EXPERIMENTING WITH MULTILEVEL MATCHING
CONCEPTS FOR SOFTWARE COMPONENTS

A Technical Report
Report Number

Submitted to the Faculty
Of Purdue University

By
Prathiba Katuri

In Partial Fulfillment of the
Requirements for the Degree
Of
Master of Science

July 2006
To Amma, Nanna, and Srinu.
ACKNOWLEDGMENTS

Being a Graduate Student at the Department of Computer and Information Science, Indiana University-Purdue University has been an excellent experience. I truly believe that the knowledge gained here would be helpful in achieving my career goals.

I would like to take this opportunity to express my gratitude to people who have helped me through the course of my project.

First, I would like to express my sincere thanks to Dr. Rajeev R. Raje, my Advisor, for his constant support and guidance throughout my graduate studies and project work. His in-depth knowledge and valuable insights have helped me to successfully complete my project and have always encouraged me to learn new things.

I truly appreciate Dr. Andrew Olson for being on the advisory committee and for his painstaking efforts to review my report and work.

I would also like to thank Dr. Xukai Zou for being on the advisory committee and reviewing my work.

Finally, I would like to thank my parents, and my husband Srinivas for their constant love, support and encouragement throughout the course of my studies.
TABLE OF CONTENTS

LIST OF FIGURES ......................................................................................................................... I
LIST OF EXPERIMENTAL GRAPHS................................................................................................ ii
LIST OF TABLES ........................................................................................................................... iii
ABSTRACT ..................................................................................................................................... vi

1. INTRODUCTION ................................................................................................................... 1
   1.1 Problem Definition and Motivation .................................................................................. 2
   1.2 Objectives ...................................................................................................................... 3
   1.3 Contributions ................................................................................................................. 4
   1.4 Organization of the Report ............................................................................................ 5

2. RELATED WORK ................................................................................................................... 6
   2.1. Resource Discovery Service ......................................................................................... 6
   2.1.1 Directory Service ....................................................................................................... 6
   2.1.1.1. Universal Description, Discovery and Integration (UDDI) .................................... 7
   2.1.1.2. CORBA Trader Services ..................................................................................... 7
   2.1.2. Discovery Service ..................................................................................................... 8
   2.1.2.1. JINI .................................................................................................................... 8
   2.1.2.2. Service Location Protocol (SLP) ......................................................................... 9
   2.1.2.3. Ninja Project: Secure Service Discovery Service (SSDS) .................................. 9
   2.1.2.4. Salutation ........................................................................................................... 10
   2.2. Discovery Services for Software Components ........................................................... 11
   2.2.1. Agora ....................................................................................................................... 13
   2.2.2. Specific Search Engine for Software Components: SE4SC .................................. 14
   2.2.3. Uniframe Resource Discovery Service (URDS) ...................................................... 15
   2.3. Search Engines ............................................................................................................ 16
   2.3.1. Lycos ......................................................................................................................... 16
   2.3.2. Google ..................................................................................................................... 17
   2.3.2. A9 .............................................................................................................................. 18
2.4 Multilevel Matching

2.4.1.1 GESTALTS

2.4.1.2 Multi-level Immune Learning Detection (MILD)

2.4.1.3 Spam Trapper for Outlook 1.3.3

2.4.2 Extending Multilevel Matching in URDS

3. PREVIOUS WORK

3.1 Overview of UniFrame

3.1.1 Unified Meta-Component Model (UMM)

3.1.1.1 Components

3.1.1.2 Service and Service Guarantees

3.1.1.3 Infrastructure

3.1.2 The Uniframe Approach

3.1.2.1 The UMM Specification

3.2 Contracts for Software Components

3.2.1 Need for Contracts

3.3 Signature and Specification Matching of Components

3.3.1 Function Signature Matching

3.3.2 Function Specification Matching

3.4 Synchronization and QoS Matching for Components

3.4.1 Synchronization Policy Matches

3.4.1.1 Generalized and Specialized Matches

3.4.2 QoS Level Matching

4. MATCHING CATEGORIES

4.1 Need for Multilevel Matching

4.1.1 Why Matching?

4.2 Component-Matching Categories

4.2.1 Syntactic Level Matching

4.2.1.1 Generic Type Match

4.2.1.1(a) Ordered Generic Type Match
4.2.1.1(b) Reordered Generic Type Match ............................................................ 58
4.2.1.1(c) Generic Type Match Definitions ......................................................... 60
4.2.2 Semantic Level Matching .......................................................................... 61
4.2.2.1 Generic Pre/Post Match ......................................................................... 63
4.2.2.1 Generic Pre/Post Match Definitions ....................................................... 63
4.2.3 Synchronization Level Matching ................................................................ 65
4.2.3.1 Generic SP Match .................................................................................. 67
4.2.3.1 Generic SP Match Definitions ............................................................... 68
4.2.3.2 Generic SP – Impl Match ........................................................................ 69
4.2.3.2 Generic SP – Impl Match Definitions ..................................................... 70
4.2.4 QoS Level Matching .................................................................................. 73
4.2.4.1 Generic QoS Match ................................................................................ 76
4.2.4.1 QoS Match Definitions ......................................................................... 78
4.3 Representation of Multilevel Match ............................................................... 80

5. PROTOTYPE DETAILS ....................................................................................... 82
5.1 Prototype Design ......................................................................................... 82
5.2 User Query Requirements ........................................................................... 85
5.3 Prototype Matchers .................................................................................... 86
5.3.1 Syntactic Matcher ..................................................................................... 86
5.3.1(a) Exact Syntactic Match ....................................................................... 87
5.3.1(b) Relaxed Syntactic Match .................................................................... 87
5.3.2 Semantic Matcher .................................................................................... 88
5.3.2(a) Exact Semantic Match ....................................................................... 89
5.3.2(b) Relaxed Semantic Match ................................................................... 90
5.3.2(c) Spec# Programming Language .......................................................... 93
5.3.2(d) Spec# Compiler ................................................................................ 94
5.3.2(e) Boogie: Static Program Verifier ......................................................... 94
5.3.2(f) Simplify: Theorem Prover .................................................................... 96
5.3.3 Synchronization Matcher ......................................................................... 97
5.3.3(a) Exact Synchronization Match ............................................................. 97
5.3.3(b) Relaxed Synchronization Match ................................................................. 98
5.3.4 QoS Matcher ................................................................................................. 100
5.3.4(a) Exact QoS Match ..................................................................................... 100
5.3.4(b) Relaxed QoS Match ................................................................................ 101

6. EXPERIMENTAL ANALYSES OF THE PROTOTYPE ........................................ 103
   6.1 Main Objectives of the Experimental Verification ........................................ 103
   6.1.1 Experimental Setup .................................................................................... 103
   6.1.2 Different Categories of Experiments ......................................................... 104
   6.1.3 Different Categories of Queries .................................................................. 105
   6.2 Experimental Graphs .................................................................................... 108
   6.2.1(a) Components Returned - Multilevel Match (Single Method Queries) ..... 108
   6.2.1(b) Time Taken - Multilevel Match (Single Method Queries) .................... 115
   6.2.1(c) Precision - Multilevel Match (Single Method Queries) ....................... 122
   6.2.2(a) Components Returned - Multilevel Match (Multiple Method Queries) ... 126
   6.2.2(b) Time Taken - Multilevel Match (Multiple Method Queries) ............... 134
   6.2.2(c) Precision - Multilevel Match (Multiple Method Queries) .................... 142
   6.3 Inference from Experimental Graphs .............................................................. 147

7. FUTURE WORK AND CONCLUSION ................................................................. 162
   7.1 Conclusion ..................................................................................................... 162
   7.2 Contributions ................................................................................................ 163
   7.3 Future Enhancements .................................................................................... 164
   7.4 Summary ...................................................................................................... 165

LIST OF REFERENCES ............................................................................................. 166

Appendix A: Case Study to Verify the Matches ....................................................... 173
   A.1 Bank System Architecture ............................................................................. 173
   A.2 Representation of Bank System Abstract & Concrete Components ............. 176
   A.3 Validation of Matching Criteria .................................................................... 186
Appendix B: Algorithms to Implement Matchers .......................................................... 217
  B.1: Representation of Query Requirement .................................................................. 217
  B.2: Representation of Input Level Parameters and Type of Level ............................ 217
  B.3: Representation of Final Result ............................................................................ 218
  B.4: Algorithms to initialize all the Matchers ............................................................ 218
  B.5: Algorithm For Syntactic Matcher ....................................................................... 220
    B.5.1: Algorithm for Signature_Match ................................................................. 222
    B.5.1 (a): Algorithm For Ordered_Exact_Match .................................................. 223
    B.5.1 (b): Algorithm For ReOrdered_Exact_Match .............................................. 223
    B.5.1 (c): Algorithm For Ordered_Relax_Match ............................................... 224
    B.5.1 (d): Algorithm For ReOrdered_Relax_Match .............................................. 224
    B.5.1 (e): Algorithm For Exact_Match .................................................................. 225
    B.5.1 (f): Algorithm For Inheritance_Match .......................................................... 226
    B.5.1 (g): Algorithm For Coercion_Match ............................................................ 226
  B.6: Algorithm For Semantic Matcher ....................................................................... 227
    B.6.1: Algorithm For SemanticExactMatcher ...................................................... 227
    B.6.1 (a): Algorithm For WriteFunctionForSemanticMatch .................................. 230
  B.7: Algorithm For Synchronization Matcher ............................................................ 235
  B.8: Algorithm For QoS Matcher .............................................................................. 238

Appendix C .................................................................................................................... 240
LIST OF FIGURES

Figure 3.1: Federated Hierarchical Organization of ICBs (Source [NAN02]) ............... 27
Figure 3.2: URDS Architecture (Source [NAN02]) .................................................. 28
Figure 3.3: Abstract Component Specification for the Coordination Server ............. 34
Figure 3.4: A UMM Specification With Contracts of Coordination Server .......... 39
Figure 4.1: Representation of Generic Component Match ..................................... 54
Figure 5.1: Representation of Multilevel Matching Prototype ............................... 84
Figure 5.2: User Query Requirement for the Prototype .......................................... 85
Figure 5.3: Spec# Programming System ................................................................. 92
Figure A.1: Bank System ......................................................................................... 174
Figure A.2: A UMM Specification for Account Operation Server ......................... 185
Figure A-3: Spec# programming file showing SemanticExactTest.ssc ................. 241
Figure A-3: Spec# programming file showing SemanticExactTest.ssc (Continued...) 242
Figure A-4: Boogie.exe Output File for Semantic Exact Match .............................. 243
Figure A-5: Spec# Programming file showing SemanticRelaxTest.ssc ................. 244
Figure A-5: Spec# Programming file showing SemanticRelaxTest.ssc (Continued..) 245
Figure A-6: Spec# Programming file showing SemanticRelaxTest.ssc ................. 246
Figure A-6: Spec# Programming file showing SemanticRelaxTest.ssc ................. 247
Figure A-7: Boogie.exe Output File for Semantic Relaxed Match .......................... 247
Figure A-8: TLA + Module Specification PCTOME.tla ............................................ 249
Figure A-8: TLA + Module Specification PCTOME.tla (Continued..) ..................... 250
Figure A-9: TLA + Configuration file PCTOME.cfg .............................................. 251
LIST OF EXPERIMENTAL GRAPHS

Graph 6.1: Number of Components returned for Exact Multilevel Match ................. 109
  (Single Method Queries) ..................................................................................... 109
Graph 6.2: Number of Components returned for Relaxed Multilevel Match .............. 112
  (Single Method Queries) ..................................................................................... 112
Graph 6.3: Time Taken for Exact Multilevel Match (Single Method Queries) ......... 116
Graph 6.4: Time Taken for Relaxed Multilevel Match (Single Method Queries) ....... 119
Graph 6.5: Calculated Precision for Exact Multilevel Match .................................. 123
  (Single Method Queries) ..................................................................................... 123
Graph 6.6: Calculated Precision for Relaxed Multilevel Match ............................... 126
  (Single Method Queries) ..................................................................................... 126
Graph 6.7: Number of Components returned for Exact Multilevel Match .............. 127
  (Multiple Method Queries) .................................................................................. 127
Graph 6.8: Number of Components returned for Relaxed Multilevel Match .......... 131
  (Multiple Method Queries) .................................................................................. 131
Graph 6.9: Time Taken to execute Exact Multilevel Match ..................................... 136
  (Multiple Method Queries) .................................................................................. 136
Graph 6.10: Time Taken to execute Relaxed Multilevel Match ............................... 139
  (Multiple Method Queries) .................................................................................. 139
Graph 6.11: Calculated Precision for Exact Multilevel Match ................................. 143
  (Multiple Method Queries) .................................................................................. 143
Graph 6.12: Calculated Precision for Relaxed Multilevel Match ............................... 146
  (Multiple Method Queries) .................................................................................. 146
Graph 6.13: Sum of Number of Components – Ratio of Sum of Time Taken w.r.t sum of
  Syntactic Level Time (Exact Match - Multiple Method Queries) ....................... 148
Graph 6.14: Sum of Number of Relevant Components – Ratio of Sum of Time Taken
  w.r.t sum of Syntactic Level Time (Exact Match - Multiple Method Queries).. 148
Graph 6.15: Sum of Number of Components – Ratio of Sum of Time Taken w.r.t Sum of Syntactic Level Time (Relaxed Match - Multiple Method Queries) .................. 151
Graph 6.16: Number of Relevant Components – Ratio of Sum of Time Taken w.r.t Sum of Syntactic Level Time (Relaxed Match - Multiple Method Queries) ............... 151
Graph 6.17: Sum of Number of Components – Ratio of Sum of Time Taken w.r.t Sum of Syntactic Level Time (Exact Match - Single Method Queries) ......................... 153
Graph 6.18: Sum of Number of Relevant Components – Ratio of Sum of Time Taken w.r.t Sum of Syntactic Level Time (Exact Match - Single Method Queries) .......... 153
Graph 6.19: Sum of Number of Components – Ratio of Sum of Time Taken w.r.t Sum of Syntactic Level Time (Relaxed Match - Single Method Queries) ....................... 154
Graph 6.20: Sum of Number of Relevant Components – Ratio of Sum of Time Taken w.r.t Sum of Syntactic Level Time (Relaxed Match - Single Method Queries). 155
Graph 6.21: Average Precision Calculated – Ratio of Sum of Time Taken w.r.t Sum of Syntactic Level Time (Exact Match - Multiple Method Queries) ................. 156
Graph 6.22: Average Precision Calculated – Ratio of Sum of Time Taken w.r.t Sum of Syntactic Level Time (Relaxed Match - Multiple Method Queries) .............. 158
Graph 6.23: Average Precision Calculated – Ratio of Sum of Time Taken w.r.t sum of Syntactic Level Time (Exact Match - Single Method Queries) ......................... 159
Graph 6.24: Average Precision Calculated – Ratio of Sum of Time Taken w.r.t Sum of Syntactic Level Time (Relaxed Match - Single Method Queries) .................. 160
## LIST OF TABLES

Table 3.1: Definition of “Better” for Exact Match ........................................................................ 49
Table 3.2: Definition of “Close” for Relaxed Match ........................................................................ 50
Table 4.1: Instantiations of Generic Type Match: ........................................................................ 59
Table 4.2: Instantiations of Generic Pre/Post Match: ............................................................. 63
Table 4.3: Instantiations of Generic SP Match: [C_{SP} R_{1} Q_{SP}] ........................................... 68
Table 4.4: Instantiations of Generic SP – Impl Match: .................................................................. 70
Table 6.1: Different Categories of Queries ................................................................................. 106
Table 6.2: Query Q1 and Q2 Requirements ............................................................................... 107
Table 6.3: CompID/IDS returned for Queries Q1 and Q2 ............................................................ 107
Table 6.4(a): Graph 6.1: Query Q1 Requirements ................................................................... 110
Table 6.4(b): Graph 6.1: CompID/IDs returned for Query Q1 .................................................... 110
Table 6.4(c): Graph 6.1: Query Q7 Requirements .................................................................... 110
Table 6.4(d): Graph 6.1: CompID/IDs returned for Query Q7 .................................................... 111
Table 6.4(e): Graph 6.2: Query Q1 Requirements ................................................................... 112
Table 6.4(f): Graph 6.2: CompID/IDs returned for Query Q1 .................................................... 113
Table 6.4(g): Graph 6.2: Query Q9 Requirements .................................................................... 113
Table 6.4(h): Graph 6.2: CompID/IDs returned for Query Q9 .................................................... 114
Table 6.4(i): Graph 6.2: Query Q14 Requirements ................................................................... 114
Table 6.4(j): Graph 6.2: CompID/IDs returned for Query Q14 .................................................... 115
Table 6.5(a): Graph 6.3: Query Q1 Requirements ................................................................... 116
Table 6.5(b): Graph 6.3: Time Taken Results for Query Q1 ....................................................... 117
Table 6.5(c): Graph 6.3: Query Q2 Requirements ................................................................... 117
Table 6.5(d): Graph 6.3: Time Taken Results for Query Q2 ....................................................... 117
Table 6.5(e): Graph 6.3: CompID/IDs returned for Query Q2 .................................................... 118
Table 6.5(f): Graph 6.4: Query Q1 Requirements ................................................................... 120
Table 6.5(g): Graph 6.4: Time Taken Results for Query Q1 ....................................................... 120

\[
\begin{align*}
&(( [ Q_{Fi} ]_{Pre} \ R_{1} [ C_{Fi} ]_{Pre} ) \ R_{2} (( [ C_{Fi} ]_{Post} \ R_{3} [ Q_{Fi} ]_{Post} )))
\end{align*}
\]
Table 6.5(h): Graph 6.4: Query Q19 Requirements ................................................................. 121
Table 6.5(i): Graph 6.4: Time Taken Results for Query Q19 ................................................ 121
Table 6.5(e): Graph 6.4: CompID/IDs returned for Query Q19 ........................................... 121
Table 6.6(a): Graph 6.5: Query Q1 Requirements ............................................................... 123
Table 6.6(b): Graph 6.5: Precision Results for Query Q1 ...................................................... 124
Table 6.6(c): Graph 6.5: CompID/IDs returned for Query Q1 ............................................. 124
Table 6.6(d): Graph 6.5: Query Q5 Requirements ............................................................... 124
Table 6.6(e): Graph 6.5: Precision Results for Query Q5 ...................................................... 125
Table 6.7(a): Graph 6.7: Query Q2 Requirements ............................................................... 128
Table 6.7(b): Graph 6.7: CompID/IDs returned for Query Q2 ............................................. 128
Table 6.7(c): Graph 6.7: Query Q5 Requirements ............................................................... 129
Table 6.7(d): Graph 6.7: CompID/IDs returned for Query Q5 ............................................. 129
Table 6.8(a): Graph 6.8: Query Q1 Requirements ............................................................... 131
Table 6.8(b): Graph 6.8: CompID/IDs returned for Query Q1 in Relaxed Match ............... 132
Table 6.8(c): Graph 6.8: CompID/IDs returned for Query Q1 in Exact Match ..................... 132
Table 6.8(d): Graph 6.8: Query Q20 Requirements ............................................................. 133
Table 6.8(e): Graph 6.8: CompID/IDs returned for Query Q20 ........................................... 133
Table 6.8(f): Graph 6.8: Query Q9 Requirements ............................................................... 134
Table 6.9(a): Graph 6.9: Query Q2 Requirements ............................................................... 136
Table 6.9(b): Graph 6.9: Time Taken Results for Query Q2 ................................................ 136
Table 6.9(c): Graph 6.9: Query Q6 Requirements ............................................................... 137
Table 6.9(d): Graph 6.9: Time Taken Results for Query Q6 ................................................ 137
Table 6.9(e): Graph 6.9: CompID/IDs returned for Query Q6 ............................................. 138
Table 6.10(a): Graph 6.10: Query Q2 Requirements ............................................................. 140
Table 6.10(b): Graph 6.10: Time Taken Results for Query Q2 ............................................ 140
Table 6.10(c): Graph 6.10: Query Q6 Requirements ............................................................. 141
Table 6.10(d): Graph 6.10: Time Taken Results for Query Q6 ............................................ 141
Table 6.10(e): Graph 6.10: CompID/IDs returned for Query Q6 ........................................ 141
Table 6.11(a): Graph 6.11: Query Q2 Representations ......................................................... 144
Table 6.11(b): Graph 6.11: Precision Results for Query Q2 ............................................... 144
Table 6.11(c): Graph 6.11: Query Q5 Representations .................................................. 144
Table 6.11(d): Graph 6.11: Precision Results for Query Q5 .............................................. 145
Table 6.11(e): Graph 6.11: CompID/IDs returned for Query Q5 .................................... 145
Table 6.13(a): Graph 6.13 & 6.14: Time Taken w.r.t Syntactic Level Time .................. 149
Table 6.13(b): Graph 6.13: Sum of No. of Comps Returned............................................. 149
Table 6.13(c): Graph 6.14: Sum of No. of Relevant Comps Returned............................... 150
Table 6.21(a): Graph 6.21: Time Taken w.r.t Syntactic Level Time............................... 156
Table 6.21(b): Graph 6.21: Average Precision Results ................................................... 157
ABSTRACT


Distributed Computing Systems (DCS) are developed using geographically scattered heterogeneous software components. In recent years, there has been an increase in the demand for high-confidence, reliable and robust DCS, which led to the shift in the software development paradigm towards Component-based Software Development (CBSD). The UniFrame approach tries to create a seamless integration of heterogeneous and distributed software components for the development of DCS. UniFrame uses UniFrame Resource Discovery Service (URDS) for the discovery and selection of components deployed over the network. The selection of these software components is based on matching the component contracts, which provide information at their syntactic, semantic, synchronization and the Quality of Service levels leading to a Multilevel matching of software components. Matching of component contracts is done using the matching categories proposed at each one of the levels. This project focuses on assessing the Multilevel matching concepts in the context of software components using a prototype. The prototype incorporates the matching categories at each one of the levels and tries to empirically validate the approach. The experimental analysis using the prototype indicates the benefits of Multilevel matching as opposed to a Single level Matching.
1. INTRODUCTION

“The presence of Distributed Systems is omnipresent these days” [RAJ00]. The evolution of Distributed Component Systems (DCS) happened due to the change from a centralized computing system to a distributed system. Design of DCS is a challenging task, as these systems are geographically distributed with shared data, have an open architecture, are time sensitive to expected solution, have dynamic system configuration, and integration, etc. Even though design of DCS has improved over the years, the users demand for a more robust, reliable, flexible, adaptable, secured software system that is easy to use and deploy has increased. Hence in recent years, there has been a shift to component-based software development (CBSD).

CBSD is defined as building software systems by integrating previously developed and deployed software components. CBSD is built on the assumption that if certain parts of large programs reappear regularly then such common parts should be written once and reused whenever needed rather than rewriting them. This introduced the concept of Commercial off the shelf (COTS) concept, which has become a trend in distributed computing systems (DCS).

Certain issues arise in the development of component-based solutions to DCS; one of the common issues is that different DCS may follow different component models. Interoperability is absent in different component models due to the underlying implementation language or object model even though they are interoperable in terms of hardware and operating systems. This marks the need for seamless interoperation of components developed under different models.

The next issue is the Quality of Service (QoS) of individual components and the distributed computing system (DCS) developed from them. There has to be a standard catalogue that guarantees not only the QoS of individual component, but also for the entire DCS.

UniFrame, the Unified Meta-Model Framework addresses this issue of seamless integration of heterogeneous components and also considers the functional and non-
functional aspect to build a DCS. It provides a comprehensive framework for constructing a high-confidence DCS. UniFrame Resource Discovery Service (URDS) [NAN02], which addresses the resource discovery part of the construction process, discovers the components over the network. URDS provides services for an automated discovery and selection of components that meets the query requirements specified for the desired software system.

This project focuses on assessing the multilevel matching concepts in the context of software components using a prototype. All the four levels of software component specification are considered for multilevel matching: the syntactic, the semantic, the synchronization, and the QoS. It tries to empirically validate the multilevel approach via a prototype. The merits of multilevel matching as opposed to single level matching are explored in this project. The whole idea of implementing the prototype is to incorporate the concept of multilevel matching in URDS to make the selection process more comprehensive. Below is the problem definition and motivation for the project.

1.1 Problem Definition and Motivation

The main aim of this project is to incorporate multilevel matching concept in URDS to obtain better search and selection of components in the component discovery phase.

The Component discovery process involves the search for and selection of components. Every component participating in the discovery process is associated with a specification that describes the behavior of the component by representing properties like the services it offers, its usage, its interaction with other components in a distributed system, and the quantitative aspect of it. Discovery services match components against their specifications. In order to match the component specifications, there must be some criteria on which the matching can be based.

According to [BEU99], a component specification should be such that it provides parameters against which the component can be verified and validated, thus providing a kind of contract between the component and its users. A component specification is
categorized into four levels by [BEU99], the syntactic, the semantic, the synchronization, and the QoS levels. Previous researches such as [ZAR96], [KUM04], have proposed different matching criteria for the component matching. [ZAR96] has proposed the matching criteria for the syntactic and the semantic levels and [KUM04] has extended it to the synchronization and the QoS levels. This project uses the matching concepts proposed by both [ZAR96] and [KUM04] and devises a matching criterion to provide a better quality of components.

As the confidence of a user in a system directly depends on the services it offers, a comprehensive matching criterion is essential. Selection of components with such criteria will match closely with the user requirements. Multilevel matching provides smaller number of components but with a better quality. Matches are provided for all the four levels of component specification and at each level, two different sets of matches are considered: the exact and the relaxed. The intention of it is to provide user with more options during the component selection process.

1.2 Objectives

The overall objectives of this project are:

- To make selection process in URDS more comprehensive by incorporating the concept of multilevel matching.
- To devise a multilevel matching criterion based on the related work of [ZAR96] and [KUM04].
- To validate the matching criterion using a theoretical case study.
- To design and develop a prototype that uses the multilevel matching criterion.
- To empirically validate the multilevel approach via the prototype.
- To verify if the components returned in relaxed matches are equal or more than the components returned in exact match.
Based on the user feedback, validate the results by applying information retrieval techniques such as Precision.

To assess the results returned using the metrics such as the number of components returned, the time taken for matching and the precision calculated from user’s feedback.

To analyze the conducted experiments to see if the extra time taken in implementing multilevel match as opposed to single level match is worthwhile in getting more relevant components.

1.3 Contributions

Multilevel matching aims at providing a better quality of components than the normal search process does. Below is the significance of each match and its contribution to the multilevel match process:

Syntactic Match: The syntactic match considers matching the function signature specifications. Match based on types is done at this level. It matches the function name, the return type of the function, the argument types of the function. The exact match matches exactly with respect to the types, where as the relaxed match applies relaxation by applying either inheritance or coercion to the types.

Semantic Match: The semantic match considers matching of a component’s pre-conditions and post-conditions of its function. This match ensures that the exchanges between the requester and the provider have a common understanding of meaning of required services and data.

Synchronization Match: This match considers the matching of synchronization policies and implementation techniques of the component’s specifications. This match ensures the concurrent accesses made on the component’s interfaces in case of concurrent systems.
QoS Match: While all the three above matches ensure the functional properties of the system, this match ensures the non-functional properties of the system. With this match, the user is given an option to choose a QoS requirement that is most appropriate to his/her needs.

Having identified the required levels of matches, the next step is to apply the matching criterion to each of these levels and assess the quality of components returned. The prototype that has been developed supports the proposed matching criterion and implements multilevel matching. The contributions for this project are:

1. Multilevel matching criterion for software components has been devised based on the work of [ZAR96] and [KUM04].
2. Design and develop a prototype that incorporates multilevel matching criterion.
3. Experimental validation of the matching criterion using the prototype.

### 1.4 Organization of the Report

The project is organized into seven chapters. The first chapter, chapter 1 gives a preliminary introduction to the problem domain, stating the objectives and contributions to the thesis. The second chapter, chapter 2 discusses related work done so far on the discovery services and provides examples of some of them. The chapter 3 describes the previous work done as part of URDS; it mainly discusses the UniFrame approach and UniFrame principles. The chapter 4 presents the multilevel matching categories considered for the component’s specification. Chapter 5 discusses the design and implementation of prototype developed that incorporates the matching criteria. Chapter 6 presents the experimental analyses and verification of results. Chapter 7 gives the conclusions and the future work for this project.
2. Related Work

This chapter focuses on the related work of the report. The first section, 2.1, focuses on few of general discovery services. The second section, 2.2, discusses about the component discovery services and presents a few examples of this kind. Section, 2.3, describes about the search engines and presents few examples of them. Section 2.4 discusses the profiling in discovery services and finally section 2.5 presents an idea of multilevel matching and considering the concept of multilevel matching in URDS.

2.1. Resource Discovery Service

Resource Discovery refers to the process of identifying the resources on a network and making the resources available to the users and the applications [SIR03]. Such a discovery of resources plays a significant role in locating, accessing, retrieving and managing pertinent resources from distributed and heterogeneous networks. The resource discovery service protocols can be categorized into two main categories:

- Lookup Services or Directory Services
- Discovery Services

The major difference between the two services is that Lookup services are passive services and the Discovery services are active services. Discovery service usually supports lookup but Lookup services do not support Discovery services.

2.1.1 Directory Service

In case of Lookup services, the entity that requests the service has to initiate a request to obtain the information of a required service. This requires the presence of some directory or other agent to respond to the request. Some of the directory services are UDDI Registry [UDD00], CORBA Trader Services [OMG01], LDAP [LDA01], Global Name Service (GNS) [LAM86] and Domain Name Service (DNS) [MOU87]. For the
sake of brevity, only a couple of the above mentioned directory services are described below.

2.1.1.1. Universal Description, Discovery and Integration (UDDI)

Universal Description, Discovery and Integration (UDDI) [UDD00] is a set of Web services. It describes a registry of Web services and programmatic interfaces for publishing, retrieving, and managing information about services. A Web service is a component or an application whose functionality is exposed over the Internet. UDDI creates a standard interoperable platform that enables companies or applications to easily, dynamically find and use Web services over the Internet. It is a cross programming platform that uses Simple Object Access Platform (SOAP) built on top of Extensible Markup Language (XML). The UDDI architecture consists of three components: Web Service Providers, Service Requesters and Service Brokers. It is a centralized model where service providers register their services with the common service broker or registry. Service Requesters use Web Service Description Language (WSDL) to search for services at service registry. UDDI aims to bridge the gap between the companies that develop and publish XML Web services and the companies that try to locate and consume them.

2.1.1.2. CORBA Trader Services

The CORBA Trader Services [OMG00] is a lookup service for service providers and service consumers in a distributed computing environment. There are three main components of CORBA Trader service: Exporters, Importers and Traders. Advertising or offering a service is called Export. Matching against the needs or discovering services is called Import. The Exporters, which are service providers, give the Trader a description of a service and the location of an interface where the service is available in order to export their services. The Importers, which are service consumers, specify the Trader with the required service characteristics. The Trader then checks for the desired service and provides the Importers with its location. Importer then can interact with the service.
The local Traders, in addition to searching the local directory, also propagate the query to other Traders. The Trading Object Service in an OMG environment allows internetworking between Traders and Objects in order to export services and import information about one or more exported services, according to some criteria. Traders are defined as CORBA interfaces using Interface Definition Language (IDL). All advertisements, requests and replies are CORBA objects.

### 2.1.2. Discovery Service

In case of Discovery service the services are active in nature, and the services allow components to discover each other in a spontaneous manner based on service description with little or no help. Some of the discovery service protocols are JINI [SUN01], Service Location Protocol (SLP) [GUT99, PER99], Ninja Project: Secure Service Discovery Service (SSDS) [CZE99], Salutation [SAL99], Universal Plug and Play (UpnP) [REK99, MIC00], Bluetooth Service Discovery Protocol [GOL99, MIL99], DReggie and Uniframe Resource Discovery Service (URDS).

#### 2.1.2.1. JINI

JINI is Java based framework for spontaneous discovery developed by Sun Microsystems [SUN01]. The purpose of JINI is to federate groups of devices and software components into a single, dynamic distributed system. It is a security system, integrated into Java Remote Method Invocation (RMI) to achieve communication among clients and services. The main components in the JINI discovery service are the Service, Client and the Lookup service. Service is an entity that can be used by a person, a program, or another service. It may be a computation, storage, a communication channel to another user, a software filter, a hardware device, or another user. Services in JINI system communicate with each other by using a service protocol, which is a set of interfaces written in Java. A Service is added to a Lookup service by a pair of protocols, a service first locates the lookup service using the discovery protocol and then joins it by the join protocol. The Client locates an appropriate service by its type. It locates the required service in the Lookup service. Lookup service acts as a directory service for
Services and Clients to register themselves and locate services respectively. The Lookup service in Jini is based on the attribute comparisons.

2.1.2.2. Service Location Protocol (SLP)

Service Location Protocol [GUT99, PER99] is a standard for dynamically discovering network resources by Internet Engineering Task Force (IETF). SLP architecture comprises of User Agents (UAs), Service Agents (SAs), and Discovery Agents (DAs). SLP uses syntactic information for matching and describing the services. UAs are responsible for discovering resources on behalf of clients that requests services. SAs advertise their services to Discovery Agents. DAs maintain all the list of services advertised by SAs and respond to UAs with their requests.

Discovery of DAs by UAs and SAs can be implemented in three ways 1) Active Discovery: SAs and UAs multicast SLP requests. 2) Passive Discovery: DAs periodically multicast advertisement messages to announce their presence. 3) Dynamic Host Configuration Protocol (DHCP) options for SLP: UAs and SAs can locate DAs using DHCP where the configured DHCP servers distribute the DAs addresses to agents that require them.

SLP configuration allows two modes of operation 1) without DAs: UAs multicast or broadcast services to SAs, which are listening on well-known ports. SAs respond to UAs using unicast when the match is found. 2) With DAs: this is a more efficient configuration. The UAs, SAs and DAs are configured as members of scope and communication takes place between them only if all of them are in same scope. UAs and SAs communicate using DAs, which listen on some well-known ports using unicast messages.

2.1.2.3. Ninja Project: Secure Service Discovery Service (SSDS)

The SSDS [CZE99] is a part of Ninja project [NIN02] at the University of California, Berkley. It provides significant reliability; scalability and security compared to other discovery protocols. SSDS is implemented in Java and uses Java-RMI for remote
calls. It uses XML for service description and location. The main components of SSDS are: Service Discovery Service (SDS) Servers, Services, Clients, Certificate Authority, and Capability Manager. The SDS servers are organized into hierarchical domains. The SDS servers periodically multicast authenticated messages, which contain a multicast address for sending service announcements. SDS servers cache the service descriptions advertised in the domain. The Services multicast their service descriptions to the multicast address after listening to the announcements from the SDS servers using an authenticated, encrypted, one-way service broadcast. Clients listen on a well-known SDS global multicast address and use authenticated RMI to connect to the SDS server and submit an XML template format query. Certificate Authority generated certificates are used by SDS to authenticate bindings between principals and their public keys. The Capability Manager generates and distributes capabilities to the users and the SDS uses these capabilities as an access control mechanism. SSDS provides a strong mandatory security with all parties being authenticated and the message traffic is encrypted. SDS servers, being arranged as a hierarchical domain increase the system scalability and also serves for the failure detection and an automatic restart of the failed servers.

2.1.2.4. Salutation

The Salutation protocol [SAL99] is a service discovery and session management protocol. It is an open standard protocol independent of operating systems, communication protocols, and hardware protocols. The main aim of Salutation is to solve problems of service discovery and utilization among a broad set of appliances and environment. The architecture provides a standard method for applications, services and devices to describe as well as to advertise their capabilities to other applications, services and devices. There are three main components namely Client, Server and Salutation Lookup Manager (SLM). SLM functions as a service broker for services. It manages all communication and bridges across different communication media. The SLM defines its protocol based on SunRPC. Salutation defines a specific record format for describing and locating services. It classifies services into a collection of Functional Units (FU). Each FU is composed of descriptive attribute record representing an essential feature. The
query from Client regarding a particular service reaches the local SLM directory. The local SLM directory is then searched for the required service type. The query is also propagated to other SLMs where one SLM can be a client to another SLM.

### 2.2. Discovery Services for Software Components

The main objective of the component discovery services is to provide a platform to publish and consume robust software components. This is viewed as any other discovery service, with the resources being discovered being software components. Some of the features supported by the component discovery service are:

- **Heterogeneity:** The general definition is defined as the difference in the networks, computer hardware, operating systems, programming languages, and the implementation of the components by different developers of the distributed applications. The discovery service of components must support heterogeneity as the components developed and deployed on to the network could belong to different distributed computing models such as: Java RMI, .NET, and CORBA etc. So, the component model supporting the discovery of components should not be limited to a particular distributing computing models, which will impose a restriction on the component developers to stick with a particular model. Hence, the component discovery of heterogeneous components is required. A component discovery environment, which supports heterogeneity, must also support interoperability. So, an environment with heterogeneous components needs a unified mechanism for locating, interacting with and representing services.

- **Scalability:** This refers to the ability of the system to function effectively when the number of resources and the number of users increases. As more and more components are developed and deployed on to the network, the number of clients and the services increase in an environment. This leads to a burden on the component discovery service architecture during the interactions and dynamic discovery of components. A system should scale in such a situation so the
scalability is required. The scalability can be improved in distributed discovery service architecture due to its hierarchical nature.

- **Activeness**: The discovery service must be active. When a client is looking for particular service, it initiates the service and the resource providing the service must respond to it spontaneously. This requires that the service providers must be actively listening to the client’s requests.

- **Availability**: Availability is the property of a system by which it always responds to a valid client request. In a system where multiple requests and responses are being exchanged, there might be a case where a particular service may not be available anymore due to a server’s failure or due to mobility. If it is a critical system, then availability of the service plays an important role. So, the component discovery architecture must quickly react to the faults and make the service available.

- **Interoperability**: In a component discovery environment, the existence of the heterogeneous components and the interactions between them are inevitable. Such a network needs a unified mechanism for locating, interacting with and representing services. So, the application developers need Interoperability to deal with the diverse components and their interaction models.

- **Profiling**: Profiling in discovery of components enhances the query propagation techniques and improves the quality of components discovered. Profiling is achieved by storing the history of the previous component discoveries.

- **Multilevel matching**: Multilevel matching is an important feature that matches the components specifications at multiple levels such as syntactic, semantic, synchronization and the QoS characteristics. This helps in improving the matching of the query as the component can be matched at four levels given in the specification giving flexibility to the user.
• Reliability: Reliability is the ability of a system or component to perform the required functions under stated conditions for the specified period of time. A reliable component discovery enables the discovery of the best components without compromising on the quality of components.

• Security: Security plays an important role in the component discovery architecture. The resources are out in an open environment and they have to communicate frequently as part of a discovery process. Hence, the component model must provide a secured environment for maintaining the confidentiality, integrity and availability of the resources.

• Concurrency: This feature allows the clients to access the resources concurrently and also allow multiple/concurrent accesses to their interfaces.

URDS, which is discussed in the later part of the chapter, includes some of the above features. Some of the discovery services specific to software components are discussed below.

2.2.1. Agora

Agora [SEA98] is a prototype being developed by the Software Engineering Institute at Carnegie Mellon University. The objective of this software prototype is to create an automatically generated, indexed, worldwide database for software products classified by component type (e.g. JavaBean or ActiveX control). It is a combination of introspection and Web search engines such that the cost to bring the software components to, and to find the software components in, software market place is reduced.

According to [SEA98], the integration of component technology and Web search can have huge impact on the emergence of an online component marketplace by:
• Providing developers with a worldwide distribution channel for software components.
• Providing consumers with a flexible search capability over a large base of available components.
• Providing a basis for the emergence of value-added component qualification services, with in and across specific business sectors.

Agora supports two basic processes, the location and indexing of components and the search and retrieval of a component. The location and indexing of components is primarily automated background task, search and retrieval needs human intervention. Components are introspected during the indexing phase to discover their interfaces. JavaBeans Introspector class is used to introspect JavaBeans and in CORBA, interface information is maintained in a separate interface repository. Once the component is identified, the information is decomposed into set of tokens and a document is created in the index that includes these tokens. Component interface information can be differentiated into different fields like methods, attributes, or events. The component name and type are also preserved as fields to enable searches by name and component type. Meta information about each component is also maintained with the document, including the Uniform Resource Locator (URL) for each component.

Searching and Retrieval in Agora is a two-step process. A query is specified by the user and optionally specifies the type of the component. These are searched against index collected by search agents. The result set of the query is sent back to the user for inspection. The result includes meta-information including the URL of the component. The user can then refine or broaden the search based on the number or quality of matches. Once the searcher completes the breadth-wise search to identify candidate components, individual components can be examined in detail. The URL of the component is returned to the component Introspector for re-introspection. Agora also includes Boolean operators to support advanced search criteria. Agora does not consider various levels of component matching, which would be the main focus of this project.

2.2.2. Specific Search Engine for Software Components: SE4SC
Specific Search Engine For Software Components SE4SC [HAO04] is a search engine that relies on reusable components as the building blocks for constructing software systems. SE4SC acquires the component resources from the component repository referring to Software Component Description Model (SCDM), and also can locate the component resources published by component provider according to the SCDM. SE4SC provides two ways for reusers to search for components, one is the keyword-based search, and the other is the topic directory based on systematic classification schema. SE4SC can only support a rough search for software components, but can forward the reusers search requests to retrieval engine of software component repository for accurate retrieval. The architecture of SE4SC consists of Repository Registry Center, crawlers, systematic classifier, search service interface, search service control, index database, component Descriptor Repository.

This model adopts two kinds of classification schema for classifying the software components. One is based on the faceted classification and the other schema is based on the systematic classification. Faceted schema is retrieval engine oriented. Component provider describes the software components based on this schema and offers the component descriptors that facilitate component reuser to retrieve components accurately with the aid of the retrieval engine. Systematic classification schema is search engine oriented. According to this schema, component descriptor offered by the component developer supports for reusers to search component roughly by using the search engine. The process of obtaining the desirable components is a two-step process: Search and Retrieval. First the component reusers find the candidate components roughly using the search engine and Second retrieval engine are used to retrieve the components from the candidate components.

### 2.2.3. Uniframe Resource Discovery Service (URDS)

Uniframe Resource Discovery Service (URDS) [NAN02] is a component resource discovery service, which is a part of Uniframe Research Project [RAJ01]. The URDS architecture provides the necessary infrastructure for an automated discovery and the selection of components meeting the necessary criteria. The components could belong
to different component models such as Java RMI, CORBA and .NET. The components are developed and deployed by the component developers. URDS is designed as a discovery service where components can be dynamically discovered and while the system developers are provided with the directory style access to the components. The motivation was to automate the process of assembling a DCS, for which components are discovered using URDS. URDS architecture has Internet Component Broker, Headhunter and Active Registry. URDS is discussed in detailed later in the chapter.

2.3. Search Engines

Most search Engines, such as Google, Lycos, HotBot, AltaVista and A9, all operate in the same way enabling users to search and locate information on the web. Search engines use a gathering program such as a crawler or spider, which explore the hyperlinked documents of the Web, looking for Web pages to index. These pages are stored in some kind of database or repository. A retrieval program takes a user query and creates a list of links to Web documents matching the words, phrases, or concepts in the query. Although the retrieval program itself is correctly called a search engine, by popular usage, the term now means a database combined with a retrieval program. A few of the prominent search engines are discussed below.

