environmentVariables_effect(C1, throughput, hardware, <url:graph
for effect of hardware>)
qosP_value(C1, throughput, Mp1, 6) /* number of responses per second
qosP_value(C1, throughput, Mp2, 8)
qosP_value(C1, throughput, Mr1, 6)
qosP_value(C1, throughput, Mr2, 2)

qosP_environmentVariables(C1, capacity, [threads, cpuSpeed, memory,
hardware])
qosP_environmentVariables_effect(C1, capacity, threads, <url:graph
for effect of threads>)
qosP_environmentVariables_effect(C1, capacity, cpuSpeed, <url:graph
for effect of cpuSpeed>)
qosP_environmentVariables_effect(C1, capacity, memory, <url:graph for
effect of memory>)
qosP_environmentVariables_effect(C1, capacity, hardware, <url:graph
for effect of hardware>)
qosP_value(C1, capacity, Mp1, 1) /* number of request per second
qosP_value(C1, capacity, Mp2, 10)
qosP_value(C1, capacity, Mr1, 5)
qosP_value(C1, capacity, Mr2, 4)
```

Figure 4.6 QoS Contract for the component C_1 (con't.)

```
/* QoS Contract for component $C_1$ continued…

qosP_environmentVariables($C_1$, turn_around_time, [algorithm, threads,
cpuSpeed, memory, load_on_system, os_access_policy])
qosP_environmentVariables_effect($C_1$, turn_around_time, algorithm,
<url:graph for effect of algorithm>)
qosP_environmentVariables_effect($C_1$, turn_around_time, cpuSpeed,
<url:graph for effect of cpuSpeed>)
qosP_environmentVariables_effect($C_1$, turn_around_time, memory,
<url:graph for effect of memory>)
qosP_environmentVariables_effect($C_1$, turn_around_time,
load_on_system, <url:graph for effect of load_on_system>)
qosP_environmentVariables_effect($C_1$, turn_around_time,
os_access_policy, <url:graph for effect of os_access_policy>)
qosP_value($C_1$, turn_around_time, $M_{p1}$, 100) /* value is measured in
milliseconds
qosP_value($C_1$, turn_around_time, $M_{p2}$, 50)
qosP_value($C_1$, turn_around_time, $M_{r1}$, 70)
qosP_value($C_1$, turn_around_time, $M_{r2}$, 20)

qosP_environmentVariables($C_1$, availability, [dependability,
fault_tolerance, efficiency_repairMechanism])
qosP_environmentVariables_effect($C_1$, availability, dependability,
<url:graph for effect of dependability>)
qosP_environmentVariables_effect($C_1$, availability, fault_tolerance,
<url:graph for effect of fault_tolerance >)
qosP_environmentVariables_effect($C_1$, availability,
efficiency_repairMechanism, <url:graph for effect of
efficiency_repairMechanism>)
qosP_value($C_1$, availability, $M_{p1}$, 70) /* Value is in percentage
qosP_value($C_1$, availability, $M_{p2}$, 50)
qosP_value($C_1$, availability, $M_{r1}$, 40)
qosP_value($C_1$, availability, $M_{r2}$, 80)
```

Figure 4.6 QoS Contract for the component C_1 (con't.)

The chapter provided a discussion on the proposed mechanism for specifying the Synchronization and QoS levels contracts. This chapter grows as one of the main contributions to the thesis. It addresses one of the objectives of the thesis, i.e., providing a mechanism for the formal specification of the components at the synchronization and the QoS level. It forms the basis for the next objective, i.e., defining a set of matches and providing a mechanism for matching the contracts at the two levels. The next chapter deals with these two objectives.

5. MATCHING THE CONTRACTS

The formal specifications or the contracts explained in the previous chapter form a part of the UniFrame knowledge base. To build a DCS using CBSD, the system developer gives a query and the components library is searched for the components that satisfy the query. For searching the software components that satisfy the user's query, the component's contracts, stored in the knowledge base, have to be matched at the four levels: the syntactic level, the semantic level, the synchronization and the QoS level, in order to determine which components are substitutable and which of them are compatible with other components of a system. The substitutability of the software components refers to the ability of the components to replace each other without change in the behavior of the system and the compatibility refers to the ability of two components to work properly together if connected, i.e., all exchanged messages and data between them are understood by each other. The component contracts are matched based on certain matching criteria and rules, defined in this chapter.

[ZAR96] defines a set of matching criteria for the syntactic and the semantic levels, whereas this chapter provides a set of matching criteria for the synchronization and the QoS levels. The chapter contributes to the second and the third objectives of the thesis. The chapter in the next section provides a brief discussion on the matching criteria defined in [ZAR96] for the two levels, the syntactic and the semantic levels. The sections 5.3 and 5.4 provide a set of matching criteria for the synchronization and the QoS levels and the mechanism to match these specifications.

The matching of the software component contracts is done separately at the four levels. Since the contracts exist at all the four levels, it is logical to match the contracts at the four levels separately and also it makes the problem of matching the contracts simple. The matching is performed in a hierarchical manner, where matching at the semantic level requires that the components have already been matched at the syntactic level, matching at the synchronization level requires that the components have already been matched for the syntactic and semantic compatibility and so on. This assumption is the consequence of the fact that even if the components are compatible at the semantic, synchronization and the QoS level, if two components are not compatible syntactically, they will not be able to interact with each other. Similarly, even if the components are compatible at the syntactic, the synchronization and the QoS levels, the components will not be able to produce valid results. And hence, matching at every level assumes that the components have been matched and are compatible at the lower levels. The next few sections provide discussion on the matching criteria at the four levels.

5.1 Syntactic Level Matching Criteria

[ZAR96] argues that the syntactic/signature matching of software components boils down to type matching of the method's parameters and the return values. It defines a set of exact and relaxed matches for matching the signature of the methods provided by the components. The relaxed matches allow reordering of elements in a tuple, uncurrying of arguments to a method, renaming of type constructors and instantiation of type variables.

In signature matching, the matches are expressed in terms of whether a transformation can be applied to the signature such that the results are equal. A transformation can be defined as a function from types to types, for example, a function that reorders elements in a tuple.

To define the set of matching criteria [ZAR96] assumes that type either belongs to type variable (TypeVar) or type constructor (tyCon) and also defines type equality and variable substitution as follows.

Type Equality ($=_T$) is defined as,
$t =_T t'$ iff,

1. They are lexically identical type variables or
2. $t = tyCon(t_1,\ldots, t_n)$, $t' = tyCon'(t_1',\ldots,t_n')$,
   $tyCon = tyCon'$, and forall i, such that $1<i<n$,
   $t_i =_T t_i'$.

Variable substitution is represented as $[t'/\alpha]t$ i.e., the type that results from replacing all occurrences of the type variable α in t with t′, provided no variables in t′ occur in t. A sequence of substitutions is right associative. If t′ is just a variable then, $[t'/\alpha]t$ is simply variable renaming.

Given the definition of type equality, a type of a method from a component library, t_l, and the type of a query, t_q, [ZAR96] defines a generic signature match predicate as follows:

$M(t_l, t_q)$ = There exists a transformation pair $T = (T_l, T_q)$, such that $T_l(t_l)$ T R $T_q(t_q)$

where, R is some relationship, such as equality, between types, that will be defined in the specific matching predicates, and T_l and T_q are transformations that are applied to the components and the query component types, respectively.

The following are the categories defined by [ZAR96] as the syntactic level matching criteria:

1. Exact Match –

   $\text{match}_E\ (t_l, t_q)$ = There exists a sequence of variable renaming, V, such that $V\ t_l =_T t_q$

   The formula suggests that two method types match exactly if they match modulo variable renaming.

2. Relaxed Matches – Relaxed match is achieved by applying transformations or variable substitutions to the type expressions. It consists of the following subcategories:

   a. Transformation Relaxation – In this a match is achieved by applying transformation to the type expressions. Transformation includes renaming type constructors, uncurrying the functions, changing the order of types in a tuple, and changing the order of arguments to a method. The matches can be defined as,

      i. Type constructor renaming – It follows renaming the user defined type constructors and built in types with different name

         $\text{match}_{tyCon}\ (t_l, t_q)$ = There exists a sequence of type constructor renaming, V_{TC}, such that $\text{match}_E(VTC\ t_l, t_q)$

      ii. Uncurrying Functions – the uncurried version of a method, with multiple arguments, has a type $(t_1 * \ldots * t_{n-1}) \text{ -> } t_n$, while the corresponding curried version, has a type $t_1 \text{ -> } \ldots \text{-> } t_{n-1} \text{ -> } t_n$. Uncurry match is defined by applying the uncurry transformation to both query and component types. The uncurry transformation, UC, produces an uncurried version of a given type. It is defined as,

$$UC(t) = \begin{cases} (t_1 * \ldots * t_{n-1}) \text{ -> } t_n & \text{if } t = t_1 \text{ ->} \ldots \text{->} t_{n-1} \text{->} t_n,\ n>2 \\ t & \text{otherwise} \end{cases}$$

The uncurry match is defined as:

$match_{uncurry}(t_l, t_q) = match_E (UC(t_l), UC(t_q))$

It takes two uncurried function types and determines whether their corresponding arguments match.

iii. Reordering Tuples – It allows matching on types that differ only in their order of the arguments.

A reorder transformation, T_σ, for a method whose argument is a tuple ($t = (t_1 * \ldots . * t_{n-1}) -> t_n$), can be defined as a permutation σ, which can applied to the tuple. σ is a bijection with domain and range 1…n-1 such that $T_\sigma(t) = (t_{\sigma(1)} * \ldots * t_{\sigma(n-1)}) -> t_n$.

A reorder Match is defined as,

$match_{reorder}(t_l, t_q)$ = there exist a reorder transformation T_σ such that $match_E(T_\sigma(t_l), t_q)$

i.e. a t_l matches with t_q, is the argument types of t_l can be reordered so that the types match exactly.

b. Partial relaxations – It takes care of the situations where a specific query type is an instantiation of a more general function, or where the user asks for a general type that does not match with anything in the components exactly. Partial relaxation matches are achieved by using variable substitution to define the partial ordering, based on the generality of the types. For example, $\alpha -> \alpha$ is a generalization of infinitely many types such as int -> int and $(int*\beta) -> (int*\beta)$, using the variable substitutions $[int/\alpha]$ and $[(int*\beta)/\alpha]$, respectively. Based on this, matches defined as,

i. Generalized match – It can be defined as,

$match_{gen}(t_l, t_q) = t_l \geq t_q$

A component type matches with a query type if the component type is more general than the query type.

ii. Specialized match – It can be defined as,

$$match_{spcl}(t_l, t_q) = t_l \leq t_q$$

A component type matches with a query type if the query type is more general than the component type.

[ZAR96], apart from defining the matches also provides few properties of the matches. [ZAR96] discusses a function signature matcher implemented for Standard ML (SML) functions. This section provided an overview of the match criteria as defined in [ZAR96, ZAR95]. A detail description of the matches can be found in [ZAR96, ZAR95]. The next section provides a brief description of the semantic level matching criteria defined in [ZAR96, ZAR97].

5.2 Semantic Level Matching Criteria

[ZAR96, ZAR97] defines a set of matching criteria for the semantic level that has been discussed briefly in this section. The matching at this level is done assuming that the component's specifications have been matched at the syntactic level.

For a function specification, S, and a query Q, the matches are defined in terms of their function's pre- conditions, S_{pre} and Q_{pre} and post-conditions S_{post} and Q_{post}, respectively.

A generic pre/post match predicate is represented as,

$match_{pre/post}(S, Q) = (Q_{pre}\ R_1\ S_{pre})\ R_2\ (S_{post}\ R_3\ Q_{post})$..........................(Eq. 5.1)

The relations R_1 and R_3 relate the behavioral pre- and the post- conditions respectively, and hence are either equivalence (<=>) or implication (=>), as defined below in the specific match predicates, but need not be the same. In most cases, it is required that both relations (R_1 and R_3) hold, and so R_2 is usually conjunction (/\), but may also be implication (=>).

The following are the matches defined by [ZAR96] for matching the specification of software components:

1. Exact Pre/Post Match – It is a strict relation with R_1 and R_3 replaced with equivalence relationships in (Eq. 5.1). And R_2 replaced with conjunction relationship that indicates that both (Q_{pre} R_1 S_{pre}) and (S_{post} R_3 Q_{post}) should be satisfied. It can be represented as follows:

   $$match_{E\text{-}pre/post}(S, Q) = (Q_{pre} \Leftrightarrow S_{pre}) \wedge (S_{post} \Leftrightarrow Q_{post})$$

2. Plug-in Match – Usually, it is not the case that the pre- and the post- condition match exactly with each other and hence, there is a need for relaxing the relations R_1 and R_3 in (Eq. 5.1). Plug in Match is achieved by replacing R_1 and R_3 with =>, i.e., an implication. It is represented as:

   $$match_{plug\text{-}in}(S, Q) = (Q_{pre} \Rightarrow S_{pre}) \wedge (S_{post} \Rightarrow Q_{post})$$

   Under this, Q is matched by any specification S whose pre-condition is weaker and whose post-condition is stronger.

3. Plug-in post match – It is yet another relaxation, considering only the post-conditions. It follows the argument that most pre-conditions could be satisfied by adding an additional check before calling the method and usually, the match is concerned only with the effects of methods, and hence, only the post-condition needs to be matched. Plug-in post match is achieved by dropping Q_{pre} and S_{pre} and

replacing R_3 with implication relation, => in (Eq. 5.1). It is represented as follows:

$$match_{plug\text{-}in\text{-}post}(S, Q) = (S_{post} \Rightarrow Q_{post})$$

4. Weak Post Match – It is an even weaker match, where R_3 and R_2 are replaced with implication relationship, => but Q_{pre} is dropped. It is used when precondition of S helps in proving the relationship between S_{post} and Q_{post}. It is represented as:

$$match_{weak\text{-}post}(S,Q) = S_{pre} \Rightarrow (S_{post} \Rightarrow Q_{post})$$

[ZAR 96] also provides properties of the matches at this level and the matches are classified on the basis of whether they are equivalence matches, partial order matches, or neither. It also provides a mechanism to relate those matches. [ZAR96] uses Larch/ML [WIN93], a Larch interface language for the ML programming language, to specify the semantic information of the methods of a component. A detailed description of these can be found in [ZAR96].

5.3 Synchronization Level Matching Criteria

The sections 5.2 and 5.3 describe the matching of the component's specifications at the syntactic and the semantic levels. This section describes the matching of the component's specification at the synchronization level, and contributes to one of the objectives of this thesis.

The component developers develop a software component by making certain assumptions about the environment and about the other software components that the component interacts with. Apart from the behavioral assumptions, the developers make assumptions about the synchronization behavior of the system as well. The components, while interacting, need to be synchronized in order to handle multiple concurrent calls on the methods of the components. The assumptions made by the software developers, about the synchronization policies, may or may not match, which may result in invalid or wrong outputs. Hence, there is a need to match the synchronization behavior of the components specified by the component developers.

Matching of software components at the synchronization level is achieved by matching the behavior of the synchronization policies, utilized by the software components to synchronize their interfaces and matching the synchronization behavior of the component's interfaces. Hence, the synchronization level matching of the software components can be divided into two main categories:

1. Matching of the behaviors of the synchronization policies – It helps in determining the relationships between the synchronization policy utilized by the software component and that utilized by the query component. The synchronization policies may be related to each other through one of the following relationships:
    a. Equivalence – This relationship indicates that the two synchronization policies are behaviorally equivalent, i.e., replacing one synchronization policy with another in a component's implementation does not change the synchronization behavior of the component.
    b. Implication – This relationship indicates that a synchronization policy, SP_1 implies the other policy SP_2, and hence, the synchronization behavior of the component's interface does not change if SP_2 is replaced with SP_1 , but the synchronization behavior of the component's interface may change if the synchronization policy SP_1 is replaced with SP_2.

2. Matching of the synchronization behavior of the component's interface – It is needed as the components can utilize more that one synchronization policy for protecting different resources within the component. It helps in determining if the system's synchronization behavior remains unchanged if a component,

constituting that system, is replaced by another component. The process requires matching the synchronization specifications of the replaced component to that of the replacing component for substitutability and matching the specifications of the replacing component with that of the other components that form the system for compatibility. The matched component should be such that replacing with the component does not change the behavior of the system and there is no deadlock or starvation. The two types of checks with respect to synchronization can be defined as follows:

a. Substitutability of the software components – With respect to synchronization, substitutability can be defined as the ability of two components to replace each other without changing the synchronization behavior of the system that is formed using that component.
b. Compatibility of the software components – With respect to synchronization, compatibility can be defined as the ability of two components to interoperate and synchronize properly when brought together to form a system.

The next two sections provide a detailed description of Matching the synchronization policy and the Matching the synchronization behaviors of the component's interface. They include defining the match criteria and providing a mechanism to match synchronization specifications.

5.3.1 Matching the Synchronization Policy

As discussed earlier, the specifications of the synchronization policies can be matched for equivalence and implication. This can be achieved by matching the preconditions, the invariants and the post-conditions of the methods supported by the synchronization policies.

For instance, assume there are two synchronization policy's specifications SP_1 and SP_2 and the policies support the methods M_{SP1i} and $M_{SP2i,}$ respectively. The pre-conditions of M_{SP1i} and M_{SP2i} are given by M_{SP1i}Pre and M_{SP2i}Pre, respectively. The post-conditions of M_{SP1i} and M_{SP2i} are given by M_{SP1i}Post and M_{SP2i}Post, respectively. The invariants of M_{SP1i} and M_{SP1i} are given by M_{SP1i}Inv and M_{SP1i}Inv, respectively. Matching of the specifications of the two synchronization policies can be defined as follows:

1. Equivalence:

   $SP_1 \equiv SP_2$, where, $\equiv$ represents the equivalence relationship

   iff,

   a. M_{SP1i}Pre <=> M_{SP2i}Pre, and
   b. M_{SP1i}Inv <=> M_{SP2i}Inv, and
   c. M_{SP1i}Post <=> M_{SP2i}Post

2. Implication:

   SP_1 => SP_2

   iff,

   a. M_{SP1i}Pre => M_{SP2i}Pre, and
   b. M_{SP1i}Inv => M_{SP2i}Inv, and
   c. M_{SP1i}Post => M_{SP2i}Post

   Or,

   SP_2 => SP_1

   iff,

   a. M_{SP2i}Pre => M_{SP1i}Pre, and
   b. M_{SP2i}Inv => M_{SP1i}Inv, and
   c. M_{SP2i}Post => M_{SP1i}Post

As explained in the Mutual Exclusion example, the pre-, post- conditions and invariants can be represented in TLA+ as shown in the Figure 4.1. The specifications written in TLA+ can be matched for the equivalence and the implication relationship using the PROPERTY and the THEOREM constructs [Chapter three, Section 3.2.1]. An example for matching two TLA+ specifications of synchronization policies has been provided below:

