To Aaji and Dattakaka.
ACKNOWLEDGEMENTS

The graduate education at the Department of Computer and Information Science at Indiana University Purdue University Indianapolis (IUPUI) has been one of the most fruitful experiences in my life. It has been a significant turning point as far as my academic career is concerned. I would like to take this opportunity to thank everyone who participated in making my graduate education a very special and rewarding experience.

I would like to thank Dr. Rajeev Raje for his guidance, help and countless valuable lessons. My experience at IUPUI would not have been the same if it wasn’t for his constant support and encouragement, and ability to always challenge me. I would also like to thank Dr. Andrew Olson for providing insightful comments and meticulously reviewing my work. I would like to thanks Dr. Zou for the effort he put in to review my thesis. I would also like to thank Dr. Tuceryan for agreeing to serve on my thesis committee.

I would like to thank Indigo Foundation for supporting this research activity. Also, I would also like to thank the entire IUPUI Computer Science Department Staff and Professors for being so helpful and friendly throughout my stay at IUPUI.

Last but not the least, I want to thank my family and close friends for always helping me stay focused. I would like to thank all the friends I made at IUPUI, some really special ones, for being more like a family away from home.
# TABLE OF CONTENTS

| LIST OF TABLES | ................. viii |
| LIST OF FIGURES | .................. x |
| ABSTRACT | ................................ xi |
| 1. INTRODUCTION | ........................................ 1 |
| 1.1 Problem Definition and Motivation | ............................ 3 |
| 1.2 Objectives | ....................................... 5 |
| 1.3 Contributions | ..................................... 6 |
| 1.4 Thesis Organization | ..................................... 6 |
| 2. BACKGROUND AND RELATED WORK | ....................................... 7 |
| 2.1 The UniFrame Project | ....................................... 7 |
| 2.1.1 Building Blocks | ....................................... 7 |
| 2.1.2 Approach | ....................................... 8 |
| 2.2 Architecture Modeling and Domain Engineering | ....................................... 10 |
| 2.2.1 Model Driven Architecture (MDA) | ....................................... 11 |
| 2.2.2 Generative Programming | ....................................... 13 |
| 2.2.3 Domain Engineering Methods | ....................................... 15 |
| 2.2.3.1 Overview of Domain Engineering Techniques | ....................................... 16 |
| 2.2.3.1.1 Feature-Oriented Domain Analysis | ....................................... 16 |
| 2.2.3.1.2 Organization Domain Modeling | ....................................... 17 |
| 2.2.3.1.3 FAST and SDA | ....................................... 18 |
| 2.2.3.2 Modeling Problem Space for a Variable Domain | ....................................... 19 |
| 2.3 Quality of Service | ....................................... 23 |
2.4 The UniFrame System-Level Generative Programming Framework (USGPF) ........................................................................................................... 24
2.5 Conclusion ....................................................................................................... 28
3. THE KNOWLEDGEBASE ARCHITECTURE ........................................................... 29
3.1 The Architecture of the KB ........................................................................... 29
  3.1.1 Domain Information .................................................................................. 31
    3.1.1.1 Description ....................................................................................... 31
    3.1.1.2 Dictionary ......................................................................................... 33
    3.1.1.3 Models ............................................................................................... 34
      3.1.1.3.1 Feature Model ........................................................................ 34
      3.1.1.3.2 Decision Model ........................................................................ 36
    3.1.1.4 Diagrams ............................................................................................ 37
    3.1.1.5 UMM-Annotated UDSL .................................................................... 39
      3.1.1.5.1 UMM Specification ................................................................. 39
      3.1.1.5.2 UMM-annotated UDSL .......................................................... 45
    3.1.1.6 QoS Associations ............................................................................. 48
    3.1.1.7 Handling Heterogeneity ...................................................................... 49
  3.1.2 QoS .......................................................................................................... 51
    3.1.2.1 Attribute Details and Metrics ............................................................ 51
    3.1.2.2 Evaluation Process (Component-Level) ........................................ 52
    3.1.2.3 Composition and Decomposition (System-of Components level) ........ 53
    3.1.2.4 Property Prediction and Instrumentation ......................................... 55
  3.1.3 Components ............................................................................................ 57
    3.1.3.1 Specifications and Implementations of Components .................... 57
    3.1.3.2 Unit Test Cases ................................................................................ 59
  3.1.4 Technology .............................................................................................. 61
  3.1.5 Query Decomposition ............................................................................. 62
    3.1.5.1 Functional Decomposition ............................................................. 62
3.1.5.2 Non-Functional Decomposition ............................................. 63
3.1.6 Multilevel Matching ....................................................................... 64
3.2 Process of Constructing the Distributed Computing System ............... 65
3.3 Conclusion ...................................................................................... 68

4. A PROTOTYPICAL REALIZATION OF UNIFRAME APPROACH (UA) .. 69
   4.1 Architecture .................................................................................. 69
   4.2 Components .................................................................................. 70
      4.2.1 UMM Specification Editor .................................................. 70
      4.2.2 Translation of the UMM Specification to a Language Specific
           Implementation (Skeletal Code Generator) .............................. 72
   4.3 Quality of Service ......................................................................... 73
   4.4 Discovery Service .......................................................................... 73
      4.4.1 Architecture for Discovery of Heterogeneous Software
           Components (URDS) .............................................................. 74
      4.4.2 Multilevel Matching ............................................................... 74
   4.5 System Generation ......................................................................... 75
      4.5.1 Glue Generation Framework in UniFrame for the CORBA-
           Java/RMI Interoperability ...................................................... 75
      4.5.2 System Generator as part of UniFrame System Generation
           Infrastructure (USGI) ............................................................. 76
   4.6 Conclusion .................................................................................... 79

5. CASE STUDY – BANKING DOMAIN ............................................... 80
   5.1 Architecture .................................................................................. 81
      5.1.1 Description ............................................................................. 82
      5.1.2 Dictionary ............................................................................... 83
      5.1.3 Models .................................................................................... 84
         5.1.3.1 Feature Model (Feature Diagram) .................................... 84
         5.1.3.2 Decision Model (Flowchart Notation) ............................. 85
      5.1.4 Diagrams ................................................................................. 87
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 3.1 Structure of Domain Description</td>
<td>31</td>
</tr>
<tr>
<td>Table 3.2 Structure of Domain Dictionary</td>
<td>33</td>
</tr>
<tr>
<td>Table 3.3 Creating A Decision Model</td>
<td>37</td>
</tr>
<tr>
<td>Table 3.4 Sample UMM Specification</td>
<td>40</td>
</tr>
<tr>
<td>Table 3.5 Structure of QoS Associations</td>
<td>48</td>
</tr>
<tr>
<td>Table 3.6 QoS Evaluation Formula</td>
<td>52</td>
</tr>
<tr>
<td>Table 3.7 QoS Composition Formula</td>
<td>53</td>
</tr>
<tr>
<td>Table 3.8 QoS Decomposition Formula</td>
<td>54</td>
</tr>
<tr>
<td>Table 3.9 Change-record Set</td>
<td>58</td>
</tr>
<tr>
<td>Table 3.10 Unit Test Case Format</td>
<td>60</td>
</tr>
<tr>
<td>Table 3.11 Technology</td>
<td>61</td>
</tr>
<tr>
<td>Table 3.12 Syntax for Multilevel Matching Algorithm</td>
<td>64</td>
</tr>
<tr>
<td>Table 5.1 Domain Description for the Banking Domain</td>
<td>82</td>
</tr>
<tr>
<td>Table 5.2 Domain Dictionary for the Banking Domain</td>
<td>83</td>
</tr>
<tr>
<td>Table 5.3 Decision Model for Banking Domain</td>
<td>86</td>
</tr>
<tr>
<td>Table 5.4 UMM-Annotated UDSL for the Banking Domain</td>
<td>90</td>
</tr>
<tr>
<td>Table 5.5 Advice Code for startTimer() for Turnaround Time</td>
<td>101</td>
</tr>
<tr>
<td>Table 5.6 Advice Code for stopTimer() for Turnaround Time</td>
<td>102</td>
</tr>
<tr>
<td>Table 5.7 Join points for measuring turnaround time for Transfer method</td>
<td>102</td>
</tr>
<tr>
<td>Table 5.8 Component Deployment Information for AccountManager of the Banking Domain</td>
<td>105</td>
</tr>
<tr>
<td>Table 5.9 Component Deployment Information for AccountServer of the Banking Domain</td>
<td>106</td>
</tr>
</tbody>
</table>
Table 5.10 Component Deployment Information for *ATM component* of the Banking Domain ................................................................. 106

Table 5.11 Test Case for ATM-1 ............................................................................................................. 107

Table 5.12 Test Case for ATM-2 ............................................................................................................. 108

Table 5.13 Test Case for ATM-3 ............................................................................................................. 109

Table A.1 Attribute: Dependability ....................................................................................................... 146

Table A.2 Attribute: Security ................................................................................................................. 149

Table A.3 Attribute: Adaptability ......................................................................................................... 153

Table A.4 Attribute: Maintainability .................................................................................................... 157

Table A.5 Attribute: Portability ............................................................................................................. 164

Table A.6 Attribute: Parallelism Constraints ....................................................................................... 167

Table A.7 Attribute: Ordering Constraints ......................................................................................... 170

Table A.8 Attribute: Priority .................................................................................................................. 172

Table A.9 Attribute: Throughput ......................................................................................................... 173

Table A.10 Attribute: Capacity ............................................................................................................ 176

Table A.11 Attribute: Turn-around Time .............................................................................................. 178

Table A.12 Attribute: Availability ....................................................................................................... 180
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1 The UniFrame Approach</td>
<td>7</td>
</tr>
<tr>
<td>Figure 2.2 Elements of a Generative Domain Model</td>
<td>14</td>
</tr>
<tr>
<td>Figure 3.1 Knowledgebase Architecture</td>
<td>30</td>
</tr>
<tr>
<td>Figure 3.2 Process of Construction of a Distributed Computing System</td>
<td>66</td>
</tr>
<tr>
<td>Figure 4.1 The Complete UniFrame Framework</td>
<td>69</td>
</tr>
<tr>
<td>Figure 4.2 UMM Specification Editor Application Interface (from [NEI04])</td>
<td>71</td>
</tr>
<tr>
<td>Figure 4.3 USGI Workflow</td>
<td>77</td>
</tr>
<tr>
<td>Figure 5.1 Feature Diagram for the Banking Domain</td>
<td>84</td>
</tr>
<tr>
<td>Figure 5.2 Decision Model for the Banking Domain</td>
<td>86</td>
</tr>
<tr>
<td>Figure 5.3 Use Case Diagram for Banking Domain</td>
<td>88</td>
</tr>
<tr>
<td>Figure 5.4 Sequence Diagram for Transfer Funds</td>
<td>89</td>
</tr>
<tr>
<td>Figure 5.5 Sequence Diagram for Withdraw at ATM</td>
<td>89</td>
</tr>
<tr>
<td>Figure 6.1 User Profiles for the KB Tool Design</td>
<td>114</td>
</tr>
<tr>
<td>Figure 6.2 Domain Expert View – Input Domain Description</td>
<td>122</td>
</tr>
<tr>
<td>Figure 6.3 Domain Expert View – Input Domain Dictionary</td>
<td>123</td>
</tr>
<tr>
<td>Figure 6.4 Domain Expert View – Input QoS Associations</td>
<td>124</td>
</tr>
<tr>
<td>Figure 6.5 QoS Expert View – Input QoS Attribute</td>
<td>125</td>
</tr>
<tr>
<td>Figure 6.6 Client View - Input System Specification</td>
<td>126</td>
</tr>
<tr>
<td>Figure 6.7 System Integrator View – Select features for DCS</td>
<td>127</td>
</tr>
<tr>
<td>Figure 6.8 System Integrator View – Components returned by Discovery Service</td>
<td>128</td>
</tr>
</tbody>
</table>
The distributed computing systems (DCS) are the systems of today and tomorrow. Construction of DCS poses challenges that are related not only to meeting functional requirements but also non-functional requirements, e.g. Quality of Service (QoS). An approach for DCS development must employ formalism in all stages of development from inception to deployment. The UniFrame Approach (UA) uses iterative development and formalism to design high confidence DCS, with a goal to automate the process of integrating heterogeneous software components to create DCS that conform to certain quality requirements and offer a significant measure of predictability. The UA rests on model-driven design, formulation of vocabulary for DCS families, multilevel component specifications, and Two-Level Grammars (TLG) and Event Grammars (EG). The Knowledgebase (KB) is a central entity in the UA, which contains detailed descriptions for DCS architectures for DCS families. It also contains rules to construct high-quality components that adhere to specifications and semi-automatic system generation rules. The content of the KB is vaguely known and frequently used as part of the various other UA-based models such as System Generators, Discovery Services and Component Construction. This thesis is an attempt to provide a detailed architecture of the KB, by enlisting its constituents and their representations and interrelationships. Having done that, the thesis also attempts to provide a logical connection between other concrete entities developed as part of the UA and the KB, since putting those together would result in a complete realization of the UA. A ‘KB Tool’ prototype is designed and implemented to input/use the KB information concerning diverse domains.
CHAPTER 1. INTRODUCTION

Today’s software systems are inherently complex and distributed in nature. Along with the advantages that distributed computing systems (DCS) lend to software applications such as scalability, fault tolerance, etc., there come challenges that need to be addressed throughout the construction of distributed systems. The manner in which the construction process handles the issues of quality, confidence, and predictability is thus a major deciding factor that will determine the success of organizations in the future. There are several approaches and methodologies that can be employed to create distributed computing systems that are reliable and predictable, for example, Component-Based Software Development (CBSD) [HAI00], which involves component-based systems encompassing ‘commercial-off-the-shelf’ components, Service-oriented architectures (SOA) [SOA], which rely on collaboration of loosely coupled software services, and the Web Services Infrastructure (WS) [WS02], which provides a means for realization of SOAs using web services.

While these approaches have thrived in the past for various solutions over diverse domains, a large percentage of these distributed systems are handcrafted using laborious and error prone processes, mostly without any formal techniques for reasoning about the quality of resulting system. For distributed systems to be designed and constructed with a high quality and predictability, it is essential that formalism be incorporated throughout the entire lifecycle – starting from specification of the system and its constituents, to actual construction and validation of the system. A mature, comprehensive approach with strong theoretical foundations combined with a pragmatic outlook is the UniFrame Approach (UA) [RAJ01]. The UA embodies iterative development and formalism to
design high confidence component-based distributed systems – the approach can be adapted to any distributed paradigm (e.g., SOA) since it is aimed at tackling challenges associated with constructing distributed systems in general. The goal of the UA is to automate the process of integrating heterogeneous components to create DCS that conform to certain quality requirements, and offer a significant measure of predictability.

The UA is based on principles of the model-driven design and construction to formulate the extensive vocabulary of a DCS family \cite{CZA00}, a distributed discovery of heterogeneous components using multi-level specifications, and two-level grammar (TLG) and event grammar (EG) \cite{BRY02} to form the formal basis for the validation of the functionality as well as quality of service (QoS) parameters of the DCS. There are several facets to the UA, some of which will be discussed in subsequent chapters. This thesis is focused on the central piece of the UniFrame research – the Knowledgebase (KB). The KB contains the detailed descriptions of DCS architectures for families of DCS, rules for specifying, constructing and validating components, semi-automatic system generation and a mechanism to incorporate QoS in the construction of DCS. The high-level objective of this thesis is resolving challenges associated with resolving the exact constituents of the KB, their representations and interrelationships, and then making a logical connection between these constituents and the other entities that have been developed as part of the UniFrame research, such as discovery service, skeletal code generator, glue-wrapper generator and the system generator, to name a few. The exact objectives will be clearer in the subsequent sections. For clarification, the term ‘KB’ used throughout this thesis is as perceived by UniFrame and is in the context of DCS and information required to generate the DCS. It is different from the notion of KB in the Artificial Intelligence domain wherein it is a special type of database for knowledge management and provides the means to collect, organize and retrieve knowledge \cite{KB}. 
1.1 Problem Definition and Motivation

As indicated above, it is desirable to incorporate formalism in the construction of DCS, throughout its lifecycle, so that the resulting systems will be reliable, conform to certain QoS requirements, and offer a predictability measure for the quality of the system. Using the CBSD process for the development of DCS poses many challenges, such as heterogeneity of component models and technologies, discovery of components and semi-automatic code generation. For example, most of the software components used today are developed using disparate technologies and models, such as Java Remote Method Invocation (RMI) [SDN], .NET [NET03], Distributed Component Object Model (DCOM) [MS98] and Common Object Request Broker Architecture (CORBA) [OMG99, ORF98, SEI96], so a DCS may require integrating such heterogeneous components.

Seamlessly integrating components adhering to such disparate component models is a definite challenge, coupled with the issue of incorporation of QoS requirements in the construction of DCS. There needs to be a mechanism to describe the QoS of a system, and also to instrument code in the components and systems to measure the actual ‘quality’ of the software piece. [BRA02] presents a QoS catalog that describes some QoS attributes and their essential characteristics. Combining this with QoS composition-decomposition rules, framed in [SUN02], provides a comprehensive QoS framework to deal with the latter challenge of incorporation of QoS requirements. As described in [HUA03], the development processes that exist today lack the ability to build QoS-aware DCS in a formal manner, and the added inherent complexity of DCS further complicates the problem.

The concept of Model-driven architectures (MDA) [OMG01] initiated by the Object Management Group (OMG) emphasizes the need for standardization in order to generate a distributed system, thus providing a clear separation between design and business functions on one hand, and implementation and other technological details on the other. The main idea in MDA is that features are modeled once as a Platform Independent Model (PIM), and, in order to ease the task of incorporating newer technologies, tools are used to convert PIM to Platform Specific Models (PSM)
Another distributed programming paradigm, which is an attempt to bridge technological heterogeneity, is that of Web Services [WS02]. Web Services provide flexibility by offering interoperability between software written in diverse programming languages, based on different models, to the extent of running on different operating systems. Web services have, so far, proven to be a very useful means to reuse software and legacy systems by exposing ‘web interfaces’ to specific clients, with an entire protocol stack that deals with other issues, such as messaging, data exchange, discovery in a standards-based manner.

Viewing MDA and Web Services from the perspective of an end-to-end solution to developing a DCS brings to notice standardization, and model-driven development, aimed at facilitating interoperability and reuse. The process of developing DCS using these methodologies, however, is still ad-hoc, and does not incorporate QoS in a formal, rigorous manner. The UA, as mentioned above, attempts to formalize this ad-hoc process by unifying existing as well as emerging models under a common meta-model, the Unified Meta-component Model (UMM), which includes these key ideas: a) a meta-component model (the Unified Meta Model – UMM [RAJ00]), with a mechanism to indicate contracts of software components, b) an integration of the QoS at the individual component and overall system levels, c) the validation and assurance of the QoS, based on event grammars, and d) generative rules with formal specifications to assemble a DCS from an ensemble of components from available choices.

To enable the UA, all the ideas mentioned above with respect to contracts of software components, QoS integration and validation, and DCS assembly, need to be consolidated in one single set, so it can be viewed by different stakeholders that participate in the development of DCS. Some of these stakeholders are domain experts, quality control experts, developers, testers, system integrators and finally, the end users. This ‘consolidation of ideas’ is The Knowledgebase (KB). The KB contains the information that is needed to build a high-confidence, QoS-aware DCS whose properties can be predicted, statically before the system is assembled from parts, and validated dynamically afterwards. The KB contains architecture models and other domain-specific
(Domain Specific Languages, Feature Diagrams) as well as domain-independent (QoS Catalog, QoS Composition and Decomposition Rules) information that can be used, for instance, by component designers to provide comprehensive specifications in terms of abstract components, and by developers to give their concrete realizations.

Since the KB is the central entity in the UA that drives the entire development process, it is important to clearly define various pieces that constitute the KB and understand the interrelationships between them. This will enable the development process of selecting an architecture, decomposing it into components, finding the components using a discovery mechanism, finding the appropriate bridges to facilitate intercommunication between heterogeneous components, and statically/dynamically validating the properties of the system, in an organized manner. The goal of this thesis is to define the architecture of the KB, to provide a clear rationale as to why those pieces are essential, to understand the relationships between these pieces, and to attempt to make a connection between the entities inside the KB with the outside world in the realm of UniFrame research across the breadth of different models and concrete implementations.

1.2 Objectives

In summary, the specific objectives of this thesis are:

- To provide architecture of the KB in terms of the description of the individual pieces and their relationships.
- To provide an individual view for each stakeholder that participates in the UA development process and to provide details of how these stakeholders access/interpret/update information in the KB.
- To provide and discuss the different interfaces that the knowledgebase may expose to the other entities in the UA, such as specification editors, skeletal code generators, QoS Catalog, QoS Composition/Decomposition, QoS Instrumentation, Discovery Services, Profiling and Monitoring Systems and System Generators.
- To provide an easy-to-use tool to enter, update or use the information in the KB.
- To empirically validate the KB by conducting a comprehensive case study.
1.3 Contributions

The contributions of this thesis can be summarized as follows:

- An architecture of the KB that will be the center piece of the UA. The model will constitute essential pieces of the KB and their interrelationship.
- The interfaces exposed by the KB to other entities in the UA, which will assist in realizing the entire UniFrame Approach in terms of concrete implementations.
- A toolkit that provides an interface to access and update the KB to enter domain information, provide component details, QoS parameters, etc.

1.4 Thesis Organization

This thesis is organized into seven chapters. Chapter 1 provides an introduction with the problem definition and motivation, objectives, contributions and thesis outline. Chapter 2 presents related work. Chapter 3 provides a comprehensive discussion of the problem, and the proposed model of the KB. Chapter 4 describes the logical connections of the entities in the KB with other entities in the UA. This is followed by Chapter 5, which provides a case study that shows the entire KB for an example system from the banking domain. Chapter 6 presents the design and implementation of a prototype for the Knowledgebase Toolkit. Chapter 7 mentions possible enhancements and tips for future work that would make this system more complete. Chapter 7 also concludes this thesis by summarizing the work done.
CHAPTER 2. BACKGROUND AND RELATED WORK

This chapter is dedicated to the background work related to this thesis. The UniFrame Project is discussed in detail to start with, which is followed by Domain Engineering and Modeling techniques. Also discussed, as part of this chapter, is the QoS catalog [BRA02] and QoS Attribute Composition and Decomposition [SUN03], and an overview of the UniFrame System-Level Generative Programming Framework (USGPF) [HUA03].

2.1 The UniFrame Project

The aim of the UniFrame project [RAJ01] is to automate the process of integrating heterogeneous components to create DCS that conform to certain quality requirements, and offer a certain measure of predictability. The UA is a mature, life-cycle approach that provides a comprehensive solution to the problem of constructing QoS-aware DCS. The important building blocks of the UniFrame and their roles in the process of complete system generation, integration and validation are briefly discussed in the following sections.

2.1.1 Building Blocks

Figure 2.1 below indicates the building blocks that constitute the UniFrame Approach (UA) along with their inter-dependence. The UA is based on principles of the model-driven design and construction to consolidate vocabulary of a family of DCS, a discovery service that selects contract-aware components described by multi-level specifications using multilevel matching [KAT06], and two-level grammar (TLG) and
event grammar (EG) [BRY02] to formally validate the functionality as well as quality of service (QoS) parameters of the DCS. The UA attempts to address the issues of quality throughout the life cycle of the development process, starting from the specification of components to QoS code instrumentation in components (as well as the resultant system) to validate QoS properties.

The central piece in the UniFrame research is a comprehensive KB which is developed by domain experts. The KB is modeled in a way that it consists of a service-based architecture (modeled by feature diagrams [OLS05]) for a family of systems for the particular type of application under consideration, along with rules for creating, matching and selecting distributed components, rules for semi-automatical generation of a DCS from selected components, and rules for the description, the instrumentation and the measurement of the quality requirements of the generated system. All the other entities in the UA assume that this information exists in a well-defined format to execute their activities. One of the goals of this thesis is to provide a concrete description of the KB’s constituents, so that all the information mentioned above can be formally modeled and stored.

Figure 2.1: The UniFrame Approach (From [DEV05])
2.1.2 Approach

Component developers independently create components using various distributed component technologies such as .NET, Java RMI or CORBA, in accordance with the rules laid down in the feature models described in the KB. These rules mostly encompass the interfaces of components and their interrelationships with other components in the system. The component developers are required to develop a multi-level specification, called the UMM (Unified Meta-component Model) [RAJ00] specification, for the developed components in addition to creating, validating, and deploying them. This UMM specification is a comprehensive contract used to describe components in terms of syntax (method signatures and interfaces), semantics (preconditions, post-conditions and invariants), synchronization (policy and implementation), and QoS (various important parameters of a component such as turnaround-time and throughput). Thus, the component development process follows the principles of design by multi-level contract [BEU99, MEY92], which is an enhancement of the design by contract [MEY92] and the multi-level contract [BEU99] concepts. As far as the component development is concerned, it is an ongoing process, that results in the developed components being deployed on the network, along with their UMM specifications (available in XML form). Once components are deployed, they are available for discovery by the discovery service, which is described below.

The UniFrame Resource Discovery Service (URDS) [SIR02] is the entity which locates and matches these deployed components based on a certain component search request. The URDS is hierarchical, proactive, interoperable, and decentralized in nature. It consists of three major constituents – the Active Registries (AR), Headhunters (HH) and Internet Component Broker (ICB). Component developers register their components with the AR – each AR being specific to a particular domain. The components can belong to different technologies, but the AR is domain specific. The HH is also domain-specific and returns components that match the issued query. Every HH maintains its own ‘Meta Repository’ of components by continually updating with the AR’s. The ICB is responsible for managing the communication in the URDS and consists of sub-entities
such as Domain Security Manager (DSM), Query Manager (QM), Link Manager (LM) and the Apapter Manager (AM). The DSM is responsible for authentication of HH, QM and AR’s. The QM intercepts the query issued by the System Integrator to the HH’s that belong to the same domain as the issued query. The QM may also propagate the query to other ICB’s through the LM which serves as a link between ICBS. The AM is a lookup service for adapter components that are used to bridge heterogeneous software components.

The discovery process is initiated by a query, by a System Integrator, to form a distributed system. This query indicates the nature and the features of the desired distributed system, which may include a combination of a variety of QoS parameters and a type of the desired system. The QM uses the KB to determine a specific design instance out of the families of systems stored in it. Once that instance is identified, the query is decomposed into sub-queries, each indicating specific types of components, along with their QoS features, that are needed to construct the desired distributed system. This decomposition process uses the rules that are described in the KB [SUN02]. These sub-queries are supplied to the URDS for locating the appropriate components matching with the required criteria. Once components are located, these are presented back to the System Integrator for selecting, in case there is more than one candidate for a given sub-query.

Once selected by the System Integrator, the components are used by the system generator [HUA03] to generate the DCS. The construction process utilizes the generation rules, which are described in the KB, that express the architecture of the system design. Also, the construction process instruments the necessary QoS-related code into the integrated system. The generation process uses the TLG and the instrumentation process is based on event grammars [CAO02].

The hallmark of the UA is its way of addressing the issues of Quality, Confidence and Predictability by using a combination of static and dynamic QoS property
composition – static before constructing the system and dynamic after the system is generated. The values returned by this composition for the various QoS properties serve as a measure of the overall performance of the generated system. The quality of the DCS is ensured by the use of a standardized KB, the creation of components using the design by multi-level contract and associated correctness principles, and the selection of appropriate pieces by the URDS. The generation process is carried out by wrapping up the components, if needed, so as to bridge the heterogeneity, and by generating glue, through which the wrapped components can communicate seamlessly. The static prediction of QoS properties happens before the system is constructed, so as to ensure that the resultant system is feasible as far as the QoS requirements are concerned.

The discovery, generation, integration and validation are performed in an iterative manner, thus providing a pragmatic solution to the realization of a DCS. If the discovery process does not yield any components, or if the System Integrator does not find the discovered components to be satisfactory or the integrated system does not meet the desired QoS requirements, then the query has to be modified and reissued, and it will be reprocessed by the URDS, following which the generation, integration and validation processes will repeats.

2.2 Architecture Modeling and Domain Engineering

One of the most important pieces of the KB is domain information about families of system, which makes Domain Engineering an important task that needs to be carried out by domain experts and analysts, before they can provide the KB with complete information about the specific domain. This section summarizes some of the popular domain engineering methods and techniques, most of which use Generative Programming [CZA00] concepts, and are not specifically oriented toward distributed systems and the challenges associated with them. They are presented as part of related work as alternatives to analyze and engineer domains in general; the model of the KB presented as part of this thesis employs a variant of [VAR02], which discusses a step-by-
step approach to examine, document and model the problem space of a domain. An overview of model-driven architecture (MDA) [OMG01] and generative programming is provided to start with, followed by an overview of domain engineering techniques. A summary of [VAR02] follows, which lists the steps and deliverables of the approach employed.

2.2.1 Model-Driven Architecture (MDA)

Model-Driven Architecture (MDA) [OMG01] was launched by the Object Management Group (OMG) as a software design approach that supports ‘model-driven engineering’ of software systems. Enterprise applications today require an approach that is flexible enough to accommodate changing business needs, and facilitating reuse of existing efforts in the context of new business needs, in a timely manner, also assuming that the target infrastructure may vary. The ideas that have driven the formalization of MDA by OMG are Service-oriented Architectures [SOA], where applications are viewed as federations of services that communicate via well-specified contracts, permitting increased flexibility and reuse, and secondly Software Product Lines [SEI], which promotes planned reuse of assets, along with increasing the level of automation by applying well-defined patterns as a way to transform descriptions of a solution from one level of abstraction to a lower level of abstraction.

