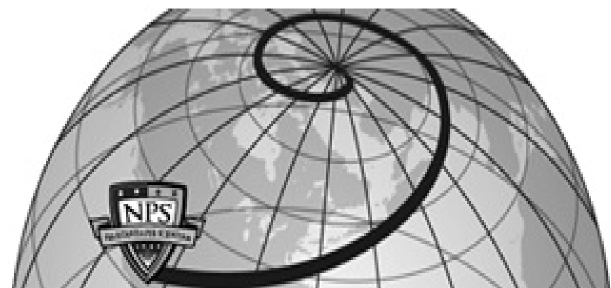
Calhoun
Institutional Archive of the Naval Postgraduate School



Calhoun: The NPS Institutional Archive

Theses and Dissertations Thesis Collection

1991-12

Reusable software component retrieval via normalized algebraic specifications

Steigerwald, Robert Allen

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/26733

Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

NAVAL POSTGRADUATE SCHOOL
Monterey, California

DISSERTATION

REUSABLE SOFTWARE COMPONENT RETRIEVAL
VIA
NORMALIZED ALGEBRAIC SPECIFICATIONS

by

ROBERT ALLEN STEIGERWALD

December, 1991

Thesis Advisor: Dr. Luqi

Approved for public release; distribution unlimited

T260314

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Field	Entry
1a. REPORT SECURITY CLASSIFICATION	UNCLASSIFIED
1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
3. DISTRIBUTION/AVAILABILITY OF REPORT	Approved for public release; distribution is unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	
5. MONITORING ORGANIZATION REPORT NUMBER(S)	
6a. NAME OF PERFORMING ORGANIZATION	Computer Science Dept. Naval Postgraduate School
6b. OFFICE SYMBOL (if applicable)	CS
7a. NAME OF MONITORING ORGANIZATION	Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code)	Monterey, CA 93943-5000
7b. ADDRESS (City, State, and ZIP Code)	Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	National Science Foundation/AJPO
8b. OFFICE SYMBOL (if applicable)	
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	CCR-9058453
8c. ADDRESS (City, State, and ZIP Code)	Washington, DC 20550

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*

REUSABLE SOFTWARE COMPONENT RETRIEVAL VIA NORMALIZED ALGEBRAIC SPECIFICATIONS (U)

12. PERSONAL AUTHOR(S)
Steigerwald, Robert A.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
PhD Dissertation	FROM ______ TO ______	December, 1991	237

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Reusability, Reusable Software Components, Component Retrieval, Algebraic Specifications, Rapid Prototyping, Computer Aided Prototyping

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Efforts in the software engineering community to reuse code are hampered by a lack of tools. Reusability is particularly beneficial in a rapid prototyping environment. Rapid prototyping with automated reusable software component retrieval is a software development method to rapidly construct and adapt software, validate and refine requirements, and check the consistency of proposed designs. This dissertation describes a tool used within the Computer Aided Prototyping System (CAPS), developed at the Naval Postgraduate School, which retrieves reusable components from a software base using a formal specification as the search key. The query specification that represents a design requirement is compared to formal specifications of Ada reusable software components stored in an object-oriented database management system. A syntactic search compares specification interfaces identifying reusable candidates based on types of parameters. The semantic search rank orders a set of candidate components based on semantic similarity to the query. The method, called query by consistency, compares terms that are reduced in the axioms of each specification. Specifications are normalized to facilitate the matching between query specifications and reusable component specifications in the retrieval. A formal proof verifies that query by consistency can retrieve components guaranteed to meet specified requirements.

Field	Entry
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	[X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS
21. ABSTRACT SECURITY CLASSIFICATION	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL	Luqi
22b. TELEPHONE (Include Area Code)	(408) 646-2735
22c. OFFICE SYMBOL	CS/Lq

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

Approved for public release; distribution is unlimited

REUSABLE SOFTWARE COMPONENT RETRIEVAL VIA NORMALIZED ALGEBRAIC SPECIFICATIONS

by

Robert Allen Steigerwald
Captain, United States Air Force
B.S., United States Air Force Academy, 1981
M.S., University of Illinois, 1985
M.B.A., Rennselaer Polytechnic Institute, 1986

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE
from the

NAVAL POSTGRADUATE SCHOOL
December 1991

ABSTRACT

Efforts in the software engineering community to reuse code are hampered by a lack of tools. Reusability is particularly beneficial in a rapid prototyping environment. Rapid prototyping with automated reusable software component retrieval is a software development method to rapidly construct and adapt software, validate and refine requirements, and check the consistency of proposed designs. This dissertation describes a tool used within the Computer Aided Prototyping System (CAPS), developed at the Naval Postgraduate School, which retrieves reusable components from a software base using a formal specification as the search key. The query specification that represents a design requirement is compared to formal specifications of Ada reusable software components stored in an object-oriented database management system. A syntactic search compares specification interfaces, identifying reusable candidates based on types of parameters. The semantic search rank orders a set of candidate components based on semantic similarity to the query. The method, called *query by consistency*, compares terms that are reduced in the axioms of each specification. Specifications are normalized to facilitate the matching between query specifications and reusable component specifications in the retrieval. A formal proof verifies that query by consistency can retrieve components guaranteed to meet specified requirements.

3... c.1

TABLE OF CONTENTS

I. INTRODUCTION 1
A. THE NEED FOR A RETRIEVAL MECHANISM 1
B. CONTRIBUTION 2
C. ORGANIZATION OF CHAPTERS 2
II. TECHNICAL BACKGROUND AND PREVIOUS RESEARCH 3
A. INTRODUCTION 3
B. REUSABLE SOFTWARE COMPONENTS 3
1. Definition 3
2. Advantages of Code Reuse 3
C. INFORMATION RETRIEVAL 4
1. Representation and Search 4
2. Measures of Performance 5
D. APPROACHES TO RETRIEVING REUSABLE COMPONENTS 6
1. Browsers 6
2. Informal Specifications 7
a. Keyword Search 7
b. Multi-attribute Search 8
c. Natural Language Interfaces 8
3. Formal Specifications 8
a. Types of Formal Specifications 8
b. Advantages and Disadvantages 9
E. SYSTEMS AND TOOLS SUPPORTING CODE REUSE 10
1. Draco 10
2. RAPID 11
3. Proto 12
4. The Reusable Software Library 12
5. ROPE 13
6. The Programmer's Apprentice 13
7. Common Ada Missile Packages (CAMP) 14
8. Object-Oriented Systems 16
9. Operation Support System 16

10. ARCS/Eli 17
11. Specifications as Search Keys 17
F. SUMMARY 18
III. A MODEL FOR REUSABLE COMPONENT RETRIEVAL 20
A. INTRODUCTION 20
B. SYSTEM OVERVIEW 20
1. The Computer Aided Prototyping System 20
a. User Interface 22
b. Execution Support System 23
c. Software Database 23
2. The Prototype System Description Language 24
3. OBJ3 26
a. Signature 26
b. Axioms 27
c. Parameterized Modules 28
d. Importing Modules 30
e. Why OBJ3? 30
f. Why not Predicate Logic? 31
4. Component Retrieval Subsystem 31
a. Formal Specifications for Component Retrieval 31
b. The Role of Normalization 32
c. System Structure 33
C. INITIAL ASSUMPTIONS AND MODELS 34
1. Initial Assumptions 34
2. Semantic Normalization 35
a. Example 36
b. Issues 38
3. Matching via Theorem Proving 39
a. Example 41
b. Issues 42
4. Modified Assumptions 42
D. SCHEMA FOR REUSABLE COMPONENT RETRIEVAL 42
1. Syntactic Normalization and Matching 43
2. Semantic Normalization and Matching 45

E. SUMMARY 45
IV. COMPARING SPECIFICATION SEMANTICS 46
A. INTRODUCTION 46
B. BACKGROUND 46
C. NORMALIZATION 47
1. Expansion and Instantiation 47
2. Interface Normalization 49
D. MAPPING QUERIES TO STORED COMPONENTS 52
1. Prolog as the Mapping Tool 52
2. Checking Generic Consistency 54
E. GENRATING A TEST SET 55
F. BUILDING THE INPUT TERMS OF THE I/O LIST 57
1. Initial Template and Expansion 57
2. Checking for Circularities 58
G. GENERATING OUTPUT TERMS IN THE QUERY 59
1. Reductions in the Query Domain 59
2. Parsing the Results 60
H. OUTPUTS IN THE CANDIDATE COMPONENT DOMAIN 60
1. I/O List Transformation 60
2. Inductionless Induction 61
3. Interpreting the Results 62
I. VERIFICATION OF THE MODEL FOR RETRIEVAL 62
1. The Specification Model 64
2. Normalization 64
3. The Export Signature 64
4. Mapping a Query to a Stored Component 65
5. The Test Set 67
6. The I/O List 68
7. Reduction in the Query Domain 68
8. Mapping Terms 69
9. Reduction in the Component Domain 70
10. Comparing Terms and Scoring 70
a. Comparing Terms 70
b. Scoring 72