```
(******************************************************************************************)
(* This module tries to prove that the specifications of Mutex.tla and                  *)
(* MutexSem.tla are equivalent                                                          *)
(******************************************************************************************)

--------------------------------- MODULE PropertyMutex ------------------------------------
(******************************************************************************************)
(* The EXTEND statement includes in the current module all the definitions              *)
(* and declarations from the module MutexSem.                                           *)
(******************************************************************************************)

EXTENDS MutexSem
-------------------------------------------------------------------------------------------
(******************************************************************************************)
(* These statements specify the different operations of Mutual Exclusion                *)
(******************************************************************************************)

RequestP(p) ==  /\ stateP[p] = "idle"
                /\ stateP' = [stateP EXCEPT ![p] = "waiting"]

AcquireP(p) == /\ stateP[p] = "waiting"
               /\ \A p1 \in Process: stateP[p] # "executing"
               /\ stateP' = [stateP EXCEPT ![p] = "executing"]

ReleaseP(p) == /\ stateP[p] = "executing"
               /\ stateP' = [stateP EXCEPT ![p] = "released"]

exitingP(p) == /\ stateP[p] = "released"
               /\ stateP' = [stateP EXCEPT ![p] = "idle"]
```

Figure 5.1 PropertyMutex.tla

```
NextP == \E p \in Process : \/ RequestP(p) \/ AcquireP(p)
                      \/ ReleaseP(p)\/ exitingP(p)

\*PNext == Next /\ Print(stateP, TRUE)
---------------------------------------------------------------------------------------------
SpecP == Init /\ [][NextP]_<<rSem>>

(*********************************************************************************************)
(* This theorem asserts that formulas Spec and SpecP are equivalent.  The                  *)
(* symbol <=> , which can also be typed as \equiv , is typeset as an                       *)
(* equivalence symbole (a three-lined equals sign).                                        *)
(*********************************************************************************************)
THEOREM Spec => SpecP
=============================================================================================
```

Figure 5.1 PropertyMutex.tla (con't.)

```
(*******************************************************************************************)
(* This is a TLC configuration file for testing that the specificaition of                *)
(* MutexSem implies the specification of Mutex                                             *)
(*******************************************************************************************)
CONSTANT
      Process = {1,2}

SPECIFICATION Spec
(*******************************************************************************************)
(* This statement tells TLC that it is to take formula Spec as                             *)
(* the specification it is checking.                                                       *)
(*******************************************************************************************)

PROPERTY SpecP
(*******************************************************************************************)
(* This statement tells TLC to check that the specification                                *)
(* implies the property SpecP.  (In TLA, a specification is also a property.              *)
(*******************************************************************************************)
```

Figure 5.2 PropertyMutex.cfg

```
(******************************************************************************************)
(* This module tries to prove that the specifications of Mutex.tla and                    *)
(* MutexSem.tla are equivalent by exploring the states of Mutex                           *)
(******************************************************************************************)

--------------------------- MODULE PropertyMutexSem----------------------------------------
(******************************************************************************************)
(* The EXTEND statement includes in the current module all the definitions                *)
(* and declarations from the module Mutex.                                                *)
(******************************************************************************************)

EXTENDS Mutex, Semaphore
VARIABLE rSem

InitP == /\ rSem = 1
         /\ stateP = [p \in Process |-> "idle"]
-------------------------------------------------------------------------------------------
(******************************************************************************************)
(* These statements specify the different operations of Mutual Exclusion                  *)
(* as implemented in Mutex.tla and those implemented in MutexSem.tla                      *)
(******************************************************************************************)

RequestP(p) ==  /\ stateP[p] = "idle"
                /\ stateP' = [stateP EXCEPT ![p] = "waiting"]
                /\ UNCHANGED rSem

AcquireP(p) == /\ stateP[p] = "waiting"
               /\ P(rSem)
               /\ stateP' = [stateP EXCEPT ![p] = "executing"]
```

Figure 5.3 PropertyMutexSem.tla

```
ReleaseP(p) == /\ stateP[p] = "executing"
               /\ stateP' = [stateP EXCEPT ![p] = "released"]
               /\ V(rSem)

exitingP(p) == /\ stateP[p] = "released"
               /\ stateP' = [stateP EXCEPT ![p] = "idle"]
               /\ UNCHANGED rSem

Request1(p) == /\ Request(p)
               /\ UNCHANGED rSem

Acquire1(p) == /\ Acquire(p)
               /\ UNCHANGED rSem

Release1(p) == /\ Release(p)
               /\ UNCHANGED rSem

exiting1(p) == /\ Exiting(p)
               /\ UNCHANGED rSem

NextP == \E p \in Process: \/ RequestP(p) \/ AcquireP(p)
                           \/ ReleaseP(p)\/ exitingP(p)

Next1 == \E p \in Process: \/ Request1(p) \/ Acquire1(p)
                           \/ Release1(p)\/ exiting1(p)
-------------------------------------------------------------------------------------------
SpecP == InitP /\ [][NextP]_<<rSem>>
Spec == InitP /\ [][Next1]_<<stateP>>
THEOREM SpecP => Spec
===========================================================================================
```

Figure 5.3 PropertyMutexSem.tla (con't.)