Considering the viewpoint of separating business-oriented decisions from lower level platform and implementation decisions, the OMG created a conceptual framework, called the MDA, which can be used by application architects as a blueprint for expressing enterprise architectures. The MDA employs several open standards, developed by OMG, such as Unified Modeling Language (UML) [OMG03], Meta-Object Facility (MOF) [OMG02], XML Meta-data Interchange (XMI) [OMG05], and Common Warehouse Meta-model (CWM) [OMG01a, OMG01b, OMG01c]. Eventually, the enterprise architectures described by the above standards can be transformed into any open (or proprietary) platform such as .NET [NET], J2EE [SM02] or CORBA [OMG99].
The concept of Model-driven Architecture rests on the notion of models expressed using well-defined notations around which systems can be built by executing a series of transformations between these models. MDA consists of a set of layers and transformations that constitute the conceptual framework and vocabulary of MDA: Computation Independent Model (CIM), Platform Independent Model (PIM), Platform Specific Model (PSM) described by a Platform Model (PM), and an Implementation Specific Model (ISM). Distinguishing between different kinds of models allows the process of system development to be viewed as refinements between different model representations. Models can be classified in terms of how explicitly they represent the aspects of the platforms being targeted. Choice of language, hardware, network topology and communication protocols dictate the software development process, and hence, form the ‘elements’ of the solution platform. MDA thus helps to focus on business functionality needs, separate from the solution platform. ‘Platform’ is also context-dependant in the sense that it could imply an operating system in some context, and a programming environment or a communication protocol or just a hardware topology in another. The point to be considered is that irrespective of this context dependency, more stress has to be given on the usage of a particular model at different levels of abstraction, rather than the precise definition of ‘platform’.

The MDA approach describes system functionality using the PIM, using an appropriate Domain Specific Language (DSL) that captures information about the domain in question. Given a PM corresponding to a particular technology (or solution platform, such as CORBA or J2EE), the PIM can be translated to one of more PSMs that can be actually executed on systems using variants of Domain Specific Languages or even General Purpose languages such as Java [SM03] or C# [MS]. These transitions can be carried out by OMG-compliant automation tools.

MDA, thus, focuses on separating the business needs from implementation and realization issues, allowing realization technologies to evolve independently, and promoting the notion of model transformation. This thesis borrows the idea, from MDA,
of developing a model for abstractly describing the structure of a DCS – although the approach of this thesis incorporates the notion of families of system as opposed to a single system, and oriented towards tackling the challenges imposed by DCS and incorporating QoS in a formal manner.

### 2.2.2 Generative Programming

Generative programming [CZA00] is concerned with automating the process of software development by generating systems from a system family based on given specifications. A system family is a group of systems that can be built from a common set of assets. This requires that the model of the system family be developed, along with a way to specify system requirements, the availability of components from which the system can be assembled, and mechanism of mapping the problem specification onto the required components (out of the available ones) to generate the system using a configuration generator (or system generator).

[CZA00] formally defines Generative Programming as: “Generative Programming is about manufacturing software products out of components in an automated way. It requires two steps: a) a design and implementation of a generative domain model, representing a family of software systems (development for reuse). This model includes also a domain-specific software generator; b) given a particular requirements specification; a highly customized and optimized end-product can be automatically manufactured from implementation components by means of generation rules (development with reuse)”. The methods presented in [CZA00] can be applied at the level of classes and procedures as well as families of large systems.
As shown in Figure 2.2, generative programming requires the development of a generative domain model (GDM) that consists of a problem space, a solution space, and the necessary configuration knowledge to map them together. The problem space consists of application concepts and features that an application programmer can use to express the requirements for generating systems from a system family and is explored using domain engineering techniques described later in this chapter. A feature can be defined as an attribute of the system that directly affects the end-user. The solution space consists of the component implementations. The configuration knowledge takes into account considerations such as illegal feature combinations, default settings, default dependencies, construction rules, and optimization rules. A configuration generator (or system generator) that is responsible for checking to see if the system can be built under the given constraints, completing the specification by computing defaults, and assembling the implementation components, can be created.

The problem and solution spaces are separated so that both the spaces can evolve independently – new components can be added to the solution space as long as they cover the functionality described by the problem space, thus permitting the existing client code to remain unaltered, since the system generator manages the mapping of the problem
specifications onto the configurations of the new components. Thus, adding new components only requires modifying the generator as per [CZA00].

The main steps necessary in generative programming are identified in [CZA00] as below. It is recommended that for best results, these steps should be applied iteratively and not necessarily in the same order.

- Domain scoping
- Feature and concept modeling
- Designing a common architecture and identifying implementation concepts
- Specifying domain specific notations for ordering systems
- Specifying the configuration knowledge
- Implementing the components
- Implementing the domain specific notations
- Implementing the configuration knowledge using generators

The KB modeled in this thesis (detailed in Chapter 3) uses the notions of families of systems and system generators, although the focus is on the structure of the families of systems. The KB is also a GDM at a conceptual level but modeled differently – the analysis is based on the roles played by various stakeholders with respect to the KB, such as Domain Knowledge Experts, Component Developers, System Integrators, etc. the KB is modeled from the viewpoint of each stakeholder.

2.2.3 Domain Engineering Methods

Some of the domain analysis and engineering methods have been presented as part of this section.
2.2.3.1 Overview of Domain Engineering Techniques

This section presents a high-level overview of a few prominent domain engineering techniques such as Feature-Oriented Domain Analysis (FODA) [SEI03], Organization Domain Modeling (ODM) [SIM96], Family-oriented Abstraction, Specification and Translation (FAST) [WEI94] and Software Design Automation (SDA) [WID98].

2.2.3.1.1 Feature-Oriented Domain Analysis

Feature-Oriented Domain Analysis [SEI03] originates from the Software Engineering Institute and is based on an in-depth study of other domain analysis approaches. The primary goal of this method is to develop generic products that have domain-wide application by applying the concepts of (a) abstraction – to abstract differences between applications within a domain and (b) refinement – to reintroduce any factors that will make the application unique. FODA consists of three phases – Context Analysis, Domain Modeling and Architectural Modeling. Context analysis is aimed at defining the scope of the domain, and analysis of relationships between the domain and external elements, and evaluation of variability of the relationships and external conditions.

Domain modeling consists of feature analysis, information analysis, and operational analysis. Feature analysis deals with capturing and modeling the end-user’s understanding of the features of the applications within that domain. This phase delivers a complete feature model that includes a structure diagram that depicts hierarchical decomposition of features that indicates type of feature (mandatory, optional or alternative) and its description along with composition rules. Information analysis (or Entity-Relationship Modeling) defines, analyzes and captures the domain knowledge that is essential for implementing applications. The resultant model may be an entity relationship model or an objected-oriented model. The last aspect of domain modeling is operational (or functional) analysis that identifies behavioral and functional
commonalities and differences of applications within a domain. The behavioral aspect includes how an application responds to events, inputs, states, conditions and state transitions whereas the functional aspect comprises the structure of the application in terms of inputs, outputs, activities, internal data, logical structures and dataflow relationships. Architectural Modeling, the final step of FODA, aims at providing a software solution to the problems discovered in the earlier phases, in a way that the model can be adapted to any future technological changes or even changes within the problem. Architectural layering can be used to define the architecture at different levels of abstraction to facilitate reuse independent of the layer; the layers are Domain Architecture layer and the Domain Utilities layer. The result of this phase is an application domain-oriented architecture, which packages functions and objects into software modules. The concurrent tasks are identified and communication and synchronization between these tasks is also defined, concluding with the design of each task as a sequential program.

2.2.3.1.2 Organization Domain Modeling

Organization Domain Modeling (ODM) [SIM96] is a formal and highly customizable domain engineering approach, formalized in [SIM96], and is most useful when applied in the context of mature, stable, and economically viable domains. The highlight of this approach is its heavy emphasis on stakeholders and their individual goals throughout the lifecycle. ODM is applicable to both families of systems (vertical domains) as well as parts of systems (horizontal domains), and hence, can be applied in re-engineering portions of legacy systems as well. The method is composed of three distinct phases: Plan Domain Engineering, Model Domain and Engineer Asset Base.

The Plan Domain phase focuses on understanding stakeholder requirements and scoping the domain. This is carried out over three sub-phases. First the stakeholders are determined and their objectives and overall project objectives are analyzed. This is followed by domain scoping, and finally domain definition, which consists of defining
domain boundaries though rules and examples of systems which are to be included, and identifying the main features of the other systems in this domain, and analyzing relationships between this and the other domains. The Model Domain phase is related to gathering and documenting relevant domain information – this is achieved by integrating information from domain experts and identifying relevant system features, followed by describing the domain by creating of a lexicon of domain terms and then modeling the semantics of key domain concepts, followed by modeling variability of the concepts through identification and representation of features. The final phase ‘Engineer Asset Base’ constitutes scoping, architecting and implementing an asset base for the relevant domain. In the final sub phase of ‘Engineer Asset Base’, the implementation is planned and the assets are implemented; the infrastructure including asset retrieval and qualification mechanisms is also implemented.

2.2.3.1.3 FAST and SDA

There are other techniques such as FAST [WEI94] (a domain engineering technique that focuses on software families or product lines, with a broad objective of creating processes and assets for producing new members of a program family as fast and cheaply as feasible), and SDA [WID98], which is a method for developing tool-supported formal specification languages or application generators. The primary drawback in all these approaches is that they promote use of models that are outdated, and that none of them advocate the use of open standards, such as UML. UML [OMG03] has become the de-facto standard for modeling enterprise software systems of today, hence, any domain engineering technique that has to thrive, should ideally incorporate the usage of UML wherever applicable.

An important piece of the KB is the domain knowledge, and the usage of a formal engineering method to model domain knowledge is thus a must. The above approaches summarize the work that has been done so far for domain engineering. The following section presents a mechanism of modeling the problem space of a variable domain
[VAR02], which incorporates the merits of some of the domain engineering techniques mentioned above and the use of UML notations wherever appropriate, and is oriented toward the development of DCS. This domain engineering technique is used to model the ‘domain information’ aspect of the KB as part of this thesis.

2.2.3.2 Modeling Problem Space for a Variable Domain

This method [VAR02] for domain engineering, or ‘modeling the problem space of a variable domain’ integrates ideas from existing domain engineering methods (discussed in previous section) and incorporates various UML notations, as and when appropriate, along with considering the concerns presented by distributed heterogeneous systems. The method employs a ten step approach, with fixed set of deliverables associated with every step. Some steps have to be applied iteratively, or in conjunction with other steps to come to more concrete conclusions. The steps and deliverables are explained below:

Step 1: Describe the Problem Domain

This step aims at gaining and initial understanding of the problem domain, by developing a problem statement, and listing down the general capabilities of the applications that fall under the domain. A listing of existing applications in the domain is also developed to gain better understanding.
Deliverables: Problem Statement, Description of General Capabilities, Existing Applications.

Step 2: Identify Stakeholders

The goal of this step is to determine the list of people who would be directly involved with the project. This group may include senior management, project management, end uses, customers, other service recipients, service providers, investors, developers and regulators. Developers who have worked on legacy systems in the domain are also included, since they have better insights into availability of existing components.
Deliverables: Potential Stakeholders and Domain Experts.
Step 3: Expand Domain Definition

This step involves moderated and repeated meetings with stakeholders to get their inputs on the exact domain definition, in terms of a more expanded domain definition and boundaries of the domain. Defining the boundaries is important since it delineates the types of applications that are and are not included in the project. A list of potential domain information sources is also consolidated as part of this step.
Deliverables: Project Objectives, Domain Description, Domain Boundaries, Potential Sources of Information.

Step 4: Acquire and Document Relevant Domain Information

The goal in this step is to gather and document relevant domain information, and this step must be repeated as many times as may be needed to gain sufficiently thorough understanding of the problem. This may be achieved by consulting with stakeholders and other information sources as identified in the previous steps. A dictionary is created that lists the common terms in the domain with their exact meanings. Following this, common and variables features of the applications in that domain have to be identified. This step concludes with a documentation of any knowledge regarding communication that takes place between the new domain and any outside domain entity, also listing the type of communication and communication interfaces, if any.
Deliverables: Domain Dictionary, List of Common and Variable features, Communication with outside entities for the domain.

Step 5: Model the Domain

The goal of this step is to model the new domain, and to develop models that will lend better understanding of the aspects of the problem, to the people involved. An extended version of the FODA Feature Diagram is used in conjunction with UML diagrams such as use case diagrams, sequence diagrams and collaboration diagrams. The domain dictionary should be updated accordingly.
Step 6: Validate Models, Dictionary and Domain Descriptions

The goal of this step is to validate the models developed in the previous step, the dictionary and also the domain description, with the appropriate stakeholders. This can be started with identifying variants (description variants, operator variants, connection variants, control variants as defined by [JAC01]) and updating the models, dictionary and domain description after the variant analysis.

**Deliverables:** Possibly updated Domain Models, Domain Dictionary and Domain Description.

Step 7: Create Decision Model

The decision model is created as part of this step, which describes the process an application engineer would need to follow to specify the requirements of a new family member using the commonalities and variability identified in the previous steps and models. The model should include all the decisions that need to be made. After following the decision model, the application engineer should have a list of what features/components will need to be included in the application created for the particular problem. The decision model can be created using one or both of the following styles – the UML Activity Diagram or basic flow-charting notation.

**Deliverables:** Decision Model (Activity Diagram or Flowcharting notation style).

Step 8: Create Application Domain Specific Language (ADSL)

This step constructs the ADSL for the problem domain. The ADSL allows expression of parameters of a specific application in a consistent way for a particular domain. As per the primary goal of generative programming (Section 2.2.2), the application can be automatically generated from the text description of the ADSL. [VAR02] slightly alters the ADSL proposed by Deurson and Klint [DEU02]. As per [DEU02], the feature expression can consist of an atomic feature, a composite feature, an optional feature, mandatory features, alternative features, non-exclusive features, default feature values, etc. [VAR02] primarily adds the notion of non-essential features and the
notion of communication between components to the ADSL in [DE02]. Other extensions and BNF grammar for the modified ADSL are detailed as part of the report. 
Deliverables: Application Domain Specific Language (ADSL).

**Step 9: Validate Decision Model and ADSL**

The main task for Step 9 is to consult with the domain experts and have the decision model and ADSL (created in the previous two steps) validated. The discussion should identify usability problems, and feedback from both the application engineers and the domain experts should be used to improve the model and/or ADSL. 
Deliverables: Possibly updated Decision Model and ADSL.

**Step 10: Get Final Signoff**

The important task in this step is to get the final signoff from all the major stakeholders by presenting all the deliverables in the last 9 steps. Any open issues or issues involving strong disagreement, have to be indicated as part of a signoff sheet.

List of Artifacts created:

- Domain Description
- Dictionary
- Feature Model
- Use Case Diagrams
- Sequence Diagrams
- Collaboration Diagrams
- Domain Model
- Decision Model
- ADSL

This approach has been used, with some annotations and modifications to describe the domain knowledge and its representation more accurately. These will be discussed in detail as part of the following chapter on the architecture of the KB. To
summarize, most of the artifacts produced as part of this approach are incorporated under ‘Architecture Information’ of the KB.

2.3 QoS

This section discusses the work that has been done, as part of the UniFrame project related to Quality of Service. The QoS Catalog is described along with an overview of QoS Composition and Decomposition.

[BRA02] describes the Quality of Service Attribute Catalog which gives significant details of several QoS attributes that are of relevance to distributed computing systems. The catalog provides information about every attribute, such as detailed description, motivation, intent and applicability of the attribute, the models and metrics used along with a list of factors that heavily influence the attribute. The catalog provides the evaluation procedure to compute the value of that attribute for a single component and the formulae to quantify the attribute for a particular component.

This is followed by other information about the attribute such as the result type, nature (Static/Dynamic), set relationship (Universal/Subset/Existential/Specific) and application dependency (Application dependent/independent). Other useful information, such as consequence of the attribute, related parameters and aliases, domain of uses (either a specific domain or domain independent), and a listing of resources about the attribute, is also made available as part of the catalog.

The catalog presently contains the following attributes: Dependability, Security, Adaptability, Maintainability, Portability, Parallelism Constraints, Ordering Constraints, Priority, Throughput, Capacity, Turn around Time and Availability. Some of these attributes are quantifiable in a more obvious way, and there are many others that need a thorough analysis and an exploration of various measures/indices, so that they can be used in conjunction with one another to compute a value for each attribute.
The most challenging aspect about QoS attributes in the context of distributed computing systems is their composition and decomposition. A distributed computing system is made up of components that interact with one another, and to determine if the combination of components satisfies the QoS requirements of the combined DCS, the QoS attributes of the individual components have to be combined in some fashion. Each attribute has different semantics when it comes to composition and decomposition [SUN03] which is also detailed as part of the QoS catalog. For each attribute, a clear rule stating the composition of values to obtain a value of the system as a whole is provided. There is also another decomposition rule that assists in calculating individual component values from that of a system. The composition and decomposition rules stated as part of the catalog play an important role in static prediction and dynamic validation of the assembly of components, which in turn, gives an estimate of the quality of the overall system.

This QoS catalog has been used in the KB modeled as part of this thesis. As discussed above, the catalog contains rules for composition and decomposition of QoS attributes, and these rules are used for static prediction of QoS properties before system generation.

2.4 The UniFrame System-Level Generative Programming Framework (USGPF)

This section provides an overview of The UniFrame System-Level Generative Programming Framework (USGPF) [HUA03]. The UniFrame Approach has two levels – the component level and the system level. [HUA03] describes the system level generative programming framework.

The USGPF consists of three major pieces. A brief description of the structure and usage is detailed below:

- **The UniFrame Generative Domain Model (UGDM):**

  The UniFrame Generative Domain Model (UGDM) is the first part of the UniFrame System Level Generative Programming Framework and it captures the
common and variable properties of a DCS family. The UGDM consists of three parts – general information, problem space and solution space. General information includes a description of the domain being modeled with information such as domain name, system family name and the author and version. Problem space includes Use Case Model, QoS Requirement Model, Architectural Model in Hierarchical form and System Level Multiplicity Model – these models provide information about the functional, QoS and architectural aspects of the DCS family respectively. These models put together provide domain specific concepts as seen by the users of the resulting DCS in the domain. Use cases model externally visible behavior of the systems describing functional aspects and consist of commonalities and variations of the use cases and secondly, the constraints between them. Solution space which contains the necessary configuration knowledge to provide solutions for a DCS family. The solution space consists of Architecture-related models, design-feature-related models and QoS related models.

Details of these models and their exact definitions along with illustrating examples can be obtained in [HUA03]. This information about the common and variable features of the domain, QoS, architectures and configuration knowledge (solution space) for the basis of the ‘domain knowledge’ of any problem domain. It covers the problem domain from the perspective of the stakeholders and the application engineer, and this information is used for the other UA activities (as described before in Figure 2.1). The UGDM corresponds to the KB piece and constitutes the knowledge of the problem domain that is used by the other entities (such as System Integrator, discovery service) to achieve the UA.
The UniFrame UDGM Development Process (UGDP):

The UniFrame UDGM Development Process (UGDP) is the process by which the UDGM is created for a particular domain and covers the generative domain engineering of the UniFrame Approach. The UGDP is a use-case-driven, architecture-centric and iterative process. The important thing to note about the UGDP is that it must be domain-independent, i.e., it is absolutely essential that it is repeatable across multiple domains. The UGDP consists of three phases: Domain Analysis, Domain Design and Ordering Language Design.

Domain analysis consists of domain definition (description, scoping and context analysis) and domain modeling (modeling functional requirements, identifying and modeling domain key concepts and QoS requirements).

Domain design is the second phase which is carried out to develop the layered architecture for a DCS family and to develop various QoS related models. This phase consists of designing a Layered Architecture, creating Component and Sequence Diagrams, refining the Critical Use Case Model to Abstract Component Level, identifying Component Interfaces and Communication Patterns, refining Critical Use Case Model to Function/Interface level, refining the Architecture Model in disjunctive normal form from Component level to Function/Interface level, mapping Architecture Model in disjunctive normal form to Critical Use Case Model (Function/Interface level), creating Abstract Component Model and finally creating the QoS Composition and Decomposition Model.

The last phase is that of ordering language design. The Ordering Language is the interface that the application engineers employ to order the concrete systems from a DCS family. The Ordering Language is also a Domain Specific Language (like the UDSL) can be represented in various forms (textual, tabular, graphical or natural language). If the UDSL were to be viewed as an ordering language, then as per [HUA03] identifies 3 levels: system architecture, functionality (inclusive of communication patterns) and QoS.
A mapping from the ordering language (Tabular form is considered as part of [HUA03]) to the UGDM is then designed as part of this phase, and this process is, to a great extent, domain dependent.

The UGDP is an incremental and iterative process which results in an incremental evolution of the UGDM over several iterations, finally creating a mature and stable UGDM. The UGDP is the process that is carried out to create the KB as shown in Figure 2.1 and results in the UGDM which constitutes the feature-related, QoS and architectural information of the required problem domain, as described in the previous bit.

• The UniFrame System Generation Infrastructure (USGI):

The UniFrame System Generation Infrastructure (USGI) is the infrastructure for realizing the USGPF. The USGI architecture shows the different entities that participate in the realization of the USGPF such as UGDMKB Generator, UGDMKB, Order Processor, UGDMKB Builder, System Generator, Application Programmer, Wrapper and Glue Generator, URDS (UniFrame Resource Discovery Service) and finally, the Application. The USGI workflow is also detailed, which shows the steps in generating the system along with algorithms and data structures used. Also, the USGI Object flow is described which is a special activity diagram that includes participating objects (modules in the USGI), and emphasizes the flow of control among different modules and shows the dependency relationships between them. The USGI helps in the automatic generation of DCS from a given DCS family using the information created as part of the UGDP and the resultant UGDM, and is carried out by the entities listed above. Their exact roles are detailed as part of [HUA03].

The USGPF is the framework described as part of [HUA03] and describes a view of the KB in terms of the UGDM, the process to create it (UGDP) and the process to realize the framework in terms of USGI. The KB proposed as part of this thesis also aims at providing a comprehensive structure for not just the domain information and QoS constraints, but other important constituents such as the entire catalog of QoS attributes,
the discovery process and rules for system generation and integration, along with property prediction and validation mechanisms. This thesis makes the KB more generic over several domains, at the same time provide a precise format for each constituent.

2.5 Conclusion

This chapter described the work that is related to the problem of modeling the architecture of the KB. The next chapter describes this architecture, followed by a chapter on how the existing concrete implementations under the UniFrame Project relate to the entities in the KB.
CHAPTER 3. THE KNOWLEDGEBASE ARCHITECTURE

The previous chapter summarized some of the related work associated with the contents of the KB and the approaches that have been used to model the contents. This chapter provides the details of the KB architecture in terms of the entities present in the KB and their interrelationships.

3.1 The Architecture of the KB

Figure 3.1 shows the architecture of the KB. Each one of the constituents shown in the architecture has a certain role to play in the process of either developing information about domains, designing components as per the guidelines given in the domain information, or associating relevant quality of service attributes with domains. The role that each entity plays with reference to other entities and with respect to the process of system generation is detailed with the description of every entity. The remainder of this section focuses on the KB architecture and is followed by a section on the process of system generation.

The KB must be comprehensive enough to constitute all the major functions that need to be carried out while designing and constructing a DCS. To start with, the KB must contain information about families of systems of the problem domain. It should contain system architectures that are generic, flexible and customizable to create a certain ‘instance’ of a system, and also a mechanism to specify valid combinations of the features in the system, and their interactions. The KB must contain a listing of QoS attributes, and a mechanism to compose properties of components to predict the quality
of the overall system. There are other issues, such as validating the QoS attributes, and handling heterogeneity that inherent in a DCS in order to generate a system from parts.

The KB architecture is divided into six major pieces; each piece represents a certain category of information that must be present in the KB for it to be complete and comprehensive. Figure 3.1 below shows the architecture in terms of the six pieces:

![Figure 3.1 KB Architecture](image)

The six pieces of the KB are Domain Information, QoS, Component Information, Technology, Query Decomposition and Multilevel Matching Algorithm. Each piece is described in the consequent sections. A case study of a banking domain example follows in Chapter 6, which provides a detailed example of an instance of KB for the banking domain.
3.1.1 Domain Information

This piece of the KB deals with storing architecture blueprints for families of systems of the problem domain. As mentioned in the previous chapter, an approach similar to that discussed in [VAR02] is used to capture and model domain information; hence, most of the artifacts produced by that approach appear in this part of the KB along with other information related to QoS. For each domain, a brief description is stored, which is followed by a dictionary of common terms in the domain. Models such as feature model and decision models are stored, along with UML use case diagrams to model use cases in the problem domain. A variant of UniFrame Domain Specific Language (UDSL) [HUA03] is used to model the domain specific features and their interrelationships – the variant is termed as UMM-annotated UDSL, since it consists of the basic UDSL annotated with the UMM notation as discussed in [NEI04]. Associations between the domain and relevant QoS attributes are also stored. Finally, a mechanism to generate wrappers is also included as part of the architecture information. The following sub-sections describe all these entities in detail.

3.1.1.1 Description

The description of a domain primarily includes the broad level objectives as agreed upon in consultation with stakeholders identified in the domain. This information is filled in after performing an initial high-level analysis of the problem statement and identifying primary stakeholders. The analysis is based on the requirements of the DCS family in terms of what is expected out of systems in the family and is stored in the form of a textual description as detailed below.

Table 3.1 Structure of Domain Description

<table>
<thead>
<tr>
<th>Domain Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Project Objectives</td>
</tr>
<tr>
<td>2. Domain Boundaries</td>
</tr>
<tr>
<td>3. Potential Sources of Information</td>
</tr>
</tbody>
</table>
• Project Objectives: This includes what is expected out of the systems in this domain in terms of the quality of work that needs to be satisfied under the given deadlines (for e.g., budget, time). This is a broad level objective of the project, which encompasses the key attributes of all the systems in the selected problem domain.

• Domain Boundaries: This includes a description of what type of systems the domain is made up of and what is the overall process that would be followed in the systems that fall under the selected domain. This may include constraints on or expectations from the applications within the domain. This also establishes the scope of the domain so as to describe exactly what the domain would be concerned with, and what it would not consider. For instance, a certain domain may only consider basic banking systems, and will not consider certain specialized functionalities that come with banking systems, so this has to be documented as part of domain boundaries.

• Potential Sources of Information: This includes sources that can potentially provide information through concepts or examples, and may be of interest to the domain. This may include people (analysts or accountants, application engineers who have worked on applications in related domains), textbooks (source of useful information about legacy systems for instance), web links and industry standards (domain specific).

The domain experts and other stakeholders of the system, such as design/application engineers who have previously worked on other applications in the domain, and users of the domain, are interviewed from the perspective of gathering as much information as possible, about the high-level design objectives and boundaries of the problem domain. This phase typically corresponds to the initial ‘requirement analysis’ phase in industrial terms. The structure for the information used as part of this thesis is described in Table 3.1. This piece, thus, gives a high-level overview of the domain and is essential in order to keep the complete perspective in mind while designing and modeling the more specific features of the system. It is these objectives which will be refined to create all the models and fine-tuned sub-goals in the consequent pieces, as discussed
further. A banking domain case study has been detailed in Chapter 5, where all the artifacts are presented.

3.1.1.2 Dictionary

The domain dictionary stores the definitions of the important relevant terms that are associated with the problem domain. This dictionary has to be formulated by a domain expert by consulting with the stakeholders identified previously, as many times as needed, to thoroughly understand all the common terms (words and expressions), thus exposing all the nuances associated with the terminology of the domain. As said before, this dictionary must be updated as many times and whenever necessary, in order to reflect any changes or additions that may be revealed in subsequent analysis.

An alternative approach to describing ‘glossary’ terms for a particular domain is using ontologies. Ontologies are usually used to describe relationships between objects (super-class, sub-class, etc.). As part of this thesis, a list of terms is maintained along with their definitions. A term can be described in terms of another ‘term’ (synonyms) as shown below in Table 3.2. More examples are listed as part of the banking domain example in Chapter 5.

<table>
<thead>
<tr>
<th>Table 3.2 Structure of Domain Dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>S ::= &lt;Dictionary_Entry&gt;</td>
</tr>
<tr>
<td>&lt;Dictionary_Entry&gt; ::=</td>
</tr>
<tr>
<td>TERM: &lt;Term&gt; - MEANING: &lt;Meaning&gt; &lt;Dictionary_Entry&gt;</td>
</tr>
<tr>
<td>TERM: &lt;Term&gt; - MEANING: &lt;Meaning&gt;</td>
</tr>
<tr>
<td>TERM: &lt;Term&gt; - SYNONYM-TERM: &lt;Term&gt; &lt;Dictionary_Entry&gt;</td>
</tr>
<tr>
<td>TERM: &lt;Term&gt; - SYNONYM-TERM: &lt;Term&gt;</td>
</tr>
<tr>
<td>&lt;Term&gt; ::= WORD</td>
</tr>
<tr>
<td>&lt;Meaning&gt; ::= TEXT</td>
</tr>
</tbody>
</table>
3.1.1.3 Models

This section discusses two models that need to be stored for any given domain – the Feature Model (modeled with an Extended FODA Feature Diagram) [HUA03] and a Decision Model (modeled either with the UML Activity Diagram [BEL00] or a simple flowchart notation). Each diagram is discussed below.

3.1.1.3.1 Feature Model

The ‘feature model’ models the ‘features’ of a system by defining the common and variable features of concept instances and the dependencies between the variable features. As stated in [CZA00], there are two definitions of features found in domain engineering literature: 1) An end-user-visible characteristic of a system, which is the definition used in Feature-Oriented Domain Analysis (FODA); 2) A distinguishable characteristic of a concept (e.g., system, component) that is relevant to some stakeholder of the concept. Since the analysis of the KB is from the perspective of different stakeholders, and being a more generic definition, the latter has been used throughout this work. Constraints (such as multiplicity constraints between features, or mappings between features at different levels of abstraction) that cannot be expressed directly through the feature model have to be dealt with separately. In this case, they are dealt with UMM-annotated UDSL discussed in section 3.1.1.5. The Feature Diagram notation models the features in a hierarchical manner, by decomposing every feature until it is brought down to a level of interest to the user. This means that the root of the tree (made up of SYSTEM, SUBSYSTEM and Abstract Components) would be the target system, the inner nodes are the subsystems, and the leaves are abstract components. Abstract components are the components that implement the actual functionality; i.e., they correspond to concrete component implementations.

Features [CZA00] are of two types – ‘mandatory’ features (a feature that must be included if its parent is included) and ‘optional’ features (a feature that may be included if its parent is included). A mandatory feature is represented by a box with an incident
edge ending in a filled circle. An optional feature is represented by a box with an incident edge ending in a hollow circle. There are also two ways to group features – ‘alternative’ set (if the parent of the set is included, then exactly one feature in the alternative set must be included) and ‘or’ set (if the parent of the ‘or’ set is included, then one or more features from the ‘or’ set may be included). An ‘alternative’ set is represented by edges connected by an arc. An ‘or’ set is represented by edges connected by a filled arc.