J. SUMMARY 72
V. IMPLEMENTATION AND EXAMPLES 74
A. INTRODUCTION 74
B. IMPLEMENTATION LANGUAGES 74
C. NORMALIZATION 74
D. MATCHING 76
1. Normalize Query 77
2. Build Test Set 77
3. Make IO List 77
4. Generate Output Terms 78
5. Match 79
a. Find Maps 79
b. Test Maps 80
E. ABSTRACT DATA TYPES AND DATA STRUCTURES 81
1. Abstract Data Types 81
2. Data Structures 82
F. MATCHING EXAMPLES 86
1. List Matching Example 86
2. Set Matching Example 89
3. Stack Matching Example 94
G. SUMMARY 102
VI. EVALUATION OF THE SOFTWARE RETRIEVAL MODEL 103
A. INTRODUCTION 103
B. A FRAMEWORK FOR SOFTWARE RETRIEVAL SYSTEM EVALUATION 103
C. RECALL AND PRECISION 105
1. Background 105
a. Indexing 106
b. Relevance 106
2. Syntax and Semantics 107
a. Recall is Linked to Syntactic Search 107
b. Precision Requires a New Method 108
D. EFFORT 110
E. TIME 111

F. PRESENTATION 111
G. COVERAGE 112
H. QBC LIMITATIONS 112
I. SUMMARY 113
VII. SUMMARY, CONCLUSIONS, AND SUGGESTIONS FOR FUTURE RESEARCH 114
A. INTRODUCTION 114
B. DISSERTATION SUMMARY 114
C. SYSTEM MODIFICATIONS TO ENHANCE PERFORMANCE 115
1. Operator Overloading 115
2. Adding Predefined Objects 116
3. Syntax Checking 116
4. Subsort Matching in Prolog 117
5. Improving Efficiency 117
6. Increasing the Number of Allowable Maps 118
7. Improving Retrieval Precision 118
D. SYSTEM EXTENSIONS AND SUGGESTIONS FOR FUTURE RESEARCH 119
1. Knuth-Bendix Completion 119
2. Theorem Proving with Axioms 119
3. Mixfix Syntax 119
4. Generalization Per Category - An Alternative Phase 120
5. Mapping Specifications using Ada 121
6. Term Rewriting in Ada 121
7. Query by Example 121
E. CONCLUDING REMARKS 122
LIST OF REFERENCES 124
APPENDIX 131
I. SOURCE CODE 131
A. ADA SOURCE CODE FOR NORMALIZATION 131
B. ADA SOURCE CODE FOR MATCHING 143
1. Normalize Query 144
2. Build Test Set 154
3. Make IO List 161

4. Generate Output Terms 169
5. Match 172
6. Support Code 194
C. INPUT SOURCE FOR ANALYZERS AND PARSERS 198
1. OBJ3 Lexical Analysis 198
2. Predefined Term Lexical Analysis 201
3. Prolog Output Lexical Analysis 203
4. Term Lexical Analysis 204
5. Term Parser 205
D. PROLOG SOURCE CODE 212
E. LISP SOURCE CODE 213
E. OBJ3 PREDEFINED OBJECTS 216
G. SUPPORT FILES 221
INITIAL DISTRIBUTION LIST 225

ACKNOWLEDGEMENTS

This PhD dissertation marks the culmination of several years of hard work and dedication to a cause. It would not have been possible without the support and cooperation of my entire PhD committee. I thank Dr. Luqi, my dissertation supervisor, for her guidance both technically and as a professional software engineer. From her I have learned a great deal more than this document reflects. I also thank Dr. Robert McGhee, my committee chairman, whose insistence on regular meetings and attention to detail helped me work through problem areas and made the final hurdles easier to overcome. I thank Dr. Carl Jones for his contributions in the area of system performance tradeoffs, which helped me to set my research into a broader perspective. I thank Dr. Timothy Shimeall for helping me focus my research and for helping me find gaps in my approach. His consistent, positive feedback and encouragement were an extra source of motivation. I thank Dr. Michael Nelson, Maj. USAF, for introducing me to object-oriented programming methodologies and for helping me to tie it into my research. Also, his "blue-suit" orientation put us on common ground for discussing academic/military career progression. Finally, I thank Dr. Tarek Abdel-Hamid for pursuing the topic of system effectiveness early in the process. His questions led ultimately to the formal verification of the method described herein.

Although not members of my committee, Dr. Mantak Shing and Dr. Valdis Berzins also supported my efforts. Dr. Shing, always willing to take the time to help students, helped me formulate an initial verification model. Dr. Berzins' expertise in algebraic specification languages and software engineering helped me to discover what was feasible and what was not. I am also grateful for his assistance in the formal verification of the query by consistency model.

My final acknowledgement goes to my wife Silvi. She is and always will be my true source of motivation and encouragement. At each turn she made the sacrifices and compromises that gave me the leverage needed to complete this degree program. While many marriages suffer because of the stress associated with attaining a PhD, ours has grown stronger due largely to her efforts. To her I owe a debt of gratitude that I shall be repaying for as long as I have a PhD, which, by my reckoning, is a very long time.

I. INTRODUCTION

Efforts in the software engineering community to reuse code are hampered by a lack of tools. Some of the major issues that make software reuse difficult are component classification, retrieval, composition and library maintenance. Research in these areas is needed to attain the potential increases in productivity, quality, and reliability. This dissertation focuses on computer-aided retrieval of reusable software components.

A. THE NEED FOR A RETRIEVAL MECHANISM

The purpose of this research is to enhance the practice of software reuse by providing a means to retrieve reusable software components from a library, or *software base*, by matching a user's query, a formal specification, to the specifications of stored software components. The tool described herein will become part of a rapid prototyping system whose aim is to provide automated mechanisms to create software prototypes of complex real-time systems. An integral part of the prototyping system is the software base, a large collection of reusable components. The software base will provide prototype designers with the means to quickly locate components and integrate them into new applications.

The key to locating components in this system is a powerful retrieval mechanism that uses the syntax and semantics of the prototype language description of each object. This method contrasts with another popular method used today, that of classification schemes. The classification scheme approach attempts to store and retrieve components based on attributes whose values are selected from a finite set of keywords. Retrieving components in this type of system requires some knowledge of the structure of the software base and knowledge of the keyword set.

Query by formal specification requires that the user be able to express the query as a formal specification. With respect to the focus of this dissertation, this is not a drawback since it is assumed that the prototyping system is based on a prototyping method that uses formal specifications to develop and document the components that make up the prototype. That is, the user must write formal specifications anyway, so the retrieval system takes advantage of this fact and uses them for retrieval. Because the retrieval mechanism relies solely on the specification, the user is not required to know anything about the structure of the software base or any list of attributes or keywords.

B. CONTRIBUTION

This dissertation describes an automated mechanism to retrieve reusable software components from a software base using a formal specification language. The formal specification for each component describes its interface (syntactic description) and its behavior (semantic description). Both the syntax and the semantics of a query specification are used to identify candidate components in a software library that will satisfy the given specification. This dissertation emphasizes the use of the semantic description of a component for retrieval. The specific contribution of this dissertation is the development and implementation of automatic techniques to retrieve components using the syntax and semantics of formal specifications.

C. ORGANIZATION OF CHAPTERS

Chapter II reviews the basic concepts and terms relevant to this and previous research. The chapter summarizes past approaches to reusable component retrieval and closely related problems, emphasizing strengths and weaknesses. Chapter III describes the model of a system for reusable component retrieval, reviews initial assumptions, and explores different alternatives to implement the model. Chapter IV focuses on the task of comparing specification semantics, introduces *query by consistency*, and verifies the correctness of the the process. Chapter V describes tests performed on the implemented retrieval tool and Chapter VI evaluates the effectiveness of the retrieval tool. Chapter VII summarizes the dissertation and suggests extensions to this research.

II. TECHNICAL BACKGROUND AND PREVIOUS RESEARCH

A. INTRODUCTION

This chapter describes some technical background concerning reusable software components and their retrieval, and reviews previous and current systems that try to solve the reusable component retrieval problem.