```
(******************************************************************************************)
(* This is a TLC configuration file for testing that the specificaition of                *)
(* MutexSem implies the specification of Mutex                                            *)
(******************************************************************************************)
CONSTANT
      Process = {1,2}

SPECIFICATION Spec
(******************************************************************************************)
(* This statement tells TLC that it is to take formula Spec as                            *)
(* the specification it is checking.                                                      *)
(******************************************************************************************)

PROPERTY SpecP
(******************************************************************************************)
(* This statement tells TLC to check that the specification                               *)
(* implies the property SpecP.  (In TLA, a specification is also a property.              *)
(******************************************************************************************)
```

Figure 5.4 PropertyMutexSem.cfg

The specification PropertyMutex.tla (Figure 5.1) provides the behavior of the synchronization policy Mutual Exclusion. It specifies the three operations of Mutual Exclusion, request, acquire, and release. It extends the MutexSem.tla [Figure 4.2], and tries to prove that the formula *Spec*, the specification of the module MutexSem.tla, implies *SpecP*. This is given by providing a theorem in PropertyMutex.tla (Figure 5.1) and a property in PropertyMutex.cfg (Figure 5.2). The statements are,

```
THEOREM SpecP => Spec
```
and,
```
PROPERTY SpecP
```

As discussed in chapter three, the THEOREM statement indicates that the SpecP should be satisfied by all the states and the PROPERTY statement is specified to check some property. The specifications PropertyMutex.tla and the PropertyMutex.cfg allows the user to determine if there exist an implication relationship between the behavior of the policy Mutual exclusion and the behavior of the policy mutual exclusion implemented using semaphores. In this example, the TLC model checker explores all the possible states that could be generated for MutexSem.tla (Figure 4.2) and checks if all the states satisfy the property *SpecP*. If any of the states violate the property, the TLC checker stops exploring further states and gives an error; otherwise it generates all possible states.

The above example does not check for an equivalence relationship between the policies. To determine if there is an equivalence relationship between the policies, the specification of Mutex.tla (Figure 4.1) has to be checked against the specifications of the MutexSem.tla (Figure 4.2). This is achieved in the Figures 5.3 and 5.4. The PropertyMutexSem.tla implements the methods of the MutexSem.tla (Figure 4.2) and extends Mutex.tla (Figure 4.1).

If both the specifications satisfy their respective properties, the policies are said to be equivalent.

5.3.2 Matching the Synchronization behavior of the Component's Interface

As discussed earlier, there is a need for matching the synchronization behavior of the component's interface for substitutability and compatibility. This is achieved by matching the synchronization pre-conditions, invariants and post-conditions of the methods supported by the components. The following subsections define the matching criteria for matching the synchronization behavior of the component's interface.

5.3.2.1 Generalized and Specialized Matches

The matching of a component's synchronization behavior with that of a query component can be divided into two main categories:

1. Generalized Match – A generalized match consists of matching the synchronization policies utilized by the components and the synchronization behavior of the component's interface but not the synchronization policy's implementation method.

   A generic match predicate for the generalized matching of the synchronization behavior of a query component, QC, and the component, C, that supports methods M_{QCi} and M_i, respectively, can be represented as,

$$\begin{aligned} match_{gen}(C, QC) = {} & \wedge (SP\ R_1\ SP_{QC}) \\ & \wedge \text{ for each method } M_i \text{ and } M_{QCi}\ (SP_{pre}(M_i)\ R_2 \\ & SP_{pre}(M_{QCi}))\ R_3\ (SP_{post}(M_i)\ R_4\ SP_{post}(M_{QCi})) \end{aligned} \qquad \text{……….(Eq. 5.2)}$$

   where, SP and SP_{QC} are the synchronization policies utilized by the component C and the query component QC, respectively. $SP_{pre}(M_i)$ and $SP_{pre}(M_{QCi})$ are the synchronization preconditions of the methods M_i and M_{QCi}, respectively, and $SP_{post}(M_i)$ and $SP_{post}(M_{QCi})$ are the synchronization post-conditions of the methods M_i and M_{QCi}, respectively. The values SP, SP_{QC}, $SP_{pre}(M_i)$, $SP_{pre}(M_{QCi})$,

$SP_{post}(M_i)$, and $SP_{post}(M_{QCi})$ are derived from their respective components synchronization contracts. R_1 is the relationship that exists between the Synchronization polices SP and SP_{QC} and is derived from the results of matching the synchronization policies as discussed in the previous section. It can take the values '≡' or '=>', depending on if there is an equivalence relationship or an implication relationship between them, respectively. R_2 and R_4 are usually the equivalence (≡) or the implication (=>) relation that exists between the synchronization pre- and the post- conditions and R_3 is usually a conjunction. If R_1, R_2 and R_4 are all '≡', and the value R_3 is a conjunction, the synchronization behaviors of C and QC, exactly match with each other (discussed below). If any of R_1, R_2 and R_4 is not '≡', then it results in a relaxed match (discussed below).

2. Specialized Match – A specialized match consists of matching the synchronization policy utilized by the software components, the synchronization policy's implementation method and the synchronization behavior of the component's interface.

   For the same example as in generalized match, a specialized match be represented as,

$$\begin{aligned} match_{spec}(C, C_{QC}) = {} & \wedge (SP\ R_1\ SP_{QC}) \\ & \wedge (SP_{impl}\ R_2\ SP_{QCimpl}) \\ & \wedge \text{ for each method } M_i \text{ and } M_{QCi}\ (SP_{pre}(M_i)\ R_3 \\ & \quad SP_{pre}(M_{QCi}))\ R_4\ (SP_{post}(M_i)\ R_5\ SP_{post}(M_{QCi})) \end{aligned} \quad \text{....(Eq. 5.3)}$$

   where, the definitions of SP, SP_{QC}, $SP_{pre}(M_i)$ and $SP_{pre}(M_{QCi})$ remains the same as described in the example for generalized matching. SP_{impl} and SP_{QCimpl} are the implementation methods used to implement the synchronization policies SP and SP_{QC} respectively for the respective components, and are derived from the respective component's synchronization contracts. R_1 is the relationship that exists between the Synchronization polices SP and SP_{QC}, derived from matching the synchronization policies. R_3 and R_5 are usually equivalence or implication relation between the synchronization pre- and the post- conditions and R4 is usually a conjunction. R_2 is '=' for the specialized match. If R_1, R_3 and R_4 are all '≡', and the value R_3 is a conjunction, the synchronization behaviors of C and QC, exactly match with each other (discussed below). If any of R_1, R_3 and R_5 is not '≡', then it results in a relaxed match (discussed below).

A substitutability check on a query component QC and the component C is done by matching the synchronization behavior of the respective provided and the required interfaces, where as a compatibility check between the query component QC and the component C is performed by matching the provided interface(s) of component C with the required interface(s) of the query component QC.

The generalized and specialized matches described above can further be divided into exact and relaxed matches. They have the following predicate form:

1. Generalized:
   a. Exact Match – In exact match, there exist an equivalence relationship between the synchronization policies of the components and the synchronization pre- and post- conditions of the methods of the components. Taking the same example of a query component QC and a component C, for exact match, the relations, R_1, R_2, R_3, and R_4 in (Eq. 5.2) have the values ≡, <=>, /\, and <=>, respectively. The following is the predicate form of exact match:

$$\text{exactMatch}_{gen}(C_1, QC) = \wedge\ (SP \equiv SP_{QC})$$
$$\wedge\ \text{for each method } M_i \text{ and } M_{QCi}\ (SP_{pre}(M_i) \Leftrightarrow SP_{pre}(M_{QCi})) \wedge (SP_{post}(M_i) \Leftrightarrow SP_{post}(M_{QCi}))$$

Exact match can be explained using the following simple example:
A component S provides an implementation of a stack, a last in first out (LIFO) data structure, with three basic operations, namely, create, pop and push operation. Another component Q provides an implementation of a queue, a first in first out data structure, with three basic operations, namely, createQ, insertElement and deleteElement. Assuming that components interfaces are being accessed by multiple clients, the component needs to protect their resources from the clients. The following are the synchronization contract for a few of the operations of the components:

Component: S

Interface: Stack

MethodName: create(C)

Signature: void create(integer size)

SynchronizationPolicy: <Mutual Exclusion>

SynchPolicyImplementation: <Semaphores>)

SynchronizationPrecondition: forall C_1 in clients[] ~execute(create, C_1)

SynchronizationInvariant: NONE

SynchronizationAction: execute(create, C)

SynchronizationPostcondition: execute(create, C)

Figure 5.5 Synchronization Contract for component S

Component: Q

Interface: Queue

MethodName: createQ(C_2)

Signature: void createQ(integer size)

SynchronizationPolicy: <Mutual Exclusion>

SynchPolicyImplementation: <Mutexes>)

SynchronizationPrecondition: forall C_3 in clients[] ~execute(createQ, C_3)

SynchronizationInvariant: NONE

SynchronizationAction: execute(createQ, C_2)

SynchronizationPostcondition: execute(createQ, C_2)

Figure 5.6 Synchronization Contract for Component Q

where, C is the client calling the operation

client[] is the set of all the clients

execute(<*MethodName*>, <*ClientName*>) indicates true if the Client with the name *ClientName* is currently executing method with the name *MethodName*.

For the above examples the generalized exact match comes out to be true for the method create with the following relationships:

$$\text{exactMatch}_{gen}(S, Q) = \wedge\ (SP_S \equiv SP_Q) \wedge (SP_{pre}(\text{create}) \Longleftrightarrow SP_{pre}(\text{createQ})) \wedge (SP_{post}(\text{create}) \Longleftrightarrow SP_{post}(\text{createQ}))$$

b. Relaxed Match – In relaxed match, there exist an implication relationship between the synchronization policies of the components and the synchronization pre- and post- conditions of the methods of the components. Taking the same example of a query component QC and a component C, for relaxed match, the relations, R_1, R_2, R_3, and R_4 in (Eq. 5.2) have the values => or ≡, => or ≡, /\, and => or ≡, respectively. Any combinations of the given values of R_1, R_2, R_3, and R_4 except '≡' for R_1, R_2, and R_4 results in relaxed match. If R_1, R_2 and R_4 are all '≡', and the value R_4 is a conjunction, the synchronization behaviors of C and QC, exactly match with each other. The following is one of the predicate forms of relaxed match, assuming that QC requires the methods M_{QCi} and C provides the methods M_i:

$$\begin{aligned} \text{relaxedMatch}_{gen}(C, C_{QC}) = {} & \wedge (SP \Rightarrow SP_{QC}) \\ & \wedge \text{ for each method } M_i \text{ and } M_{QCi} \\ & \quad (SP_{pre}(M_i) \Rightarrow SP_{pre}(M_{QCi})) \wedge \\ & \quad (SP_{post}(M_i) \Rightarrow SP_{post}(M_{QCi})) \end{aligned}$$

For the example explained in the Figures 5.5 and 5.6, if the synchronization precondition of component Q in the Figure 5.6 is changed to

SynchronizationPrecondition: ~execute(createQ, C)

i.e., the client calling the function createQ is not executing, then the components S and Q match with respect to the generalized relaxed match with the following relationships:

relaxedMatch$_{gen}$ (S, Q) = $\wedge$ (SP$_S$ => SP$_Q$)
$\wedge$ (SP$_{pre}$(create) => SP$_{pre}$(createQ)) $\wedge$
(SP$_{post}$(create) <=> SP$_{post}$(createQ))

2. Specialized:
   a. Exact Match – The exact match for the specialized case, is the same as exact match for the generalized case except that it also checks if the implementation method of the synchronization policy match or not. Taking the same notations, QC for the query component and C for the component, for the exact match for the specialized case, the relations R$_1$, R$_2$, R$_3$, R$_4$, and R$_5$ in (Eq. 5.3) would take the values, ≡, =, <=>, $\wedge$, and <=>, respectively. The predicate form of the same is as follows:

   exactMatch$_{spec}$ (C, C$_{QC}$) = $\wedge$ (SP ≡ SP$_{QC}$)
   $\wedge$ (SP$_{impl}$ = SPQC$_{impl}$)
   $\wedge$ for each method M$_i$ and M$_{QCi}$ (SP$_{pre}$(M$_i$) <=> SP$_{pre}$(M$_{QCi}$)) $\wedge$ (SP$_{post}$(M$_i$) <=> SP$_{post}$(M$_{QCi}$))

   In the example explained in the Figures 5.5 and 5.6 the methods create and createQ do not match with respect to specialized exact match as the synchronization policy implementation does not match.

   b. Relaxed Match – The relaxed match for the specialized case is also the same as that of the generalized except that the implementation method for the synchronization policy has to be matched for the query component QC and the component C. Taking the same notations, QC for the query component and C for the component, for the exact match for the specialized case, the relations R$_1$, R$_2$, R$_3$, R$_4$, and R$_5$ in (Eq. 5.3) would take the values, => or ≡, = or ≠, => or ≡, $\wedge$, and => or ≡, respectively. Any combinations of the given values of R$_1$, R$_2$, R$_3$, R$_4$ and R$_5$ except '≡' for R$_1$,

R_3, and R_5 results in relaxed match. If R_1, R_3 and R_5 are all '≡', the value R_4 is a conjunction, and the value R_2 is '=', the synchronization behaviors of C and QC, exactly match with each other The following is one of the predicate forms of the relaxed match, assuming that QC requires the methods M_{QCi} and C provides the methods M_i:

$$\begin{aligned} relaxedMatch_{spec}(C, C_{QC}) = {} & \wedge (SP \Rightarrow SP_{QC}) \\ & \wedge (SP_{impl} \neq SP_{QCimpl}) \\ & \wedge \text{ for each method } M_i \text{ and } M_{QCi}\ (SP_{pre}(M_i) \Rightarrow \\ & \quad SP_{pre}(M_{QCi})) \wedge (SP_{post}(M_i) \Rightarrow SP_{post}(M_{QCi})) \end{aligned}$$

In the example explained in the Figures 5.5 and 5.6 the methods create and createQ match with respect to specialized relaxed match.

The section provided a set of matching criteria for matching the synchronization contracts of two software components. It also provided examples to illustrate the same. The next section discusses the matching criteria for matching the QoS level contracts of the software components.

5.4 QoS Level Matching Criteria

As discussed earlier, matching of the QoS level contract is performed assuming that the component and the query component have already been matched and are compatible with respect to the other three levels, i.e., the syntactic, the semantic, and the synchronization levels.

Matching of QoS parameters of the software components can be divided into two main categories based on the match of the values of the QoS parameters:

1. Exact Match – A component is said to be an exact match of a query component, if the value of the QoS parameters of the component is equal to or better than that of the query component. "Better" is one of the logical operators, greater than equal to, equal to, or less that equal to. It is defined differently for different QoS parameters and it depends on the nature of the QoS parameter. For instance, dependability of a component is better than that of a query component if its value is greater than or equal to the value of dependability for the query component, whereas, turn around time of a method of a component is better than that of a query component if its value is less than or equal to the value of turn around time of the respective method of the query component. For the parameters that have subcontracts, the match should satisfy for each of the methods specified. Table 5.1 provides the list defining "better" for each of the QoS parameters.

   The notation used to represent the values of a QoS parameter are: Initials of the parameter name in capitals with a subscript 'qc' represents the value of the parameter for the query component and with a subscript 'c' represents the value of the parameter for the component.

QoS Parameter	**Definition of “better” for Exact Match**
Dependability	$\mathbf{D_c \geq D_{qc}}$
Security	$\mathbf{S_c \geq S_{qc}}$
Adaptability	$\mathbf{A_c \geq A_{qc}}$
Maintainability	$\mathbf{M_c \geq M_{qc}}$
Portability	$\mathbf{P_c = P_{qc}}$
Parallelism Constraints	$\mathbf{PC_c = PC_{qc}}$
Ordering Constraints	$\mathbf{OC_c = OC_{qc}}$
Priority	$\mathbf{P_c = P_{qc}}$
Throughput	$\mathbf{T_c \geq T_{qc}}$
Capacity	$\mathbf{C_c \geq C_{qc}}$
Turn Around Time	$\mathbf{TAT_c \leq TAT_{qc}}$
Availability	$\mathbf{Av_c \geq Av_{qc}}$

Table 5.1 Definition of “Better” for Exact Match

2. Relaxed Match – A component is said to be a relaxed match of a query component, if value of the QoS parameters of the component is close to the value of the parameter for the query component. A “close” value, for some of the parameters, is defined by the user of the component based on the requirements of the user. The user gives the measure of the deviation that is allowed in the values of the parameters while performing the matching. For parameters that return a Boolean value, such as portability, parallelism constraints, ordering constraints, and priority, “close” takes a value 'greater than equal to'. For other parameters the value of “close” is defined by the user, through the acceptance rates. For instance, a user can specify the query that has a requirement for the value of dependability to be 0.75 but is ready to accept a component that provides 5% less dependability

than that of the required value, where 5% is the acceptance rate. Providing a relaxed match, broadens the scope of finding a component and is especially useful in searching for a domain specific component, where a particular QoS parameter is given more importance than others. Table 5.2 gives the definition of "close" for relaxed matches.

The acceptance rate in Table 5.2 is represented as $acr_{<parameterInitials>}$. The notation for the value of the QoS parameter is the same as Table 5.1.

QoS Parameter	**Definition of "better" for Exact Match**
Dependability	$D_c \geq D_{qc} - acr_d/100*D_{qc}$
Security	$S_c \geq S_{qc} - acr_s/100*S_{qc}$
Adaptability	$A_c \geq A_{qc} - acr_a/100*A_{qc}$
Maintainability	$M_c \geq M_{qc} - acr_m/100*M_{qc}$
Portability	$P_c = P_{qc}$
Parallelism Constraints	$PC_c = PC_{qc}$
Ordering Constraints	$OC_c = OC_{qc}$
Priority	$Pr_c = Pr_{qc}$
Throughput	$T_c \geq T_{qc} – acr_t/100*T_{qc}$
Capacity	$C_c \geq C_{qc} – acr_c/100*C_{qc}$
Turn Around Time	$TAT_c \leq TAT_{qc} + acr_{tat}/100*TAT_{qc}$
Availability	$Av_c \geq Av_{qc} - acr_{av}/100*Av_{qc}$

Table 5.2 Definition of "Close" for Relaxed Match

The requirement of a user for a particular QoS is usually based on the domain of the system, where it is going to be used. A user may or may not be concerned with the value of all the QoS parameters in the UQoS catalog. For example, for real time applications a user may be more interested in a time-related parameter such as turn around time than the parallelism constraints. A user may or may not specify QoS requirements for all the QoS parameters, and not all parameters have to be matched against the requirements. And hence, exact and relaxed matches defined for the two cases, 1) match all the parameters, and 2) match the specified parameters as follows:

1. Exact Match –
    a. Match all the parameters – match each of the QoS parameters from the component's QoS contract with that of the query component for a "better" value of the parameter.
    b. Match the specified parameters – match only the specified parameters of the components QoS contract with that of the query component for a "better" value of the parameter.
2. Relaxed Match –
    a. Match all the parameters – match all the QoS parameters from the component's QoS contract with that of the query component for a "close" value of the parameter.
    b. Match the specified parameters – match only the specified parameters of the components QoS contract with that of the query component for a "close" value of the parameter.

Example: Following is the QoS Contract for a query component QC.

/* QoS Contract for the Query Component QC. It follows the UQoS standards.

qosP_value(QC, dependability, 0.65)

qosP_value(QC, portability, 0)

qosP_value(QC, parallelismConstraints, M_{p1}, 1)

qosP_value(QC, parallelismConstraints, M_{p2}, 1)

qosP_value(QC, parallelismConstraints, M_{r1}, 0)

qosP_value(QC, parallelismConstraints, M_{r2}, 0)

qosP_value(QC, throughput, M_{p1}, 4)

qosP_value(QC, throughput, M_{p2}, 3)

qosP_value(QC, throughput, M_{r1}, 2)

qosP_value(QC, throughput, M_{r2}, 8)

qosP_value(QC, turn_around_time, M_{p1}, 100)

qosP_value(QC, turn_around_time, M_{p2}, 60)

qosP_value(QC, turn_around_time, M_{r1}, 100)

qosP_value(QC, turn_around_time, M_{r2}, 50)

Figure 5.7 QoS Contract for the query component QC

Referring to the QoS contract of the component C1 in the Figure 4.3, following are the exact and relaxed matches with respect to each of the parameters for the query component QC:

Parameter	Match
Dependability	**No**
Security	**Yes**
Adaptability	**Yes**
Maintainability	**Yes**
Portability	**Yes**
Parallelism Constraints	**No**
Ordering Constraints	**Yes**
Priority	**Yes**
Throughput	**No**
Capacity	**Yes**
Turn Around Time	**Yes**
Availability	**Yes**

Table 5.3 Results of Exact Match for each of the parameters of C and QC

Relaxed Match with the acceptance rates 5% for all the parameters:

Parameter	**Match**
Dependability	**Yes**
Security	**Yes**
Adaptability	**Yes**
Maintainability	**Yes**
Portability	**Yes**
Parallelism Constraints	**No**
Ordering Constraints	**Yes**
Priority	**Yes**
Throughput	**No**
Capacity	**Yes**
Turn Around Time	**Yes**
Availability	**Yes**

Table 5.4 Results of Relaxed Match for each of the parameters of C and QC

The chapter provided a discussion on the set of matching criteria at the synchronization and the QoS levels. The matching of the synchronization contracts has been implemented using the TLA+ specification language as discussed in the previous sections. The matching of QoS contracts can be implemented using logic programming. The next section gives a case study to illustrate the contracts and their matching at the two levels.

6. VALIDATION AND VERIFICATION

The chapters four and five discussed the specification of a software component's properties, creation of contracts for the interface of the software components and matching the contracts to determine the substitutability and compatibility of software components. This chapter provides a case study to illustrate the creation of the contracts and matching them. The chapters four and five contribute to the first and he second objectives of the thesis, where as, this chapter contributes to the fourth objective, i.e., providing a case study to illustrate the contracts and the matching of the contracts. The next section describes the Document Management System in detail. The sections 6.2, 6.3, 6.4, and 6.5 give the syntactic, the semantic, the synchronization and the QoS level contracts for the components of the Document Management System.

6.1 Document Management System

The case study for demonstrating the key concepts of the thesis consists of a Document Management System. The system belongs to the document management domain and provides the following basic document management services:

1) Validating a user – The service allows the users to validate their username and password to check if he/she is allowed to perform the following functions on the documents: create a new document, delete an existing document, read an existing document, and write an existing document.

2) Create a new document – This service enables the users to create a new document.
3) Deleting an existing document – The services enables the users to delete an existing document.
4) Read an existing document – It allows the users to read an existing document, but restricts them to change the document.
5) Write a document – It allows the users to read as well as write/update the document.
6) List the documents – It allows the user to list the names of the documents in the folder.

The Document Management System according to the UniFrame Generative Domain Model (UGDM) consists of the following subsystems: 1) User Subsystem, 2) User Validation Subsystem, and 3) Document Subsystem. The user subsystem acts as the interface to the users and provides all the functionalities needed by the users of the Document Management System. It provides the user with the functionalities and requires some of the functionalities from the other two subsystems. The user validation subsystem provides the service of validating a user to the User subsystem. The document subsystem provides the functionalities needed to manage the documents. The functionalities provided by the user validation subsystem and the document subsystem is required by the user subsystem to perform its functionalities.

The Figure 6.1 provides the feature diagram of the document management system. It indicates the family of systems that can be formed from its subsystems.

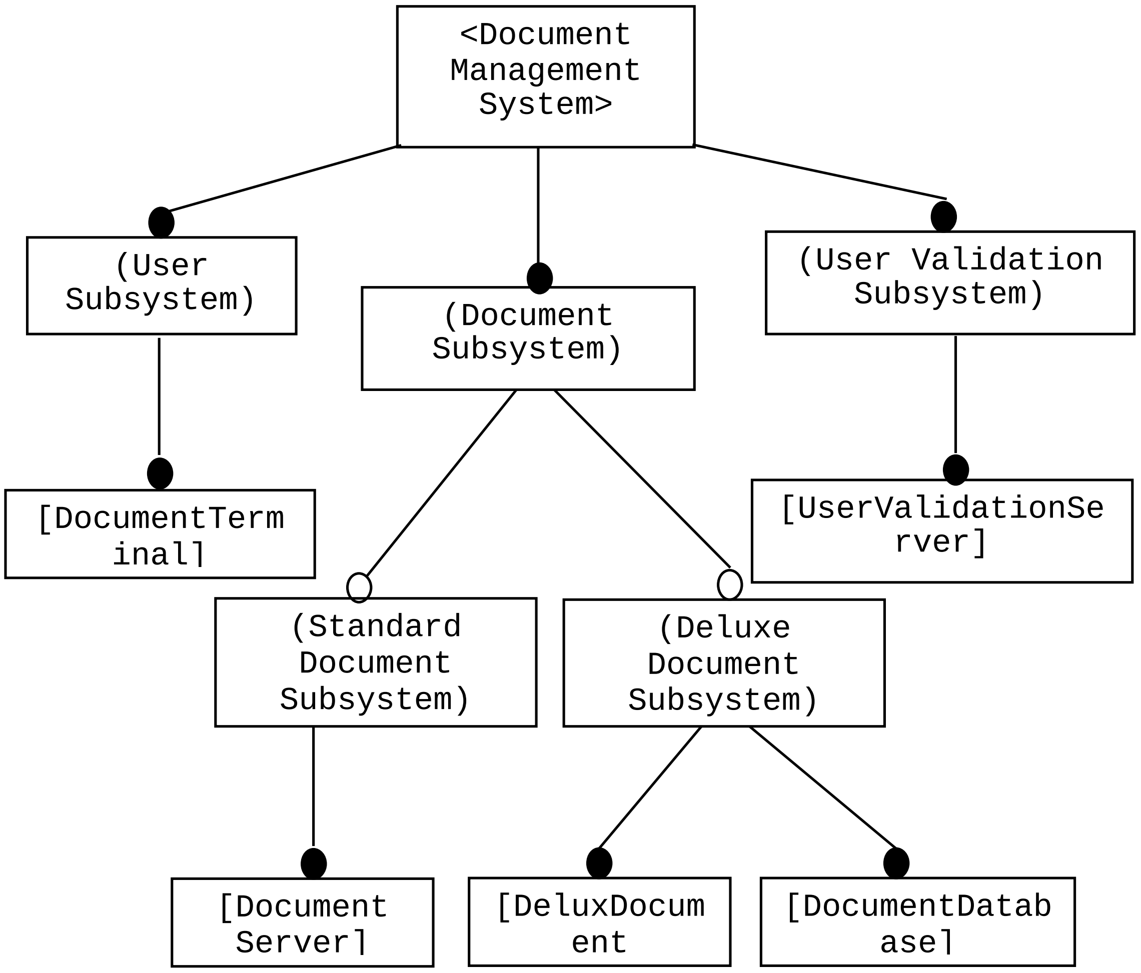

Figure 6.1 Feature diagram of the Document Management System

The leaf nodes in the above diagram represent the abstract components. Abstract components are the guidelines for developing reusable concrete components. The abstract components connected to a subsystem with a line and a solid circle represents required components where as those with hollow circles represents a choice between the abstract components. For instance, in the above feature diagram, the Standard Document Subsystem and the Deluxe Document Subsystem are the choices for the Document Subsystem of the Document Management System.

Two types of systems can be formed from the subsystems shown in the feature diagram (Figure 6.1):

1. A Simple Document Management System consisting of a DocumentTerminal, a DocumentServer and a UserValidationServer.
2. A Deluxe Document Management System consisting of a DocumentTerminal, a UserValidationServer, a DeluxeDocumentServer, and a DocumentDatabase.

The thesis used Simple Document Management System for illustrating the key concepts of the objectives of the thesis. The next section gives the description of the functionalities provided by the components of the Simple Document Management System.

6.1.1 Description of the Components of Simple Document Management System

The Simple Document Management System consists of three components as mentioned in the previous section. A component diagram of the Simple Document Management System is shown in the Figure 6.2. It also shows the interactions between the individual components:

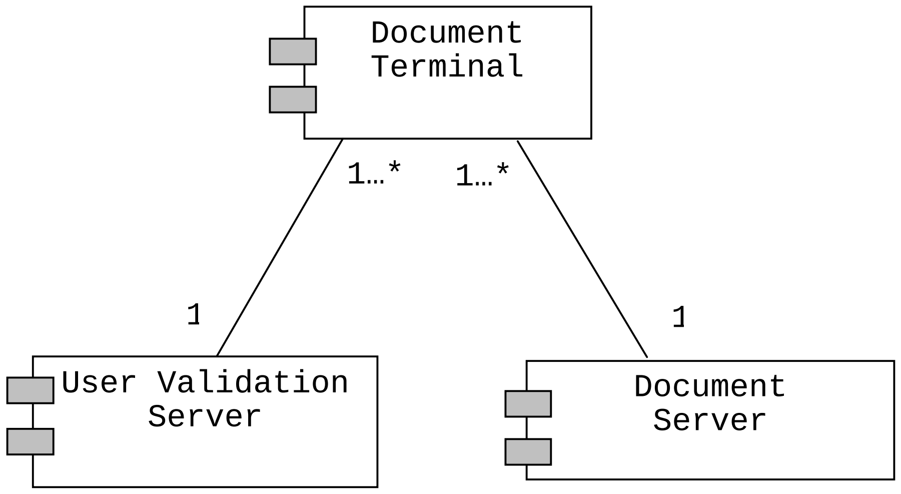

Figure 6.2 Component diagram of Simple Document Management System

The following describes the functionalities of the three components along with the description of their interfaces and collaborator components. Collaborator components are other components that a component interacts with. There are two types of collaborators, preprocessing collaborators – other components on which this component depends, and post-collaborators – other components that may depend on this component.

The three components of the system are being described here:

1. Document Server – It provides basic document services to the document terminal like create document, get document, write document and delete document.

   Interfaces:

   Provided: IDocumentManagement (Section 6.1.1.1)

   Required: None

   Collaborator components:

   Preprocessing: DocumentTerminal

   Postprocessing: NONE

2. User Validation Server – It provides user validation service for the users of the system like validate user.

   Interfaces:
   Provided: IValidation (Section 6.1.1.1)
   Required: NONE

   Collaborator components:
   Preprocessing: DocumentTerminal
   Postprocessing: NONE

3. Document Terminal – It provides a Graphical User Interface for the users of the system. The component gets the services from the other two components (User Validation Server, Document Server) and provides those services to the user.

   Interfaces:
   Provided: IDocumentTerminal
   Required: IDocumentTerminal

   Collaborator components:
   Preprocessing: NONE
   Postprocessing: DocumentValidationServer, DocumentServer

6.1.1.1 Interface Model

The section describes the methods of the interfaces used by the components of the Simple Document Management System. The interfaces are:

1. IDocumentManagement – It consists of the following methods –
    a. void createDocument (String documentName) – A method that allows users to create a new document with the name specified by the argument 'documentName'.
    b. void deleteDocument (String documentName) – A method that allows users to delete a document with the name specified by the argument 'documentName'.
    c. void readDocument (String documentName) – A method that allows users to open a document, with the name specified by the argument 'documentName'. The contents cannot be modified or updated.
    d. void writeDocument (String documentName) – A method that allows users to open a document, with the name specified by the argument 'documentName'. The contents can be modified and stored back into the database.
    e. String[] listDocuments () – A method that returns a list of all the documents in the folder to the users.
2. IValidationServer – It consists of the following methods –
    a. Boolean validateUser (String username, String password) – A method that validates a user by validating the username and the password entered by the user.
3. IDocumentTerminal – It consists of the following methods –
    a. void createDocument (String documentName) – A method that allows users to create a new document with the name specified by the argument 'documentName'.
    b. void deleteDocument (String documentName) – A method that allows the users to delete a document with the name specified by the argument 'documentName'.
    c. void readDocument (String documentName) – A method that allows the users to open a document, with the name specified by the argument 'documentName'. The contents cannot be modified or updated.
    d. void writeDocument (String documentName) – A method that allows the users to opens the document, with the name specified by the argument

'documentName'. The contents can be modified and stored back into the database.

e. String[] listDocuments () – A method that returns a list of all the documents in the folder to the users.