Feature diagrams are represented in a textual manner using a domain specific language, or the UMM-annotated UniFrame Domain Specific Language (Section 3.1.1.5), which provides notations to describe various types of features, constraints and grouping, along with several other notations. The feature diagram is more intuitive in some ways since it depicts the interrelationships between features more explicitly, but is not as expressive (in terms of described fine-grained details, such as constraints, as mentioned above) as the textual DSL. So, the feature diagram notation is used to model the features of the problem domain, and the constraints not described through the feature diagram, are listed as part of the DSL, in a more precise, formal manner (BNF).

A feature model (or a feature diagram) has to be created to model the common and variable features of the applications that fall within a problem domain. This feature diagram is a powerful tool for modeling features and their relationships (in the form of a hierarchy), and is useful for the System Integrator to customize the architecture (by making certain choices about optional features, for instance). It is also useful for component developers to provide concrete implementations of the abstract components (leaves), since the tree structure depicts exactly where a certain feature fits in the context of the system as a whole. The feature model in conjunction with the UMM-annotated UDSL provides a significant insight into the construction of distributed computing systems, in terms of interrelationships between concrete component implementations, and hence, is included as part of the KB for reasons mentioned above. Any new information revealed while constructing the feature diagram must be appropriately propagated to the domain dictionary.
3.1.1.3.2 Decision Model

As discussed in [VAR02] and described in Section 2.2.3.2, the decision model must be created to document any decisions that have to be made by the application engineer, while constructing the system. Since this model attempts to document the decisions that have to be taken by the application engineer, it relieves the application engineer of the responsibility of taking the core domain-specific decisions, which would usually need a domain expert.

The decision model can be derived from the feature diagram by the domain expert; mandatory features need not be included in the decision model since they have to be present as part of any application within the domain. Choices and implications of the optional features should be included in the decision model, since they are the features that potentially will cause variations between applications across the domain. The decision model should also lay out the logical order, if any, of making decisions about choosing certain optional features, since there may be an inter-dependency involved, so that the application engineer can make the choice by reviewing the implications of the decisions.

There are two notations that can be used to construct the decision model. One of the notations is that of the UML Activity Diagram. Using the activity diagram notation lends more meaning to the decision model, when it comes to depicting decisions made concurrently, and also where a strict order needs to be imposed on decisions as mentioned above. An activity diagram is a simple, more standardized, and hence, the preferred notation for problems related to DCS. The other notation is to use a flowchart to show the decisions in a simple and more straightforward manner. While this may be simpler and elegant, it may not be as succinct as the activity diagram notation.

The pictorial notation of the UML Activity Diagram depicts the decision model in a more intuitive manner, but the basic idea is to incorporate feature(s) based on the choices made. The decision model stored as part of the KB is described in Table 3.3 below.
A concrete example of the decision model of the banking domain follows in Chapter 5. This example has the pictorial representation of the decision diagram shown as a flowchart, and the language generated by the grammar in Table 3.3. The language generated is stored as part of the KB for the problem domain, i.e., banking domain. This language contains the decision model that can be used by the application engineer during construction of the DCS. The role of the application engineer, in case of the UA, is carried out by the System Integrator, i.e., the System Integrator decides (based on the query) what features (and, hence, the exact optional feature list) need to be included as part of the resultant DCS.

### 3.1.1.4 Diagrams

The diagrams mandated by [VAR02] are the UML Use Case Diagrams, Sequence Diagrams and Collaboration Diagrams. These diagrams highlight many different aspects of the domain, such as the its interaction with entities outside the domain and interactions between the entities in the domain, and if drawn in the context of the feature diagram (as recommended in [VAR02]), can give comprehensive information about the functionality and behavior of the systems falling under the problem domain. As part of the approach used to develop this KB architecture, the UML and Sequence Diagrams are used to show the entities (actors, i.e., users or other systems outside the domain and processes) and the sequence of the activities that take place within them.
Use case diagrams [UML] are a powerful notation to show how actors, i.e., outside entities, interact with the applications within the domain, and also provide an insight into how the system is perceived by entities foreign to this domain (such as clients or end users of the system). There may be any number of use case diagrams documented for a given domain; there is no upper limit as such, but as long as there is sufficient variation in the use case diagrams, such that after combining them, a comprehensive picture of the system can be observed, the purpose of creating use case diagrams is served. It is a common software engineering practice to always start off with describing use cases of the system, since they are the unit of the functionality of the system, as visible to the outside world.

Sequence diagrams [UML] are drawn to describe the behavior of the objects within the use case, in terms of the messages passed between them. A sequence diagram would give a more comprehensive view of the nature of the exact interactions, in terms of messages sent over a period of time. It is desirable to illustrate at least one sequence diagram per use case. More sequence diagrams can be drawn to refine the use case, in order to describe more than just the one scenario. The goal is to not go to the depth of the exact implementation details of how the components interact, but to document as much information as possible about the decisions, processes and computations involved.

These two notations, put together, create a concrete picture of the systems within the problem domain and the overall interaction pattern between the entities within the domain. This information can be used by component developers (who provide concrete implementation for abstract components in the feature model) to accurately implement the interaction, by bearing the entire system in mind, and still make local decisions about individual component interactions.

As far as this KB design is concerned, use case diagrams and sequence diagrams are used to model the problem domain and its functionality as visible to the outside world. About the connection with the models discussed in Section 3.1.1.3, the models are to
describe the overall family of DCS, i.e., the broad-level architecture and the decisions that would have to be taken to construct a particular instance. The diagrams, on the other hand, show the functionality of the DCS and the interactions between the entities that fall under the problem domain.

3.1.1.5 UMM-Annotated UDSL

As discussed earlier, the UniFrame Domain Specific Language can be used to represent the feature model and other information (constraints, use cases, architectures) in a textual format. The UDSL is basically a Domain Specific Language, i.e., a specialized, problem-oriented language which may be “a programming language or an executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain” [VAN00]. As said above, feature modeling and domain specific languages can be used in conjunction to adequately describe a system or a family of systems to any level of detail.

The UDSL modifies the basic DSL proposed in [DEU02], to model and document the domain knowledge (architectural) for a DCS. The work adopted as part of this thesis further modifies the UDSL by annotating it with a multi-level UMM Specification [NEI04, CAT05] in order to refine the notion of ‘Abstract Component’ as explained in the UDSL. This section first gives an example of the UMM specification (Table 3.4), to motivate the need for the annotated UMM-annotated UDSL (Appendix B), which is described following the UMM Specification.

3.1.1.5.1 UMM Specification

This section describes an example called ‘DatabasePro’ created to describe the UMM Specification. UMM Specifications of some of the banking domain components have been listed as part of the case study in Chapter 5.
**Table 3.4 Sample UMM Specification**

```xml
<?xml version="1.0" ?>
<UmmSpecification Name="" Version="Umm.1.0.0.x">
  <Specification>
    <AttributeList Presentation="concrete">
      <ComponentAttributes>
        Name="DatabasePro"
        Author=""
        Subcase=""
        Description="Implements Database functionality by providing wrappers for core database function calls in various languages."
      </ComponentAttributes>
      <ComputationAttributes>
        <Inherent
          Version="1.0.1"
          Author="DataPro Software"
          Id=""
          ExecutionEnvironment="Windows 95\2000\ME\2003\XP"
          SystemName=""
          DomainName="Database Management"
          Validity="500"
          ComponentModel=""
          Atomicity="false"
          DateDeployed="4-20-2004">
          <Registration Data="SQL Pro DB" />
          <RegistrationData="Access DB"/>
          <Registration Data=".NET DB Pro"/>
        </Inherent>
        <Functional>
          <FunctionsAndContracts>
            <Function
              Name="Global Component Function"
              SyntacticContract=""
              ConcurrencyContract=""
              Technology=""
              Description=""
              Precondition=""
              Postcondition=""
              Invariant=""/>
            <Function
              Name="ParseQuery"
              SyntacticContract="String ParseQuery(String, String, String)"
          </FunctionsAndContracts>
        </Functional>
    </ComputationAttributes>
  </Specification>
</UmmSpecification>
```
ConcurrencyContract="" 
Technology="" 
Description="This function intercepts a query in the given language, and converts it to the query format of the target database."
Precondition="QueryString != null"
Postcondition="ResultantString != null"
Invariant="" />
</FunctionsAndContracts>
</Algorithms />
</DesignPattern />
</KnownUsage />
</Alias Data="Database Manager" /> 
</Resources> 
 </Resource Architecture="CPU" Speed="233 Mhz" Load="" />
</Resources> 
</Functional>
</ComputationAttributes>
</CooperationAttributes>
</CollaboratorList>
</CooperationAttributes>
</AuxiliaryAttributes>
Mobile="false"
Mobility=""
FaultTolerance="This maintains an intermediate storage buffer where transfer data is stored. This facility provides fault tolerance to the degree of a single operation. If a particular operation fails it will be reexecuted when the system is restored.">
</SecurityList>
</SecurityList>
</AuxiliaryAttributes>
</ServiceAttributes>
ParallelismConstraint="synchronous"
OrderingConstraint=""
Table 3.4 shows an example of the database management component. The leaf-node level abstract components (Section 3.1.1.3.1) are described using the UMM Specification (as shown in Table 3.4) for components. The annotation is made in the UDSL proposed by [HUA03], and is described following the UMM Specification. The UMM Specification forms the ‘multilevel’ specification of the component in terms of syntax, semantics, synchronization and QoS. More discussion on multilevel specifications, and multilevel matching follows in Section 3.1.6.

An abstract component is described in terms of Component Attributes, Computation Attributes, Cooperation Attributes, Auxiliary Attributes and Service Attributes [NEI04]. The Component Attributes consist of:

- **Name:** Includes the name of the specification.
- **Author:** Includes the name of the author of the specification.
• Subcase: Indicates information about communication patterns of functions of components that reflect the synchronization aspect of functions.
• Description: Includes a brief description of the component.
• Change List: A log of the changes that have been made in the specification in terms of the editor, date/time of change and description of the change.

Component attributes are followed by the Computational attributes of the component, which include the inherent and functional attributes. The inherent attributes consist of:
• Version: Includes the version of the described component.
• Author: Includes the author (developer) of the described component.
• Id: An ID of the component.
• Execution Environment: Includes a detailed description of the execution environment of the component.
• System Name: Indicates the family of systems that this component belongs to.
• Domain Name: Indicates the domain name of the system.
• Validity: Indicates the time to live for this component.
• Component Model: Indicates the type of component model, that is: Domain, Wrapper, Representation, Classification, User Interaction or Headhunter.
• Atomicity: Indicates whether or not the component is atomic.
• Date Deployed: Indicates the date of deployment.
• Registration: A list of all the registries where this component is registered.

The functional attributes of the component primarily consist of the functions of the component and also the algorithms, design patterns, aliases, known usages and resources associated with the component. The functional contract includes the syntactic and semantic contract of each function, along with a description, concurrency and technology related information about the function. The functional contract spans over all the functions provided by the component. The syntactic contract is of the form:

\(<\text{Return Type}\> \ <\text{Function\_Name}\> \ ( \ <\text{Arg\_List}\> \ )\)
It basically indicates the prototype of the function signature. The semantic contract consists of function pre-conditions, post-conditions and invariants, all of these being logical assertions that dictate the semantics of the functions. The concurrency contract defines the function’s concurrency control. Design patterns indicate the design patterns employed by the component, and aliases are a list of any representative aliases for the component. Known usages indicate any known or proposed uses of the component.

The computational attributes are followed by the co-operational attributes of the described component, which basically include a list of collaborating components that can be categorized into ‘expected’, ‘required’ and ‘provided’. ‘Expected collaborators’ is a list of collaborating components that expects to use of the described component. Required Collaborators are the collaborating components that this component depends on, and Provided Collaborators is a list of components that depend on or use the described component.

The specification details some auxiliary attributes such as an indication of whether the component is mobile or not, with details of mobility information if mobile, and also the fault tolerance scale and information about the component. There is also a security list which shows the security functions used by the component and the level of security the component exhibits.

The specification concludes with service attributes that include parallelism constraints (synchronous or asynchronous), ordering constraints, execution rate, and a list of resources (type, value pairs) and finally a list of Quality of Service attributes and their values that describe the QoS profile of the component. These attributes may be traced down to the QoS catalog of service attributes, as will be discussed in Section 3.1.2.
3.1.1.5.2 UMM-annotated UDSL

The UDSL (Refer to Appendix B) is made up of the following constituents – Feature Expressions, Constraint Expressions, Design-Feature Expressions and Use Case expressions. Each one of these constituents is explained as follows.

There are five types of feature expressions listed as part of the grammar – Optional, Mandatory, Composite, Non-exclusive and Alternative. Optional features are expressed as ‘<Feature>?’, and may be included if the parent is included in the design of the concept instance. Mandatory features are expressed as ‘<Feature>’ or ‘<Feature>!’ and must be included if the parent is included in the design of the concept. Composite features are composed of a list of features, that may be mandatory, optional or a combination of both, and are expressed by ALL (<Feature_List>). Non-exclusive features are expressed as MORE-OF (<Optional_List>) and describe the ‘one or more’ aspect of feature selection. Alternative features are expressed as ONE-OF (<Optional_List>), and describe ‘at least one’ semantics of feature selection in a feature model.

The UDSL details four types of constraint expressions – Multiplicity Constraint, Default Constraint, Mapping Constraint and Satisfaction Constraint. The Multiplicity constraint is expressed by the keyword MULTIPLICITY in the form MULTIPLICITY(<feature1>, <feature2>: [cardinality]). ‘Default constraint’ is used to express certain default values by using the keyword DEFAULT in the form DEFAULT(<feature1>, <feature2>). This means, that the default feature for the first feature is the second feature. For example, DEFAULT(UserInterface, CounterRep) means that by default, the feature ‘UserInterface’ is realized by the feature ‘CounterRep’. The Mapping constraint is used for mapping a feature from one model to another model, using the keyword MAP, potentially at a different level of abstraction – mapping is thus a transformation, expressed as MAP(<feature1>, <feature2>). The first feature could be an instance at abstract component level and the second feature could be at function/interface level, with the first feature being mapped onto the second.
Satisfaction constraints are of five types [VAN02]: require, reject, include, exclude and mutual require. The Require constraint uses the keyword REQUIRE, expressed as \text{REQUIRE(<feature1>, <feature2>)} and means that if the feature1 is present, the feature2 must be present too. Reject constraint uses the work REJECT, expressed as \text{REJECT(<feature1>, <feature2>)} and means that if the first feature is present, then the second feature must not be present. The Mutual require constraint uses the keyword \text{MUTUAL\_REQUIRE} and means that if any feature in the feature list is present, then all should be present. The Include constraint uses the keyword INCLUDE and means that all the features present in the feature list must be a part of the generated system. The Exclude constraint uses the keyword EXCLUDE to indicate that all the features included in the feature list must not be a part of the generated system.

The next part of the grammar shows design feature expressions that are used to capture the hierarchical system architecture of the system. There are three types of elements: System, Subsystem and Abstract Component. The system is the root of the tree, abstract components are the leaves, and all the remaining nodes fall under ‘subsystem’. The design feature expression shows the interaction between design features and also the interfaces of the design features, using the keywords INTERACT and INTERFACE (along with \text{PROVIDED\_INTERFACE} and \text{REQUIRED\_INTERFACE}). INTERACT means that the first design feature interacts with the second design feature. The first design feature initiates the interaction, and the second design feature is the responder. INTERFACE on the other hand, shows the provided and required interfaces of the design feature, by using the keywords \text{PROVIDED\_INTERFACE} and \text{REQUIRED\_INTERFACE}, respectively. What follows these keywords is a list of interfaces that are provided and required by the design feature.

As part of this thesis, the notion of ‘Abstract Component’ (which is one of the elements in the architecture at the leaf node level of the feature diagram as described in Section 3.1.1.3.1) has been refined further, to describe properties of the abstract components. The grammar for the annotation is explained above (Section 3.1.1.5.1).
This concludes the ‘Design Feature Expression’ part of the UDSL describing the design features that can be SYSTEM, SUBSYSTEM or <Abstract_Component>.

The grammar for UDSL concludes with Use Case Expressions, which in essence, capture the realization of the use case, as viewed in a sequence diagram. The expression can be for a use case, first indicating the components involved in the use case by providing an abstract component list (Abstract Component Level), and then describing the actual activity using a set of function calls (Function/Interface level). The keywords used are PATH_C (<Abstract_Component_List>) which denotes the list of components that describe the use case. Ordering can be imposed between component interactions by enclosing a subset of abstract components in angular brackets (‘<…>’). At the function/interface level, the keyword used is PATH_F (<Function_Call_List>), and it means that the use-case is realized by executing the function calls listed in the Function_Call_List. Each function call in turn, is described by the abstract component that invokes the function, the name of the method and optionally, a communication pattern (described by keywords CP1, CP2S and CP2A, which represent One-way, Two-way synchronous and Two-way synchronous respectively). Only these three modes of communications are considered as part of the UDSL.

The UDSL (annotated with UMM Specification as <Abstract_Component>) is thus a comprehensive tool for describing the architecture of the system, at various levels of abstraction, i.e., System, Subsystem and Abstract Component. The annotation of the UDSL to describe the contract of the abstract components is one of the ways in which [VAR02] has been improvised. Associating the details of the abstract component is used by other entities in the KB (for glue generation, etc). It embodies the features of the system and their interactions in a manner that is understandable to the system designer as well as the component developer, since it details the architecture of the system at different levels of abstraction. The UDSL should be created for every domain, and along with its visual counterpart (feature model), it can provide a comprehensive and potentially, a fine-grained view of the problem domain and its components.
3.1.6 QoS Associations

The UA aims at constructing a QoS-aware system, which justifies the presence of a catalog of QoS attributes, since the catalog contains rules and relevant information about QoS attributes that can be used to evaluate the QoS attributes and determine, even before constructing the system, whether the performance of the system with respect to meeting QoS requirements will be satisfactory or not.

Another issue to be considered is that the QoS catalog contains a listing of possible QoS attributes, but not all may apply in the context of a particular domain. So, it is necessary to associate the QoS attributes with a particular domain, and that will guide the selection of QoS-aware components and a prediction about the overall system QoS properties based on the individual ones. Once the requirement is statically met, the components can be made to interact with one another, and then the dynamic behavior of the interactions can be measured, to finally determine whether the system fully meets the QoS requirements or not. The functional requirements are met by selecting the appropriate feature diagram, and components with the required semantics, but non-functional requirements, being an equally important concern, have to be satisfied. This leads to the static QoS property prediction and dynamic QoS property measurement. These are the two techniques that are used to determine whether the non-functional requirements are being met, the rules of which are a part of the the ‘QoS’ entity in the architecture of the KB (Section 3.1.2.4). The QoS associations are the QoS attributes that are of relevance to the problem domain. Thus, for handling the QoS predictions, the ‘domain-specific’ aspect is the association between the problem domain and QoS attributes. This is a list of QoS attribute names, and is described with Table 3.5. The domain-independent component of handing QoS predictions for static and dynamic QoS properties is dealt with in the following sections.

Table 3.5 Structure of QoS Associations

<table>
<thead>
<tr>
<th>Domain Name, &lt;QoS-attribute-list&gt;</th>
</tr>
</thead>
</table>
3.1.1.7 Handling Heterogeneity

A DCS is made up of heterogeneous software components that are accessible through various registries over the network. When the component specifications are inspected by the matching algorithm of the discovery service (Section 2.1.2, URDS), their multi-level contracts are compared against the query specification of the system to determine if there is a match, at least at the syntactic and semantic levels, and possibly even at the synchronization and QoS level. The technological details of the components are not considered. What the discovery service returns after receiving a query from the System Integrator is a set of components that may be heterogeneous with respect to programming language, platform, component model and even semantics.

The approach taken as part of this thesis generalizes the approach adopted by [TUM04]. [TUM04], which uses a template-based approach to generate ‘glue’ for components to communicate with one another. For every pair of heterogenous components that need to interact as per the underlying architecture of the DCS, glue is generated based on the technologies of the individual components, such that the initiating component of the interaction can communicate with the glue component, and the glue component can communicate with the responding component (the component that responds to the initiator’s query). [TUM04] considers Java-CORBA interoperability as the case study, and the discussion revolves around the glue component generation/deployment for Java and CORBA components to interact with another. As part of this thesis, however, a more generic framework is proposed, which can then be mapped on to specific technologies during system construction. The specific example is explained in Chapter 5.

The issue of handling heterogeneity surfaces when a component subset (returned by the discovery service) has components with different technologies, for e.g., Java, CORBA, .NET etc. After the component developer provides concrete implementations for UMM Specifications of the components (Section 3.1.1.5.1), the components are deployed and advertised on separate registries (based on the component’s technology).
Given the component interactions (from UMM-annotated UDSL), the decision is made by the glue generation framework as to whether the components are technologically disparate, and if they are, then the next step is to actually proceed to the glue generation. The generation of glue is based on a template; the language of this template is based on the technologies of the interacting components.

The UMM Specifications of components (C1, C2, …Cn) are recorded on registries (R1, R2,…Rm). Each registry corresponds to a specific technology, for example, Java components are on a Java RMI Registry, CORBA components are on the Naming Service and .NET Web Services are deployed on UDDI Registry. Given a component interaction (<C_init> and <C_resp>, where <C_init> is the initiating component and <C_resp> is the responding component), the glue generation framework first examines the corresponding technology information associated with components (detailed in Section 3.1.4), to determine if the glue needs to be generated. If the components are heterogenous, then the glue code needs to be generated for the component pair to communicate.

The framework retrieves other component specific details, such as the required interface of <C_init> and the provided interface of <C_resp>, and creates the glue code in a specific language (e.g., Java) using the standard template (for Java). The template code contains placeholders for methods (based on the interface) and also to bind the glue code to the <Glue_Registry>. The glue generator replaces the tags in the template with actual component specific details (from the UMM Specification) such as interface names, actual method names (arguments and return types). The glue code generated is deployed and recorded on the <Glue_Registry> with the name Glue_Code_<C_init>_<C_resp>. The glue configuration process, then, configures C_init to communicate with the Glue_Code_<C_init>_<C_resp>, and Glue_Code_<C_init>_<C_resp> to communicate with <C_resp>. This is done by performing

<Glue_Registry>.Lookup (Glue_Code_<C_init>_<C_resp>)
in order to obtain the handle to the glue. The configuration process also configures \textit{Glue\_Code\_<C\_init>\_<C\_resp>\_<C\_resp>} to communicate with \textit{<C\_resp>} in order to actually invoke \textit{<C\_resp>\_Interface}.

Glue Code Templates and Glue Code Configuration Templates for component-pairs are stored in the KB as text files, and these are mapped to specific programming languages based on the application requirements and deployed on \textit{<Glue\_Registry>}’s accordingly. For example, if glue code has to be written in Java, it is deployed on a Java RMI Registry [SDN], and the RMI Registry API’s are used to realize the actual procedure calls.

3.1.2 QoS

Since the goal of the UniFrame Approach is to construct a QoS-aware distributed computing system that is predictable and reliable, documenting the non-functional aspects of computing systems have to be given the appropriate importance. As discussed in Chapter 2, the QoS catalog developed as part of [BRA02], is incorporated in the KB as a reference for QoS attributes and their properties. The QoS catalog is a general catalog that stores information about the QoS attributes that are generally of relevance in the context of distributed systems. This section describes in detail the QoS catalog that is stored in the KB, and provides information on how this catalog is used for system generation.

3.1.2.1 Attribute Details and Metrics

The QoS catalog enlists several significant properties of the common QoS attributes. For each attribute, the most important properties that need attention are: 1) overall description, intent and metrics, 2) Evaluation procedure and formulae, 3) composition and decomposition rules, and 4) prediction and instrumentation techniques. Below is the list of the attributes with their general description. For details of evaluation process and composition-decomposition formulae, refer Appendix A.
1. Dependability
2. Security
3. Adaptability
4. Maintainability
5. Portability
6. Parallelism constraints
7. Ordering constraints
8. Priority
9. Throughput
10. Capacity
11. Turn around Time
12. Availability

3.1.2.2 Evaluation Process (Component-Level)

The QoS attributes listed above, also have an Evaluation Procedure and Evaluation Formula associated with them. Each attribute is associated with:

- A detailed evaluation procedure in terms of the step by step instructions to compute the various measurement indices as mentioned above.
- A formula, that uses metrics associated with the code of the component, to compute the measure. The formula can be described in the form of the following grammar:

Table 3.6 QoS Evaluation Formula

<table>
<thead>
<tr>
<th>S</th>
<th>::= &lt;Formulae&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Formulae&gt;</td>
<td>::= &lt;Formula&gt;&lt;Formulae&gt;</td>
</tr>
<tr>
<td>&lt;Formula&gt;</td>
<td>::= &lt;trig&gt; ( &lt;sign&gt;&lt;term&gt; ) &lt;op&gt; &lt;formula&gt;</td>
</tr>
</tbody>
</table>
The result of the evaluation process is a value, of a certain type (e.g., Boolean/Float/Integer) and range (e.g., Float between 0 and 1) that can be used for further computation. For example, the evaluation formula for throughput is:

Throughput = NR / NT

Where,
NR: Number of responses
NT: Total time taken

The composition rules are used (as described in the next section) to compose the QoS attributes of a set of components, and the measures computed by above procedure can be used for the same. Refer to the Appendix A for a complete list of evaluation procedures and corresponding formulae for the attributes listed above.

3.1.2.3 Composition and Decomposition (System-of-Components level)

As mentioned above, the QoS attributes of individual components are combined to compute a single, comprehensive value, which denotes the value of the attribute for the entire system made of the selected component subset. The composition and decomposition rules listed as part of the catalog are used to compute a system-wide value from values of components.

The composition rules are of a higher significance in the context of assembly, and decomposition rules come into picture while searching for relevant components. The standard structure for composition rules is:

| S | ::= | <Formula> |
| <Formula> | ::= | ATTR = <func> <attribute_set> <n> |
| <func> | ::= | MIN|MAX|SUM|MEAN|AVERAGE|WT_SUM|
For example, $S = \min (C_i)$, where $C_i$ is the value of the attributes for the i’th component for the particular attribute, and $S$ is the value of the attribute for the entire system. This rule means that the value of the attribute for the system is determined by the least value of the attribute for the components selected. This applies to attributes such as dependability, security, adaptability, throughput (in a sequential system), capacity and availability. For example, the system is as secure as the least secure component in the set.

Another example of a different composition rule is:

$$S = C_1 \& C_2 \& C_3 \ldots C_N$$

Where $C_1, C_2, C_3 \ldots CN$ are the values of the attributes for Components $1,2,3\ldots N$ and $S$ is the value of the attribute for the entire system. Obviously enough, this rule applies to attributes such as Parallelism Constraint, Ordering Constraint and Priority. For example, the system will satisfy the parallelism constraint only if all the participating components satisfy the constraint too.

The standard structure for decomposition rules, on the other hand, can be described as:

Table 3.8 QoS Decomposition Formula

<table>
<thead>
<tr>
<th>S</th>
<th>::= &lt;Formula&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Formula&gt;</td>
<td>::= ATTR &lt;n&gt; = &lt;func&gt; ATTR</td>
</tr>
<tr>
<td>&lt;n&gt;</td>
<td>::= 1|2|3...*</td>
</tr>
<tr>
<td>&lt;func&gt;</td>
<td>::= MIN|MAX|SUM|MEAN|AVERAGE|WT_SUM | SIN|COS|TAN|COT|SEC|COS</td>
</tr>
<tr>
<td>&lt;op&gt;</td>
<td>::= &lt;=</td>
</tr>
</tbody>
</table>

The corresponding decomposition rule for the example discussed above, would involve the converse of the composition rule, i.e.,

$$C_i = S$$

Where
Ci is the value of the attributes for the i’th component for the particular attribute, and S is the value of the attribute for the entire system. While decomposing the non-functional requirement, if the desired system needs to satisfy the parallelism constraint, then all the components that need to be searched for should satisfy that constraint as well, otherwise the requirement will not be met.

Some of the composition and decomposition rules are listed above, with examples of the attributes they relate to, just to provide an idea of how these rules are used in the process of generation of the system. The complete listing is available in Appendix A.

3.1.2.4 Property Prediction and Instrumentation

As discussed before in Section 3.1.1.6, the domain-independent component of QoS measurement (static as well as dynamic properties) is detailed as part of the ‘QoS’ entity. For both types of QoS properties, the measurement information has to be present in order to:

1. Statically compose properties of relevant QoS attributes, to determine if the QoS requirement will be met if the selected component set is finalized for system generation.
2. Dynamically measure certain other QoS attributes, by fusing certain instrumentation code that measures some of the QoS attributes, and use the information generated by running the instrumentation code, to make a prediction of the dynamic properties of the system.

Bearing these two requirements in mind, the KB must contain QoS associations that relate the QoS attributes that are relevant to a certain domain, with the domain information. As discussed above, after a set of components is selected, the static properties (which do not depend on any runtime information) can be composed using the composition formula’ in the catalog (Section 3.1.2.3), and based on the value obtained, a decision can be made as to whether the component set is worth pursuing, in order to follow the remaining process of system generation (i.e., bridging the heterogeneity,
which is discussed in the next section, and then making the dynamic prediction). If the component set fails the static check, the set can be disbanded, and a new set of components can be searched for by either:

1. Retrying the same query later, or
2. Retrying with a modified subset of the query, or
3. Selecting a different architecture to start with, by choosing a different feature diagram.

The instrumentation of code while constructing a DCS will enable the measurement of certain dynamic QoS attributes. This can be achieved through the principles of Aspect-Oriented Programming (AOP) [AOP00]. Aspect-Oriented Programming and Software Development are based on the separation of crosscutting concerns, i.e., concerns that crosscut through all the modules and layers of the system. AOP employs an advanced modularization strategy by encapsulating the concern in one place, by the use of certain expressions called aspects. Aspects can contain advice which is just code that can be joined to certain points in the body of a program. The mechanism to identify where and how the advice should join the code and run, is also specified through point-cuts and join points. Join points are points in the program where additional code can be joined, and hence they need to be easily addressable. Join points are the points where the advice can be made to run either ‘before’, ‘after’ or ‘around’ the join point. A Point-cut is a way to determine if a given join point matches the advice to be run.