2.3.1. Lycos

Lycos [MAU97] is a search engine used for collecting, storing, and retrieving information about pages on the web. Spiders are the key for the search engines, which crawl in the web to search for relevant documents. One major difference between Lycos and Spiders is in the way the searching is done. Some Spiders explore the most recently found page first, resulting in a depth-first search of the web. In this case, the load on the target servers is high, sometimes causing the servers to crash. Lycos uses best-first search, which tends to find home pages rather than subsidiary pages, and so the Lycos catalog is biased toward more popular and more useful web pages. Lycos uses a best-first search based on the popularity heuristic. They define popularity as the number of external Web servers with at least one link to a Web page. External means that only link from one
server to a page on another count as evidence of popularity; by only counting one link per
server, Lycos limits the ability of authors to manipulate popularity data. This approach
tends to find home pages rather than subsidiary pages, so the Lycos catalog is biased
toward more popular and therefore more useful Web pages. Lycos pioneered the use of
automated abstracts. This means that Lycos identifies the 100 most important terms and
creates an abstract that is about one-fourth of the original document (for all the
documents that are available for matching). These abstracts are used during the search to
find the relevant documents. They are also displayed along with the list of relevant links,
allowing the users to quickly determine the relevant documents. For matching the query
with the relevant documents, three basic schemes have been used: Boolean keyword
query, regular expression matching, and vector space (statistical) retrieval. Lycos also
uses inverted file indexing. An inverted file (sometimes called a postings file) is a list of
all occurrences of words or tokens in the text database. For each unique word in the
database, the search engine keeps a list of documents containing that word, sometimes
with a list of all the positions where the word occurs.

2.3.2. Google

Google [BRI98] is a large-scale search engine, which makes heavy use of the
structure present in the hypertext. The main goal of this search engine is to provide more
efficient results than any other search engine and it does it by crawling and indexing the
Web. This search engine uses two important features to produce high precision results.
First, it uses PageRank algorithm where the quality ranking for each web page is
calculated using the link structure of the Web. Second, the link is used to improve the
search results.

The PageRank algorithm relies on the democratic nature of the Web by using its
vast link structure as an indicator of an individual page’s value. A link from page A to
page B is interpreted as a vote, by page A, for page B. A page can have a high PageRank
if there are many pages that point to it, or if there are some pages that point to it and have
a high PageRank. So, the weight associated to the vote depends on the page that casts the
vote, the more important the page the more weight the vote gets. In addition to the vote,
Google also remembers the search it conducts each time, which helps important and high-quality sites to achieve higher PageRank. Google combines PageRank with sophisticated text-matching techniques to find pages that are both important and relevant to the given search. Google does not limit its search just to how many number of times a term appears on a page but it examines all the aspects of the page’s content, including the content of the pages linking to it. This way it ensures if the match is relevant to the user or not. It also presents personalized search results to the users based on their respective profile settings, where a user can select the categories of topic (ex. Computers, Science, Music, Sports etc). It also incorporates multilevel matching techniques to provide the efficient search results.

2.3.2. A9

A9 [AMA04] is a new search engine from Amazon.com, which is mainly to be used as a search engine for e-commerce. It researches and builds innovative technologies to improve search experience for e-commerce applications. This uses search results from five powerful information sources: Web and Image search provided by Google, book text from Amazon, movie information from the internet movie database, and reference information such as encyclopedia, dictionary etc., through GuruNet.com. In addition to regular search capabilities, A9 also offers a set of additional features like saving of the search results that is fairly new has been incorporated in A9. So, A9.COM can be called as a search engine with a memory. A9 offers tools that personalize the users search experience such as 1) History: it allows recording a trail of all sites that the user has visited and this trail can be searched anytime. 2) Bookmarks: user is provided with server-side bookmarks that are always available and searchable from anywhere. 3) Diary: user is provided with an option to create notes while browsing the Web and it is stored which can be searched later. This adds user customization to the search results.

2.4 Multilevel Matching

The term "Multilevel" refers to a hierarchical or nested data structure, usually individuals within organizational groups, but the nesting may also consist of repeated
measurements on individuals over time or individuals within clusters as in cluster sampling [HOX02]. The expression Multilevel model or Multilevel analysis is used as a generic term for all models for hierarchical or nested data. This is a useful search process as it narrows down the results from a wide set of results. Many search engines such as Google, Alta Vista, HotBot and Lycos offer this feature.

Multilevel matching is one of the desirable features in the component discovery service. Multilevel matching matches the components specifications at multiple levels such as syntactic, semantic, synchronization and the QoS characteristics. This helps in improving the matching of the query as the component can be matched at four levels given in the specification giving flexibility to the user. Since the motivation for this project is to make the selection process in URDS more comprehensive, a prototype is designed and developed based on the Multilevel matching criterion. Below is the explanation of some of the applications which use Multilevel matching.

2.4.1.1 GESTALTS

GESTALT RDS [DON01] was designed to improve the resource discovery over the Internet in order to help customers locate educational courses and resources. This discovery service offers facilities such as searching, previewing and ordering educational courses. Three different types of services are offered to the user 1) Text-Based search, 2) Structured search and 3) Complex search. The set of requirements proposed on the search to satisfy the user requirements are the search utility should be fast and easy to use, the search result must provide all the information needed by the user, the authenticity and quality of product and supplier must be confirmed and the search should also indicate to the user if the product is available or not. The system also maintains the user profile along with the proposed set of requirements.

2.4.1.2. Multi-level Immune Learning Detection (MILD)

Multi-level Immune Learning Detection (MILD) [MILD] is an anomaly detection tool for spacecraft subsystems that is based on the negative selection algorithm inspired
by the human immune system. MILD is a software tool, which implements an immunity-based technique for anomaly and fault detection. Since spacecraft subsystems are highly critical the detection of anomalies is really important for safe operation. MILD uses an immunity-based approach that can detect both known and unknown anomalies. The approach is based on the concept of negative space, i.e. looking for behaviors outside of what is defined as normal.

2.4.1.3. Spam Trapper for Outlook 1.3.3

Spam Trapper (ST) is a multilevel spam filter for Microsoft Outlook version 2000 and later. This is a plug-in component with a convenient interface. The first level of ST checks the address book and validates all the clients. The other levels check for message language, subject, body, http links and other mail feature. ST has special statistical levels that are based on unique algorithm; the last layer is Bayes Filter. The architecture of ST helps catching spam even before the last layer i.e. Bayes Filter has to be trained reducing the filter-training period. Any level can be switched off.

URDS introduces the concept of multilevel matching of software components. The component specifications are considered for matching at different levels. This matching aims at providing a better selection of relevant components rather than a large number of components obtained in the normal approach. Below is the explanation of this approach in URDS.

2.4.2 Extending Multilevel Matching in URDS

This project focuses on presenting the advantages of multilevel approach when applied to URDS through a prototype, while selecting the components available over the
network. The advantage of this approach when applied to URDS reflects in the improvement of quality of discovered components.

The UMM which is the central part of Uniframe contains the specification of different components offering services on the network. It contains information regarding the multilevel contract of the component. The multiple levels in UMM specification are the syntactic, the semantic, the synchronization and the QoS of the component. When a query is given for a component search, the information regarding the components in UMM is used by the multilevel matching approach. The Uniframe multilevel matching uses different matching level operators for each one the levels. As the multilevel matching is a more comprehensive matching, fewer components that are more relevant are obtained than a general matching. There are four levels of matching in URDS, the syntactic level matching, the semantic level matching, the synchronization level matching and the QoS level matching. The different levels of matching as per the Uniframe Approach is indicated below:

1. Syntactic Level Matching: This matches the function signature of available components against the required query. Two categories of matches can be implemented here the exact match and the relaxed match. Exact match requires the types in the signature of the query and the available services, to be exactly equivalent or related by coercion or inheritance. A relaxed match allows the order of components to differ even though the types of arguments should still be related.

2. Semantic Level Matching: This match uses the pre- and post-conditions of the functions indicated in the syntactic contract. The pre- and post-conditions of the candidate components are matched against the pre- and post-conditions of the query requirement. The notion of Exact and Relaxed matches is extended to the semantic matching as well. The Exact match is fulfilled if the pre- and post-conditions of the query are exactly satisfied as specified. If the pre- and post-conditions of the candidate component matches a subset of pre- and post-condition requirements of query then it is considered as Relaxed match.

The concept of syntactic and the semantic level matching are being taken from the matching levels described by [ZAR96].
3. Synchronization Level Matching: This is the third level of matching which is reached if the above two levels have been satisfied. Synchronization level matching defines the dependencies between the services offered by the components. This level matches the synchronization policies and the implementation technique used by the components against the required query component. Exact and Relaxed matches are also considered here. If the component matches with the required components synchronization policy and implementation exactly then it is a required match. In Relaxed match the synchronization policies and implementing techniques of the candidate components provided are the subset of the required query component.

4. QoS level Matching: This is the fourth level in the matching of components. The component that reaches this level are already matched with either exact/relaxed at the above three levels. This is a non-functional aspect of the component like throughput, end-to-end delay etc. The comparisons done here are numerical comparisons. For Exact match the candidate component must provide the Exact QoS attribute as required. In relaxed a certain deviation in the returned and expected values is accepted.

This project focuses on validating the Multilevel matching criterion using a prototype with an intention of incorporating the same concept in “Live” URDS. It tries to assess the merits of Multilevel matching as opposed to single level matching. The coming chapters provide the details about the prototype. Next chapter introduces the previous work done as a part of Uniframe project, the overview of Uniframe and URDS and different previous works done to propose the matching levels that are used to implement the ideas of this project.
3. Previous Work

This chapter outlines the previous work done as part of UniFrame Project and also provides a discussion of the background and few of the related research works of the thesis. The first section 3.1 starts with an overview of UniFrame and the following sections give an explanation of Unified Meta-Component Model (UMM) and the UniFrame Approach. The second section 3.2 focuses on the need for contracts for software components. The section 3.3 focuses on the previous work done on the signature and specification matching of the components. The fourth section 3.4 focuses on previous work done as part of synchronization and QoS matching of components.

3.1 Overview of UniFrame

The main objective of UniFrame is to allow a seamless integration of heterogeneous and distributed software components. It provides an efficient and effective framework for the development of high confidence DCS. UniFrame incorporates the principles of distributed, component-based computing, Model-Driven Architecture, service and quality of service guarantees, and the generative techniques. It provides a framework for the component developers to create, test and verify QoS. The component developers can deploy these components and the application developers can select the components generating a DCS in an automatic and semi-automatic fashion. UniFrame supports the following features [RAJ05]:

- The creation of a standard-based meta-model for components and an associated hierarchical setup to indicate the contracts and the constraints of the components.
- Interoperability achieved using automatic generation of glue and wrappers
- Specifies guidelines for specifying and verifying the quality of individual components
- Functionality to automatically discover components on the network
• Supports development of distributed component based systems with service-oriented architecture
• Mechanisms to evaluate the quality of the resulting component

The UniFrame consists of the Unified Meta-Component Model (UMM) and the UniFrame Approach (UA). The UA is a component based software engineering process, which is based on UMM for creating a DCS from the available heterogeneous components in an automatic or semi-automatic fashion. The UMM is the central concept of the UniFrame. A detailed description of the UniFrame Approach is provided in [RAJ00], [RAJ01] and [RAJ05].

3.1.1 Unified Meta-Component Model (UMM)

UMM is a meta-model that abstracts and enhances the features of different models, incorporates innovative concepts in order to facilitate the creation of service-oriented systems. It mainly consists of three parts 1) components 2) service and service guarantees and 3) the infrastructure.

3.1.1.1 Components

Components in Uniframe are autonomous entities, whose implementations are non-uniform, i.e., each component adheres to some distributed-object model and there is no notion of either a centralized controller or a unified implementational framework [RAJ00]. Each component has a state, an identity, a behavior, a well-defined public interfaces and a private implementation. In addition, each component in UMM has three aspects: 1) Computational Aspect, 2) Cooperative Aspect, and 3) Auxiliary aspect.

Computational Aspect: The computational aspect indicates the tasks carried out by each component. It depends on the objectives of the task; the techniques used out achieve these objectives and the specification of the functionality offered by the component. The computational aspect of the component is defined by its inherent attributes such as
bookkeeping information about the component such as the author, the version, etc and the functional attributes of the component contain its interfaces with necessary pre- and post-conditions, algorithms used, nature of computation, its associated contracts and level of service offered by the component.

Cooperative Aspect: The cooperative aspect of the component indicates the interaction with other components. In a Distributed Computing System (DS) in order to achieve a certain task the components have to associate with other components, delegate sub-tasks to them and coordinate their activities. This aspect depends on many factors such as detection of other components, cost of service, inter-object negotiations, aggregations etc. The cooperative aspect of a component may contain expected collaborators (potential components that may interact with this component), pre-processing collaborators (other components on which this component depends) and post-processing collaborators (other components which depend on this component).

Auxiliary Aspect: This aspect represents the mobility, security and fault tolerance features of an open DS. For example, the mobility of an attribute will contain the necessary information, such as its implementation details and the execution environment. Similarly, the security attribute of the component will contain information about its security features. Similar to the mobility and security each component in UMM contains the fault-tolerant attribute.

3.1.1.2 Service and Service Guarantees

A service in UniFrame is a computational effort or an access to underlying resource. This part of UMM consists of the computational tasks and guarantees that a component performs. In DCS, there can be several choices to obtain a specific service. Thus each component, in addition to specifying its functionality, must also specify the cost and the quality of services offered. The service offered by each component depends on the algorithms implemented by the component, the expected computational effort and the resources required. The cost of each service is determined by the motivation of the
owner and the demand and supply. The quality of service indicates the confidence to carry out the required services in a constantly changing environment and in an environment of partial failures. The service offered by each component is divided mainly into three categories 1) the syntactic and quality features are represented as a part of computational aspect, 2) the behavior and the synchronization are incorporated into the cooperative aspect and 3) any other special features like fault-tolerance are indicated in the auxiliary aspect. These are the features each component publishes and exports to the concerned seekers.

### 3.1.1.3 Infrastructure

UniFrame infrastructure consists of the System Generation Process, Resource Discovery Service (URDS), and Glue and Wrapper Generator. The first employs a knowledgebase to carry out the steps in creating a component system. It relies on the URDS to provide the necessary infrastructure to discover components as well as to select and integrate components that adhere to different component models in order to develop a distributed computing system. URDS requires the components to publish UMM specifications for hosting the services on distributed systems, receives queries for locating the deployed components and selects the appropriate components based on the criteria. URDS invokes the Glue and Wrapper Generator, which accommodates the heterogeneity across the components, configures the selected services and incorporates mechanisms that are needed to measure their resulting QoS. Below is the detailed explanation of URDS, which is the central part of the UniFrame.

#### 3.1.1.3.1 UniFrame Resource Discovery Service (URDS)

The URDS architecture is organized as a federated hierarchy in order to achieve scalability. URDS is designed as a Discovery Service to dynamically discover new services and to provide clients a Directory Service style to access the services. It is arranged in a federation of Internet Component Brokers (ICB) and entities such as Headhunters, Active Registries, Services and Adapter Components. The ICBs are linked
together by Link Managers (LM) with unidirectional links to form a directed graph. Each ICB has zero or more entities called Headhunters responsible for discovering components attached to it. The discovery process in URDS is administratively scoped, i.e. it locates services within an administratively defined logical domain. In UniFrame, a domain is defined as industry specific to markets such as Health Care Services, Manufacturing Services, and Financial Services. Figure 3.1, taken from [NAN02], below represents the federated hierarchical organization of ICBs.

Figure 3.1: Federated Hierarchical Organization of ICBs (Source [NAN02])

URDS infrastructure in Figure 3.2 taken from [NAN02] consists of the following entities to carry out their specified tasks:
i) Internet Component Broker (ICB) which is a collection of services: Query Manager (QM), the Domain Security Manager (DSM), Link Manager (LM) and Adapter Manager (AM).

ii) Headhunters (HHs)

iii) Active Registries (ARs)

iv) Services ($S_1$…$S_n$)

v) Adapter Components ($AC_1$…$AC_n$)

vi) Clients ($C_1$…$C_n$)

Figure 3.2: URDS Architecture (Source [NAN02])

The explanation of the components is given below:
i) Internet Component Broker (ICB)

The ICB acts as an all-pervasive component broker in an interconnected environment. It supports the communication infrastructure necessary to identify and locate services, enforce domain security and handle mediation between heterogeneous components. All these services are reachable at well-known addresses. ICBs provide the following functionalities:

- It performs functions of a conventional broker and also authenticates the principles of the system Headhunters and Active Registries
- Cooperates with other ICBs deployed on the network to provide the required components and provide matchmaking between service producers and consumers.
- Acts as a mediator between two components that belong to different component models

It consists of Domain Security Manager, Query Manager, Link Manager, and the Adapter Manager. Description of these is given below

i.a) Domain Security Manager (DSM)

The DSM performs as an authorized third party that handles the secret key generation and distribution. It also enforces group membership and access controls to multicast resources through authentication and use of access control lists (ACL). DSM has an associated repository (database) of valid users, passwords, multicast address resources and domains.

i.b) Query Manager (QM)

The purpose of QM is to translate a system integrator’s natural language like query into a structured query language statement and dispatch the query to the HHs available in the domain. The HHs then return a list of service provider components matching the search criteria expressed in the query. The query request is propagated only to the HHs available in that domain. The QM, in conjunction with LM, is also responsible for propagating the queries to other linked ICBs.
i.e) Link Manager (LM)

The LM serves the purpose of establishing the links with other ICBs for the purpose of federation and to propagate queries received from the QM to the linked ICBs. ICB administrator configures Link Manager LM with the location information of other ICBs with which links are to be established.

i.d) Adaptive Manager (AM)

The AM serves as a Lookup service for clients seeking adapter components. The adapter components register with the AM and also indicate their specialization, i.e., with which component models they can bridge effectively. Clients contact the AM to search for adapter components matching their needs.

ii) Headhunter (HH)

The HH is equivalent to a binder or trader in other models. Unlike trader, the responsibility of registering components lies with the headhunter and not on the components themselves. HH performs the following tasks: a) Service Discovery: detect the presence of service providers (Exporters) b) register the functionality of these service providers, and c) return the list of service providers to the ICB which match the query requirement forwarded by the QM. The service discovery process performs the search using multicasting. The HH stores the UniFrame specification information of exporters adhering to different models in Meta Repositories. These repositories are implemented as standard relational databases.

iii) Active Registry (AR)

The native registries/lookup services of various component models (RMI, CORBA, .NET, etc.) are extended to be able to listen and respond to multicast messages from the HHs and also have introspection capabilities to discover not only the instances, but also the specification of the components registered with them.

iv) Services ($S_1...S_n$)
Services (Components) are implemented in different components models (RMI, CORBA, etc.). They are identified by the service type name, components informal UniFrame specification and QoS values for that service. The informal specification is a XML specification outlining the computational, functional, and co-operational and auxiliary attributes of the component. The interfaces of the component are registered with its local registry. One method in the interface returns the URL of its informal specification. Each component is specific to its domain, i.e., it adheres to the standards of the domain.

v) Adapter Components (AC\textsubscript{1}…AC\textsubscript{n})
Adapter components serve as bridges between components implemented in the diverse models, say for instance RMI and Corba.

vi) Clients (C\textsubscript{1}…C\textsubscript{n})
Clients may be Component Assemblers, System Integrators or System Developers who may be searching for services matching certain functional and non-functional requirements.

### 3.1.3.2 Overview of Resource Discovery Process in URDS

The resource discovery process in URDS can be classified into two main categories: 1) updating the meta-repository with the component information which happens periodically and 2) the discovery of components whenever a query is specified. In URDS, Domain Security manager (DSM) is the entity that authenticates the principals participating in the resource discovery process. When the Headhunters (HHs) or the Active Registries (ARs) contact DSM with their credentials, the DSM authenticates the principal by checking its authorization against the domain Access Control Lists (ACL). The DSM then returns to the principal, a secret key and the multicast address mapped to the corresponding domain. If the principal is HH, DSM registers its contact information that it later passes to the Query Manager (QM). QM uses this information to propagate queries.
QM generates a structured query language statement when the query is issued. QM gets the list of HHs from DSM and propagates the query to all HHs in that particular domain. Each HH responds to QM by returning the list of service providers present in its meta-repository that match the query requirement. The Internet Component Brokers (ICB) propagates the search query issued by Clients (System Integrator) to other ICBs to which they are linked apart from the HHs. Link Manager (LM) performs the functions of ICB associated with establishing the links and propagating the queries. Links represent the path for the propagation of queries from one ICB to a target ICB.

Once authenticated, the HHs periodically multicast their addresses which were issued by DSM. The Active Registries (ARs) that listen to the multicast messages from HHs at a group address respond back by passing their contact information to the HH. HHs then query the responded ARs to get the components UniFrame specification information registered with them. The ARs respond back with the list of matching components UniFrame Specification information registered with them. The HHs stores the component information in their respective meta-repositories. The updating of component information between HHs and ARs happens periodically so that the new components registered with the ARs are reflected in the meta-repositories of HHs. The component information stored in the meta-repositories of HHs is used when being queried by QM or other HHs. The matching of components can be both with respect to functional or non-functional attributes. The matched components are then returned to the system developer, who selects the components from the matching subset to form the system. If the selected components are heterogeneous, then the Adapter Manager (AM) is contacted to search for appropriate adapter components that match the needs. The AM checks against its repository for matches and returns the results, which are used to build the system. With all the required components to form a system, a DCS is formed.

3.1.2 The Uniframe Approach

The UniFrame Approach (UA) is the UMM-based technique for the automatic production of a DCS. The UA specifies two levels for the creation of a DCS:
1. The component level- In this level, the components are designed and developed with UMM specifications. These components are tested and verified against the appropriate QoS and are then deployed on to the network, and

2. The system level- The system level supports integrators/application programmers to select and generate an automatic or semi-automatic generation of a specific DCS to the maximum possible extent.

### 3.1.2.1 The UMM Specification

The UMM specifications are informally indicated in a natural language like style. The specification indicates factors such as computational, cooperative, auxiliary attributes and QoS metrics of the component. The component developer who wish to agree to and adopt the UniFrame should follow the UMM specification guidelines in designing and development of the component. A sample UMM specification is provided below in the Figure 3.3. The example UMM specification describes the abstract component CoordinationServer of the Bank System considered as part of the case study.
Abstract Component: *CoordinationServer*

1. Component Name: *CoordinationServer*
2. Component Subcase: *CoordinationServerCase*
3. Domain Name: Bank
4. System Name: Bank System
5. Informal Description: Coordinates operations between the users and servers in the Bank System.

6. Computational Attributes:
   6.1 Inherent Attributes:
      - 6.1.1 id: N/A
      - 6.1.2 Version: version 1.0
      - 6.1.3 Author: N/A
      - 6.1.4 Date: N/A
      - 6.1.5 Validity: N/A
      - 6.1.6 Atomicity: Yes
      - 6.1.7 Registration: N/A
      - 6.1.8 Model: N/A
   6.2 Functional Attributes:
      - 6.2.1 Function description: Stores the general account info and coordinates requests to other servers in the system
      - 6.2.2 Algorithm: N/A
      - 6.2.3 Complexity: N/A
      - 6.2.4 Syntactic Contract
      - 6.2.4.1 Provided Interface: *IAddAccount, IAccountInfo*
      - 6.2.4.2 Required Interface: *IAddAccount, IAccountInfo, IAccountGeneralInfo, IAccountOperation*
      - 6.2.5 Technology: N/A
      - 6.2.6 Expected Resources: N/A
      - 6.2.7 Design Patterns: NONE
      - 6.2.8 Known Usage: NONE
      - 6.2.9 Alias: NONE

7. Cooperation Attributes
   7.1 Preprocessing Collaborators: *UserTerminalCase*
   7.2 Postprocessing Collaborators: *AccountGeneralInfoCase, AccountOperationsCase*

8. Auxiliary Attributes:
   - 8.1 Mobility: No
   - 8.2 Security
   - 8.3 Fault tolerance: L0

9. Quality of Service
   - 9.1 QoS Metrics: throughput, end-to-end delay
   - 9.2 QoS Level: N/A
   - 9.3 Cost: N/A
   - 9.4 Quality Level: N/A

Figure 3.3: Abstract Component Specification for the Coordination Server

The component developer is responsible for specifying the parameters in the UMM specification during the development and deployment of the component. So, a formal notation of the specification is important for developers to access the computational, cooperative, auxiliary attributes and QoS parameters of the component. The formal
notation will also be helpful in searching for a component with a certain requirement and as well as for enabling effective and efficient matching of the specifications against the developer’s requirement.

The next part of this chapter focuses on the contracts for software components, the need for contracts, the four classes of contracts, the representation of contracts in URDS, the research work done so far regarding the matching of software components both by [ZAR96] and [KUM04] and the further work done as part of this project.

3.2 Contracts for Software Components

This section introduces the concept of contracts in the software components, the different levels of contracts specified in the software components, the matching category of the components and the related work done in this context.

3.2.1 Need for Contracts

The main challenge faced in the development of a DCS, is to build a correct and robust distributed system using heterogeneous components. A correct software system performs according to the proposed specifications and a robust software system handles the abnormal situations that arise in the system. For reusability of software components, Component Based Software Development (CBSD) is an ideal approach for the development of DCS. Since the reliability and reusability of the components depends on the trust the user/application developer has on the correctness of the component, developing the software systems based on CBSD is a major issue. A systematic approach for specifying and implementing software components is needed for an effective and efficient distributed system.

Meyer [MEY92] has introduced a theory of Design By Contract where a software system is viewed as a set of communicating components whose interaction is based on precisely defined specifications of the mutual obligations, called Contracts [ISE01].
The Design by Contract [ISE01] provides the benefits such as a better understanding of the component-based software development method, a systematic approach to building bug-free DCS, an effective framework for debugging, testing and quality assurance, a method for documenting software components, better understanding and control of the inheritance mechanism, a technique to deal with abnormal cases leading to a safe and effective language construct for exception handling.

The Design by Contract forms the basis for the making components contract aware by [BEU99], in which the authors raise the issue of trusting a software component on its use and reusability in mission-critical applications. If the component behaves in an unexpected way, rebooting of such critical systems would be a huge task. So, it will be helpful if the application developer can find out if the component being used is the right one for the requirement. It would be useful if the component being considered can specify what it can offer by providing a specification of itself. The specification should also provide parameters against which the component can be verified and validated; this leads to a Contract between the component and its users.

[BEU99] talks about four levels of contract in the context of software components:

- Syntax Level Contract: This is the first level of contract, it specifies the operations that a component has to perform, input and output parameters of the operation and the possible exceptions that might be raised during the execution of the operations. Static and dynamic type checking are an important part of the syntax level contract. This contract is considered as the basic contract and non-negotiable.

- Semantic Level Contract: This is the second level of contract also called as the behavioral contract, it describes the mutual obligations and benefits between the software components. It also describes any consistency conditions that could go wrong during the execution of the operations and explicitly assigns the responsibility of its enforcement. The behavioral property of component can be represented using Boolean assertions, pre and post conditions.

- Synchronization Level Contract: The behavioral contract assumes that the services are atomic or executed as transactions, which may not be true in all
applications. The next level synchronization contract is defined to describe the dependencies between the services provided by the component such as sequence, parallelism etc. Synchronization contract is mainly needed in a single server-multiple client scenario, where each client’s requested service is guaranteed irrespective of the other client’s requests.

- Quantitative Level Contract: The first three levels of the contract are related to the functional attributes of the component, the Quantitative level relates to the non-functional attributes of the component. This contract specifies the QoS attributes of the components such as Turn Around Time, Throughput, Latency, etc.

The contracts specified above start from non-negotiable syntactic level at the bottom of the hierarchy and progress in hierarchical order from semantic, synchronization to more negotiable and dynamic quantitative level.

The concept of multilevel contracts has also been adapted in URDS. Below is the representation of contracts in URDS, based on the Quality of Service catalogue [UMM02]. An example specification of the Coordination component that is considered as a part of the case study representing contracts at each level is given below in Figure 3.4:

Abstract Component: CoordinationServer

1. Component Name: CoordinationServer
2. Component Subcase: CoordiantionServerCase
3. Domain Name: Bank
4. System Name: Bank System
5. Informal Description: Coordinates the operations between the users and the servers in Bank System
6. Computational Attributes:
   6.1 Inherent Attributes:
      6.1.1 id: N/A
      6.1.2 Version: version 1.0
      6.1.3 Author: N/A
      6.1.4 Date: N/A
6.1.5 Validity: N/A

6.1.6 Atomicity: Yes
6.1.7 Registration: N/A
6.1.8 Model: N/A

7. Deployment Attributes: N/A

I & II. Syntactic Contract & Semantic Contract

1. Functional Attributes
   1.1 Function Description: Stores the general account info and coordinates requests to other servers in the system
   1.2 Algorithm: N/A
   1.3 Complexity: N/A
   1.4 Interface Contract
      1.4.1 Provided Interface: IAddAccount, IAccountInfo
      1.4.2 Required Interface: IAddAccount, IAccountInfo, IAccountGeneralInfo, IAccountOperation
   1.5 Technology: N/A
   1.6 Expected Resources: N/A
   1.7 Design Patterns: NONE
   1.8 Known Usage: NONE
   1.9 Alias: NONE

2. Coorperation Attributes:
   2.1 Preprocessing Collaborators: UserTerminalCase
   2.2 Postprocessing Collaborators: AccountGeneralInfoCase, AccountOperationsCase

3. Auxiliary Attributes:
   3.1 Mobility: No
   3.2 Security: L0
   3.3 Fault tolerance: L0

III. Synchronization Contract

1. Synchronization policy: Mutual Exclusion
2. Synchronization Implementation: Mutexes

IV. Quality of Service Contract

1. QoS Metrics: throughput, end-to-end delay
2. QoS Level: N/A
3. Cost: N/A
4. Quality Level: N/A
5. Effect of Environment: N/A
6. Effect of Usage Pattern: N/A

Figure 3.4: A UMM Specification With Contracts of Coordination Server

Below are the details of the work done so far as part of representation of contracts in URDS by [KUM04] and the origin for the work taken from [ZAR96]. [ZAR96] is the basis for matching of software specifications at the syntactic level and the semantic level and [KUM04] extended the matching levels to synchronization and QoS levels of the components.

3.3 Signature and Specification Matching of Components

The research described in Signature and Specification Matching [ZAR96] presents a way to retrieve components from a library, index a library and compare two components to help realize the potential of software components even further. The thesis uses the semantic information of the components and focuses on what the component does rather than how it does it. The semantic information of the component describes both the behavior as well as the specification of the component. Behavior describes the type information of a component and the specification describes the dynamic behavior of the component. [ZAR96] defines matches for the semantic abstracts with an assumption that components are stored in a library and each component in a library has a signature and a specification. A component’s behavior plays an important role during the retrieval, indexing, navigation, substitution, sub-typing, and modification, and so, the matching of
their specifications is important. [ZAR96], proposes a comprehensive set of matches for matching component specifications.

Three parameters of the components are taken into consideration for the matches

1. The kind of the information used to describe the components. Components can be described in a variety of forms from textual description, a control- or data-flow graphs to the signatures to the semantic information of the operations of the components.

2. The granularity of components. Components vary in size from individual language constructs to moderately sized blocks of code to large software systems.

3. The degree of relaxation of the match. For each kind an Exact and Relaxed set of matches are proposed.

A brief overview of the matches is provided below:

### 3.3.1 Function Signature Matching

A set of matches are proposed to match the function signature of components. Function matching based on signature information considers only matching of function types which is the syntax level of the component. The degree of match is categorized into two types, exact and relaxed. For the relaxed match, different scenarios are considered like the reordering of elements in a tuple, uncurrying of arguments to a function, renaming of type constructors and instantiation of type variables.

The following are the categories of matches defined by [ZAR96] for the Syntactic Level Match:

1. Exact Match:

   For an exact match, the function signature match verifies if a transformation can be applied to a signature so that the results are equal.

   Match\(_E\) \((t_1, t_q)\) = There exists a sequence of variable renamings, \(V\), such that \(V t_1 = t_q\)

   Where \(t_1, t_q\) are the type of the component in the library and type of the query respectively. It suggests that two matches will match exactly if they match modulo renaming.
2. Relaxed match: this match is obtained by applying transformations or variable substitutions to the type expressions. Below are the representations:

   a. Transformation Relaxation: In this match, a transformation is applied to type constructors. It includes transforming a type expression, for example renaming type constructors, changing whether a function is curried or uncurried, changing the order of types in the tuple and changing the order of arguments of a function. Below is the representation of the matches:

   1. Type Constructor Match: This match renames the user-defined constructors and built in types with a different name.
   
   \[
   \text{Match}_{\text{TyCon}} \left( t_\text{i}, t_\text{q} \right) = \text{There exists a sequence of type constructor renamings } V_{TC}, \text{ such that Match}_{E} \left( V_{TC} t_\text{i}, t_\text{q} \right)
   \]

   2. Uncurrying Functions: A function that takes multiple arguments may be either curried or non-curried. The uncurried function has a type \((t_1 \ldots t_{n-1}) -> t_1\) and the curried version has a type \(t_1 -> \ldots -> t_{n-1} -> t_n\). Uncurried match is defined as applying Uncurry transformation to both query and component. Uncurry transformation produces a curried transformation of given type. It is defined as
   
   \[
   \text{Match}_{\text{Uncurry}} \left( t_\text{i}, t_\text{q} \right) = \text{Match}_{E} \left( \text{UC} \left( t_\text{i} \right), \text{UC} \left( t_\text{q} \right) \right)
   \]
   
   where, \(\text{UC}\) is the Uncurry transformation. Two uncurried function types are checked to determine if their arguments match.

   3. Reordering Tuples: Tuples group multiple arguments into a function, but sometimes the order of arguments does not matter. So this match is allowed on types that differ only in their order of arguments. A Reorder transformation, \(T \sigma\) for a method whose argument is a tuple \((t = (t_1 \ldots t_{n-1}) -> t_n)\), can be defined as a permutation \(\sigma\), which can be applied to the tuple \(T \sigma\) is bijection with domain and range \(1 \ldots n-1\) such that \(T \sigma \left( t \right) = (t_{\sigma \left( 1 \right)} \ldots t_{\sigma \left( n-1 \right)}) -> t_n\). A reorder match is defined as
   
   \[
   \text{Match}_{\text{Reorder}} \left( t_\text{i}, t_\text{q} \right) = \text{there exists a reorder transformation } T \sigma \text{ such that } \text{Match}_{\text{Reorder}} \left( T \sigma \left( t_\text{i} \right), t_\text{q} \right)
   \]
b. Partial relaxations: Partial relaxation matches involve finding a partial order relationship between the types and transformation relaxation. This match arises to allow more general types to match a query type. If a specific query is an instantiation of a more general function, or if the user requirement does not match with any of the components, then this match is considered. Variable substitutions are done on the partial ordering to achieve the partial relaxations. For example, $\alpha \rightarrow \alpha$ is a generalization of infinitely many types such as int $\rightarrow$ int and (int*β) $\rightarrow$ (int*β), using the variable substitutions [int/α] and [(int*β)/α], respectively. Different types of matches that are defined are

1. Generalized match: It can be defined as
   \[ \text{Match}_\text{Gen}(t_l, t_q) = t_l \geq t_q \]
   A component type matches with the query type if it is a more general than the query type. Exact Match, with variable renaming is a special case of the generalized match where all variable substitutions are variable renamings.

2. Specialized match: It is defined as
   \[ \text{Match}_\text{Spec}(t_l, t_q) = t_l \leq t_q \]
   A component type matches with the query type if the query type is more general than component type.

3.3.2 Function Specification Matching

In function specification matching, the matches are defined in terms of a logical relationship, such as equivalence, implication between two specifications or between parts of specifications. Different categories of matches considered here are based on the pre and the post conditions, and the specification predicates. These categories of matches are further divided into exact and relaxed matches. A match in the function signature is considered exact pre/post if the relationship between the pre and the post conditions is Equivalence. Similarly, for the predicates if the relationship is equivalence it is named as exact predicate match. For relaxed matches, the relation between the pre and the post as well as the predicates is represented as implication and reverse implication. Below is the explanation of matches at the Semantic Level:
For a function specification, $S$, and a query $Q$, the pr-conditions, $S_{\text{pre}}$ and $Q_{\text{pre}}$ and the post-conditions are $S_{\text{post}}$ and $Q_{\text{post}}$, respectively. A generic pre/post match predicate is represented as,

$$\text{Match}_{\text{pre/post}}(S, Q) = (Q_{\text{pre}} R_1 S_{\text{pre}}) R_2 (S_{\text{post}} R_3 Q_{\text{post}})$$

The relations $R_1$ and $R_3$ relate the pre and post conditions respectively and the relations can be either equivalence ($\Leftrightarrow$) or implication ($\Rightarrow$). The relation $R_2$ is usually conjunction ($\land$) and can also be a ($\Rightarrow$). Below are the exact and the relaxed matches for the above match:

1. Exact Pre/Post Match: This is a strict relation with $R_1$ and $R_3$ replaced with the equivalence relation and $R_2$ is replaced with conjunction relationship. The representation of this match is given below:

$$\text{Match}_{\text{e-pre/post}}(S, Q) = (Q_{\text{pre}} \Leftrightarrow S_{\text{pre}}) \land (S_{\text{post}} \Leftrightarrow Q_{\text{post}})$$

2. Plug-in Match: For this match both the relations $R_1$ and $R_3$ are implications and $R_2$ is a conjunction. The representation of this match is given below

$$\text{Match}_{\text{plug-pre/post}}(S, Q) = (Q_{\text{pre}} \Rightarrow S_{\text{pre}}) \land (S_{\text{post}} \Rightarrow Q_{\text{post}})$$

3. Plug-in Post Match: For this match the plug in match is considered only for post condition of the method. These are the cases where pre-conditions can be satisfied by adding an additional check.

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

4. Weak Post Match: This is the weaker match where only components pre-condition is considered. This is a case that is used when component’s precondition helps in proving the relationship between the post conditions.

$$\text{Match}_{\text{plug-in-post}}(S, Q) = S_{\text{pre}} \land (S_{\text{post}} \Rightarrow Q_{\text{post}})$$
[ZAR96] provides properties of the matches at this level and the matches are classified on the basis of whether they are equivalence matches, partial order matches or neither.

Both these levels of matches specify various aspects of the components, which the application developer can use before actually using the component. The implementation language used is ML and Larch/ML is used as the specification language. Larch/ML is a Larch interface language for Modula and provides a mechanism to specify the syntax of components operations and behavior of the component.

3.4 Synchronization and QoS Matching for Components

[ZAR96], is taken as the basis for the work done by [KUM04] to extend the matching for the Synchronization and the QoS levels. Below is the description of matches provided at these levels.

3.4.1 Synchronization Policy Matches

Synchronization plays an important part in software components. In order to handle multiple request calls being made on the methods supported by the components, they need to be synchronized. It defines the dependencies between the methods or services supported by the components, such as sequence, parallelism, or shuffle. Matching of components at the synchronization level is important, as components may not always behave as expected by the software developers. The synchronization level of matching is divided into two main categories:

- Matching of behaviors of the synchronization policies: The synchronization policies of the required and the provided components are matched here. The relation between these policies can be either Equivalence or Implication.
- Matching of the synchronization behavior of the components interfaces: this is the case when components can utilize more than one synchronization policy for protecting different resources within the component. It helps to find out how the system’s synchronization behavior changes with replacing other components. Two types of checks with respect to synchronization policy are defined. First,
substitutability of the software components, which is defined as the ability of two components to replace each other without changing the synchronization behavior of the system that is formed using that component. Compatibility of the software components is the ability of two components to interoperate and synchronize properly when brought together to form a system. Below is the representation of matches with respect to the synchronization policy

### 3.4.1.1 Generalized and Specialized Matches

The matching of the components synchronization behavior with that of a query component is divided into two main categories:

1. Generalized Match: This match represents the synchronization policy of the components and the synchronization behavior of the components interface. A generic match for the synchronization behavior of the component is represented below:

   \[
   \text{Match} \text{ gen} (C, QC) = \land (\text{SP} R_1 \text{SP}_{QC}) \\
   \land \text{For each method } M_i \text{ and } M_{QC_i} \text{ (SP}_{pre} (M_i) R_2 \\
   \text{SP}_{pre} (M_{QC_i})) R_3 \text{ (SP}_{post} (M_i) R_4 \text{SP}_{post} (M_{QC_i}))
   \]

   Where SP and SP\text{ QC} are the synchronization policies utilized by the component C and the query component QC. SP\text{ pre} (M_i) and SP\text{ post} (M_i) are the synchronization pre and post conditions of the methods. SP\text{ pre} (M_{QC_i}) and SP\text{ post} (M_{QC_i}) are the pre and post conditions of the query component. This generalized match can be further divided into exact and relaxed matches.

1.1 Exact Match: In exact match, there exists an equivalence relationship between the two synchronization policies of the component and the synchronization pre and post conditions of the methods. Representation of it is given below:

   \[
   \text{ExactMatch} \text{ gen} (C_1, QC) = \land (\text{SP} \equiv \text{SP}_{QC})
   \]
\[ \text{For each method } M_i \text{ and } M_{QC}, (SP_{pre} (M_i) \leftrightarrow SP_{pre} (M_{QC})) \land (SP_{post} (M_i) \leftrightarrow SP_{post} (M_{QC})) \]

1.2 Relaxed Match: In relaxed match, there exists an implication relationship between the synchronization policies of the components and the synchronization pre and post conditions of the methods of the components. Representing the relaxed match below:

\[
\text{RelaxedMatch}_{\text{gen}} (C, C_{QC}) = \land (SP \Rightarrow SP_{QC}) \land \text{For each method } M_i \text{ and } M_{QC}, \\
(SP_{pre} (M_i) \Rightarrow SP_{pre} (M_{QC})) \land \\
(SP_{post} (M_i) \Rightarrow SP_{post} (M_{QC}))
\]

The above Generalized match considers only the synchronization policy and the pre and post conditions of the synchronization policy of the method. In the specialized match given below, the implementation technique of the synchronization policy is also included.

2. Specialized Match: A specialized match consists of matching the synchronization policy utilized by the components, the synchronization implementation technique of the methods and the pre/post conditions of the synchronization policies of the method. The representation of the specialized match is given below:

\[
\text{Match}_{\text{spec}} (C, C_{QC}) = \land (SP R_1 SP_{QC}) \land \land (SP_{impl} R_2 SP_{QCimpl}) \land \text{For each method } M_i \text{ and } M_{QC}, (SP_{pre} (M_i) R_3 \land \\
SP_{pre} (M_{QC})) R_4 (SP_{post} (M_i) R_5 SP_{post} (M_{QC}))
\]

Where SP and SP_{QC} are the synchronization policies utilized by the component C and the query component QC. SP_{impl} and SP_{QCimpl} are the implementation techniques of the synchronization policies of component and query component respectively. SP_{pre}
(Mi) and SPpost (Mi) are the synchronization pre and post conditions of the methods. SPpre (M_Q(C)) and SPpost (M_Q(C)) are the pre and post conditions of the query component. This specialized match can be further divided into exact and relaxed matches.