f. Boolean validateUser (String username, String password) – A method that validates a user by validating the username and the password entered by the user.

6.1.1.2 Component Interactions

The user interacts with the Document Terminal, by providing the username and the password. The Document Terminal uses the services of User Validation Server to validate the user. Once validated, the user is allowed to send requests to the Document Terminal. The Document Terminal uses the services of the Document Server to fulfill the requests of the users and returns back the results to the users.

The next few sections provide the contracts at the four levels discussed in previous chapters, the syntactic, the semantic, the synchronization and the QoS levels, for few of the components of the simple document management system.

6.2 Syntactic Level Contract

The syntactic level contracts for the components DocumentTerminal, DocumentServer and UserValidationServer are as follows:

1. Component: DocumentTerminal

   Interface: IDocumentTerminal

   a. createDocument
      - Signature: void createDocument(String documentName)

- Function Name: createDocument
- Return Type: void
- Parameter Type: String

b. deleteDocument
   - Signature: void deleteDocument(String documentName)
   - Function Name: deleteDocument
   - Return Type: void
   - Parameter Type: String

c. readDocument
   - Signature: DocumentType readDocument(String documentName)
   - Function Name: readDocument
   - Return Type: DocumentType
   - Parameter Type: String

d. writeDocument
   - Signature: void writeDocument(String documentName)
   - Function Name: writeDocument
   - Return Type: void
   - Parameter Type: String

e. listDocuments
   - Signature: String [] listDocuments()
   - Function Name: listDocument
   - Return Type: String []
   - Parameter Type: NONE

f. validate
   - Signature: Boolean validate(String username, String password)
   - Function Name: validate
   - Return Type: Boolean
   - Parameter Type: String, String

2. Component: UserValidationServer

   Interface: IValidation

   a. validate
      - Signature: Boolean validate(String username, String password)
      - Function Name: validate
      - Return Type: Boolean
      - Parameter Type: String, String

3. Component: DocumentServer

   Interface: IDocumentManagement

   a. createDocument
      - Signature: void createDocument(String documentName)
      - Function Name: createDocument
      - Return Type: void
      - Parameter Type: String
   b. deleteDocument
      - Signature: void deleteDocument(String documentName)
      - Function Name: deleteDocument
      - Return Type: void
      - Parameter Type: String
   c. readDocument
      - Signature: DocumentType readDocument(String documentName)
      - Function Name: readDocument
      - Return Type: DocumentType
      - Parameter Type: String
   d. writeDocument
      - Signature: void writeDocument(String documentName)
      - Function Name: writeDocument
      - Return Type: void
      - Parameter Type: String

e. listDocuments
   - Signature: String [] listDocuments()
   - Function Name: listDocument
   - Return Type: String []
   - Parameter Type: NONE

The formal specification of the syntactic level has been discussed in [ZAR96].

6.3 Semantic Level Contract

The semantic level contracts for the three components, DocumentTerminal, UserValidationServer and DocumentServer of the simple document management system has been provided in this section.

1. Component: DocumentTerminal

   Interface: IDocumentTerminal

   a. createDocument – creates a new Document with the name specified by the argument 'documentName'.
      - Signature: void createDocument(String documentName)
      - Precondition: ~existDocument(documentName)
      - Invariant: Forall fn in filenames[]: (~changeName(fn) /\ ~changeContent (fn))
      - Post-condition: filenames[] = filenames[] + documentName

   b. deleteDocument – deletes the document with name specified by the argument 'documentName'.
      - Signature: void deleteDocument(String documentName)
      - Precondition: existDocument(documentName)

- Invariant: Forall fn in (filename[] – documentName) ~changeName(fn) /\ ~changeContent(fn)
- Post-condition: filenames[] = filenames[] – documentName

c. readDocument – The document with the name specified by the argument 'documentName' is retrieved and the user allowed to read the document.
   - Signature: DocumentType readDocument(String documentName)
   - Precondition: existDocument(documentName)
   - Invariant: Forall fn in filename[]: ~changeName(fn) /\ ~changeContent(fn)
   - Post-condition: return(documentName)

d. writeDocument – This function opens the document with the name specified by the argument documentName, and the user is allowed to read as well as update the document.
   - Signature: void writeDocument(String documentName, String content)
   - Precondition: existDocument(documentName)
   - Invariant: Forall fn in filename[]: ~changeName(fn) /\ for all fn in (filename[] – documentName): ~changeContent(fn)
   - Post-condition: changeContent(documentName, content)

e. listDocuments – This method is used to list the name of all the documents in the folder.
   - Signature: String [] listDocuments()
   - Precondition: True
   - Invariant: Forall fn in filename[]: ~changeName(fn) /\ ~changeContent(fn)
   - Post-condition: return(filenames[])

f. Validate – the component verifies the user by checking the username and the password entered by the user.
    - Signature: Boolean validate(String username, String password)
    - Precondition: True
    - Invariant: forall record in [usernames[],passwords[]] ~change(rec)
    - Post-condition: isValid([username, password])

2. Component: UserValidationServer

   Interface: IValidation:

   a. validate – the component verifies the user by checking the username and the password entered by the user.
       - Signature: Boolean validate(String username, String password)
       - Precondition: True
       - Invariant: forall record in [usernames[],passwords[]] ~change(rec)
       - Post-condition: isValid([username, password])

3. Component: DocumentServer

   Interface: IDocumentTerminal

   a. createDocument – creates a new Document with the name specified by the argument 'documentName'.
       - Signature: void createDocument(String documentName)
       - Precondition: ~existDocument(documentName)
       - Invariant: Forall fn in filenames[]: (~changeName(fn) /\ ~changeContent (fn))
       - Post-condition: filenames[] = filenames[] + documentName

   b. deleteDocument – deletes the document with name specified by the argument 'documentName'.
       - Signature: void deleteDocument(String documentName)
       - Precondition: existDocument(documentName)

- Invariant: Forall fn in (filename[] – documentName) ~changeName(fn) /\ ~changeContent(fn)
- Post-condition: filenames[] = filenames[] – documentName

c. readDocument – The document with the name specified by the argument 'documentName' is retrieved and the user allowed to read the document.
    - Signature: DocumentType readDocument(String documentName)
    - Precondition: existDocument(documentName)
    - Invariant: Forall fn in filename[]: ~changeName(fn) /\ ~changeContent(fn)
    - Post-condition: return(documentName)

d. writeDocument – This function opens the document with the name specified by the argument documentName, and the user is allowed to read as well as update the document.
    - Signature: void writeDocument(String documentName, String content)
    - Precondition: existDocument(documentName)
    - Invariant: Forall fn in filename[]: ~changeName(fn) /\ for all fn in (filename[] – documentName): ~changeContent(fn)
    - Post-condition: changeContent(documentName, content)

e. listDocuments – This method is used to list the name of all the documents in the folder.
    - Signature: String [] listDocuments()
    - Precondition: True
    - Invariant: Forall fn in filename[]: ~changeName(fn) /\ ~changeContent(fn)
    - Post-condition: return(filenames[])

Notations:

filenames[]: string array that contains all the filenames

existDocument(<*filename*>): Document with the name file exists

changeName(<*filename*>): change in the name of the document with the name <filename>

~changeName(<*filename*>): No change in the names of the document with the name <filename>

changeContent(<*filename*>,<*new content*>): change in the content of the document with the name <filename> with <new content>

~changeContent(<*filename*>): No change in the names of the document with the name <filename>

return(<data>): returns the data to the user

change(<data>): change data

isValid([username, password]): returns true if the username and password pair is valid else returns false.

- = except

~ = Not

The formal specification for the semantic level contract has been discussed in [ZAR96] in detail.

6.4 Synchronization Level Contract

The synchronization level contract for the three components DocumentTerminal, UserValidationServer and the DocumentServer are as follows:

1. Component: DocumentTerminal

   Interface: IDocumentTerminal

   a. createDocument (documentName, C)

      - Signature :- void createDocument(String documentName)

- Synch Policy :- SP("createDocument", "Mutual Exclusion")
- Pre-condition :-

  forall C1 in clients[]: ~execute("createDocument", documentName, C1)
- Invariant :- NONE
- Action :- execute("createDocument", documentName, C)
- Post-condition:- ~execute("createDocument", documentName, C)

b. deleteDocument (documentName, C)

- Signature :- void deleteDocument(String documentName)
- Synch Policy :- SP("deleteDocument", "Mutual Exclusion")
- Precondition :-

  /\ forall C1 in clients[]: ~execute("readDocument", documentName, C1) ----------- (A)

  /\ forall C1 in clients[]: ~execute("writeDocument", documentName, C1) ----------- (B)

  /\ forall C1 in clients[]: ~execute("deleteDocument", documentName, C1)

  /\ forall C1 in clients[]: ~execute("createDocument", documentName, C1) ----------- (C)

- Invariant :- (A) /\ (B) /\ (C)
- Action :- execute("deleteDocument", documentName,C)
- Post-condition :- ~execute("deleteDocument", documentName,C)

c. readDocument (documentName, C)

- Signature :- DocumentType readDocument(String documentName)
- Synch Policy :- SP("readDocument", "Readers Writers")
- Precondition:

/\ forall C1 in clients[]: ~execute("writeDocument", documentName, C1) ----------- (A)

/\ forall C1 in clients[]: ~execute("deleteDocument", documentName, C1) ----------- (B)

/\ forall C1 in clients[]: ~execute("createDocument", documentName, C1) ----------- (C)

- Invariant :- (A) /\ (B) /\ (C)
- Action :- execute("readDocument", documentName, C)
- Post-condition: ~ execute("readDocument", documentName, C)

d. writeDocument (documentName, C)

- Signature :- void writeDocument(String documentName)
- Synch Policy :- SP("writeDocument", "Readers Writers")
- Precondition :-

/\ forall C1 in clients[]: ~execute("readDocument", documentName, C1) ----------- (A)

/\ forall C1 in clients[]: ~execute("writeDocument", documentName, C1)

/\ forall C1 in clients[]: ~execute("deleteDocument", documentName, C1) ----------- (B)

/\ forall C1 in clients[]: ~execute("createDocument", documentName, C1) ----------- (C)

- Invariant :- (A) /\ (B) /\ (C)
- Action :- execute("writeDocument", documentName, C)
- Post-condition :- ~ execute("writeDocument", documentName, C)

e. listDocuments (C)

- Signature :- String [] listDocuments()
- Synch Policy :- NONE
- Precondition: True
- Invariant :- NONE

- Action :- execute("listDocument", C)
- Post-condition :- ~ execute("listDocument", C)

f. validate(C)

- Signature :- Boolean validate(String username, String password)
- Synch Policy :- NONE
- Precondition :- True
- Invariant :- NONE
- Action :- execute("validate", C)
- Post-condition :- ~ execute("validate", C)

2. Component: UserValidationServer

   Interface: IValidation

   a. validate(C)

   - Signature :- Boolean validate(String username, String password)
   - Synch Policy :- NONE
   - Precondition :- True
   - Invariant :- NONE
   - Action :- execute("validate", C)
   - Post-condition :- ~ execute("validate", C)

3. Component: DocumentServer

   Interface: IDocumentManagement

   a. createDocument (documentName, C)

   - Signature :- void createDocument(String documentName)
   - Synch Policy :- SP("createDocument", "Mutual Exclusion")
   - Pre-condition :-

     forall C1 in clients[]: ~execute("createDocument", documentName, C1) ----------- (A)

     forall C1 in clients[]: ~execute("deleteDocument", documentName, C1) ----------- (B) forall C1 in clients[]: ~execute("readDocument", documentName, C1) ----------- (C)

forall C1 in clients[]: ~execute("writeDocument", documentName, C1) ----------- (D)

- Invariant :- (B) /\ (C) /\ (D)
- Action :- execute("createDocument", documentName, C)
- Post-condition:- ~execute("createDocument", documentName, C)

b. deleteDocument (documentName, C)

- Signature :- void deleteDocument(String documentName)
- Synch Policy :- SP("deleteDocument", "Mutual Exclusion")
- Precondition :-

/\ forall C1 in clients[]: ~execute("readDocument", documentName, C1) ----------- (A)

/\ forall C1 in clients[]: ~execute("writeDocument", documentName, C1) ----------- (B)

/\ forall C1 in clients[]: ~execute("deleteDocument", documentName, C1)

/\ forall C1 in clients[]: ~execute("createDocument", documentName, C1) ----------- (C)

- Invariant :- (A) /\ (B) /\ (C)
- Action :- execute("deleteDocument", documentName,C)
- Post-condition :- ~execute("deleteDocument", documentName,C)

c. readDocument (documentName, C)

- Signature :- DocumentType readDocument(String documentName)
- Synch Policy :- SP("readDocument", "Readers Writers")
- Precondition:

/\ forall C1 in clients[]: ~execute("writeDocument", documentName, C1) ----------- (A)

/\ forall C1 in clients[]: ~execute("deleteDocument", documentName, C1) ----------- (B)

/\ forall C1 in clients[]: ~execute("createDocument", documentName, C1) ------------(C)

- Invariant :- (A) /\ (B) /\ (C)
- Action :- execute("readDocument", documentName, C)
- Post-condition: ~ execute("readDocument", documentName, C)

d. writeDocument (documentName, C)

- Signature :- void writeDocument(String documentName)
- Synch Policy :- SP("writeDocument", "Readers Writers")
- Precondition :-

/\ forall C1 in clients[]: ~execute("readDocument", documentName, C1) ----------- (A)

/\ forall C1 in clients[]: ~execute("writeDocument", documentName, C1)

/\ forall C1 in clients[]: ~execute("deleteDocument", documentName, C1) ----------- (B)

/\ forall C1 in clients[]: ~execute("createDocument", documentName, C1) ----------- (C)

- Invariant :- (A) /\ (B) /\ (C)
- Action :- execute("writeDocument", documentName, C)
- Post-condition :- ~ execute("writeDocument", documentName, C)

e. listDocuments (C)

- Signature :- String [] listDocuments()
- Synch Policy :- NONE
- Precondition: True
- Invariant :- NONE

- Action :- execute("listDocument", C)
- Post-condition :- ~ execute("listDocument", C)

Notations:

execute(Method M, Client C): returns true if client C is executing Method M.
~execute(Method M, Client C): returns true if client C is not executing method M.
execute(Method M, DocumentName DN, Client C): returns true if client C is executing method M on DocumentName DN.
~execute(Method M, DocumentName DN, Client C): returns true if client C is not executing method M on DocumentName DN.
SP(Method M, Synchronization Policy P): returns true if the method M follows the synchronization policy P for multiple accesses by the clients.

The following provides the formal specification of the synchronization contract for one of the components (DocumentServer).