According to [CAO03], aspects can be used to join instrumentation code to component code and specific points, and then run the additional code to log runtime information about the components. The instrumentation code can also be joined to the glue code that is used to bridge heterogeneity. Event traces that are generated by running the instrumentation code can be logged, and then analyzed in order to make a prediction about the dynamic QoS properties of the system.
The domain expert needs to indicate:

- *Advice* corresponding to each attribute, which in this context, would mean the instrumentation code that measures that particular QoS attribute. The advices will be stored as a QoS-Advice-Code library as part of the KB in the ‘QoS’ information. A sample ‘Advice Code’ for the QoS parameter ‘Turn around time’ has been provided as part of the Banking Domain case study in Chapter 5.

- Appropriate join points in the component code, or glue/wrapper, or both, where this advice can be run and logged. The point-cuts have to be provided in AspectJ style [AOP00], and have been elaborated as part of the banking domain example.

### 3.1.3 Components

This part of the KB is for the benefit of the component developer in providing:

- Refinements in the abstract component specifications from the UDSL or Feature Diagram.
- Concrete implementations of components developed as per the specification in the UDSL and Feature Diagram.
- Testing interfaces for the components.

This section discusses the above issues in detail over the following sections.

#### 3.1.3.1 Specifications and Implementation of Components

Component developers need to provide implementations for abstract component specifications detailed in the feature diagram and/or UDSL. The KB stores abstract component specifications of all components developed by the component developers.

While providing an implementation for a particular component, there are many issues that the developer has to be concerned about, such as maintaining the exact structure of the specification while writing code, and if any specification details are
modified (only minor modifications are allowed that do not affect the remaining features in the system in a very significant way), then the specification has to be updated accordingly. If the developer wishes to suggest any updates to the domain expert to incorporate newer features, and hence, modify the feature diagram, the developer can suggest the change offline (by sending an updated UDSL by including the UMM Specification of the new abstract component), after which the domain expert can decide whether the change is appropriate, based on the context of the domain. The communication between the developer and domain expert is through modification of the UDSL and a change-record set. The change-record describes the update requested and is of the form:

Table 3.9 Change-record Set

| <Change-Record-Set> ::= <Change-Record> <Change-Record-Set> | <Change-Record> |
| <Change-Record> ::= ADD-FEATURE <Feature-Description> | ADD-INTERFACE <Interface-Description> | ADD-DESIGN-FEATURE <DesignFeature-Description> |
| <Feature-Description> ::= FEATURE <Name> <Type> |
| <Type> ::= MANDATORY | OPTIONAL | COMPOSITE <Feature_List> |
| <Name> ::= TEXT |
| <Interface-Description> ::= INTERFACE <Name> |
| <DesignFeature-Description> ::= SUBSYSTEM | <Abstract_Component> |

By reviewing the UDSL and the Change-Record-Set for new additions, the domain expert can decide what additions need to be pursued and implemented as part of the domain information. The model proposed here for updating the domain information handles only new additions – only optional and composite features can be added, since adding mandatory features would affect the current instances of DCS family of the problem domain. The domain expert can create a new feature diagram altogether, if any mandatory features are suggested for addition.
Also, the advices have to be joined to the code at appropriate join points, for instrumentation of measurement code, which is required for validation of dynamic properties of systems. If the developer handcrafts the code for the component, abiding by the constraints laid out in the feature model, he/she still has to ensure that the code for advices fits in at the join points specified by the domain expert as the part of the ‘QoS’ (Section 3.1.2.4).

The KB stores a mechanism to automatically generate skeletal code that deals with the above issues by default, in accordance with the constraints in the feature model. This mechanism will create the appropriate placeholders for the developer to key in the business logic, and other communication, validation and prediction issues will be taken care of by the skeletal code generator. The next chapter describes some of the concrete implementations that have been carried out as part of the UniFrame Project, and it discusses the realization of this mechanism in more detail [CHE06].

3.1.3.2 Unit Test Cases

There is no significant effort made in the area of ‘testing’ in the context of generative programming and the whole notion of families of systems [HUA03]. Testing is an important part of software development, and in conjunction with the more theoretical and scientific validation and prediction techniques, will add to the quality and robustness of the overall system.

The KB should contain a mechanism to automatically generate unit test cases from the specification of the component in order to check for boundary conditions, violations of the semantic contracts (method pre-conditions, post-conditions and invariants), and so on and so forth. This mechanism can be used to create unit test cases, which can later be executed by testing experts, before the component is deployed. The developer also has an option of providing a test case, using a testing script or a similar mechanism, which is a standardized testing approach, and these test cases can be run on
the components as well. In this thesis, the following format [TES] (Table 3.9) for test-cases has been used as a starting point. Test cases using this format have been described as part of the banking domain case study in Chapter 5.

Table 3.10 Unit Test Case Format

<table>
<thead>
<tr>
<th>Information</th>
<th>Test Case ID</th>
<th>Test Creator</th>
<th>Version</th>
<th>Name</th>
<th>Intent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>Testing Environment</td>
<td>Programming Environment</td>
<td>Database System</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initialization of Parameters</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Termination Rules</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action List</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conclusion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expected Status:</td>
<td>PASS/FAIL</td>
<td>Rules</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

There are other prevalent approaches, such as the Test Specification Language (TSL) as described in [BAL00], where a test specification is written in the TSL and then compiled into an executable test script. For this KB, the test case format provided above is used. More automation in running test scripts can be added, but this format covers the basic information needed in a test case.

In essence, the KB must contain test cases for every component that is developed, so that the component can be tested by a testing expert, and log failures, if any, so the creator of the test case can be informed and the test case be fixed.
3.1.4 Technology

An important aspect of the KB is the technology that is used to deploy and register components, make them available for discovery and usage. Important information on the technological front is primarily the deployment mechanism for developed and tested components and information about the registry and the runtime environment of the components. The KB must provide a mechanism for the component developers and/or testers to specify some deployment information of the component as described below:

Table 3.11 Technology

<table>
<thead>
<tr>
<th>Name and Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Domain</td>
</tr>
<tr>
<td>Description</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Component Technology:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming Language</td>
</tr>
<tr>
<td>Data Source Used</td>
</tr>
<tr>
<td>Component Model</td>
</tr>
<tr>
<td>OS Requirements</td>
</tr>
<tr>
<td>Virtual Machine Requirements</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deployment:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification URL</td>
</tr>
<tr>
<td>Code-behind URL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Registry:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
</tr>
<tr>
<td>Access Control</td>
</tr>
<tr>
<td>Passphrase</td>
</tr>
</tbody>
</table>

- Name and Description of the component and its Identifier over the network
- The component’s development technology in terms of
  - Programming Language (Java, C#, C++, etc.)
  - Data Sources Used (Database connections, their types, e.g., Oracle, SQL, MySQL)
  - Component Model (.NET or J2EE, etc.)
  - Operating System requirements (Windows XP Pro, or Sun Solaris, etc.)
  - Virtual Machine requirements (JVM for J2EE applications, etc.)
• The component’s deployment information in terms of:
  o The exact location of the specification (XML/Plain Text etc.)
  o The exact location of the code behind files (FTP address or a link to a stub for the component, or a web service interface)

The KB must also maintain information about the registries which actually hold the component specifications. The information stored is:

• Exact address of the registry. If the registry is distributed over many locations, then addresses of all locations must be stored.

• If the registry has any access control mechanism, then the appropriate information to log on to the registry (for the discovery service) and the interface to do so, must be made available.

This is the information that has been identified to be recorded in the KB as far as the technology is concerned.

3.1.5 Query Decomposition

This section discusses the issue of query decomposition. Given a query, the first step is to convert that to a form that is machine-process-able, i.e., resolve the query into precise fields (considering that a natural language parser is used to process the query initially). The query has to be split into functional requirements (pertaining to the functionality of the desired system in terms of its functional behavior) and non-functional requirements (pertaining to the behavior of the system’s performance, quality and reliability, i.e., dealing with the QoS Framework). Both are discussed below.

3.1.5.1 Functional Decomposition

After the client inputs a query stating the requirements of the desired system, the System Integrator has to review some feature diagrams in the KB of related domains, to
select an architecture blueprint. From the blueprint, the System Integrator creates an ‘instance’ of the system, by making a selection from the variable features (the mandatory features must be present in any application within the domain, as per definition of the feature model). The decision model described in Section 3.1.1.3 is used here by the System Integrator to decide what features need to be incorporated. Once the instance is decided upon, the query is modified and decomposed to search for the abstract components that appear at the leaf node level – this would be a series of queries, each searching for a particular feature (abstract component). The algorithm used to match the components as per the criteria is discussed in Section 3.1.6. This is how the functional aspect of query decomposition is handled.

3.1.5.2 Non-functional Decomposition

From the precise query statement (parsed) input by the client to the System Integrator, the System Integrator consults the QoS catalog to look for decomposition rules (discussed in Section 3.1.2.3) for the attributes stated in the query statement, and based on the attributes listed as ‘relevant’ under QoS Associations (Section 3.1.1.6), the decomposition rules for the subset of relevant QoS attributes are applied. Following this individual attribute values are associated with the feature sub-queries (resulting from functional decomposition), and the queries are ready to be given to the discovery service, i.e., the multilevel matching algorithm.

The query decomposition rules are just derived from information present in the KB (part of the QoS catalog). This is part of the starting phase for system construction, and hence for simplicity, the rules have been shown as consolidated into the ‘Query Decomposition’ piece in the KB.
3.1.6 Multilevel Matching

The multilevel matching algorithm is used by the discovery service to match the available component specifications against the query specification (provided by the System Integrator after functional and non-functional query decomposition). This has been discussed before as part of Section 2.1.2. The multilevel matching algorithm has been experimented with respect to the time taken to compute the match against the quality of the results returned (precision and recall) [KAT06] and is proven to be effective in the context of DCS and distributed component discovery. The matching can be summarized as below:

Table 3.12 Syntax for Multilevel Matching Algorithm

<table>
<thead>
<tr>
<th>Result =</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match_Syntax * C1 +</td>
</tr>
<tr>
<td>Match_Semantic * C2 +</td>
</tr>
<tr>
<td>Match_Synch * C3 +</td>
</tr>
<tr>
<td>Match_QoS * C4</td>
</tr>
</tbody>
</table>

| Match_Syntax = |
| Query_Argument_List <op1> Candidates_Argument_List |
| <op1> = VERBATIM | INHERITENCE | COERCION |

| Match_Semantic = |
| PRE_Q <op2> PRE_C <op3> |
| POST_Q <op2> POST_C <op3> |
| INV_Q <op2> INV_C |
| <op2> = EQUIVALENT | IMPLICATION |
| <op3> = AND | OR | NOT | EXOR |

| Match_Synch = |
| SYNCH_POLICY_CAND <op4> SYNCH_POLICY_QUERY <op3> |
| SYNCH_IMPL_CAND <op4> SYNCH_IMPL_QUERY |
| <op4> = EQUAL | IMPLIED |

| Match_QoS = |
| ATTR_VALUE_CANDIDATE <op5> ATTR_VALUE_QUERY |
| <op5> = <|<=|>|=|=|!=|<<|>> |

C1, C2, C3, C4 = [-] FLOAT
As shown in Table 3.11, the multilevel matching algorithm computes the match over 4 levels – syntax (argument list, return type), semantics (preconditions, postconditions and invariants), synchronization (policy and implementation) and QoS (attribute-value pairs). Match at each level is computed as an ‘extent of match’, i.e., the higher the extent of match (VERBATIM is higher than INHERITENCE, also, EQUIVALENT is higher than IMPLICATION), the higher the individual level output. Finally, outputs at all levels are combined using weights (C1, C2, C3 and C4) to compute the final match. The algorithm has been implemented as part of [KAT06] and the scheme for composing the results has also been experimented with. Analysis of the algorithm can be found in [KAT06].

3.2 Process of Constructing the Distributed Computing System

This section of the chapter summarizes the actual process of constructing the DCS from the phase of obtaining and parsing client input all the way up to deploying the system. This section first describes the workflow as detailed in Figure 3.2, which presents all the important phases and actors that are involved in making decisions in case of conflict or lack of output. The flow is described step by step, and the entities are introduced as and when they appear in the workflow.

Most importantly, the role of the KB is indicated as part of the workflow by showing the information it supplies to the workflow at every stage and correlating it with the discussion so far (Section 3.1).
The process starts with receipt of a query from the client. The System Integrator intercepts the query and reviews some of the feature models present in the ‘Architecture’ part of the KB, and selects the one which suits the requirements most appropriately. As explained in the previous sections of this chapter, the feature model documents common and variable features of the families of systems in the given domain. So the System Integrator selects an instance from the feature model by making a choice about some of the optional features and finalizes on the instance, i.e., architecture blueprint.

A component list can be derived from this blueprint, and the query entered by the client can be functionally decomposed to express the requirement for the abstract components instead. Simultaneously, the System Integrator has to consult with the QoS Catalog to apply decomposition rules to the QoS attributes entered as part of the query.
Rules are applied to those attributes that have been associated as ‘relevant’ for the given domain. Once the decomposition rules are applied and QoS requirements for the individual components are obtained, they can be appended to the functional sub-queries and submitted to the discovery service.

The discovery service processes each query by further decomposing it into a form that is as close to a multilevel specification as possible (Syntax, Semantics, Synchronization and QoS). The discovery service performs multi-level matching for each multilevel query against multilevel specifications of the components listed under the domain in concern. If the feature of profiling and rank ordering is available as part of the discovery service, the algorithm is applied during the matching process and the most relevant results are presented to the System Integrator. The System Integrator can review the results, and choose the components that he/she wants to pursue with.

After a component set is selected, the static QoS attributes of the individual components are composed using the composition rules stated in the QoS Catalog. The values obtained after composition are compared against the requirement stated by the client, and if the requirements are met, then the System Integrator proceeds to the next step. If not, there are many choices available. The System Integrator can retry the query, after some time, or retry the query by modifying part or all of it, or go back and select a different architecture altogether, and repeat the process carried out so far.

Once a set of components statically meeting the QoS requirements is obtained, the System Integrator has to find the appropriate wrappers that will assist in bridging the heterogeneity among the selected components. The wrappers can either be searched for (with the help of the discovery service) or be generated, since the KB stores the information needed to generate wrappers for the particular domain. Adaptor components are readily available for most technology-combinations, and the wrappers have to be selected so that the components can be glued together to generate the system.
Once the system is generated, the code instrumented to log the dynamic behavior of components with respect to the relevant, dynamic QoS attributes, has to be run to measure the values of the attributes. This will provide another estimate of the dynamic behavior as opposed to the static prediction made before. If the set of components fails to meet the requirements following this validation check, the they have to be disbanded and the System Integrator is again presented with a choice of retrying after some time, or selecting a different combination of components, or selecting a different architecture altogether. After a valid set of components with the appropriate wrappers is obtained, the technological analysis of the system thus generated is presented to the System Integrator along with a system that is ready to be deployed. This system meets the functional as well as non-functional requirements as stated in the query, thus completing the process of construction of a QoS-aware distributed computing system.

3.3 Conclusion

This chapter provides the architecture of the KB, and presents a workflow that can be used to build a QoS-aware DCS by using the contents of the KB. the architecture of the KB has been described in terms of the pieces and their interrelationships. The structure of each piece is formally represented as part of this chapter. The next chapter discusses the connection of some software applications that have been developed as part of the UniFrame activity, with respect to the KB and workflow described above.
CHAPTER 4. A PROTOTYPICAL REALIZATION OF THE UNIFRAME APPROACH

The previous chapter detailed the structure of the KB by describing the entities present in the KB and the role of each entity in the process of constructing a QoS-aware DCS. As part of the UA, many prototypical systems have been designed and successfully implemented on various platforms, to provide a concrete realization of some of the concepts that have evolved over the course of the project. Almost every piece of the framework has been developed, starting from specification editors for developing component specifications in UMM form (as discussed in Chapter 3), to discovery service and system generation prototypes. This chapter is dedicated to describing some of these concrete realizations that are relevant to the KB in terms of how the KB would interface with each one of these prototypes.

Figure 4.1 describes the complete UA with the infrastructure that comprises the tools that have been created as a part of the project. In the figure, the square boxes represent the KB entities that partially overlap with the tools developed, e.g., Skeletal Code Generator [CHE06], UMM Specification Editor [NEI04]. The circles represent the processes followed as part of the UniFrame Approach, e.g., Discovery Service (UniFrame Resource Discovery Service) [SIR02] and System Generation [HUA03]. The remainder of this chapter describes these implementations briefly.
4.1 Architecture

As discussed in Chapters 2 and 3, [VAR02] and [HUA03] have been the major attempts in modeling the domain information for the development of distributed computing system. [VAR02] primarily focused on capturing all the relevant domain information by integrating several domain engineering and modeling techniques, and incorporating UML notations (Use Case Diagrams, Sequence Diagrams and Collaboration Diagrams) as a part of the domain description. The work presented in [VAR02] is more oriented toward taking into consideration the development of distributed systems made of heterogeneous components than some of the other approaches (discussed in Section 2.2.3.1), but the UniFrame Generative Domain Model (UGDM) [HUA03] developed as a part of the UniFrame System Level Generative
Programming Framework (USGPF) (Section 2.4) was the major effort in developing a QoS aware distributed system.

The KB proposed as part of this thesis in Chapter 3 (Section 3.1.1) builds upon [VAR02] as far as domain modeling and engineering is concerned and uses the UDSL described in [HUA03] to model the features, constraints, design features and use cases. The approach detailed in [VAR02] is used as a starting point, since it is more oriented to constructing DCS than other domain modeling approaches (Section 2.2.3.1). Appending the UMM Specification refines the granularity of the UDSL, and facilitates the domain description at any level desired by the domain expert (From system level to function level of individual components). The artifacts produced from the approach detailed in [VAR02] form a major portion of the data stored in the KB. Thus, these two efforts are inputs to the KB in a sense, and can be connected as sources to add domain related information to the KB.

4.2 Components

This section describes two tools that have been developed for the component developer to refine the design of and provide implementation for the specifications laid out in the feature model as part of the architecture information. Both of these are centered around the UMM Specification (elaborated as part of Section 3.1.1.5) and are discussed briefly as part of this section.

4.2.1 UMM Specification Editor

The UMM Specification Editor [NEI04] can be used by the component developer to input specifications. This editor generates the UMM Specification by converting values entered by the component designer/developer into the XML format. Since the XML Format of the specification is not easily understandable by humans, this editor hides the implementation details by adding user interface layers on top of the raw XML
specification. This software is essentially an editor that inputs textual information (multilevel specification of the component) from the component designer/developer using a tabbed interface and creates an XML Document from it that can be fed to any application (such as a Discovery Service for matching, or Testing Expert for automatic test case generation for the component). Figure 4.2 shows the interface developed by [NEI04].

As far as the interconnection with the KB is concerned, this application can be used by the Component Developer to use and refine the UMM Specification of Abstract Components (available through UMM-annotated UDSL), and convert it to a XML form so that it can be deployed to a registry and made available for discovery. The UMMSE is thus an user-interface for inputing component specifications in UMM form.
4.2.2 Translation of the UMM Specification to a Language Specific Implementation

(Skeletal Code Generator)

The Skeletal Code Generator developed in [CHE06] can be used by a component developer to use the UMM Specification (from UMM-Annotated UDSL <Abstract_Component> definition) in the XML form to generate skeletal code. The skeletal code generator creates Java code that corresponds to the Component, by providing the necessary placeholders, for the developer, to which method functionality is added and method pre-conditions, post-conditions and invariants for individual methods can be entered and verified by the generator for consistency with the specification.

After the developer views the skeletal code generated, he/she still has the option of making changes in the structure of the specification (modifying type information of function signatures, etc), and the changes made will be reflected as part of the specification. Thus, the developer can go back and forth between specification view and the code view, and the code generator ensures that the specification and the backend code remain in sync with one another. The details of the actual translation can be obtained in [CHE06]. The Developer can toggle between Graphical and XML Views of the Component Specification using the ‘Graphical to XML Converter/XML to Graphical Converter’. The Skeletal Code Generator is a switch between the XML Specification and the corresponding Java Code (XML to Java Converter/Java to XML Converter).

[CHE06] is essentially an enhancement of the work described in Section 4.2.1 [NEI04] since it appends the functionality of reflecting code changes in the specification, to the UMMSE [NEI04]. [NEI04] just created a XML Specification out of the more textual UMM Specification. [CHE06] can be used very conveniently as part of the KB’s mechanism of skeletal code generation from UMM Specifications. As discussed in Chapter 3, the ‘advices’ for measuring dynamic QoS properties can be embedded in the functionality of [CHE06]. This measurement is required in the dynamic validation of the system (composition of dynamic QoS properties) before the generated system can be delivered to the client. The Skeletal Code Generator simplifies the Component
Developer’s task by dealing with consistency issues (between code and specification), so that the Component Developer can focus on the business logic of the component.

**4.3 Quality of Service**

The QoS Catalog (Appendix A) is directly used in the KB and constitutes the complete QoS subsystem of the KB, since it contains a listing of QoS attributes that apply to DCS. Section 3.1.1.6 provides the mechanism to associate the QoS attributes that are more relevant to the domain, with the domain information. The Catalog details evaluation procedures and metrics for each attribute along with the composition and decomposition rules that are used at various points during the process of DCS construction (Section 3.2).

**4.4 Discovery Service**

The discovery service discovers the heterogeneous software components over a network of registries and selects the components that match best with the query specifications given by the System Integrator. A lot of work has been undertaken in the category of Discovery Services as part of the UniFrame Project. The basic discovery architecture, called the UniFrame Resource Discovery Service (URDS) [SIR02], is explained as part of this section. The work undertaken for discovery services centers around the architectural concepts described in [SIR02]. [KAT05] deals with multilevel matching of software component specifications and query specifications, and focuses on the matching process that can be used in any discovery service. [DEV05] provides some enhancements to the basic URDS given in [SIR02]. All these are described as part of this section.
4.4.1 An Architecture for Discovery of Heterogeneous Software Components (URDS)

[SIR02] presents an architecture and implementation of the UniFrame Resource Discovery Service (URDS) that facilitates dynamic discovery of heterogeneous software components which meet the necessary functional as well as non-functional requirements of the system. [SIR02] proposes a platform independent architectural model for the URDS. The detailed architecture and organization of the different modules can be obtained in [SIR02]. The basic outline of URDS can be summarized as follows: the URDS consists of headhunters, which have access to a number of active registries (who constantly collect component specifications). Given a query, a headhunter can communicate it to other headhunters and/or try to find a match for the query in its local repository.

For URDS to be interfaced with the KB, the query specifications should be passed on to the headhunters, who will spread the query to other headhunters as well, and try to locate the components that match the query to largest possible extent. The System Integrator will provide the URDS with a set of queries (for each component that is required in the architecture blueprint), and the URDS will present to the System Integrator the component set that matches the queries entered.

4.4.2 Multilevel Matching

[KAT06] focuses on the part wherein the headhunter (from URDS Architecture [SIR02]) tries to match a query specification as issued by the System Integrator against candidate component specifications present in its local repository. The ‘multilevel matching algorithm’ as developed by [KAT06] is an attempt to perform multilevel matching of component specifications at Syntax, Semantics, QoS and Synchronization Levels. [KAT06] describes the representation of the component specifications at each level, and also defines a matching algorithm at each level, since the notion of ‘match’ at each level is different. The matching algorithm at each level does not return a Yes/No match indicating how appropriate a certain candidate component is for that level; instead
it returns the ‘extent’ of match described using ‘exact match’ and ‘relaxed match’, which indicates how the component fares at each level. Once again, the definition of exact and relaxed matches differs from level to level. The meaning of ‘exact’ and ‘relaxed’ for each level is also defined as part of the multilevel matching algorithm.

The multilevel matching algorithm can be easily incorporated in the KB, under discovery services, since the algorithm uses the UMM Specification of components. The ‘extent of match’ feature can provide the System Integrator with specific information about each level, so that the decision can be made based on how important the match at a certain level is.

4.5 System Generation

4.5.1 Glue Generation Framework in UniFrame for the CORBA-Java/RMI Interoperability

[TUM04] focuses on the issue of heterogeneity which is inherent to distributed computing in general and provides a solution to tackle the challenges posed while integrating heterogeneous components. The challenges include those pertaining to heterogeneity of technology, component model and semantics. [TUM04] proposes a framework to generate Glue Code for interoperating between Java RMI and CORBA components using pre-defined code generation templates and also discusses the issues related to the placement of the glue code. As per [TUM04], the choice of location of the glue code greatly affects the performance of the generated system.

The template-based approach adopted by [TUM04] has been extended as part of this KB to provide a generic Glue Generation Framework. [TUM04] focusses on Java
RMI and CORBA interoperability, and generates glue code and glue configuration code based on glue code templates and glue configuration code templates, respectively. The framework proposed by [TUM04] first examines the technology information of the components in order to determine if glue is needed for the components to interoperate, i.e., whether or not they belong to disparate technological models. If glue generation is required, then the glue generation framework obtains the component’s information, i.e., the interface of the component, and creates a .java file from the glue code template by filling in the tags of the template with actual component specific information. Since the underlying idea is to make the initiating component communicate with the glue code, and the glue code with the responding component, the glue configuration generation process then configures the initiating component, glue code and responding component accordingly. Both glue code and glue configuration code are compiled and deployed, and the communication can then be carried out between the initiating and responding component.

4.5.2 System Generator as part of UniFrame System Generation Infrastructure (USGI) [HUA03]

The UniFrame System Generation Infrastructure has been explained as part of Chapter 2. USGI deals with construction of the DCS from the UniFrame Generative Domain Model (UGDM) generated during the UniFrame UGDM Generation Process (UDGP). The System Generator generates the system given the components that need to be assembled. It validates the system (statically and dynamically validates QoS requirements) and assembles the system to be deployed.

Section 3.2 described the process of constructing a QoS-aware DCS, and it highlighted how the various KB entities collaborate to form a DCS from components and the appropriate bridges as directed by the System Integrator. USGI focusses on constructing the system out of components, by elaborating on how the different entities such as such as UGDMKB Generator, UGDMKB, Order Processor, UGDMKB Builder,
System Generator, Application Programmer, Wrapper and Glue Generator, URDS collaborate. USGI documents the process of system construction in a formal manner as described below in Figure 4.3. This thesis focusses on the structure of the KB, more than the actual process carried out for system construction. The process of constructing DCS from the KB proposed as part of this thesis is elaborated in Section 3.2.

Figure 4.3: USGI Workflow (from [HUA03])
The approach taken in USGI [HUA03] starts with the System Integrator intercepting a specification of the desired DCS. The process of system construction starts with discovering a set of components that match the specification of the desired DCS. Once the set is discovered, glue and wrapper components are located for the components in order to handle heterogeneity. The resultant system (after incorporating the glue-wrappers) is then validated for QoS constraints and deployed if the constraints are satisfied. If not, the system is disbanded and the process is repeated to look for another set of components.

The approach taken as part of this thesis is more comprehensive than USGI in the sense that it involves the process of creating the entire KB in a formal manner, followed by using all the pieces of the KB in order to generate the system. The domain experts input domain information in the form of detailed domain description, dictionary of keywords, feature diagrams and decision models along with some of the main use cases and sequence diagrams that describe the interactions within the use case. The QoS related information such as domain-specific QoS associations, rules for evaluating, composing, decomposing and measuring QoS attributes are also entered in the KB. Component developers take up component specifications entered by the domain experts (in the form of UMM Specifications) and provide concrete implementations and register them with component registries to make them available to the discovery service. When the System Integrator receives a query, the query is decomposed into subqueries (functional and non-functional) and is issued to the discovery service. The discovery service then uses the multilevel matching algorithm in order to match the query specification against candidate components (UMM Specifications) and returns a component set. This is then statically verified for QoS requirements using the composition rules for static QoS attributes. The glue code is generated for the components to communicate with one another in case of technological heterogeneity. After the system is constructed, code for QoS measurement is instrumented into the code and the results are analyzed to determine if the dynamic QoS constraints are met, after which the system is ready to be deployed.
The objective of this work is to describe the KB and its pieces, and provide some insights into how the KB can be used, with more emphasis on the former part.

4.6 Conclusion

This chapter connects the KB with various concrete implementations that are products of the UniFrame Approach and provides a logical framework of the UA that includes the KB architecture proposed as part of this thesis. This logical architecture can be used as a starting point to create the complete realization of the UA by first solving the issue of technological heterogeneity involved in the various implementations. This is discussed as part of future work in Chapter 7.
CHAPTER 5. CASE STUDY – BANKING DOMAIN

This chapter provides a Banking Domain case study for the knowledgebase architecture described as part of Chapter 3. The Banking domain was selected as an example domain, since it is easy to comprehend and offers enough variety of modules and features to make an interesting example, to describe the knowledgebase with the purpose of presenting an actual instance of the knowledgebase. As part of this exercise, the following artifacts are prepared for the Banking Domain:

- Architecture Information:
  - Description
  - Dictionary
  - Models
  - Diagrams
  - UMM-annotated UDSL
  - QoS Associations
  - Wrapper Generation Information

- QoS Catalog

- Components
  - Deployment Information
  - Test Cases

These artifacts are presented in the following sections. The techniques to arrive at these artifacts have been described over Chapters 2 and 3, so the focus here is to apply these techniques to this example and create every piece of the KB, using the Banking domain concepts and terminology.
Another case study, from the document management domain, was developed as an independent effort as part of UniFrame Research, to describe the Generative Domain Model (GDM) as per [HUA03]. The case study describes the artifacts that result during the development of the GDM, such as Domain Description in terms of Problem Description, Description of General Capabilities, Domain Boundaries, Potential Sources of Information and Potential stakeholders. Following this information, is the Use Case Model and Feature Diagram of the Use Case Model, followed by Feature Diagram of components. The UniFrame Generative Domain Model (UGDM) consists of a problem space (Use Caes Model, QoS Requirement Model, Architecture Model in Heirarchical Form, System-Level Multiplicity Model) and a solution space (Architecture Related Models, Design Feature Related Models, QoS Related Models), and has been described for the Document Management domain. Some of the details of the terms mentioned under UGDM have been briefly described in Section 2.4.

The case study created for the UGDM of the Document Management domain, also follows the principles laid down by the UGDP (UGDM Development Process [HUA03] in order to describe the contents of the UGDM. The case study presented as part of this chapter is an attempt to elaborate further the KB architecture, by providing actual examples as to how the pieces of the KB look for the banking domain, with the same objective as that of the previous case study.

5.1 Architecture

This section describes the architecture related information including the description, domain dictionary, models and some of the UML diagrams associated, the UDSL for the Banking Domain, and a list of QoS Associations. Information about Wrapper generation is included as well.
5.1.1 Description

As explained in Section 3.1.1.1, Domain Description consists of the Project Objectives, Description, Boundaries and Potential Information Sources. This is a high-level description of the problem domain and the technique used (interviewing with stakeholders, etc.) have been discussed in Chapters 2 and 3. The domain experts are interviewed in order to gather as much information about the problem domain, as possible.