1.1 Exact Match: This match for specialized case is same as the generalized match except that it also includes the synchronization implementation. Representation of it is given below:

\[ \text{ExactMatch}_{\text{spec}}(C_1, QC) = \land (SP \equiv SP_QC) \]
\[ \land (SP_{\text{impl}} = SP_{Q(C)\text{impl}}) \]
\[ \land \text{For each method } M_i \text{ and } M_{Q(C)} \]
\[ (SP_{\text{pre}}(M_i) \Leftrightarrow SP_{\text{pre}}(M_{Q(C)})) \land (SP_{\text{post}}(M_i) \Leftrightarrow SP_{\text{post}}(M_{Q(C)})) \]

1.2 Relaxed Match: In relaxed match, the implementation of the synchronization policy is considered. Representing the relaxed match below:

\[ \text{RelaxedMatch}_{\text{spec}}(C, QC) = \land (SP \Rightarrow SP_QC) \]
\[ \land (SP_{\text{impl}} \neq SP_{Q(C)\text{impl}}) \]
\[ \land \text{For each method } M_i \text{ and } M_{Q(C)} \]
\[ (SP_{\text{pre}}(M_i) \Rightarrow SP_{\text{pre}}(M_{Q(C)})) \land \]
\[ (SP_{\text{post}}(M_i) \Rightarrow SP_{\text{post}}(M_{Q(C)})) \]

This section provided a set of matching criteria for matching synchronization contracts of two software components. The TLA+ language is used by [KUM04] to specify the proposed synchronization policies and synchronization aspects of a software component. A brief description of TLA+ is given below:

3.4.1.2. Temporal Language of Actions [TLA]

Temporal Logic of Actions [TLA] designed by Leslie Lamport is a logic-based mechanism for specifying and verifying the concurrent systems. In concurrent systems, algorithms are usually specified using a program, and the correctness of the algorithm,
indicates that the program satisfies the desired property. TLA makes it practical to describe the algorithm and the property of the program by a single formula. It is built on a logic of actions, which is a language for writing predicates, state functions, and actions, and logic for reasoning about them [LAM03]. TLA+ is the language used to write TLA specifications.

TLA allows specifying two kinds of properties of a program: 1) Safety properties, which assert that the program does not do something bad. This requires no temporal logic. 2) Liveness properties, which assert that the program eventually, will do good. The specification of a system can be represented as a TLA+ formula, represented as

\[ \text{Spec} = \text{Init} \land \Box [\text{Next}]_{\text{vars}} \land \text{Liveness} \]

where, Init represents the initial predicate that specifies the possible initial values of all the variables in the specification. Next is the next-state action, which indicates the relation between the current state of all variables and the next state (or states). Vars is the tuple of all variables. The temporal operator \( \Box \) asserts that the formula is always true. Liveness, represents what must have happen in the model is the conjunction of formulas of the form WF \( \text{vars} (A) \) [WF represents weak fairness expression] and SF \( \text{vars} (A) \) [SF represents strong fairness], for action A. So the specification, Init constrains the initial state, Next constrains what steps may occur and liveness describes what must happen eventually. Given a specification, it is possible to evaluate safety and liveness properties through a model checker to explore the state space [LAM03].

TLA+ has been used to express the synchronization specifications of the software components in this current project as well.

The next section gives the matching criteria for QoS levels of the software components proposed by [KUM04].

### 3.4.2 QoS Level Matching

The matches being hierarchical in order, when the component matching reaches this level it must have already satisfied the three levels before i.e., syntactic, semantic and the synchronization levels.
Matching of software components at the QoS level is categorized into two types, the Exact Match and the Relaxed Match.

1. Exact Match: A component is exactly matched with the query component at the QoS level if the value of the QoS parameter required is same as that being provided or if it has a better value than that being required. Better value refers to either, greater than equal to, equal to, or less than equal to. This will depend on the QoS parameter being considered and it varies from parameter to parameter. Table below 3.1 is the representation of the better QoS parameter.

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

Table 3.1: Definition of “Better” for Exact Match
where, subscripts ‘c’ and ‘qc’ represent the component and the query component respectively.

2. Relaxed Match: In this match, the QoS parameter value of the component is close to the query component’s QoS value. The user, based on the requirements, defines close value for a particular parameter. The user gives the measure of the deviation that is allowed in the values of the parameters while performing the matching. Table 3.2 below gives the definition of close for relaxed matches:

<table>
<thead>
<tr>
<th>QoS Parameter</th>
<th>Definition of “Close” for Relaxed Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependability</td>
<td>$D_c \geq D_{qc} - \text{acr}<em>d/100*D</em>{qc}$</td>
</tr>
<tr>
<td>Security</td>
<td>$S_c \geq S_{qc} - \text{acr}<em>s/100*S</em>{qc}$</td>
</tr>
<tr>
<td>Adaptability</td>
<td>$A_c \geq A_{qc} - \text{acr}<em>a/100*A</em>{qc}$</td>
</tr>
<tr>
<td>Maintainability</td>
<td>$M_c \geq M_{qc} - \text{acr}<em>m/100*M</em>{qc}$</td>
</tr>
<tr>
<td>Portability</td>
<td>$P_c = P_{qc}$</td>
</tr>
<tr>
<td>Parallelism Constraints</td>
<td>$P_{Cc} = P_{Cqc}$</td>
</tr>
<tr>
<td>Ordering Constraints</td>
<td>$O_{Cc} = O_{Cqc}$</td>
</tr>
<tr>
<td>Priority</td>
<td>$P_c = P_{qc}$</td>
</tr>
<tr>
<td>Throughput</td>
<td>$T_c \geq T_{qc} - \text{acr}/100*T_{qc}$</td>
</tr>
<tr>
<td>Capacity</td>
<td>$C_c \geq C_{qc} - \text{acr}<em>c/100*C</em>{qc}$</td>
</tr>
<tr>
<td>Turn Around Time</td>
<td>$TAT_c \leq TAT_{qc} + \text{acr}<em>t/100*TAT</em>{qc}$</td>
</tr>
<tr>
<td>Availability</td>
<td>$Av_c \geq Av_{qc} - \text{acr}<em>v/100*Av</em>{qc}$</td>
</tr>
</tbody>
</table>

Table 3.2: Definition of “Close” for Relaxed Match
where, subscripts ‘c’ and ‘qe’ represent the component and the query component respectively.

The value of either the “Better” or “Close” depends on the application under consideration, and it varies from one domain to the other. For example, real-time applications will be more interested in time related parameters such as Turn Around Time than Parallelism constraints. There is no guarantee that all the parameters are going to be considered as well. The user may just require the parameters that are important to their application. So the exact and relaxed matches can be defined for two cases, first for all the parameters for both exact and relaxed, and second for only the specified parameters for both the exact and relaxed matches.

The work done as part of this project is to use the concepts of matching proposed by [ZAR96] and [KUM04] and devise a Multilevel Matching criterion to aid in the better selection of components. The matches considered at each level are validated using a case study given in Appendix A, and a prototype is designed and developed that uses the Multilevel matching criterion and finally, providing experimental validation of the prototype.

The next chapter focuses on the Multilevel Matching Categories considered in the prototype.
4. Matching Categories

This chapter discusses the proposed hierarchical Multilevel matching categories considered in the prototype. The first section 4.1 of the chapter discusses the need for matching of the software components and extending it to the concept of Multilevel matching. The second section 4.2 discusses different matching categories proposed at the four levels: the syntactic, the semantic, the synchronization and the QoS levels. The third section 4.3 briefly introduces the concept of Multilevel match functions using the matches proposed in the previous section.

4.1 Need for Multilevel Matching

As stated in the previous chapters, the motivation for this project is to make the selection process in Uniframe Resource Discovery Service (URDS) more comprehensive. This project tries to achieve this objective of incorporating the concept of Multilevel matching in URDS, by designing and developing a prototype and empirically validating the approach. This section focuses on the need for Multilevel matching and the states the related works considered for the matching criterion.

4.1.1 Why Matching?

Matching of component is essential for the substitutability and for the compatibility of the components. Substitutability is replacing one component with other with out changing the behavior of the system. Compatibility is the ability to work properly if connected with one another. Matching is also important to find out if a component differs form the required component, what should be done to make the component compatible and substitutable.

Component matching involves matching of specifications, which describe the behavior of the component. [BEU99], identified that component’s behavior must be expressed in a specification form which would inform user of the tasks the component does, and the specification must also provide parameters against which the component
can be verified and validated. According to [BEU99], a specification is a contract for a software component that describes its properties. Component properties have been classified into four levels: the syntactic (basic), the semantic (behavioral), the synchronization, and the Quantitative.

Syntactic Level: This is considered to be the basic level, and is really important to make the system work.

Semantic Level: Semantic Level also called as Behavioral level, is important in improving the confidence in the sequential system.

Synchronization Level: Synchronization level describes the dependencies between services provided by the component. It plays an important role in improving the level of confidence in distributed system.

QoS Level: Quality of Service quantifies the quality of service and is the most negotiable level.

This prototype follows the component specification levels proposed by [BUE99], and classifies the component specifications into four levels: the syntactic, the semantic, the synchronization and the QoS levels. Component specification is represented using UMM adapted from URDS. With multilevel concept being supported by the prototype, the next step is to incorporate matching of components based on the specifications that describe its behavior. Below is the description of component match function described by [ZAR96] and adapted by the prototype.

The general idea of component match function \( M \) defined in [ZAR96], is that the match function \( M \), takes two components and returns a boolean value indicating whether the relation between the components holds or not.

\[
M : \text{Component} \times \text{Component} \rightarrow \text{Boolean}
\]

The multilevel matching of components considered in this project is a combination of the specification level taxonomy proposed by [BUE99], and above match function concept proposed by [ZAR96]. The basic idea of syntactic and semantic matching of [ZAR96], and also the matching criteria proposed by [KUM04], for the synchronization and the QoS levels is being used. In this project, the Multilevel matching criterion is devised based on the related works of [ZAR96] and [KUM04] with adapting
some of the proposed matches, discarding some matches and adding some modifications to the matches to get better component results.

The multilevel matching of components aims at providing more relevant components than the normal approach of the component selection. Component specifications published using the UMM, are compared against the search requirement given by the user. At each one of the levels the required query component criteria is matched against the candidate components offered. The prototype is designed and developed based on the proposed matches and the effectiveness of the returned search results with Multilevel as opposed to the normal search is verified as part of the project. The generic matching criteria proposed in the prototype can be represented as shown in Figure 4.1.

![Generic Component Match Diagram](image-url)
The generic match proposed at each level is instantiated to get different categories of matches either exact or relaxed. The user can select all the four levels or the required levels of matches in a hierarchical order. The representation of generic match at all the levels is explained in detail in the following section: Component-Matching Categories.

4.2 Component-Matching Categories

This section discusses the various matching criteria that are proposed at each level of the component specification. The UMM specification of the component adapted from URDS has multiple levels: the syntactic, the semantic, the synchronization and the QoS levels. At each level the proposed set of matches are compared against the query requirement and the appropriate components are selected. Below is the explanation of the set of matches proposed at each of the levels.

4.2.1 Syntactic Level Matching

The Syntactic Level match as stated by [BUE99], is considered to be the basic level of matching in component matching, a must requirement for the system to work. This is considered to be the non-negotiable level of component matching. The component specification at this level represents the syntax of interfaces of the components. It represents syntactic properties of function names, number and type of parameters, the return type of the function, etc.

The matching concepts proposed by [ZAR96], under signature matching are taken as basis for the matches considered at the syntactic level. In signature matching proposed by [ZAR96], matches are expressed in terms of whether a transformation can be applied to the signature such that the results are equal. The transformations can be a function from types to types or a function that reorders elements in the tuple. [ZAR96], proposed a comprehensive matching criterion considering type variable, type constructors, etc.

The matches considered at the syntactic level in this project take the concept of type matching from [ZAR96], and considers two conditions for matching:

- The function return type and
The function argument types

In case of function return types and function argument types, the emphasis is made on the types of the function arguments rather than on reordering of arguments in a function.

At the syntactic level, two different categories of matches are considered: the Exact Type Match and the Relaxed Type Match. The Exact Type Match, matches two types only if they match exactly. The Relaxed Type Match, matches two types through either Inheritance or Coercion. Below is the explanation of these types of matches considered for matching:

- For the Exact matching of types, the types that are considered must be exactly the same. i.e., Structural Type equivalence.
- If the types being considered do not match exactly, then the Inheritance relation between the types is explored. The concept of supertype and subtypes is explored in this category of type match. One of the relations between subtype and supertype is that, a subtype can be defined from a supertype either directly or indirectly, through multiple levels of other subtypes. Subtypes inherit all the properties of supertype. This relation is explored not only on primitive data types, but as well on the user-defined data types, e.g.; Director -> Manager -> Employee, etc.
- Coercion of types is considered if the Exact type match or Inheritance does not match. Coercion of types is to force a variable of one type to be another type.

Below is the explanation of matches considered under the syntactic level starting with definition of generic match and followed by instantiations of the generic match.

For a Component, \( C \),
The functions supported by the component can be denoted as \( C_{F1}...C_{Fl} \)
Types are considered for three different categories:
- The type of a function in component $C$ is represented as a combination of the return type of the function and the argument types of the function given as
  \[ C_{F1} \,|\, (RT, AT) \cdots C_{Fi} \,|\, (RT, AT) \], where $RT$ represents the return type of the function and the argument types supported by the functions of component $C$ are represented as $AT$, where $AT = \{ AT_1 \ldots AT_j \}$ and each $AT_j$ represents the type of one argument.

For a Component, $Q$,

The functions supported by the component can be denoted as $Q_{F1} \ldots Q_{Fi}$.

Types are considered for two different categories:
- The type of a function in component $Q$ is represented as a combination of the return type of the function and the argument types of the function given as
  \[ Q_{F1} \,|\, (RT, AT) \cdots Q_{Fi} \,|\, (RT, AT) \], where $RT$ represents the return type of the function and the argument types supported by the functions of component $Q$ are represented as $AT$, where $AT = \{ AT_1 \ldots AT_j \}$ and each $AT_j$ represents the type of one argument.

Below is the definition of Generic Type match at the syntactic Level.

### 4.2.1.1 Generic Type Match

Generic Type Match defined at the syntactic level is a combination of two matches either:
- Ordered Generic Type Match or
- Reordered Generic Type Match

Below is the definition of Generic Type match

\[
\text{Match} \ Gen-Type \ (C, Q) = [(\text{Ordered Generic Type Match} \mid \mid \text{Re-Ordered Generic Type Match})] = [ (\text{Match} \ Ordered \ Gen-Type \ (C, Q) \mid \mid \text{Match} \ Reordered \ Gen-Type \ (C, Q)) ]
\]
Below is the representation of individual matches that form the Generic Type Match.

4.2.1.1(a) Ordered Generic Type Match

Ordered Generic Match, as the name suggests, allows matching on types of the functions that have the arguments in the same order. The Ordered Generic Type Match is represented below:

\[
\text{Match}_{\text{Ordered Gen-Type}} (C, Q) = \{ \text{For each Function } C_{Fi} \text{ and } Q_{Fi} \\
\quad \quad \quad \quad ( [ C_{Fi} ]_{RT, AT} R_1 [ Q_{Fi} ]_{RT, AT} ) \}
\]

where, \( AT = \{ AT_1, \ldots, AT_j \} \) is the argument types supported by the function and each \( AT_j \) represents the type of one argument. The relation \( R_1 \) is Equality ( = ). The relation \( R_1 \) relates the function return type and the argument types of the query component and candidate component.

4.2.1.1(b) Reordered Generic Type Match

Reordered Generic Match allows matching between functions that differ in the order of arguments. This is similar to the Generic type match except that there is no emphasis on the order of arguments considered.

\[
\text{Match}_{\text{Reordered Gen-Type}} (C, Q) = \{ \text{For each Function } C_{Fi} \text{ and } Q_{Fi} \\
\quad \quad \quad \quad ( [ C_{Fi} ]_{RT, AT^*} R_2 [ Q_{Fi} ]_{RT, AT} ) \}
\]

where, \( AT^* = \{ AT_1^*, \ldots, AT_j^* \} \) is the argument types supported by the function,
not in the same order as expected by the query component. The relation \( R_2 \) is Equality (\( = \)). The relation \( R_2 \) relates the function return type and the argument types of the query component and candidate component.

The relation \( R_1 \) and \( R_2 \) defined as Equality (\( = \)), in both the types of generic matches relate the types of the function return types and argument types. Equality in context of types means that the two operands compared are equal with respect to their types. This is instantiated into two different equality conditions: Exact Equality (\( =_E \)) and Relaxed Equality (\( =_R \)).

The Exact Equality (\( =_E \)) is applied when Exact Type match is considered, where the types of the component and the query function must be exactly of same type, i.e., structural type equivalent. The Relaxed Equality (\( =_R \)) is considered when the types are matched using either Inheritance or Coercion. This is explored in Relaxed Type match where the types of the component and query functions are matched by applying either inheritance or coercion rules. In case of relaxed type match, the types are first checked to see if there exists an inheritance relation between them or else if coercion is applied to the types.

Instantiations of Generic Type Match are represented below in Table 4.1:

<table>
<thead>
<tr>
<th>Match</th>
<th>( R_1 )</th>
<th>( R_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact Type Match</td>
<td>( =_E )</td>
<td>( =_E )</td>
</tr>
<tr>
<td>Relaxed Type Match</td>
<td>( =_R )</td>
<td>( =_R )</td>
</tr>
</tbody>
</table>

| Table 4.1: Instantiations of Generic Type Match: |
| \( \text{Match}_{\text{Gen-Type}} = [ ( \text{Match Ordered Gen-Type} (C, Q) | | \text{Match Reordered Gen-Type} (C, Q) ) ] \) |
4.2.1.1(c) Generic Type Match Definitions

The Generic type match considered at the syntactic level above is instantiated to two different categories of matches: the Exact Type Match and the Relaxed Type Match. Below is the explanation of instantiations of matches proposed under syntactic level:

4.2.1.1 Exact Type Match

Match\_E-Type (C, Q)

= | (Match \_Ordered\_E-Type (C, Q) | | Match \_Reordered\_E-Type (C, Q)) |

= | For each Function C\_Fi and Q\_Fi

{ ( [C\_Fi]_{RT, AT} = E [Q\_Fi]_{RT, AT} ) | | ( [C\_Fi]_{RT, AT*} = E [Q\_Fi]_{RT, AT} ) } |

Exact Type Match is an instantiation of Generic Type match. It is a strict match of types. Exact Type match is either an ordered exact type match or reordered exact type match. Ordered exact type match allows matching on the argument types of the function in the same order as required. Reordered exact type match allows matching on the argument types that differ in the order of arguments. If components are not matched under ordered exact match then reordered exact match is considered.

The Exact Equality (\=_E) is a strict relation, which allows type matching only on the exact type of return type and argument types. Any two specifications satisfying this match can be replaced with one another, hence this is considered to be the strongest match in the set of matches proposed under the syntactic level.

4.2.1.2 Relaxed Type Match

Match\_R-Type (C, Q)

= | (Match \_Ordered\_R-Type (C, Q) | | Match \_Reordered\_R-Type (C, Q)) |


For each Function $C_{Fi}$ and $Q_{Fi}$

\[
\{ ( \{ C_{Fi} \}_{RT, AT} = \mathbf{R} \{ Q_{Fi} \}_{RT, AT} ) \mid ( \{ C_{Fi} \}_{RT, AT^*} = \mathbf{R} \{ Q_{Fi} \}_{RT, AT} ) \}
\]

Relaxed Type Match is an instantiation of Generic Type match. It is the most relaxed match in the instantiations of the matches considered under the syntactic level. It is either an ordered match or reordered match under this category. Ordered exact match allows matching on the argument types of the function in the same order as required. Reordered exact match allows matching on the argument types that differ in the order of arguments. If components are not matched under ordered exact match then reordered exact match is considered.

The Relaxed Equality ($=_{\mathbf{R}}$) relation allows type matching using either inheritance or coercion between function return types and argument types of function.

Appendix A provided at the end of the report provides examples to theoretically prove the matches using a case study.

### 4.2.2 Semantic Level Matching

Semantic matching of components considers the function specifications in the form of pre- and post-conditions. The function specifications describe the dynamic behavior of the components. The set of matches proposed under semantic level define a stronger relationship between components than in the case of the syntactic level. At this level, compared to the syntactic level, the effect of a function execution is defined. The behavior of a function execution is specified using pre-conditions, post-conditions and invariants.

The pre- and post-conditions of the function specifications, offered by the component, are used to determine the type of match between the components. The pre-conditions of a function define the states in which the function is allowed to be called by the client component. The post-conditions of a function define the state in which the function is allowed to return after its execution. The matching of pre- and post-conditions, which are Boolean assertions, use logical operators such as Equivalence, Implication, reverse implication, conjunction etc. Different category of matches are
defined from Exact matches to Relaxed matches. Defining multiple matches provides the flexibility to address a range of applications. This is useful as the user can be provided with the option to choose from the wide variety of available matches.

The matches proposed at this level use the operators proposed by [ZAR96] for the behavioral-level matching. The weakest pre/post match, proposed by [ZAR96], is discarded among the set of matches considered as it is quiet specific to a particular situation where the pre-condition of the candidate component is assumed to be true to prove the relation between the post-conditions of the query and candidate component.

The components that fit this match are usually captured in the Relaxed Post match, which only considers the post-conditions part of the generic match. This was verified using Spec# and Boogie [SPEC], and most of the results produced never reached to the weakest level match since the components were captured in the Relaxed Post match itself.

The generic match, specified at the semantic level, considers the pre- and post-conditions of the function specifications. The generic match can be further instantiated to provide a variety of kinds of matches. Below is the representation of generic pre/post match.

Consider these component notations for the matches below:

For a Component, $C$,
The functions supported by the component can be denoted as $C_{F1}...C_{Fl}$.
The pre-condition and post-conditions of the function, $C_{Fi}$, can be represented as $[C_{Fi}]_{Pre}$ and $[C_{Fi}]_{Post}$ respectively.

For a Query Component, $Q$,
The functions supported by the component can be denoted as $Q_{F1}...Q_{Fl}$.
The pre-condition and post-conditions of the function, $Q_{Fi}$, can be represented as $[Q_{Fi}]_{Pre}$ and $[Q_{Fi}]_{Post}$ respectively.
**4.2.2.1 Generic Pre/Post Match**

Generic Pre/Post match relates the pre-conditions of each component and the post-conditions of each component. The representation of generic Pre/Post match is given below

\[
\text{Match}_{\text{Gen-Pre/Post}}(C, Q) = \left[ \text{For each Function } C_{F_i} \text{ and } Q_{F_i} \right.
\]
\[
( [ Q_{F_i} ]_{\text{Pre}} \ R_1 \ [ C_{F_i} ]_{\text{Pre}} ) \ R_2 \ ( [ C_{F_i} ]_{\text{Post}} \ R_3 \ [ Q_{F_i} ]_{\text{Post}} ) \left. \right]
\]

where, the relations \( R_1 \) and \( R_3 \) relate the pre-conditions and the post-conditions of the component and can be either equivalence (\( \iff \)) or an implication (\( \Rightarrow \)). The relation \( R_2 \) is usually a conjunction (\( \land \)). The relations \( R_1 \), \( R_2 \) and \( R_3 \) are instantiated for each of the matches proposed under the semantic matching. The Table 4.2 below shows the instantiations of the pre/post conditions of the generic pre/post match specified above.

<table>
<thead>
<tr>
<th>Match</th>
<th>( R_1 )</th>
<th>( R_2 )</th>
<th>( R_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact Pre/Post Match</td>
<td>( \iff )</td>
<td>( \land )</td>
<td>( \iff )</td>
</tr>
<tr>
<td>Relaxed Pre/Post Match</td>
<td>( \Rightarrow )</td>
<td>( \land )</td>
<td>( \Rightarrow )</td>
</tr>
<tr>
<td>Relaxed Post Match</td>
<td>pre-conditions not considered</td>
<td>Nil</td>
<td>( \Rightarrow )</td>
</tr>
</tbody>
</table>

**Table 4.2 : Instantiations of Generic Pre/Post Match :**

\[
( [ Q_{F_i} ]_{\text{Pre}} \ R_1 \ [ C_{F_i} ]_{\text{Pre}} ) \ R_2 \ ( [ C_{F_i} ]_{\text{Post}} \ R_3 \ [ Q_{F_i} ]_{\text{Post}} )
\]

**4.2.2.1 Generic Pre/Post Match Definitions**

Below are the definitions of the different instantiations of Generic pre/post match and as well as the explanation of the matches with an example.
4.2.2.1.1 Exact Pre/Post Match

\[
\text{Match}_{E-\text{Pre/Post}}(C, Q) = \left[ \text{For each Function } C_{Fi} \text{ and } Q_{Fi} \right.
\]
\[
\left( [Q_{Fi}]_{\text{Pre}} \iff [C_{Fi}]_{\text{Pre}} \land ([C_{Fi}]_{\text{Post}} \iff [Q_{Fi}]_{\text{Post}}) \right]
\]

Exact Pre/Post Match is an instantiation of generic Pre/Post match. It is a strict match where Relations R₁, R₃ are equivalence (\(\iff\)) and Relation R₂ is a conjunction (\(\land\)). Two specifications hold exact pre/post match if their pre-conditions and post-conditions satisfy equivalence relation. Any two specifications satisfying this match can be replaced with one another, hence this is considered to be the strongest match in the set of matches proposed under semantic level.

4.2.2.1.2 Relaxed Pre/Post Match

In this match, the pre/post conditions are relaxed compared to exact pre/post where equivalence is used. Equivalence is a strict relation, so in this match the relation between the pre/post conditions is given an implication relation.

\[
\text{Match}_{R-\text{Pre/Post}}(C, Q) = \left[ \text{For each Function } C_{Fi} \text{ and } Q_{Fi} \right.
\]
\[
\left( [Q_{Fi}]_{\text{Pre}} \Rightarrow [C_{Fi}]_{\text{Pre}} \land ([C_{Fi}]_{\text{Post}} \Rightarrow [Q_{Fi}]_{\text{Post}}) \right]
\]

Relaxed Pre/Post Match is an instantiation of generic Pre/Post match. It is a relaxed match where Relations R₁, R₃ are equivalence (\(\Rightarrow\)) and Relation R₂ is a conjunction (\(\land\)). Relaxation is applied to this match by applying implication (\(\Rightarrow\)) to the relations between pre and post conditions. This match in [ZAR96], is considered as a plug-in match with the idea to plug-in candidate component for the required query component.
4.2.2.1.3 Relaxed Post Match

In this match, only the post-conditions of the function are explicitly indicated in the match. There is more relaxation in this match compared to the relaxed pre/post match as it considers only the post condition part of the conjunction. The pre-conditions are assumed to be satisfied using an additional check in the code as often only the post-conditions determine the outcome of the function.

\[
\text{Match}_{\text{R-Post}} (C, Q) = \left[ \text{For each Function } C_{Fi} \text{ and } Q_{Fi} \right.

\quad \left( [C_{Fi}]_{\text{Post}} \Rightarrow [Q_{Fi}]_{\text{Post}} \right) \]

Relaxed Post Match is an instantiation of generic Pre/Post match. It is a relaxed match where the Relation \( R_3 \) is equivalence (\( \Rightarrow \)).

These are the matches considered at the semantic level limiting the matches only to function specifications. The theoretical proof of these matches is provided using a case study in the Appendix A. In the prototype implementation, each of these matches is verified using Spec# programming system [SPEC], the details of which are explained in the next chapter.

4.2.3 Synchronization Level Matching

This level defines the dependencies between the services offered by the components. The inclusion of this level in the component matching improves the level of confidence in distributed or concurrent systems. At this level, the synchronization policy and the implementation technique are the matching parameters used to compare the specifications. The synchronization policy is used by the components to handle multiple client accesses on the services offered by it. Some of the examples of the policies are mutual exclusion, producer consumer, readers writers, etc. The synchronization implementation technique is the method of implementation of the policies. Some of the examples are mutexes, semaphores etc.
Matches proposed for Synchronization by [KUM04] have been modified in this project in both the Exact and Relaxed categories of matches. Below is the explanation of changes made to the matches:

1. The concept of multiple synchronization policies being implemented by a single component has been discarded. In this project, all the interfaces supported by a component implement the same synchronization policy and the implementation technique (if any).

2. For the matches proposed at the synchronization level, only the synchronization policy and the implementation technique are considered. The synchronization behavior of a component’s interface that represents the pre-condition, the post-condition and the invariants of the policy’s implementation method have not been considered because the pre-condition, post-condition and invariants have already been considered as part of functions behavior at the Semantic level. The above consideration of behavior of the policy will be useful if the component implements a dynamic or a new synchronization policy whose behavior is not known to the user. In such a case, considering the synchronization behavior of the method or components interface would be interesting and would lead to a different set of new categories of matches. In this project, well known synchronization policies such as MutualExclusion, ProducerConsumer and ReadersWriters have been used to implement the match at the policy level.

3. Addition of a new logical relation Reverse Implication ( \( \Rightarrow \) ) between the synchronization policies.

Three categories of logical relations, applied to the synchronization policies, are Equivalence ( \( \Leftrightarrow \) ), Implication ( \( \Rightarrow \) ), and Reverse Implication ( \( \Leftarrow \) ). Two different categories of arithmetic operators have been used for the implementation techniques used to implement the policies, Equality ( \( = \) ), and Inequality ( \( \neq \) ). The matches proposed here explore different categories of matches that can be obtained when these operators are
applied to the respective parameters considered at the synchronization level. Below is the explanation of generic matches proposed at the Synchronization Level.

Two different categories of generic matches are specified at the Synchronization Level.

1. **Generic SP match**: This match considers only the relation between the synchronization policies.

2. **Generic SP-Impl match**: This match considers the relation between synchronization policies and the implementation of the policy.

For each kind of the match, both the exact and relaxed matches are defined. Below is the representation of the Generic SP match, followed by Generic SP-Impl match.

Consider these component notations for the matches below:

For a Component, $C$,

The Synchronization Policy implemented by the component can be denoted as $C_{SP}$. The implementation of Synchronization Policy can be represented as $C_{SP-Impl}$.

For a Query Component, $Q$,

The Synchronization Policy implemented by the component can be denoted as $Q_{SP}$. The Implementation of Synchronization Policy can be represented as $Q_{SP-Impl}$.

### 4.2.3.1 Generic SP Match

The Generic SP match is the representation of Synchronization policy implemented by the function/functions supported by the component.

The representation of Generic SP match is:

$$\text{Match}_{\text{Gen-SP}} (C, Q) = [C_{SP} \ R_1 \ Q_{SP}]$$

where, $C_{SP}$ and $Q_{SP}$ represent the synchronization policies implemented by the components. The relation $R_1$ relates the synchronization policies of the components $C$ and $Q$, and can be either equivalence ($\Leftrightarrow$), implication ($\Rightarrow$) or Reverse Implication ($\Leftarrow$).
The Table 4.3 below shows the instantiations of the synchronization policy part of the Generic SP match specified above:

<table>
<thead>
<tr>
<th>Match</th>
<th>$R_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact SP Match</td>
<td>$\Leftrightarrow$</td>
</tr>
<tr>
<td>Relaxed SP Match</td>
<td>$\Rightarrow$</td>
</tr>
<tr>
<td>Reverse Implication SP Match</td>
<td>$\Leftarrow$</td>
</tr>
</tbody>
</table>

| Table 4.3: Instantiations of Generic SP Match: $[C_{SP} \ R_1 \ Q_{SP}]$ |

### 4.2.3.1 Generic SP Match Definitions

This section presents different kinds of matches under Generic SP match.

#### 4.2.3.1.1 Exact SP Match

$$\text{Match}_{E-SP} (C, Q) = [C_{SP} \ Leftrightarrow Q_{SP}]$$

Exact SP Match is an instantiation of generic SP match. It is a strict match where Relations $R_1$ is equivalence ($\Leftrightarrow$).

#### 4.2.3.1.2 Relaxed SP Match

$$\text{Match}_{R-SP} (C, Q) = [C_{SP} \Rightarrow Q_{SP}]$$
Relaxed SP Match is an instantiation of generic SP match. It is a relaxed match where Relations $R_1$ is implication ($\Rightarrow$).

4.2.3.1.3 Reverse Implication SP Match

$$\text{Match}_{RI-SP} (C, Q) = [C_{SP} \iff Q_{SP}]$$

Reverse Implication SP Match is an instantiation of generic SP match. It is a relaxed match where Relations $R_1$ is implication ($\iff$).

4.2.3.2 Generic SP – Impl Match

Generic SP-Impl match is a combination of:

1. Synchronization policy implemented by the function/functions supported by the component.
2. Implementation of the synchronization policy.

The representation of Generic SP-Impl match is given below:

$$\text{Match}_{Gen-SP-Impl} (C, Q) = [C_{SP} \ R_1 \ Q_{SP}]$$

$$\quad \quad R_2 [C_{SP-Impl} \ R_3 \ Q_{SP-Impl}]$$

where, as before, $C_{SP}$ and $Q_{SP}$ represent the synchronization policies implemented by the components. The relation $R_1$ relates the synchronization policies of the components $C$ and $Q$, and can be either equivalence ($\iff$), implication ($\Rightarrow$). The relation $R_3$ relates the implementation techniques of synchronization policies $C_{SP-Impl}$ and $Q_{SP-Impl}$, and can be either equal ($=$) or not equal ($\neq$), i.e., either can be same implementation technique or different. The relation $R_2$ is a conjunction ($\land$).
The main synchronization policies considered in this project are MutualExclusion, ProducerConsumer and ReadersWriters. The implementation techniques that are considered are Mutexes and Semaphores. TLA+ specifications have been written for all the policies and the implementation of the policies using the implementation techniques. The relation between the policies has been proved using TLA+ specifications.

The Table 4.4 shows the instantiations of the synchronization policy and implementation part of the generic implementation match specified above.

<table>
<thead>
<tr>
<th>Match</th>
<th>( R_1 )</th>
<th>( R_2 )</th>
<th>( R_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact SP - Exact Impl</td>
<td>( \Leftrightarrow )</td>
<td>( \land )</td>
<td>( = )</td>
</tr>
<tr>
<td>Exact SP - Relaxed Impl</td>
<td>( \Leftrightarrow )</td>
<td>( \land )</td>
<td>( \neq )</td>
</tr>
<tr>
<td>Relaxed SP - Exact Impl</td>
<td>( \Rightarrow )</td>
<td>( \land )</td>
<td>( = )</td>
</tr>
<tr>
<td>Relaxed SP - Relaxed Impl</td>
<td>( \Rightarrow )</td>
<td>( \land )</td>
<td>( \neq )</td>
</tr>
<tr>
<td>Reverse Implication SP - Exact Impl</td>
<td>( \Leftrightarrow )</td>
<td>( \land )</td>
<td>( = )</td>
</tr>
<tr>
<td>Reverse Implication SP - Relaxed Impl</td>
<td>( \Leftrightarrow )</td>
<td>( \land )</td>
<td>( \neq )</td>
</tr>
</tbody>
</table>

Table 4.4 : Instantiations of Generic SP – Impl Match : \([C_{SP} \ R_1 \ Q_{SP}] \)

\( R_2 \) \([C_{SP-Impl} \ R_3 \ Q_{SP-Impl}] \)

The section below gives the definitions of different instantiations of Generic SP match and Generic SP – Impl match.

### 4.2.3.2 Generic SP – Impl Match Definitions

This section presents different kinds of matches under Generic SP-Impl match.
4.2.3.2.1 Exact SP-Exact Impl Match (Exact Synch. Policy, Exact Impl)

\[
\text{Match}_{E-[SP, \text{Impl}]}(C, Q) = [C_{SP} \iff Q_{SP}]
\]
\[
\land [C_{SP-\text{Impl}} = Q_{SP-\text{Impl}}]
\]

Exact SP-Impl Match is an instantiation of Generic SP match with implementation. It is a strict match where the relations R₁ is equivalence (\(\iff\)), the relation R₃ is (\(=\)) and the relation R₂ is conjunction (\(\land\)).

4.2.3.2.2 Exact SP-Relaxed Impl Match (Exact Synch. Policy, Relaxed Impl)

This match is a variation of the above match where the implementation policy of the required query is not the same as the component.

\[
\text{Match}_{[E-SP, R-\text{Impl}]}(C, Q) = [C_{SP} \iff Q_{SP}]
\]
\[
\land [C_{SP-\text{Impl}} \neq Q_{SP-\text{Impl}}]
\]

Exact SP - Relaxed Impl Match is an instantiation of generic SP match with implementation. It is a strict match where the relations R₁ is equivalence (\(\iff\)), the relation R₃ is (\(\neq\)) and the relation R₂ is conjunction (\(\land\)).

4.2.3.2.3 Relaxed SP-Exact Impl Match (Relaxed Synch. Policy, Exact Impl)

This match is a variation of the above match where the implementation policy of the required query is not the same as the component.

\[
\text{Match}_{[R-SP, E-\text{Impl}]}(C, Q) = [C_{SP} \Rightarrow Q_{SP}]
\]
\[
\land [C_{SP-\text{Impl}} = Q_{SP-\text{Impl}}]
\]
Relaxed SP- Exact Impl Match is an instantiation of generic SP match with implementation. It is a relaxed match where the relations $R_1$ is implication ($\Rightarrow$), the relation $R_3$ is ($\neq$) and the relation $R_2$ is conjunction ($\wedge$).

### 4.2.3.2.4 Relaxed SP-Relaxed Impl Match
*(Relaxed Synch. Policy, Relaxed Impl)*

This match is a variation of the above match where the implementation policy of the required query is not the same as the component.

$$\text{Match}_{[R-SP, R-Impl]}(C, Q) = \left[ C_{SP} \Rightarrow Q_{SP} \right]$$

$$\wedge \left[ C_{SP-Impl} \neq Q_{SP-Impl} \right]$$

Relaxed SP-Relaxed Impl Match is an instantiation of generic SP match with implementation. It is a relaxed match where the relations $R_1$ is implication ($\Rightarrow$) the relation $R_3$ is ($\neq$) and the relation $R_2$ is conjunction ($\wedge$).

### 4.2.3.2.5 Reverse Implication SP-Exact Impl Match
*(Reverse Implication Synch. Policy, Exact Impl)*

$$\text{Match}_{[RI-SP, E-Impl]}(C, Q) = \left[ C_{SP} \Leftarrow Q_{SP} \right]$$

$$\wedge \left[ C_{SP-Impl} = Q_{SP-Impl} \right]$$

Reverse Implication SP- Exact Impl Match is an instantiation of generic SP match with implementation. It is a relaxed match where the relations $R_1$ is reverse implication ($\Leftarrow$), the relation $R_3$ is ($\neq$) and the relation $R_2$ is conjunction ($\wedge$).
4.2.3.2.6 Reverse Implication SP-Relaxed Impl Match

*(Reverse Implication Synch. Policy, Relaxed Impl)*

\[
\text{Match}_{[RI-SP, R-Impl]}(C, Q) = \left[ C_{SP} \iff Q_{SP} \right] \\
\land \left[ C_{SP-Impl} \neq Q_{SP-Impl} \right]
\]

Reverse Implication SP- Relaxed Impl Match is an instantiation of generic SP match with implementation. It is a relaxed match where the relations \( R_1 \) is reverse implication (\( \iff \)), the relation \( R_3 \) is (\( \neq \)) and the relation \( R_2 \) is conjunction (\( \land \)). This match is a variation to the above reverse implication SP- exact Impl match, with the relation between the implementation techniques being relaxed. The candidate component returned for this match, matches with the Query component under reverse implication in case of synchronization policy but does not return the same implementation technique as required by the Query.

For all the synchronization policies considered in the project and as well as the implementation techniques considered for the synchronization policies TLA+ specifications have been written proving the relation that holds between them.

4.2.4 QoS Level Matching

Quality of Service, QoS is the non-functional attribute of the component. This is the final level of matching in the multilevel matching process. QoS defines how well the system operates or how well the functionality is exhibited. These are the design decisions taken to implement system’s functional attributes. QoS is important in providing robust, scalable and secure distributed component systems. QoS plays an important role in distributed systems as the non-functional aspects of the system play an important role in ensuring the reliability and high confidence to the user on the system.

QoS parameters can be categorized into two different categories: Static QoS parameters and Dynamic QoS parameters. Static QoS parameters are the ones that remain stable in different environments provided the internal structure of the components is unchanged. Some of the QoS static parameters are: reliability, maintainability,
parallelism constraints, portability, scalability, dependability, priority, security, ordering constraints, adaptability etc. On the other hand, dynamic QoS parameters are measured by observing the system behavior at the run-time. Some of the examples of these are: throughput, turn around time, capacity, availability, etc. The requirement of the QoS parameters depends on the system under consideration; some of the systems may require only a minimum number of parameters.

This QoS aspect of the component is taken into consideration and different kinds of matches are proposed for the component at this level. A component match that reaches this level must have already satisfied all the first three functional levels of the component (either exact or relaxed). The matches proposed at this level are numerical and so a Boolean comparison of the values of the parameters is considered. Matches proposed for QoS in this project extend the matches proposed by [KUM04] by adding a modification to the Exact and Relaxed QoS matches. The concept of “Better” and “Close” values is enhanced in this project. “Better” when applied returns a value of the parameter that is either equal to or greater than required by the user. “Close” value is defined as the value either less than or equal to or greater than the required value. “Better” is applied to the Exact match and “Close” is applied to the Relaxed matches. Below is the explanation of changes made to the matches:

- A formal representation of the QoS Generic match with the relations
- In case of both Exact and Relaxed QoS matches, a user-specified deviation, ( provided by the user) for the value of the required QoS parameter is considered.
- In Exact QoS Match, the idea of “Better ” match still holds, with a modification done in the range of values being accepted for comparison. The accepted deviation value of the parameter is either added or subtracted to/from the original value based on the nature of parameter. Any candidate component, which lies with in the range of expected original value of the parameter and the accepted deviation in the value is considered as an Exact Match. The main aim of Exact match is to provide a parameter with better or exact value than the required parameter value.
In Relaxed QoS Match, the idea of “Better” and “Close” match holds, with added relaxation in the range of the values being considered for the match. The accepted deviation value of the parameter is both added and subtracted to the original value. The range of values taken in the match is broadened here. Any component that lies with in the wide range is given as a candidate component to the required QoS parameter value. In case of Relaxed, the value of parameter returned could be better, equal or close to the value being required.

The above decisions have been made as compared to the matches proposed by [KUM04], as accepting the deviation for both the Exact and Relaxed match gives a cutoff to the values under consideration. The reason for adding the cutoff is to limit the time required for the search process, which in case of a large database with many components can become a bottleneck and degrade the performance of the Multilevel matching. The next change of considering a wider range for the Relaxed match is to provide the user with more components that do not fall under the Exact match. The component search that reaches the QoS level must have satisfied all the three upper levels (either at exact or relaxed) and adding more relaxation at this level both for exact and relaxed can provide the user with wider set of probable components. As the QoS level is considered as the most negotiable level among all the four levels considered for component matching [BEU99] so, providing more relaxation at this level is justifiable.

Below is the representation of Generic QoS match. The generic match considered at the QoS level considers the QoS parameter and the QoS value for that parameter.