```
------------------------------------ MODULE DocumentServer --------------------------------
EXTENDS TLC
CONSTANT cdclientSet, ddclientSet, Document, Method, rdclientSet, wdclientSet
VARIABLE stateC

client == cdclientSet \cup ddclientSet \cup rdclientSet \cup wdclientSet

cdpre(d) == \A c \in client: stateC["createDocument", d, c] # "executing"
                             /\ stateC["deleteDocument", d, c] # "executing"
                             /\ stateC["readDocument", d, c] # "executing"
                             /\ stateC["writeDocument", d, c] # "executing"

ddpre(d) == \A c \in client: stateC["deleteDocument", d, c] # "executing"
                             /\ stateC["createDocument", d, c] # "executing"
                             /\ stateC["readDocument", d, c] # "executing"
                             /\ stateC["writeDocument", d, c] # "executing"

ddInv(d) == \A c \in client: stateC["createDocument", d, c] # "executing"
                             /\ stateC["readDocument", d, c] # "executing"
                             /\ stateC["writeDocument", d, c] # "executing"

rdpre(d) == \A c \in client: stateC["deleteDocument", d, c] # "executing"
                             /\ stateC["createDocument", d, c] # "executing"
                             /\ stateC["writeDocument", d, c] # "executing"

rdInv(d) == \A c \in client: stateC["createDocument", d, c] # "executing"
                             /\ stateC["writeDocument", d, c] # "executing"
                             /\ stateC["deleteDocument", d, c] # "executing"
```

Figure 6.3 DocumentServer.tla