The next section defines reusable software components and lists the advantages and disadvantages of using them. Section C abstracts the component retrieval problem to an information retrieval problem and describes the concepts of representation, search, and measures of performance. Section D reviews the better known approaches used to retrieve components and Section E describes some of the actual systems that have implemented the approaches.

B. REUSABLE SOFTWARE COMPONENTS

1. Definition

"Software reuse is the reapplication of a variety of kinds of knowledge about one system to another similar system in order to reduce the effort of development and maintenance of that other system." [BP89a. p. xv] Reuse extends across a wide range of products, including documentation, analyses, domain knowledge, designs, and source code. This is the broad view of reusability. A narrower view is *code* reuse, that is, the reuse of actual source code modules. The focus of this dissertation is on the reuse of source code modules.

2. Advantages of Code Reuse

The concept of code reuse is not new. It has been prevalent as long as people have been programming. There already exist large program and subroutine libraries that implement well-known algorithms in many problem domains [Stei86]. The primary benefit of using a previously written module rather than writing your own is that you expect to increase both productivity and quality. As Standish put it, "Software reuse has the same advantage as theft over honest toil." [Stan84, p. 494] There are a few examples of success in code reuse today [Prie91a, Stei86, Booc87] but reuse of modules is not as widespread as one might expect due to the technological, managerial, and organizational issues that still need to be resolved.

To make code reuse a success, several problems must be resolved including software classification, retrieval, adaptation, composition, and library maintenance. Software classification is the problem of categorizing the component so that it may be stored in a repository. The class in which the component is placed must lend itself to straightforward retrieval, the second problem. Having found the component, there is the task of adapting it to suit one's needs and then finding a way to integrate it with the other components of your system (adaptation and composition). Finally, there is the problem of maintaining the collection of components and the tools for classifying, storing and retrieving them.

It is fashionable today to talk of code reuse and of large component libraries, but the promises of increased productivity and quality remain elusive because the above issues have not been resolved. Recently, the focus of the research in this area has been on component classification and retrieval.

C. INFORMATION RETRIEVAL

The problem of retrieving reusable software components from a library is in general an information retrieval problem. The research in the area of information retrieval is extensive, most of it dedicated to keyword search and string matching algorithms in document retrieval applications [SM83]. The important concepts from information retrieval that relate directly to reusable software component retrieval are: 1) representation, 2) search and 3) measures of performance.

1. Representation and Search

A general information retrieval tool has two parts. The first is the method of representation, that is, the way the object sought is structured to facilitate retrieval. For instance, a document may be scanned to garner a list of important keywords for the basis of its future retrieval or a person may have to examine a finite list of keywords to select the ones that closely relate to the document. The method of representation must necessarily support the method used to search for the object.

The second important part of an information retrieval tool is the method of search. Considerable research in computer science has been dedicated to search mechanisms, most notably in database management systems and artificial intelligence.

The method of representation and the method of search work together to form a cohesive environment for information retrieval, hence there is a tradeoff in the amount of sophistication one applies to either part. The more refined and precise the method of representation, the easier the search mechanism becomes. For instance, if everything one

must store has a unique key that can be computed and translated to a physical address, the search for that object is trivial. On the other hand, if little effort is applied to representation, the search for an object will be more complex.

Representation and search methods applied to reusable software component retrieval are discussed in Section II.D.

2. Measures of Performance

How well an information retrieval system performs is based on the nature of the objects returned for a given query. The two most important measures of performance are *precision* and *recall* [SM83]. Given R as a set of *relevant* components in the database for a query and Q as the set of components *returned* for the query, precision is defined as

$$|Q \cap R| : |Q|$$

or the ratio between the number of relevant components retrieved and the total number retrieved [RW90c]. Precision asks the question, "What percentage of the components in Q are relevant?".

Recall is defined as:

$$|Q \cap R| : |R|$$

the ratio between the number of relevant components retrieved and the number of relevant components in the database. Recall asks the question, "What percentage of the relevant components in the database did my query find?".

Precision and recall obtain ideal values when Q = R, that is, when the set of components retrieved is exactly the same as the set of components that are relevant. In that case, both ratios will have a value of one. Not surprisingly, there is a tradeoff between precision and recall. For example, if a query returned every component in the database (N components), recall would be one, but precision would be $|R|/N$, which is poor when N is large and R is small. At the other extreme, suppose the query yielded one relevant component. In this case, the precision achieves a value of one, but the recall is $1/|R|$, which is poor if R is large.

There is a caveat associated with these measures of performance since *relevance* is a subjective term. It is up to the individuals performing the tests to decide which components are relevant and which are not. This will definitely have an impact on the values given for precision and recall. Despite this apparent misgiving, these measures of performance are used among others (such as effort, time, presentation, and coverage) to assess the performance of the component retrieval systems described in Section II.E as well as in our system described in Chapters IV and V.

D. APPROACHES TO RETRIEVING REUSABLE COMPONENTS

As the interest in reusable software components has grown, the demand for tools that aid in retrieving, classifying, storing, and retrieving components has increased. We are particularly interested in and focus on those tools that offer mechanisms for component retrieval. Almost all of the tools we have encountered in the literature use one (or more) of three different approaches for retrieval; browsers, informal specifications, or formal specifications. Since many of the systems use more than one of these approaches, we review the fundamentals of each approach in this section and then describe the features particular to each tool in Section II.E.

1. Browsers

A *browser* is a general purpose, usually window-based tool for looking through collections, categories, or hierarchies of components at various levels of abstraction [Meye88b]. The interface can range from purely textual to sophisticated graphics. In any case, the objective is to allow the system user to manually search for the desired component.

The notion of a browser comes from the information retrieval domain, but its first use with respect to component retrieval was in object-oriented programming systems. In an object-oriented system, reusability is inherent because all new objects are defined in terms of other objects already defined in an object hierarchy. It would be nearly impossible to manage this type of programming environment without some method to scan the hierarchy of components to find a suitable "jumping off point". Thus we see sophisticated graphical browsers for object-oriented systems like Smalltalk-80 [Gold84], the Knowledge Engineering Environment (KEE) [Inte88], and Eiffel [Meye88a, Meye88b].

The advantages of a browser are that it gives the user free reign over the entire collection of components, and in object-oriented programming systems allows the user to see which objects depend on other objects.

There are, however, several disadvantages to the browser approach. The first is that the method is basically manual, relying on significant user knowledge of the structure of the component collection. Second, the focus of search is local, meaning that a semantically similar component defined elsewhere in the system will not be found at all unless the user knows to look there also. Third (and this is related to the second point), unless the user has found exactly the component needed, they will not know when to stop looking. Fourth, unless the component contains some accompanying documentation, the user is forced to read the source code to determine if the component meets his needs. A final point relates to the size of the software base. A browser is "...well suited where classes are

contributed by a small number of people, and the total number of classes does not exceed a few tens or perhaps a few hundreds. For large-scale reusability, it is no longer sufficient." [Meye88a, pp. 445-446] In other words, as the number of components in the software base increases, the value of a browser decreases .

Many of the systems that offer browsers, such as those discussed in Section II.E, use other techniques such as keyword or multi-attribute search to help mitigate some of these disadvantages.

2. Informal Specifications

Retrieval techniques based on informal specifications require the user to describe or list some of the attributes of the component sought. Informal specification methods include keyword search, multi-attribute search, and natural language interfaces.

a. Keyword Search

Keyword search mechanisms require the user to specify a list of words relevant to the object being sought. For example, if a user were searching for a component that implemented a stack, he would use the keyword *stack* to perform the search. Keywords can be drawn from a known system vocabulary (controlled vocabulary), or they can be unconstrained (uncontrolled vocabulary). In the case of unconstrained keywords, synonym tables are often used to find more standard words on which to perform the query [SM83].

One problem with using keywords is that the number and choice of words is crucial to success. Using a single keyword will often result in high recall but low precision, whereas too many keywords will have the opposite effect. The search for a component, then, becomes an exercise in trial and error, with the user performing multiple searches until an appropriate object is found. It often takes an experienced user to achieve the desired results. Thus, the fundamental disadvantage to using keywords lies in their limited expressive power both individually and in combinations [MCT87].