Consider these component notations for the matches below:

For a Component, \( C \),
The QoS parameters of the component, \( C \), is represented as
\[
\begin{align*}
\{ C \} _{\text{QoS-ParamName}} (1) \cdot \cdot \cdot \{ C \} _{\text{QoS-ParamName}} (i)
\end{align*}
\]
and the corresponding values are represented as
\[
\begin{align*}
\{ C \} _{\text{ParamValue}} (1) \cdot \cdot \cdot \{ C \} _{\text{ParamValue}} (i)
\end{align*}
\]
For a Query Component, \( Q \),

The QoS parameters of the component, \( Q \), is represented as

\[
[ Q ]_{\text{QoS-ParamName}(1)} \cdots [ Q ]_{\text{QoS-ParamName}(i)}
\]

and the corresponding values are represented as

\[
[ Q ]_{\text{ParamValue}(1)} \cdots [ Q ]_{\text{ParamValue}(i)}
\]

The value of the accepted deviation in the parameter value represented as

\[
[ Q ]_{\text{ParamDeV}(1)} \cdots [ Q ]_{\text{ParamDeV}(i)}
\]

### 4.2.4.1 Generic QoS Match

Generic QoS match relates the QoS parameters considered and their corresponding associated values for the component. QoS parameters are associated to the component as a whole but not to individual function. The representation of Generic QoS match is given below

\[
\text{Match}_{\text{Gen-QoS}}( C, Q ) = \{ \text{For each or required QoS Parameter of } Q \text{ and } C, \\
( [ C ]_{\text{QoS-ParamName}(i)} \ R_1 [ Q ]_{\text{QoS-ParamName}(i)} ) \\
R_2 \\
( [ Q ]_{\text{ParamValue}(i)} \ R_3 \\
[ C ]_{\text{ParamValue}(i)} \ R_4 \\
\{ [ Q ]_{\text{ParamValue}(i)} \ R_5 [ Q ]_{\text{ParamDeV}(i)} \} )
\}
\]

where,

\[
[ Q ]_{\text{ParamDeV}(i)} = \{ [ Q ]_{\text{ParamValue}(i)} \times \text{Dev/100} \}
\]

where, the relation \( R_1 \) relates the QoS parameters of the Components and can be either equal (\( = \)) or inequality (\( \neq \)). Relation \( R_2 \) is a conjunction (\( \land \)) operator. The Relations \( R_3 \) and \( R_4 \) are comparison operators either greater than equal to (\( \geq \)), equal to (\( = \)), or
less than equal to (≤). The Relation $R_5$ is an arithmetic operator either an addition (+) or subtraction (−) or both. The variable “Dev” represents the user specified deviation in the QoS value. The value of the operators, $R_3$, $R_4$, and $R_5$ depends on the type of QoS parameters being considered. Below is the explanation of the operators used for two different parameters: Throughput and Turn Around Time.

For example, if QoS parameter Throughput is considered, then the operators $R_3$ and $R_4$ considered must be less than or equal to (≤). The operator $R_5$ is addition (+). The reason for this is that the provided component throughput value must lie with in the range of the required throughput value and the accepted throughput value with deviation. The deviation in this case is positive deviation. Below is the representation of the parameter Throughput applying the relations:

\[
[Q]_{\text{Throughput}} \ R_5 \ [Q]_{\text{ThroughputDeV}} = [Q]_{\text{Throughput}} + [Q]_{\text{ThroughputDeV}}
\]

where,

\[
[Q]_{\text{ThroughputDeV}} = \{ [Q]_{\text{Throughput}} \times \text{Dev} \div 100 \}
\]

so, the candidate component with Throughput value with in the range of original required value and deviation value is provided by applying the comparison operator greater than equal to (≥).

\[
( [Q]_{\text{Throughput}} \leq [C]_{\text{Throughput}} \leq \{ [Q]_{\text{Throughput}} + [Q]_{\text{ThroughputDeV}} \})
\]

As mentioned above the value of the operators used depends on the type of the QoS parameter under consideration. Now consider, QoS parameter Turn-Around Time, applying the same relations to the above parameter. The operators $R_3$ and $R_4$ considered must be greater than equal to (≥). The operator $R_5$ is subtraction (−). The lesser value of Turn-Around Time is considered to be the better value and hence the deviation value is subtracted from the original value hence, negative deviation. Below is the representation of Turn-Around Time(TAT) applying the relations:
\[
[Q]_{\text{TAT}} \text{ Rs } [Q]_{\text{TAT}Dev} = [Q]_{\text{TAT}} - [Q]_{\text{TAT}Dev}
\]

where,
\[
[Q]_{\text{TAT}Dev} = \{ [Q]_{\text{TAT}} \times Devl 100 \}
\]

So, the candidate component with TAT value with in the range of original required value and deviation value is provided by applying the comparison operator greater than equal to (\(\geq\)).

\[
( [Q]_{\text{TAT}} \geq [C]_{\text{TAT}} \geq \{ [Q]_{\text{TAT}} - [Q]_{\text{TAT}Dev} \})
\]

Tabular representation of the operators used is not shown in QoS case as for the exact as well as the relaxed, the same types of operators are used. The difference lies in the range of values accepted for comparisons in both the matches. Detailed explanation of the Exact and the Relaxed match is provided in the next section.

4.2.4.1 QoS Match Definitions

Below is the explanation of instantiations of matches proposed under QoS level, starting with the Exact QoS match and followed by the Relaxed QoS match.

4.2.4.1.1 Exact QoS Match

The Exact QoS match is represented below:

\[
\text{Match}_{\text{E-QoS}} (C, Q) = \{ \text{For each or required QoS Parameter of } Q \text{ and } C,}
\]

\[
( [C]_{\text{QoS-ParamName}(i)} = [Q]_{\text{QoS-ParamName}(i)} )
\]

\[
\land
\]

\[
( [Q]_{\text{ParamValue}(i)} \text{ R}_3 ) \land
\]

\[
( [C]_{\text{ParamValue}(i)} \text{ R}_4 )
\]
\[
\{ [Q]_{ParamValue(i)} \; R_5 \; [Q]_{ParamDev(i)} \}
\]

where,
\[
[Q]_{ParamDev(i)} = \{ [Q]_{ParamValue(i)} \times Dev / 100 \}
\]

Exact QoS Match is an instantiation of generic QoS match. It is a strict match where Relations \(R_1\), is binary relation equality (=) and Relation \(R_2\) is a conjunction (\(\land\)). The Relations \(R_3\) and \(R_4\) are comparison operators either greater than equal to (\(\geq\)), equal to (\(=\)), or less than equal to (\(\leq\)). The Relation \(R_5\) is an arithmetic operator either an addition (\(+\)) or subtraction (\(-\)). “\(Dev\)” represents the user specified deviation.

### 4.2.4.1.2 Relaxed QoS Match

The Relaxed QoS match is represented below:

\[
\text{Match}_{R-QoS} (C, Q) = \{ \text{For each or required QoS Parameter of } Q \text{ and } C, \}
\]

\[
([C]_{QoS-ParamName(i)} = [Q]_{QoS-ParamName(i)})
\]

\[\land\]

\[
\{ [Q]_{ParamValue(i)} \; R_5 \; [Q]_{ParamDev(i)} \}
\]

\[
[Q]_{ParamValue(i)} \; R_4 \; [C]_{ParamValue(i)}
\]

\[
\{ [Q]_{ParamValue(i)} \; R_5' \; [Q]_{ParamDev(i)} \}
\]

where,
\[
[Q]_{ParamDev(i)} = \{ [Q]_{ParamValue(i)} \times Dev / 100 \}
\]

Relaxed QoS Match is an instantiation of generic QoS match. It is a relaxed match where Relations \(R_1\), is binary relation equality (=) and Relation \(R_2\) is a
conjunction ($\land$). The Relations $R_3$ and $R_4$ are comparison operators either greater than equal to ($\geq$), equal to ($=$), or less than equal to ($\leq$). The Relation $R_5$ and $R_5'$ is either an addition ($+$) or subtraction ($-$). The value of the relation depends on the parameter type considered. “Dev” represents the user specified deviation value.

The main difference between the Exact and the Relaxed match is increasing the range of values for the search in the later. The range of values that are considered in Exact QoS lies in between the original required value and the added deviated value. But in case of Relaxed match, the range considered is in between the positive and negative deviations added to the original value. The Relaxed QoS match aims at providing more components as the search results by applying a wide range of parameter values to the search. The match at the QoS either Exact or Relaxed aims at providing the match for all the proposed parameters in QoS or only the required parameters required in the query.

This concludes the matching criteria proposed for all the four levels in the component matching.

### 4.3 Representation of Multilevel Match

This section briefly introduces the concept of combining all the matches proposed to obtain a Multilevel matching of component specifications. The matches proposed at all the levels can be combined into one single match function based on the levels of matches selected and the kind of match selected. The component Multilevel matching must satisfy the following conditions:

- The Multilevel match, can be a combination of all the four levels or less depending on the user requirement, with the only constraint on the user being that the levels must be in hierarchical order.
- Different combinations of kinds of matches can be selected at each level but at any point only either Exact or Relaxed can be selected at each level.

Below is the representation of Generic Multilevel match that considers all the four levels of component behavior:

**Generic Multilevel Match**
For example if the Exact match at all the four levels have to be implemented, the Exact Multilevel match function for such a case can be represented as:

**Exact Multilevel Match**

\[
\text{Exact Multilevel Match} = [ \text{Exact Type Match} \\land \text{Exact Pre/Post Match} \\land (\text{Exact SP Match} \lor \text{Exact SP-Impl Match}) \\land \text{Exact QoS Match} ]
\]

where, each individual match implements their corresponding proposed definitions given in the above sections.

Consider a case where, Exact Syntactic, Exact Semantic, Relaxed Synchronization and Relaxed QoS are selected then the representation of combinations of Multilevel match is:

**Combination Multilevel Match**

\[
\text{Combination Multilevel Match} = [ \text{Exact Type Match} \\land \text{Exact Pre/Post Match} \\land (\text{Relaxed SP Match} \lor \text{Relaxed SP-Impl Match}) \\land \text{Relaxed QoS Match} ]
\]

Based on the matching categories proposed, a prototype that implements these matches was designed and developed. The next chapter discusses the prototype, its architecture and the implementation details.
5. Prototype Details

In the previous chapter, the matching categories have been discussed; this chapter focuses on the design and implementation of the prototype based on the matching categories proposed. The first section 5.1 of the chapter discusses the design issues of the prototype, the architecture of the prototype. The second section 5.2 describes the query requirements. The third section 5.3 describes the four matchers: the syntactic matcher, the semantic matcher, the synchronization matcher and the QoS matcher considered in the architecture and their implementation of the prototype.

5.1 Prototype Design

The prototype designed and developed tries to assess Multilevel matching of components using the matching criteria proposed in the last chapter. The main focus of the prototype implementation is to validate the matches proposed and verify the search results returned. Based on the user feedback, it uses standard measures of information retrieval such as precision and recall to analyze the search results. For the implementation of the prototype, the component specifications are represented using a centralized database implemented via MS-Access. The database holds various data tables that represent the component details and services provided. When the query is given to the prototype with required information, it searches against the database to provide the user with appropriate candidate components.

As mentioned in the previous chapters, multilevel matching starts with the syntactic matching followed by the semantic level, the synchronization level and the QoS level matching. The matching process, being hierarchical, does not allow the user to specify the matching levels at random. The emphasis is on hierarchical matching, but all the levels of matching need not be considered. The user can select only required levels, but in a hierarchical approach. At each level, every matcher provides the user with an option to choose either an exact match or a relaxed match. Figure 5.1 below is the representation of prototype implementation.
The prototype has four levels: syntactic matcher, semantic matcher, synchronization matcher and QoS matcher. The user input is given to prototype and the required level of matches is selected. Components are returned at each level based on the kind of match selected, i.e. either Exact or Relaxed. An Exact match at each level represents the category of matches that match exactly. Relaxed match represents the combination of matches under Exact and Relaxed match, i.e. all the matches that have been proposed under exact and relaxed categories are implemented. This design decision is made on the belief that the relaxation when applied must provide a wide variety of components including the specific components, which the exact match returns.

After providing the required information, selecting the required levels, and the type of match at each level, the user requirement is queried against the existing database, which contains the component specifications. The returned results are verified and validated based on the user feedback. The user provides a feedback about the returned components as to either relevant or non-relevant. The Information retrieval technique such as Precision, which is the metric for measuring performance, is calculated based on the relevant and non-relevant components. Precision is defined as the number of relevant documents retrieved divided by the total number of documents retrieved. Applying the same to component search, Precision is calculated as:

\[
\text{Precision} = \frac{\text{Number of Relevant Components}}{\text{Total Number of Components Retrieved}}
\]

Precision is calculated for the relevant components at each level. A graphical representation of the precision values at each level to the query is given in the experimental analysis chapter. This can be used to visualize the effectiveness of the multilevel search process.

The prototype is implemented using the object-oriented programming language C# on Microsoft Visual Studio.Net 2005 Version (8.0.50727.42). It also uses SPEC# 1.0.6003 (RTM), a theorem prover based on C#. The centralized database used is MS-Access. Figure 5.1 below is the representation of the prototype architecture.
Figure 5.1: Representation of Multilevel Matching Prototype
5.2 User Query Requirements

The user is allowed to enter through the front-end of the prototype, the required query requirements. The requirement entered represents the required component specification the user is interested in. Figure 5.2 below represents the requirements that are mandatory with respect to the required levels.

1. Component Domain Type
2. Syntactic and Semantic Details
   - Required Function/Functions Signature
   - Pre-conditions required for the Corresponding Function
   - Post-conditions required for the Corresponding Function
3. Synchronization Details
   - Synchronization Policy
   - Synchronization Implementation
4. QoS Attributes Details
   - QoS Attribute Name
   - QoS Value
   - Accepted Deviation

Figure 5.2: User Query Requirement for the Prototype

The prototype also takes input parameters, such as component name, author name and description of component, function name, style, and technology supported by the component, that are not mandatory. The next step for the user is to decide on the level of matches and type of match required at each level. Since it is hierarchical, the user has to go level by level starting from syntactic to QoS. Based on the requirements of the user and the level of match specified, the corresponding search components are returned.

As specified above, the prototype has four different matchers representing the four different levels in component specification. Each Matcher is further categorized into
two categories of matches: Exact and Relaxed. Based on the category selected by the user, the corresponding match is executed. Below are the assumptions made in categorizing the matches.

- Irrespective of the level, any Matcher with the Exact option implements only the strict matches proposed with respect to that level.
- Any Matcher, when chosen with the Relaxed option, covers all the matches proposed under that respective level. i.e., the matches considered under the Relaxed option is a combination of the Exact match and all the Relaxed matches proposed under that level.

With the above categorization of matches, the query requirement is given to the component specifications and appropriate components are returned to the user.

### 5.3 Prototype Matchers

This section describes the four matchers used in the prototype to implement multilevel matching.

#### 5.3.1 Syntactic Matcher

Syntactic Matcher represents the syntactic level matching of the component specification. Syntactic level indicates how a particular interface of a component can be accessed. Syntactic level matching represents the basic level of matching in the prototype. This is considered as a non-negotiable level, as component specifications have to match syntactically to consider any further matching levels.

Syntactic level defines the syntactic information of the component specification, the function signature, the return type, the argument types and the variable types in pre- and post-conditions. The user requirements that are needed at this level are Component Domain Type, Syntactic and Semantic Details. Below is the explanation of implementation of the syntactic level match in the prototype.

The match functions proposed in the previous chapter for syntactic match are implemented here. At the syntactic level four different instantiations of generic type
match were proposed. For implementation these instantiations are further categorized into Exact Syntactic Match and Relax Syntactic Match.

5.3.1(a) Exact Syntactic Match

This match is implemented when the user selects the “Exact” option at Syntactic Level. Match instantiations that is defined under this category is “ Exact-Type Match ”. The representation of this match is given below:

Exact Syntactic Match

\[
= \text{Match}_{E\text{-Type}, E\text{-Vtype}} (C, Q)
\]

\[
= [ ( \text{Match}_{\text{Ordered E\text{-Type}}}(C, Q) \mid \text{Match}_{\text{Reordered E\text{-Type}}}(C, Q) ) ]
\]

where, C and Q represent Component and Query respectively.

The Exact Syntactic match is either an ordered exact match or a reordered exact match. Main difference between the ordered and reordered match is the order of argument types of the function. The query requirement is first checked against the ordered exact match; if it is not satisfied, then the reordered match is executed. The possible outcomes of this match are given below:

1. All the Candidate components are matched under the “Exact Type Match”.
2. Among the set of Candidate components given to the “Exact Type Match” only some are satisfied.
3. No match returned. “Exact Type Match” not matched for any of the candidate components. None of the components are satisfied under Exact Syntactic Match.

5.3.1(b) Relaxed Syntactic Match
This match is implemented when the user selects the “Relaxed” option in the Syntactic Matcher. Relaxed match considers all the matches proposed under the level including the exact match. The combination of instantiations considered here are: “Exact Type Match” OR “Relaxed Type Match”. Below is the list of match instantiations considered under this category.

**Relaxed Syntactic Match**

\[
= [ \text{Match } E\text{-Type } (C, Q) \; | \; \text{Match } R\text{-Type } (C, Q) ]
\]

\[
= [ (\text{Match } Ordered E\text{-Type } (C, Q) \; | \; \text{Match } Reordered E\text{-Type } (C, Q)) ]
\]

\[
| |
\]

\[
[ (\text{Match } Ordered R\text{-Type } (C, Q) \; | \; \text{Match } Reordered R\text{-Type } (C, Q)) ]
\]

The matches are applied to the candidate components and query requirement in the same order specified above. If the components satisfy any one match, they are passed on as a result of Relaxed syntactic match.

The possible outcomes of this match are given below:

1. All the candidate components are matched under the Relaxed Syntactic Match, any one of the above matches might have been satisfied.
2. Among the set of candidate components given to the matcher only some of them are satisfied.
3. No match returned. None of the components are matched under Relaxed Syntactic Match.

**5.3.2 Semantic Matcher**
The Semantic Matcher is the second level option selected in the component search process. This level ensures semantically matched components either Exact or Relaxed, based on the option selected by the user. For this matcher the following conditions have to be satisfied:

- The Syntactic Match (Exact or Relaxed) must have been selected even before the Semantic Matcher is selected.
- The returned components from Syntactic Matcher (Exact or Relaxed) are considered as the candidate components for the Semantic Matcher.
- If the Syntactic Matcher does not yield any components then the Semantic Matcher even if selected, is not executed.

In general, a Semantic Match compares the pre- and the post-conditions of function requirements and tries to prove a relation between them. The Match Functions proposed in the previous chapters under Semantic Match are implemented in the Semantic Matcher. The Semantic Generic Pre/Post match has been instantiated to three different categories of matches:

1. Exact Pre/Post Match
2. Relaxed Pre/Post Match
3. Relaxed Post Match

The Semantic Matcher categorizes the above instantiation of matches as either exact or relaxed. If the user chooses the exact option, then it executes Exact Pre/Post match. If the relaxed option is chosen under semantic matcher, all the three matches that are mentioned above are considered. The representations of exact and relaxed semantic matches is given below:

5.3.2(a) Exact Semantic Match

Semantic Matcher executes Exact Semantic Match when the user selects the “Exact” option. The representation of match executed here is given below:

\[ \text{Match}_{\text{E-Pre/Post}}(C, Q) = \;[\text{For each Function } C_{Fj} \text{ and } Q_{Fj}] \]
\[( | Q_{Fi} |_{Pre} \iff | C_{Fi} |_{Pre} ) \land ( | C_{Fi} |_{Post} \iff | Q_{Fi} |_{Post} ) \]

The possible outcomes of this match are given below:

1. All the syntactically matched Candidate components are matched under the “Exact Pre/Post Match”.
2. Among the set of Candidate components given to the “Exact Pre/Post Match” only some are satisfied.
3. No match returned. “Exact Pre/Post Match” not matched for any of the candidate components. None of the components are satisfied under Semantic Level Match.

5.3.2(b) Relaxed Semantic Match

The Semantic Matcher executes Relaxed semantic match, when the user selects the Relaxed option. This option considers the two relaxed matches proposed under Semantic match as part of implementation. It does not consider the Exact Pre/Post match with equivalence, instead it uses Relaxed Pre/Post match with implication since any relation that satisfies Equivalence also satisfies Implication. The Relaxed Semantic Match implements in the following manner:

1. First, all the syntactically satisfied candidate components are applied to “Relaxed Pre/Post Match”.
2. If all the components are matched under “Relaxed Pre/Post Match”, then the results are given to user.
3. If only some of the components are matched under “Relaxed Pre/Post Match”, then the rest of the components are applied to “Relaxed Post Match”.
4. If none of the components are matched under “Relaxed Pre/Post Match”, then all the components are applied to “Relaxed Post Match” and the results are given to the user.

Below are the definitions of the matches implemented in the prototype:

Relaxed Semantic Match
\[
\text{Match}_{R-Pre/Post} (C, Q) = \left\{ \begin{array}{ll}
\text{Match}_{R-Post} (C, Q)
\end{array} \right. \\
\]

\[
\text{Match}_{R-Pre/Post} (C, Q) = \left[ \text{For each Function } C_{Fi} \text{ and } Q_{Fi} \right. \\
\left. \left( [Q_{Fi}]_{pre} \Rightarrow [C_{Fi}]_{pre} \right) \wedge \left( [C_{Fi}]_{post} \Rightarrow [Q_{Fi}]_{post} \right) \right] \\
\]

\[
\text{Match}_{R-Post} (C, Q) = \left[ \text{For each Function } C_{Fi} \text{ and } Q_{Fi} \right. \\
\left. \left( \left[ C_{Fi} \right]_{post} \Rightarrow \left[ Q_{Fi} \right]_{post} \right) \right] \\
\]

Different possible outcomes when Relaxed Semantic Match is executed are:

1. All the syntactically satisfied candidate components are matched under the “Relaxed Pre/Post Match”.
2. Among the set of candidate components given to the “Relaxed Pre/Post Match” some are satisfied under it and the rest of all components are matched under “Relaxed Post Match”.
3. Among the set of candidate components given to the “Relaxed Pre/Post Match” only some are satisfied, and in the rest of components, only some are matched under “Relaxed Post Match”.
4. None of the candidate components satisfy “Relaxed Pre/Post Match” and all are matched under “Relaxed Post Match”.
5. None of the candidate components satisfy “Relaxed Pre/Post Match” and only some of them are matched under “Relaxed Post Match”.
6. No match returned. Neither “Relaxed Pre/Post Match” nor “Relaxed Post Match” is satisfied. None of the candidate components satisfy the requirement under Semantic Level Match.

The semantic matcher in this project uses Microsoft Spec# programming system [SPEC] for the verification of pre and post conditions. The pre/post conditions are written to a “. SSC” file and given as input to the Spec# Compiler. Spec# compiles the “. SSC” file using the command

\%
SSC.EXE SemanticInput.SSC –Debug
where, “SemanticInput” is the input file with the pre- and post-conditions of the Query component and the Candidate component. The compiled Spec# file is given as input to Boogie.EXE [SPEC] where the relations between pre/post conditions are validated. Boogie is a static program verifier, which generates verification conditions from a Spec# program.

% Boogie SemanticInput.EXE

Boogie returns the output indicating the functions whose pre/post conditions have failed. From this output the number of components matched to the query specification can be known. The next sections give a detailed description of the architecture of the Spec# programming system used in this Project for Semantic Matching.

Figure 5.3: Spec# Programming System
This Project uses Microsoft Spec# programming system for the verification of pre- and post-conditions at semantic level. Below is an explanation of the architecture of the Spec# programming system, consisting of the Spec# programming language, the Spec# compiler, and the Boogie static program verifier which internally uses the Simplify theorem prover.

The Spec# programming system aims to develop and maintain a high-quality software in a cost effective way. Below are the main contributions of the Spec# programming system as mentioned in [BAR04]:

- Providing a small extension to an already popular language
- Methodology that permits specification and reasoning about object invariants even in the presence of deadlocks
- Tools that enforce the methodology, which range from easily usable dynamic checking to high-assurance static verification, and
- Enabling programmers to take advantage of the benefits of specification.

The Spec# programming system is developed with the above-mentioned goals. Below is the explanation of the architecture of the system.

### 5.3.2(c) Spec# Programming Language

Spec# is an extension of an object-oriented programming language C#. It incorporates all the features of C#, and adds type support for distinguishing non-null object references from null object references, method contracts in the form of pre- and post-conditions, and checked exceptions.

Spec# supports a non-null type system and enforces it by adding a type support for nullity discrimination. It handles this problem by syntactically allowing the constructors to initialize the fields before the object being constructed becomes reachable by the program. It requires initializers for every non-null field.

Spec#, mainly focuses on the concept of design by contract. One of the important features of it is “method contracts”. Spec# provides a specification for every method that describes the use of the method and also outlines a contract between caller and
implementations. The specification includes pre-conditions and post-conditions as well as invariants. Specification pre-conditions describe the states in which the method is called and the post-conditions describe the states in which the method is allowed to return.

Exceptions are handled in Spec# by categorizing them based on the conditions they signal. The exceptions handling is enhanced in Spec# compared to what is supported by C#. Two categories of exceptions are declared as in Java: checked or unchecked exceptions. The failures that can be admissible/accepted are signaled with checked exceptions, but failures such as client failures and observed program errors are signaled as unchecked exceptions. Checked exceptions are the ones that should be caught by the user, and any specification that implements the interface ICheckedException is considered as a checked exception. Runtime default exception handler usually catches the unchecked exceptions.

5.3.2(d) Spec# Compiler

Spec# compiler is integrated into Microsoft Visual Studio development environment for the .Net platform. The main difference between spec# and other compilers is that spec# produces executable code from a program written in spec# language and also preserves all the specifications into a language-independent format. Spec# compiler can preserve the specifications in the same binary with the compiles code. In the background the spec# code is first translated to Microsoft .Net CLI, Common Language Interface bytecode and then static program verifier converts it to BoogiePL. The spec# compiler attaches a specification to each program component for which a specification exists.

5.3.2(e) Boogie: Static Program Verifier

The component Boogie generates logical verification conditions from a Spec# program [SPEC]. Internally, the bytecode from Spec# is converted to an intermediate language BoogiePL, by Boogie, a static program verifier. BoogiePL is a procedural language, much similar to an assembly language, which is used for program analysis and program verification. From BoogiePL program, one can generate verification conditions
or perform other program analyses such as the inference of program invariants [DEL05].
BoogiePL is a language with procedures whose implementations are basic blocks consisting of four kinds of statements: assignments, asserts, assumes and procedural calls. BoogiePL goes through several transformations, and ends as verification conditions which are fed to automatic theorem prover. BoogiePL supports most of the arithmetic operations and also provides a special operator “\( \langle: \rangle \)” supported to express the partial order on name type.
BoogiePL program is a set of declarations:
Program ::= Decl*  
  Decl ::= Type|Constant|Function|Axiom|Variable|Procedure|Implementation  

BoogiePL includes types to include a guard against certain easy-to-make mistakes in the input, although the theorem prover included in it is untyped. BoogiePL supports four built-in basic types, user-defined types, one- and two-dimensional arrays, and the supertype any:
Type ::= bool|int|ref|name  
| Id|any  
| “[” Type “]” Type  
| “[’ Type “,’ Type “]” Type

The type bool represents boolean type with values false and true. The type int represents the mathematical integers. Arithmetic operations are allowed on integer types. Type ref represents references like object, pointers, and addresses. Operations allowed on reference types are equality and inequality. The type name represents defined names such as types and field names with operations such as equality, inequality and the partial order “\( \langle: \rangle \)” . User-defined type allows the user to identify certain set of values. Id is defined as a type and there must be distinct from all other user-defined types. Type any is superset of all types. Any expression can be assigned to a variable or array element of type any. BoogiePL program has three separate global namespace: one namespace for user-defined types, one for the names of functions and procedures, and the third one for the names of symbolic constants and global variables.

Boogie after generating verification conditions for BoogiePL forwards them to Simplify, an automatic theorem prover that attempts to prove the verification conditions.
Boogie in addition to accepting MSIL programs can also accept BoogiePL programs directly. The BoogiePL programs must be files whose names end with .bpl and these programs are written in Unicode or ASCII.

5.3.2(f) Simplify: Theorem Prover

Simplify is an automatic theorem prover, which Boogie uses internally. It analyzes the verification conditions to prove the correctness of the program showing that the BoogiePL and original Spec# program are correct or finds counter examples showing that there is an error in the BoogiePL program and the original Spec# program. Simplify uses something called goal property heuristic, to analyze the verification conditions [BAR05]. It thrives on the kind of formulas generated by Extended Static Checking Project ESC/Modula-3 and, especially, by ESC/Java [DNS05]. The input to Simplify is a formula of untyped first-order logic with function symbols and equality. The theorem prover supports propositional connectives such as ∧, ∨, ¬, ⇒, ⇔; the universal quantifier ∀, and the existential quantifier ∃. It also supports arithmetic symbols such as +, −, × and the relational symbols <, >, ≥, and ≤. Any terms that are either arguments to the functions or relations of the arithmetic theory are assumed to be integers. Simplify also supports theory of maps, which is characterized by the postulated semantics of the function symbols Select and Store. Simplify takes the verification condition of a BoogiePL program and returns the result back to Boogie.

Spec# is used in this prototype to check the semantic level matching by providing the pre- and post-conditions of the functions of both the query requirement and candidate components. It is also provided with the relations between the pre/post conditions and as well as logical operators. The output of the system is used to conclude as to which components have satisfied the specified query requirement.

Next section discusses the Synchronization Matcher and the functionality implemented by it in matching of specifications at the synchronization level.
5.3.3 Synchronization Matcher

The Synchronization Matcher is the third level option selected in the component search process. The components returned at this level, either exact or relax, satisfy the synchronization behavior of the query requirement. The conditions that need to be satisfied for this matcher to execute are:

- The user can only select this option if the upper levels, syntactic and semantic (in that order), are selected.
- Components that have matched at the syntactic and semantic level (exact or relax) are considered as candidate components at this level.
- If all the three levels of matches are selected and the upper level matches do not return any components then the synchronization matcher even if selected, is not executed.

As indicated in the previous chapter, the synchronization policy and its implementation technique are the matching parameters used to compare the specifications at the synchronization level. Two kinds of generic matches are proposed under this level: Generic SP Match and Generic SP-Impl Match. The Generic SP Match is considered if the query requirement only considers the policy match. If the query requires both the policy and the implementation technique then Generic SP-Impl Match is considered. The Representation of exact and relaxed synchronization matches is given below:

5.3.3(a) Exact Synchronization Match

This match is executed when the user selects the “Exact” option in the synchronization matcher. Two different implementations are executed based on the requirement given by the user, if only policy match is required then the Synchronization Matcher executes Exact SP Match or else it executes Exact SP-Exact Impl match. The definition of matches are given below:

If Synchronization Policy is required, “Exact-SP Match” is implemented. The representation of it is given below:
\textbf{Match}_{E-SP}(C, Q) = \ [( C_{SP} \Leftrightarrow Q_{SP} ) \] 

If Synchronization Policy & Implementation is required, “Exact SP-Exact Impl Match” is implemented. The representation of it given below:

\textbf{Match}_{E-[SP, Impl]}(C, Q) = \ [( C_{SP} \Leftrightarrow Q_{SP} ) \\
\land ( C_{SP-Impl} = Q_{SP-Impl} ) \] 

The possible outcomes of this match are given below:

If Synchronization Policy or Synchronization Policy-Impl is considered,

1. All the syntactically and semantically matched candidate components are matched under the either “Exact SP Match” or “Exact SP-Impl Match” based on the requirement.
2. Among the set of candidate components given to the “Exact SP Match” or “Exact SP-Impl Match” only some are satisfied.
3. No match returned. “Exact SP Match” or “Exact SP-Impl Match” not matched for any of the candidate components.

\textbf{5.3.3(b) Relaxed Synchronization Match}

Here “Relaxed” option is selected under the Synchronization Match. The implementations incorporated in the prototype are again based on the user requirements. Below are the definitions of these matches:

If only the Synchronization Policy is required, then the following instantiations have been considered for implementation:

\[ \ [( \text{Match}_{E-SP}(C, Q) ) \lor ( \text{Match}_{R-SP}(C, Q) ) \lor ( \text{Match}_{RL-SP}(C, Q) ) \] 
\[ = \ [( C_{SP} \Leftrightarrow Q_{SP} ) \lor ( C_{SP} \Rightarrow Q_{SP} ) \lor ( C_{SP} \Leftarrow Q_{SP} ) \] 

If the Synchronization Policy & Implementation technique is required, four different instantiations have been considered. The matches that have been discarded
during the implementation of the prototype are the “Relaxed SP - Relaxed Impl Match” and “Reverse Implication SP - Relaxed Impl Match” since, they are the most relaxed matches and relaxation applied to both the categories has not been considered.

\[
\begin{align*}
&\text{ Match } \{E-SP, \text{ Impl}\} (C, Q) \quad \text{||} \quad \text{ Match } \{E-SP, R-\text{Impl}\} (C, Q) \quad \text{||} \\
&\text{ Match } \{R-SP, E-\text{Impl}\} (C, Q) \quad \text{||} \quad \text{ Match } \{R-SP, E-\text{Impl}\} \ (C, Q) \\
&= \left\{ \begin{array}{l}
[ C_{SP} \Leftrightarrow Q_{SP} ] \land [ C_{SP-\text{Impl}} = Q_{SP-\text{Impl}} ] \ \\
[ C_{SP} \Leftrightarrow Q_{SP} ] \land [ C_{SP-\text{Impl}} \neq Q_{SP-\text{Impl}} ] \ \\
[ C_{SP} \Rightarrow Q_{SP} ] \land [ C_{SP-\text{Impl}} = Q_{SP-\text{Impl}} ] \\
[ C_{SP} \Leftrightarrow Q_{SP} ] \land [ C_{SP-\text{Impl}} = Q_{SP-\text{Impl}} ]
\end{array}\right.
\end{align*}
\]

In case of the relaxed synchronization match, based on the requirement of the user, the above-mentioned matches are applied. Among the set of matches proposed if any of the matches is satisfied, it is considered as a candidate component.

Different possible outcomes when Relaxed Synchronization Match is executed are:

If Synchronization Policy or Synchronization Policy-Impl is considered,

1. All the components applied to either only SP or SP-Impl matches are matched.
2. Among the set of candidate components only some are satisfied.

This project proves the synchronization policy and policy-Impl using TLA+ specifications. TLA+ language has been used in this project to specify the synchronization polices and the implementation techniques of the policies. The specifications have been written for policies such as Mutual Exclusion, ProducerConsumer, and ReadersWriters. Specifications have also been given for Mutexes, Semaphores in implementing the policies. Appendix provided at the end
includes some of the TLA+ specifications of the synchronization policy and implementation considered.

5.3.4 QoS Matcher

QoS is the final level of matching in the component specification. This level represents the non-functional attributes of the component. The components that are returned at this level, exact or relax, satisfy the QoS properties of the query requirement. The conditions that need to be satisfied for this matcher to execute are:

- The user can only select this option if the upper levels, the syntactic, the semantic and the synchronization (in that order), are selected.
- Candidate components to this match are the components that have matched at all the upper levels.
- If all the levels of matches are selected and the upper level matches do not return any components then QoS matcher even if selected, is not executed.

The QoS parameters are the matching parameters used to compare in this match. Different instantiations of Generic QoS Match proposed at this level are implemented in QoS matcher. Representation of Exact QoS Match and Relaxed QoS Match is given below:

5.3.4(a) Exact QoS Match

In this case, “Exact” option is selected by the user and implements the following match.

\[
\text{Match } E-QoS(C, Q) = \{ \text{For each or required QoS Parameter of } Q \text{ and } C, \\
( |C|_{QoS-ParamName(i)} = |Q|_{QoS-ParamName(i)} ) \land \\
( |Q|_{ParamValue(i)} R_3 |C|_{ParamValue(i)} R_4 \{ |Q|_{ParamValue(i)} R_5 |Q|_{ParamDeV(i)} \} ) \}
\]
where,
\[
\{ \mathcal{Q} \}_{ParamDeV(i)} = \{ \mathcal{Q} \}_{ParamValue(i)} \times \text{Dev} / 100
\]

The Relations \( R_3 \) and \( R_4 \) are comparison operators whose value depend on the QoS parameter being considered. It can be either greater than equal to (\( \geq \)), equal to (\( = \)), or less than equal to (\( \leq \)). The Relation \( R_5 \) is an arithmetic operator either an addition (+) or subtraction (−). “Dev” is the user specified deviation value.

The possible outcomes of this match are given below:

1. The Candidate components that have matched at all the upper three levels (exact or relax) are matched under “Exact QoS Match”.
2. Among the set of candidate components given to the “Exact QoS Match” only some are satisfied.
3. No match returned. None of the components match the requirement at this level, implying that the prototype did not yield any components.

5.3.4(b) Relaxed QoS Match

In this case, “Relaxed” option is selected by the user and implements the following match. The difference between this match and the Exact match is the range of deviation used in selecting the components. In Relaxed QoS match, the range of deviation is considered to include the Exact QoS match components as well.

\[
\text{Match } R_{-\text{QoS}}(C, \mathcal{Q}) = \{ \text{For each or required QoS Parameter of } \mathcal{Q} \text{ and } C, \\
\quad ( [ C ]_{\text{QoS-ParamName}(i)} = [ \mathcal{Q} ]_{\text{QoS-ParamName}(i)} ) \land \\
\quad ( \{ [ \mathcal{Q} ]_{ParamValue(i)} \}_{R_5} [ \mathcal{Q} ]_{ParamDev(i)} \} \{ [ C ]_{ParamValue(i)} \}_{R_4} \\
\quad \{ [ \mathcal{Q} ]_{ParamValue(i)} \}_{R_5'} [ \mathcal{Q} ]_{ParamDev(i)} \} )
\]
where, 
\[
[Q]_{ParamDev(i)} = \{ [Q]_{ParamValue(i)} \times Dev / 100 \}
\]

The Relations \(R_3\) and \(R_4\) are comparison operators whose value depend on the QoS parameter being considered. It can be either greater than equal to (\(\geq\)), equal to (\(=\)), or less than equal to (\(\leq\)). The Relation \(R_5\) is an arithmetic operator either an addition (\(+\)) or subtraction (\(-\)). “\(Dev\)” is the user specified deviation value.

The possible outcomes of this match are given below:

1. The Candidate components that have matched at all the upper three levels (exact or relax) are matched under “Relaxed QoS Match”.
2. Among the set of candidate components given to the “Relaxed QoS Match” only some are satisfied.
3. No match returned. None of the components match the requirement at this level, implying that the prototype did not yield any components.

This chapter introduced the architecture of the prototype and the matchers that are implemented to obtain multilevel matching. Appendix provided at the end of the report discusses various algorithms implemented in the prototype as part of implementation. The next chapter focuses on the experimentation done and the verification of results returned.
6. Experimental Analyses of the Prototype

Chapter 5 provided the design details of the prototype and its experimental details. It also described the matching criteria implemented by each matcher as a part of the proposed Multilevel matching concepts. This chapter focuses on the empirical validation of the Multilevel matching concepts implemented in the prototype.

6.1 Main Objectives of the Experimental Verification

The main objective of the prototype implementation is to show the merit of Multilevel matching as opposed to a Single Level matching. The Multilevel matching approach aims at providing more meaningful components than the Single level matching. The experiments conducted verify some of the issues, such as: the merit of the components returned at each level, the variations in the number of components returned when the same query is given using an Exact match and a Relaxed match, the time taken to provide the components to the user for both kinds of matches, using information retrieval techniques, such as Precision, to analyze the search results.

6.1.1 Experimental Setup

The prototype developed is implemented using object oriented programming language C# on Microsoft Visual Studio.Net 2005 Version (8.0.50727.42). It also uses SPEC#, a theorem prover based on C# with release Spec# 1.0.6003 (RTM). The centralized database used is MS-Access, which holds the component specifications represented using various data tables. The database has four different domain types: Bank, Math, Weather and Search domains. The total number of components in the database is 334. The queries that were given in the experimentations mainly focused on the Bank domain, which has 47 components.
6.1.2 Different Categories of Experiments

The main aim of the prototype is to implement the Multilevel matches proposed and also to verify if Multilevel matching returns more relevant components as compared to Single Level matching. Based on this, the experimentation conducted focused on validating the well-known established concepts of Multilevel matching, such as more relevant matches, more time and high Precision components. Below are the categories of experimentation done

1. To determine whether the component matches returned implement the respective category of proposed matching criteria or not.
2. To determine whether or not if matches returned in Multilevel matching have more relevant components than the normal single level matching.
3. To justify the extra time taken to execute Multilevel matching as opposed to Single Level matching.
4. To compare the number and quality of components returned in exact match and relaxed match. To see if relaxed match returns more or equal number of components than exact match or not.
5. To calculate the Precision of the components returned at each level, and determine whether or not if the Precision of the returned components increases gradually or not.
6. Based on the above experiments, justify the claim that executing, Multilevel matching returns more relevant components per unit time than Single Level matching.

All the above experiments are represented graphically and explanations are given for the graphs.
6.1.3 Different Categories of Queries

Two different categories of queries are given to the prototype: Single Method Queries and Multiple Method Queries. Single method queries, as the name suggests represent a particular method specification, which the user requires. Multiple method queries are a combination of different methods. Compared to single method queries, multiple method queries have to satisfy more requirements, as the provided components must satisfy all the conditions of the query.

The experimentation was done against the Bank domain, which has 47 different components supporting different methods. For each kind of match 20 different queries were given for both the single method and multiple methods. In the experiments that were done, all the four levels of component matches have been considered. Two kinds of matches “Exact” and “Relaxed” were considered at all the levels. Since there are four levels and two kinds of matches, many categories of queries are possible with the combination of levels and kinds of matches. As a part of the experimentation, some of these categories have been explored.

Let the 20 queries that were given be represented as $Q_1...Q_{20}$. Each query is a combination of four levels of component specification: syntactic, semantic, synchronization and QoS. Table 6.1 below represents different categories of queries considered. All the queries shown in the table here are multiple method queries. Each combination of queries is considered with exact and relaxed match as well.

<table>
<thead>
<tr>
<th>Category Number</th>
<th>Different Categories</th>
<th>Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Same Signature</td>
<td>Every Query implementing Exact and Relaxed Match</td>
</tr>
<tr>
<td></td>
<td>Same Pre/Post</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Same Synch.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Same QoS</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Query Details</td>
<td>ID Numbers</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------</td>
<td>------------</td>
</tr>
<tr>
<td>2</td>
<td>Same Signature Same Pre/Post Same Synch. Diff QoS</td>
<td>Q15, Q16</td>
</tr>
<tr>
<td>3</td>
<td>Same Signature Same Pre/Post Diff Synch. Diff QoS</td>
<td>Q11, Q12</td>
</tr>
<tr>
<td>4</td>
<td>Same Signature Diff Pre/Post Diff Synch. Diff QoS</td>
<td>Q6, Q9</td>
</tr>
<tr>
<td>5</td>
<td>Diff Signature Same Pre/Post Same Synch. Same QoS</td>
<td>Q13, Q14</td>
</tr>
<tr>
<td>6</td>
<td>Same Signature Diff Pre/Post Same Synch. Diff QoS</td>
<td>Q1, Q2</td>
</tr>
<tr>
<td>7</td>
<td>Diff Signature Same Pre/Post Same Synch. Same QoS</td>
<td>Q3, Q5, Q7, Q9</td>
</tr>
<tr>
<td>8</td>
<td>Diff Signature Diff Pre/Post Same Synch. Diff QoS</td>
<td>Q7, Q10</td>
</tr>
<tr>
<td>9</td>
<td>Diff Signature Diff Pre/Post Diff Synch. Diff QoS</td>
<td>Q9, Q10</td>
</tr>
</tbody>
</table>

**Table 6.1: Different Categories of Queries**
Consider an example of a query category:
Category Number: 6
Query Category: Same Signature, Different Pre/Post, Same Synch, And Different QoS
Queries: Q1, Q2
The Table 6.2 below represents the query specifications:

<table>
<thead>
<tr>
<th>Q No</th>
<th>Method Signature</th>
<th>Pre-Conditions</th>
<th>Post-Conditions</th>
<th>Synch-Impl</th>
<th>QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>void deposit(string, float)</td>
<td>acctid!=null amount&gt;0.0</td>
<td>newbalance&gt;oldbalance</td>
<td>MEC Mutexes</td>
<td>Turn Around Time (TAT), Throughput</td>
</tr>
<tr>
<td></td>
<td>void withdraw(string, float)</td>
<td>acctid!=null amount&gt;0.0</td>
<td>newbalance&lt;oldbalance</td>
<td>MEC Mutexes</td>
<td></td>
</tr>
<tr>
<td>Q2</td>
<td>void deposit(string, float)</td>
<td>acctid!=null</td>
<td>newbalance&gt;oldbalance</td>
<td>MEC Mutexes</td>
<td>TAT, Security</td>
</tr>
<tr>
<td></td>
<td>void withdraw(string, float)</td>
<td>acctid!=null</td>
<td>newbalance&lt;oldbalance</td>
<td>MEC Mutexes</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: Query Q1 and Q2 Requirements

As seen in the Table 6.2 above, the two queries are variations of one another with different query requirements for the same methods. The number of components returned from the two queries is represented below in Table 6.3:

<table>
<thead>
<tr>
<th>Query No.</th>
<th>CompID/IDs For Exact Match</th>
<th>CompID/IDs For Relaxed Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>2, 23</td>
<td>2,3,5,10,13,16,21,23,28,26,30,37,40,42</td>
</tr>
<tr>
<td>Q2</td>
<td>2,11,21,23</td>
<td>1,2,3,5,8,10,11,13,16,20,21,23,26,30,28,36,37,38,39,40,42,47</td>
</tr>
</tbody>
</table>

Table 6.3: CompID/IDS returned for Queries Q1 and Q2
The emphasis of the experimentation is not to consider all the combinations of query categories that arise from four different levels and two kinds of matches, but only to give different queries implementing Multilevel matching. So, not all the 16 categories that can be formed from 4 different levels are considered.