Table 5.1 Domain Description for the Banking Domain

<table>
<thead>
<tr>
<th>Domain Description for Banking Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Project Objectives:</strong></td>
</tr>
<tr>
<td>1. Complete all phases of domain analysis in the time frame specified by the senior management.</td>
</tr>
<tr>
<td>2. Generate quality work products at all stages from planning through product development.</td>
</tr>
<tr>
<td>3. Produce comprehensive, high quality documentation as specified by the process.</td>
</tr>
<tr>
<td>4. Successfully implement the project while keeping with the budget outlined.</td>
</tr>
<tr>
<td><strong>Domain Boundaries:</strong></td>
</tr>
<tr>
<td>This domain is made up of banking systems that provide basic banking functions such as account maintenance, balance inquiries, transfers and deposit, and optionally, debit card handling. The banking systems under this domain must be secure and robust, and must provide quick responses to customers at all times. Advanced loan options may be provided to bank customers. Customers should be able to manage their accounts through various interfaces (Online, telephone, ATM Terminals) over and above the traditional counter interface. This is a simple banking application domain, to develop basic banks, with optional basic loan options (such as home loans) made available to customers. Advanced features such as auto loans and credit card management are not dealt with as part of this domain.</td>
</tr>
<tr>
<td><strong>Potential Sources of Information:</strong></td>
</tr>
<tr>
<td>1. Banking Staff (Cashiers, Executives, Counter Representatives) – Have knowledge of day-to-day banking operations.</td>
</tr>
<tr>
<td>2. Application Engineers – have knowledge of other banking applications developed in the past; Have better understanding of application requirements.</td>
</tr>
<tr>
<td>3. Literature (Textbooks, Standards) – May contain key terms and concepts elaborated in a formal manner which is unambiguous and unanimously adopted by the banking community.</td>
</tr>
</tbody>
</table>
5.1.2 Dictionary
The next artifact is the consolidation of the domain dictionary [FIN00a, FIN00b].

Table 5.2 Domain Dictionary for the Banking Domain

<table>
<thead>
<tr>
<th>Domain Dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ASSETS: Personal possessions of value such as cash, real estate and investments.</td>
</tr>
<tr>
<td>2. BANK: An institution that acts as a financial intermediary by receiving money from depositors and lenders, and lends money to borrowers, along with managing accounts.</td>
</tr>
<tr>
<td>3. DEBIT CARD: A payment card that is linked directly to a customer’s bank account that removes the purchase price from the customer’s checking account immediately and typically requires a personal identification number.</td>
</tr>
<tr>
<td>4. DEPOSIT: Money given by a buyer when making a formal offer to bind the sale.</td>
</tr>
<tr>
<td>5. DIRECT DEPOSIT: An automatic deposit of wages or benefits to a customer’s bank account.</td>
</tr>
<tr>
<td>6. COMMERCIAL BANK: A financial institution that provides a broad range of service, from checking and savings accounts to business loans and credit cards.</td>
</tr>
<tr>
<td>7. E-COMMERCE: A system used to conduct business transactions of buying and selling goods and services over a computer network.</td>
</tr>
<tr>
<td>8. ELECTRONIC FUNDS TRANSFER: The transfer of money between accounts by consumer electronic systems such as automated teller machines (ATMs), and electronic payment of bills.</td>
</tr>
<tr>
<td>9. CASHIER: Person who manages bank accounts on behalf of customers.</td>
</tr>
<tr>
<td>10. CUSTOMER: Person who owns the account(s).</td>
</tr>
<tr>
<td>11. USER: Cashiers, Customers, Counter Representatives.</td>
</tr>
<tr>
<td>12. ATM: Acronym for ‘Automated Teller Machine’, a machine at a bank branch or other location which enables a customer to perform basic banking activities even when the bank is closed.</td>
</tr>
<tr>
<td>13. PIN: Acronym for ‘Personal Identification Number’ required while accessing an ATM or a Phone Interface in conjunction with a user identifier.</td>
</tr>
</tbody>
</table>
5.1.3 Models

Continuing with the approach described in Chapter 3, the next artifact is a set of two models – the feature model (represented with a feature diagram) and a decision model (modeled using the language generated by the grammar in Section 3.1.1.3.2).

5.1.3.1 Feature Model (Feature Diagram)

The feature diagram is shown below (Figure 5.1). The systems under the banking domain are modeled as 4 subsystems – the User Interface Subsystem, Transaction Subsystem, Validation Subsystem and Loan Management Subsystem. The details of each subsystem are also indicated in Figure 5.1.

![Feature Diagram](image)

**Figure 5.1: Feature Diagram for the Banking Domain**

The *User Interface Subsystem* is made up by *Counter Representatives*, and optionally *ATMs* and *Phone Services*. *Transaction Manager* and *Account Server* come
under the *Transaction Subsystem*. *Transaction Manager* consists of *Basic Functions Manager* and optionally, a *Debit Card Manager*. The *Basic Functions Manager* must have the *Account Manager* that deals with account operations, and may have the *Statement Manager* that deals with managing statements and its views. *Account Server* keeps track of customer accounts and operations pertaining to them. Since the system must be secure in order to meet the domain requirements, the *Validation Subsystem* is in place, which validates Counter Representatives (who manage accounts on the customer’s behalf) and Customers through the *Counter Rep Validation Manager* and *Customer Validation Manager*. The *Loan Management System* is optional, as defined in Table 5.1, and the *Basic Loan Manager* is mandatory if the System Integrator opts for *Loan Management Subsystem*. *Advanced Loan Manager* remains optional.

As mentioned in Chapter 3, feature diagrams depict features of the system but do not necessarily communicate the constraints and dependencies between the features. These are expressed through the UMM-annotated UDSL.

### 5.1.3.2 Decision Model (Flowchart Notation)

This section describes the Decision Model created for the Banking Domain. The mandatory features described in the feature diagram (Figure 5.1) have to be included as part of any system under the Banking Domain, so the decision model includes only the features that are optional, and lend more domain-specific knowledge to the application programmer, or the System Integrator in this case, as to what the ‘choice’ of the optional feature implies. Following is the Decision Model for the Banking Domain Example that details the relevant decisions in the process of application design. Mandatory features are excluded, and the decision flow of opting for optional features is described. The Flowcharting notation has been drawn for simplicity, and the model is elaborated in the form of the language that is generated by the grammar in Table 3.3. Its this language that is stored as part of the KB.
Figure 5.2: Decision Model for the Banking Domain

The language generated by the grammar can be described as below:

Table 5.3 Decision Model for Banking Domain

IF “Loan Management Required” {
    INCLUDE “Loan Management System”
    IF “Loans with Adv. Options Required”{
        INCLUDE “Adv Loan Manager”
    }
}

IF “Provide Customer Interface”{
    INCLUDE “Customer Validation Server”
    IF “ATM Required”{
        INCLUDE “ATM”
    }
    IF “Phone Interface Required”{
        INCLUDE “Phone”
    }
}
5.1.4 Diagrams

As per the approach detailed in Chapter 3, UML Use Case Diagrams and Sequence Diagrams have to be drawn for the basic use cases involved in any system under the problem domain. This section provides Use Case and Sequence Diagrams for some of the important use cases as part the Banking Domain applications.

5.1.4.1 Use Case Diagrams

This section describes the use case diagram created for the banking domain. This diagram attempts to cover most of the use cases in the banking domain. Figure 5.3 shows two actors – customer and counter representative (cashier). It highlights the main use cases such as Apply for Loan, Withdraw at ATM, Get Statement and Transfer on Phone. The associations are indicated in the use case diagram.
5.1.4.2 Sequence Diagrams

As described in Section 3.1.1.4, at least one sequence diagram per use case should be drawn, but more the number of sequence diagrams, higher the clarity of the domain model. This section presents 2 sequence diagrams. Figure 5.4 describes the sequence diagram for Transferring Funds across accounts over the phone interface and Figure 5.5 describes the sequence diagram for withdrawing funds at an ATM.
Figure 5.4: Sequence Diagram for Transfer Funds

Figure 5.5: Sequence Diagram for Withdraw at ATM
5.1.5 UMM-Annnotated UDSL

This section describes the UMM-Annnotated UDSL which refines the feature diagram discussed before, and describes the constraints and dependencies in a more elaborate manner. The BNF Grammar for the UMM-Annnotated UDSL is described in Section 3.1.1.5 (Table 3.1), and illustrates feature expressions, constraint expressions, design feature expressions and use case expressions. The <Abstract_Component> is expanded as part of the UMM Annotation and UMM Specifications for the Abstract Components (the leaves of the feature diagram) is given. Table 5.3 below describes the UMM-Annnotated UDSL for the Banking Domain. As part of the UDSL, some important components and use cases are described to provide an idea of the UDSL structure.

Table 5.4 UMM-Annnotated UDSL for the Banking Domain

<table>
<thead>
<tr>
<th>UMM-Annnotated UDSL for the Banking Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank = ALL (UserInterfaceSubsystem, TransactionSubsystem, ValidationSubsystem, LoanManagementSubsystem?)</td>
</tr>
<tr>
<td>UserInterfaceSubsystem = ALL (Counter, ATM?, Phone?)</td>
</tr>
<tr>
<td>TransactionSubsystem = ALL (TransactionManager, AccountServer)</td>
</tr>
<tr>
<td>ValidationSubsystem = ALL (CounterRepValidationServer, CustomerValidationServer)</td>
</tr>
<tr>
<td>LoanManagementSubsystem = ALL (BasicLoanManager, AdvancedLoanManager?)</td>
</tr>
<tr>
<td>TransactionManager = ALL (BasicFunctionsManager, DebitCardManager?)</td>
</tr>
<tr>
<td>BasicFunctionsManager = ALL (AccountManager, StatementsManager?)</td>
</tr>
<tr>
<td>MULTIPLICITY ((UserInterfaceSubsystem, Counter): 1…*)</td>
</tr>
<tr>
<td>MULTIPLICITY ((UserInterfaceSubsystem, ATM): 0…*)</td>
</tr>
<tr>
<td>MULTIPLICITY ((UserInterfaceSubsystem, Phone): 0…*)</td>
</tr>
<tr>
<td>MULTIPLICITY ((TransactionManager, DebitCardManager): 0…1)</td>
</tr>
<tr>
<td>MULTIPLICITY ((BasicFunctionsManager, StatementsManager): 0…1)</td>
</tr>
<tr>
<td>MULTIPLICITY ((BasicFunctionsManager, AccountManager): 1)</td>
</tr>
<tr>
<td>MULTIPLICITY ((Bank, LoanManagementSubsystem): 0…1)</td>
</tr>
<tr>
<td>MULTIPLICITY ((ValidationSubsystem, CustomerValidationServer): 0…1)</td>
</tr>
<tr>
<td>MULTIPLICITY ((ValidationSubsystem, CounterRepValidationServer): 1)</td>
</tr>
<tr>
<td>MULTIPLICITY ((LoanManagementSubsystem, AdvancedLoanManager): 0…1)</td>
</tr>
<tr>
<td>MULTIPLICITY ((LoanManagementSubsystem, BasicLoanManager): 1)</td>
</tr>
<tr>
<td>MULTIPLICITY ((TransactionSubsystem, TransactionManager): 1)</td>
</tr>
<tr>
<td>MULTIPLICITY ((TransactionSubsystem, AccountServer): 1)</td>
</tr>
<tr>
<td>MULTIPLICITY ((TransactionManager, BasicFunctionsManager): 1)</td>
</tr>
</tbody>
</table>
DEFAULT (UserInterfaceSubsystem: Counter)
DEFAULT (ValidationSubsystem: CounterRepValidationServer)

REQUIRE (ATM, CustomerValidationServer)
REQUIRE (Phone, CustomerValidationServer)

INTERACT (Counter, CounterRepValidationServer)
INTERACT (Counter, AccountManager)
INTERACT (Counter, AccountServer)
INTERACT (Counter, StatementsManager)
INTERACT (Counter, DebitCardManager)
INTERACT (ATM, CustomerValidationServer)
INTERACT (ATM, AccountManager)
INTERACT (ATM, StatementsManager)
INTERACT (Phone, CustomerValidationServer)
INTERACT (Phone, AccountManager)
INTERACT (BasicLoanManager, StatementsManager)
INTERACT (AdvancedLoanManager, StatementsManager)

INTERFACE (Counter: PROVIDED_INTERFACE: (IAccountManager),
            REQUIRED_INTERFACE:
            (IAccountManager, IValidateUser))
INTERFACE (ATM: PROVIDED_INTERFACE: (IAccountManager),
            REQUIRED_INTERFACE:
            (IAccountManager, IValidateUser))
INTERFACE (Phone: PROVIDED_INTERFACE: (IAccountManager),
            REQUIRED_INTERFACE:
            (IAccountManager, IValidateUser))
INTERFACE (AccountManager: PROVIDED_INTERFACE: (IAccountManager),
            REQUIRED_INTERFACE: (IValidateUser))
INTERFACE (AccountServer: PROVIDED_INTERFACE: (IAccountServer),
            REQUIRED_INTERFACE:
            (IAccountServer))
INTERFACE (DebitCardManager: PROVIDED_INTERFACE: (IDebitCard),
            REQUIRED_INTERFACE:
            (IValidateUser))
INTERFACE (CounterRepValidationServer: PROVIDED_INTERFACE:
            (IValidateUser),
            REQUIRED_INTERFACE:
            (IValidateUser))
INTERFACE (BasicLoanManager: PROVIDED_INTERFACE: (ILoanManager),
            REQUIRED_INTERFACE: (IStatementManager))
INTERFACE (StatementsManager: PROVIDED_INTERFACE:
            (IStatementManager),
            REQUIRED_INTERFACE: (IValidateUser))
ABSTRACT COMPONENT: Counter

COMPONENT ATTRIBUTES:
NAME: Counter
AUTHOR: N/A
SUBCASE: N/A
DESCRIPTION: The component performs the functions carried out by a Counter representative in the bank.
CHANGELIST: N/A

COMPUTATION ATTRIBUTES:
INHERENT ATTRIBUTES:
VERSION: N/A
COMP_AUTHOR: N/A
ID: N/A
EXECUTION ENVIRONMENT: N/A
SYSTEM NAME: Bank
DOMAIN NAME: Banking Systems
VALIDITY: N/A
COMPONENT MODEL: N/A
ATOMICITY: Yes
DATE DEPLOYED: N/A
REGISTRATION: N/A

FUNCTIONAL ATTRIBUTES:
FUNCTION CONTRACTS:
  FUNCTION: Boolean ValidateCounterRep (String RepName, String Password)
  SEMANTIC CONTRACT:
    PRE: RepName != NULL && Password != NULL
    POST: RetValue != NULL
    INVARIANT: TRUE
  CONCURRENCY: N/A
  TECHNOLOGY: N/A
  DESCRIPTION: Validates Counter Representative’s Identity so he/she can proceed to account operations

  FUNCTION: Float GetBalance (String AccountNumber)
  SEMANTIC CONTRACT:
    PRE: AccountNumber != NULL && ValidateCounterRep (RepName, Password)
    POST: RetValue != NULL
    INVARIANT: TRUE
  CONCURRENCY: N/A
  TECHNOLOGY: N/A
DESCRIPTION: Obtains current balance on Customer Account supplied.

FUNCTION: Int Deposit(String AccountNumber, Float Amount)

SEMANTIC CONTRACT:
PRE: AccountNumber != NULL && ValidateCounterRep (RepName, Password) && Amount != 0
POST: RetValue != 0
INVARIANT: TRUE
CONCURRENCY: N/A
TECHNOLOGY: N/A

DESCRIPTION: Deposits given amount on Customer Account supplied.

FUNCTION: Int Withdraw(String AccountNumber, Float Amount)

SEMANTIC CONTRACT:
PRE: AccountNumber != NULL && ValidateCounterRep (RepName, Password) && Amount != NULL
POST: RetValue != 0
INVARIANT: TRUE
CONCURRENCY: N/A
TECHNOLOGY: N/A

DESCRIPTION: Deducts specified amount from the Customer Account supplied and updates current balance.

FUNCTION: Int Transfer(String AccountNumber, String FromType, String ToType, Float Amount)

SEMANTIC CONTRACT:
PRE: AccountNumber != NULL && ValidateCounterRep (RepName, Password) && Amount != NULL && FromType != ToType
POST: RetValue != NULL
INVARIANT: TRUE
CONCURRENCY: N/A
TECHNOLOGY: N/A

DESCRIPTION: Transfers specified amount from Checking to Savings or vice versa in the Customer Account supplied.

ALGORITHMS: N/A
DESIGN PATTERNS: N/A
KNOWN USAGES: Used as Counter Representative
ALIASES: Cashier
RESOURCES: N/A

COOPERATION ATTRIBUTES:

COLLABORATORS:
- FUNCTIONNAME: ValidateCounterRep
  EXPECTED: CounterRepValidationServer
  REQUIRED: CounterRepValidationServer
  PROVIDED: UserInterfaceSubsystem

FUNCTIONNAME: GetBalance
EXPECTED: CounterRepValidationServer
REQUIRED: CounterRepValidationServer, AccountOperations
PROVIDED: UserInterfaceSubsystem

AUXILIARY ATTRIBUTES:
- MOBILE: N/A
- MOBILITY: N/A
- FAULT TOLERANCE: N/A
- SECURITY LIST: N/A

SERVICE ATTRIBUTES:
- PARALLELISM CONSTRAINT: Asynchronous
- ORDERING CONSTRAINT: N/A
- EXECUTION RATE: N/A
- AVAILABLE RESOURCES: N/A

QoS ATTRIBUTES:
- QoS ID: Throughput
  VALUE: N/A
- EXTENSION:
- QoS ID: TurnAroundTime
  VALUE: N/A
- EXTENSION:
- QoS ID: Security
  VALUE: N/A
- EXTENSION:

ABSTRACT COMPONENT: ATM

COMPONENT ATTRIBUTES:
- NAME: ATM
- AUTHOR: N/A
- SUBCASE: N/A
- DESCRIPTION: The component performs the functions carried out by a bank ATM (Automated Teller Machine).
- CHANGELIST: N/A
COMPUTATION ATTRIBUTES:
INHERENT ATTRIBUTES:
   VERSION: N/A
   COMP_AUTHOR: N/A
   ID: N/A
   EXECUTION ENVIRONMENT: N/A
   SYSTEM NAME: Bank
   DOMAIN NAME: Banking Systems
   VALIDITY: N/A
   COMPONENT MODEL: N/A
   ATOMICITY: Yes
   DATE DEPLOYED: N/A
   REGISTRATION: N/A

FUNCTIONAL ATTRIBUTES:
FUNCTION CONTRACTS:
   FUNCTION: Boolean ValidateCustomer (String CustomerId, String Password)
   SEMANTIC CONTRACT:
      PRE: CustomerId != NULL && Password != NULL
      POST: RetValue != NULL
      INVARIANT: TRUE
   CONCURRENCY: N/A
   TECHNOLOGY: N/A
   DESCRIPTION: Validates Customer’s Identity so he/she can proceed to account operations

   FUNCTION: Float GetBalance (String AccountNumber)
   SEMANTIC CONTRACT:
      PRE: AccountNumber != NULL && ValidateCustomer (CustomerId, Password)
      POST: RetValue != NULL
      INVARIANT: TRUE
   CONCURRENCY: N/A
   TECHNOLOGY: N/A
   DESCRIPTION: Obtains current balance on Customer Account supplied.

   FUNCTION: Int Withdraw(String AccountNumber, Float Amount)
   SEMANTIC CONTRACT:
      PRE: AccountNumber != NULL && ValidateCustomer (CustomerId, Password) && Amount != NULL
      POST: RetValue != 0
INVARIANT: TRUE
CONCURRENCY: N/A
TECHNOLOGY: N/A
DESCRIPTION: Deducts specified amount from the Customer Account supplied and updates current balance.

FUNCTION: Int Transfer(String AccountNumber, String FromType, String ToType, Float Amount)

SEMANTIC CONTRACT:
PRE:  AccountNumber != NULL &&
      ValidateCustomer (CustomerId, Password) &&
      Amount != NULL &&
      FromType != ToType
POST:  RetValue != NULL
INVARIANT: TRUE
CONCURRENCY: N/A
TECHNOLOGY: N/A
DESCRIPTION: Transfers specified amount from Checking to Savings or vice versa in the Customer Account supplied.

ALGORITHMS: N/A
DESIGN PATTERNS: N/A
KNOWN USAGES: Used as ATM
ALIASES: N/A
RESOURCES: N/A

COOPERATION ATTRIBUTES:
COLLABORATORS:
FUNCTIONNAME: ValidateCustomer
EXPECTED: CustomerValidationServer
REQUIRED: CustomerValidationServer
PROVIDED: UserInterfaceSubsystem
FUNCTIONNAME: GetBalance
EXPECTED: CustomerValidationServer
REQUIRED: CustomerValidationServer, AccountOperations
PROVIDED: UserInterfaceSubsystem

AUXILIARY ATTRIBUTES:
MOBILE: N/A
MOBILITY: N/A
FAULT TOLERANCE: N/A
SECURITY LIST: N/A

SERVICE ATTRIBUTES:
PARALLELISM CONSTRAINT: Asynchronous
ORDERING CONSTRAINT: N/A
EXECUTION RATE: N/A
AVAILABLE RESOURCES: N/A
QoS ATTRIBUTES:
  QoS ID: Throughput
  VALUE: N/A
  EXTENSION:
  QoS ID: TurnAroundTime
  VALUE: N/A
  EXTENSION:
  QoS ID: Security
  VALUE: N/A
  EXTENSION:

ABSTRACT COMPONENT: StatementManager

COMPONENT ATTRIBUTES:
  NAME: ATM
  AUTHOR: N/A
  SUBCASE: N/A
  DESCRIPTION: The component performs the functions of keeping track of
bank account history, and providing different types of statements
  CHANGELIST: N/A

COMPUTATION ATTRIBUTES:
  INHERENT ATTRIBUTES:
    VERSION: N/A
    COMP_AUTHOR: N/A
    ID: N/A
    EXECUTION ENVIRONMENT: N/A
    SYSTEM NAME: Bank
    DOMAIN NAME: Banking Systems
    VALIDITY: N/A
    COMPONENT MODEL: N/A
    ATOMICITY: Yes
    DATE DEPLOYED: N/A
    REGISTRATION: N/A

FUNCTIONAL ATTRIBUTES:
  FUNCTION CONTRACTS:
    FUNCTION: String[] GetQuickStatement(String
    AccountNumber)

    SEMANTIC CONTRACT:
      PRE: AccountNumber
      POST: RetValue != NULL
INVARIANT: TRUE
CONCURRENCY: N/A
TECHNOLOGY: N/A
DESCRIPTION: Provides a quick, concise statement of account activities.

FUNCTION: String[] GetHistory (String AccountNumber)
SEMANTIC CONTRACT:
  PRE: AccountNumber != NULL
  POST: RetValue != NULL
  INVARIANT: TRUE
CONCURRENCY: N/A
TECHNOLOGY: N/A
DESCRIPTION: Obtains detailed activity report of the Customer Account supplied.

ALGORITHMS: N/A
DESIGN PATTERNS: N/A
KNOWN USAGES: To provide various views of account activities
ALIASES: StatementExpress, EasyStatementMaker
RESOURCES: N/A

COOPERATION ATTRIBUTES:
  COLLABORATORS:
    FUNCTIONNAME: GetHistory
    EXPECTED: AccountManager
    REQUIRED: AccountManager
    PROVIDED: BasicFunctionsManager

AUXILIARY ATTRIBUTES:
  MOBILE: N/A
  MOBILITY: N/A
  FAULT TOLERANCE: N/A
  SECURITY LIST: N/A

SERVICE ATTRIBUTES:
  PARALLELISM CONSTRAINT: Asynchronous
  ORDERING CONSTRAINT: N/A
  EXECUTION RATE: N/A
  AVAILABLE RESOURCES: N/A
  QoS ATTRIBUTES:
    QoS ID: Throughput
    VALUE: N/A
    EXTENSION:
    QoS ID: TurnAroundTime
VALUES: N/A
EXTENSION:
QoS ID: Security
VALUES: N/A
EXTENSION:

DepositFundsCase_1: PATH_C (<Counter, CounterRepValidationServer>,
AccountManager)
WithdrawFundsCase_1: PATH_C (<ATM, CustomerValidationServer>,
AccountManager)
TransferFundsCase_1: PATH_C (<ATM, CustomerValidationServer>,
AccountManager)
TransferFundsCase_2: PATH_C (<Phone, CustomerValidationServer>,
AccountManager)
OpenAccountCase_1: PATH_C (<Counter, CounterRepValidationServer>,
AccountServer)
IssueNewDebitCardCase_1: PATH_C (<Counter, CounterRepValidationServer>,
DebitCardManager)

DepositFundsCase_1: PATH_F (Counter.ValidateCounterRep (RepName, Password)[CP2S],
AccountManager.Deposit(AccountId, Amount)[CP2S])
WithdrawFundsCase_1: PATH_F (Counter.ValidateCounterRep (RepName, Password)[CP2S],
AccountManager.Withdraw(AccountId, Amount)[CP2S])
TransferFundsCase_1: PATH_F (ATM.ValidateCustomer(Custname, Password)[CP2S],
AccountManager.Transfer(AccountID, FromType, ToType, Amount)[CP2S])
TransferFundsCase_2: PATH_F (Phone.ValidateCustomer(Custname, Password)[CP2S],
AccountManager.Transfer(AccountID, FromType, ToType, Amount)[CP2S])
OpenAccountCase_1: PATH_F (Counter.ValidateCounterRep(RepName, Password)[CP2S],
AccountServer.OpenAccount(CustomerInfo, AccountType, InterestRate, StartingBalance) [CP2S]
AccountServer.SaveAccount(AccountId) [CP2A]
IssueNewDebitCardCase_1: PATH_F (Counter.ValidateCounterRep(RepName, Password)[CP2S],
DebitCardManager.IssueNewCard(CustomerName,
5.1.6 QoS Associations

Since the project objectives indicate that security, and quick response round the clock is an essential characteristic of the system, the following QoS attributes have been identified as relevant to this domain:

1. Turnaround Time
2. Security

These attributes are used for static QoS property prediction as well as dynamic QoS property validation. These are the attributes of interest to the Banking Domain as per the Domain Description in Table 5.1. As elaborated in Chapter 3, when a component set is selected by the System Integrator, the static QoS properties are composed using the QoS attribute composition formulae (Appendix A). Details of these properties can be found in Appendix A.

5.2 QoS Catalog

The entire QoS Catalog is part of the knowledgebase. The QoS associations, i.e., the list of QoS attributes relevant to the Banking Domain are listed in Section 5.1.6. QoS attributes may not be relevant to every domain in concern, but the QoS Catalog is a domain-independent entity that is present in the knowledgebase for all Domains to reference. The catalog is presented in complete detail with QoS attribute metrics, evaluation processes, composition/decomposition rules is available as part of Appendix A.
5.2.1 Property Prediction and Instrumentation

As elaborated in Section 3.1.2.4, the QoS properties of the DCS are composed in order to test the overall quality of the system. The values of the static QoS attributes are composed as per the static composition rules (Appendix A), and the dynamic QoS attributes are composed using the AspectJ style advice and join-point notation to log values [AOP00]. The two attributes relevant to this banking domain example are security and turnaround time.

Security is a static QoS property following the composition rule [SUN03]:

\[
MTTI = \min(MTTI_1, MTTI_2, \ldots, MTTI_n)
\]

where MTTI is the system security value (Mean-Time-to-Intrusion or Minimum-Time-to-Intrusion) and MTTI_i (i=1, 2, …, n) is the security value of the i^{th} component. So, the overall system is as secure as the least-secure component. Hence, for a given component set, the minimum of all MTTI_i’s is computed, and if that is lesser than the MTTI of the desired system, then the component set is pursued further for system generation.

As far as turnaround time is concerned, it is a dynamic QoS property. As per the KB architecture, dynamic QoS properties are measured using advice code that is run at the specified join-points. The advice code (written in Java for this example) for turn around time, consists of two method calls, startTimer() and stopTimer(). The code snippet is shown below (Tables 5.4, 5.5).

<table>
<thead>
<tr>
<th>Table 5.5 Advice Code for startTimer() for Turnaround Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public void QoS_start()</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>//declarations:</td>
</tr>
<tr>
<td>Public Calendar QoS_Turnaround_Time;</td>
</tr>
<tr>
<td>Public long QoS_start_Time, QoS_stop_Time, QoS_time_Taken;</td>
</tr>
<tr>
<td>//start timer</td>
</tr>
<tr>
<td>Calendar QoS_Turnaround_Time = new Calendar();</td>
</tr>
<tr>
<td>long QoS_start_Time = QoS_Turnaround_Time.getTimeInMillis();</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>
Table 5.4 shows the advice code for starting the timer, and Table 5.5 shows the code snippet for stopping the timer and obtaining the time difference and writing it out to a file. The file is then read and the judgment is made whether the turnaround time satisfies the turnaround time requirement of the desired QoS.

Table 5.6 Advice Code for stopTimer() for Turnaround Time

```java
Public void QoS_stop()
{
    //stop timer
    QoS_stop_Time = QoS_Turnaround_Time.getTimeInMillis();
    QoS_time_Taken = stop_Time – start_Time;
    Long T = new Long(time_Taken);
    try
    {
        BufferedWriter out = new BufferedWriter
        (new FileWriter("Log_Turnaround_Time.log"));
        out.write( T.intValue());
        out.close();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}
```

As for the join-points to run these advices, they are specified using AspectJ style notation. For instance, to get the turnaround time of the Transfer() method, the following syntax can be used:

Table 5.7 Join points for measuring turnaround time for Transfer method

(`“PointCut_Turnaround_Time_Transfer.txt”“`)
The “Log_Turnaround_Time.log” file can then be read to obtain the exact time taken in milliseconds. For other dynamic QoS attributes, similar advice codes can be inserted at different point-cuts in the component code (and/or the glue code), and the results can be logged and processed. The KB stores an advice-code library which contains advice code pieces to measure other QoS attributes. The AOP-style notation is used to measure dynamic QoS properties and values written out by the advice code are parsed to validate these properties. This mechanism is proposed as part of this KB architecture in order to perform dynamic validation. Event Grammars are used as part of UA to log the information that is generated while running advices and interpret it to validate the dynamic QoS properties.