The advantages of a keyword approach are easy implementation and its conceptual simplicity for the user. Most document retrieval systems are keyword based and many of the software component retrieval mechanisms described in Section II.E have keyword search mechanisms.

b. Multi-attribute Search

Multi-attribute search mechanisms [Prie85, BLW90] use keywords, but also rely on other characteristics of the object being sought to be used as search keys. In the area of component retrieval, characteristics of components that can be used for retrieval are the class of the object (procedure, function, package, etc.), the number and types of parameters, the number of operations it supports, its domain of use, etc.

An advantage to a multi-attribute search is that a component description contains more than just keyword information. The attributes taken together make up a classification scheme that provides more information than would be present in a pure keyword search.

A disadvantage to a multi-attribute search is that the classification and subsequent storage location of a component defined by its attributes is left to the author and/or the library administrator, but different people will not necessarily classify the same component in the same way. If the user succeeds in filling in the same values, the search mechanism will be very precise, but unless some sort of partial matching function is used, recall of similar components will suffer.

c. Natural Language Interfaces

Historically, research in information retrieval has focused on textual document retrieval. It seems fitting to use natural language queries to retrieve natural language data. The distinct advantage offered by this method lies in the ease of language query formulations by system users. In addition, the same techniques may be applied to derive content information from documents destined for storage. [SM83]

Language processing may be performed at various levels from phonological to semantic and pragmatic. In reusable component retrieval, the *higher* levels of language processing need to be applied. Of course these are the most difficult. The main challenge lies in dealing with the ambiguity inherent in the broad semantics of natural language.

Natural language query systems for information retrieval have been built within constrained domains or by using restricted languages [RG91, Kolo83], but a general purpose tool remains elusive.

3. Formal Specifications

a. Types of Formal Specifications

Many types of formal specification languages have been used to describe the semantics of software processes. Factors that contribute to their use as a means for component retrieval in the context of this research include 1) a syntax or structure that is consistent with the structure of the underlying implementation language (Ada) [Ada83], 2)

a means to execute the specification, and 3) a facility for specifying generic components. Three candidate specification formalisms are discussed here: predicate calculus [RW90c], plan calculus [RW90a], and algebraic formalisms [GTW78, Wirs88]. The reason usually cited for using formal specification languages is to achieve precise communication and a high degree of automation throughout the software lifecycle [BL91]. Using them for component retrieval is a natural extension to their original usage.

Predicate calculus is a specification language with a rigorous mathematical foundation. It is an *executable* specification language as well if you consider logic programming languages such as Prolog [CM84, Rowe88]. One system that makes use of predicate calculus as a basis for component retrieval is described in Section II.E.11 [RW90c].

The Plan Calculus is a formalism developed for a system called the Programmer's Apprentice [RW90a] (see Section II.E.6). It combines the "...representation properties of flowcharts, dataflow schemas, and abstract data types" [RW88, p. 12] to depict modules as a hierarchical graph structure. We mention it here not because it is widely used, but because it is a formal method particularly well suited for comparing program fragments (a form of reuse) in the Programmer's Apprentice environment (see Section II.E.6).

The theory of algebraic specifications is based on the notions of classical algebra in mathematics and on the concepts of abstract data types in computer science [EM85]. It has its origins in the mid 1970's and has been realized in many forms such as Clear [BG80], LARCH [GHW85], and OBJ3 [GW88]. Algebraic specifications consist of a *signature* describing the interface to an object and some *axioms* that describe the object's semantics. Algebraic specifications may be *executable* when the axioms are treated as rewrite rules. Section III.B.3 describes the structure of OBJ3, an algebraic specification language.

b. Advantages and Disadvantages

The above formalisms have all been employed as a means to retrieve reusable components. The advantages of using formal specifications are that they are free from ambiguity and they are subject to stronger forms of transformation than are other specification methods. In the case of algebraic specifications, the logic and theory of term rewriting can be exploited. With predicate calculus systems, theorem proving is a natural asset.

There are also disadvantages. Specifications may be difficult for designers to write. Additionally, processing times for the search algorithms may be excessive depending on the approach taken. Finally, matching formal specifications is a hard problem. In fact, the general word problem, which is proving the equivalence of two terms composed of variables and operators, is undecidable [KB67].

E. SYSTEMS AND TOOLS SUPPORTING CODE REUSE

This section describes systems that have been built to perform reusable component retrieval and identifies the methods used by each system. While this survey is extensive, it is certainly not exhaustive. Reusable component retrieval has become a popular research area and new ideas and projects are surfacing all the time.

1. Draco

The Draco project [Neig84], named after the constellation, is an approach to software engineering that has had a large impact on software reusability in general. Born in the early 1980's at the University of California, Irvine, the Draco approach focuses on *domain engineering* of software. The goal of the project is to increase the productivity of software engineers in the construction of similar systems by organizing reusable components by problem area or domain [Neig84]. Draco was among the first systems to promote the reuse of products from all phases of the software lifecycle, from analyses and designs to components.

The most important aspect of Draco is the *domain language*. A domain language describes objects and operations of a particular domain and hence represents analysis information about the domain. The objects and operations are also suitable for describing design information or *how* the problem is to be modeled. A given domain language is characteristic of a particular problem area. Reuse of analysis information takes place each time a new project is cast in the domain language. Reuse of designs occurs each time source code is constructed from a design possibility. Even more reusability is possible when objects and operations of one domain are mapped to those of another domain.

At the lowest level are the software components, which realize the semantics of a domain. There is a reusable component associated with each domain language object or operation. Since there is a potentially large number of components within a domain, Draco researchers have developed a classification scheme for the components called

faceted classification to aid in organizing and retrieving the components [Prie85, Prie91b, PF87].

Using faceted classification, each component is described by a set of attributes or tuple. The attributes are chosen to best characterize the components of a particular domain. Each attribute slot is filled with a value (term) from a controlled vocabulary to avoid duplicate and ambiguous descriptors. A thesaurus is provided to determine the proper term to use. A query, then, is a tuple with selected terms used as a key to search the database. In general, a query session begins with the most specific query, that is, all attributes filled in. If the results of the query are unsatisfactory, the user may *generalize* the query by inserting wildcards (*) for attribute values.

As mentioned in Section II.D.2.b, a disadvantage of a multi-attribute search such as this one is that semantically similar components may not be found when their attribute definitions are different. Draco alleviates this problem by maintaining a measure of conceptual closeness for the term lists of each attribute as a weighted, acyclic, directed graph. This way, an unsuccessful search can be tried again using an alternative but similar term in one of the attributes.

In evaluating the effectiveness of faceted classification, the Draco researchers compared their retrieval mechanism to a database retrieval system not organized by a classification scheme. Using faceted classification, the number of components retrieved for a given query was reduced by more than 50%, while the precision of the queries improved by 100%.

The advantages of faceted classification are that it is conceptually simple for users and relatively easy to implement. Because of this, the concept has been borrowed to implement the retrieval mechanisms in both RAPID [VR90] (see Section II.E.2) and OSS [Rott91] (see Section II.E.9).

There are also disadvantages to faceted classification. Classification, in general, is not suitable for unconstrained domains. Also, even with a conceptual closeness measure, semantically similar components may be missed, especially components from other domains.

2. RAPID

The RAPID (Reusable Ada Packages for Information System Development) project is an ongoing effort sponsored by the U.S. Army Information Systems Software Development Center in Washington (USAISSDCW) [Voge89]. The contractor implementing the system is SofTech Inc. The objective of RAPID is to provide software

engineers with quick access to reusable Ada packages in the information systems domain. The functions it performs are reusable software component classification, storage, and retrieval.

RAPID uses a faceted classification scheme to organize and retrieve components (see Section II.E.1) and falls into the category of multi-attribute search [VR90]. The Naval Weapons Center is currently serving as a beta test site for the RAPID product, but no measures of performance or quality assessments are available yet.

3. Proto

Proto is a rapid prototyping system developed by International Software Systems, Inc. (ISSI) under contract for the Air Force's Rome Lab (formerly Rome Air Development Center - RADC) [Burn90]. Using Proto, a software engineer may describe the activities of a system with functional specifications, search for components to model the specifications, and execute the prototype. The development environment is based on a graphical model in which an engineer develops functional specifications with data flow diagrams. As a prototype system is defined, the engineer searches for components to serve as implementations for each specification. The engineer may then execute the prototype.

Keywords are the basis for the component search mechanism [Burn90]. Since the system is still under development, the researchers have made no measures of performance.

4. The Reusable Software Library

The Reusable Software Library (RSL) is a system designed to make software reuse an integral part of the software development process [BW87]. Developed in-house and for use at Intermetrics, the system couples a passive software database with interactive software design tools to help software developers find and evaluate components to meet their requirements.