The rest of the queries that were considered were those that had no specific relation between them and the other queries. In a similar manner, for single method queries i.e., queries that require only a single method specification, some were related and some disjoint. The results of the experimentation done are represented in the graphs shown in the next section.

6.2 Experimental Graphs

This section represents different graphs related to the experimentation done to evaluate Multilevel matching.

6.2.1(a) Components Returned - Multilevel Match (Single Method Queries)

This section is divided into two parts: the first part of the section presents the graphs showing the components returned for the Single Method Queries for Exact Match. The second part presents the graphs for Relaxed Match.

Part 1: Graph 6.1 presenting the number of components returned for Exact match of Single Method Queries.

Objective: The main objective of presenting this graph is to show:

1. The gradual decrease in the number of components at each level is consistent with the hypothesis that Multilevel matching gives fewer components than single level matching.
2. If, for any particular query the number of components returned is same for a Multilevel match and single level match, results of such queries needs to be justified.

The representation of the X-axis and Y-axis is given below:
Horizontal-Axis: Query Number.
Vertical-Axis: Number of Components returned implementing Exact Multilevel Match for Single Method Queries.

Graph 6.1: Number of Components returned for Exact Multilevel Match (Single Method Queries)

1. Validation of Query Results for Graph 6.1

This section validates a couple of queries shown in the Graph 6.1. It considers two queries: Query Q1 and Query Q7.

i) Results of Query Q1
The requirements of Query Q1 is presented in the Table 6.4(a) below:
(TAT represents Turn Around Time)

<table>
<thead>
<tr>
<th>Q</th>
<th>Method Signature</th>
<th>Pre-</th>
<th>Post-</th>
<th>Synch-Impl</th>
<th>QoS</th>
</tr>
</thead>
</table>

Table 6.4(a): Query Q1 Requirements

The number of components returned is presented below in Table 6.4(b):

<table>
<thead>
<tr>
<th>Query No.</th>
<th>CompID - Syntactic Exact Match</th>
<th>CompID - Semantic Exact Match</th>
<th>CompID - Synch. Exact Match</th>
<th>CompID - QoS Exact Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 6.4(b): CompID/IDs returned for Query Q1

As seen from the Table 6.4(b) the query requirement Q1 returns only single valid component with CompID = 6 satisfying all the levels and so the Graph 6.1 shows all the levels as equal for the Query Q1. Here, the number of components returned for single level matching and the Multilevel matching are same. Since the components returned at the previous levels form the input to the following levels, the CompID=6 is compared against all the levels for Multilevel match to check if it satisfies the query requirement or not.

ii) Results of Query Q7

The requirements of Query Q7 is presented in the Table 6.4(c) below:

<table>
<thead>
<tr>
<th>Q No</th>
<th>Method Signature</th>
<th>Pre-Conditions</th>
<th>Post-Conditions</th>
<th>Synch-Impl</th>
<th>QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q7</td>
<td>void Deposit(string, float)</td>
<td>acctid!=null balance&gt;0.0</td>
<td>newbalance&gt;oldbalance</td>
<td>ME-Mutexes</td>
<td>Throughput</td>
</tr>
</tbody>
</table>

Table 6.4(c): Query Q7 Requirements
<table>
<thead>
<tr>
<th>Query No.</th>
<th>CompID - Syntactic Exact Match</th>
<th>CompID - Semantic Exact Match</th>
<th>CompID - Synch. Exact Match</th>
<th>CompID - QoS Exact Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q7</td>
<td>2,3,5,7,8,9,10,11,13,16,21,23,37,39,40,41,42,47</td>
<td>2,3,5,7,8,9,10,11,13,16,21,23,37,39,40,42,47</td>
<td>2,10,11,21,23,47</td>
<td>2,10,21,23,47</td>
</tr>
</tbody>
</table>

Table 6.4(d): CompID/IDs returned for Query Q7

As represented in Table 6.4(d), for query requirement Q7 there is a gradual decrease in the number of components from syntactic to QoS levels, suggesting that the Multilevel match returns smaller set of components, which are more relevant than single level matching also satisfying one of the objectives of the experimentation.

**Part 2:** Graph 6.2 showing the number of components returned for Relaxed match of Single Method Queries.

**Objective:** The main objective of presenting this graph is to show:

1. The gradual decrease in the number of components at each level, proving the fact that Multilevel match gives fewer relevant components than single level matching.
2. Determine whether the components returned for the Relaxed match includes those components returned for Exact match or not. This is checked to see if the set of components returned under Relaxed match contains atleast the components returned from Exact match or not, if not more.
3. If, for any particular query the number of components returned is the same for a Multilevel match and single level match, results of such queries need to be justified.

The representation of the X-axis and Y-axis is given below:

**Horizontal-Axis:** Query Number.

**Vertical-Axis:** Number of Components returned executing Relaxed Multilevel match for Single Method Queries.
Graph 6.2: Number of Components returned for Relaxed Multilevel Match (Single Method Queries)

1. Validation of Query Results for Graph 6.2

This section validates a couple of queries shown in the Graph 6.2. It considers two queries: Query Q1 and Query Q9.

iii) Results of Query Q1

The requirements of Query Q1 is presented in the Table 6.4(e) below:

(TAT represents Turn Around Time)

<table>
<thead>
<tr>
<th>Q No</th>
<th>Method Signature</th>
<th>Pre-Conditions</th>
<th>Post-Conditions</th>
<th>Synch-Impl</th>
<th>QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>void deposit(string, int)</td>
<td>acctid!=null</td>
<td>newbalance&gt;oldbalance</td>
<td>ME-Semaphores</td>
<td>TAT, Throughput</td>
</tr>
</tbody>
</table>

Table 6.4(e): Query Q1 Requirements
The number of components returned is presented below in Table 6.4(f):

<table>
<thead>
<tr>
<th>Query No.</th>
<th>ComplID - Syntactic Relaxed Match</th>
<th>ComplID - Semantic Relaxed Match</th>
<th>ComplID- Synch. Relaxed Match</th>
<th>ComplID - QoS Relaxed Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>2,3,5,6,7,8,9,10,11, 13,16,21,23,37,39, 40,41,42,47</td>
<td>2,3,5,6,7,8,9,10,11, 13,16,21,23,37,39, 40,41,42,47</td>
<td>2,5,6,7,8,9,10,11, 21,23,37,42,47</td>
<td>2,5,6,7,10,21, 23,37,42,47</td>
</tr>
</tbody>
</table>

**Table 6.4(f): CompID/IDs returned for Query Q1**

The components returned at each level for query Q1 are plotted in the Graph 6.2 and a gradual decrease in the number of components can be seen that are more relevant in case of Multilevel matching than for the single level matching.

Also, compare the performance of the same query when given under Exact match vs. Relaxed match. The Exact match has returned only one relevant component as shown in Table 6.4(b) but, the Relaxed match as seen in Table 6.4(f) has returned 10 components, including the component returned in the Exact match, ComplID: 6.

iv) **Results of Query Q9 and Q14**

Consider Query Q9 and Q14 to address the third objective in presenting the Graph 6.2.

The requirements of Query Q9 is presented in the Table 6.4(g) below:

<table>
<thead>
<tr>
<th>Q No</th>
<th>Method Signature</th>
<th>Pre-Conditions</th>
<th>Post-Conditions</th>
<th>Synch-Impl</th>
<th>QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q9</td>
<td>void get(int)</td>
<td>Balance&gt;0</td>
<td>Newbalance&lt;oldbalance</td>
<td>RW-Semaphores</td>
<td>Throughput</td>
</tr>
</tbody>
</table>

**Table 6.4(g): Query Q9 Requirements**
The number of components returned is presented below in Table 6.4(h):

<table>
<thead>
<tr>
<th>Query No.</th>
<th>CompID - Syntactic Relaxed Match</th>
<th>CompID - Semantic Relaxed Match</th>
<th>CompID - Synch. Relaxed Match</th>
<th>CompID - QoS Relaxed Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q9</td>
<td>14,18,22,24,35</td>
<td>14,18,22,24,35</td>
<td>18,22</td>
<td>18,22</td>
</tr>
</tbody>
</table>

**Table 6.4(h): CompID/IDs returned for Query Q9**

The components returned at each level for query Q9 is plotted in the Graph 6.2. As seen in the graph, the syntactic and the semantic have returned the same number of components, so did the synchronization, and the QoS levels. From the Table 6.4(h) it can be seen that initially at the syntactic level, 5 components have matched the query requirement and also the same set have matched the semantic as well. But, when the synchronization matcher is applied, the components returned have reduced to 2, and the same components hold good for the QoS match as well. Although, the upper two levels have returned the same number of components and the lower two levels have same number of components, the Multilevel match still is better than the single level match. Hence the Graph 6.2 shows same number of components for the syntactic and semantic levels and also for the synchronization and the QoS levels.

The requirements of Query Q14 is presented in the Table 6.4(i) below:

<table>
<thead>
<tr>
<th>Q No</th>
<th>Method Signature</th>
<th>Pre-Conditions</th>
<th>Post-Conditions</th>
<th>Synch-Impl</th>
<th>QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q14</td>
<td>void GetBalance (string)</td>
<td>acctid!=null</td>
<td>acctid!=null</td>
<td>RW-Semaphores</td>
<td>Throughput</td>
</tr>
</tbody>
</table>

**Table 6.4(i): Query Q14 Requirements**

The number of components returned is presented below in Table 6.4(j):
Table 6.4(j): CompID/IDs returned for Query Q14

As seen in the Graph 6.2, the components plotted for query Q14 returned same number of components, which can be seen in the Table 6.4(j) as well. At all the four levels, for the given query requirement only one component with CompID: 9 is matched and hence the graph shows all the four levels as equal for query Q14. Here the number of components returned for the single level match and the Multilevel match are same but since only one component is returned at the syntactic level, only it is matched across the remaining levels.

This concludes the validation of the graphs presenting the components returned for both the Exact and the Relaxed matches for Single Method Queries.

6.2.1(b) Time Taken - Multilevel Match (Single Method Queries)

This section is divided into two parts: the first part of the section presents the graphs showing the time taken for Single Method Queries for Exact Match. The second part presents the graphs showing the time taken for Relaxed match.

Part 1: Graph 6.3 presenting the time taken for Exact match of a Single Method Queries.
Objective: The main objective of presenting this graph is to show:
1. The time taken to execute a Multilevel match is more than a single level match and the time taken to implement Multilevel match must increase as the number of levels increases.
2. If, for any particular query time taken remains the same after a level, then justify why such a case exists.
The representation of the X-axis and Y-axis is given below:
Horizontal-Axis: Query Number.
Vertical-Axis: Time Taken in ms (milliseconds) to execute Exact Match for Single Method Queries.

Graph 6.3: Time Taken for Exact Multilevel Match (Single Method Queries)

1. Validation of Query Results for Graph 6.3

This section validates a couple of queries shown in the Graph 6.3. It considers two queries: Query Q1 and Query Q2.

i) Results of Query Q1

The requirements of Query Q1 is presented in the Table 6.5(a) below:

(TAT represents Turn Around Time)

<table>
<thead>
<tr>
<th>Query No</th>
<th>Method Signature</th>
<th>Pre-Conditions</th>
<th>Post-Conditions</th>
<th>Synch-Impl</th>
<th>QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>void deposit(string, int)</td>
<td>acctid! = null</td>
<td>newbalance &gt; oldbalance</td>
<td>ME-Semaphores</td>
<td>TAT, Throughput</td>
</tr>
</tbody>
</table>

Table 6.5(a): Query Q1 Requirements
Table 6.5(b): Time Taken Results for Query Q1

The time taken by the query Q1 at each level is plotted in the Graph 6.3 and can be seen in the Table 6.5(b). As seen in the Graph 6.3, with each level the time gradually increases, hence validating the claim that Multilevel match takes more time than a single level match. The time taken at each level is the sum of all the times at the previous levels and the corresponding level, which validates the claim that time taken increases as levels increase in Multilevel match.

i) Results of Query Q2

Consider Query Q2 to address the second objective in presenting the Graph 6.3. The requirements of Query Q2 are presented in the Table 6.5(c) below:

Table 6.5(c): Query Q2 Requirements

Table 6.5(d): Time Taken Results for Query Q2
As seen in the Table 6.5(d), and shown in the Graph 6.3, the synchronization and the QoS level time taken is same as the semantic level. The query did not match to any components at the semantic level, and hence no components were returned, which can be seen in Table 6.5(e).

<table>
<thead>
<tr>
<th>Query No.</th>
<th>CompID - Syntactic Exact Match</th>
<th>CompID - Semantic Exact Match</th>
<th>CompID- Synch. Exact Match</th>
<th>CompID - QoS Exact Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2</td>
<td>22</td>
<td>Nil</td>
<td>Nil</td>
<td>Nil</td>
</tr>
</tbody>
</table>

Table 6.5(e): CompID/IDs returned for Query Q2

Since, the components matched at the semantic level form the input to the next level, the time taken to execute when the synchronization level or the synchronization and the QoS levels are selected is the time taken to execute till the semantic level. As, after that even if the synchronization and the QoS matchers are selected they are not executed as there are no input components against which these matchers have to execute. So, the time taken to execute at the Synchronization level is:

Time Taken to Execute [ {Previous Levels} + Synch.]
= Time Taken to Execute [ {Syntactic + Semantic} + Synch.]
= [ 32.421 + 0.0] = 32.421 msec

Hence, the Graph 6.3 shows same time taken for Query Q9 for the three levels: the semantic, the synchronization and the QoS.

Part 2: Graph 6.4 presenting the time taken for Relaxed match of a Single Method Queries.

Objective: The main objective of presenting this graph is to show:

1. The time taken to execute a Multilevel match is more than a single level match and the time taken to execute Multilevel match must increase as the number of levels increases.
2. Justify the case if, such a case exists as to why the time taken to execute is same for two or more levels for a particular query.
3. Since, applying relaxation to the matches is to include wide category of components than the Exact match, check if the time taken to implement Relaxed match is more than the Exact match for a particular query.

The representation of the X-axis and Y-axis is given below:

Horizontal-Axis: Query Number.

Vertical-Axis: Time Taken in milliseconds implementing Relaxed Multilevel Match for Single Method Queries.

Graph 6.4: Time Taken for Relaxed Multilevel Match (Single Method Queries)

1. Validation of Query Results for Graph 6.4

This section validates the objective in presenting the Graph 6.4 using Query Q1 and Query Q19.
ii) **Results of Query Q1**

The requirements of Query Q1 are presented in the Table 6.5(f) below:

(TAT represents Turn Around Time)

<table>
<thead>
<tr>
<th>Query No</th>
<th>Method Signature</th>
<th>Pre-Conditions</th>
<th>Post-Conditions</th>
<th>Synch-Impl</th>
<th>QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>Void deposit(string, int)</td>
<td>acctid!=null</td>
<td>newbalance&gt;oldbalance</td>
<td>ME-Semaphores</td>
<td>TAT, Throughput</td>
</tr>
</tbody>
</table>

**Table 6.5(f): Query Q1 Requirements**

<table>
<thead>
<tr>
<th>Query No</th>
<th>Time Taken (msec)-Syntactic</th>
<th>Time Taken (msec)-Semantic</th>
<th>Time Taken (msec)-Synchronization</th>
<th>Time Taken (msec)-QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>22.16</td>
<td>58.156</td>
<td>65.78</td>
<td>67.39</td>
</tr>
</tbody>
</table>

**Table 6.5(g): Time Taken Results for Query Q1**

The time taken by the query Q1 at each level is plotted in the Graph 6.4, and can be seen in the Table 6.5(g). As seen in the Graph 6.4, with each level the time gradually increases, hence validating the claim that Multilevel match takes more time than a single level match. The time taken at each level is the sum of all the times at the previous levels and the corresponding level.

Comparing the time taken values for the query Q1 in both the Tables 6.5(b) and 6.5(g), the time taken to return the Relaxed match components is more than the Exact match components.

iii) **Results of Query Q19**

Consider Query Q19 to address the second objective in presenting the Graph 6.4. The requirements of Query Q19 are presented in the Table 6.5(c) below:
<table>
<thead>
<tr>
<th>Query No</th>
<th>Method Signature</th>
<th>Pre-Conditions</th>
<th>Post-Conditions</th>
<th>Synch-Impl</th>
<th>QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q19</td>
<td>float get(string, float)</td>
<td>acctid!=null; balance&gt;0.0;</td>
<td>newbalance &gt; oldbalance</td>
<td>ME-Semaphores</td>
<td>TAT Throughput</td>
</tr>
</tbody>
</table>

Table 6.5(h): Query Q19 Requirements

<table>
<thead>
<tr>
<th>Query No</th>
<th>Time Taken (msec)-Syntactic</th>
<th>Time Taken (msec)-Semantic</th>
<th>Time Taken (msec)-Synchronization</th>
<th>Time Taken (msec)-QoS</th>
</tr>
</thead>
</table>

Table 6.5(i): Time Taken Results for Query Q19

The time taken to execute the levels remains same for Query Q19, as seen in the Table 6.5(i), and as shown in the Graph 6.4. The query requirement Q19 did not return any components at the syntactic level and since the components from syntactic match are sent as input to the semantic match for further matching, and so on, the time taken to execute is only the syntactic level time which is 19.562 (msec). Table 6.5(e) shows the component results returned for Query 19.

<table>
<thead>
<tr>
<th>Query No</th>
<th>CompID - Syntactic Exact Match</th>
<th>CompID - Semantic Exact Match</th>
<th>CompID- Synch. Exact Match</th>
<th>CompID - QoS Exact Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q19</td>
<td>Nil</td>
<td>Nil</td>
<td>Nil</td>
<td>Nil</td>
</tr>
</tbody>
</table>

Table 6.5(e): CompID/IDs returned for Query Q19

Hence, the Graph 6.4 shows the same time taken for Query Q19 for all the four levels: the syntactic, the semantic, the synchronization and the QoS.
This ends the discussion for the Time Taken to execute Single Method Queries for both the Exact and Relaxed matches.

6.2.1(c) Precision - Multilevel Match (Single Method Queries)

This section describes the last metrics considered for experimental validation of this prototype: Precision. Precision, as stated in earlier chapters is calculated from user’s feedback. It is used for measuring the performance and is calculated as:

\[
\text{Precision} = \frac{\text{Number of Relevant Components}}{\text{Total Number of Components Retrieved}}
\]

Precision is calculated for the relevant components at each level and graphical presentation of the Precision values at each level to the query is shown, which can be used to indicate the effectiveness of the Multilevel search process. This section is divided into two parts: the first part of the section presents the graphs showing the Precision for Single Method Queries for Exact match. The second part presents the graphs of the time taken for Relaxed match.

Part 1: Graph 6.5 presenting Precision for Exact match of Single Method Queries, which is used as a measure of performance.

Objective: The main objective of presenting this Graph 6.5 is to show:

1. The Precision of relevant components returned increases across the multiple levels.
2. Determine why, if any particular query does not show increase in the Precision at all levels or shows the same Precision at all the four levels.

The representation of the X-axis and Y-axis is given below:

Horizontal-Axis: Query Number.
Vertical-Axis: Precision calculated in executing Exact Multilevel Match for Single Method Queries.
Graph 6.5: Calculated Precision for Exact Multilevel Match (Single Method Queries)

1. Validation of Query Results for Graph 6.5

This section validates a couple of queries shown in the Graph 6.5. It considers two queries: Query Q1 and Query Q5.

i) Results of Query Q1

The requirements of Query Q1 are presented in the Table 6.6(a) below:

(TAT represents Turn Around Time)

<table>
<thead>
<tr>
<th>Query No</th>
<th>Method Signature</th>
<th>Pre-Conditions</th>
<th>Post-Conditions</th>
<th>Synch-Impl</th>
<th>QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>void deposit(string, int)</td>
<td>acctid!=null</td>
<td>newbalance&gt;oldbalance</td>
<td>ME-Semaphores</td>
<td>TAT, Throughput</td>
</tr>
</tbody>
</table>

Table 6.6(a): Query Q1 Requirements
Table 6.6(b): Precision Results for Query Q1

The Precision as shown in the Graph 6.5 and the Table 6.6(b) shown above, indicates the highest Precision components were returned to the user for the query Q1. As the returned components match exactly under each of the levels, the Precision is same for all the levels. Below is the tabular representation of components returned for Query Q1 in Table 6.6(c):

<table>
<thead>
<tr>
<th>Query No.</th>
<th>CompID - Syntactic Exact Match</th>
<th>CompID - Semantic Exact Match</th>
<th>CompID - Synch. Exact Match</th>
<th>CompID - QoS Exact Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 6.6(c): CompID/IDs returned for Query Q1

As the number of components returned at each level is one so, the Precision at each level is 1.

ii) Results of Query Q5

The requirements of Query Q5 are presented in the Table 6.6(d) below:

(TAT represents Turn Around Time)

<table>
<thead>
<tr>
<th>Query No.</th>
<th>Method Signature</th>
<th>Pre-Conditions</th>
<th>Post-Conditions</th>
<th>Synch-Impl</th>
<th>QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q5</td>
<td>void withdraw(string, double)</td>
<td>acctid!=null balance&gt;0.0</td>
<td>acctid!=null newbalance&gt;oldbalance</td>
<td>ME-Mutexes</td>
<td>TAT, Throughput</td>
</tr>
</tbody>
</table>

Table 6.6(d): Query Q5 Requirements
<table>
<thead>
<tr>
<th>Query No</th>
<th>Precision - Syntactic</th>
<th>Precision - Semantic</th>
<th>Precision - Synchronization</th>
<th>Precision - QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q5</td>
<td>0.25</td>
<td>0.3333</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 6.6(e): Precision Results for Query Q5**

The Precision in the case of query Q5 gradually increases as the level increases as seen in Table 6.6(e). Since, the Precision is calculated based on user’s feedback and as the above Table 6.6(e) shows highly relevant components are produced at the end of all the levels. Hence, the Precision of relevant components returned increases across the multiple levels.

**Part 2:** Graph 6.6 presents the Precision for Relaxed match of Single Method Queries. The objectives here remain the same as with the Precision for Exact Single Method Queries.

The representation of the X-axis and Y-axis is given below:

**Horizontal-Axis:** Query Number.

**Vertical-Axis:** The Precision obtained in executing the Relaxed Multilevel Match for Single Method Queries.
All the objectives set for the Precision of Relaxed Multilevel Match have been met. To avoid repetition the objectives have not been discussed in detail for the Precision of Relaxed Multilevel match for Single Method Queries.

This ends the discussion of the three metrics considered: Number of Components returned, Time Taken to execute the match and the Precision calculated from user’s feedback for the Single Method Queries. The next section 6.2.2(a) discusses the same metrics in case of Multiple Method Queries for both the Exact and Relaxed Match.

6.2.2(a) Components Returned - Multilevel Match (Multiple Method Queries)

This section presents the graphs and explanation to some of the queries consisting of multiple methods. Multiple Method Queries include more than one method. This section is divided into two parts: the first part of the section represents the graphs showing the components returned for Multiple Method Queries for Exact match. The second part presents the graphs for Relaxed match.
**Part 1:** Graph 6.7 shows the number of components returned for the Multiple Method Queries for Exact Match. The objective of presenting the graph for components returned for Multiple Method Queries is same as the Single Method Queries for Exact match presented in subsection 6.2.1(a), which are:

Objective: The main objective of presenting this graph is to show:

1. The gradual decrease in the number of components at each level is consistent with the hypothesis that Multilevel matching gives fewer components than single level matching.
2. If, for any particular query the number of components returned is same for a Multilevel match and single level match, results of such queries needs to be justified.

The representation of the X-axis and Y-axis is given below:

Horizontal-Axis: Query Number.

Vertical-Axis: Number of Components returned executing the Exact Multilevel Match for Multiple Method Queries.

**Graph 6.7: Number of Components returned for Exact Multilevel Match**  
(Multiple Method Queries)
1. Validation of Query Results for Graph 6.7

This section validates a couple of queries shown in the Graph 6.7. It considers two queries: Query Q2 and Query Q5.

i) Results of Query Q2
The requirements of Query Q2 is presented in the Table 6.7(a) below:
(TAT represents Turn Around Time)

<table>
<thead>
<tr>
<th>Query No</th>
<th>Method Signature</th>
<th>Pre-Conditions</th>
<th>Post-Conditions</th>
<th>Synch-Impl</th>
<th>QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2</td>
<td>1.void Deposit (string, int) acctid!=null</td>
<td>newbalance&gt;oldbalance</td>
<td>ME - Semaphores</td>
<td>TAT, Throughput</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.void Withdraw (string, float) acctid!=null</td>
<td>newbalance&lt;oldbalance</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.7(a): Query Q2 Requirements

The number of components returned is shown in Table 6.7(b) below:

<table>
<thead>
<tr>
<th>Query No</th>
<th>CompID - Syntactic Exact Match</th>
<th>CompID - Semantic Exact Match</th>
<th>CompID - Synch. Exact Match</th>
<th>CompID - QoS Exact Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2</td>
<td>1,2,3,5,7,8,9,10,11,13,16,21,23,28,36,37,40,39,41,42,47,</td>
<td>1,2,3,5,7,8,9,10,11,13,16,21,23,36,28,37,39,40,42,47</td>
<td>2,10,11,21,23,47</td>
<td>2,11,21,23</td>
</tr>
</tbody>
</table>

Table 6.7(b): CompID/IDs returned for Query Q2

As seen in Table 6.7(b), for query requirement Q2 there is a gradual decrease in the number of components from syntactic to QoS levels, suggesting that the Multilevel
match returns smaller set of components, which are more relevant than single level matching also satisfying one of the objectives of the experimentation.

ii) **Results of Query Q5**

The requirements of Query Q5 is presented in the Table 6.7(c) below:

(TAT represents Turn Around Time)

<table>
<thead>
<tr>
<th>Query No</th>
<th>Method Signature</th>
<th>Pre-Conditions</th>
<th>Post-Conditions</th>
<th>Synch-Impl</th>
<th>QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q5</td>
<td>1. void Deposit (string, int) acctid!=null</td>
<td>newbalance&lt;oldbalance</td>
<td>ME-Semaphores</td>
<td>TAT, Throughput</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. void Withdraw (string, int) acctid!=null</td>
<td>newbalance&gt;oldbalance</td>
<td>ME-Semaphores</td>
<td>TAT, Throughput</td>
<td></td>
</tr>
</tbody>
</table>

**Table 6.7(c): Query Q5 Requirements**

The number of components returned is presented below in Table 6.7(d):

<table>
<thead>
<tr>
<th>Query No.</th>
<th>CompID - Syntactic Exact Match</th>
<th>CompID - Semantic Exact Match</th>
<th>CompID - Synch. Exact Match</th>
<th>CompID - QoS Exact Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

**Table 6.7(d): CompID/IDs returned for Query Q5**

As seen from the Table 6.7(d) the query requirement Q1 returns only single valid component with CompID = 6 satisfying all the levels and so the Graph 6.7 shows all the levels as equal for the Query Q5. Since, the components returned at the syntactic level form the input to the corresponding levels only one component has been returned at the
syntactic level and it is checked across all the levels. The number of components returned here are same for the single level matching and Multilevel matching.

**Part 2:** Graph 6.8 presents the number of components returned for exact match of Single Method Query. The objective of presenting the graph for components returned for Multiple Method Queries is same as the Single Method Queries for relaxed match presented in subsection 6.2.1(a), which are:

**Objective:** The main objective of presenting this graph is to show:

1. The gradual decrease in the number of components at each level, proving the fact that Multilevel match gives fewer relevant components than single level matching.

2. Determine whether the components returned for the Relaxed match includes those components returned for Exact match or not. This is checked to see if the set of components returned under Relaxed match contains at least the components returned from Exact match or not, if not more.

3. If, for any particular query the number of components returned is the same for a Multilevel match and single level match, results of such queries need to be justified.

The representation of the X-axis and Y-axis is given below:

**Horizontal-Axis:** Query Number.

**Vertical-Axis:** Number of Components returned executing Relaxed Multilevel Match for Multiple Method Queries.
Graph 6.8: Number of Components returned for Relaxed Multilevel Match
(Multiple Method Queries)

1. Validation of Query Results for Graph 6.8

This section validates a couple of queries shown in the Graph 6.8. It considers two queries: Query Q1 and Query Q9.

iii) Results of Query Q1

The requirements of Query Q1 is presented in the Table 6.8(a) below:

(TAT represents Turn Around Time)

<table>
<thead>
<tr>
<th>Query No</th>
<th>Method Signature</th>
<th>Pre-Conditions</th>
<th>Post-Conditions</th>
<th>Synch-Impl</th>
<th>QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>1. void Deposit(string, float)</td>
<td>acctid!=null</td>
<td>newbalance&gt;oldbalance</td>
<td>ME-Mutexes</td>
<td>TAT, Throughput</td>
</tr>
<tr>
<td></td>
<td>2. void Withdraw(string, float)</td>
<td>amount&gt;0.0</td>
<td>newbalance&lt;oldbalance</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.8(a): Query Q1 Requirements
The number of components returned is presented below in Table 6.8(b):

<table>
<thead>
<tr>
<th>Query No.</th>
<th>CompID - Syntactic Relaxed Match</th>
<th>CompID - Semantic Relaxed Match</th>
<th>CompID - Synch. Relaxed Match</th>
<th>CompID - QoS Relaxed Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>1,2,3,5,7,8,9,10,11,12,13,15,16,21,20,23,26,28,29,30,31,36,37,38,39,40,41,42,47</td>
<td>2,3,5,7,8,9,10,12,13,15,16,21,23,28,26,30,31,37,40,42</td>
<td>2,3,5,8,10,13,16,21,23,26,28,30,37,42,40</td>
<td>2,3,5,10,13,16,21,23,28,26,30,37,40,42</td>
</tr>
</tbody>
</table>

**Table 6.8(b): CompID/IDs returned for Query Q1 in Relaxed Match**

The components returned at each level for query Q1 are plotted in the Graph 6.8 and a gradual decrease in the number of components can be seen that are more relevant in case of Multilevel matching than for the single level matching.

<table>
<thead>
<tr>
<th>Query No.</th>
<th>CompID - Syntactic Exact Match</th>
<th>CompID - Semantic Exact Match</th>
<th>CompID- Synch. Exact Match</th>
<th>CompID - QoS Exact Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>1,2,3,5,7,8,9,10,11,13,16,21,23,28,36,37,39,40,41,42,47</td>
<td>2,3,13,16,23,28</td>
<td>2,23</td>
<td>2,23</td>
</tr>
</tbody>
</table>

**Table 6.8(c): CompID/IDs returned for Query Q1 in Exact Match**

Also, When the same query has been given under Exact match it has returned 2 components as seen in Table 6.8(c). The Relaxed match as seen in Table 6.8(b) has returned 14 components, including the two components returned in the Exact match, CompID: 2,23.

iv) Results of Query Q20 and Q9
Consider Query Q20 to address the third objective in presenting the Graph 6.8.

The requirement of Query Q20 is presented in the Table 6.8(d) below:

(TAT represents Turn Around Time)

<table>
<thead>
<tr>
<th>Query No</th>
<th>Method Signature</th>
<th>Pre - Conditions</th>
<th>Post - Conditions</th>
<th>Synch - Impl</th>
<th>QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q20</td>
<td>1. void put (double)</td>
<td>amount&gt;0.0</td>
<td>newbalance&gt;oldbalance</td>
<td>RW-Semaphores</td>
<td>Throughput</td>
</tr>
<tr>
<td></td>
<td>2. void get (double)</td>
<td>balance&gt;0.0</td>
<td>newbalance&lt;oldbalance</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. float getbalance (string)</td>
<td>acctid!=null</td>
<td>acctid!=null</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.8(d): Query Q20 Requirements

The number of components returned is presented below in Table 6.8(e):

<table>
<thead>
<tr>
<th>Query No.</th>
<th>CompID - Syntactic Relaxed Match</th>
<th>CompID - Semantic Relaxed Match</th>
<th>CompID- Synch. Relaxed Match</th>
<th>CompID - QoS Relaxed Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q20</td>
<td>4,27,38</td>
<td>4,27,38</td>
<td>4,27,38</td>
<td>4,27,38</td>
</tr>
</tbody>
</table>

Table 6.8(e): CompID/IDs returned for Query Q20

The results of Graph 6.8 show the same number of components for all the levels, the same can be validated in the Table 6.8(e), which shows the result of the query Q20 where, same number of components is returned at each level.

Among the set of queries given under Multiple Method Queries for Relaxed match some of the queries have returned same number of components at two levels as seen in the Graph 6.8, the syntactic and the semantic have returned the same number of components, so did the synchronization, and the QoS levels. The results of Query Q9 are used to validate the same.

The requirements of Query Q9 is presented in the Table 6.8(f) below:
<table>
<thead>
<tr>
<th>Query No</th>
<th>Method Signature</th>
<th>Pre - Conditions</th>
<th>Post - Conditions</th>
<th>Synch - Impl</th>
<th>QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q9</td>
<td>1. void get(int)</td>
<td>balance&gt;0</td>
<td>newbalance&lt;oldbalance</td>
<td>RW-Semaphores</td>
<td>Throughput</td>
</tr>
<tr>
<td></td>
<td>2. void put(int)</td>
<td>amount&gt;0</td>
<td>newbalance&gt;oldbalance</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 6.8(f): Query Q9 Requirements**

The number of components returned is presented below in Table 6.8(e):

<table>
<thead>
<tr>
<th>Query No.</th>
<th>CompID - Syntactic Relaxed Match</th>
<th>CompID - Semantic Relaxed Match</th>
<th>CompID - Synch. Relaxed Match</th>
<th>CompID - QoS Relaxed Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q9</td>
<td>14,18,22,24,35</td>
<td>14,18,22,24,35</td>
<td>18,22</td>
<td>18,22</td>
</tr>
</tbody>
</table>

**Table 6.8(g): CompID/IDs returned for Query Q9**

From the Table 6.8(g), it can be seen that same set of components have matched at both the syntactic and the semantic levels. But, at the synchronization level the components returned have reduced to 2, and the same components hold good for the QoS match as well. Although, the upper two levels have returned the same number of components and the lower two levels have same number of components, the Multilevel match still is better than the single level match.

This concludes the validation of the graphs presenting the components returned for both the Exact and the Relaxed matches for Multiple Method Queries.

**6.2.2(b) Time Taken - Multilevel Match (Multiple Method Queries)**

This section is divided into two parts: the first part of the section presents the graphs showing the time taken to execute Exact match for Multiple Method Queries. The second part presents the time taken graph for Relaxed match.
**Part 1:** Graph 6.9 presenting the time taken to execute Exact match of Multiple Method Queries. The objective of presenting the graph for time taken to execute Exact match for Multiple Method Queries is same as the Single Method Queries presented in subsection 6.2.1(b), which are:

Objective: The main objective of presenting this graph is to show:

1. The time taken to execute a Multilevel match is more than a single level match and the time taken to implement Multilevel match must increase as the number of levels increases.
2. If, for any particular query time taken remains the same after a level, then justify why such a case exists.

The representation of the X-axis and Y-axis is given below:

**Horizontal-Axis:** Query Number.

**Vertical-Axis:** Time Taken to execute Exact Multilevel Match for Multiple Method Queries.

**Graph 6.9: Time Taken to execute Exact Multilevel Match**
(Multiple Method Queries)

1. Validation of Query Results for Graph 6.9

This section validates a couple of queries shown in the Graph 6.9. It considers two queries: Query Q2 and Query Q6.

i) Results of Query Q2

The requirements of Query Q2 is presented in the Table 6.9(a) below:
(TAT represents Turn Around Time)

<table>
<thead>
<tr>
<th>Query No</th>
<th>Method Signature</th>
<th>Pre-Conditions</th>
<th>Post-Conditions</th>
<th>Synch-Impl</th>
<th>QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2</td>
<td>1. void Deposit(string, float) acctid!=null</td>
<td>newbalance&gt;oldbalance</td>
<td>ME-Mutexes</td>
<td>TAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. void Withdraw(string, float) acctid!=null</td>
<td>newbalance&lt;oldbalance</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.9(a): Query Q2 Requirements

<table>
<thead>
<tr>
<th>Query No</th>
<th>Time Taken(msec)-Syntactic</th>
<th>Time Taken(msec)-Semantic</th>
<th>Time Taken(msec)-Synchronization</th>
<th>Time Taken (msec)-QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2</td>
<td>20.468</td>
<td>33.99</td>
<td>36.93</td>
<td>40.07</td>
</tr>
</tbody>
</table>

Table 6.9(b): Time Taken Results for Query Q2

Graph 6.9, presents the time taken to execute exact Multilevel match for multiple method queries. The above Table 6.9(b) validates the first objective of presenting the
Graph 6.9, which is: the time taken increases gradually as the level increases and the time taken to execute Multilevel match is more than the single level match.

ii) Results of Query Q6

The results of Query Q6 are used to validate the second objective of presenting the Graph 6.9. The requirements of Query Q6 is presented in the Table 6.9(c) below:

<table>
<thead>
<tr>
<th>Query No</th>
<th>Method Signature</th>
<th>Pre-Conditions</th>
<th>Post-Conditions</th>
<th>Synch-Impl</th>
<th>QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q6</td>
<td>1. void Deposit(int)</td>
<td>acctid&gt;0</td>
<td>newbalance&gt;oldbalance</td>
<td>ME-Mutexes</td>
<td>Portability, Priority</td>
</tr>
<tr>
<td></td>
<td>2. void Withdraw(int)</td>
<td>acctid&gt;0</td>
<td>newbalance&lt;oldbalance</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.9(c): Query Q6 Requirements

<table>
<thead>
<tr>
<th>Query No</th>
<th>Time Taken(msec)-Syntactic</th>
<th>Time Taken(msec)-Semantic</th>
<th>Time Taken(msec)-Synchronization</th>
<th>Time Taken (msec)-QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q6</td>
<td>15.218</td>
<td>25.156</td>
<td>25.156</td>
<td>25.156</td>
</tr>
</tbody>
</table>

Table 6.9(d): Time Taken Results for Query Q6

As seen in the Table 6.9(d), and shown in the Graph 6.9, the time taken to execute the synchronization and the QoS level is same as the semantic level. As discussed previously in subsection 6.2.1(b), if no components are returned at a particular level then the time taken to execute the next levels is the time taken to execute till that particular level. Table 6.9(e) shows the components returned for query Q6.
### Table 6.9(e): CompID/IDs returned for Query Q6

<table>
<thead>
<tr>
<th>Query No.</th>
<th>CompID - Syntactic Exact Match</th>
<th>CompID - Semantic Exact Match</th>
<th>CompID - Synch. Exact Match</th>
<th>CompID - QoS Exact Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q6</td>
<td>22</td>
<td>Nil</td>
<td>Nil</td>
<td>Nil</td>
</tr>
</tbody>
</table>

Hence, the Graph 6.9 shows same time taken for Query Q6 for the three levels: the semantic, the synchronization and the QoS.

**Part 2:** Graph 6.10 presenting the time taken to execute Relaxed Multilevel match for Multiple Method Queries. As stated in the precious sections, the objective of presenting the graph for time taken to execute the Relaxed match for Multiple Method Queries is same as the Single Method Queries presented in subsection 6.2.1(b), which are:

**Objective:** The main objective of presenting this graph is to show:

1. The time taken to execute Multilevel match must increase as the number of levels increases and to check if in most cases, time taken to execute a Multilevel match is more than a single level match.
2. If, for any particular query time taken remains the same after a level, then justify why such a case exists.
3. Since, applying relaxation to the matches is to include wide category of components than the Exact match, check if the time taken to implement Relaxed match is more than the Exact match for a particular query.

The representation of the X-axis and Y-axis is given below:

**Horizontal-Axis:** Query Number.

**Vertical-Axis:** Time Taken to execute Relaxed Multilevel Match for Multiple Method Queries.
1. Validation of Query Results for Graph 6.10

This section considers the same queries Query Q2 and Query Q6 in validating the objective in presenting the Graph 6.10.

iii) Results of Query Q2
The requirements of Query Q2 is presented in the Table 6.10(a) below:
(TAT represents Turn Around Time)
### Table 6.10(a): Query Q2 Requirements

<table>
<thead>
<tr>
<th>Query No</th>
<th>Method Signature</th>
<th>Pre-Conditions</th>
<th>Post-Conditions</th>
<th>Synch-Impl</th>
<th>QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2</td>
<td>1. void Deposit (string, float) acctid!=null</td>
<td>acctid!=null</td>
<td>newbalance&gt;oldbalance</td>
<td>ME-Mutexes</td>
<td>TAT</td>
</tr>
<tr>
<td></td>
<td>2. void Withdraw (string, float) acctid!=null</td>
<td>acctid!=null</td>
<td>newbalance&lt;oldbalance</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 6.10(b): Time Taken Results for Query Q2

<table>
<thead>
<tr>
<th>Query No</th>
<th>Time Taken(msec)-Syntactic</th>
<th>Time Taken(msec)-Semantic</th>
<th>Time Taken(msec)-Synchronization</th>
<th>Time Taken (msec)-QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2</td>
<td>24.796</td>
<td>51.068</td>
<td>55.857</td>
<td>58.201</td>
</tr>
</tbody>
</table>

Graph 6.10, presents the time taken to execute exact Multilevel match for multiple method queries. As seen in Table 6.10(b), the time taken to execute Relaxed Multilevel match gradually increases as the levels increase and the time taken to execute Multilevel match is more than the single level match for this query.