```
wdpre(d) == \A c \in client: stateC["deleteDocument", d, c] # "executing"
                             /\ stateC["createDocument", d, c] # "executing"
                             /\ stateC["readDocument", d, c] # "executing"
                             /\ stateC["writeDocument", d, c] # "executing"

wdInv(d) == \A c \in client: stateC["createDocument", d, c] # "executing"
                             /\ stateC["readDocument", d, c] # "executing"
                             /\ stateC["deleteDocument", d, c] # "executing"


Init == /\ stateC = [m \in Method, d \in Document, c \in client |-> "idle"]

---------------------------------------------------------------------------------------------
createDocumentStart(d, c) == /\ IF /\ stateC[<<"createDocument", d, c>>] = "idle"
                                   /\ cdpre(d)
                                      THEN stateC' = [stateC EXCEPT !["createDocument", d, c]
                                          = "executing"]
                                      ELSE UNCHANGED stateC

createDocumentEnd(d, c) == /\ IF stateC[<<"createDocument", d, c>>] = "executing"
                             THEN /\ stateC' = [stateC EXCEPT ![<<"createDocument", d, c>>]
                                  = "finished"]
                              ELSE /\ UNCHANGED <<stateC>>

deleteDocumentStart(d, c) == /\ ddInv(d)
                             /\ IF /\ stateC[<<"deleteDocument", d, c>>] = "idle"
                                   /\ ddpre(d)
                                THEN stateC' = [stateC EXCEPT ![<<"deleteDocument", d, c>>] =
                                     "executing"]
                                ELSE UNCHANGED stateC
```

Figure 6.3 DocumentServer.tla (con't.)

```
deleteDocumentEnd(d, c) == /\ ddInv(d)
                           /\ IF /\ stateC[<<"deleteDocument", d, c>>] = "executing"
                              THEN /\ stateC' = [stateC EXCEPT ![<<"deleteDocument", d, c>>]
                                      = "finished"]
                              ELSE /\ UNCHANGED <<stateC>>

readDocumentStart(d, c) == /\ rdInv(d)
                           /\ IF /\ stateC["readDocument", d, c] = "idle"
                                 /\ rdpre(d)
                              THEN stateC' = [stateC EXCEPT !["readDocument", d, c] =
                                   "executing"]
                              ELSE UNCHANGED stateC

readDocumentEnd(d, c) == /\ rdInv(d)
                         /\ IF /\ stateC[<<"readDocument", d, c>>] = "executing"
                            THEN /\ stateC' = [stateC EXCEPT ![<<"readDocument", d, c>>] =
                                    "finished"]
                            ELSE /\ UNCHANGED <<stateC>>

writeDocumentStart(d, c) == /\ wdInv(d)
                            /\ IF /\ stateC["writeDocument", d, c] = "idle"
                                  /\ wdpre(d)
                               THEN stateC' = [stateC EXCEPT !["writeDocument", d, c] =
                                    "executing"]
                               ELSE UNCHANGED stateC

writeDocumentEnd(d, c) == /\ wdInv(d)
                          /\ IF /\ stateC[<<"writeDocument", d, c>>] = "executing"
                             THEN /\ stateC' = [stateC EXCEPT ![<<"writeDocument", d, c>>] =
                                    "finished"]
                              ELSE /\ UNCHANGED <<stateC>>
```

Figure 6.3 DocumentServer.tla (con't.)

```
createDocument(d, c) == \/ createDocumentStart(d, c) \/ createDocumentEnd(d, c)
deleteDocument(d, c) == \/ deleteDocumentStart(d, c) \/ deleteDocumentEnd(d, c)
readDocument(d, c) == \/ readDocumentStart(d, c) \/ readDocumentEnd(d, c)
writeDocument(d, c) == \/ writeDocumentStart(d, c) \/ writeDocumentEnd(d, c)

Next == \E d \in Document, cc \in cdclientSet, dc \in ddclientSet, rc \in rdclientSet, wc \in
wdclientSet:
                                                  \/ createDocument(d, cc)
                                                        \/ deleteDocument(d, dc)
\*                                                \/ readDocument(d, dc)
\*                                                \/ writeDocument(d, dc)


PNext == Next /\ Print(stateC, TRUE)
---------------------------------------------------------------------------------------------
Spec1 == Init /\ [][Next]_<<stateC>>
=============================================================================================
```

Figure 6.3 DocumentServer.tla (con't.)

6.5 Matching Synchronization Level Contracts

The component interaction in simple document management system as explained in the section 6.1.1.2 indicates that the user interacts with the DocumentTerminal and send request to the component and the components DocumentServer and the UserValidationServer provide services to the DocumentTerminal for the component to provide results to the users. In order to determine whether the components are compatible to each other, the contracts of the components DocumentServer and DocumentTerminal and that of the components UserValidationServer and the DocumentTerminal have to be matched. The matching of the syntactic and the semantic level contracts of the components follows the matching criteria described in [ZAR96]. This section provides the matching of the synchronization contracts of the component pairs. For simplicity the section describes the matching of few of the methods of the components DocumentServer and the DocumentTerminal. The complete set of matching can be is defined in Appendix B.

The component DocumentTerminal requires the functionalities of the component DocumentServer and hence, the specification of the DocumentServer should satisfy the properties of the DocumentTerminal. This is achieved in this thesis using the following TLA+ specifications.

```
-------------------------------------- MODULE matchST----------------------------------------
EXTENDS TLC, DocumentServer

cdpreT(d) == \A c \in client:      stateC["createDocument", d, c] # "executing"

ddpreT(d) == \A c \in client:      stateC["deleteDocument", d, c] # "executing"
                          /\ stateC["createDocument", d, c] # "executing"
                          /\ stateC["readDocument", d, c] # "executing"
                          /\ stateC["writeDocument", d, c] # "executing"

ddInvT(d) == \A c \in client:      stateC["createDocument", d, c] # "executing"
                          /\ stateC["readDocument", d, c] # "executing"
                          /\ stateC["writeDocument", d, c] # "executing"

rdpreT(d) == \A c \in client:      stateC["deleteDocument", d, c] # "executing"
                          /\ stateC["createDocument", d, c] # "executing"
                          /\ stateC["writeDocument", d, c] # "executing"

rdInvT(d) == \A c \in client:      stateC["createDocument", d, c] # "executing"
                          /\ stateC["writeDocument", d, c] # "executing"
                          /\ stateC["deleteDocument", d, c] # "executing"

wdpreT(d) == \A c \in client:      stateC["deleteDocument", d, c] # "executing"
                          /\ stateC["createDocument", d, c] # "executing"
                          /\ stateC["readDocument", d, c] # "executing"
                          /\ stateC["writeDocument", d, c] # "executing"

wdInvT(d) == \A c \in client:      stateC["createDocument", d, c] # "executing"
                          /\ stateC["readDocument", d, c] # "executing"
                          /\ stateC["deleteDocument", d, c] # "executing"
```

Figure 6.4 matchST.tla

```
InitT == /\ stateC = [m \in Method, d \in Document, c \in client |-> "idle"]
--------------------------------------------------------------------------------------------
createDocumentStartT(d, c) ==/\ IF /\ stateC[<<"createDocument", d, c>>] = "idle"
                                   /\ cdpreT(d)
                                THEN stateC' = [stateC EXCEPT !["createDocument", d, c] =
                                    "executing"]
                                ELSE UNCHANGED stateC

createDocumentEndT(d, c) ==/\ IF stateC[<<"createDocument", d, c>>] = "executing"
                              THEN /\ stateC' = [stateC EXCEPT ![<<"createDocument", d, c>>]
                                      = "finished"]
                              ELSE /\ UNCHANGED <<stateC>>

deleteDocumentStartT(d, c) ==/\ ddInvT(d)
                             /\ IF /\ stateC[<<"deleteDocument", d, c>>] = "idle"
                                   /\ ddpreT(d)
                                THEN stateC' = [stateC EXCEPT ![<<"deleteDocument", d, c>>] =
                                    "executing"]
                                ELSE UNCHANGED stateC

deleteDocumentEndT(d, c) ==/\ ddInvT(d)
                           /\ IF /\ stateC[<<"deleteDocument", d, c>>] = "executing"
                              THEN /\ stateC' = [stateC EXCEPT ![<<"deleteDocument", d, c>>]
                                      = "finished"]
                              ELSE /\ UNCHANGED <<stateC>>

readDocumentStartT(d, c) ==/\ rdInvT(d)
                           /\ IF /\ stateC["readDocument", d, c] = "idle"
                                 /\ rdpreT(d)
                              THEN stateC' = [stateC EXCEPT !["readDocument", d, c] =
                                    "executing"]
                              ELSE UNCHANGED stateC
```

Figure 6.4 matchST.tla (con't.)

```
readDocumentEndT(d, c) ==/\ rdInvT(d)
                         /\ IF /\ stateC[<<"readDocument", d, c>>] = "executing"
                            THEN /\ stateC' = [stateC EXCEPT ![<<"readDocument", d, c>>] =
                                    "finished"]
                            ELSE /\ UNCHANGED <<stateC>>

writeDocumentStartT(d, c) ==/\ wdInvT(d)
                            /\ IF /\ stateC["writeDocument", d, c] = "idle"
                                  /\ wdpreT(d)
                               THEN stateC' = [stateC EXCEPT !["writeDocument", d, c] =
                                    "executing"]
                               ELSE UNCHANGED stateC

writeDocumentEndT(d, c) ==/\ wdInvT(d)
                          /\ IF /\ stateC[<<"writeDocument", d, c>>] = "executing"
                             THEN /\ stateC' = [stateC EXCEPT ![<<"writeDocument", d, c>>] =
                                    "finished"]
                             ELSE /\ UNCHANGED <<stateC>>

createDocumentT(d, c) == \/ createDocumentStartT(d, c) \/ createDocumentEndT(d, c)
deleteDocumentT(d, c) == \/ deleteDocumentStartT(d, c) \/ deleteDocumentEndT(d, c)
readDocumentT(d, c) == \/ readDocumentStartT(d, c) \/ readDocumentEndT(d, c)
writeDocumentT(d, c) == \/ writeDocumentStartT(d, c) \/ writeDocumentEndT(d, c)

NextT == \E d \in Document, cc \in cdclientSet, dc \in ddclientSet, rc \in rdclientSet,
                                                wc \in wdclientSet:
                                                  \/ createDocumentT(d, cc)
                                                  \/ deleteDocumentT(d, dc)
\*                                                \/ readDocumentT(d, dc)
\*                                                \/ writeDocumentT(d, dc)
```

Figure 6.4 matchST.tla (con't.)

6.6 QoS Level Contract

The Figure 6.5 provides the QoS contract for the component DocumentTerminal of the simple document management system. This thesis does not provide an implementation of the QoS contract and their matching criteria. It follows the argument that the QoS contract and the matching criteria consist of predicates which can easily be implemented in any of the logic programming languages.