Components are stored in the database with attribute values that provide a basis for search. There are two methods available to search for components, standard multi-attribute search and natural language. The multi-attribute approach provides a menu driven interface in which the user selects the attributes with which to perform the search. The designers' report [BW87] does not state whether the vocabulary for the attributes is controlled or uncontrolled and does not give any performance measures.

Alternatively, the user may express his query in the form of natural language, such as "I need a stack package." The system parses the input, extracts keywords from it and uses those words as attributes to perform the search. The designers report that the

natural language front end is considerably easier to use but the search is significantly slower, by a factor of five to ten because of the natural language parsing overhead involved.

Another component of RSL is a subsystem called Score [BW87] which attempts to rank order the retrieved components based on user specified preferences. In a Score session, the user must give values for object and subjective metrics such as line count, complexity, readability, structure, style, documentation and testing. Score presents the user with graphical "barometers" to rate the relative importance of the metrics. While the Score subsystem is particularly important for evaluating reusable component alternatives, the designers gave no performance results in their report.

5. ROPE

The Reusability Oriented Parallel programming Environment (ROPE) is a software reuse system developed at the University of Texas, Austin, as part of a system called the Computation-Oriented Display Environment (CODE) [BLW90]. The purpose of CODE is to aid software engineers in constructing parallel programs using a declarative and hierarchical graph model of computation. The purpose of ROPE is to support CODE by giving engineers the ability to find and understand reusable software components [BLW90].

Component storage and retrieval is based on a new technique called the structured relational classification method. This method apparently offers the browsing capabilities of a hierarchical system as well as the flexibility and ease of reorganization of a relational model. With the structured relational method, components are described using attributes in a normal relational database, but associated with each attribute domain is a graph structure relating the elements of the domain. The graphs may be lattices, linear sequences, networks, etc. Thus a group of components may be described by a relation, but the individual characteristics of components within this group are isolated via the hierarchical structure of the attributes. This assumes the user has some knowledge of the structure of a particular attribute and how to specify a structured value.

The designers claim, based on studies performed with student programmers, that the subjects had high rates of reuse, 68% precision for component retrieval, and increases in both productivity and quality [BLW90].

6. The Programmer's Apprentice

The goal of the Programmer's Apprentice project is to apply artificial intelligence techniques in an effort to automate the programming process [RW88]. It is

designed to provide intelligent assistance in all phases of a programming task. The designers think of the Apprentice as a new agent in the process rather than as a tool.

A reusable component in the Programmer's Apprentice is called a *cliché*. A cliché represents a commonly used combination of elements. Examples are abstract data types, binary searches, and list enumerations. When programming, a software engineer tends to think in terms of clichés rather than reasoning from first principles. Thus, programs may be considered as collections of interrelated clichés.

A formalism called the Plan Calculus has been developed to represent clichés [RW89]. A plan defines a single cliché in three parts: a plan diagram, a logical annotation, and an overlay. Plan diagrams are hierarchical data flow schemas that represent computations, control flow, and data flow. Logical annotations are predicate calculus assertions that describe the nonalgorithmic aspects of a plan. Overlays are transformations or mappings between plans. Together these parts constitute a language independent formalism for describing reusable software components.

The Programmer's Apprentice researchers do not emphasize reusable component retrieval per se, but rather see *automated cliché recognition* as a means to understand existing programs and facilitate program optimization [RW90a]. They have devised a method to recognize clichés in programs using graph parsing in order to recognize a program's design [RW90b]. A maintenance tool called the Recognizer automatically finds all occurrences of a given set of clichés in a program and builds a hierarchical description of the program in terms of the clichés found. Since a plan is essentially a directed graph, the system uses graph-parsing to identify sub-graphs that are then replaced with more abstract operations.

At this point it is not clear whether the Recognizer will ever be used as a general purpose component retrieval tool. It is currently limited to finding algorithmic clichés but the researchers hope to extend its capability to find data structures and data abstractions as well. A limiting factor of their method is the inefficiency of the exhaustive, purely structural approach used in sub-graph parsing. The researchers acknowledge this and plan to add heuristics based on a program's documentation to focus the search.

7. Common Ada Missile Packages (CAMP)

The Common Ada Missile Packages (CAMP) project is a Department of Defense sponsored effort to create a software engineering system and reusable software library of components [CAMP89, Ande88]. The application is software for missiles and the stored

source code is Ada. One of the main components of the system is the Parts Engineering System (PES) Catalog.

The designers of the PES catalog liken it to a library card catalog for books [CAMP89]. The catalog system, used by both software engineers and domain engineers, is written in Ada and provides a menu driven interface for storing, modifying and retrieving components (*parts*). Each part has an attribute list associated with it, thus attributes are the basis for retrieval.

Searches for parts are based on a single attribute whose value must be selected from a finite list of values. The result is a "search-list". A search-list is obtained by searching either the entire database or another search-list. Multi-attribute search is based on *and* and *or* combinations of attributes. It may be simulated by combining the results of single attribute searches, that is combining search-lists. Examples of attributes are keyword, part ID, part number, part name, classification, developer name, developer project, etc.

Since there are a finite number of possible values for each attribute, "canned" searches are also provided by the system to increase performance. What this means is that the system has already created an index into the database for all components with, for example, keyword "navigation" or type "bundle". Whenever a component is added to the database, these indexes must be updated. These canned searches are only useful when the search is performed on all components in the data base, not on a subset of the components.

The CAMP documentation did not assess the performance of the PES catalog. No measures were given for precision and recall, but from the search method used, it is easy to see that measures of precision and recall are not meaningful from just search results. This is because a component's supposed relevance is *predetermined* by the value given to one of its attributes. Hence, a search for all components whose keyword attribute is filled with the value "navigation" will return all components in the current search-list with that value. This might lead one to believe that precision and recall values are one for this method. Unfortunately, the question of relevance is not simply a matter of having the right value for an attribute. Relevance depends on how well the requirements of a particular design can be met by the candidate component. Simply having the requested attribute value does not guarantee relevance. Therefore, we must question the accuracy with which attribute values are assigned to components. Since the possible values for each attribute is finite, the same limitation that besets keyword search mechanisms is present here, that is, the choice of descriptors may be close, but not quite right. A more appropriate method for

determining relevance is a subjective look at the retrieval based on how well attribute values describe the actual component and the extent to which other components with different attribute values are relevant.

8. Object-Oriented Systems

Object-oriented design is a software decomposition technique that has become popular since it is a natural way of mapping a problem to a solution [Booc86]. Object-oriented systems support object-oriented design by allowing the programmer to define a hierarchy of interrelated objects. A key feature of object-oriented systems is *inheritance*. This feature makes object-oriented systems particularly "reusable" because new applications are readily defined on the basis of previously defined applications and an object's properties may be shared by many different kinds of sub-objects. In systems such as Smalltalk [Gold84], Eiffel [Meye88a, Meye88b], and KEE [Inte88], a library of components is at your fingertips, ready to be exploited. Unfortunately, in the author's opinion, finding the right component to use in an object-oriented system is not easier just because the system is object-oriented, at least for programming in the large.

The discussion on browsers in Section II.D.1 sums up the problem with finding components in these object-oriented systems; the search technique is manual and familiarity with the structure of the object base is required. If a designer finds an object with half of the methods he needs, how does he know whether or not to stop searching?

Of course, object-oriented systems are not limited to browsers. Other methods can be integrated with a browser to provide multiple search mechanisms. Unfortunately, because research on retrieving components in object-oriented systems is still in the early stages, we have found no experimental results in the literature. Good discussions of reusability in object-oriented systems can be found in Biggerstaff and Perlis' book on software reusability [BP89b].

9. Operation Support System

The Operation Support System (OSS) is an in-house effort undertaken by the Naval Ocean Systems Center to develop an integrated software engineering environment [Rott91]. One goal of the project is to establish a Navy software library of reusable software components [Rott91]. The current prototype library subsystem allows component retrieval using faceted classification (see Section II.E.1 on Draco), keywords, or a textual browser. Once a component of interest is found, the user may display the structure of the component with an integrated, vendor supplied tool called Software Through Pictures [Inte90]. The components currently stored in the library are large command, control, and

communications (C^3) software subsystems. Since the library is still in its early stages, the developers do not have information on its performance characteristics.