Compare the values in Table 6.9(b) and Table 6.10(b) showing the time taken to execute the Exact and the Relaxed matches for the same query Q2. At each level, the time taken to execute in case of the Relaxed match is more than the Exact match for the considered query Q2.

iv) Results of Query Q6

The results of Query Q6 are used to validate the second objective of presenting the Graph 6.10. The requirements of Query Q6 is presented in the Table 6.10(c) below:
Table 6.10(c): Query Q6 Requirements

<table>
<thead>
<tr>
<th>Query No</th>
<th>Method Signature</th>
<th>Pre-Conditions</th>
<th>Post-Conditions</th>
<th>Synch-Impl</th>
<th>QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q6</td>
<td>1. void Deposit(int)</td>
<td>acctid&gt;0</td>
<td>newbalance&gt;oldbalance</td>
<td>ME-Mutexes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. void Withdraw(int)</td>
<td>acctid&gt;0</td>
<td>newbalance&lt;oldbalance</td>
<td>Priority</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.10(d): Time Taken Results for Query Q6

As seen in the Table 6.10(d), and shown in the Graph 6.10, the time taken to execute gradually increases from the syntactic to the QoS level unlike the values shown in the Table 6.9(d), where the time taken remains the same across the semantic, the synchronization and the QoS levels. Table 6.10(e) shows the components returned for query Q6 for Relaxed match unlike Table 6.9(e) where no components were returned for the Exact mach after the syntactic level. Hence, the time taken to execute shows gradual increase at each level for Query Q6.

Table 6.10(e): CompID/IDs returned for Query Q6

<table>
<thead>
<tr>
<th>Query No</th>
<th>CompID - Syntactic Exact Match</th>
<th>CompID - Semantic Exact Match</th>
<th>CompID- Synch. Exact Match</th>
<th>CompID - QoS Exact Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q6</td>
<td>14,18,22,24,35</td>
<td>14,18,22,24,35</td>
<td>14,24,35</td>
<td>14,24,35</td>
</tr>
</tbody>
</table>
For the experiments conducted for Relaxed match for Multiple Method Queries, all the considered queries have returned components and executed all the levels so, the Graph 6.10 does not show any query presenting the time taken to execute at more than two levels as equal. But, as in the case of Exact match, if the query does not return any components at a particular level, then the time taken to execute the next level is the time taken to execute till that particular level.

6.2.2(c) Precision- Multilevel Match (Multiple Method Queries)

This section is divided into two parts: the first part of the section presents the graphs showing the Precision for Single Method Query for exact match. The second part presents the time taken graphs for relaxed match.

Part 1: Graph 6.11 presenting the Precision calculated in executing Exact Multilevel match for Multiple Method Queries, which is used as a measure of performance. Objective: The main objective of presenting this Graph 6.11 is to show:

1. The Precision of relevant components returned increases across the multiple levels.

2. Determine why, if any particular query does not show increase in the Precision at all levels or shows the same Precision at all the four levels.

The representation of the X-axis and Y-axis is given below:

Horizontal-Axis: Query Number.

Vertical-Axis: Precision calculated in executing Exact Multilevel Match for Multiple Method Queries.
Graph 6.11: Calculated Precision for Exact Multilevel Match (Multiple Method Queries)

1. Validation of Query Results for Graph 6.11

This section validates a couple of queries shown in the Graph 6.11. It considers two queries: Query Q2 and Query Q5.

i) Results of Query Q2

The requirements of Query Q2 are presented in the Table 6.11(a) below:

(TAT represents Turn Around Time)
<table>
<thead>
<tr>
<th>Query No</th>
<th>Method Signature</th>
<th>Pre-Conditions</th>
<th>Post-Conditions</th>
<th>Synch-Impl</th>
<th>QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2</td>
<td>1.void Deposit (string, float)</td>
<td>acctid!=null</td>
<td>newbalance&gt;oldbalance</td>
<td>MEC Mutexes</td>
<td>TAT, Throughput</td>
</tr>
<tr>
<td></td>
<td>2.void Withdraw(string, double)</td>
<td>acctid!=null</td>
<td>newbalance&lt;oldbalance</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.11(a): Query Q2 Representations

<table>
<thead>
<tr>
<th>Query No</th>
<th>Precision-Syntactic</th>
<th>Precision-Semantic</th>
<th>Precision-Synchronization</th>
<th>Precision-QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2</td>
<td>0.0952</td>
<td>0.10</td>
<td>0.3334</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.11(b): Precision Results for Query Q2

The Precision calculated from user’s feedback is gradually increased as the level increases as seen in Table 6.11(a). As seen in the Graph 6.11, highly relevant components are produced at the end of all the levels. Hence, the Precision of relevant components returned increases across the multiple levels for this query Q2.

ii) Results of Query Q5

The requirements of Query Q5 are presented in the Table 6.11(a) below: (TAT represents Turn Around Time)

<table>
<thead>
<tr>
<th>Query No</th>
<th>Method Signature</th>
<th>Pre-Conditions</th>
<th>Post-Conditions</th>
<th>Synch-Impl</th>
<th>QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q5</td>
<td>1.void Deposit(string, int)</td>
<td>acctid!=null</td>
<td>newbalance&gt;oldbalance</td>
<td>MEC Semaphores</td>
<td>TAT, Throughput</td>
</tr>
<tr>
<td></td>
<td>2.void Withdraw(string, int)</td>
<td>acctid!=null</td>
<td>newbalance&lt;oldbalance</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.11(c): Query Q5 Representations
Table 6.11(d): Precision Results for Query Q5

From the Graph 6.11 and the Table 6.11(b), the calculated Precision for the components returned remains same across all the levels for Query Q5 and highest Precision components were returned at each level. Table 6.11(c) presents the components returned for Query Q5. As the number of components returned at each level is one so, the Precision at each level is 1.

<table>
<thead>
<tr>
<th>Query No</th>
<th>Precision - Syntactic</th>
<th>Precision - Semantic</th>
<th>Precision - Synchronization</th>
<th>Precision - QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.11(c): CompID/IDs returned for Query Q5

<table>
<thead>
<tr>
<th>Query No</th>
<th>CompID Syntactic Exact Match</th>
<th>CompID Semantic Exact Match</th>
<th>CompID Synch. Exact Match</th>
<th>CompID QoS Exact Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Part 2: Graph 6.12 presenting Precision calculated in executing Relaxed Multilevel match for Multiple Method Queries, which is used as a measure of performance. The representation of the X-axis and Y-axis is given below:
Horizontal-Axis: Query Number.
Vertical-Axis: Precision calculated in executing Relaxed Multilevel Match for Multiple Method Queries.
Graph 6.12: Calculated Precision for Relaxed Multilevel Match
(Multiple Method Queries)

The objectives for presenting the Graph 6.12 are same as the objectives for presenting the Graph 6.11. To avoid repetition, explanation of the graph using queries is not given.

This ends the graphical presentation of the experimental results for both the Single Method Queries and the Multiple Method Queries. The next section focuses on the inference that can be made from the experimental graphs obtained.
6.3 Inference from Experimental Graphs

This section focuses on the inferences made from the experimental results. It presents the following inferences graphically:

1. For a given query, check if the Multilevel matching hypothesis that the number of components gradually decrease as the levels increase holds good with the relevant components chosen by the user at each level or not.
2. If the extra time taken in executing Multilevel matching is justifiable to obtain more relevant components.

Part 1: Graphs 6.13 and 6.14 presenting the number of components returned and number of relevant components returned respectively in executing Exact Multilevel match for Single Method Queries.

Objective: The main objective of presenting the Graphs 6.13 and 6.14 is to show:

1. For all the queries given graphically present the total number of components returned and the total number of relevant components chosen by the user.
2. The total number of relevant components chosen by the user decreases as the level increases holding the Multilevel matching hypothesis.

The representation of the X-axis and Y-axis in Graph 6.13 is given below:
Horizontal-Axis: Ratio of Time Taken to execute is w.r.t Syntactic level time.
Vertical-Axis: Sum of the number of components returned at each level for all the given queries for Exact match. (Multiple Method Queries)

The representation of the X-axis and Y-axis in Graph 6.14 is given below:
Horizontal-Axis: Ratio of Time Taken to execute is w.r.t Syntactic level time.
Vertical-Axis: Sum of number relevant components chosen by the user at each level for all the given queries for Exact match. (Multiple Method Queries)
Graphs 6.13 and 6.14 show a gradual decrease in the number components returned and as well as a decrease in the number of relevant components returned. The
The main objective of these graphs is to see if for all the queries considered under a particular category, the trend of gradual decrease in the number of components returned holds good for the relevant components chosen by the user or not.

For all the Multiple Method queries given under Exact match, Table 6.13(a) showing the ratio of sum of time taken to execute at each level w.r.t sum of Syntactic Level Time:

<table>
<thead>
<tr>
<th>Ratio of sum of Syntactic Level Time (msec) w.r.t sum of Syntactic Level Time</th>
<th>Ratio of sum of Semantic Level Time (msec) w.r.t sum of Syntactic Level Time</th>
<th>Ratio of sum of Synch. Level Time (msec) w.r.t sum of Syntactic Level Time</th>
<th>Ratio of sum of QoS Level Time (msec) w.r.t sum of Syntactic Level Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.703</td>
<td>1.809</td>
<td>1.852</td>
</tr>
</tbody>
</table>

**Table 6.13(a): Time Taken w.r.t Syntactic Level Time**

For all the multiple method queries given under Exact Match, Table 6.13(b) showing the sum of the number of components returned at each level:

<table>
<thead>
<tr>
<th>Sum of No. of Comps - Syntactic Level</th>
<th>Sum of No. of Comps - Semantic Level</th>
<th>Sum of No. of Comps - Synch. Level</th>
<th>Sum of No. of Comps - QoS Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>221</td>
<td>187</td>
<td>45</td>
<td>27</td>
</tr>
</tbody>
</table>

**Table 6.13(b): Sum of No. of Comps Returned**

For all the multiple method queries given under Exact Match, Table 6.13(c) showing the sum of the number of relevant components returned at each level:
As seen in the graphs above, for the set of queries considered, it is observed that there is a gradual decrease in the relevant components chosen by the user just as the components returned at each level have decreased. Below is the graphical presentation of the same for: Relaxed Match - Multiple Method Queries (Graph 6.15 & Graph 6.16), Exact Match – Single Method Queries (Graph 6.17 & Graph 6.18), and Relaxed Match – Single Method Queries (Graph 6.19 & Graph 6.20). Explanation to these graphs as given for the above graphs is not given to avoid repetition.

Table 6.13(c): Sum of No. of Relevant Comps Returned

<table>
<thead>
<tr>
<th>Sum of No. of Relevant Comps - Syntactic Level</th>
<th>Sum of No. of Relevant Comps - Semantic Level</th>
<th>Sum of No. of Relevant Comps - Synch. Level</th>
<th>Sum of No. of Relevant Comps - QoS Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>117</td>
<td>74</td>
<td>22</td>
<td>15</td>
</tr>
</tbody>
</table>

Part 2: Graphs 6.15 and 6.16 presenting the number of components returned and number of relevant components returned respectively to execute the Relaxed match for Multiple Method Queries.

The representation of the X-axis and Y-axis in Graph 6.15 is given below:
Horizontal-Axis: Ratio of Sum of Time Taken to execute is w.r.t Sum of Syntactic level time.
Vertical-Axis: Sum of the number of components returned at each level for all the given queries for Relaxed match. (Multiple Method Queries)

The representation of the X-axis and Y-axis in Graph 6.16 is given below:
Horizontal-Axis: Ratio of Sum of Time Taken to execute is w.r.t Sum of Syntactic level time.
Vertical-Axis: Sum of the number of Relevant components returned at each level for all the given queries for Relaxed match. (Multiple Method Queries)
Graph 6.15: Sum of Number of Components – Ratio of Sum of Time Taken w.r.t Sum of Syntactic Level Time (Relaxed Match - Multiple Method Queries)

Graph 6.16: Number of Relevant Components – Ratio of Sum of Time Taken w.r.t Sum of Syntactic Level Time (Relaxed Match - Multiple Method Queries)
Part 3: Graphs 6.17 and 6.18 presenting the number of components returned and number of relevant components returned respectively to execute Exact match for Single Method Queries.

The representation of the X-axis and Y-axis in Graph 6.17 is given below:
Horizontal-Axis: Ratio of Sum of Time Taken to execute is w.r.t Sum of Syntactic level time.
Vertical-Axis: Sum of the number of components returned at each level for all the given queries for Exact match. (Single Method Queries)

The representation of the X-axis and Y-axis in Graph 6.16 is given below:
Horizontal-Axis: Ratio of Sum of Time Taken to execute is w.r.t Sum of Syntactic level time.
Vertical-Axis: Sum of the number of Relevant components returned at each level for all the given queries for Exact match. (Single Method Queries)
Graph 6.17: Sum of Number of Components – Ratio of Sum of Time Taken w.r.t Sum of Syntactic Level Time (Exact Match - Single Method Queries)

Graph 6.18: Sum of Number of Relevant Components – Ratio of Sum of Time Taken w.r.t Sum of Syntactic Level Time (Exact Match - Single Method Queries)
Part 4: Graphs 6.19 and 6.20 presenting the number of components returned and number of relevant components returned respectively to execute Relaxed match for Single Method Queries.

The representation of the X-axis and Y-axis in Graph 6.19 is given below:
Horizontal-Axis: Ratio of Sum of Time Taken to execute is w.r.t Sum of Syntactic level time.
Vertical-Axis: Sum of the number of components returned at each level for all the given queries for Relaxed match. (Single Method Queries)

The representation of the X-axis and Y-axis in Graph 6.20 is given below:
Horizontal-Axis: Ratio of Sum of Time Taken to execute is w.r.t Sum of Syntactic level time.
Vertical-Axis: Sum of the number of Relevant components returned at each level for all the given queries for Relaxed match. (Single Method Queries)
Part II: This section presents the average Precision calculated for all the 20 queries given under all the different categories of matches considered.

Objective: The main objective of presenting the Graphs 6.21 is to show:

1. To show that the average Precision calculated increases as the ratio of time taken to execute increases.

2. The time spent in executing is justifiable when compared to the Precision of components returned.

The representation of the X-axis and Y-axis in Graph 6.21 is given below:

Horizontal-Axis: Ratio of Sum of Time Taken to execute is w.r.t Sum of Syntactic level time.

Vertical-Axis: Average Precision calculated for all the Multiple Method Queries executing Exact Match
Graph 6.21: Average Precision Calculated – Ratio of Sum of Time Taken w.r.t Sum of Syntactic Level Time (Exact Match - Multiple Method Queries)

As seen in the Graph 6.21, for the set of multiple method queries considered (20 different queries), the average Precision calculated and the ratio of sum of time taken at each level w.r.t to sum of syntactic level time has increased.

For the set of multiple method queries given under Exact match, Table 6.21(a) shows the ratio of sum of time taken to execute at each level w.r.t sum of syntactic level time:

<table>
<thead>
<tr>
<th>Ratio of sum of Syntactic Level Time (msec) w.r.t sum of Syntactic Level Time</th>
<th>Ratio of sum of Semantic Level Time (msec) w.r.t sum of Syntactic Level Time</th>
<th>Ratio of sum of Synch. Level Time (msec) w.r.t sum of Syntactic Level Time</th>
<th>Ratio of sum of QoS Level Time (msec) w.r.t sum of Syntactic Level Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.703</td>
<td>1.809</td>
<td>1.852</td>
</tr>
</tbody>
</table>

Table 6.21(a): Time Taken w.r.t Syntactic Level Time
For the set of multiple method queries given under Exact match, Table 6.21(b) shows the average Precision calculated:

<table>
<thead>
<tr>
<th>Average Precision- Syntactic</th>
<th>Average Precision- Semantic</th>
<th>Average Precision- Synchronization</th>
<th>Average Precision- QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1675</td>
<td>0.1855</td>
<td>0.3643</td>
<td>0.4583</td>
</tr>
</tbody>
</table>

Table 6.21(b): Average Precision Results

As seen from the Tables 6.21(a) and 6.21(b), for the set of multiple methods (20 different queries) considered the Precision of the components obtained at the end of the QoS match is more than doubled when compared with the Precision of the components at the Syntactic level. The ratio of time taken to execute the levels has also increased from the Syntactic level to the QoS level. The increase in the Precision of the components calculated proves that the quality of components returned is increased in Multilevel matching as compared to Single level matching. Thus, for the set of queries considered the extra time taken in executing Multilevel matching is justifiable as more relevant components are returned which can be seen from the increase in the Precision calculated from user’s feedback. Although the time taken to execute the levels has increased, when compared with the quality of components returned the extra time taken is justifiable.

Below are the graphs presenting the average Precision calculated – Ratio of sum of time taken w.r.t sum of syntactic level time for: Relaxed match of Multiple Method Queries (Graph 6.22), Exact match of Single Method Queries (Graph 6.23), Relaxed match of Single Method Queries (Graph 6.24).

The representation of the X-axis and Y-axis in Graph 6.22 is given below:
Horizontal-Axis: Ratio of Sum of Time Taken to execute is w.r.t Sum of Syntactic level time.
Vertical-Axis: Average Precision calculated for all the Multiple Method Queries executing Relaxed Match.
Graph 6.22: Average Precision Calculated – Ratio of Sum of Time Taken w.r.t Sum of Syntactic Level Time (Relaxed Match - Multiple Method Queries)

The representation of the X-axis and Y-axis in Graph 6.23 is given below:

Horizontal-Axis: Ratio of Sum of Time Taken to execute is w.r.t Sum of Syntactic level time.
Vertical-Axis: Average Precision calculated for all the Single Method Queries executing Exact Match
The representation of the X-axis and Y-axis in Graph 6.24 is given below:

Horizontal-Axis: Ratio of Sum of Time Taken to execute is w.r.t Sum of Syntactic level time.

Vertical-Axis: Average Precision calculated for all the Single Method Queries executing Exact Match
In the above graphs presenting the Precision, for the set of queries considered for the Multiple method and Single method, the Precision calculated gradually increased as the time taken at each level increases. From the graphs, it can be inferred that although some extra time is spent in executing Multilevel as opposed to Single level, the calculated Precision in almost all the cases more than doubled for the Multilevel as opposed to the Single level. Thus more relevant components are obtained at the end of the Multilevel matching than the Single level matching.

This chapter has presented the experimentation analyses of the prototype showing the experimental results and presenting the graphs for the following metrics: Number of components returned, Time Taken to execute, Precision calculated from user’s feedback. It proves the hypothesis of Multilevel matching regarding these metrics and gives explanation to them using graphs. Lastly, two inferences from experimental graphs have been provided for the set of queries considered. The next chapter summarizes the project
by providing conclusions of the work, the contributions made by the project, some of the possible future works, and provides summary of the work done as part of this project.
7. Future Work and Conclusion

This chapter focuses on the future work and conclusion of the project. The first section 7.1 states briefly of the conclusion to the project. The second section 7.2 discusses the contributions of the project. The third section 7.3 discusses the future enhancements to the designed prototype and the matching categories. The final section 7.4 provides the summary of the project.

7.1 Conclusion

The main aim of this project is to incorporate the concept of Multilevel matching in URDS and it achieves this by using the multilevel contracts of the component specifications and matching the contracts at each level. This project has used the previous work of matching criteria proposed by [ZAR96] and [KUM04], modified them to obtain better search results. A prototype has been designed and developed for the selection process of components based on multilevel matching. The prototype that has been developed has been tested over a Bank database and the following conclusions have been drawn from the implementation:

1. Multilevel matching returns a smaller set of more relevant components when compared to single level matching.
2. Multilevel matching takes more time than the normal approach, as it has to check all the levels before returning the result to the user. But in many cases, it has been observed that the extra time taken is worthwhile since it returns more relevant components.
3. The performance measure Precision used to calculate the outcome of the search process increases from the syntactic level to the QoS level. It is always greater at the last level as only the most relevant components are present at the end of the multilevel match process
4. It is observed that the “Relaxed” match always produced more, or an equal number of components than the “Exact” match.
5. An analysis has been made on the increase in time taken and compared against the quality of components being retrieved.
6. The prototype can be implemented for all the levels or only the required levels in hierarchical order.

### 7.2 Contributions

This section describes the contributions of the project, which are as follows:

1. Using Multilevel contracts of component specification and applying matching criteria to each contract level.
2. Some of the matching criteria used have been adapted from previous work [ZAR96], [KUM04], and some of them have been modified to apply to the contracts.
3. Design and Implement a prototype that implements Multilevel matching by matching the contract specifications of the component.
4. To show experimentally that the Multilevel matching of components yields more relevant components when compared to single level matching.
5. The concepts of “Exact” and “Relaxed” have been implemented, and the components that have been retrieved in both the cases have been verified.
6. Graphical representation of the metrics considered: the number of components retrieved, time taken for the matching and the precision calculated from user’s feedback has been shown.
7. Inference has been made from the components retrieved and the time taken, if the extra time the Multilevel match takes is justifiable or not.

Above are the contributions to the project by incorporating Multilevel matching. The next section discusses the future enhancements to the project.
7.3 Future Enhancements

This section discusses the future enhancements that could be incorporated to enhance the prototype as well as the matching criteria.

1. To integrate the prototype with a “Live” URDS.
2. To enhance the matching criteria proposed at each level of the component specification.
3. To incorporate more contracts to the component specification in addition to the existing contracts.
4. To enhance the performance of multilevel matching by giving the feedback of the user i.e. user selected relevant components as the input to the lower levels after the syntactic level so that the number of components that are sent as input reduces and searches can only be made on the components that the user thinks are relevant.
5. To incorporate the proposed matching criteria as a part of the UniFrame Generative Domain Model (UGDM), so that the existing search process can be refined.
6. One of the enhancements to prototype is to maintain a history of queries given and the user feedback to the obtained results.
7. To incorporate this prototype as knowledge to mobile agents so that they can use it to discover more appropriate components.
8. To provide this prototype as a web service so that it can be used from anywhere.
9. To Test this prototype over a larger number of components and measuring its performance.
10. To incorporate more contracts to the component specification in addition to the existing contracts.
11. Providing more access to the user of the existing components if they satisfy certain credentials. For example: incorporating role of the user, a manager has more access to the system than an employee.
With the above-specified extensions, a comprehensive Multilevel matching system can be implemented and integrated into URDS so that the selection process for components can be improved, and better results can be provided to the user with high degree of confidence.

### 7.4 Summary

This project has introduced the concept of Multilevel matching among software components using the Multilevel contracts and matching of the contracts based on user requirements. It proposed and enhanced some of the matching criteria used for matching of components at each level to obtain better results. The matching criteria have been verified and validated using a theoretical case study and also empirically validated using a prototype. The prototype has been tested over a range of components with various queries, and the effectiveness of the prototype and the matching criteria have been verified. Graphical presentations of the metrics used to assess the results are shown. Inferences from the graphs have been made to see if the time taken is justifiable when compared to the quality of components returned. Finally, this project has been successful in proving that the Multilevel match provides smaller set of relevant components than the single level matching thus, better component results are provided to the user.

Thus, the Multilevel matching if incorporated into URDS aids in providing a better search and selection of software components with high degree of confidence.
List of References


Last Referenced: Apr 2006.

[LDA01] Open LDAP. http://www.openldap.org/jldap/overview.html
Last Referenced: Apr 2006.


Last Referenced: Apr 2006.


Last Referenced: Apr 2006.


Last Referenced: Apr 2006.

Last Referenced: Apr 2006.
Last Referenced: Apr 2006.


http://www.cs.iupui.edu/uniFrame/pubs-openaccess/monterey.pdf

http://www.cs.iupui.edu/uniFrame/pubs-openaccess/monterey.pdf

Last referenced: Apr 2006.

[RDN02] Resource Discovery Network-Subject Portal Project.


Last referenced: Apr 2006.

Last referenced: Apr 2006.

Last referenced: Apr 2006.


Appendix A: Case Study to Verify the Matches

Appendix A theoretically verifies the different categories of matches considered for Multilevel matching of software components. The proposed matching criteria are presented using a case study based on a Bank System. Although this case study uses a specific domain, the ideas can be extended to any other application domains. The chosen system for the case study does not represent the entire bank system but uses an instance of the bank system to demonstrate the ideas proposed with respect to the matching at the Synchronization level of the component. Section A.1 provides the overview of the functionality of the Bank System. Section A.2 represents the abstract and concrete components of the proposed system. Section A.3 validates the matching criteria proposed and tries to prove them using the case study.

A.1 Bank System Architecture

The bank system case study provides the basic operations provided by the bank such as to withdraw amount from the account, to get balance the balance in the account and to deposit the amount into the account. A comprehensive model of bank system is not the goal of this case study, but instead the functionality provided by the system is used to represent the matching criteria proposed. The bank system used in this case study consists of five separate components: User Terminal, Coordinator, User Validation Server, Account General Info Server, and Account Operations Server. Figure A.1 represents the bank system and the how the components are composed in the system. The description of functionality of each of the components is given below in Table A.1.

The user’s contact with the bank system is via the User Terminal, which validates the user using Validation Server. Once the user is validated the request is forwarded to the Coordination Server, which routes the user requests to either the Account General Info Server or the Account Operations Server. Based on the above, the bank system is divided to five sub-systems, which is represented in the Figure A.1 given below.
The functionality of the bank system is given in the below Table A.1 and Table A.2 represents the different interfaces of the bank system and the methods they support.

<table>
<thead>
<tr>
<th>Component</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Terminal</td>
<td>This acts as a access point to all users. It validates the user using the services of the Validation Server and Forwards the user requests to the Coordination Server.</td>
</tr>
<tr>
<td>Validation Server</td>
<td>Validates the user trying to access the Bank System.</td>
</tr>
<tr>
<td>Coordination Server</td>
<td>Holds the general information about the account. It forwards the user requests to the respective servers.</td>
</tr>
<tr>
<td>Account General Info Server</td>
<td>Represents the Account information such as the Account ID, Account holder details, type of account (savings/checking) etc.</td>
</tr>
<tr>
<td>Account Operations Server</td>
<td>Represents the operations such as Withdraw, Deposit, getBalance etc.</td>
</tr>
</tbody>
</table>
Table A.2: Interface – Method of Bank System

<table>
<thead>
<tr>
<th>Interface</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAddAccount</td>
<td>String createAccount (String acctID, Float balance)</td>
</tr>
<tr>
<td>IValidate</td>
<td>Boolean validate (String username, String password)</td>
</tr>
<tr>
<td>IAccountInfo</td>
<td>AccountInfo readAccountInfo (String acctID)</td>
</tr>
<tr>
<td></td>
<td>SaveAccountInfo (String acctID, AccountInfo info)</td>
</tr>
<tr>
<td>IAccountGeneralInfo</td>
<td>String getAccountType (String acctID)</td>
</tr>
<tr>
<td></td>
<td>AccountGeneralInfo readAccountGeneralInfo (String acctID)</td>
</tr>
<tr>
<td></td>
<td>SaveAccountGeneralInfo (String acctID, AccountGeneralInfo genInfo)</td>
</tr>
<tr>
<td>IAccountOperation</td>
<td>Float getBalance (String acctID)</td>
</tr>
<tr>
<td></td>
<td>Void withdraw (String acctID, Float amount)</td>
</tr>
<tr>
<td></td>
<td>Void deposit (String acctID, Float amount)</td>
</tr>
</tbody>
</table>

Below Table A.3 represents the component to interface information of the system

<table>
<thead>
<tr>
<th>Component</th>
<th>Interface/Interfaces Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserTerminal</td>
<td>IAddAccount, IValidate, IAccountInfo</td>
</tr>
<tr>
<td>Validation Server</td>
<td>IValidate</td>
</tr>
<tr>
<td>Coordinator</td>
<td>IAddAccount, IAccountInfo</td>
</tr>
<tr>
<td>Account General info Server</td>
<td>IAddAccount, IAccountGeneralInfo</td>
</tr>
<tr>
<td>Account Operations Server</td>
<td>IAddAccount, IAccountOperation</td>
</tr>
</tbody>
</table>

Table A.3: Component – Interface/Interfaces of Bank System

The above mentioned bank system will be used as a case study and different variations of concrete components are created from the abstract components. The section below represents the different variations of the systems.
A.2 Representation of Bank System Abstract & Concrete Components

This section describes the abstract and concrete implementations of the component types specified in the above Bank System. Each system has five concrete implementations of abstract component types UserTerminal, Coordinator, Validation Server, Account General Info Server and Account Operations Server. Below is an example representation of the Bank Server System “P-BANK” with abstract components.

Bank Server System: P-BANK

Abstract Component: UserTerminal

Interfaces:

Provided Interfaces: IAddAccount, IValidate, IAccountInfo
Required Interfaces: IValidate, IAccountGeneralInfo, IAccountOperation

Collaborator Components:

Preprocessing Collaborators: None
Postprocessing Collaborators: Validation Server, Account General Info Server, Coordinator, Account Operations Server

Abstract Component: ValidationServer

Interfaces:

Provided Interfaces: IValidate
Required Interfaces: None

Collaborator Components:

Preprocessing Collaborators: User Terminal
Postprocessing Collaborators: None

Abstract Component: Coordination Server

Interfaces:

Provided Interfaces: IAddAccount, IAccountInfo
Required Interfaces: IAddAccount, IAccountInfo, IAccountGeneralInfo, IAccountOperation

Collaborator Components:
Preprocessing Collaborators: User Terminal
Postprocessing Collaborators: Account General Info Server, Account Operations Server

**Abstract Component:** Account General Info Server

**Interfaces:**
- Provided Interfaces: IAddAccount, IAccountGeneralInfo
- Required Interfaces: IAddAccount, IAccountGeneralInfo

**Collaborator Components:**
- Preprocessing Collaborators: User Terminal, Coordination Server
- Postprocessing Collaborators: None

**Abstract Component:** Account Operations Server

**Interfaces:**
- Provided Interfaces: IAddAccount, IAccountOperation
- Required Interfaces: IAddAccount, IAccountOperation

**Collaborator Components:**
- Preprocessing Collaborators: User Terminal, Coordination Server
- Postprocessing Collaborators: None

To the above bank system, concrete components can be created. Below is the representation of methods supported by the concrete components in the bank system: P-BANK

**Bank Server System:** P-BANK

**Concrete Component:** UserTerminal

- Synchronization Policy: Mutual Exclusion
- Synchronization Implementation: Semaphores
- Interface: IAddAccount
- Method/Methods Supported:
  - void createAccount (String acctID, Float balance)
Pre-condition:
\( \land \text{acctID}. \text{exists} = \text{FALSE} \)

Post-condition:
\( \text{acctID}. \text{exists} = \text{TRUE} \)
\( \land \text{balance (acctID)} = 0.0 \)
\( \land \text{alert Message} \Rightarrow \text{“Account Created with AcctID = ‘X’ and Balance = 0.0”} \)

**Concrete Component**: ValidationServer

Synchronization Policy: Mutual Exclusion
Synchronization Implementation: Mutexes
Interface: IValidate
Method/Methods Supported:
- Boolean validate (String acctID, String Password)

Pre-condition:
\( \neg \text{execute validate (String acctID, String Password)} \)
\( \land \text{acctID}. \text{exists} = \text{TRUE} \)

Post-condition:
If \( \land \text{acctID}. \text{exists} = \text{FALSE} \)
\( \land \text{alert Message} \Rightarrow \text{“Not a Valid User”} \)

Else
\( \land \text{alert Message} \Rightarrow \text{“User Validated”} \)

**Concrete Component**: CoordinationServer

Synchronization Policy: ProducerConsumer
Synchronization Implementation: Locks & Condition Variables
Interface: IAccountInfo
Method/Methods Supported:
- AccountInfo readAccountInfo (String acctID)

Pre-condition:
\( \land \text{acctID}. \text{exists} = \text{TRUE} \)
Post-condition:
   If \( \land \) acctID. exists = FALSE
      \( \land \) alert Message => “Account ID does not exist ”
   Else
      \( \land \) Return Account Info

- SaveAccountInfo (String acctID, AccountInfo info)
  Pre-condition:
     \( \land \) acctID. exists = TRUE
  Post-condition:
     If \( \land \) acctID. exists = FALSE
        \( \land \) alert Message => “Account ID does not exist ”
     Else
        \( \land \) alert Message => “Account Info Saved ”

**Concrete Component**: AccountGeneralInfoServer

- Synchronization Policy: Mutual Exclusion
- Synchronization Implementation: Semaphores
- Interface: IAccountGeneralInfo
- Method/Methods Supported:
  - String getAccountType (String acctID)
    Pre-condition:
       \( \land \) acctID. exists = TRUE
    Post-condition:
       If \( \land \) acctID. exists = FALSE
          \( \land \) alert Message => “Account ID does not exist ”
       Else
          \( \land \) Return Account Type
  - AccountGeneralInfo readAccountGeneralInfo (String acctID)
    Pre-condition:
       \( \land \) acctID. exists = TRUE
    Post-condition:
If $\land \text{acctID. exists} = \text{FALSE}$
$\land \text{alert Message} \Rightarrow \text{“Not a Valid Account”}$

**Concrete Component**: AccountOperationServer

Synchronization Policy: ProducerConsumer
Synchronization Implementation: Semaphores
Interface: IAccountOperation
Method/Methods Supported:

- **Float getBalance (String acctID)**
  Pre-condition:
  $\land \text{acctID. exists} = \text{TRUE}$
  Post-condition:
  If $\land \text{acctID. exists} = \text{TRUE}$
  $\land \text{Return Account Balance}$
  Else
  alert Message $\Rightarrow \text{“Not a Valid Account”}$

- **void Withdraw (String acctID, Float amount)**
  Pre-condition:
  $\land \text{acctID. exists} = \text{TRUE}$
  $\land \text{acctID. isValid} = \text{TRUE}$
  $\land \text{amount} < \text{‘X’}$
  $\land \text{balance (account)} - \text{amount} > 0$
  Post-condition:
  If $\land \text{acctID. exists} = \text{FALSE}$
  $\land \text{alert Message} \Rightarrow \text{“Account ID Does Not EXIST”}$
  Else If $\land \text{acctID. isValid} = \text{FALSE}$
  $\land \text{alert Message} \Rightarrow \text{“Not a Valid Account ID”}$
  Else If
  $\land \text{balance (account)} = \text{balance(account}@\text{pre} - \text{amount}$
  $\land \text{newBalance (account)} < \text{oldBalance (account)}$
\( \land \) balance(account) < minimumBalance(account) \( \Rightarrow \) alertUser

- void deposit (String acctID, float amount)

  Pre-condition:
  \( \land \) acctID. exists = TRUE
  \( \land \) acctID. isValid = TRUE

  Post-condition:
  If \( \land \) acctID. exists = FALSE
  \( \land \) alert Message \( \Rightarrow \) “Account ID Does Not EXIST”
  Else If \( \land \) acctID. isValid = FALSE
  \( \land \) alert Message \( \Rightarrow \) “Not a Valid Account ID ”
  Else If
  \( \land \) balance (account) = balance(account)@pre + amount
  \( \land \) newBalance (account) > oldBalance (account)
  \( \land \) alert Message \( \Rightarrow \) “Amount Deposited in Account ”

Different systems can be created with variations in the methods supported by the concrete components. In this case study to verify the matches one concrete implementation of AccountOperationServer is considered. Below Figure A.2 is the complete UMM representation of the four contract levels: the syntactic contract, the semantic contract, the synchronization contract and the QoS contract in AccountOperationServer.

1. Component Name: AccountOperationServer
2. Component Subcase: AccountOperationServerCase
3. Domain Name: Bank
4. System Name: P-Bank
5. Description: Represents the main operations performed on the Account
6. Computational Attributes:
   6.1 Inherent Attributes:
      6.1.1 id: AOS
      6.1.2 Version: version1.0
      6.1.3 Author: Prathiba Katuri
6.1.4 Date: May 2006
6.1.5 Validity: N/A
6.1.6 Atomicity: Yes
6.1.7 Registration: N/A
6.1.8 Model: N/A

7. Deployment Attributes: N/A

8. Functional Attributes:
   8.1 Functional Description: Provides operations on the Account
   8.2 Algorithm: N/A
   8.3 Complexity: N/A

**Syntactic & Semantic Contract**

8.4 Interfaces

8.4.1 Provided Interfaces:

8.4.1.1 Interface: IAddAccount

8.4.1.1.1 Method/Methods Supported:

1. AddAccount
   - Method Signature: String AddAccount (String acctID)
   - Description: Adds a New Account into the Database
   - Return Type: String
   - Parameter Type/Types: String
   - Pre-Condition:
     \( \\land \text{acctID} \neq \text{Null} \)
     \( \\land \text{acctID.Exists} \neq \text{TRUE} \)
   - Invariant:
     \( \\land \text{acctID.Exists} \neq \text{TRUE} \)
   - Post-Condition:
     \( \\land \text{acctID.Exists} == \text{TRUE} \)

8.4.1.2 Interface: IAccountOperation

8.4.2.1.1 Method/Methods Supported:

1. getBalance
   - Method Signature: Float getBalance (String acctID)
• Description: Gets the current balance of the Account
• Return Type: Float
• Parameter Type/Types: String
• Pre-Condition:
  \( \land \text{acctID} \neq \text{Null} \)
  \( \land \text{acctID}.\text{Exists} = \text{TRUE} \)
• Invariant:
  \( \land \text{acctID}.\text{Exists} = \text{TRUE} \)
• Post-Condition:
  If \( \land \text{acctID}.\text{Exists} = \text{TRUE} \)
    \( \land \text{Return Account Balance} \)
  Else
    alert Message => “Not a Valid Account”

2. Withdraw
• Method Signature:
  \( \text{void withdraw (String acctID, Float amount)} \)
• Description: Used to withdraw amount from Account
• Return Type: void
• Parameter Type/Types: String, Float
• Pre-Condition:
  \( \land \text{acctID} \neq \text{Null} \)
  \( \land \text{acctID}.\text{Exists} = \text{TRUE} \)
  \( \land \text{amount} < \text{‘X’} \)
  \( \land \text{balance (account)} – \text{amount} > 0 \)
• Invariant:
  \( \land \text{acctID}.\text{Exists} = \text{TRUE} \)
  \( \land \text{amount} > 0 \)
• Post-Condition:
  If \( \land \text{acctID}.\text{exists} = \text{FALSE} \)
    \( \land \text{alert Message} => \text{“Account ID Does Not EXIST”} \)
  Else If
\( \text{acctID. isValid = FALSE} \)
\( \text{alert Message => “Not a Valid Account ID” } \)

Else If
\( \text{Balance(account)=Balance(account)@pre – amount} \)
\( \text{NewBalance (account) < OldBalance (account)} \)
\( \text{balance(account) < minimumBalance(account) => alertUser} \)

3. Deposit
- Method Signature:
  void deposit (String acctID, Float amount)
- Description: Used to Deposit amount into Account
- Return Type: void
- Parameter Type/Types: String, Float
- Pre-Condition:
  \( \text{acctID != Null} \)
  \( \text{acctID.Exists = TRUE} \)
  \( \text{amount > 0} \)
- Invariant:
  \( \text{acctID.Exists = TRUE} \)
  \( \text{amount > 0} \)
- Post-Condition:
  If \( \text{acctID. exists = FALSE} \)
    \( \text{alert Message => “Account ID Does Not EXIST” } \)
  Else If
    \( \text{acctID. isValid = FALSE} \)
    \( \text{alert Message => “Not a Valid Account ID ” } \)
  Else If
    \( \text{Balance (account) = Balance(account)@pre + amount} \)
    \( \text{NewBalance (account) > OldBalance (account)} \)
    \( \text{alert Message => “Amount Deposited in Account” } \)

8.4.2 Required Interfaces: None

8.5 Technology: N/A
8.6 Expected Resources: N/A
8.7 Design Patterns: NONE
8.8 Known Usage: NONE
8.9 Alias: NONE

9. Cooperation Attributes:
   9.1 Preprocessing Collaborators: User Terminal, Coordination Server
   9.2 Postprocessing Collaborators: NONE

10. Auxiliary Attributes:
   10.1 Mobility: No
   10.2 Security: L0
   10.3 Fault Tolerance: L0

**Synchronization Contract**

11. Synchronization Policy
   11.1 Policy Name: Producer Consumer
   11.2 Implementation Technique: Semaphores

**QoS Contract**

12. Quality of Service
   12.1 QoS Metrics
      12.1.1 Metric
         • Parameter Name: Throughput
         • Parameter Value: 83
      12.1.2 Metric
         • Parameter Name: Turn Around Time
         • Parameter Value: 990
   12.2 QoS Level: N/A
   12.3 Cost: N/A
   12.4 Quality Level: N/A
   12.5 Effect of Environment: N/A
   12.6 Effect of Usage Pattern: N/A

Figure A.2: A UMM Specification for Account Operation Server
A.3 Validation of Matching Criteria

This section presents the validation of matching criteria considered for matching at all the four levels of component specification. The syntactic level matches are considered first. The query requirement is given for each match and theoretical validation of matches is provided.

Only a part of the UMM specification of Account Operation Server representing the syntactic and semantic contract is shown here.

I. Syntactic Level Matches

The representation of Generic Type match at the Syntactic Level

\[
\text{Match}_\text{Gen-Type} (C, Q)
\]

\[
= \lfloor \text{Match}_\text{Ordered Gen-Type} (C, Q) \mid \text{Match}_\text{Reordered Gen-Type} (C, Q) \rfloor
\]

\[
= \lfloor \text{For each Function } C_{Fi} \text{ and } Q_{Fi} \\
\{ ( \lfloor C_{Fi} \rfloor_{RT, AT} \text{ } R_1 \lfloor Q_{Fi} \rfloor_{RT, AT} ) \mid ( \lfloor C_{Fi} \rfloor_{RT, AT*} \text{ } R_1 \lfloor Q_{Fi} \rfloor_{RT, AT} ) \} \rfloor
\]

Below are the definitions of the different instantiations of Generic Type match and as well as the explanation of the matches with theoretical proof.

1. Exact Type Match

\[
\text{Match}_\text{E-Type} (C, Q)
\]

\[
= \lfloor \text{Match}_\text{Ordered E-Type} (C, Q) \mid \text{Match}_\text{Reordered E-Type} (C, Q) \rfloor
\]

\[
= \lfloor \text{For each Function } C_{Fi} \text{ and } Q_{Fi} \\
\{ ( \lfloor C_{Fi} \rfloor_{RT, AT} = E \lfloor Q_{Fi} \rfloor_{RT, AT} ) \mid ( \lfloor C_{Fi} \rfloor_{RT, AT*} = E \lfloor Q_{Fi} \rfloor_{RT, AT} ) \} \rfloor
\]
Below is the explanation of this match with a query requirement by the user and the corresponding function returned under this match.

Consider a query requirement for Bank Domain. User needs a function to withdraw an amount less than a cutoff limit ‘X1’ from an account in Bank domain and also imposes that the account will stay positive after operation such that when the method executes the new balance of the account is less than the previous balance and also alerts user if the balance is less than minimum balance. The signature of the withdraw function is given and the types of pre/post variables is given.