5.3 Components

This section discusses some of the component-related information for the banking domain. As part of the process of component development, the following steps are carried out:

1. The Component Developer selects a particular Feature Diagram, and a specific Abstract Component for development, and has access to the UMM Specification of the component.

2. The Component Developer can then choose to use the Skeletal Code Generator described in Section 4.2.1, which inputs the UMM Specification and generates skeletal code, which contains placeholders for the Component Developer to add functionality and concentrate on business logic. The code generator handles several issues related to maintaining consistency between code and specifications, so that the specifications remain in sync with the code at all times, since it is the specifications that are used by the discovery service to make a selection about what component set to use.

3. The Component Developer can also handwrite the code as per the specification, but this may involve extra effort on his/her part to ensure correctness with respect to the specification. The Developer, in that case, must use formal techniques such as Design
by Contract [MEY92] while developing the implementation. The Component Developer also needs to incorporate advices and join points (for measuring dynamic QoS properties) as described in Chapter 3.

4. In either case, the above steps should result in a Component Implementation that is in synchronization with the specification. The component specification has to then be registered with one of the registries where the discovery service can examine it against query specifications (using multilevel matching for instance) to decide if the component is appropriate for the given query (issued by system integrator).

The knowledgebase maintains the skeletal code generator so that code can be generated from UMM Specifications, and the Component Developer is assisted in the process of provide a complete, correct implementation.

The knowledgebase also contains unit test cases that are written by testing experts for testing the basic functionality of components. One category of test cases is hand written test cases that are executed by testing experts by following the steps given in the test case description. A general format for writing a test case has been formulated for this purpose. This section describes some unit test cases that have been written to test the functionality of a couple of concrete components. Another category of test cases are those that are automatically generated using the functional interface of the component (available from the UMM Specification); these are not covered as part of the case study (and the knowledgebase at this point). The mechanism to generate automatic test cases from functional contracts of components can nevertheless be incorporated as part of the knowledgebase at a later stage when more work on automatic test case generation is carried out, since it is a vast field in itself.

The two sections that follow show how the component deployment information can be described, and also provide snapshots of some handwritten test cases.
5.3.1 Component Deployment Information

This section describes Component Deployment Information for the AccountManager, AccountServer and ATM Components. The information is detailed in Tables 5.4, 5.5 and 5.6. Part of this information is available as part of the UMM Specification of the component (embedded in the UMM-annotated UDSL), but it is consolidated here as part of the actual deployment/implementation related information in the KB.

Table 5.8 Component Deployment Information for AccountManager of the Banking Domain

<table>
<thead>
<tr>
<th>Component Deployment Information for AccountManager of the Banking Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name and Description:</strong></td>
</tr>
<tr>
<td>Name: AccountManager</td>
</tr>
<tr>
<td>Domain: Banking Domain</td>
</tr>
<tr>
<td>Description: Performs the functions related to operations on customer accounts such as deposit, withdrawal and transfer of funds, balance checks, etc.</td>
</tr>
</tbody>
</table>

| **Component Technology:**                       |
| Programming Language: Java Version 1.4.2       |
| Data Sources Used: Database connections to MySQL Server |
| Component Model: J2EE 1.2.1                     |
| Operating System requirements: Sun Solaris or Redhat Linux 9 |
| Virtual Machine requirements: Requires JDK 1.5  |

| **Component Deployment:**                       |
| Location of Specification:                      |
| [http://www.bankingdomainexamples/specifications/AccountManager.xml](http://www.bankingdomainexamples/specifications/AccountManager.xml) |
| Location of Code:                               |
| [ftp://bankingdomainexamples/code/AccountManager.jar](ftp://bankingdomainexamples/code/AccountManager.jar) |
Table 5.9 Component Deployment Information for *AccountServer* of the Banking Domain

<table>
<thead>
<tr>
<th>Component Deployment Information for <em>AccountServer</em> of the Banking Domain</th>
</tr>
</thead>
</table>
| **Name and Description:**  
Name: AccountServer  
Domain: Banking Domain  
Description: Performs the operations on handling customer accounts such as open, save and close accounts, etc.  |
| **Component Technology:**  
Programming Language: Java Version 1.4.2  
Data Sources Used: Database connections to MySQL Server 4.1  
Component Model: J2EE 1.3.1  
Operating System requirements: Redhat Linux 9  
Virtual Machine requirements: Requires JDK 1.5  |
| **Component Deployment:**  
Location of Specification:  
http://www.bankingdomainexamples/specifications/AccountServer.xml  
Location of Code:  
ftp://bankingdomainexamples/code/AccountServer.jar  |

Table 5.10 Component Deployment Information for *ATM* of the Banking Domain

<table>
<thead>
<tr>
<th>Component Deployment Information for <em>ATM</em> of the Banking Domain</th>
</tr>
</thead>
</table>
| **Name and Description:**  
Name: ATM  
Domain: Banking Domain  
Description: Performs the functions of an Automated Teller Machine (ATM) such as validating customers, and withdrawal and balance inquiry operations.  |
| **Component Technology:**  
Programming Language: Visual C#  
Data Sources Used: Database connections to MySQL Server 4.1  
Component Model: Visual Studio .NET 2.0  
Operating System requirements: Windows XP Pro  
Virtual Machine requirements: Requires .NET Runtime Support  |
5.3.2 Component Test Cases

This section describes sample test cases concentrating only on the ATM component. The test cases described in Tables 5.7, 5.7 and 5.9, test the ‘Login’, ‘Withdraw’ and ‘Transfer’ features of the ATM component respectively. The format for test cases is the one described in [TES] and involves primarily the execution environent for test cases and the action list. These test cases do not cover the entire functionality of the software component. The format for test cases is pre-defined as part of the KB and test cases are created for each software component based on this format.

Table 5.11 Test Case for ATM

<table>
<thead>
<tr>
<th>Information:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Id: TCI-001</td>
</tr>
<tr>
<td>Test Creator: XYZ</td>
</tr>
<tr>
<td>Version: v1.0</td>
</tr>
<tr>
<td>Name: ATM Login</td>
</tr>
<tr>
<td>Intent: The purpose of this test is to test the login validation feature of the ATM.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing Environment: The ATM component should be tested in Visual Studio .NET Environment. The ATM Component should have access to the Accounts Database (MySQL) (local or remote).</td>
</tr>
<tr>
<td>Initialization of Parameters: N/A</td>
</tr>
<tr>
<td>Termination Rules: N/A</td>
</tr>
<tr>
<td>Action List:</td>
</tr>
<tr>
<td>1. Enter username: “Admin” //E.g. of a valid username</td>
</tr>
<tr>
<td>2. Enter PIN: “” //Blank PIN (Personal Identification Number)</td>
</tr>
<tr>
<td>3. Perform Login</td>
</tr>
</tbody>
</table>

Input Data:
Table 5.12 Test Case for ATM -2

Test Case for ATM

Information:
Test Case Id: TCI-002
Test Creator: XYZ
Version: v1.0
Name: ATM Withdraw
Intent: The purpose of this test is to test the withdraw feature of the ATM.
Process:
Testing Environment: The ATM component should be tested in Visual Studio .NET Environment. The ATM Component should have access to the Accounts Database (MySQL) (local or remote).
Initialization of Parameters: N/A
Termination Rules: N/A
Action List:
1. Enter username: “Admin” //E.g. of a valid username
2. Enter PIN: “8888” // PIN corresponding to ‘Admin’
3. Perform Login
4. Select type of account: “Checking1” or “Checking2”
5. Enter amount to be withdrawn: “0.00 USD”
6. Perform Vend
7. Enter amount to be withdrawn: “10,000 USD”
8. Perform Vend

Input Data:
“Admin”, “8888”
[Choice] Checking1, Checking2
“0.00 USD”
“10,000 USD”

Conclusion:
Expected:
1. The login screen should first display the message “Welcome Admin”
2. The login screen should display “Account Selected: Checking 1” or “Account Selected: Checking 2”
3. After entering “0.00 USD”, and ‘Vend’, the screen should prompt “Enter an amount in multiples of 20”
4. After entering “10,000 USD” and ‘Vend’, the screen should prompt “You have exceeded the limit set for ATM Withdrawals. Contact your Counter Representative for assistance”
5. Exit

Status: If above flow is exhibited, then PASS else FAIL.

### Table 5.13 Test Case for ATM-3

| Information: |
| Test Case Id: TCI-003 |
| Test Creator: XYZ |
| Version: v1.0 |
| Name: ATM Transfer |
| Intent: The purpose of this test is to test the ‘Transfer Amount’ feature of the ATM. |

**Process:**

Testing Environment: The ATM component should be tested in Visual Studio .NET Environment. The ATM Component should have access to the Accounts Database (MySQL) (local or remote).

Initialization of Parameters: N/A

Termination Rules: N/A

**Action List:**

1. Enter username: “Admin” //E.g. of a valid username
2. Enter PIN: “8888” //PIN corresponding to ‘Admin’
3. Perform Login
4. Select ‘TransferFrom’ Account as ‘Savings1’ or ‘Savings2’ or ‘Checking1’ or ‘Checking2’ – Choose ‘Savings1’
5. Select ‘TransferTo’ Account as ‘Savings1’ or ‘Savings2’ or ‘Checking1’ or ‘Checking2’ – Choose ‘Savings1’

**Input Data:**

“Admin”, “8888”

[Choice] Savings1, Savings2, Checking1, Checking2 – Selected: Savings1
[Choice] Savings1, Savings2, Checking1, Checking2 – Selected: Savings1

**Conclusion:**

**Expected:**

1. The login screen should display the message “TransferFrom and TransferTo accounts cannot be the same”
2. Attempt retry

Status: If login screen shows “TransferFrom and TransferTo accounts cannot be the same” then PASS else FAIL.
These test cases do not fully test the functionality of the ATM Component, but attempt to provide a structure for the test developers to construct more test cases with higher complexity.

Other pieces of the KB, such as QoS decomposition rules (for non-functional decomposition of the query) and QoS composition rules (for static property prediction for a component set) have been discussed as part of Section 3.1.2.3. In this example, the static property prediction, for instance, would be for the attribute ‘security’. The composition rule corresponding to security (Appendix A) is:

\[ MTTI = \min (MTTI_1, MTTI_2, \ldots, MTTI_n) \]

where MTTI is the system security value, MTTI_i (i=1, 2, ..., n) is the security value of component i. This essentially means that the system as secure as the least secure component. So, given a component set with MTTI_i’s for each component, the minimum of all MTTI_i’s would be chosen as the MTTI of the resulting system. If this value of MTTI is greater than or equal to the MTTI of the desired system (as requested by the client), then the component set is pursued further, otherwise it is disbanded for a new selection, and the process is repeated.

Decomposition rules (non-functional decomposition of queries) are applied when the query is intercepted by the system integrator. For example, for the attribute ‘turn around time’, the corresponding decomposition rule (Appendix A) is

\[ MTAT_i < MTAT \]

where MTAT_i is the turn-around time measure of component i (i=1, 2, ..., n), MTAT is the turn-around time measure of system. This means that each component in the resulting system must have a turnaround time that is lesser than the turnaround time of the desired system, in order to satisfy the non-functional requirement. So when the query is functionally decomposed (based on feature model and decision model) and the list of ‘abstract components’ is derived, a constraint MTAT_i < MTAT is applied to the query specification of every component, where MTAT is the turn around time for the desired system. Information about other pieces of the KB is elaborated as part of Chapter 3.
5.4 Conclusion

This chapter provides a case study of the Banking Domain example, to describe the appearance of the knowledgebase for a single instance – the Banking Domain. The knowledgebase contains similar information for diverse domains such as Matrices, Insurance, Weather, etc. This chapter provides several exhibits that depict the architecture related information (includes domain specific information, architecture blue prints, use cases, QoS associations) and information related to component specifications, technologies, deployment and testing. The following chapter describes the implementation details of the prototype of a ‘Knowledgebase Tool’ created as part of this thesis.
CHAPTER 6. THE KNOWLEDGEBASE TOOL, DESIGN AND IMPLEMENTATION

Chapter 3 described the architecture of the KB in detail, and Chapter 5 provided a case study of the Banking Domain as an example to describe the contents of the KB for a specific domain instance. The KB Tool provides different views to the KB allowing specific list of available operations under each, to view and edit the properties of a domain in the KB. This chapter describes the tool in detail, by first enlisting the objectives of creating the tool, its capabilities, followed by the design and implementation details of the tools. Some screenshots of the tool are provided at the end of the chapter.

6.1 The KB Tool – Objectives and Capabilities

The KB consolidates Domain-specific, QoS, Component-related and Technological Information about a given domain. It consists of both, domain-specific and domain-independent information. For instance, the QoS catalog is common to all domains, and hence domain-independent, but the domain-specific QoS associations that enlist the QoS attributes relevant to the domain, fall under domain-specific information. The information viewable and/or editable also depends on the profile of the user who is working on the KB. For instance, the Domain Expert may want to add domain-specific information to a domain of expertise by say, updating the Domain Dictionary. The Component Developer needs to review diagrams entered by the Domain Expert, and use a UMM Specification to provide concrete a implementation t meet the specification, and deploy it. Every profile requires a different view into the KB, and allows specific operations for each profile.
As evident from the case study, there is a significant amount of information in the knowledgebase for each individual domain, and there needs to be a clear separation between the tasks allowed under different profiles. A tool with a graphical interface would thus be in place:

- To help organize and categorize the information in the KB (domain-specific and domain-independent)
- To maintain different profiles that view/update the KB in a predefined manner that is in line with the profile
- To obtain a snapshot of the complete KB for a particular domain and provide a unified view of the domain details.
- For scalability and ease of incorporation of newer mechanisms, such as editing feature diagrams, adding new matching algorithms, etc., this tool will facilitate a continuous updation of the KB.

The following sections explain the KB Tool first at a design level (Section 6.2), and then the implementation details are provided in Section 6.3, followed by a few screenshots obtained from the KB Tool Prototype.

6.2 The KB Tool Prototype

The steps involved in the design of the KB Tool are as follows: The KB Tool as mentioned above consists of different profiles of users, each profile having a set of permissible operations associated with it. The first step was to determine these profiles, so that the specific tasks under each profile can then be drawn out, and then design would be based on the profile-task list combination. Figure 6.1 shows these profiles highlighting their interrelationships. A description of the profiles is given following the Figure 6.1 which provides an overview of the profile and the exact tasks carried out by the users falling under the profiles as per the design of the KB Tool Prototype.
The following distinct profiles were identified as part of this exercise. Following is a brief description of the profiles:

1. **Domain Expert:**

   Domain Experts are analysts, application engineers, members of a standards organizations (such as OMG), and come from the list of ‘potential information sources’ listed in Section 3.1.1.1. Domain Experts add to the architecture related information in the domain of expertise, at the level of granularity described in the KB architecture. This information is used by the System Integrator, Component Developer and Component Tester for System Construction, Component Development and Component Testing respectively. The exact tasks that the Domain Expert carries out can be listed as follows:
   
   a. Enter Domain Architecture Information such as Domain Description, Dictionary.
b. Enter Models (Feature Model, Decision Model).
c. Enter Diagrams (Use Case and Sequence Diagrams).
d. Enter UMM-Annotated UDSL (Contains the Specifications of Abstract Components).
e. Enter QoS-related details and associated rules for that domain.
f. Enter Wrapper Generation Information.
g. Save all the information listed above and associate it with a ‘Domain Name’.

2. **QoS Expert:**

   The QoS Expert enters the QoS catalog into the KB that includes information about the QoS attributes, such as metrics and evaluation procedures, and composition and decomposition rules. This information is used by the System Integrator to decompose Client query’s non-functional component, and then later to statically determine the QoS properties of the system using the composition rules. The Component Developer uses the catalog as a reference to developing QoS-aware components. The exact tasks that the QoS Expert carries out can be listed as follows:
   
a. Enter Information about every QoS attribute (as defined in the QoS Catalog detailed in Appendix A) such as General Description.
b. Enter Metrics and Evaluation Procedures (with Formulae).
c. Enter Composition and Decomposition Rules for attributes.
d. Save Attribute Information into the KB.

3. **Component Developer:**

   As described in Section 5.3 the Component Developer is responsible for using the specifications of components provided as the part of the architecture information (input by the Domain Expert), and develop components that adhere to the specification. The code developed can be tested by the Component Tester, either as per the test cases written by the Component Tester, or the ones automatically generated from the specifications. The component specifications (multilevel specifications) are used by the multilevel matching algorithm of the discovery service for computing ‘extent’ of match between
query specification and candidate component specification. The exact tasks that the Component Developer carries out can be listed as follows:

a. Select a Feature Diagram and a particular Abstract Component (Leaf Node).
b. Use the skeletal code generator to generate skeletal code with necessary instrumentation for QoS attribute measurement.
c. Handwrite the component code as per the specification by using formal techniques to ensure consistency and correctness.
d. Enter the Deployment Information associated with the Component and deploy it.
e. Make Component available to Component Tester.

4. **Component Tester:**

   The job of a Component Tester is to create test cases for components developed by the Component Developer, and run them to check for errors in the developed code and log the errors for the Component Developer’s reference. The Component Tester can also run tests that are automatically generated from specifications given in the Architecture Information. The exact tasks that the Component Tester carries out can be listed as follows:

   a. Use the test cases generated by the specifications, or create test cases.
   b. Run test cases on Components made available by the Component Developer for testing, and report errors (PASS/FAIL).

5. **System Integrator:**

   The role of the System Integrator is a crucial one, since it drives the entire process of intercepting and parsing the query submitted by the client, finding the architecture and creating a system instance as per the requirement, locate the appropriate components, validate them for QoS requirements and generate the system. The System Integrator uses the QoS rules from the QoS Catalog, and architecture-related information from the Architectures input by the Domain Expert. The System Integrator communicates with the
discovery service to search for components. The exact tasks that the System Integrator carries out can be listed as follows:

a. Intercept, interpret and parse query input by the Client – to split into functional and non-functional components.

b. Consult with the Architecture Information provided by Domain Experts to select a Feature Model.

c. Select an instance of the system based on the requirements given by making selection of optional features.

d. For abstract components selected, send a query to the discovery service and obtain the results.

e. Select a set of components that match the requirements of the desired system and validate the static QoS properties using the composition rules in the QoS Catalog (entered by the QoS Expert).

f. If the result of the composition is satisfactory for all the QoS attributes then proceed with the set of components; if not, retry with the same query, or retry with a subset of the query modified, or retry with a different architecture altogether, and repeat till satisfactory results are not obtained.

g. For the components selected, review the technological heterogeneity and order appropriate adaptors present in the repository. The wrappers can also be searched for in the same repository, or if a mechanism to generate wrappers is recorded in the Architecture Information, then wrappers can be generated.

h. Construct the system based on the interactions specified in the Architecture Information using the component set and the adaptors-wrappers.

i. Dynamically validate the QoS properties using the information generated from the advices run at various join points in the component code as well as wrappers.

j. If the QoS requirement is met with the generated system, deploy the system and return to the client; if not, disband the components, and retry as in Step (f).
6. **Client:**

The Client is the actual ‘user’ of the knowledgebase. The Client issues a query to the System Integrator in a certain format, and the remaining process is driven by the System Integrator as described above. The exact tasks that the Client carries out can be listed as follows:

a. Issue query to the System Integrator specifying the requirements in a format employed by [HUA03] (Tabular form).

b. Review the system returned by the System Integrator for functional and non-functional requirements, and reissue query if behavior is not as per desired requirements.

The prototypical system consists of 6 modules (represented as tabs in the graphical interface) each consisting of the complete functionality that the profile demands. All modules read and update the same information source that consists of all the KB Contents, such as diagrams, models, textual descriptions of UDSL, QoS associations between domains and QoS attributes, and also, component related information such as specifications and test cases.

### 6.3 Design Details of the KB Tool Prototype

This section describes the mappings between the structures defined in Chapter 3 and the actual implementation of the prototype.

The entire KB (the banking domain example) is stored in the form of a database, that contains information or pointer to the information (links to files, folders, etc.). The diagrams (feature diagrams, use case diagrams and sequence diagrams) are stored as actual diagrams on disk, and the database contains links to the diagrams in that case. The remainder of this section goes over how each piece in the KB is mapped to a concrete entity on disk.
6.3.1 Domain Information

The domain description is stored in the database in the form of tables. The structure of the table is as follows:

```plaintext
{
    Objective_1, Objective_2,
    Boundary_1, Boundary_2, Boundary_3,
    InfoSource_1, InfoSource_2, InfoSource_3, InfoSource_4
}
```

The dictionary is also stored as a “term-meaning” pair list in the database for the banking domain, as part of the prototype. The structure allows synonyms and hence “term-term” pairs are also allowed. The structure of the dictionary is:

```plaintext
{
    Term,
    Meaning,
    Term_Synonym
}
```

The feature model is stored graphically as a diagram on disk. The domain expert can use any tool at his disposal to create one or more feature diagrams for the problem domain, and their locations on disk are stored in the database.

The decision model (shown graphically using the flowchart notation in Chapter 5) is stored as a language generated by the grammar in Section 3.1.1.3.2. For the banking domain example, the decision model is stored as an XML document. The domain expert enters the choices and corresponding feature list (of optional features) and this is stored as an XML schema on disk. The structure of the decision model document is:

```xml
<if choice_of_feature="xxx">
    <include>
        <feature name="yyy"/>
    </include>
</if>
```
Both use case and sequence diagrams are stored as diagrams on disk, with a link to the diagram in the database.

The UDSL is also stored on disk as a flat file that is generated by the tool. The tool also generates individual UMM Specifications for the abstract components as separate files, so that for each component, a separate UMM Specification is available to the developer.

The list of QoS associations is stored as a separate table called QoS_Associations. The structure of the table is as follows:

\{Id, QoS_Attribute\}

The QoS catalog is stored as a database table with the same structure as that shown in Appendix A. All the information about the QoS attributes is stored in a textual manner. The composition rule is identified by the function (for e.g., MIN, MAX) and decomposition rule is identified by the operator (for e.g., <, <=) and function (for e.g., SUM, MEAN).

An important constituent of the QoS framework is the property prediction and instrumentation. The prediction for static QoS attributes is carried out with the composition rules in the database. The validation for the dynamic QoS attributes uses the advice code library, which is a java library with method definitions in it. Point cuts are stored as separate files for each QoS attribute-function call pair. The AspectJ framework [AOP00] can then be used to realize the weaving, but the information is made available as mentioned above.

Change record sets for updates to the feature diagram, as suggested by the component developer are stored as text files – one file each for every record set. The contents of this file are the language that is generated by the grammar detailed in Table 3.8. Paths to these files are stored as part of the domain information table in the database.
Component technology information is stored as tables in the database for each component, along with pointers to the unit test case scripts which are stored as text files on disk.

6.4 Implementation Details of the KB Tool Prototype

A prototype of the KB Tool has been developed as part of this thesis, so that information can be stored in the KB and used by the different profiles of users in ways defined above. The prototype is developed as a Visual C# Application on Visual Studio .NET 2005. Microsoft Office Visio 2007 is the tool used for the Domain Expert to draw the use case and sequence/collaboration diagrams, with links to the diagrams stored in the common KB Database. The KB Database is implemented using Microsoft Office Access 2003.

6.5 Screenshots of the KB Tool Prototype

This section describes some of the screenshots of the prototype that has been developed as part of this thesis. The first few screenshots show how the Domain Expert and QoS Expert can add domain information (description, dictionary, new QoS attributes). Following those, are screenshots for client view (input system specification) and view of the System Integrator wherein the actual features of the resultant are selected and the query is submitted to the discovery service. There is also a screenshot of the components returned by the discovery service from which the System Integrator can choose the components that suit the requirements the most.
Figure 6.2: Domain Expert View – Input Domain Description

This figure shows the Domain Expert’s view of the tool, through which domain description can be input in the system.
Figure 6.3: Domain Expert View – Input Domain Dictionary

In this figure, the domain expert’s view of domain dictionary is shown. The domain expert can enter terms and their meanings using this view.
Figure 6.4: Domain Expert View – Input QoS Associations

This figure shows the Domain Expert’s view where he can associate relevant QoS attributes with the problem domain.
Through this view, the QoS expert can input all the information associated with a new QoS attribute into the system.
This figure shows the client’s view of the system. Through this interface, the client can enter the specification of the desired system in terms of the features (Counters, ATM, Loan Management options, etc.). after this view, is the System Integrator’s view of
the system where he reviews the Feature Diagram for the problem domain and selects the features as per the client’s request.

Figure 6.7: System Integrator View – Select features for DCS
Figure 6.8: System Integrator View – View components returned by Discovery Service
This figure shows the result returned from the discovery service. Using this interface, the System Integrator can pick the components that match the system requirements best.

### 6.6 The Workflow

The workflow as part of the tool can be described in terms of the different activities that are carried out by the stakeholders. The Domain Expert enters the domain-specific architecture-related information in the KB through interfaces shown in Figures 6.2, 6.3 and 6.4. Information such as domain description, diagrams (use case and sequence), models (feature diagrams and decision diagrams) and the UMM-annotated UDSL is entered in the KB through the interfaces described above and this information is mapped on to database tables and files at the backend.

Similarly, the QoS expert enters information about QoS attributes into the KB using interface described in Figure 6.5. Component Developers view the feature diagram and the UMM specifications of software components embedded in the UMM-annotated UDSL in use these to provide concrete component implementations and deploy them on various component registries. Test cases are entered for testing the functionality of software components.

These activities mentioned above are on-going. When a client wants to ‘order’ a DCS, the interface shown in Figure 6.6 is used. As shown in that figure, the client can make a choice of the features in the desired terms, for instance, the client can specify the number of ATMs, indicate whether Debit Card Handing needs to be enabled, what kind of loan options are available, etc. The client can specify the desired values (or ranges of values) for the QoS attributes that are identified as ‘relevant’ to the domain – in this case, Security and Turnaround Time. After the client enters the specification of the system, the system integrator takes over by first examining the feature diagrams associated with the problem domain as shown in Figure 6.7. Based on the information entered by the client,
the system integrator makes a selection of the optional features. The constraints within features (such as the ‘requires’ constraint, wherein, if an ‘ATM’ feature is included, then the ‘Customer Validation Server’ feature also must be included) are incorporated within the design of the prototype itself. So when the system integrator selects ‘ATM’ or ‘Phone’ features, ‘Customer Validation Server’ gets selected. Also, if the system integrator selects 'Loan Management Subsystem', then the mandatory feature ‘Basic Loan Manager’ gets included as well. After having selected the list of abstract components, the queries for individual components are dispatched to a discovery service. The discovery service performs multilevel matching between the query specification and the specifications of candidate components, and returns the output in terms of matching components as shown in Figure 6.8. Following this, static QoS property prediction check is performed using the composition rules (Appendix A) and glue generation templates are looked up. This is followed by the system construction and validation of dynamic QoS properties.

6.7 Motivation

This section is dedicated toward emphasizing on some of the novel aspects of the tool. Although this tool is a prototypical development to validate, or empirically evaluate the structure of the KB, it is a starting point towards a more generic, model-driven approach to creating DCS. This tool attempts to incorporate formalism, standardization (wherever applicable), and strict adherence to specifications throughout the life cycle of DCS development starting from modeling the domain for families of systems, to specifying individual pieces and finally constructing a system that is an ensemble of the specified pieces. Although there are other tools available in the industry for application development, the issues of model-driven design/construction, adherence to specifications and formally addressing QoS in terms of specifying QoS attributes, and predicting/validating the properties of the resulting system, are not formally addressed. This tool is the first step towards using theoretically-sound concepts of model-driven
design, and construction by correctness, during the entire life cycle of the construction of DCS.

Using a tool of this nature for development of DCS would help tackle many challenges imposed by DCS, such as predictability and quality. By incorporating theoretically sound principles such as multilevel matching for software components, the pool of selected components would be as close as possible to the query specification, not only in terms of basic attributes (such as syntax contracts and other meta-data), but also in terms of more semantic information about components, such as method pre-conditions and post-conditions, etc. A choice of a certain component is based on how it fares at four different levels, and this is a comprehensive enough test for how well a component may fit in the resulting system in terms of its functionality, performance and interactions with other components. The multilevel matching algorithm is just an example of how theoretically proven concepts are applied as part of the UA. Selecting ‘correct’ components will save a lot of empirical testing effort later on after the system is fully generated. One of the most important issues, is that of heterogeneity between software components, and this is tackled by the Unified Meta-component Model Specification.

Thus, this tool can be used by different participants of the DCS development process in order to enable the UA’s goal of developing a high-quality, predictable DCS from heterogeneous software components that adhere to certain QoS requirements. The prototype was empirically verified by executing different roles as provided by the system in order to see if the set of artifacts obtained as a result of this prototype is the same as derived from the KB architecture (Chapter 3). This chapter tries to map the case study example and describe how the prototype handles the information in the KB for the banking domain. The following chapter concludes the thesis along with providing a brief summary of how this work can be further expanded and made more useful.
CHAPTER 7. CONCLUSION AND FUTURE WORK

This chapter summarizes the thesis and also provides some insight into work that can be done using this thesis as the starting point.

7.1 Conclusions

The primary goal of this thesis was to formulate the KB architecture by specifying its exact constituents, their representations and interrelationships. The constituents were identified after a detailed analysis of the requirements of the KB. This analysis was done from the viewpoints of: a) the users of the KB (other entities participating in development of DCS) and b) the actual lifecycle of development of DCS. This analysis was carried out prior to formulating the architecture of the KB. The actual constituents are described in detail in Chapter 3. Chapter 5 provides a detailed case study of the KB contents for an example domain (Banking Domain), and describes the exact artifacts that would be stored in the KB for the domain. The artifacts produced are detailed in Chapter 5.

A background objective of the thesis was to provide a concrete representation of the KB so that it can be used in all the future systems and concepts that will evolve out of the UniFrame Approach. In earlier UniFrame-related efforts, the exact nature of the KB was not explicitly stated and formally described. Every prototypical system developed as part of the UniFrame Project referenced the KB as a consolidation of all the information needed, without considering how this information is stored and how it is used by other prototypical systems. The architecture, presented in this thesis, is generic and scalable, and will definitely evolve with systems that are yet to come. For the systems that will be developed in future as part of the UniFrame Project, this KB would provide as a well
-defined center piece that contains the architecture-related, domain specific information, as well as the QoS framework and necessary system integration mechanisms.