While the OSS library subsystem is not yet integrated with the software development environment, their goal is to eventually integrate it to foster reuse throughout the lifecycle. To increase the extent of code reuse, the developers have also proposed efforts to perform domain analysis of the C3 discipline to determine what components are common to the systems. Thus, it is their aim to design components with reusability as a goal rather than an afterthought [Rott91].

10. ARCS/Eli

The Automated Reusable Software Toolset (ARCS), also known as Eli (for Eli Whitney) is a reuse library system and set of cooperating tools under development by Software Productivity Solutions [SPS91]. The purpose of the system is to support software development centered on reusable software assets.

The ARCS developers believe that effective information retrieval requires classification flexibility. According to its product description, ARCS uses a combination of techniques for software asset classification and retrieval including faceted classification, keyword indexing, text indexing, characteristics-based attributes, metrics criteria, taxonomies, component relationships, and a browser. Using this broad range of classification schemes, it would seem that the overhead for the variety of search mechanisms and cross referencing would be somewhat taxing.

Detailed information on this system is not available, since it is proprietary. A beta release of the system is planned for late 1991. Hence, there are no measures of performance available.

11. Specifications as Search Keys

An experimental system developed at Carnegie Mellon University uses formal specifications to search software libraries [RW90c]. Their system allows a user to search a library containing functions for a particular function. Each function in the library has a corresponding formal specification. Specification matching is the process of determining whether a specification s for a library function satisfies a query q. Specifications and queries are written in λProlog. Each specification has a signature and some semantic information. Their aim is to match first on signature and then increase precision by matching on specification semantics.

Signature matching checks that the types in the signature of the query match those of the stored functions. The matching algorithm allows matching on signatures with minor structural differences such as flipped operators or "curried" [MacL90] arguments.

As each candidate is found by signature matching, the system performs semantic matching. Specification semantics are defined using pre-conditions and post-conditions. For each function there is a predicate that defines the function's pre-condition and another predicate that defines its corresponding post-condition. In the process of matching, a query pre-condition is satisfied if the query pre-condition implies the pre-condition of the function. Likewise, a query post-condition is satisfied if the function post-condition implies the query post-condition. Since standard Prolog unification and backtracking is used as the search method, a list of candidates may be obtained by forcing the system to backtrack and search for other alternatives.

The system designers claim, as we do in this dissertation, that the use of semantics in specification matching increases precision. They show in their report using examples that precision is improved but they do not give any general statistics that indicate how much. The designers feel that using λProlog offers the distinct advantage of higher-order logic for matching but admit that the lack of equational reasoning limits the capabilities of the system.

F. SUMMARY

This chapter introduced the concept of reusable software components, reviewed the fundamentals of information retrieval and the methods available for retrieving components, and identified a number of systems that use these methods to retrieve reusable software components. An overriding characteristic of all of the systems is the lack of any measures of success. Reports on some of the systems mention the need for improvements in precision and recall, but none give actual results from practice. The most likely reason for this is that there is no link between the existing software libraries and these new systems. Because each system requires a unique form for the representation of the components to be stored, each system will have to "grow" its own library of reusable components . That process will take some time. If the library can be placed into service as it expands, then performance measures can be made to determine the actual success of the retrieval mechanisms and of the concept of reusability in general.

The system described in this dissertation must bear the same burden. A formal explanation will verify the process and show how the algorithms work (see Chapter IV).

Examples will be provided to offer evidence that the implementation realizes the algorithm (see Chapter V). Unfortunately, actual results obtained by using the system in practice are not yet available.

An additional issue that the designers of most of the systems described fail to address is component *granularity*. Some of the methods described are completely independent of the size of the stored component, while for others granularity is an important factor. When using a browser or informal specifications, the size of the stored component is transparent to the user and is not an important factor in the search. Use of formal specifications, however, requires the user to write some sort of specification that models the component sought. In this case, more effort is required to write the query and more processing *may* be required to perform the search. Each individual system assumes some sort of component granularity. Systems with browsers or search mechanisms which rely on informal specifications can afford to be more flexible with regard to the size and content of the components stored. Systems using formal specifications as the basis for retrieval are not limited in any fundamental way to small components, but for practical reasons, tend to focus on small, atomic, cohesive program units.

The system described in this dissertation relies on the prototype designers to decompose the system they intend to build into modular, functionally (or informationally) cohesive program units (according to the tenets of software engineering [Fair85]) and perform the search for reusable components at that level. The system and methods do not preclude the designer from searching for a more complex object. Various users of CAPS will have alternative views about the type of objects and granularities of objects that will be stored in the software base.

III. A MODEL FOR REUSABLE COMPONENT RETRIEVAL

A. INTRODUCTION

The reusable component retrieval tool which is the subject of this dissertation is a part of a much larger system under development at the Naval Postgraduate School which is designed for computer aided rapid prototyping. This chapter begins by describing the Computer Aided Prototyping System (CAPS) [LK88, Luqi91] and its specification language PSDL (Prototype System Description Language) [LB88, LBY88]. We then narrow the focus and abstractly describe the component retrieval subsystem, how it fits within CAPS, and some of the characteristics of formal specifications in their role as search keys. This will give a broad overview of the system and enough information about the retrieval subsystem to understand the explanation of the initial assumptions and models in Section III.C.

We include a section describing the initial assumptions and models because the path taken from the initial understanding of the problem to the eventual solution was not direct. There are valuable lessons to be learned by knowing what approaches were evaluated and why certain paths were not taken. The section on initial assumptions and models describes two hypothetical approaches to reusable component retrieval: the concept of *normalization* as if it were the predominant factor in retrieval and the concept of *theorem proving* as if it were the predominant factor in retrieval. The section concludes by elaborating modified assumptions.

The last section reiterates the contents of Section III.B.4, the description of the component retrieval subsystem, this time providing more details about the role of normalization and the form of matching.

B. SYSTEM OVERVIEW

1. The Computer Aided Prototyping System

The computer aided prototyping system (CAPS) is an integrated environment aimed at rapidly prototyping hard real-time embedded systems [LK88, Luqi91]. The integrated set of software tools provided includes an execution support system, a syntax directed editor with graphics capabilities, a software base with an embedded rewrite system, and an engineering database management system with an embedded design management system. Figure 3.1 shows the high level structure of CAPS.

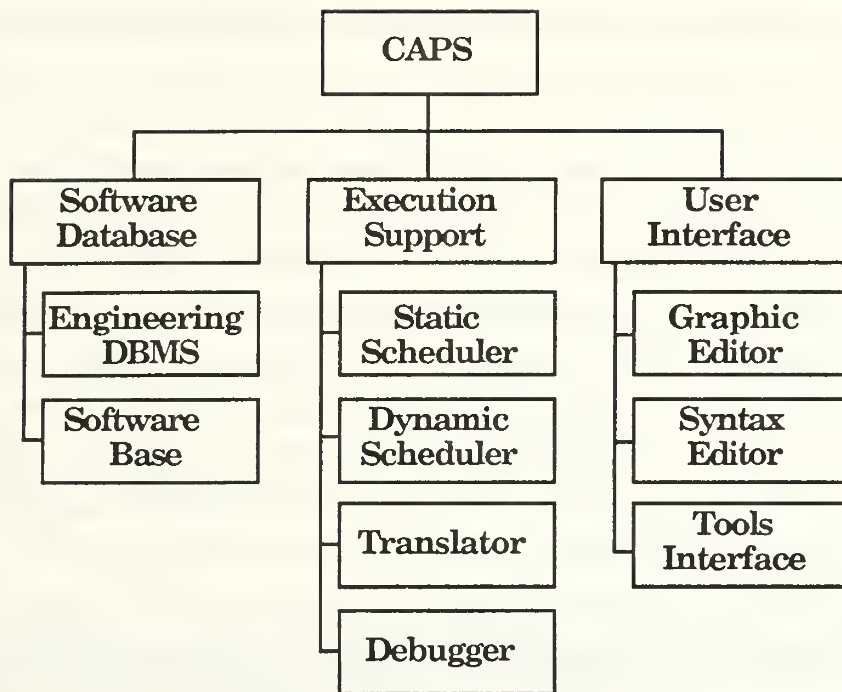


Figure 3.1 - Structure of CAPS

Embodied within the CAPS software development approach is a systematic design method for rapid prototype construction. System or subsystem descriptions are started at a problem-oriented, abstract level and iteratively refined into a hierarchically structured prototype using a uniform decomposition method that combines the advantages of data flow and control flow methodologies. At each level of the hierarchy, the designer focuses only on the details important at that level. To generate a prototype, the designer of the prototype uses the graphic editor to create a graphic representation of the proposed system. The graphic representation is used to generate part of an executable description of the proposed system, represented in a Prototype System Description Language (PSDL) [LB88, LBY88]. PSDL descriptions are used to search the software base to find reusable components that match the specifications. A transformation schema is then used to transform the PSDL specification into Ada [Ada83] code that controls and connects the retrieved reusable components. The prototype is then compiled and executed. The end user of the proposed system evaluates the prototype's behavior against the expected behavior. Successive iterations of this process should lead to a system that ultimately satisfies the user's requirements. [Cumm90]