```
/* QoS Contract for the Component C1. It follows the UQoS standards.

qosP_value(DocumentTerminal, dependability, 0.62) /* value should be
in between [0, 1]
qosP_value(DocumentTerminal, security, 800)
qosP_value(DocumentTerminal, adaptability, 0.75) /* value should be
in between [0, 1]
qosP_value(DocumentTerminal, maintainability, 10) /* ratio of man
months over man months
qosP_value(DocumentTerminal, portability, 1)

qosP_value(DocumentTerminal, parallelismConstraints, createDocument,
1)
qosP_value(DocumentTerminal, parallelismConstraints, deleteDocument,
0)
qosP_value(DocumentTerminal, parallelismConstraints, readDocument, 1)
qosP_value(DocumentTerminal, parallelismConstraints, writeDocument,
0)
qosP_value(DocumentTerminal, parallelismConstraints, listDocument, 0)
qosP_value(DocumentTerminal, parallelismConstraints, validate, 0)

qosP_environmentVariables(DocumentTerminal, orderingConstraints,
[networkTransmission, networkProtocol])
qosP_environmentVariables_effect(DocumentTerminal,
orderingConstraints, networkTransmission, <url:graph for effect of
networkTransmission>)
qosP_environmentVariables_effect(DocumentTerminal,
orderingConstraints, networkProtocol, <url:graph for effect of
networkProtocol>)
```

Figure 6.5 QoS contract for the Document Terminal

```
/* QoS Contract for component C1 continued…

qosP_value(DocumentTerminal, orderingConstraints, createDocument, 0)
qosP_value(DocumentTerminal, orderingConstraints, deleteDocument, 1)
qosP_value(DocumentTerminal, orderingConstraints, readDocument, 0)
qosP_value(DocumentTerminal, orderingConstraints, writeDocument, 1)
qosP_value(DocumentTerminal, orderingConstraints, listDocument, 1)
qosP_value(DocumentTerminal, orderingConstraints, validate, 1)

qosP_value(DocumentTerminal, priority, createDocument, 0)
qosP_value(DocumentTerminal, priority, deleteDocument, 0)
qosP_value(DocumentTerminal, priority, readDocument, 0)
qosP_value(DocumentTerminal, priority, writeDocument, 0)
qosP_value(DocumentTerminal, priority, listDocument, 0)
qosP_value(DocumentTerminal, priority, validate, 0)

qosP_environmentVariables(DocumentTerminal, throughput, [algorithm,
cpuSpeed, memory, hardware])
qosP_environmentVariables_effect(DocumentTerminal, throughput,
algorithm, <url:graph for effect of algorithm>)
qosP_environmentVariables_effect(DocumentTerminal, throughput,
cpuSpeed, <url:graph for effect of cpuSpeed>)
qosP_environmentVariables_effect(DocumentTerminal, throughput,
memory, <url:graph for effect of memory>)
qosP_environmentVariables(DocumentTerminal, throughput, [algorithm,
cpuSpeed, memory, hardware])
qosP_environmentVariables_effect(DocumentTerminal, throughput,
algorithm, <url:graph for effect of algorithm>)
qosP_environmentVariables_effect(DocumentTerminal, throughput,
cpuSpeed, <url:graph for effect of cpuSpeed>)
qosP_environmentVariables_effect(DocumentTerminal, throughput,
memory, <url:graph for effect of memory>)
```

Figure 6.5 QoS contract for the Document Terminal (con't.)

```
qosP_environmentVariables_effect(DocumentTerminal, throughput,
hardware, <url:graph for effect of hardware>)

qosP_value(DocumentTerminal, throughput, createDocument, 6) /* number
of responses per second
qosP_value(DocumentTerminal, throughput, deleteDocument, 8)
qosP_value(DocumentTerminal, throughput, readDocument, 6)
qosP_value(DocumentTerminal, throughput, writeDocument, 2)
qosP_value(DocumentTerminal, throughput, listDocument, 2)
qosP_value(DocumentTerminal, throughput, validate, 2)


qosP_environmentVariables(DocumentTerminal, capacity, [threads,
cpuSpeed, memory, hardware])
qosP_environmentVariables_effect(DocumentTerminal, capacity, threads,
<url:graph for effect of threads>)
qosP_environmentVariables_effect(DocumentTerminal, capacity,
cpuSpeed, <url:graph for effect of cpuSpeed>)

qosP_environmentVariables_effect(DocumentTerminal, capacity, memory,
<url:graph for effect of memory>)
qosP_environmentVariables_effect(DocumentTerminal, capacity,
hardware, <url:graph for effect of hardware>)
qosP_value(DocumentTerminal, capacity, createDocument, 1) /* number
of request per second

qosP_value(DocumentTerminal, capacity, deleteDocument, 10)
qosP_value(DocumentTerminal, capacity, readDocument, 5)
qosP_value(DocumentTerminal, capacity, writeDocument, 4)
qosP_value(DocumentTerminal, capacity, listDocument, 4)
qosP_value(DocumentTerminal, capacity, validate, 4)
```

Figure 6.5 QoS contract for the Document Terminal (con't.)

```
qosP_environmentVariables(DocumentTerminal, turn_around_time,
[algorithm, threads, cpuSpeed, memory, load_on_system,
os_access_policy])
qosP_environmentVariables_effect(DocumentTerminal, turn_around_time,
algorithm, <url:graph for effect of algorithm>)
qosP_environmentVariables_effect(DocumentTerminal, turn_around_time,
cpuSpeed, <url:graph for effect of cpuSpeed>)
qosP_environmentVariables_effect(DocumentTerminal, turn_around_time,
memory, <url:graph for effect of memory>)
qosP_environmentVariables_effect(DocumentTerminal, turn_around_time,
load_on_system, <url:graph for effect of load_on_system>)
qosP_environmentVariables_effect(DocumentTerminal, turn_around_time,
os_access_policy, <url:graph for effect of os_access_policy>)

qosP_value(DocumentTerminal, turn_around_time, createDocument, 100)
/* value is measured in milliseconds
qosP_value(DocumentTerminal, turn_around_time, deleteDocument, 50)
qosP_value(DocumentTerminal, turn_around_time, readDocument, 70)
qosP_value(DocumentTerminal, turn_around_time, writeDocument, 20)
qosP_value(DocumentTerminal, turn_around_time, listDocument, 20)
qosP_value(DocumentTerminal, turn_around_time, validate, 20)

qosP_environmentVariables(DocumentTerminal, availability,
[dependability, fault_tolerance, efficiency_repairMechanism])
qosP_environmentVariables_effect(DocumentTerminal, availability,
dependability, <url:graph for effect of dependability>)
qosP_environmentVariables_effect(DocumentTerminal, availability,
fault_tolerance, <url:graph for effect of fault_tolerance >)
qosP_environmentVariables_effect(DocumentTerminal, availability,
efficiency_repairMechanism, <url:graph for effect of
efficiency_repairMechanism>)
```

Figure 6.5 QoS contract for the Document Terminal (con't.)

```
/* QoS Contract for component C1 continued…

qosP_value(DocumentTerminal, availability, createDocument, 70) /*
Value is in percentage
qosP_value(DocumentTerminal, availability, deleteDocument, 50)
qosP_value(DocumentTerminal, availability, readDocument, 40)
qosP_value(DocumentTerminal, availability, writeDocument, 80)
qosP_value(DocumentTerminal, availability, listDocument, 80)
qosP_value(DocumentTerminal, availability, validate, 80)
```

Figure 6.5 QoS contract for the Document Terminal (con't.)

The chapter provided an example for illustrating the key concepts of the thesis. It provides a detailed description of the system under consideration with the component and the interface model. It gives the contract of the all the components in the system at all the four levels, the syntactic, the semantic, the synchronization and the QoS levels. The chapter also provides a few examples of the formal specification of these contracts and illustrates the matching at the synchronization level. The next chapter summarizes the thesis by providing the conclusions of the research work, the contributions of the thesis and a few of the possible future extensions.

7. CONCLUSION AND FUTURE WORK

This thesis presented a mechanism for formally specifying the components synchronization and the QoS contracts and mechanism to match the contracts at the two levels. The next section provides the conclusions of the research work. Section 7.2 presents an overview of the mechanism for specifying the contracts and defining the matching criteria at the two levels. The section 7.3 gives the possible future works to the research work and the last section summarizes this thesis.

7.1 Conclusions of the Research Work

The following are the conclusions drawn from the research work:

1. Specifying each component's interface at the four levels should accompany the development of every DCS developed using CBSD.
2. Matching of the synchronization contracts of the software components is necessary for the resulting system to work effectively.
3. Matching of the QoS contracts of the software components is especially essential for the development of quality-critical systems.
4. Not all QoS parameters are significant for all the systems, and hence, the matching should be performed in such a way that it does not involve all the parameters at all times.
5. The matching of the contracts at all the levels may be time consuming for complex real time systems.

7.2 Overview of the Synchronization and the QoS Level Contracts

For selecting the components to form a system from the component library and to predict the behavior of the resultant system, the component developers need to specify the component's interface at each of the four levels namely, the syntactic, the semantic, the synchronization and the QoS levels and create contracts at each of the levels. [ZAR96] provides a mechanism for specifying the component's interface at the syntactic and the semantic levels. This thesis provides the mechanism to specify a component' interface at the other two levels [Chapter five]. It helps users to match the specifications of two or more components. Matching of the contracts is needed as the component library consists of a large number of components and there usually exists more than one component that provides the same functionality. The contracts help users match the specifications and make a selection decision amongst the components with the same functionality. It also helps the users predict the behavior of the resultant system before actually integrating the components. The behavior of the resultant system then can be verified against the behavior of the desired system.

The following are the features of the specifications at the synchronization and the QoS levels:

1. The mechanism for specifying the contracts at the synchronization level (using TLA+) explores the set of all possible sets and checks for problems such as deadlocks and starvation in a software component.
2. The contracts at the QoS level specify the component independent properties as well as the component dependent properties of the QoS parameters. It also indicates the effect of environment on the parameter in the contract. This helps component developers to give reasoning for the difference in the values of the parameters specified in the contract and the values computed by the users. It also helps the users to predict the value of the QoS parameter for an execution environment of his choice.

3. The TLA+ specification of the synchronization behavior of the whole system verifies the system against the desired system and checks for synchronization problems such as deadlock and starvation.
4. The relaxed match defined for matching the contracts at the QoS level gives users the choice to specify desired degree of relaxation.
5. The contracts and the matches have been provided irrespective of the component model and the implementation language of the component and hence, are applicable to a variety of software components.

7.3 Contributions of the Thesis

The contribution of the thesis is as follows:

1. Provision of a mechanism to create generalized contracts at the synchronization and the QoS levels.
2. Generalized specification of the component independent and component dependent properties of the QoS parameters.
3. Definition of a set of matching criteria for matching the synchronization and the QoS level contracts of a software component.
4. Provision of a mechanism to match the contracts at the synchronization and the QoS levels.
5. Creation of catalog of the basic synchronization policies which can be standardized to create a standard catalog.
6. Specification of the behavior of the basic synchronization policies and a mechanism to formally represent them.
7. Provision of a mechanism to match the behaviors of the basic synchronization policies for compatibility.
8. Provision of a mechanism to specify the synchronization behavior of the components that use these synchronization policies.

7.4 Future Work

Several future extensions to this research work are possible and few of them have been suggested below:

1. The catalog of the basic synchronization policies can be extended to include other synchronization policies as well.
2. Running TLA+ specifications with the TLC model checker for complex systems takes a large amount of time. This might slow the verification process of the whole system. A more efficient implementation method can be suggested to make the verification process faster.
3. The formal representation of the contracts at both the levels is done manually. For complex system it might be time consuming. Moreover the user has to learn a new language to represent the contracts. Hence, a system to automatically generate these formal representations would make the job task easier.
4. The contracts and the matching criteria can be incorporated within the UniFrame Resource Discovery Service [SIR02] to refine the searching of components.
5. A runtime monitoring system can be designed to verify the actual system against the specification of the system derived from composing the component's specification.

7.5 Summary

The thesis has presented a generalized software component contracts at the synchronization and the QoS levels, which facilitates the selection of components from the component library. The matching criteria defined for the two levels aids in the creation of high-confidence DCS. The thesis also provides a promising case study of document management system to validate the research work.