The requirement of the query component Q is given below:

- **Signature**: void withdraw (String acctID, Float amount)
- **Function Name**: Withdraw
- **Return Type**: void
- **Argument Type/Types**: String, Float
- **Pre-Condition**:
  \[ \land \text{acctID} \neq \text{Null} \]
  \[ \land \text{amount} < \text{‘X1’} \]
  \[ \land \text{Balance (account)} - \text{amount} > 0 \]
- **Post-Condition**:
  \[ \land \text{Balance (acctID)} = \text{Balance (acctID)}@\text{pre} - \text{amount} \]
  \[ \land \text{Balance (account)} < \text{minimumBalance(account)} \Rightarrow \text{alertUser} \]

Below is the explanation of how the candidate component: “AccountOperationServer” matches with query requirement. Let the candidate component be represented as “C” for the matches under the syntactic category. The Exact Type match first checks if the candidate component satisfies the given query under the ordered match, if not reordered match is applied. Applying the information to the Exact Type match,

Function Return Type and Argument Types for Query Component Q: \[ Q_{RT,AT} \]
Function Return Type and Argument Types for Candidate Component C: \[ C_{RT,AT} \]
Below is the representation of return type, argument types of $Q_{Fi}$, $C_{Fi}$.
The comparison of types starts with return type of the required functions:

\[
\begin{align*}
[Q_{Fi}]_{RT} &= \text{void} & [Q_{Fi}]_{AT} &= \text{String, Float} \\
[C_{Fi}]_{RT} &= \text{void} & [C_{Fi}]_{AT} &= \text{String, Float}
\end{align*}
\]

Applying the ordered match, \[\text{Match \textit{Ordered E-Type}}(C, Q)\]

\[
\text{Match \textit{Ordered E-Type}}(C, Q) = \begin{cases}
\text{For each Function } C_{Fi} \text{ and } Q_{Fi} \\
( [C_{Fi}]_{RT, AT} = E [Q_{Fi}]_{RT, AT} )
\end{cases}
\]

- Comparing the types and applying the match: ( \( [C_{Fi}]_{RT, AT} = E [Q_{Fi}]_{RT, AT} \) )
  1. Comparing the return types of the function

\[
( [C_{Fi}]_{RT} = E [Q_{Fi}]_{RT} )
\]

\[
= (\text{“void”} = E \text{“void”})
\]

The exact equality relation is satisfied and return types are matched.

2. Comparing the argument types of the function

\[
( [C_{Fi}]_{AT} = E [Q_{Fi}]_{AT} )
\]

\[
= (\text{“String”} = E \text{“String”}) && (\text{“Float”} = E \text{“Float”})
\]

The exact equality relation is satisfied and the argument types are matched.

The return types and the argument types of the components match under Exact Equality
\((= E)\), hence

\[
( [C_{Fi}]_{RT, AT} = E [Q_{Fi}]_{RT, AT} ) = \text{TRUE}
\]

So, the ordered Exact match, \(\text{Match \textit{Ordered E-Type}} = \text{TRUE}\)

Since the candidate component function matches the query component function under the same order of arguments, the reordered argument match is not considered.

Hence,

\(\text{Match \textit{E-Type}}(C, Q) = \text{TRUE}\)
The candidate component specification matches exactly with the query component Q specification under ordered Exact Type match. Below is the explanation of the Exact Type match with reordered argument types with a different query requirement.

Consider a query requirement for Bank Domain. User needs a function to withdraw an amount less than a cutoff limit ‘X1’ from an account in Bank domain and also imposes that the account will stay positive after operation such that when the method executes the new balance of the account is less than the previous balance and also alerts user if the balance is less than minimum balance. The signature of the withdraw function is given and the types of pre/post variables is given.

The requirement of the query component Q is given below:

- Signature: void withdraw (Float amount, String acctID)
- Function Name: Withdraw
- Return Type: void
- Argument Type/Types: String, Float
- Pre-Condition:
  \[ \land \text{acctID} \neq \text{Null} \]
  \[ \land \text{amount} < \text{'}X1\text{' } \]
  \[ \land \text{Balance (account)} – \text{amount} > 0 \]
- Post-Condition:
  \[ \land \text{Balance (acctID)} = \text{Balance (acctID)@pre} – \text{amount} \]
  \[ \land \text{Balance (account)} < \text{minimumBalance (account)} \Rightarrow \text{alertUser} \]

Below is the explanation of how the bank component: “AccountOperationServer” matches with query requirement. Applying the information to the Exact Type match,

\[
\text{Match } E\text{-Type } ( C, Q ) \\
= [ ( \text{Match } Ordered E\text{-Type } ( C, Q ) ) \mid ( \text{Match } Reordered E\text{-Type } ( C, Q ) ) ] \\
= [ \text{For each Function } C_{fi} \text{ and } Q_{fi} ]
\]
\[
\{ ( [ C_{Fi} ]_{RT, AT} = E [ Q_{Fi} ]_{RT, AT} ) \mid ( [ C_{Fi} ]_{RT, AT} = E [ Q_{Fi} ]_{RT, AT} ) \}
\]

The Exact Type match first checks if the candidate component satisfies the given query under the ordered match, if not reordered match is applied.

Function Return Type and Argument Types for Component Q: \([ Q_{Fi} ]_{RT, AT}\)
Function Return Type and Argument Types for Component C: \([ C_{Fi} ]_{RT, AT}\)

Below is the representation of return type, argument types and variable types of \(Q_{Fi}\), \(C_{Fi}\).

\[
[ Q_{Fi} ]_{RT} = \text{void} \quad [ Q_{Fi} ]_{AT} = \text{Float, int}
\]
\[
[ C_{Fi} ]_{RT} = \text{void} \quad [ C_{Fi} ]_{AT} = \text{int, Float}
\]

Applying the ordered match, \([ \textbf{Match \ Ordered \ E-Type} \ (C, Q) ]\)

\textbf{Match \ Ordered \ E-Type} \ (C, Q) = \{ \text{For each Function } C_{Fi} \text{ and } Q_{Fi} \}

\[
( [ C_{Fi} ]_{RT, AT} = E [ Q_{Fi} ]_{RT, AT} )
\]

- Comparing the types and applying the match: \(( [ C_{Fi} ]_{RT, AT} = E [ Q_{Fi} ]_{RT, AT} )\)
  1. Comparing the return types of the function

\[
( [ C_{Fi} ]_{RT} = E [ Q_{Fi} ]_{RT} )
\]

\[
= (\text{“void”} = E \text{“void”})
\]

The equality relation is satisfied and return types are matched.

2. Comparing the argument types of the function

\[
( [ C_{Fi} ]_{AT} = E [ Q_{Fi} ]_{AT} )
\]

\[
= (\text{“String”} = E \text{“Float”}) \&\& (\text{“Float”} = E \text{“String”})
\]

The equality relation is not satisfied as the order of argument types is not matched. The ordered argument match is discarded and the next match, reordered exact match is applied.
Applying the reordered match, \([ \text{Match} \text{ Reordered E-Type} ]\)

**Match Reordered E-Type** = [ \(\text{For each Function } C_{Fi} \text{ and } Q_{Fi} \)

\[\{ ( [ C_{Fi} ]_{RT,AT^*} = E [ Q_{Fi} ]_{RT,AT} ) \} \]

- Comparing the types and applying the match: \( ( [ C_{Fi} ]_{RT,AT} = E [ Q_{Fi} ]_{RT,AT} ) \)
  
  1. Comparing the return types of the function
     \[ ( [ C_{Fi} ]_{RT} = E [ Q_{Fi} ]_{RT} ) \]
     
     \(= ( \text{“void”} = E \text{“void”} )\)
     
     The equality relation is satisfied and return types are matched.

  2. Comparing the argument types of the function
     \[ ( [ C_{Fi} ]_{AT^*} = E [ Q_{Fi} ]_{AT} ) \]
     
     The argument type match is checked till the types of the candidate component are matched with the query component function.
     
     \(= ( \text{“String”} = E \text{“String”} ) \&\& ( \text{“Float”} = E \text{“Float”} )\)
     
     The exact equality relation is satisfied and the argument types are matches.

So, components match under return type and arguments types

\( ( [ C_{Fi} ]_{RT,AT^*} = E [ Q_{Fi} ]_{RT,AT} ) = \text{TRUE} \)

So, the reordered Exact Type match,

**Match Reordered E-Type** = \(\text{TRUE}\)

The candidate component function matches the query component function under reordered argument match.

Hence,

**Match E-Type** \((C, Q) = \text{TRUE}\)

The candidate component specification matches exactly with the query component Q specification under Reordered Exact Type match.
2. **Relaxed Type Match**

\[
\text{Match}_{\text{R-Type}} (C, Q) = \left\{ \begin{array}{l}
\text{Match}\ \text{Ordered}\ R\text{-Type} (C, Q) \ \text{||} \ \text{Match}\ \text{Reordered}\ R\text{-Type} (C, Q) \\
\text{For each Function } C_{Fi} \text{ and } Q_{Fi} \\\n\{ (\ [C_{Fi}]_{RT, AT} = R \ [Q_{Fi}]_{RT, AT} ) \ \text{||} \ (\ [C_{Fi}]_{RT, AT^*} = R \ [Q_{Fi}]_{RT, AT} ) \}
\end{array} \right.
\]

Consider a query requirement for Bank Domain. User needs a function to withdraw an amount less than a cutoff limit 'X1' from an account in Bank domain and also imposes that the account will stay positive after operation such that when the method executes the new balance of the account is less than the previous balance and also alerts user if the balance is less than minimum balance. The signature of the withdraw function is given and the types of pre/post variables is given.

The requirement of the query component Q is given below:

- **Method Signature**: void withdraw (String acctID, Double amount)
- **Function Name**: withdraw
- **Return Type**: void
- **Argument Type/Types**: String, Double
- **Pre-Condition**:
  \[ \wedge \text{acctID} > 0 \]
  \[ \wedge \text{Balance (acctID)} > 0.0 \]
- **Post-Condition**:
  \[ \wedge \text{Balance (acctID)} < \text{minimumBalance (acctID)} \Rightarrow \text{alertUser} \]
Below is the explanation of how the bank component: “AccountOperationServer” matches with query requirement. Applying the information to the Relaxed Type match,

\[ \text{Match}_{R\text{-Type}} \left( C, Q \right) \]

\[ = \left[ \text{Match}_{\text{Ordered } R\text{-Type}} \left( C, Q \right) \mid \text{Match}_{\text{Reordered } R\text{-Type}} \left( C, Q \right) \right] \]

\[ = \left[ \text{For each Function } C_{Fi} \text{ and } Q_{Fi} \right. \]

\[ \left\{ \left( \left[ C_{Fi} \right]_{RT, AT} = R \left[ Q_{Fi} \right]_{RT, AT} \right) \mid \left( \left[ C_{Fi} \right]_{RT, AT'} = R \left[ Q_{Fi} \right]_{RT, AT} \right) \right\} \]

The Relaxed Type match first checks if the candidate component satisfies the given query under the ordered match, if not reordered match is applied.

Function Return Type and Argument Types for Component Q: \[ \left[ Q_{Fi} \right]_{RT, AT} \]
Function Return Type and Argument Types for Component C: \[ \left[ C_{Fi} \right]_{RT, AT} \]

Below is the representation of return type, argument types and variable types of \( Q_{Fi}, \) \( C_{Fi}. \) The comparison of types starts with return type of the required functions:

\[ \left[ Q_{Fi} \right]_{RT} = \text{void} \quad \left[ Q_{Fi} \right]_{AT} = \text{int, Double} \]
\[ \left[ C_{Fi} \right]_{RT} = \text{void} \quad \left[ C_{Fi} \right]_{AT} = \text{int, Float} \]

Applying the ordered Relaxed match, \[ \text{Match}_{\text{Ordered } R\text{-Type}} \left( C, Q \right) \]

\[ \text{Match}_{\text{Ordered } R\text{-Type}} \left( C, Q \right) = \left[ \text{For each Function } C_{Fi} \text{ and } Q_{Fi} \right. \]

\[ \left. \left( \left[ C_{Fi} \right]_{RT, AT} = R \left[ Q_{Fi} \right]_{RT, AT} \right) \right] \]

- Comparing the types and applying the match: \( \left( \left[ C_{Fi} \right]_{RT, AT'} = R \left[ Q_{Fi} \right]_{RT, AT} \right) \)

1. Comparing the return types of the function

\[ \left( \left[ C_{Fi} \right]_{RT} = R \left[ Q_{Fi} \right]_{RT} \right) \]

\[ = \left( \text{“void”} = R \text{“void”} \right) \]

\[ = \left[ \left( \text{“void”} = \text{Inheritance “void”} \right) \mid \left( \text{“void”} = \text{Coercion “void”} \right) \right] \]
The inheritance relation and the coercion matches are satisfied.

2. Comparing the argument types \( \text{String, Float} \) and \( \text{String, Double} \) of the functions

\[
\left[ C_{Fi} \right]_{AT^*} = \left[ Q_{Fi} \right]_{AT}
\]

\[
= (\text{"String"} = \text{R"String"})
\]

\[
= \left[ \text{("String" = Inheritance "String") \mid ("String" = Coercion "String")} \right]
\]

The inheritance relation and the coercion matches are satisfied.

\[
\left[ C_{Fi} \right]_{AT^*} = \left[ Q_{Fi} \right]_{AT}
\]

\[
= (\text{"Float" = Inheritance "Double"}) \mid (\text{"Float" = Coercion "Double"})
\]

The Inheritance match is satisfied.

So, components match under return type and arguments types

\[
\left[ C_{Fi} \right]_{RT,AT} = \left[ Q_{Fi} \right]_{RT,AT} = \text{TRUE}
\]

So,

\textbf{Match Ordered R-Type} = \text{TRUE}

Since the candidate component function matches the query component function under the same order of arguments, the reordered argument match is not considered. Hence,

\textbf{Match R-Type} \( (C, Q) \) = \text{TRUE}

The candidate component \( C \) (AccountOperationServer) specification matches exactly with the query component \( Q \) specification under Relaxed Type match.

If the above query requirement function is modified in the order of argument types then the reordered match of arguments \textbf{Match Reordered R-Type} is satisfied. Both ordered and reordered matches implement the match with variation in the order of type arguments.
II. Semantic Level Matching

Generic Pre/Post Match

Generic Pre/Post match relates the pre-conditions of each component and the post-conditions of each component. The representation of generic Pre/Post match is given below

\[
\text{Match}_{\text{Gen-Pre/Post}}(C, Q) = \left[ \text{For each Function } C_{Fi} \text{ and } Q_{Fi} \left( [Q_{Fi}|_{Pre} \iff [C_{Fi}|_{Pre}] \land ( [C_{Fi}|_{SP-Post} \iff [Q_{Fi}|_{Post}]) \right) \right]
\]

Below are the different instantiations of Generic pre/post match and as well as validation of the matches.

1. Exact Pre/Post Match

\[
\text{Match}_{\text{E-Pre/Post}}(C, Q) = \left[ \text{For each Function } C_{Fi} \text{ and } Q_{Fi} \left( [Q_{Fi}|_{Pre} \iff [C_{Fi}|_{Pre}] \land ( [C_{Fi}|_{Post} \iff [Q_{Fi}|_{Post}]) \right) \right]
\]

Consider a query requirement for Bank Domain. User needs a function to withdraw an amount less than a cutoff limit ‘X’ from an account in Bank domain and also imposes that the account will stay positive after operation such that when the method executes the new balance of the account is less than the previous balance and also alerts user if the balance is less than minimum balance. The requirement of the query component Q representing only the semantic information is given below:

- Method Signature: void withdraw (String acctID, Float amount)
- Function Name: withdraw
- Return Type: void
- Argument Type/Types: String, Double
- Pre-Condition:
\( \wedge \text{acctID} \neq \text{NULL} \)
\( \wedge \text{acctID.Exists} = \text{TRUE} \)
\( \wedge \text{amount} < \ 'X' \)
\( \wedge \text{Balance (account)} - \text{amount} > 0 \)

- **Post-Condition:**
  
  \( \wedge \text{Balance (acctID)} = \text{Balance (acctID)}@pre - \text{amount} \)
  
  \( \wedge \text{newBalance (account)} < \text{oldBalance (account)} \)
  
  \( \wedge \text{Balance (account)} < \text{minimumBalance(account)} \Rightarrow \text{alertUser} \)

The bank component: “AccountOperationServer” is the candidate component used to match the query requirement and the semantic specification below shows only the method that is being matched. Let the candidate component be represented as “C” in the explanation below. The whole UMM specification of the component: “AccountOperationServer” is given in Figure A.2.

- **Method Signature:** void withdraw (String acctID, Float amount)
- **Description:** Used to withdraw amount from Account
- **Return Type:** void
- **Parameter Type/Types:** String, Float
- **Pre-Condition:**
  
  \( \wedge \text{acctID} \neq \text{Null} \)
  
  \( \wedge \text{acctID.Exists} = \text{TRUE} \)
  
  \( \wedge \text{amount} < \ 'X' \)
  
  \( \wedge \text{Balance (account)} - \text{amount} > 0 \)

- **Invariant:**
  
  \( \wedge \text{acctID.Exists} = \text{TRUE} \)
  
  \( \wedge \text{amount} > 0 \)

- **Post-Condition:**
  
  If \( \wedge \text{acctID. exists} = \text{FALSE} \)
  
  \( \wedge \text{alert Message} \Rightarrow \text{“Account ID Does Not EXIST”} \)
  
  Else If
  
  \( \wedge \text{acctID. isValid} = \text{FALSE} \)
\[ \text{alert Message } \Rightarrow \text{“Not a Valid Account ID”} \]

Else If
\[ \land \text{Balance(account)=Balance(account)@pre – amount} \]
\[ \land \text{newBalance (account) < oldBalance (account)} \]
\[ \land \text{Balance(account) < minimumBalance(account) } \Rightarrow \text{alertUser} \]

Below is the explanation of how the candidate component matches with query requirement. Applying the information to the Exact Pre/Post match,

\[
\text{Match}_{\text{E-Pre/Post}} (\mathcal{C}, \mathcal{Q}) = \left[ (\mathcal{Q}_{\text{Pre}} \Leftrightarrow \mathcal{C}_{\text{Pre}}) \land (\mathcal{C}_{\text{Post}} \Leftrightarrow \mathcal{Q}_{\text{Post}}) \right]
\]

Pre/Post Conditions for Query Component \( \mathcal{Q} \): \( [\mathcal{Q}]_{\text{Pre}}, [\mathcal{Q}]_{\text{Post}} \)
Pre/Post Conditions for Candidate Component \( \mathcal{C} \): \( [\mathcal{C}]_{\text{Pre}}, [\mathcal{C}]_{\text{Post}} \)

- Comparing the pre-conditions:
  \[ [\mathcal{Q}]_{\text{Pre}} = \land \text{acctID } \neq \text{ NULL} \]
  \[ \land \text{acctID.Exists } = \text{ TRUE} \]
  \[ \land \text{amount } < \text{’ X ’} \]
  \[ \land \text{Balance (account)} – \text{ amount } > 0 \]
  \[ [\mathcal{C}]_{\text{Pre}} = \land \text{acctID } \neq \text{ NULL} \]
  \[ \land \text{acctID.Exists } = \text{ TRUE} \]
  \[ \land \text{amount } < \text{’ X ’} \]
  \[ \land \text{Balance (account)} – \text{ amount } > 0 \]

so,

\[ ( [\mathcal{Q}]_{\text{Pre}} \Leftrightarrow [\mathcal{C}]_{\text{Pre}} ) = \text{TRUE} \]

- Comparing the post-conditions:
  \[ [\mathcal{C}]_{\text{Post}} = \land \text{Balance (acctID) = Balance (acctID)@pre – amount} \]
  \[ \land \text{newBalance (account) < oldBalance (account)} \]
  \[ \land \text{Balance (account) < minimumBalance(account) } \Rightarrow \text{alertUser} \]
\[ [Q]_{post} = \land \text{Balance (acctID)} = \text{Balance (acctID)}@\text{pre} - \text{amount} \\
\land \text{newBalance (account)} < \text{oldBalance (account)} \\
\land \text{Balance (account)} < \text{minimumBalance(account)} \Rightarrow \text{alertUser} \]

so,

\[
([C]_{post} \leftrightarrow [Q]_{post}) = \text{TRUE}
\]

So, the Exact Pre/Post match

Match \( E-\text{Pre/Post}(C, Q) \) = \text{TRUE}

The candidate component \( C \) (AccountOperationServer) specification matches exactly with the query component \( Q \) specification under Exact Pre/Post match.

2. \textbf{Relaxed Pre/Post Match}

In this match, the pre/post conditions are relaxed compared to exact pre/post where equivalence is used. Equivalence is a strict relation, so in this match the relation between the pre/post conditions is given an implication relation.

\[
\text{Match} \ R-\text{Pre/Post}(C, Q) \ = \ [\text{For each Function } C_{Fi} \text{ and } Q_{Fi}] \\
([Q_{Fi}]_{Pre} \Rightarrow [C_{Fi}]_{Pre}) \land ([C_{Fi}]_{Post} \Rightarrow [Q_{Fi}]_{Post})
\]

Consider a query requirement for Bank Domain. User needs a function to withdraw an amount less than a cutoff limit ‘\( X_1 \)’ from an account in Bank domain and also imposes that the account will stay positive after operation such that when the method executes the new balance of the account is less than the previous balance. The requirement of the query component \( Q \) is given below:

- Function Signature: void withdraw (String acctID, Float amount)
- Function Name: withdraw
• Return Type: void
• Argument Type/Types: String, Double
• Pre-Condition:
  \[ \land \text{acctID} \neq \text{NULL} \]
  \[ \land \text{acctID.Exists} = \text{TRUE} \]
  \[ \land \text{amount} < 'X' \]
  \[ \land \text{Balance (account)} - \text{amount} > 0 \]
• Post-Condition:
  \[ \text{Balance (acctID)} = \text{Balance (acctID)@pre} - \text{amount} \]

Below is the part of semantic specification of the candidate component: “AccountOperationServer” representing only the method required.
• Method Signature: void withdraw (String acctID, Float amount)
• Description: Used to withdraw amount from Account
• Return Type: void
• Parameter Type/Types: String, Float
• Pre-Condition:
  \[ \land \text{acctID} \neq \text{Null} \]
  \[ \land \text{acctID.Exists} = \text{TRUE} \]
  \[ \land \text{amount} < 'X' \]
  \[ \land \text{Balance (account)} - \text{amount} > 0 \]
• Invariant:
  \[ \land \text{acctID.Exists} = \text{TRUE} \]
  \[ \land \text{amount} > 0 \]
• Post-Condition:
  If \[ \land \text{acctID. exists} = \text{FALSE} \]
    \[ \land \text{alert Message} => \text{“Account ID Does Not EXIST”} \]
  Else If
  \[ \land \text{acctID. isValid} = \text{FALSE} \]
  \[ \land \text{alert Message} => \text{“Not a Valid Account ID”} \]
  Else If
Below is the explanation of how the candidate component matches with query requirement. Let the AccountOperationServer be represented as ‘C’. Applying the information to the Relaxed Pre/Post match,

\[
\text{Match} \; \text{R-Pre/Post} \; (C, Q) = \left( (Q|_{\text{pre}} \Rightarrow C|_{\text{pre}}) \land (C|_{\text{post}} \Rightarrow Q|_{\text{post}}) \right)
\]

Pre/Post Conditions for Query Component \( Q \): \( [Q]_{\text{pre}}, [Q]_{\text{post}} \)

Pre/Post Conditions for Candidate Component \( C \): \( [C]_{\text{pre}}, [C]_{\text{post}} \)

- Comparing the pre-conditions:
  
  \[
  [Q]_{\text{pre}} = \land \text{acctID} \neq \text{NULL} \\
  \land \text{acctID.Exists = TRUE} \\
  \land \text{amount < ‘X’} \\
  \land \text{Balance (account) – amount > 0}
  \]
  
  \[
  [C]_{\text{pre}} = \land \text{acctID} \neq \text{NULL} \\
  \land \text{acctID.Exists = TRUE} \\
  \land \text{amount < ‘X_1’} \\
  \land \text{Balance (account) – amount > 0}
  \]

  \[
  ([Q]_{\text{pre}} \Rightarrow [C]_{\text{pre}}) = (\text{amount < ‘X’}) \Rightarrow (\text{amount < ‘X_1’})
  \]
  
  \[
  = \text{TRUE}
  \]

  The maximum amount required ‘X’ is less than the maximum amount provided ‘X_1’.

- Comparing the post-conditions:

  \[
  [C]_{\text{post}} = \land \text{Balance (acctID) = Balance (acctID)@pre – amount} \\
  \land \text{Balance (account) < minimumBalance(account) => alertUser}
  \]
  
  \[
  [Q]_{\text{post}} = \text{Balance (acctID) = Balance (acctID)@pre – amount}
  \]
\[
( [C \mid \text{Post} \Rightarrow [Q \mid \text{Post}]) = \\
[ \quad \text{Balance (acctID)} = \text{Balance (acctID)}@\text{pre} - \text{amount} \\
\quad \& \text{Balance (account)} < \text{minimumBalance(account)} \Rightarrow \text{alertUser} \quad ] \\
\Rightarrow \quad \text{Balance (acctID)} = \text{Balance (acctID)}@\text{pre} - \text{amount} = \text{TRUE}
\]

So, the Relaxed Pre/Post match

\text{Match} \ R\text{-Pre/Post} (C, Q) = \text{TRUE}

The candidate component \( C \) (AccountOperationServer) specification matches with the query component \( Q \) specification under Relaxed Pre/Post match.

The pre- and the post-conditions of both the candidate and the query components are matched using Spec\# programming language.

3. \textit{Relaxed Post Match}

In this match, only the post-conditions of the function are explicitly indicated. The pre-conditions are assumed to be satisfied using an additional check in the code.

\textbf{Match} \ R\text{-Post} (C, Q) = [ \text{For each Function } C_{Fi} \text{ and } Q_{Fi} \\
( [C_{Fi} \mid \text{Post} \Rightarrow [Q_{Fi} \mid \text{Post}])]

Consider a query requirement for Bank Domain. User needs a function to return the balance of the account. No pre-condition is specified in this query. The requirement of the query component \( Q \) is given below:

- Function Signature: Float getBalance (String acctID)
- Function Name: withdraw
- Return Type: void
- Argument Type/Types: String, Double
- Pre-Condition: Not required
- Post-Condition: return Balance(acctID)
Below is the semantic specification of the candidate component: “AccountOperationServer” represented as “C” indicating only the method required:

Method Signature: Float getBalance (String acctID)
Description: Gets the current balance of the Account
Return Type: Float
Parameter Type/Types: String
Pre-Condition:
\[ \land \text{acctID} \neq \text{Null} \]
\[ \land \text{acctID.Exists} = \text{TRUE} \]
Invariant:
\[ \land \text{acctID.Exists} = \text{TRUE} \]
Post-Condition:
If \[ \land \text{acctID.Exists} = \text{TRUE} \]
\[ \land \text{return Balance (acctID)} \]
Else
alert Message => “Not a Valid Account”

Below is the explanation of how the candidate component matches with query requirement under the relaxed pre/post match. Applying the information to the Relaxed Pre/Post match,

\[ \text{Match}_{RPost} (C, Q) = ( \{ C \}_Post \Rightarrow \{ Q \}_Post) \]

Post Conditions for Query Component Q: \[ \{ Q \}_Post \]
Post Conditions for Candidate Component C: \[ \{ C \}_Post \]

- Comparing the post-conditions:
\[ \{ C \}_Post = \text{If} \ \land \text{acctID.Exists} = \text{TRUE} \]
\[ \land \text{return Balance (acctID)} \]
Else
alert Message => “Not a Valid Account”
\[ [Q]_{post} = \text{return Balance(acctID)} \]

\(([C]_{post} \Rightarrow [Q]_{post}) = \)

If \(\land\) acctID. Exists = TRUE
\(\land\) return Balance (acctID)
\(\Rightarrow\) return Balance(acctID) = TRUE

So, the Relaxed Post match

\(\text{Match}_{R-Post} (C, Q) = \text{TRUE} \)

The candidate component C specification matches with the query component Q specification under Relaxed Post match.

These are the matches considered at the semantic level limiting the matches only to function specifications. Each of these matches is verified using Spec# programming system the details of which are explained in chapter 5 under the URDS prototype.

*** III. Synchronization Level Matching ***

Two different categories of generic matches are specified at the Synchronization Level: the Generic SP match and the Generic SP-Impl match. For each kind of the match both the exact and relaxed matches are defined. Instantiations of Generic SP match is given below, followed by Generic SP-Impl match.

**Generic SP Match**

Generic SP match is the representation of Synchronization policy implemented by the function/functions supported by the component. The representation of Generic SP match is given below

\(\text{Match}_{Gen-SP} (C, Q) = [C_{SP} \ R_1 \ Q_{SP}] \)

Below is the validation of different matches considered under the Generic SP matches.
1. **Exact SP Match**

\[
\text{Match}_{E-SP} (C, Q) = [C_{SP} \leftrightarrow Q_{SP}]
\]

Consider the query component requirement for a bank system, the query search needs a method that returns the balance of the account. The synchronization policy that it should execute is ProducerConsumer. Assuming that the syntactic and the semantic requirements are matched the requirement of the query is then matched to getBalance method of AccountOperationServer under the exact SP match.

Synchronization policy of Query Component Q represented as: \( Q_{SP} \)

Synchronization policy of Candidate Component C represented as: \( C_{SP} \)

Below is the explanation at how the component AccountOperationServer specification represented as “C” matches with the query requirement.

Applying the information to the exact SP match,

\[
\text{Match}_{E-SP} (C, Q) = [C_{SP} \leftrightarrow Q_{SP}]
\]

\[
[ C ]_{SP} \leftrightarrow [ Q ]_{SP} : \{ \text{ProducerConsumer} \leftrightarrow \text{ProducerConsumer} \} = \text{TRUE}
\]

TLA specification has been written for the ProducerConsumer synchronization policy to specify the behavior of the policy.

So, the exact SP match

\[
\text{Match}_{E-SP} (C, Q) = \text{TRUE}.
\]

The candidate component specification matches exactly with the query component Q specification under exact SP match.

2. **Relaxed SP Match**

\[
\text{Match}_{R-SP} (C, Q) = [C_{SP} \Rightarrow Q_{SP}]
\]

Consider the query component requirement for a bank system, the query search needs a method that returns the account balance. The synchronization policy that the
function should execute is MutualExclusion. Assuming that the requirement is matched at
the syntactic and the semantic levels, the synchronization polices of the query and
candidate component are matched under Relaxed SP match. Below is the explanation at
how the candidate component: “AccountOperationServer” specification is matched with
the query requirement.

Synchronization policy of Query Component Q represented as: $Q_{SP}$
Synchronization policy of Candidate Component C represented as: $C_{SP}$

Applying the information to the relaxed SP match,

$$\text{Match}_{R-SP}(C, Q) = [C_{SP} \Rightarrow Q_{SP}]$$

$$[C_{SP} \Rightarrow [Q_{SP}]: \text{ProdCons} \Rightarrow \text{MutualExclusion}] = \text{TRUE}$$

TLA specification has been written for ProducerConsumer and MutualExclusion
synchronization policy showing the relation of Implication between the policies.
So, the relaxed SP match
Match $R-SP(C, Q)$ = TRUE.
The candidate component specification matches exactly with the query component Q
specification under relaxed SP match.

**Generic SP-Impl Match**

The representation of Generic SP-Impl match is given below

$$\text{Match}_{Gen-SP-Impl}(C, Q) = [C_{SP} \text{ R}_1 Q_{SP}]$$

$$\text{R}_2 [C_{SP-Impl} \text{ R}_3 Q_{SP-Impl}]$$

The exact and relaxed matches proposed under Generic SP-Impl match are
validated below. The same specification is used for both the generic SP-match and
generic SP-Impl match as the idea is to explain the matches proposed with different
relations in both the matches.
1. **Exact SP-Exact Impl Match** *(Exact Synch. Policy, Exact Impl)*

\[
\text{Match}_{E-}[SP, \text{Impl}] (C, Q) = [C_{SP} \Leftrightarrow Q_{SP}] \\
\quad \wedge [C_{SP-\text{Impl}} = Q_{SP-\text{Impl}}]
\]

Consider the query component requirement for a bank system, the query search needs a method that returns the account balance. The synchronization policy that the method should execute is ProducerConsumer. The implementation required is Semaphores. The requirement of the query then matches that of a getBalance method under the Exact SP-Impl match.

Below is the explanation at how the component AccountOperationServer specification matches with the query requirement.

Applying the information to the exact SP-Impl match,

\[
\text{Match}_{E-}[SP, \text{Impl}] (C, Q) = [C_{SP} \Leftrightarrow Q_{SP}] \\
\quad \wedge [C_{SP-\text{Impl}} = Q_{SP-\text{Impl}}]
\]

The exact SP-exact Impl match above is a combination of two parts, the synchronization policy, the implementation of the function.

- Synchronization policy of Query Component Q represented as: \(Q_{SP}\)
- Implementation technique of Query Component Q represented as: \(Q_{SP-\text{Impl}}\)
- Synchronization policy of Candidate Component C represented as: \(C_{SP}\)
- Implementation technique of Query Component C represented as: \(C_{SP-\text{Impl}}\)

1. \(\mid C \mid_{SP} \Leftrightarrow \mid Q \mid_{SP} : \{\text{ProducerConsumer} \leftrightarrow \text{ProducerConsumer}\} = \text{TRUE}\)

2. \(\mid C \mid_{SP-\text{Impl}} = \mid Q \mid_{SP-\text{Impl}} : \{\text{Semaphores} = \text{Semaphores}\} = \text{TRUE}\)

So, the exact SP-Impl match

\[
\text{Match}_{E-}[SP, \text{Impl}] (C, Q) = \text{TRUE}
\]

The candidate component specification matches with the query component Q specification under the Exact SP-Impl match.
2. **Exact SP-Relaxed Impl Match**  (Exact Synch. Policy, Relaxed Impl)

This match is a variation of the above match where the implementation policy of the required query is not the same as the component.

\[
\text{Match}_{[E-SP, R-Impl]}(C, Q) = [C_{SP} \Leftrightarrow Q_{SP}] \\
\land [C_{SP-Impl} \neq Q_{SP-Impl}]
\]

Consider the query component requirement for a bank system, the query search needs a method that returns the balance of the account. The synchronization policy that the method should execute is ProducerConsumer. The implementation required is Mutexes. The requirement of the query then matches that of a getBalance method under the Exact SP-Relaxed Impl match.

Below is the explanation at how the component specification matches with the query requirement.

Applying the information to the match,

\[
\text{Match}_{[E-SP, R-Impl]}(C, Q) = [C_{SP} \Leftrightarrow Q_{SP}] \\
\land [C_{SP-Impl} \neq Q_{SP-Impl}]
\]

The Exact SP-Relaxed Impl match above is a combination of two parts, the synchronization policy, the implementation of the function. TLA specifications are written to prove the synchronization policies and the implementation techniques used to prove the policies.

Synchronization policy of Query Component Q represented as: \(Q_{SP}\)

Implementation technique of Query Component Q represented as: \(Q_{SP-Impl}\)

Synchronization policy of Candidate Component C represented as: \(C_{SP}\)

Implementation technique of Query Component C represented as: \(C_{SP-Impl}\)

1. \(C_{SP} \Leftrightarrow Q_{SP} : [\text{ProducerConsumer} \Leftrightarrow \text{ProducerConsumer}] = \text{TRUE}\)

2. \(C_{SP-Impl} \neq Q_{SP-Impl} : [\text{Semaphores} \neq \text{Mutexes}] = \text{TRUE}\)
So, the Exact SP-Relaxed Impl match

\[ \text{Match}_{[E-SP, \ R-Impl]} \left( C, Q \right) = \text{TRUE.} \]

The component AccountOperationServer specification matches with the query component Q specification under the Exact SP-Relaxed Impl match.

3. **Relaxed SP-Exact Impl Match**  (Relaxed Synch. Policy, Exact Impl)

This match is a variation of the above match where the implementation policy of the required query is not the same as the component.

\[ \text{Match}_{[R-SP, E-Impl]} \left( C, Q \right) = \left[ C_{SP} \Rightarrow Q_{SP} \right] \]

\[ \wedge \left[ C_{SP-Impl} = Q_{SP-Impl} \right] \]

Consider the query component requirement for a bank system, the query search needs a method that returns account balance. The synchronization policy that the method should execute is MutualExclusion. The implementation required is Semaphores. The requirement of the query then matches that of a getBalance method under the exact SP-relaxed Impl match.

Below is the explanation at how the candidate component specification matches with the query requirement.

Applying the information to the match,

\[ \text{Match}_{[R-SP, E-Impl]} \left( C, Q \right) = \left[ C_{SP} \Rightarrow Q_{SP} \right] \]

\[ \wedge \left[ C_{SP-Impl} = Q_{SP-Impl} \right] \]

The Exact SP-Relaxed Impl match above is a combination of two parts, the synchronization policy, the implementation of the function. TLA specifications are written to prove the synchronization policies and the implementation techniques used to prove the policies.

Synchronization policy of Query Component Q represented as: \( Q_{SP} \)
Implementation technique of Query Component Q represented as: \( Q_{SP-Impl} \)

Synchronization policy of Candidate Component C represented as: \( C_{SP} \)

Implementation technique of Query Component C represented as: \( C_{SP-Impl} \)

1. \( | C |_{SP} \Rightarrow | Q |_{SP} : [ProducerConsumer \Rightarrow MutualExclusion] = TRUE \)

2. \( | C |_{SP-Impl} = | Q |_{SP-Impl} : [Semaphores = Semaphores] = TRUE \)

So, the Relaxed SP-Exact Impl match

\[ Match_{[R-SP, E-Impl]} (C, Q) = TRUE. \]

The component AccountOperationServer specification matches with the query component Q specification under the Relaxed SP-Exact Impl match. Similarly, the matches: Relaxed SP-Relaxed Impl match, Reverse Implication SP-Exact Impl match, and the Reverse Implication SP-Relaxed Impl match can be verified.

For all the synchronization policies considered in the project and as well as the implementation techniques considered for the synchronization policies TLA+ specifications have been written proving the relation that holds between them.

\[ IV. \text{ QoS Level Matching} \]

\[ \text{Generic QoS Match} \]

The representation of generic QoS match is given below

\[ \text{Match}_{\text{Gen-QoS}} (C, Q) = \{ \text{For each or required QoS Parameter of } Q \text{ and } C, \]

\[ (|C|_{QoS-ParamName(i)} \ R_1 \ |Q|_{QoS-ParamName(i)}) \]

\[ R_2 \]

\[ (|Q|_{ParamValue(i)} \ R_3 \]

\[ |C|_{ParamValue(i)} \ R_4 \]

\[ \{ |Q|_{ParamValue(i)} \ R_5 \ |Q|_{ParamDeV(i)} \} \} \]

\[ \} \]
Exact and Relaxed matches are considered for the generic QoS match and the validations of them are given below.

1. **Exact QoS Match**

The Exact QoS match is represented below:

\[
\text{Match}_{E-QoS}(C, Q) = \left\{ \text{For each or required QoS Parameter of } Q \text{ and } C, \right. \\
\left. (\lfloor C \rfloor \text{QoS-ParamName}(i) = \lfloor Q \rfloor \text{QoS-ParamName}(i)) \wedge \\
(\lfloor Q \rfloor \text{ParamValue}(i) R_3 \\
\lfloor C \rfloor \text{ParamValue}(i) R_4 \\
\{ \lfloor Q \rfloor \text{ParamValue}(i) R_5 [\lfloor Q \rfloor \text{ParamDeV}(i)] \}) \right\}
\]

where,

\[
[Q] \text{ParamDeV}(i) = \{ [Q] \text{ParamValue}(i) \times \text{Dev} / 100 \}
\]

Consider the query component requirement for the bank system, User needs a function to deposit an amount ‘X’ into an account in Bank domain and also imposes that the account balance must increase after the deposit. The synchronization policy that the method should execute is Mutual Exclusion. The implementation required is semaphores. Now adding the QoS requirement to it, the user needs two QoS parameters:

Throughput = 80 MIPS, Deviation = 5%
Turn-Around Time = 1000 msec, Deviation = 2%

The above requirement of the query so far matches the deposit Method of AccountOperationServer, under Exact Syntactic, Exact semantic and the Exact SP-Relaxed Impl match and the query requirement is applied to the Exact QoS match
Below is the explanation at how the component C specification matches with the query requirement.

Starting with the Throughput,
Let the candidate component matched at the upper level has the following QoS parameter values in addition to other parameters.
Throughput = 83 MIPS and Turn-Around Time = 990 msec
First, Applying information to the Exact QoS match,

\[
\text{Match } E-QoS(C, Q) = \{ \text{For each or required QoS Parameter of } Q \text{ and } C,}
\]

\[
(C \text{ QoS-Param}(i) = Q \text{ QoS-Param } (i))
\]

\[
\land
\]

\[
(Q \text{ ParamValue}(i) \ R_3
\]

\[
(C \text{ ParamValue}(i) \ R_4
\]

\[
\{ (Q \text{ ParamValue}(i) \ R_5 \ Q \text{ ParamDev(i) } ) \}
\]

where,

\[
Q \text{ ParamDev(i) } = \{ Q \text{ ParamValue(i) } \times \text{Dev} / 100 \}
\]

Consider for Throughput values of both Query component and candidate component.

\[
Q \text{ ThroughputValue } = 80 \text{ and } \text{Dev} = 5%
\]

\[
C \text{ ThroughputValue } = 83
\]

First, Applying Throughput information to the Exact QoS match,

1. \([(C \text{ QoS-ParamName}) = (Q \text{ QoS-ParamName})] = [\text{Throughput = Throughput}] = \text{TRUE}

2. Calculating the deviation value of the Query parameter

\[
Q \text{ ThroughputDev } = \{ Q \text{ ThroughputValue } \times \text{Dev} / 100 \} = 80 \times 5 / 100 = 4
\]

The deviation above is added to the original value of the Throughput to get the new value of the acceptable Throughput.

\[
Q \text{ Throughput } + Q \text{ ThroughputDev } = 80 + 4 = 84.
\]
So, the Exact QoS match is applied to the range of throughput values ranging from the original required Throughput and the accepted deviation Throughput value.

\[(|Q| ThroughputValue \leq |C| ThroughputValue \leq \{|Q| ThroughputValue + |Q| ThroughputDev\})\]

= (80 \leq 83 \leq 84) = TRUE

Finally,

TRUE \land TRUE = TRUE

So, throughput is matched under Exact QoS match.

Consider Turn-Around Time (TAT) values of both Query component and candidate component.

\[|Q| TAT = 1000 \text{ and } Dev = 2\%
\]

\[|C| TAT = 990\]

First, Applying TAT information to the Relaxed QoS match,

1. \[([C| QoS-ParamName) = (|Q| QoS-ParamName)] = [TAT = TAT] = TRUE\]

2. Calculating the deviation value of the Query parameter

\[|Q| TATDev = \{|Q| TATValue \times \frac{Dev}{100}\} = 1000 \times 2 / 100 = 20\]

The deviation above is subtracted to the original value of the throughput to get the new value of the acceptable throughput.

\[|Q| TATValue + |Q| TATDev = 1000 - 20 = 980\]

So, the Exact QoS match is applied to the range of Turn-Around Time (TAT) values ranging from the original required Turn-Around Time and the accepted deviation Turn-Around Time value.

\[(|Q| TATValue \geq |C| TATValue \geq \{|Q| TATValue - |Q| TATDev\})\]

= (1000 \geq 990 \geq 980) = TRUE

Finally,

TRUE \land TRUE = TRUE
So, TAT is matched under Exact QoS match.

Finally, the candidate component matches under the Exact QoS match for the Query Requirement.

**Match** \( E-QoS \left( C, Q \right) = \text{TRUE}. \)

The main aim behind the Exact QoS match is to provide a match that is equal or greater (better) than the required parameter values.