CAPS is divided into three major subsystems. They are the user interface, the execution support system, and the software database. The following sections describe each in turn.

a. User Interface

The CAPS interface provides a cohesive software development environment integrating the tools of CAPS (see Figure 3.2). At the core of the environment is the host operating system. The windowing system, X-windows [Jone89], is the next layer. InterViews [LVC89], the toolkit chosen to develop the user interface, provides the interface between the upper layers of the environment and X-windows. The CAPS tools sit on top of InterViews and are surrounded by the tool interface. The tool interface provides all communication between the tools and the user interface. The outermost layer of the environment is the user interface. This layer hides the underlying implementation details from the designer. [Cumm90]

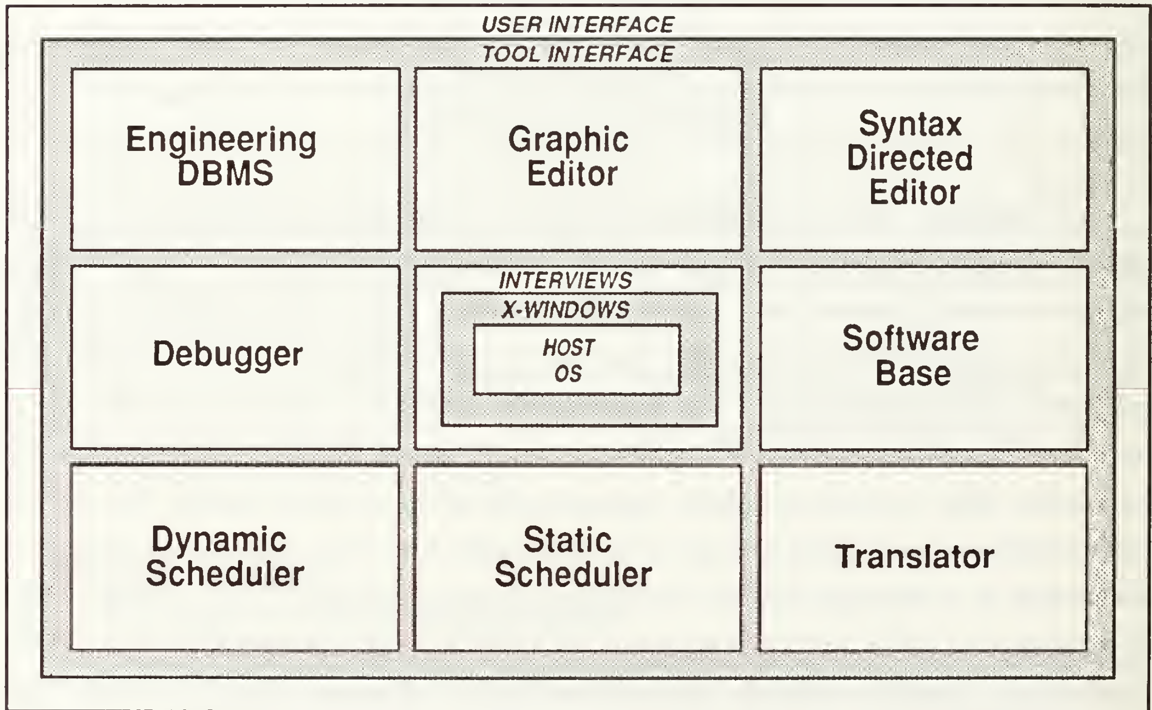


Figure 3.2 - CAPS Tools and Interfaces

b. *Execution Support System*

The execution support system gives the designer the ability to execute the prototype. This support system consists of four major components: a translator, a static scheduler, a dynamic scheduler and a debugger. The translator generates code, binding together the reusable components retrieved from the software base. Its primary functions are to implement data streams and control constraints. The static scheduler allocates time slots for operators with real-time constraints before execution begins. If the allocation succeeds, all operators are guaranteed to meet their deadlines even with worst case execution times. The dynamic scheduler invokes operators without real-time constraints in the time slots not used by the operators with real-time constraints. The debugger offers designer support for locating logical errors during prototype execution. [Pala90]

c. *Software Database*

The software database has two primary subsystems, the engineering database management system and the repository of reusable components, called the software base. An engineering database management system should provide the following facilities to support computer-aided software development environments [DL91]:

- Persistence
- Concurrency control
- Version control
- Reuse of past design objects
- Configuration control
- A wide variety of data storage
- Guarantees that data will not be corrupted due to security violations or media failure

Persistence means that objects in the database will exist after the process that created them has terminated. Concurrency control allows many design engineers concurrent access to design information. To keep data on several design alternatives, version control is required. Reuse of past design objects improves productivity and helps design engineers exploit past successes. Configuration control is needed to record the history of evolving systems and in guiding and controlling their evolution. A varied data store provides features for storing variable length text and graphical objects. Finally, security of data is important to safeguard valuable design information. [Dwye91]

The engineering database management system of CAPS supports all of the above facilities using an object-oriented approach (Ontos) [Nest86] supporting a graph model of software evolution [Luqi90].

The second subsystem, the software base, is a repository for reusable software components. The software base management system provides graphical tools to store components in the software base and search for components using a browser, keyword search, or query using a formal specification [McDo91]. While the mechanisms implemented to perform component retrieval are language independent, the software base in our implementation will be populated with reusable Ada components. More details about the structure of the software base and component retrieval mechanisms may be found in Sections B.4 and D of this chapter.

2. The Prototype System Description Language

The prototype system description language (PSDL) [LB88, LBY88] forms the basis of CAPS. It serves as an executable prototyping language at a specification or design level and has special features for real-time system design. The PSDL model is based on data flow under real-time constraints and uses an enhanced data flow diagram that includes non-procedural control constraints and timing constraints.

PSDL provides two kinds of building blocks for prototypes: abstract data types and operators. Software systems are modeled as networks of operators communicating via data streams. Figure 3.3 shows an example of a PSDL specification for an abstract data type component that implements a set and some of its operations.

The set package defines the operators *Empty*, *Add*, *In*, *Subset*, and *Equal* for a set of integers. Each operator description includes a specification that may optionally include inputs, outputs, exceptions, generic parameters, states and timing information. These interface characteristics are defined by the software engineer during the design process. An integral part of the design process in this rapid prototyping paradigm is to search for an existing component before writing any code to satisfy a requirement. The software base component retrieval tool exploits the interface characteristics of the specification entered by the designer to quickly partition the database and isolate components that are potential candidates. The details of this process, known as *syntactic* normalization and matching, are discussed in Chapter IV.

```
type SET
specification

   operator EMPTY
      specification
         output
            S1 : set
      end


   operator ADD
      specification
         input
            ELEMENT : integer
            S1 : set
         output
            S2 : set
      end


   operator IN
      specification
         input
            ELEMENT : integer
            S1 : set
         output
            RESULT : boolean
      end


   operator SUBSET
      specification
         input
            S1 : set
            S2 : set
         output
            RESULT : boolean
      end

   operator EQUAL
      specification
         input
            S1 : set
            S2 : set
         output
            RESULT : boolean
      end

   keywords SET, INTEGER

   description {Implements a set of
   integers}

   axioms {
      ***(operations empty add in
            subset equal}
      obj SET is sort Set .
         protecting Int .
         op empty : -> Set .
         op add : Int Set -> Set .
         op in : Int Set -> Bool .
         op subset : Set Set -> Bool .
         op equal : Set Set -> Bool .
         vars s1 s2 : Set .
         vars e1 e2 : Int .
         cq add(e1, s1) = s1 if in(e1, s1) .
         eq in(e1, empty)  = false .
         eq in(e1, add(e2, s1)) =
            or(==(e1, e2), in(e1, s1)).
         eq subset(empty, s1) = true .
         eq subset(s1, empty) = false .
         eq subset (add(e1, s1), s2) = and
            (in(e1, s2), subset(s1, s2)) .
         eq equal(s1, s2) =
            and(subset(s1, s2),
                  subset(s2, s1)) .
      endo  }
end
```

Figure 3.3 - A PSDL Specification for a Set

One of the latter parts of a PSDL component specification is the formal description of the component, that is, the axioms. In its current version, PSDL does not require any specific syntax for formal axioms. This part of the language definition has been left unspecified intentionally to provide flexibility, allowing alternative forms of specification. The author has chosen to augment PSDL with an algebraic specification

language known as OBJ3 [GW88, Wink91]. The OBJ3 axioms express the semantics of the specification and are the basis of *semantic* normalization and matching, another phase of the retrieval process. Figure 3.3 includes an OBJ3 specification in the axioms portion of the PSDL.