A user-profile based KB Tool prototype was also developed as part of this thesis. As detailed in Chapter 6, this tool can be used by various ‘profiles’ of users (Domain and QoS experts, Component Developers and Testers and the System Integrator and Client) to view/use/update the KB information.

The primary contribution of this thesis is the outline of the KB proposed, in terms of its constituents and their interrelationships. As mentioned above, the notion of the KB was very vague so far, so an attempt was made as part of this thesis, to come up with a concrete structure that communicates with other entities in the UA in order to enable the complete lifecycle of the UA (Chapter 4). The architecture of the KB aims to provide a ‘unified’ view of any problem domain, in terms of all the information that is needed to represent the domain, and construct DCS that adhere to the specification of the problem domain (Chapter 3). In order to automate the process of creating the KB for a problem domain (in this case, the banking domain), a prototype was created and experimented with. Details of this prototype are included as part of Chapter 6. The tool provides convenient options to its users (domain experts, component developers, etc.) to enter/update/reference information in the KB.

7.2 Future Work

This thesis provides a framework for the generic KB that contains all the information that is needed to develop a QoS-aware DCS. The future work can be carried out in many directions, some of which are discussed below.

- Concrete Realization of the entire UA: As elaborated in Chapter 4, the KB can be made to interface with all the concrete implementations that have been produced so far as part of the UniFrame Activity. They are all implemented as standalone applications on diverse platforms on a variety of technologies. If the logical
framework described in Chapter 4 can actually be realized by integrating all the systems produced as part of UniFrame Project, that would result in a complete toolkit which would contain:

- Mechanism to store domain specific artifacts
- QoS framework that incorporates evaluation, composition and decomposition of QoS attributes
- Discovery Service that discovers heterogeneous software components deployed over a network, incorporating multilevel matching principles that results in a very precise result set. The discovery service can be enhanced in more ways than one and also monitored for performance.
- A System Generation framework that assembles components by bridging the technological and semantic heterogeneity between software components using a generic template-based approach.
- A mechanism for predicting the actual properties of the composed system.
- Advanced component management functions that generate elaborate specifications of components and also the corresponding backend code skeleton.

A toolkit that provides all the features listed above would be comprehensive and would tackle most of the important challenges posed in developing DCS.

- Implementation/Solution-time Framework: This work deals with the design-level architecture details of families of systems, but does not provide any process/mechanism to lay down solution-level implementation details. These include interaction between components in terms of actual database/application servers, middleware layers, etc. This activity of providing solution space details complements the process of problem space modeling, and for complete realization of UA, these two activities have to be closely tied up.

- Elaborate Testing Framework: This work did not investigate the in-depth testing of software components. The KB framework provides hooks for basic testing functionality, but more work needs to be done on incorporating ‘automatic test case generation’ from specifications of components. If this feature can be incorporated,
then the test cases generated can be used to rigorously test the components from functionality and correctness viewpoints.

- Seamless incorporation of QoS measurement in the process of skeletal code generation: As briefly discussed in Chapter 4, the skeletal code generator can be used by the Component Developer to generate placeholders for the component functionality to be added on. If the mechanism to incorporate measurement code for dynamic QoS properties can be embedded into the design of the skeletal code generator, then the system generator’s task of dynamic validation would be made easier, since the individual component properties would already be available using the above mechanism.

The ideas discussed above are mere guidelines to work that can be done to enhance the KB.

7.3 Summary

A KB architecture was formulated as part of this thesis. This was indicated in terms of all its constituents, such as Domain Information, QoS Catalog, Query Decomposition Rules, Multilevel Matching Algorithm, Component Implementation and Deployment Information. Each piece was formally described and interrelationships between pieces was discussed in the view of development of DCS. The proposed architecture of the KB was elaborated with the help of a case study from the Banking Domain. A logical framework for interconnections between the proposed KB architecture and other UA entities was discussed in terms of the nature of information that is passed between the KB and UA entities. This is a starting point towards the integration of all the disparate pieces developed under the UniFrame Approach. Finally, a prototype of the KB Tool was created and experimented with in order to provide a unified view of the KB for banking domain.
LIST OF REFERENCES
LIST OF REFERENCES


Last Accessed in March 2007.

Last Accessed in March 2007.


http://java.sun.com/j2ee/tutorial/

[SOA] Service Oriented Architecture.

[SUN03] Sun, C. QoS Composition and Decomposition Model in UniFrame. MS Thesis, Indiana University Purdue University Indianapolis, Indianapolis, Indiana, USA, August 2003.


APPENDICES
Appendix A: QoS Catalog for Software Components

This appendix contains the QoS Catalog for software components developed as part of the UniFrame Quality of Service Framework by [BRA02]. The appendix details the following information about QoS attributes:

- General information about the attribute i.e. description, intent, applicability and other properties (for e.g. Static/Dynamic?).
- Metrics used for the attribute in terms of a value type and range
- Evaluation Procedure and Formulae with appropriate term definitions
- Composition and Decomposition rules

Tables 1 through 16 describe sixteen QoS attributes that were analyzed in the context of distributing computing systems.
<table>
<thead>
<tr>
<th><strong>Intent:</strong></th>
<th>It is a measure of confidence that the component is free from errors.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong></td>
<td>It is defined as the probability that the component is defect free.</td>
</tr>
</tbody>
</table>
| **Motivation:** | 1. It allows an evaluation of degree of Dependability of a given component.  
2. It allows Dependability of different components to be compared.  
3. It allows for modifications to a component to increase its Dependability. |
| **Applicability:** | This model can be used in any system, which requires its components to offer a specific level of dependability. Using the model, the Dependability of a given component can be calculated before being incorporated into the system. |
| **Model Used:** | Dependability model by Jeffrey Voas based on Voas and Miller's Squeeze Play Model. |
| **Metrics used:** | Testability Score, Dependability Score |
| **Influencing Factors:** | 1. Degree of testing.  
2. Fault hiding ability of the code.  
3. The likelihood that a statement in a component is executed.  
4. The likelihood that a mutated statement will infect the component's state.  
5. The likelihood that a corrupted state will propagate and cause the component output to be mutated. |
| **Evaluation Procedure:** | 1. Perform Execution Analysis on the component  
2. Perform Propagation Analysis on the component  
3. Calculate the Testability value of the component  
4. Calculate the Dependability Score of the Component |
|---------------------------|-----------------------------------------------------------------------------------------------------|
| **Evaluation Formulae:**  | \( T = E \times P. \)  
\( T: \) Testability Score *(a prediction of the likelihood that a particular statement in a component will hide a defect during testing).*  
\( E: \) Execution Estimate *(the likely hood of executing a given fault).*  
\( P: \) Propagation Estimate *(the conditional probability of the corrupted data state corrupting the software's output after the state gets infected).*  

\( D = 1 - (1 - T)^N. \)  
\( D: \) Dependability Score.  
\( N: \) Number of successful tests. |
| **Result Type:**         | Floating Point Value between \([0,1]\) |
| **Static / Dynamic:**    | Static |
| **Universal / Subset / Existential / Specific:** | Universal |
| **Application dependent Independent:** | Independent |
| **Decomposition rule:**  | \( D_i \geq D, \) where \( D_i \) is the dependability measure of component \( i \) (\( i=1, 2, \ldots, n \)), \( D \) is the dependability measure of system |
| **Composition rule:**    | \( D = \text{minimum} (D_i), \) where \( D \) is the dependability measure of system, \( D_i \) is the dependability measure of component \( i \) (\( i=1, 2, \ldots, n \)). |
| **Consequence:**         | 1. Greater amounts of testing and greater Testability scores result |
2. Lower amounts of testing and lower Testability scores result in lesser Dependability.

3. Doing additional testing and / or increasing the software's Testability by adding assertions can improve a poor score.

4. Lesser amount of testing is required to provide a fixed dependability score for higher Testability Scores.

5. Additional testing can improve a poor dependability score.

<table>
<thead>
<tr>
<th>Related Parameters:</th>
<th>Availability, Error Rate, Stability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain of Usage:</td>
<td>Domain Independent</td>
</tr>
<tr>
<td>Error Situation:</td>
<td>Low dependability results in:</td>
</tr>
<tr>
<td></td>
<td>1. Unreliable component behavior.</td>
</tr>
<tr>
<td></td>
<td>2. Improper execution / termination.</td>
</tr>
<tr>
<td></td>
<td>3. Erroneous results.</td>
</tr>
<tr>
<td>Aliases:</td>
<td>Maturity, Fault Hiding Ability, Degree of Testing</td>
</tr>
<tr>
<td>Resources:</td>
<td></td>
</tr>
<tr>
<td>• Links</td>
<td>Jeffrey Voas: <a href="http://www.cigital.com/services/reliability.html">http://www.cigital.com/services/reliability.html</a></td>
</tr>
<tr>
<td></td>
<td>Dick Hamlet: Testing for Software Quality</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.cs.pdx.edu/~hamlet/testres.html">http://www.cs.pdx.edu/~hamlet/testres.html</a></td>
</tr>
<tr>
<td>• Papers</td>
<td>Dependability Certification of Software Components: J. Voas and J. Payne</td>
</tr>
<tr>
<td></td>
<td>An Approach to Certifying Off-the-Shelf Software Components: J.</td>
</tr>
</tbody>
</table>
Table A.2 Attribute: Security

| Intent: | It is a measure of the ability of the component to resist an intrusion |
| Description: | Minimum-Time-To-Intrusion (MinTTI): It is the shortest predicted period of time before any intrusion. Mean-Time-To-Intrusion (MTTI): It is the average time interval before an intrusion will occur. |
| Motivation: | This attribute reflects the component’s security weaknesses in the face of known and unknown threats that may occur in future. Hence they can be used to better fortify the security defenses of the system. Also, it makes possible an objective comparison of effectiveness of security strategies, allowing the user to pick the strategy most effective for his system. |
| Applicability: | These measures can be used to verify the ability of a component to |
resist intrusion before being incorporated into the system. Hence, they can be used in any system that specifies some security requirements for its components.

<table>
<thead>
<tr>
<th><strong>Model Used:</strong></th>
<th><strong>Adaptive Vulnerability Analysis (AVA)</strong> by Jeffery Voas based on Extended Propagation Analysis.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Metrics Used:</strong></td>
<td><strong>Minimum-Time-To-Intrusion (MinTTI), Mean-Time-To-Intrusion (MTTI)</strong></td>
</tr>
</tbody>
</table>
| **Influencing Factors:** | 1. The likelihood that a location in a component is executed.  
2. The likelihood that a mutated location will infect the component’s state.  
3. The likelihood that a corrupted state will propagate and cause the component output to be mutated.  
4. The likelihood that the output event produced satisfies the definition of what constitutes a security violation  
5. The input cases in the usage probability distribution.  
6. The classes of fault injections that are used  
7. The classes of intrusions that are defined. |
| **Evaluation Procedure:** | 1. Determine the location where the fault injection should occur.  
2. Select the test cases upon which the program will run. Preferably those that are most likely to trigger a successful intrusion, like: rare events, known input sequences that are unusual or likely to be threatening, totally random events or even the operational profile of the system.  
3. Perform the program state corruption or syntactic mutation of the code. This is the Fault Injection Step. |
4. Determine if the fault injected caused the program to produce an output that satisfies the users’ definition of what constitutes a security violation.

5. Calculate the Vulnerability Assessment ‘V’ for each location L in the program.

6. Calculate the Execution Probability ‘E’ for each location L in the program.

7. Using the Vulnerability Assessment and Execution Probabilities calculate the MTTI and MinTTI.

**Evaluation Formulae:**

\[ \text{MTTI} = \left( \sum_{l=1}^{m} [V_{alPQ} \times E_{lPQ}] \times (\text{program executions}) / (\text{unit of time} \times M) \right)^{-1} \]

\[ \text{MinTTI} = \left[ \max_{l} [V_{alPQ} \times E_{lPQ}] \times (\text{program executions}) / (\text{unit of time} \times M) \right]^{-1} \]

Where,

- \( Q \): Normal usage probability distribution of all inputs to the system
- \( V_{alPQ} \): The Vulnerability Assessment for line \( l \) for program P and Q
- \( E_{lPQ} \): The Execution Probability of line \( l \) for program P and Q

**Result Type:** Floating Point Value in the chosen unit of time

**Static / Dynamic:** Static

**Universal / Subset / Existential / Specific:** Universal

**Application dependent Independent:** Independent

**Decomposition rule:** \( \text{MTTI}_i \geq \text{MTTI} \), where MTTI\(_i\) is the security measure of component i (i=1, 2, ..., n), MTTI is the security measure of system.
**Composition rule:**
\[ \text{MTTI} = \min(\text{MTTI}_1, \text{MTTI}_2, \ldots, \text{MTTI}_n) \]
Where MTTI is the system security value, MTTI\(_i\) (i=1, 2, …, n) is the security value of component \(i\).

**Consequence:**
1. The larger the values of MinTTI and MTTI, the more secure the system.
2. The smaller the values of MinTTI and MTTI, the less secure the system.
3. The process can be modified to address new threats by adding new threat classes.
4. The results obtained are only with respect to those threat classes being simulated. A new threat class with a lesser MinTTI than any previous threat will result in an overestimated level of confidence provided by the metric.
5. The process can be modified for application specific intrusions by adding application specific threat classes.
6. The conditions for what constitutes a reasonable or unreasonable output event should be clearly defined.

**Related Parameters:**
Reliability, Stability, Adaptability

**Domain of Usage:**
E-Commerce, C4I

**Error Situation:**
Low Security results in:
1. Leakage: The acquisition of classified information by unauthorized individuals
2. Tampering: The unauthorized alteration of information
3. Vandalism: Interference with the operation of a system without gain to the perpetrator.
4. Compromised defenses against intrusions
5. Vulnerability of the system to malicious external agents
<table>
<thead>
<tr>
<th><strong>Aliases:</strong></th>
<th>Vulnerability, Protection, Authentication, Intrusion Detection, Threat Handling</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Resources:</strong></td>
<td></td>
</tr>
<tr>
<td>• Links</td>
<td>Cigital: <a href="http://www.cigital.com/services/security.html">http://www.cigital.com/services/security.html</a></td>
</tr>
</tbody>
</table>
| • Papers | Defining an Adaptive Software Security Metric from a Dynamic Software Failure-tolerance Measure  
Testing Software for Characteristics Other than Correctness: Safety, Failure-tolerance, and Security  
J. Voas |
| • Books | Software Fault Injection: Inoculating Programs Against Errors  
Jeffrey Voas and Gary McGraw  
Security and Privacy for E-Business  
Anup Ghosh  
E-Commerce Security: Weakest links, Best defenses  
Anup K. Ghosh |

Table A.3 Attribute: Adaptability

<table>
<thead>
<tr>
<th><strong>Intent:</strong></th>
<th>It is a measure of the ability of the component to tolerate changes in resources and user requirements.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong></td>
<td>It is the extent to which a software system adapts to change in its execution environment without any external intervention.</td>
</tr>
<tr>
<td><strong>Motivation:</strong></td>
<td>1. It allows adaptability of different components to be compared</td>
</tr>
</tbody>
</table>
2. It allows an evaluation of degree of adaptability of a component
3. It allows for modifications to a component to increase it’s adaptability.

**Applicability:**
This measure can be used whenever a given component being incorporated into a system should be able to tolerate changes in the environment.

**Model Used:**
Software Adaptability Model by Nary Subramanian and Lawrence Chung

**Metrics Used:**
Element Adaptability Index (EAI)
Architecture Adaptability Index (AAI)
Software Adaptability Index (SAI)

**Influencing Factors:**
1. Number of components in the system architecture.
2. Number of connectors in the system architecture.
3. Whether the system components and connectors are adaptable or not.

**Evaluation Procedure:**
1. Define Adaptability in accordance with the application.
2. Define the Element Adaptability Index (EAI) for all elements (components and connectors) in the system.
   
   \[ \text{EAI} = 1 \text{ (for adaptable element)} \]
   \[ \text{EAI} = 0 \text{ (for non-adaptable element)} \]
3. Determine the Architecture Adaptability Index (AAI) for the
4. Determine the Software Adaptability Index (SAI) for the system.
5. Repeat steps 2 to 4 for all architectures of the system.

| Evaluation Formulae: | Total EAI = \( \sum \) EAI for each dimension of adaptability for the element / Total number of dimensions
Where, Dimension is an attribute external to the software system that impacts on the system.
AAI= EAI for all elements of architecture / Total number of elements
SAI= AAI for all architectures of the software / Total number of Architectures of the Software |

| Result Type: | Floating Point Value between 0 to 1 |
| Static / Dynamic: | Static |
| Universal / Subset / Existential / Specific: | Universal |
| Application dependent / Independent: | Independent |
| Decomposition rule: | SAI\(_i\) \(\geq\) SAI, where SAI\(_i\) is the adaptability measure of component \(i (i=1, 2, ..., n)\), SAI is the adaptability measure of system. |
| Composition rule: | SAI=\(\min (SAI_1, SAI_2, ..., SAI_n)\) Where: SAI is the system adaptability, SAI\(_i\) \((i=1, 2, ..., n)\) is the adaptability of component \(i\). |
| Consequence: | 1. Larger the value of SAI, the more adaptable the system |
2. Smaller the value of SAI, the less adaptable the system
3. The adaptability can be improved by increasing the number of Adaptable components or connectors
4. A large number of non-adaptable elements can lead to a low adaptability for the system.

<table>
<thead>
<tr>
<th>Related Parameters:</th>
<th>Evolvability, Stability, Dependability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain of Usage:</td>
<td>C4I</td>
</tr>
<tr>
<td>Error Situation:</td>
<td>A low Adaptability results in:</td>
</tr>
<tr>
<td></td>
<td>1. System malfunction in the event of a change in execution environment</td>
</tr>
<tr>
<td></td>
<td>2. The need for user intervention in case of any change in resources or user requirements.</td>
</tr>
<tr>
<td></td>
<td>3. A unreliable system in the event of a change in normal execution environment.</td>
</tr>
<tr>
<td>Aliases:</td>
<td>Adaptivity, Tolerability, Robustness</td>
</tr>
<tr>
<td>Resources:</td>
<td></td>
</tr>
<tr>
<td>• Links</td>
<td>Lawrence Chung:</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.utdallas.edu/~chung/adaptability.html">http://www.utdallas.edu/~chung/adaptability.html</a></td>
</tr>
<tr>
<td></td>
<td>Design patterns for Adaptable software:</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.utdallas.edu/~nskamat/patterns/">http://www.utdallas.edu/~nskamat/patterns/</a></td>
</tr>
<tr>
<td>• Papers</td>
<td>Metrics for Software Adaptability,</td>
</tr>
<tr>
<td></td>
<td>N. Subramanian and L. Chung,</td>
</tr>
<tr>
<td></td>
<td>Process-Oriented Metrics for Software Architecture Adaptability</td>
</tr>
<tr>
<td></td>
<td>N. Subramanian and L. Chung,</td>
</tr>
</tbody>
</table>
### Table A.4 Attribute: Maintainability

<table>
<thead>
<tr>
<th><strong>Intent:</strong></th>
<th>It is a measure of the ease with which a software system can be maintained.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong></td>
<td>It is defined as the ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment</td>
</tr>
</tbody>
</table>
| **Motivation:** | 1. Using the definition of maintainability, the client can incorporate quantitative maintainability requirements in the software development contracts it awards to component developers.  
2. Using the methods of estimating maintainability, the client can verify whether the component meets the maintainability requirements.  
3. Using the methods of estimating maintainability at design time, the component developers can monitor the expected maintainability of the component and adjust it as needed.  
4. Using the methods of estimating maintainability, the client can make specific quantitative planning of maintenance activities and determine whether a problem is product related or process related.  
5. Using the methods of monitoring maintainability, the client can incorporate maintainability into it’s organization-wide standards thus ensuring graceful ageing of it’s applications |
| **Applicability:** | The measure can be used whenever the client needs a quantitative estimate of the amount of effort required for performing periodic maintenance activities on a component. |
| **Model Used:** | Maintainability Model by R. Cheaito, M. Frappier, S. Matwin, A. Mili D. Crabtree |
| **Influencing Factors:** | **CEIS:** Contract End Item Specification.  
**RS:** Requirement Specification.  
**SPDD:** Software Preliminary Design Document.  
**SDDD:** Software Detailed Design Document.  
**SC:** Source Code.  
Unreferenced Requirements: The number of requirements not referenced by a lower document in the documentation hierarchy. It measures the traceability of the documentation. Traceability is an important aspect of maintainability. It helps in determining the requirements applying to a module or the modules associated to a requirement. The metric applies to CEIS, RS, SPDD and SDDD.  
Non-Referencing Items: The number of items not referencing a requirement in an upper document of the documentation hierarchy. It is the dual of the previous measure.  
COCOMO Product Cost Factors: The combination of the COCOMO product cost factors **RELY** (the required reliability), **DATA** (the database size, to the extent that it is meaningful) and **CPLX** (the product complexity). |
Module Coupling: This measures the flow of information between modules. Only the nature of the coupling between module is addressed, not the quantity of information. Maintainability should decrease as coupling increases. This measure is computed on SDDD or source code.

Module Cohesion: This measures the flow of information within a module. Maintainability is expected to decrease as cohesion increases. This measure is computed on SDDD or source code.

Design Complexity: A measure of intermodule and intramodule complexity of a system based on fan-out and global variables. This measure is computed on SDDD or source code.

Cyclomatic Complexity Number: This well-known measure represents the number of independent basic paths (covering all segments of code) in a module. The more basic paths a module has, the more difficult it is to comprehend and test.

Knots: A measure of the number of crossing lines (unstructured goto statements) in a control flow graph. There is found to be a significant correlation between this measure and the effort required to change a module.

Comments Volume of Declarations: The total number of characters found in the comments of the declaration section of a module. The declaration section comprises comments before the module heading up to the first executable statement of the module body. Several studies have shown that comments affect comprehensibility and
<table>
<thead>
<tr>
<th><strong>Readability of a Module.</strong></th>
</tr>
</thead>
</table>

**Comments Volume of Structures:** The total number of characters in the comments found anywhere in the module except in the declaration section.

**Average Length of Variable Names:** This is the mean number of characters of all variables used in a module. Unused declared variables are not included.

**Lines of Code:** The number of lines in the source code of a module excluding blank lines or comment lines. This well-known metric has been the subject of many criticisms when used as a measure of productivity. However, it is a good indicator of the effort required to change a module.

**Documentation Accuracy Ratio:** A verification of the accuracy of the CEIS, RS, SPDD and SDDD with respect to the source code. A sample of modules is randomly selected. For each module, an expert verifies that the requirements applicable to the module are those implemented in the source code. Weights are associated to various components of a specification like inputs, outputs, validations and algorithms. The average sum of the weights is computed for the set of modules selected. Documentation accuracy is a key aspect of maintainability, especially when maintenance must be done over a long period of time with a high turnover of maintenance personnel.

**Consistency:** This measures the extent to which the documentation and source code contains uniform notation, terminology and
symbology within itself. It is computed by counting, for a sample of programs and documents, the number of inconsistencies like, e.g., using the same name for different entities, using different names for the same entity or using different structure of comments for program headings. Consistency should improve the understandability of documents and programs.

**Evaluation Procedure:**

1. Determine the functionality of the component
2. Determine the average intrinsic effort it takes to carry out a modification of the software system.
3. Determine the maintainability of the component

**Evaluation Formulae:**

\[ MNT = \frac{FNC}{COST_{MODIF}} \]

Where,

- **MNT**: Maintainability of the component
- **FNC**: measure of functionality of component
- **COST_{MODIF}**: Average intrinsic effort it takes to carry out a modification of the software system.

\[ COST_{MODIF} = (1.15)^{AVG-3} \times \alpha \times (N/NS)^{\beta} \]

Where,

- \( \alpha = 6.32 \)
- \( \beta = -0.79 \)
- **N**: measure of size of the component
- **NS**: Average size of a subprogram
- **AVG**: Average Cyclomatic Complexity of procedures and functions

**Result Type:**

- If FNC is measured in function points; then maintainability is measured in function points per man month.
- If FNC is measured in lines of code; then maintainability is measured in lines of code per man month.
If FNC is measured by the intrinsic effort required to develop the software product; then maintainability is measured as the ratio of man months over man months.

<table>
<thead>
<tr>
<th>Static / Dynamic:</th>
<th>Static</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universal / Subset / Existential / Specific:</td>
<td>Universal</td>
</tr>
<tr>
<td>Application dependent / Independent:</td>
<td>Independent</td>
</tr>
</tbody>
</table>

**Decomposition rule:**
\[ MNT_i \geq MNT, \] where \( MNT_i \) is the maintainability measure of component \( i \) \((i=1, 2, ..., n)\), \( MNT \) is the maintainability measure of system.

**Composition rule:**
\[ MNT = W_1 \cdot MNT_1 + W_2 \cdot MNT_2 + ... + W_n \cdot MNT_n, \] where \( MNT \) is the maintainability measure of system, \( MNT_i \) is the maintainability measure of component \( i \) \((i=1, 2, ..., n)\), \( W_i = \text{LOC}_i / (\text{LOC}_1 + \text{LOC}_2 + ... + \text{LOC}_n) \), LOC denotes Line Of Code.

**Consequence:**
1. The larger the maintainability, fewer the man months required to carry out a typical modification of a software product with a given functionality.
2. The smaller the maintainability, greater the man months required to carry out a typical modification of a software product with a given functionality.
3. Higher the Average Cyclomatic Complexity of the component the; greater the man months required to carry out a typical modification of a software product with a given functionality.
4. Higher the \( \text{COST}_{\text{MODIF}} \) of the component, the greater the maintainability.
5. Lower the \( \text{COST}_{\text{MODIF}} \) of the component, the greater the
<table>
<thead>
<tr>
<th><strong>maintainability.</strong></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Related Parameters:</strong></th>
<th>Testability, Understandability, Modifiability, Portability</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Domain of Usage:</strong></td>
<td>Domain Independent</td>
</tr>
<tr>
<td><strong>Error Situation:</strong></td>
<td>A low maintainability results in:</td>
</tr>
<tr>
<td></td>
<td>1. A large amount of effort to carry out a typical modification of a software product with a given functionality.</td>
</tr>
<tr>
<td></td>
<td>2. A code that is difficult to understand, modify and migrate.</td>
</tr>
<tr>
<td></td>
<td>3. A prohibitive cost of maintenance over the lifetime of the system.</td>
</tr>
<tr>
<td><strong>Aliases:</strong></td>
<td>Maintenance cost</td>
</tr>
</tbody>
</table>

**Resources:**

- **Links**
  - Marc Frappier:
    - http://www.dmi.usher.ca/~frappier/papers.html

- **Papers**
Table A.5 Attribute: Portability

<table>
<thead>
<tr>
<th>Intent:</th>
<th>It is a measure of the ease with which a component can be migrated to a new environment.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>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 portable across a class of environments to the degree that the cost to transport and adapt it to a new environment in the class is less than the cost of redevelopment.</td>
</tr>
</tbody>
</table>
| Motivation: | 1. It helps to characterize the costs and benefits of incorporating portability in a software design.  
2. It helps to perform a meaningful comparison between the costs and benefits of portability-based methods and other methods.  
3. It helps to estimate the effort or cost of porting a given component to a specified environment. |
| Applicability: | It can be used in any system, which specifies a bound on the amount of effort required to port the component on to a specified environment. |
| Model Used: | Portability Model by James Mooney |
| Metrics Used: | 1. Degree of Portability (DP)  
2. Cost of Porting ($C_{port}$)  
3. Cost of Redevelopment ($C_{dev}$) |
| Influencing Factors: | 1. Identification of minimum necessary set of environmental |
requirements and assumptions.
2. Elimination of all unnecessary assumptions throughout the design
3. Identification of specific environment interfaces required (procedure calls, parameters, data structures etc)
4. Encapsulation of interface completely in a suitable module, package, object etc
5. Identification of a suitable standard for the interface, which is expected to be available in most target environments

| Evaluation Procedure: | 1. Calculate the cost of Porting (DP) |
| | 2. Calculate the cost of Redevelopment ($C_{\text{port}}$) |
| | 3. Calculate the degree of Portability ($C_{\text{rdev}}$) |

<p>| Evaluation Formulae: | $DP(su) = 1 - (C_{\text{port}}(su,e2)) / (C_{\text{rdev}}(\text{req},e2))$ |
| | Where, |
| | $(C_{\text{port}})$: cost of Porting |
| | $(C_{\text{rdev}})$: cost of Redevelopment |
| | Su: A software unit |
| | e2: new environment |
| | req : requirement specification |
| | $C_{\text{port}}(su,e2)) = C_{\text{mod}}(su,e2) + C_{\text{ptd}}(\text{req},e2) + C_{\text{pdoc}}(\text{req},e2)$ |
| | Where, |
| | $C_{\text{mod}}$: Cost of modification |
| | $C_{\text{ptd}}$: Cost of portable testing and debugging |
| | $C_{\text{pdoc}}$: Cost of portable documentation |
| | $C_{\text{rdev}} (\text{req},e2) = C_{\text{rdes}} (\text{req}) + C_{\text{rcod}} (\text{req},e2) + C_{\text{rtd}} (\text{req},e2) + C_{\text{rdoc}}(\text{req},e2)$ |
| | Where, |</p>
<table>
<thead>
<tr>
<th>Relationship</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{rdes}$</td>
<td>Cost of redeveloping design</td>
</tr>
<tr>
<td>$C_{rcod}$</td>
<td>Cost of redeveloping code</td>
</tr>
<tr>
<td>$C_{rtd}$</td>
<td>Cost of testing and debugging redeveloped code</td>
</tr>
<tr>
<td>$C_{rdoc}$</td>
<td>Cost of redeveloping documentation</td>
</tr>
</tbody>
</table>

**Result Type:** Floating point value between 0 and 1

**Static / Dynamic:** Static

**Universal / Subset / Existential / Specific:** Universal

**Application dependent / Independent:** Independent

**Decomposition rule:** $DP_i \geq DP$, where $DP_i$ is the portability measure of component $i$ ($i=1, 2, \ldots, n$), $DP$ is the portability measure of the system.

**Composition rule:** $DP = W_1*DP_1 + W_2*DP_2 + \ldots + W_n*DP_n$, where $DP$ is the portability measure of the system, $DP_i$ is the portability measure of component $i$ ($i=1, 2, \ldots, n$), $W_i=1/n$.