2. **Relaxed QoS Match**

The Relaxed QoS match is represented below:

\[ \text{Match} \ R-QoS \left( C, Q \right) = \{ \text{For each or required QoS Parameter of } Q \text{ and } C, \]

\[ \left( C \right)_{\text{QoS-ParamName}(i)} = \left( Q \right)_{\text{QoS-ParamName}(i)} \]

\[ \wedge \]

\[ \left\{ \left( Q \right)_{\text{ParamValue}(i)} R_5 \left( Q \right)_{\text{ParamDev}(i)} \right\} R_3 \]

\[ \left( C \right)_{\text{ParamValue}(i)} R_4 \]

\[ \left\{ \left( Q \right)_{\text{ParamValue}(i)} R_5' \left( Q \right)_{\text{ParamDev}(i)} \right\} \}

\]

where,

\[ \left[ Q \right]_{\text{ParamDev}(i)} = \left\{ \left[ Q \right]_{\text{ParamValue}(i)} \times \text{Dev} / 100 \right\} \]

Consider the query component requirement for the bank system, the query search needs a method that deposit an amount “X” into an account and the method when executed must ensure that the newbalance is greater than the oldbalance. The synchronization policy that the method should execute is Mutual Exclusion. The implementation required is semaphores. Now adding the QoS requirement to it, the user needs three QoS parameters:

Throughput = 80 MIPS, Deviation = 5%

Turn-Around Time = 1000 msec, Deviation = 2%
The above requirements of the query so far matches the deposit Method of AccountOperationServer, under Syntactic, Semantic and the exact SP-relaxed Impl match and the query requirement is applied to the Relaxed QoS match

Below is the explanation at how the candidate component specification matches with the query requirement.
Starting with the Throughput,
Let the candidate component matched at the upper level has the following QoS parameter values in addition to other parameters.
Throughput = 83 MIPS and Turn-Around Time = 990 msec
Throughput = 79 MIPS and Turn-Around Time = 1110 msec

First, Applying information to the Relaxed QoS match,

\[
\text{Match}_{R-QoS}( C, Q) = \{\text{For each or required QoS Parameter of } Q \text{ and } C, \}
\]

\[
( [C]_{QoS-Param(i)} = [Q]_{QoS-Param(i)} ) \land
\]

\[
( [Q]_{ParamValue(i)} R_3 [Q]_{ParamDev(i)} ) R_3 \land
\]

\[
[Q]_{ParamValue(i)} R_4 [C]_{ParamValue(i)} R_4
\]

\[
[Q]_{ParamValue(i)} R_5 [Q]_{ParamDev(i)}
\]

Consider for Throughput values of both Query component and candidate component.

\[
[Q]_{ThroughputValue} = 80 \text{ and } \text{Dev} = 5%\
\]

\[
[C_1]_{ThroughputValue} = 83, [C_2]_{ThroughputValue} = 79
\]

First, Applying Throughput information to the Exact QoS match,

\[
3. \quad \text{[([C]_{QoS-ParamName}) = ([Q]_{QoS-ParamName})]} = \text{[Throughput = Throughput]} = \text{TRUE}
\]
4. Calculating the deviation value of the Query parameter

\[ [Q]_{ThroughputDev} = \{ [Q]_{ThroughputValue} \times \text{Dev} / 100 \} = 80 \times 5 / 100 = 4 \]

The deviation above is added and subtracted to the original value of the Throughput to get the new value of the acceptable Throughput.

\[ [Q]_{Throughput} - [Q]_{ThroughputDev} = 80 - 4 = 76 \]
\[ [Q]_{Throughput} + [Q]_{ThroughputDev} = 80 + 4 = 84 \]

So, the Relaxed QoS match is applied to the range of throughput values ranging from the Min. and Max. values of Throughput after applying deviation:

\[ \{ [Q]_{ThroughputValue} - [Q]_{ThroughputDev} \} \leq [C]_{ThroughputValue} \leq \{ [Q]_{ThroughputValue} + [Q]_{ThroughputDev} \} \]

For the component \( C \), \( 76 \leq 83 \leq 84 \) = TRUE

Finally,

\[ \text{TRUE} \land \text{TRUE} = \text{TRUE} \]

So, throughput is matched for both the components under Relaxed QoS match.

Consider Turn-Around Time (TAT) values of both Query component and candidate component.

\[ [Q]_{TAT} = 1000 \text{ and } \text{Dev} = 2\% \]
\[ [C_1]_{TAT} = 990, [C_2]_{TAT} = 1100 \]

First, Applying TAT information to the Relaxed QoS match,

3. \[ ( [C]_{QoS-ParamName} = ([Q]_{QoS-ParamName}) ) = [TAT = TAT] = \text{TRUE} \]

4. Calculating the deviation value of the Query parameter
\[ \{ Q \}_{TATDev} = \{ Q \}_{TATValue} \times Dev / 100 \} = 1000 \times 2 / 100 = 20 \]

The deviation above is subtracted to the original value of the throughput to get the new value of the acceptable throughput.

\[ \{ Q \}_{TATValue} - \{ Q \}_{TATDev} = 1000 - 20 = 980 \]

\[ \{ Q \}_{TATValue} + \{ Q \}_{TATDev} = 1000 + 20 = 1020 \]

So, the Relaxed QoS match is applied to the range of TAT values ranging from the Min. and Max. values of TAT after applying deviation:

\[ \{ \{ Q \}_{TATValue} + \{ Q \}_{TATDev} \} \geq \{ C \}_{TATValue} \geq \{ \{ Q \}_{TATValue} - \{ Q \}_{TATDev} \} \) \]

For the component \( C \), \( (1020 \geq 990 \geq 980) = \text{TRUE} \)

Finally,

\[ \text{TRUE} \land \text{TRUE} = \text{TRUE} \]

So, TAT is matched for both the components under Relaxed QoS match.

The Relaxed QoS match aims at providing more components as the search results by applying a wide range of parameter values to the search. The match at the QoS either exact or relaxed aims at providing the match for all the proposed parameters in QoS or only the required parameters required in the query.

This concludes the validation of matching criteria considered for all the four levels in the component matching.
Appendix B: Algorithms to Implement Matchers

The algorithms implemented by the matchers are given in this Appendix B.

B.1: Representation of Query Requirement

The User Query requirement given in the front end is represented as a data structure and the required parameters are used for matching in the entire prototype. The data given by the user is divided into two data structures: Functions and ServiceAttributes.

- Functions, holds the functional attribute information of the Query. This represents the syntactic, semantic and synchronization information of the requirement.
  
  Functions []: FunctionID, FunctionName, Prototype, Style, Description, Technology, PreConditions, PostConditions, Invariants, Concurrency Contract, SynchPolicy, SynchImpl

- Service Attributes holds the non-functional information of the Query, which is the information regarding QoS Parameters.
  
  ServiceAttributes []: AttributeID, AttributeValue, AttributeDevValue, AttributeDescription, ParameterList

B.2: Representation of Input Level Parameters and Type of Level

Input Level Parameters: Syn, Sem, Sync, QoS
Where Syn represents Syntactic level, Sem represents Semantic, Sync represents Synchronization Level, and QoS represents Quality of Service Level.

Type Of Level: Syn_e, Syn_r, Sem_e, Sem_r, Sync_e, Sync_r, QoS_e, QoS_r
Syn_e represents the Exact option of Syntactic level and Syn_r represents the Relax option of Syntactic level. Similar representation is used for the rest of the levels.
**B.3: Representation of Final Result**

Final Result is a data structure that holds the information about the time required for each matcher, the count of components returned, and the Match Results.

**FinalResult []: RunningTimeInfo, Total Count of Components Returned, Match Results**

**B.4: Algorithms to initialize all the Matchers**

The query specifications are taken from the front end and the level of match selected and type of match selected is also taken as input. This returns the time required for each matcher, total count of components returned and Match results.

**Start_Matchers_Function**

**Input:** Input Level, Type of Level, Query Requirements, FunctionCount, DomainType

**Output:** FinalResult

/* Initialize FinalResult.RunningTimeInfo[ i ] where i = 0,1,2,3,4,5,for all the Matchers*/

FinalResult. RunningTimeInfo [i]= new TimeSpan (0,0,0,0,0)

/* If the Level of Match selected is Syntactic and the Type of Match selected is either Exact or Relax, the SyntacticMatcher is invoked. */

IF LevelOfMatch EQUALS “Syn”

IF TypeOfMatch EQUALS (Syn_e OR Syn_r)

/* SyntacticMatcher is used to calculate the number of components that satisfy the Syntactic match either Exact or Relax, and the time it takes to do the matching is returned */

FinalResult.RunningTimeInfo [0]

= CALL SyntacticMatcher (Functions, FunctionCount, DomainType)

ENDIF
IF LevelOfMatch EQUALS “Sem”
    IF TypeOfMatch EQUALS (Sem_e)
        /* SemanticExactMatcher is used to calculate the number of components that
           satisfy the semantic match with Exact type and the time it takes to do the
           matching is returned */
        FinalResult.RunningTimeInfo [1] = CALL SemanticExactMatcher (Functions, FunctionCount, DomainType)
    ELSEIF TypeOfMatch EQUALS (Sem_r)
        /* SemanticRelaxMatcher is used to calculate the number of components that
           satisfy the semantic match with Relax type and the time it takes to do the
           matching is returned */
    ENDIF
ENDIF

IF LevelOfMatch EQUALS “Sync”
    IF TypeOfMatch EQUALS (Sync_e OR Sync_r)
        /* SynchronizationMatcher is used to calculate the number of components that
           satisfy the synchronization match with Exact or Relax and the time it takes to do
           the matching is returned */
        FinalResult.RunningTimeInfo [3] = CALL SynchronizationMatcher (Functions, FunctionCount, DomainType)
    ENDIF
ENDIF
/* If the LevelOfMatch selected is QoS and the TypeOf Match selected is “Exact” or “Relax” then QoSMatcher is invoked. */

IF LevelOfMatch EQUALS “QoS”
  IF TypeOfMatch EQUALS (QoS_e OR QoS_r)
    /* QoSMatcher is used to calculate the number of components that satisfy the QoS match with Exact or Relax and the time is returned */
    FinalResult.RunningTimeInfo = CALL QoSMatcher (Functions, FunctionCount, DomainType)
  ENDIF
ENDIF

RETURN FinalResult. MatchResults = GetComponentInfo (CompID)

---

**B.5: Algorithm For Syntactic Matcher**

Syntactic Matcher
Input: Functions, FunctionCount, DomainType
Output: TimeSpan

For each component (CompID) in the database

/* Check if the component (CompID) being considered is that of required Type. i.e. check if same as the DomainType provided by the User*/
CHECK IF ValidType (DomainType, CompID)
/* For the same Component, (CompID) Get the number of Functions supported by the component */

\[\text{CompFunction} = \text{GetNumberOfFunctions}(\text{CompID})\]

/* If the Component, (CompID) being considered is of ValidType and the number of functions, \text{CompFunction} supported by that component is Greater than or Equal to the \text{FunctionCount} by the Query then get the rest of the info regarding syntactic match*/

\[
\text{IF}\ (\text{ValidType} (\text{DomainType}, \text{CompID}) \&\& \text{CompFunction} \geq \text{FunctionCount})
\]

/* Create a MatchMatrix with FuncitonCount and Comp Function */

\[
\text{MatchMatrix} = [\text{FunctionCount}, \text{CompFunction}]
\]

For Each Function specified in the Query

Get the Query Function Signature from Functions Data structure

\[\text{QuerySignature} = \text{Functions.Protoype}\]

For Each Function specified in the Component

Get the Component Function Signature for that component from database

\[\text{CFunctionSignature} = \text{Functions.Protoype}\]

\[\text{QFunctionParameters} = \text{QFunctionReturnType}, \text{QFunctionParameterType}\]

\[\text{CFunctionParameters} = \text{CFunctionReturnType}, \text{QFunctionParameterType}\]

/* Signature Match takes the Query parameters, Component parameters, Query Variable Type, Component Variable Type which returns the MatchMatrix with Function Count and Comp Function. */

\[\text{MatchMatrix} = \text{CALL Signature\_Match}\]

Increment FunctionCount

\[\text{ENDIF}\]
OfTypeMatch_Syntax [CompID] = SyntacticEvaluateMatchMatrix [MatchMatrix, FunctionCount, CompFunction]
END FOR

Return the TimeSpan for Implementing Syntactic Matcher
Return Time

**B.5.1: Algorithm for Signature Match**

Signature_Match
Input: QFunctionParameters, CFunctionParameters
Output: Match of MatchMatrix returned

/* If Syntactic Exact option is checked, then the match Exact Type Match is Implemented. If the above match is satisfied then the Exact match is satisfied and the Match is given a value of 2 */
IF Syn_e EQUALS TRUE

    IF ((Ordered_Exact_Match || ReOrdered_Exact_Match) == TRUE)
        Match = 2
    ENDIF

/* If Syntactic Relax option is checked, the suitable match among the four matches is implemented. The two matches implemented here are Exact Type Match, Relax Type Match. If any of the matches is satisfied it is assigned a match value of 1. */
ELSE IF Syn_r EQUALS TRUE

    IF ((Ordered_Exact_Match || ReOrdered_Exact_Match) == TRUE)
        Match = 1
    ELSE IF ((Ordered_Relax_Match || ReOrdered_Relax_Match) == TRUE)
\[
\text{Match} = 1
\]

/* Match returned is assigned to the Match Matrix, which gives the type of match between the Query Function and the Candidate Function */
RETURN Match

**B.5.1 (a): Algorithm For Ordered\_Exact\_Match**

Ordered\_Exact\_Match
Input: QueryType, CandidateType
Output: Boolean Result (TRUE or FALSE)

IF QueryType.Length EQUALS CandidateType.Length
   For all the types in the Query and Candidate
      CALL Exact\_Match (QueryType, CandidateType)
   Return Match
ENDIF
ELSE
   Return FALSE

**B.5.1 (b): Algorithm For ReOrdered\_Exact\_Match**

ReOrdered\_Exact\_Match
Input: QueryType, CandidateType
Output: Boolean Result (TRUE or FALSE)

IF QueryType.Length EQUALS CandidateType.Length
   /* This match is different from the Ordered Exact Match in the order of the arguments. So, each type in the query is compared with every argument type in candidate. If the match is found then the match is returned */
For each type in the Query
For each type in the Candidate
CALL Exact_Match (QueryType, CandidateType)
Return Match
ENDIF
ELSE
Return FALSE

B.5.1 (c): Algorithm For Ordered_Relax_Match

Ordered_Relax_Match
Input: QueryType, CandidateType
Output: Boolean Result (TRUE or FALSE)

IF QueryType.Length EQUALS CandidateType.Length
For all the types in the Query and Candidate
CALL [Exact_Match (QueryType, CandidateType) OR
Inheritance_Match (QueryType, CandidateType) OR
Coercion_Match (QueryType, CandidateType)]
Return Match
ENDIF
ELSE
Return FALSE

B.5.1 (d): Algorithm For ReOrdered_Relax_Match

ReOrdered_Relax_Match
Input: QueryType, CandidateType
Output: Boolean Result (TRUE or FALSE)

IF QueryType.Length EQUALS CandidateType.Length
/*Each type in Query is compared with all the types in candidate component since the order of arguments here is different*/

For each type in the Query
For each type in the Candidate

CALL [Exact_Match (QueryType, CandidateType) OR
Inheritance_Match (QueryType, CandidateType) OR
Coercion_Match (QueryType, CandidateType)]

Return Match

ENDIF
ELSE
Return FALSE

B.5.1 (e): Algorithm For Exact_Match

/* Exact Match as name suggests compares the types of the Query and Candidate and if
Equal returns TRUE or else FALSE */

Exact_Match

Input: QueryType, CandidateType
Output: Boolean Result (TRUE or FALSE)

/* For the corresponding Query type and Candidate type get the ID from the Exact Table
in Database */

Q_ID =Get ID from Exact Table where Type EQUALS “QueryType”
C_ID =Get ID from Exact Table where Type EQUALS “CandidateType”

IF Q_ID EQUALS C_ID
Match = TRUE
ELSE
Match = FALSE
Return Match
**B.5.1 (f): Algorithm For Inheritance Match**

/* Inheritance Match as name suggests compares if inheritance relation exists between the Query and Candidate types. If relation exists it returns TRUE or else FALSE */

Exact_Match

Input: QueryType, CandidateType

Output: Boolean Result (TRUE or FALSE)

/* For the corresponding Query type and Candidate type get the ID from the Inheritance Table in Database */

Q_ID = Get ID from Inheritance Table where Type EQUALS “QueryType”

IF Q_ID EQUALS CandidateType
   Match = TRUE
ELSE
   Match = FALSE

Return Match

**B.5.1 (g): Algorithm For Coercion Match**

/* Coercion Match checks if coercion relation can be applied between the Query and Candidate types. If relation exists it returns TRUE or else FALSE */

Exact_Match

Input: QueryType, CandidateType

Output: Boolean Result (TRUE or FALSE)

/* For the corresponding combination of Query type and Candidate type get the CoercionID from the Coercible Table in Database */

CoercionID = Get ID from Coercible Table where Type EQUALS “QueryType”
IF CoercionID > 0
Match = TRUE
ELSE
Match = FALSE

Return Match

The above algorithms are implemented in Syntactic Matcher and the returned components satisfy either Exact match or Relax match. Exact match is assigned a value of 2 and Relax components are assigned value 1. The values assigned are used to determine the type of match in the MatchMatrix.

B.6: Algorithm For Semantic Matcher

There are two implementations for Semantic Matcher based on the kind of match selected either “Exact” or “Relax”. If Exact is selected, SemanticExactMatcher is implemented and if Relax is selected, SemanticRelaxMatcher is implemented. The algorithms for SemanticExactMatcher and SemanticRelaxMatcher are given below.

B.6.1: Algorithm For SemanticExactMatcher

SemanticExactMatcher
Input: Functions, FunctionCount, DomainType
Output: TimeSpan

For each component (CompID) in the database

/* Check if the component (CompID) being considered is that of required Type. i.e. check if same as the DomainType provided by the User*/
CHECK IF (TypeOfMatch_Syntax [CompID] ==2) ||
(TypeOfMatch_Syntax [CompID] ==1)
/* For the same Component, (CompID) Get the number of Functions supported by the component */

\[
\text{CompFunction} = \text{GetNumberOfFunctions} (\text{CompID})
\]

/* If the Component, (CompID) being considered satisfied the syntactic match either Exact or Relax and the number of functions, CompFunction supported by that component is Greater than or Equal to the FunctionCount by the Query then get the rest of the information regarding semantic match */

\[
\text{IF } ((\text{TypeOfMatch\_Syntax} [\text{CompID}] == 2 || \\
\text{TypeOfMatch\_Syntax} [\text{CompID}] == 1) \\
\text{&&} \\
\text{CompFunction} \geq \text{FunctionCount})
\]

/* Create a MatchMatrix with FunctionCount and Comp Function */

\[
\text{MatchMatrix} = [\text{FunctionCount}, \text{CompFunction}]
\]

For Each Function specified in the Query

/* Get the Pre-Conditions and Post-Conditions from Functions Data structure */

\[
\text{Qpre} = \text{Functions. Preconditions} \\
\text{Qpost} = \text{Functions. Preconditions}
\]

For Each Function specified in the Component [CompID]

/* Get the Component Function Signature for that component from database */

\[
\text{CSemanticData} = \text{Select PreConditions, PostConditions from Functions} \\
\text{Where ComponentID EQUALS ComponentID}
\]

/* With CSemanticData open a Database Connection and read the data in the command using a Reader */

\[
\text{While (Reader. Read ())}
\]
/* Get the Pre-Conditions and Post-Conditions from CSemanticData */

pre EQUALS Get Pre-Conditions in the Reader
post EQUALS Get Post-Conditions in the Reader

/* Create a String Extension with the CompID, Individual Function Count of Query, Individual Function Count of Component */

Extension EQUALS “_” + CompID + “_” + individual Query Function Count + Individual Comp Function Count

/* WriteFunctionForSemanticMatch takes the TypeOfMatch, Extension, Query pre-conditions, Query post-conditions, Component pre-conditions, Component pre-conditions which returns the MatchMatrix with Function Count and Comp Function. */

MatchMatrix = CALL WriteFunctionForSemanticMatch
Increment FunctionCount

/* Close the Reader and Close the Connection*/

Reader. Close

ENDIF
END FOR

/* Write Pre/Post conditions to ExactSemantic.ssc file and compile using Spec# */

CALL Compile (“ExactSemantic.ssc”, -debug)

/* Pass the output of the ExactSemantic.ssc file to Boogie. It also takes other arguments such as the FunctionCount in CompID, FunctionCount in Query, Semantically considered components, TypeOfMatch */

CALL Boogie (“ExactSemantic.exe”, FunctionCount [CompID], Function Count, SemanticCompCount, “Exact”)

END
Return the TimeSpan for Implementing Semantic Matcher

Return Time

In a similar manner the algorithm for SemanticRelaxMatcher is also implemented with
TypeOfMatch equal to Relax.

**B.6.1 (a): Algorithm For WriteFunctionForSemanticMatch**

WriteFunctionForSemanticMatch
Input: TypeOfMatch, Extension, Qpreconditions, Qpostconditions, Cpreconditions, Cpostconditions
Output: Writes Pre conditions and Post conditions to a file and given as input to the
Spec# programming system

**Define a set of Operators**

/* To get the variables in the preconditions and postconditions of query and candidate
component */

/* Get variables in Precondition of Query*/

For each Qprecondition
/*GET precondition Variable using Split operation*/
QpreVariable = Qprecondition. Split (operators)
Store the QpreVariable in a global array of Variables
END FOR

/* Get variables in Postcondition of Query*/

For each Qpostcondition
/*GET postcondition Variable using Split operation*/
QpostVariable = Qpostcondition. Split (operators)
Store the QpostVariable in a global array of Variables
END FOR

/* To get the variables in the preconditions of component */
For each Cprecondition
/*GET precondition Variable using Split operation*/
CpreVariable = Cprecondition. Split (operators)
IF already exists do not add
ELSE Store the CpreVariable in a global array of Variables
END FOR

/* To get the variables in the preconditions and postconditions of query and candidate component */
For each Qpostcondition
/*GET postcondition Variable using Split operation*/
CpostVariable = Cpostcondition. Split (operators)
IF already exists do not add
ELSE Store the CpostVariable in a global array of Variables
END FOR

/*After Getting the Pre/Post conditions of Query and Candidate component for each function, write it onto to a file, which would be given as an input to Spec# file. Depending on the TypeOfMatch selected, the matches proposed are written on to the file*/

IF TypeOfMatch EQUALS “Exact”

Open TextWriter TW
/*Write the definition of Variables of the Pre and Post conditions considered*/
TW. WriteLine (Variables)
/*Write the Pre Conditions of Candidate Component Function*/
TW. WriteLine ("public void original" + Extension + "()"
TW. WriteLine ("requires")
For Each Cprecondition
TW. WriteLine (Cprecondition)

/*Write the Post Conditions of Query Function*/
TW. WriteLine (“ensures”)
TW. WriteLine (Qpostcondition)
TW. WriteLine (“;”)

/*Write the Pre Conditions of Query Function*/
TW. WriteLine ("public void refinement" + Extension + “()”)
TW. WriteLine (“requires”)
For Each Qprecondition
TW. WriteLine (Qprecondition)

/*Write the Pre Conditions of Candidate Component Function*/
TW. WriteLine (“ensures”)
TW. WriteLine (Cpostcondition)
TW. WriteLine (“;”)

/* Write the Match Function for Exact Pre/Post*/
TW. WriteLine (“ensures”)
TW. WriteLine (Qprecondition)

/*Exact Pre/Post Match Implements Equivalence Relation between the Pre/Post conditions*/
TW. WriteLine (“ ⇔ ”)
TW. WriteLine (Cprecondition)
TW. WriteLine (&&)
TW. WriteLine (Cpostcondition)
TW. WriteLine (“ ⇔ ”)
TW. WriteLine (Qpostcondition)

ENDIF
/* If TypeOfMatch is Relaxed then this is written to the file*/

IF TypeOfMatch EQUALS “Relax1”

   Open TextWriter TW
   /*Write the definition of Variables of the Pre and Post conditions considered*/
   TW. WriteLine (Variables)
   /*Write the Pre Conditions of Candidate Component Function*/
   TW. WriteLine (“public void original” + Extension + “()”)
   TW. WriteLine (“requires”)
   For Each Cprecondition
   TW. WriteLine (Cprecondition)

   /*Write the Post Conditions of Query Function*/
   TW. WriteLine (“ensures”)
   TW. WriteLine (Qpostcondition)
   TW. WriteLine (“;”)

   /*Write the Pre Conditions of Query Function*/
   TW. WriteLine (“public void refinement” + Extension + “()”)
   TW. WriteLine (“requires”)
   For Each Qprecondition
   TW. WriteLine (Qprecondition)

   /*Write the Pre Conditions of Candidate Component Function*/
   TW. WriteLine (“ensures”)
   TW. WriteLine (Cpostcondition)
   TW. WriteLine (“;”)

   /* Write the Match Function for Relax Pre/Post*/
   TW. WriteLine (“ensures”)
   TW. WriteLine (Qprecondition)
/*Exact Pre/Post Match Implements Implication Relation between the Pre/Post conditions*/

TW. WriteLine (“⇒”)
TW. WriteLine (precondition)
TW. WriteLine (&&)
TW. WriteLine (postcondition)
TW. WriteLine (“⇒”)
TW. WriteLine (Qpostcondition)

ENDIF

/* If TypeOfMatch is Relaxed then this is written to the file*/

IF TypeOfMatch EQUALS “Relax2”

Open TextWriter TW

/*Write the definition of Variables of the Pre and Post conditions considered*/

TW. WriteLine (Variables)

/*Write the Post Conditions of Query Function*/

TW. WriteLine (“public void original” + Extension + “()”)
TW. WriteLine (“requires”)
For Each Qpostcondition
TW. WriteLine (Qpostcondition)
TW. WriteLine (“;”)

/*Write the Post Conditions of Candidate Component Function*/

TW. WriteLine (“public void refinement” + Extension + “()”)
TW. WriteLine (“ensures”)
TW. WriteLine (Cpostcondition)
TW. WriteLine (“;”)

/* Write the Match Function for Relax Post*/

TW. WriteLine (“ensures”)
TW. WriteLine (Cpostcondition)
TW. WriteLine (“ ⇒ ”)
TW. WriteLine (Qpostcondition)
TW. WriteLine (“;”)

ENDIF

B.7: Algorithm For Synchronization Matcher

Synchronization Matcher
Input Parameters: QuerySynchronizationPolicy, QuerySynchImplementation, DomainType, FunctionCount
Output: TimeSpan

For each component (CompID) in the database

/* Check if the component (CompID) being considered is that of required Type. i.e. check if same as the DomainType provided by the User*/
    CHECK IF (TypeOfMatch_Semantic [CompID] ==2) ||
            TypeOfMatch_Semantic [CompID] ==1 ||
            TypeOfMatch_Semantic [CompID] ==3)

/* For the same Component, (CompID) Get the number of Functions supported by the component */
    CompFunction = GetNumberOfFunctions (CompID)
/* If the Component, (CompID) being considered satisfied the semantic match either Exact or Relax and the number of functions, CompFunction supported by that component is Greater than or Equal to the FunctionCount by the Query then get the rest of the information regarding semantic match*/
IF ((TypeOfMatch_Semantic [CompID] ==2 ||
    TypeOfMatch_Semantic [CompID] ==1 ||
    TypeOfMatch_Semantic [CompID] ==3)
&&

CompFunction >= FunctionCount)

For Each Function specified in the Query
/* Get the Pre-Conditions and Post-Conditions from Functions Data structure */
QSynchPolicy = Functions. SynchronizationPolicy
QSynchImpl = Functions. SynchronizationImpl

For Each Function specified in the Component [CompID]
/* Get the Component SynchPolicy and SynchImpl for that component from database */
CSynchData = Select SynchPolicy, SynchImpl from Functions Where
ComponentID EQUALS CompID

/* With CSynchData open a Database Connection and read the data in the command using a Reader */
While (Reader. Read ())
/* Get the SynchPolicy and SynchImpl from CSynchData */
CSynchPolicy EQUALS Get SynchPolicy in the Reader
CSynchImpl EQUALS Get SynchImpl in the Reader
Reader. Close ()

/* Considering only the Synchronization Policies and implementing the Exact and Relax matches under synchronization level */
IF (CSynchPolicy! = NULL && CSynchImpl ==NULL)
   IF (Sync_e EQUALS TRUE && CSynchPolicy EQUALS QSynchPolicy)
      TypeOfMatch_Synch [CompID] = 2
   IF (Sync_r EQUALS TRUE)
      IF (CSynchPolicy EQUALS QSynchPolicy || ! CSynchPolicy EQUALS QSynchPolicy)
         TypeOfMatch_Synch [CompID] = 1
/* Considering Synchronization Policies and Synchronization Implementations and implementing the Exact and Relax matches under synchronization level */
IF (CSynchPolicy != NULL && CSynchImpl != NULL)
    IF (Sync_e EQUALS TRUE &&
        CSynchPolicy EQUALS QSynchPolicy &&
        CSynchImpl EQUALS QSynchImpl)
        TypeOfMatch_Synch [CompID] = 2
    IF (Sync_r EQUALS TRUE)

        IF (CSynchPolicy EQUALS QSynchPolicy ||
            CSynchImpl EQUALS QSynchImpl)
            TypeOfMatch_Synch [CompID] = 1

    ELSE IF (CSynchPolicy EQUALS QSynchPolicy ||
            CSynchImpl EQUALS QSynchImpl)
            TypeOfMatch_Synch [CompID] = 1

    ELSE IF (CSynchPolicy EQUALS QSynchPolicy ||
            CSynchImpl EQUALS QSynchImpl)
            TypeOfMatch_Synch [CompID] = 1

ENDIF
ENDFOR
Return the TimeSpan for Implementing Synchronization Matcher
Return Time
B.8: Algorithm For QoS Matcher

QoS Matcher
Input Parameters: Service Attributes, QoS Count, DomainType
Output: TimeSpan

For each component (CompID) in the database

/* Check if the component (CompID) being considered is that of required Type. i.e. check if same as the DomainType provided by the User*/
CHECK IF (TypeOfMatch_Synchronization [CompID] ==2) ||
    TypeOfMatch_Synchronization [CompID] ==1)

/* If the Component, (CompID) being considered satisfied the Synchronization match either Exact or Relax and the number of functions, then get the rest of the information regarding QoS*/
IF ((TypeOfMatch_Synchronization [CompID] ==2 ||
    TypeOfMatch_Synchronization [CompID] ==1)

For each specified Query
/* Get the Q_QoSAttributeName, Q_QoSAttributeValue, and DeviationValue */
Q_QoSAttributeName = ServiceAttributes. AttributeName
Q_QoSAttributeValue = ServiceAttributes. AttributeValue
Q_QoSAttributeDevValue = ServiceAttributes. AttributeDevValue

For each specified Candidate Component [CompID]
/* Get the Component QoSAttributeName, QoSAttributeValue for that component from database */
CQoSData = Select QoSID, QoSAttribute, QoSValue from QoSContracts
    Where ComponentID EQUALS CompID
/* With CQoSData open a Database Connection and read the data in the command using a Reader */

While (Reader. Read ())
/* Get the QoSAttributeName, QoSAttributeValue from CQoSData */
C_QoSPropertyName EQUALS Get QoSAttribute in the Reader
C_QoSPropertyValue EQUALS Get QoSValue in the Reader

/*Applying Matches proposed under QoS Match*/

IF Q_QoSPropertyName EQUALS C_QoSPropertyName
  IF (ActualVal <= C_QoSPropertyValue &&
      C_QoSPropertyValue <= EMaxVal)
    TypeOfMatch_QoS [CompID] = 2
  ELSE IF (RMinVal <= C_QoSPropertyValue &&
           C_QoSPropertyValue <= RMaxVal)
    TypeOfMatch_QoS [CompID] = 1
  ENDFILE
  Reader. Close ()
ENDFOR

Return the TimeSpan for Implementing QoS Matcher

Return Time

Above is the explanation of important algorithms implemented in the prototype to prove the merit of Multilevel matching as opposed to single level matching.
Appendix C

This Appendix presents:

- Representations of Spec# files and the Boogie output for a sample query requirement.
- An Example for matching the TLA specifications of two synchronization policies: Single ProducerConsumer and MutualExclusion.

1. Representation of Spec# and Boogie.exe Output Files

    The function requirement of a sample query is given below. The semantic matcher uses Spec# programming language to get the candidate components that match the given query requirement. Based on the type of match selected, i.e. either exact or relaxed the corresponding match functions are executed.

    Function Required: void withdraw (float)
    Pre-Condition Requirement: acctID!= null; balance>0.0;
    Post-Condition Requirement: balance ==balance-amount;

    Spec# Programming files for Semantic Exact Match and Semantic Relaxed Match and their corresponding Boogie.exe output files are shown below. Figure A-3 corresponds to the Spec# input file for Semantic Exact Match and Figure A-4 presents the corresponding Boogie.exe output file. Figure A-4 and Figure A-5 represent the spec# input file corresponding to the two relaxed matches considered under Semantic Relaxed Match and Figure A-6 shows the Boogie.exe output file for the given input.
Figure A-3: Spec# programming file showing SemanticExactTest.ssc
public void refinement_18_0_0()  
requires balance>0 && acctid!=null;  
ensures newbalance<oldbalance;  
ensures ( balance>0 && acctid!=null \iff balance>0 )  
&& (newbalance<oldbalance \iff balance==balance-amount);  
{  
  assume false;  
}

public void original_24_0_0()  
requires balance>0 && acctid!=null;  
ensures balance==balance-amount;  
{  
  refinement_24_0_0();  
}

public void refinement_24_0_0()  
requires balance>0 && acctid!=null;  
ensures newbalance<oldbalance;  
ensures ( balance>0 && acctid!=null \iff balance>0 )  
&& (newbalance<oldbalance \iff balance==balance-amount);  
{  
  assume false;  
}

public void original_35_0_0()  
requires balance>0;  
ensures balance==balance-amount;  
{  
  refinement_35_0_0();  
}

public void refinement_35_0_0()  
requires balance>0 && acctid!=null;  
ensures balance==balance-amount;  
ensures ( balance>0 && acctid!=null \iff balance>0 )  
&& (balance==balance-amount \iff balance==balance-amount);  
{  
  assume false;  
}

public static void Main (string[] args){}
Below is the outcome of the above Spec# file when given to Boogie.exe which uses Simplify theorem prover to find which candidate components match to the given query requirement. Boogie.exe indicates which components do not match among the considered candidate components and also specifies whether the match fails at the pre- or post-conditions. Boogie.exe output file finishes with three errors for the given query requirement in Figure A-3. The match is failed for three candidate components with component ID’s: 14, 18 and 35 for the Semantic Exact Match.

```
Spec# Program Verifier Version 0.70, Copyright (c) 2003-2006, Microsoft.

14_0_0(), unsatisfied precondition: requires balance>0 && acctid!=null ;
18_0_0(), unsatisfied precondition: requires balance>0 && acctid!=null ;
35_0_0(), unsatisfied precondition: requires balance>0 && acctid!=null ;

finished with 3 errors
```

Figure A-4: Boogie.exe Output File for Semantic Exact Match

The same sample query is considered for the relaxed type option under Semantic Match, two different relaxed type matches are executed: relaxed pre/post match and relaxed post match. The components are first matched against relaxed pre/post match and relaxed post match. The components are first matched against relaxed pre/post match represented in SemanticRelaxTest.ssc (shown in Figure A-5) and the components that do not satisfy relaxed pre/post match if any, are then matched against relaxed post match represented in SemnaticRelax1Test.ssc (shown in Figure A-6). Both the files are given to Boogie.exe and the final outcome of Semantic Relaxed Match is shown in Figure A-7.
```csharp
//SemanticRelaxTest.ssc
//This Spec# file is created for Semantic Matching...Do not Edit!

using System;
using System.Data;

public class SemanticRelaxTest{
    public int balance;
    public string acctid;
    public int amount;
    public int newbalance;
    public int oldbalance;

    public void original_14_0_0()
    requires balance>0;
    ensures balance==balance-amount;
    {
        refinement_14_0_0();
    }

    public void refinement_14_0_0()
    requires balance>0 && acctid!=null;
    ensures newbalance<oldbalance && balance==balance-amount;
    ensures ( balance>0 && acctid!=null ==> balance>0 )
    && ( newbalance<oldbalance && balance==balance-amount
    ==> balance==balance-amount ) ;
    {
        assume false;
    }

    public void original_18_0_0()
    requires balance>0;
    ensures balance==balance-amount;
    {
        refinement_18_0_0();
    }
}
```

Figure A-5: Spec# Programming file showing SemanticRelaxTest.ssc
```java
public void refinement_18_0_0()
requires balance>0 && acctid!=null;
ensures newbalance<oldbalance;
ensures (balance>0 && acctid!=null ==> balance>0)
&& (newbalance<oldbalance ==> balance==balance-amount);
{
  assume false;
}

public void original_24_0_0()
requires balance>0 && acctid!=null;
ensures balance==balance-amount;
{
  refinement_24_0_0();
}

public void refinement_24_0_0()
requires balance>0 && acctid!=null;
ensures newbalance<oldbalance;
ensures (balance>0 && acctid!=null ==> balance>0 && acctid!=null)
&& (newbalance<oldbalance ==> balance==balance-amount);
{
  assume false;
}

public void original_35_0_0()
requires balance>0;
ensures balance==balance-amount;
{
  refinement_35_0_0();
}

public void refinement_35_0_0()
requires balance>0 && acctid!=null;
ensures balance==balance-amount;
ensures (balance>0 && acctid!=null ==> balance>0)
&& (balance==balance-amount ==> balance==balance-amount);
{
  assume false;
}

public static void Main (string[] args){}
```

Figure A-5: Spec# Programming file showing SemanticRelaxTest.ssc (Continued..)
using System;
using System.Data;

class SemanticRelax1Test {
    public int balance;
    public string acctid;
    public int amount;
    public int newbalance;
    public int oldbalance;

    public void original_14_0_0()
    {
        refinement_14_0_0();
    }

    public void refinement_14_0_0()
    {
        assume false;
    }

    public void original_18_0_0()
    {
        refinement_18_0_0();
    }

    public void refinement_18_0_0()
    {
        assume false;
    }

    public void original_14_0_0()
    {
        refinement_14_0_0();
    }

    public void refinement_14_0_0()
    {
        assume false;
    }

    public void original_18_0_0()
    {
        refinement_18_0_0();
    }

    public void refinement_18_0_0()
    {
        assume false;
    }
}

Figure A-6: Spec# Programming file showing SemanticRelax1Test.ssc
```java
public void original_35_0_0()
requires balance==balance-amount;
{
    refinement_35_0_0();
}

public void refinement_35_0_0()
requires balance==balance-amount;
ensures (balance==balance-amount ==> balance==balance-amount);
{
    assume false;
}

public static void Main (string[] args){}
```

Figure A-6: Spec# Programming file showing SemanticRelax1Test.ssc

Spec# Program Verifier Version 0.70, Copyright (c) 2003-2006, Microsoft.

18_0_0(), unsatisfied precondition: requires balance==balance-amount;

finished with 1 error

Figure A-7: Boogie.exe Output File for Semantic Relaxed Match

Boogie.exe output file finishes with 1 error for the given query requirement under Semantic Relaxed Match. The match has failed for candidate component with component ID : 18 as seen in Figure A-7.
2. Example TLA⁺ Specification Module

Below is an example TLA⁺ specification matching the two synchronization policies: ProducerConsumer and Mutual Exclusion and its corresponding Configuration file.

```plaintext
(* This module specification compares the behavior of synchronization policies ProducerConsumer & *)
(* MutualExclusion. The CONSTANT Process indicates the set of process that are trying access the *)
(* shared variable that is protected using the ProducerConsumer Policy *)

EXTENDS Sequences, Naturals, TLC
CONSTANT PClient, CClient, N, Input, Client1
VARIABLE Buffer, stateP, stateC

ASSUME (N \in Nat) \land (N > 0)

Client == PClient \cup CClient
Invariant == Len (Buffer) \leq N

TypeCheck == \land Buffer \in Seq (Input)
             \land Len (Buffer) \leq N

InitPC == \land Buffer = << >>
         \land TypeCheck
         \land stateP = "IDLE"
         \land stateC = "IDLE"
```
(***************************************************************************************)
(*Defining the methods supported by the synchronization policies ProducerConsumer and MutualExclusion*)
(***************************************************************************************)

Start_Produce (p, i) == \ p \in PClient
\ i \in Input
\ stateP = "IDLE"
\ Len(Buffer) < N
\ stateC = "IDLE"
\ stateP' = "EXECUTING"
\ UNCHANGED <<stateC, Buffer>>

End_Produce (p, i) == \ p \in PClient
\ stateP = "EXECUTING"
\ stateC = "IDLE"
\ Buffer' = Append(Buffer, i)
\ stateP' = "IDLE"
\ UNCHANGED <<stateC>>

Start_Consume (c) == \ c \in CClient
\ stateP = "IDLE"
\ stateC = "IDLE"
\ Len(Buffer) > 0
\ stateC' = "EXECUTING"
\ UNCHANGED <<stateP, Buffer>>

End_Consume (c) == \ c \in CClient
\ stateC = "EXECUTING"
\ stateP = "IDLE"
\ Buffer' = Tail(Buffer)
\ stateC' = "IDLE"
\ UNCHANGED <<stateP>>

RequestP (p) == \ stateP = "IDLE"
\ stateP' = "WAITING"
\ UNCHANGED <<stateC>>

AcquireP (p) == \ stateP = "WAITING"
\ \ A q \in Client1: stateP[q] # "EXECUTING"
\ stateP' = "EXECUTING"
\ UNCHANGED <<stateC>>
ReleaseP (p) ⇔ \( \land \) stateP = "EXECUTING"
\( \land \) stateP' = "RELEASED"
\( \land \) UNCHANGED <<stateC>>

exitingP (p) ⇔ \( \land \) stateP = "RELEASED"
\( \land \) stateP' = "IDLE"
\( \land \) UNCHANGED <<stateC>>

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

NextPC = \( \forall \) p \( \in \) PClient, c \( \in \) CClient, i \( \in \) Input: Start_Produce(p, i) \( \lor \) End_Produce(p, i)
\( \lor \) Start_Consume(c) \( \lor \) End_Consume(c)

NextP = \( \forall \) p \( \in \) Client: RequestP(p) \( \lor \) AcquireP(p) \( \lor \) ReleaseP(p) \( \lor \) exitingP(p)

PNextPC1 = NextPC \( \land \) Print(Buffer, TRUE)
PNextPC2 = NextPC \( \land \) Print(stateP, TRUE)
PNextPC3 = PNextPC2 \( \land \) Print(stateC, TRUE)
PNextP = NextP \( \land \) Print(stateP, TRUE)

(*******************************************************************************)
(* Defining the Specification. *)
*******************************************************************************)

SpecPC = InitPC \( \land \)[NextPC] <<stateP, stateC, Buffer>>

SpecME = InitPC \( \land \)[NextP] <<<>

THEOREM SpecPC ⇔ SpecME

(*******************************************************************************)
(*******************************************************************************)

Figure A-8: TLA+ Module Specification PCTOME.tla (Continued..)
TLA\textsuperscript{+} uses TLC Model Checker to find errors in the specification written to match the two synchronization policies. TLA\textsuperscript{+} module and Configuration file are the inputs to the TLC. The shown TLA\textsuperscript{+} specification matching the synchronization policies executes without errors indicating that the two policies can be matched.

This ends the sample examples shown in this Appendix for the Semantic Match and the Synchronization Match.