**Consequence:**
1. Higher the degree of portability, the lower the cost of porting compared to cost of redevelopment
2. Lower the degree of portability, the higher the cost of porting compared to cost of redevelopment
3. Portability is cost effective if and only if degree of portability is greater than 0.

**Related Parameters:** Adaptability, Evolvability, Maintainability

**Domain of Usage:** Domain Independent
Error Situation: A low degree of portability results in:
1. Higher cost of porting the component onto a new environment.
2. Higher complexity of the porting procedure.
3. Lower cost of redevelopment compared to porting.

Aliases: Interoperability, compatibility

Resources:
- Links
  Software Portability Homepage: http://www.csee.wvu.edu/~jdm/research/portability/home.html
- Papers
  - Issues in the Specification and Measurement of Software Portability
  - Bringing Portability to the Software Process
  - Developing Portable Software
- Books
  - James Mooney

<table>
<thead>
<tr>
<th>Table A.6 Attribute: Parallelism Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intent:</strong> It is used to determine whether a component can accept asynchronous calling from other components.</td>
</tr>
<tr>
<td><strong>Description:</strong> It is an indication of whether a component supports Synchronous or asynchronous calling.</td>
</tr>
<tr>
<td><strong>Motivation:</strong></td>
</tr>
<tr>
<td><strong>Applicability:</strong></td>
</tr>
<tr>
<td><strong>Model Used:</strong></td>
</tr>
</tbody>
</table>
| **Influencing Factors:** | 1. Algorithm used by the component  
2. Provision for multithreading |
| **Evaluation Procedure:** | Empirical |
| **Evaluation Formulae:** | -NA- |
| **Result Type:** | Binary value  
(Yes – Asynchronous  
No – Synchronous) |
<p>| <strong>Static / Dynamic:</strong> | Static |
| <strong>Universal / Subset / Existential / Specific:</strong> | Universal |
| <strong>Application dependent</strong> | Independent |</p>
<table>
<thead>
<tr>
<th>/ Independent:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Decomposition rule:</strong></td>
<td>( PC_i = PC ), where ( PC_i ) is the measure of parallelism constraints of component ( i ) ((i=1, 2, ..., n)), ( PC ) is the measure of parallelism constraints of system</td>
</tr>
<tr>
<td><strong>Composition rule:</strong></td>
<td>( PC = PC_1 &amp; PC_2 &amp; ... &amp; PC_n ), where ( PC ) is the measure of parallelism constraints of system, ( PC_i ) is the measure of parallelism constraints of component ( i ) ((i=1, 2, ..., n)).</td>
</tr>
</tbody>
</table>
| **Consequence:** | 1. If the component is asynchronous then, other components can call the component in parallel without having to wait for the result to return. 
2. If the component is synchronous then, the component can accept only one call at a time i.e. other components have to wait for the previous call to return before calling. 
3. If a component is asynchronous then, the response time of the component is low 
4. If a component is synchronous then, the response time of the component is high |
| **Related Parameters:** | Capacity, Ordering Constraint, Priority |
| **Domain of Usage:** | Domain independent |
| **Error Situation:** | If a component is synchronous then, the response time of the component can be higher than the response time for an asynchronous component. |
| **Aliases:** | -NA- |
Table A.7 Attribute: Ordering Constraints

<table>
<thead>
<tr>
<th><strong>Intent:</strong></th>
<th>It refers to the ordering of the results returned by a component</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong></td>
<td>It indicates whether the results returned by a component are in the proper order.</td>
</tr>
<tr>
<td><strong>Motivation:</strong></td>
<td>In some applications the order of the results returned can affect the behavior of the application. Hence the order of the results plays an important role in the proper functioning of the application.</td>
</tr>
<tr>
<td><strong>Applicability:</strong></td>
<td>This attribute can be used in any system, which requires its components to return the results in the proper order. Using this attribute, we can determine whether a component returns its results in the right order before incorporating it into the system.</td>
</tr>
<tr>
<td><strong>Model Used:</strong></td>
<td>-NA-</td>
</tr>
</tbody>
</table>
| **Influencing Factors:** | 1. Reliability of the network transmission  
2. The ordering behavior of the protocol used on the network |
| **Evaluation Procedure:** | Empirical |
| **Evaluation Formulae:** | -NA- |
| **Result Type:** | Binary value  
(Yes – ordered results  
No – unordered results) |
<table>
<thead>
<tr>
<th>Static / Dynamic:</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universal / Subset / Existential / Specific:</td>
<td>Universal</td>
</tr>
<tr>
<td>Application dependent / Independent</td>
<td>Independent</td>
</tr>
<tr>
<td>Decomposition rule:</td>
<td>$\text{OC}_i = \text{OC}$, where $\text{OC}_i$ is the measure of ordering constraints of component $i$ ($i=1, 2, ..., n$), $\text{OC}$ is the measure of ordering constraints of system</td>
</tr>
<tr>
<td>Composition rule:</td>
<td>$\text{OC} = \text{OC}_1 &amp; \text{OC}_2 &amp; ... &amp; \text{OC}_n$, where $\text{OC}$ is the measure of ordering constraints of system, $\text{OC}_i$ is the measure of ordering constraints of component $i$ ($i=1, 2, ..., n$).</td>
</tr>
<tr>
<td>Consequence:</td>
<td>The ordered delivery of results can be achieved by incorporating ordering mechanisms in the network protocol. This may lead to longer response time due to extra processing and extra transmissions over the network.</td>
</tr>
<tr>
<td>Related Parameters:</td>
<td>Transmission, Error Rate, Parallelism constraints</td>
</tr>
<tr>
<td>Domain of Usage:</td>
<td>Domain Independent</td>
</tr>
<tr>
<td>Error Situation:</td>
<td>If the ordering constraints are violated, the result will be out of order and lead to malfunctioning of the application.</td>
</tr>
<tr>
<td>Aliases:</td>
<td>Out-of-order</td>
</tr>
</tbody>
</table>
Table A.8 Attribute: Priority

<table>
<thead>
<tr>
<th>Intent:</th>
<th>It refers to the ability to utilize priorities to serve clients in a multiple client scenario.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>It is an indication of whether the component can execute client requests based on their priority.</td>
</tr>
<tr>
<td>Motivation:</td>
<td>In case a component is serving multiple clients in unison, there is a need to utilize priorities to serve the client request. Otherwise there might be a deadlock or even a system crash.</td>
</tr>
<tr>
<td>Applicability:</td>
<td>This model can be used in any system, which requires its components to utilize priorities to serve clients. Using this attribute, we can determine whether a component utilizes priorities to serve clients before incorporating it into the system.</td>
</tr>
<tr>
<td>Model Used:</td>
<td>-NA-</td>
</tr>
<tr>
<td>Influencing Factors:</td>
<td>Algorithm used in the component</td>
</tr>
<tr>
<td>Evaluation Procedure:</td>
<td>Empirical</td>
</tr>
<tr>
<td>Evaluation Formulae:</td>
<td>-NA-</td>
</tr>
</tbody>
</table>
| Result Type: | Binary value  
(Yes – uses priority  
No – does not use priority) |
<table>
<thead>
<tr>
<th>Static / Dynamic:</th>
<th>Static</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universal / Subset / Existential / Specific:</td>
<td>Universal</td>
</tr>
<tr>
<td>Application dependent / Independent:</td>
<td>Independent</td>
</tr>
<tr>
<td>Decomposition rule:</td>
<td>$\text{PRI}_i = \text{PRI}$, where $\text{PRI}_i$ is the priority measure of component $i$ ($i = 1, 2, ..., n$), $\text{PRI}$ is the priority measure of system</td>
</tr>
<tr>
<td>Composition rule:</td>
<td>$\text{PRI} = \text{PRI}_1 &amp; \text{PRI}_2 &amp; ... &amp; \text{PRI}_n$, where $\text{PRI}$ is the priority measure of system, $\text{PRI}_i$ is the priority measure of component $i$ ($i = 1, 2, ..., n$).</td>
</tr>
</tbody>
</table>
| Consequence: | 1. Higher the priority of the client the faster the response.  
2. Lower the priority of the client the slower the response |
| Related Parameters: | Parallelism constraints |
| Domain of Usage: | E-commerce, C4I |
| Error Situation: | In a multiple client scenario, the absence of a priority scheme can lead to longer response times for critical clients. It can also lead to a deadlock or even a system crash. |
| Aliases: | Privilege |

Table A.9 Attribute: Throughput

| Intent: | It indicates the speed or efficiency of the component. |
| Description: | It is a measure of the number of methods or requests the component |
Motivation: Some applications like DBMS applications or web server applications require components that can serve a given number of requests per unit time. This ensures that the clients get a fast response.

Applicability: This attribute can be used in any system, which requires its components to serve a given number of requests per unit time. Using this attribute, we can determine whether a component serve the specified number of requests per unit time before incorporating it into the system.

Model Used: Empirical

Metric used: Number of responses per second

Influencing Factors:
1. Algorithm used by the methods
2. Speed of the CPU
3. Available Memory
4. Other hardware like: I/O devices, system bus etc

Evaluation Procedure: Empirical

Evaluation Formulae: Throughput = NR / NT
Where,
NR: Number of responses
NT: Total time taken

Result Type: Floating point value
<table>
<thead>
<tr>
<th>Static / Dynamic:</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universal / Subset / Existential / Specific:</td>
<td>Universal</td>
</tr>
<tr>
<td>Application dependent / Independent:</td>
<td>Dependent</td>
</tr>
</tbody>
</table>
| Decomposition rule:       | *Parallel:* $TP_i >= TP/n$, where $TP_i$ is the throughput measure of component $i$ ($i=1, 2, ..., n$), $TP$ is the throughput measure of system.  
*Sequential:* $TP_i >= TP$, where $TP_i$ is the throughput measure of component $i$ ($i=1, 2, ..., n$), $TP$ is the throughput measure of system. |
| Composition rule:         | *Parallel:* $TP = TP_1 + TP_2 + ... + TP_n$, where $TP$ is the throughput measure of system, $TP_i$ is the throughput measure of component $i$ ($i=1, 2, ..., n$).  
*Sequential:* $TP = \text{minimum} (TP_i)$, where $TP$ is the throughput measure of system, $TP_i$ is the throughput measure of component $i$ ($i=1, 2, ..., n$). |
| Consequence:              | 1. Higher the throughput of the component the lesser the response time  
2. Lower the throughput of the component the higher the response time |
| Related Parameters:       | Capacity, Parallelism Constraints, End-to-End Delay |
| Domain of Usage:          | Domain Independent |
| Error Situation:          | A low value of throughput results in Longer response time |
| Aliases:                  | Execution Rate |
Table A.10 Attribute: Capacity

<table>
<thead>
<tr>
<th>Intent:</th>
<th>It indicates the maximum number of concurrent requests the component can serve</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>It is a measure of the number of concurrent requests the component can serve at a given time instant.</td>
</tr>
<tr>
<td>Motivation:</td>
<td>Some applications like DBMS applications or web server applications require components that can serve a given number of concurrent requests at any given time. This ensures that multiple clients get a fast response.</td>
</tr>
<tr>
<td>Applicability:</td>
<td>This attribute can be used in any system, which requires its components to serve a given number of concurrent requests at a given time. Using this attribute, we can determine whether a component serve the specified number of concurrent requests at a given time before incorporating it into the system.</td>
</tr>
<tr>
<td>Model Used:</td>
<td>Empirical</td>
</tr>
<tr>
<td>Metric used:</td>
<td>Number of concurrent requests per unit time</td>
</tr>
<tr>
<td>Influencing Factors:</td>
<td>1. Implementation details like multithread mechanism</td>
</tr>
<tr>
<td></td>
<td>2. Speed of CPU</td>
</tr>
<tr>
<td></td>
<td>3. Available memory</td>
</tr>
<tr>
<td></td>
<td>4. Other hardware devices like I/O devices, system bus etc</td>
</tr>
<tr>
<td>Evaluation Procedure:</td>
<td>Empirical</td>
</tr>
</tbody>
</table>
## Evaluation Formulae:

<table>
<thead>
<tr>
<th>Capacity = C_{mean} / R_{mean}</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{mean} = \sum C_i / N$</td>
</tr>
<tr>
<td>$C_i = Z_i / T_i$</td>
</tr>
</tbody>
</table>

Where,

- $Z_i$: number of concurrent responses observed over the set period of time for i-th evaluation
- $T_i$: Set period of time for i-th evaluation
- $N$: Number of evaluations

## Result Type:

- Floating point value

## Static / Dynamic:

- Dynamic

## Universal / Subset / Existential / Specific:

- Universal

## Application dependent / Independent:

- Independent

## Decomposition rule:

- $CAP_i \geq CAP$, where $CAP_i$ is the capacity measure of component i (i=1, 2, ..., n), CAP is the capacity measure of system

## Composition rule:

- $CAP = \text{minimum} (CAP_i)$, where CAP measure is the capacity of system, $CAP_i$ is the capacity measure of component i (i=1, 2, ..., n).

## Consequence:

1. Higher the capacity of the component, the higher the number of concurrent requests served and lesser the response time
2. Lower the capacity of the component, the lower the number of concurrent requests served and higher the response time

## Related Parameters:

- Throughput, Parallelism constraint, End-to-End Delay
<table>
<thead>
<tr>
<th><strong>Domain of Usage:</strong></th>
<th>Domain Independent</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Error Situation:</strong></td>
<td>A low value of throughput results in a low number of concurrent requests served and longer response time.</td>
</tr>
<tr>
<td><strong>Aliases:</strong></td>
<td>Degree of concurrency</td>
</tr>
</tbody>
</table>

Table A.11 Attribute: Turn-around Time

<table>
<thead>
<tr>
<th><strong>Intent:</strong></th>
<th>It is a measure of the time taken by the component to return the result.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong></td>
<td>It is defined as the time interval between the instant the component receives a request until the final result is generated.</td>
</tr>
</tbody>
</table>
| **Motivation:** | 1. It indicates the delay involved in getting results from a component.  
2. It is one of the measures of the performance offered by a component. |
| **Applicability:** | This attribute can be used in any system, which specifies bounds on the response times of its components. |
| **Model Used:** | Empirical approach. |
| **Metrics Used:** | Mean Turn-around Time. |
| **Influencing Factors:** | 1. Implementation (algorithm used, multi-thread mechanism etc).  
2. Speed of the CPU. |
<table>
<thead>
<tr>
<th><strong>3. Available memory.</strong></th>
<th><strong>4. Load on the system.</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>5. Operating System's access policy for resources like: CPU, I/O, memory, etc.</strong></td>
<td></td>
</tr>
</tbody>
</table>

**Evaluation Procedure:**
1. Record the time instant at which the request is received.
2. Record the time instant at which the final result is produced.
3. Repeat steps 1 and 2 for 'n' representative requests.
4. Calculate the Mean Turn-around Time.

**Evaluation Formulae:**
\[
\text{MTAT} = \frac{\sum_{i=1}^{n} (t_2(t_1))}{n}.
\]

MTAT: Mean Turn-around Time.

- \(t_1\): time instant at which the request is received.
- \(t_2\): time instant at which the final result is produced.
- \(n\): number of representative requests.

**Result Type:** Floating Point Value in milliseconds.

**Static / Dynamic:** Dynamic.

**Universal / Subset / Existential / Specific:** Universal

**Application dependent / Independent:** Independent

**Decomposition rule:** \(\text{MTAT}_i < \text{MTAT}\), where \(\text{MTAT}_i\) is the turn-around time measure of component \(i (i=1, 2, ..., n)\), \(\text{MTAT}\) is the turn-around time measure of system.

**Composition rule:** \(\text{MTAT} = \text{MTAT}_1 + \text{MTAT}_2 + ... + \text{MTAT}_n\), where \(\text{MTAT}\) is the turn-around time measure of system, \(\text{MTAT}_i\) is the turn-around time.
measure of component \( i \) (\( i=1, 2, ..., n \)).

**Consequence:** Lower the time interval between the instant the request is received and the response is generated; lower the Mean Turn-around Time.

**Related Parameters:** Throughput, Capacity.

**Domain of Usage:** Domain Independent.

**Error Situation:** A high value of Internal Response Time results in:
1. Longer delays in producing the result.
2. Higher round trip time.

**Aliases:** Latency, Delay.

<table>
<thead>
<tr>
<th>Table A.12 Attribute: Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intent:</strong> It indicates the duration when a component is available to offer a particular service.</td>
</tr>
<tr>
<td><strong>Description:</strong> It is defined as the percentage of time a component is available to offer its services.</td>
</tr>
<tr>
<td><strong>Motivation:</strong> 1. It allows an evaluation of the downtime of the component. 2. It is a reflection of the dependability of the component.</td>
</tr>
<tr>
<td><strong>Applicability:</strong> This parameter can be used in any system, which requires its components to offer a specific level of Availability. Using this parameter, the Availability of a given component can be calculated before being incorporated into the system.</td>
</tr>
<tr>
<td><strong>Model Used:</strong></td>
</tr>
<tr>
<td>---------------</td>
</tr>
</tbody>
</table>

| **Metrics used:** | Mean Time Between Failures (MTBF)  
Mean Time To Repair (MTTR) |
|-------------------|-----------------------------|

| **Influencing Factors:** | 1. Dependability of the component.  
2. Fault tolerance of the hardware.  
3. Efficiency of the repair mechanism in place. |
|-------------------------|------------------------------------------------|

| **Evaluation Procedure:** | 1. Calculate the MTBF of the component.  
2. Calculate the MTTR of the component.  
3. Calculate the availability of the component. |
|---------------------------|------------------------------------------------|

<table>
<thead>
<tr>
<th><strong>Evaluation Formulae:</strong></th>
<th>Availability = (MTBF – MTTR) / MTBF</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Result Type:</strong></th>
<th>Floating Point Value.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Static / Dynamic:</strong></th>
<th>Dynamic</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Universal / Subset / Existential / Specific:</strong></th>
<th>Universal</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Application dependent / Independent:</strong></th>
<th>Independent</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Decomposition rule:</strong></th>
<th>A_i &gt;= A, where A_i is the availability measure of component i (i=1, 2, ..., n), A is the availability measure of system</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Composition rule:</strong></th>
<th>A = minimum (A_i), where A is the availability measure of system, A_i is the availability measure of component i (i=1, 2, ..., n).</th>
</tr>
</thead>
</table>

| **Consequence:** | 1. Higher dependability of the component results in higher availability.  
2. Lower the recovery time, higher the availability. |
|------------------|------------------------------------------------|

| **3.** Higher the security rating of the component, higher its availability during denial of service attacks. |
| **Related Parameters:** Dependability, Stability, Security. |
| **Domain of Usage:** Domain Independent. |
| **Error Situation:** Low availability results in: |
| 1. Higher Turn Around Time. |
| 2. Unreliable service. |
| 3. Lower system performance. |
| **Aliases:** Uptime. |
Appendix B: UMM-annotated UniFrame Domain Specific Language

\[
\text{<UMM_UDSL>::=} \ \text{<Feature_Expression> | <Constraint_Expression> |} \\
\text{<Design_Feature_Expression> | <Use_Case_Expression>}
\]

\[
\text{<Feature_Expression>::=} \ \text{<Optional> | <Mandatory> | <Composite> |} \\
\text{<Non_Exclusive> | <Alternative>}
\]

\[
\text{<Optional> ::= <Feature>?} \\
\text{<Mandatory> ::= <Feature> | <Feature>!} \\
\text{<Composite> ::= ALL ( <Feature_List> )} \\
\text{<Non_Exclusive> ::= MORE-OF ( <Optional_List> )} \\
\text{<Alternative> ::= ONE-OF ( <Optional_List> )} \\
\text{<Feature_List> ::= <Mandatory_List> | <Optional_List> |} \\
\text{<Mandatory_List> <Optional_List> | <Optional_List> <Mandatory_List> |} \\
\text{<Optional_List> <Mandatory_List> | <Mandatory_List> <Optional_List> |} \\
\text{<Mandatory_List> <Optional_List> <Mandatory_List>}
\]

\[
\text{<Optional_List> ::= <Optional> | <Optional> <Optional_List> |} \\
\text{<Mandatory_List> ::= <Mandatory> | <Mandatory> <Mandatory_List> |} \\
\text{<Feature> ::= <Atomic_Feature> | <Feature_Expression>}
\]

\[
\text{<Atomic_Feature> ::= FEATURE}
\]

\[
\text{<Constraint_Expression>::=} \ \text{<Multiplicity> | <Default> | <Mapping> | <Satisfaction>}
\]

\[
\text{<Multiplicity> ::= MULTIPLICITY( ( <Feature>, <Feature> ) :} \\
\text{<Multiplicity_Expression> )}
\]

\[
\text{<Multiplicity_Expression> ::= NUMBER | NUMBER...* | NUMBER...NUMBER}
\]

\[
\text{<Default> ::= DEFAULT( <Feature>:<Feature> )}
\]

\[
\text{<Mapping> ::= MAP ( <Feature>:<Feature> )}
\]

\[
\text{<Satisfaction> ::= <Require> | <Reject> | <Mutual_Require> | <Include> |} \\
\text{<Exclude>}
\]

\[
\text{<Require> ::= REQUIRE( <Feature_List> )}
\]

\[
\text{<Reject> ::= REJECT( <Feature_List> )}
\]

\[
\text{<Mutual_Require> ::= MUTUAL_REQUIRE( <Feature_Lists> )}
\]

\[
\text{<Include> ::= INCLUDE( <Feature>, <Feature> )}
\]

\[
\text{<Exclude> ::= EXCLUDE( <Feature>, <Feature> )}
\]

\[
\text{<Design_Feature_Expression>::=} \ \text{<Design_Feature_Interaction> |} \\
\text{<Design_Feature_Interface>}
\]
<Design_Feature_Interaction> ::= INTERACT(<Design_Feature>,<Design_Feature>)
<Design_Feature_Interface> ::= INTERFACE( <Design_Feature> :
   PROVIDED_INTERFACE: <Interface_List> ,
   REQUIRED_INTERFACE: <Interface_List> )
<Interface_List> ::= INTERFACE <Interface_List> | INTERFACE
<Design_Feature> ::= SYSTEM | SUBSYSTEM | <Abstract_Component>
<Abstract_Component> ::= 'ABSTRACT COMPONENT: ' <Comp_Name>
   'COMPONENT ATTRIBUTES: '
   <Component_Attr>
   'COMPUTATION ATTRIBUTES: '
   <Computation_Attr>
   'COOPERATION ATTRIBUTES: '
   <Cooperation_Attr>
   'AUXILIARY ATTRIBUTES: '
   <Auxiliary_Attr>
   'SERVICE ATTRIBUTES: '
   <Service_Attr>

<Component_Attr> ::= 'NAME: '<Comp_Name>
   'AUTHOR: '<Author>
   'SUBCASE: '<Subcase>
   'DESCRIPTION: '<Description>
   'CHANGELIST: '<ChangeList>
<Comp_Name> ::= STRING
<Author> ::= STRING
<Subcase> ::= STRING
<Description> ::= TEXT
<ChangeList> ::= 'DATE: ' DATE, 'TIME: 'TIME,
   'DESCRIPTION: '<Description>,
   'EDITOR: '<Editor> <ChangeList> |
   'DATE: ' DATE, 'TIME: 'TIME,
   'DESCRIPTION: '<Description>,
   'EDITOR: '<Editor>
<Editor> ::= STRING

<Computation_Attr> ::= 'INHERENT ATTRIBUTES: ' <Inherent_Attr>
   'FUNCTIONAL ATTRIBUTES: '<Functional_Attr>

>Inherent_Attr> ::= 'VERSION: '<Version>
   'COMP_AUTHOR: ' <Author>
   'ID: ' <Id>
   'EXECUTION ENVIRONMENT: '
   <Execution_Env>
   'SYSTEM NAME: ' <System_Name>
   'DOMAIN NAME: ' <Domain_Name>
‘VALIDITY:’ <Validity>
‘COMPONENT MODEL: ’<Component_Model>
‘ATOMICITY: ’ <Atomicity>
‘DATE DEPLOYED: <Date_Deployed>
‘REGISTRATION: ’ <Registration>

<Version> ::= STRING
<Author> ::= STRING
<Id> ::= STRING
<Execution_Envt> ::= STRING
<System_Name> ::= STRING
<Validity> ::= NUMBER
<Component_Model> ::= STRING
<Atomicity> ::= BOOLEAN
<Date_Deployed> ::= DATE
<Registration> ::= ‘REGISTERED AT: ‘ STRING,
    <Registration> |
    ‘REGISTERED AT: ‘ STRING

<Functonal_Attr> ::= ‘FUNCTION CONTRACTS:
    <Function_Contracts>
    ‘ALGORITHMS: <Algorithm>
    ‘DESIGN PATTERNS: ‘<Design_Patters>
    ‘KNOWN_USAGES: ‘ <Known_Usages>
    ‘ALIASES: ‘ <Aliases>
    ‘RESOURCES: ‘ <Resources>
    <Function_Contracts> ::= ‘FUNCTION: ‘<Function>
    ‘SEMANTIC CONTRACT: ‘<Semantic_Contract>
    ‘CONCURRENCY: ‘<Concurrency>
    ‘TECHNOLOGY: ‘ <Technology>
    ‘DESCRIPTION: ‘ <Description>
    <Function_Contracts> |
    ‘FUNCTION: ‘<Function>
    ‘SEMANTIC CONTRACT: ‘<Semantic_Contract>
    ‘CONCURRENCY: ‘<Concurrency>
    ‘TECHNOLOGY: ‘ <Technology>
    ‘DESCRIPTION: ‘ <Description>
    ‘INTEGRITY: ‘<Invariant>
    <Pre> ::= LOGICAL_ASSERTION
    <Post> ::= LOGICAL_ASSERTION
    <Invariant> ::= LOGICAL_ASSERTION
    <Concurrency> ::= STRING
<Technology> ::= STRING

<Algorithm> ::= ‘ALGO NAME: ‘<Name>
   ‘ALGO DESCRIPTION: ‘<Description>
   ‘COMPLEXITY: ‘<Complexity>
   ‘ALGO FUNCTION: ‘<Function>
   <Algorithm> | 
   ‘ALGO NAME: ‘<Name>
   ‘ALGO DESCRIPTION: ‘<Description>
   ‘COMPLEXITY: ‘<Complexity>
   ‘ALGO FUNCTION: ‘<Function>

(Name) ::= STRING

<Complexity> ::= STRING

<Design_Patterns> ::= ‘DESIGN PATTERNS:’ STRING,
   <Design_Patterns> | 
   ‘DESIGN PATTERNS: ‘ STRING

<Known_Usages> ::= ‘KNOWN USAGE: ‘ STRING,
   <Known_Usages> | 
   ‘KNOWN USAGE: ‘ STRING

<Aliases> ::= ‘ALIAS: ‘ STRING, <Aliases> | 
   ‘ALIAS: ‘ STRING

<Resources> ::= ‘ARCHITECTURE: ‘ <Architecture>
   ‘SPEED:’ <Speed> ‘LOAD: ‘ <Load>
   <Resources> | 
   ‘ARCHITECTURE: ‘ <Architecture>
   ‘SPEED:’ <Speed> ‘LOAD: ‘ <Load>

<Architecture> ::= STRING

<Speed> ::= NUMBER

<Load> ::= NUMBER

<Cooperation_Attr> ::= ‘COLLABORATORS: ‘<Collaborators>

<Collaborators> ::= ‘FUNCTIONNAME: <Function>
   ‘EXPECTED: ’ <Expected>
   ‘REQUIRED: <Required>
   ‘PROVIDED: <Provided>
   <Collaborators> | 
   ‘FUNCTIONNAME: <Function>
   ‘EXPECTED: ’ <Expected>
   ‘REQUIRED: <Required>
   ‘PROVIDED: <Provided>

<Expected> ::= STRING

<Required> ::= STRING

<Provided> ::= STRING
<Auxiliary_Attr> ::= 'MOBILE: ' <Mobile> 'MOBILITY: ' <Mobility> 'FAULT TOLERANCE: ' <Fault_Tolerance> 'SECURITY LIST: ' <Security_List>

<Mobile> ::= BOOLEAN
<Mobility> ::= STRING
<Fault_Tolerance> ::= TEXT
<Security_List> ::= 'SECURITY: ' STRING, <Security_List> | 'SECURITY: ' STRING

<Service_Attr> ::= 'PARALLELISM CONSTRAINT: ' <Parallelism_Constraint> 'ORDERING CONSTRAINT: ' <Ordering_Constraint> 'EXECUTION RATE: ' <Execution_Rate> 'AVAILABLE RESOURCES: ' <Available_Resources> 'QoS ATTRIBUTES: ' <QoS_Attributes>

<Parallelism_Constraint> ::= STRING
<Ordering_Constraint> ::= STRING
<Execution_Rate> ::= NUMBER
<Available_Resources> ::= ADDITIONAL REFS: <Add_References> TYPE: <Type> TYPE UNIT: <Type_Unit> VALUE: <Value> VALUE UNIT: <Value_Unit> RESOURCE PAIRS: <Resource_Pairs>

<Add_References> ::= TEXT
<Type> ::= STRING
<Type_Unit> ::= STRING
<Value> ::= STRING
<Value_Unit> ::= STRING
<Resource_Pairs> ::= TYPE VALUE: <Type_Value> DATA VALUE: <Data_Value>

<Type_Value> ::= NUMBER
<Data_Value> ::= NUMBER


<QoSId> ::= STRING
<Value> ::= NUMBER
<Extension> ::= TEXT

<?Use_Case_Expression> ::= <Use_Case_Component_Level> | <Use_Case_Function_Level>
  <Use_Case_Component_Level> ::= USE-CASE: PATH_C (<Abstract_Component_List>)
  <Use_Case_Function_Level> ::= USE-CASE: PATH_F (<Function_Call_List>)
  <Abstract_Component_List> ::= <Abstract_Component> <Abstract_Component_List> |
                             <Abstract_Component>
  <Function_Call_List> ::= <Function_Call> <Function_Call_List> |
                         <Function_Call>
  <Function_Call> ::= <Comp_Name>.<Function_Name> [<Communication_Pattern>]
  <Function> ::= <Ret_Type> <Function_Name> ( <Arg_List> ) [THROWS <Exception_List>]
  <Ret_Type> ::= TYPE
  <Function_Name> ::= STRING
  <Arg_List> ::= TYPE, <Arg_List> | TYPE
  <Exception_List> ::= EXCEPTION, <Exception_List> | EXCEPTION
  <Communication_Pattern> ::= CP1 | CP2S | CP2A