The OBJ3 portion of the specification is contained within the curly brackets that delimit the axioms portion of the PSDL specification. The line containing ***(operations empty add in subset equal) is an OBJ3 *comment* which is used here to indicate which of the operators the object will export. This information is used by the semantic normalization and matching algorithms described later.

3. OBJ3

OBJ3 is a functional programming language rigorously based on order sorted logic [GW88, Wink91]. It may be used to describe the syntactic and semantic properties of sequential processes but does not the have facilities for specifying the dynamics of concurrent processes[1]. The dominant construct in OBJ3 is the *module*. Modules can be *objects* or *theories*. An object completely determines the behavior of a type or parameterized set of types and a theory partially constrains the behavior of a set of types. Objects are fully executable and theories are partially executable because the theory may not contain enough constraints to fully determine the values of some of the operations. Because our retrieval mechanism requires the specifications to be fully executable, as we will show later, we focus on objects. The **axioms** part of the PSDL specification in Figure 3.3 defines an OBJ3 object, in this case an abstract data type for a set. OBJ3 objects consist of a *signature* and a set of *axioms* , the focus of the next two sections.

a. Signature

An OBJ3 definition of an object introduces a new set of values that contains all the instances of the type or *sort*[2] being defined. The *principal sort* of the abstract data type is the name of this set of values. The principal sort of the OBJ3 specification in Figure 3.3 is *Set*. The signature defines the *syntax* of the object's interface. It consists of a list of *op* definitions that have the following form [GW88]:

op (OpForm) : (Sort)... -> (Sort) [(Attributes)] .

---

[1]This is not a drawback, since the focus of this research is on process input/output characteristics as opposed to real-time processing characteristics.

[2]Order sorted logic uses the term “sort” rather than “type”.

A single op definition defines the name (OpForm), domain sorts, range sort and attributes of an operator[3]. OBJ3 offers tremendous flexibility in the OpForm, allowing *mixfix* syntax. Mixfix syntax allows the designer to specify the syntactic format of the operators and the operands within expressions. For simplicity, we restrict the OpForm to *prefix* syntax. We require the OpForm to be a simple identifier adhering to the following regular expression[4]: [a-z][a-z0-9]*. The axioms corresponding to the OpForm must be in prefix format also. For example, given the following op definition:

```
op subset : Set Set -> Bool .
```

the axioms used to define subset could look like:

```
eq subset(empty, s1) = true .
eq subset(s1, empty) = false .
eq subset (add(e1, s1), s2) = and(in(e1, s2), subset(s1, s2)) .
```

All sorts used in the op definition must be previously defined by the user or predefined in the language as one would expect with any typed language. The predefined sorts offered by OBJ3 include Bool (Boolean), Nat (Natural), NzNat (Positive), Int (Integer), Float, Rat (Rational), Qid, Qidl, and Id (Identifiers). The sorts in the object defined in Figure 3.3 are {Set, Int, Bool}. An operator whose range is the same as the principal sort is called a *constructor*. An operator whose range is a sort other than the principal sort is called an *accessor*.

Attributes may be added optionally to an op definition. Attributes add additional properties to operators such as associativity, commutativity, etc. that affect parsing, order of evaluation, and efficiency. We shall see later that attributes play an important role in semantic matching. The following example shows the use of associativity and commutativity attributes declared for a *sum* op definition:

```
op sum : Nat Nat -> Nat [assoc comm] .
```

b. Axioms

Axioms define the semantics of an object and are implemented as equations. The basic syntax for an equation in OBJ3 is

```
eq (Exp1) = (Exp2) .
```

where (Exp1) and (Exp2) are well-formed expressions of operations and variables present in the current context. The form of expressions in OBJ3 offers "...abstract denotational

---

[3]Since OBJ3 is a functional programming language, all operators are functions.

[4]An identifier begins with a lower case letter, followed by zero or more lower case letters and digits.

semantics based on order sorted algebra, and a more concrete operational semantics based on order sorted rewriting." [GW88, p. 7] The language is thus *executable* by treating the equations as rewrite rules, substituting matched instances of left-hand sides with corresponding right-hand sides.

There are also conditional equations of the form:

```
cq (Exp1) = (Exp2) if (Bexp) .
```

where the condition is a boolean expression. This type of rule fires only when the left-hand side is matched and the boolean expression on the right hand side evaluates to true.

Two final forms provided are:

```
bq (Exp) = (Lisp) .
```

and

```
cbq (Exp) = (Lisp) if (Bexp) .
```

which allow the user to perform Lisp operations in lieu of term replacement.

c. *Parameterized Modules*

Figure 3.4 shows an example of an OBJ3 specification for an *environment*, an abstract data type that keeps track of values bound to variables. This object is *parameterized*. There is an interface to the object in the form of ENVIRONMENT[Item Key :: TRIV] . The sorts Item and Key are called *parameterized sorts*, meaning that this a generic object that must be instantiated with *theories* that correspond to the generic parameters. A theory, which has a structure similar to that of an object, describes the structure and properties of the parameter. "Semantically, a theory defines a 'variety' of models, containing all the (order sorted) algebras that satisfy it, whereas an object defines just one model (up to isomorphism), its initial algebra." [GW88, p. 22] In the case of Figure 3.4, the theory used for both parameters is TRIV. TRIV is a predefined theory in OBJ3 of the form:

```
th TRIV
  sort Elt .
endth
```

```
obj ENVIRONMENT[Item Key :: TRIV] is
        sort Env .
        protecting BOOL .
        op null : -> Env .
        op default : -> Elt.Item .
        op bind : Elt.Item Elt.Key Env -> Env .
        op lookup : Elt.Key Env -> Elt.Item .
        op combine : Env Env -> Env .
        var E1 E2 : Elt.Item .
        var K1 K2 : Elt.Key .
        var Env1 Env2 : Env .
        eq lookup(K1,null) = default .
        eq lookup(K1,bind(E1, K1, Env1)) = E1 .
        cq lookup(K1,bind(E1, K2, Env1)) =
                lookup(K1,Env1) if K1 =/= K2 .
        eq combine(null, Env1) = Env1 .
        eq combine(Env1, null) = Env1 .
        cq combine(bind(E1,K1,Env1),Env2) =
                combine(Env1,bind(E1,K1,Env2))
                if lookup(K1,Env2) == default .
        cq combine(bind(E1,K1,Env1),Env2) =
                combine(Env1,Env2)
                if lookup(K1,Env2) =/= default .
endo
```

Figure 3.4 - OBJ3 Specification for an Environment

There is obviously very little in the way of structure or properties in the TRIV theory. To add structure and properties a *view* is required. A view specifies the way in which a certain module satisfies a certain theory. Thus we can create a new module (ENVT1) by instantiating the parameterized module with an actual parameter using a particular view. For example, the following statements could be used to instantiate the object in Figure 3.4 with objects NAT and FLOAT:

```
view ITEM1 from TRIV to FLOAT is endv
view KEY1 from TRIV to NAT is endv
make ENVT1 is ENVIRONMENT[ITEM1, KEY1] endm
```

Alternatively one could write:

```
make ENVT1 is ENVIRONMENT[FLOAT, NAT] endm
```

and have the views defined automatically. The new object, ENVT1, now defines an abstract data type that binds items of sort Float to keys of sort Nat. (By convention OBJ3

uses all capital letters for *module* names and a capitalized identifier for the *sort* defined by the module.)

d. Importing Modules

Objects may import operations and sorts from other objects using the *protecting* , *extending*, or *using* statement. The difference between these three forms of importation is related to the initial algebra semantics of objects [GW88]. When importing objects in the context of initial algebras, we must be aware of two properties related to the importation: "no junk" and "no confusion."[GW88, p. 18] The "no junk" property states that if a module M' is imported into a module M, then M' will not add any *new* data items of sorts already defined in M. "No confusion" states that if M' is imported into M, then M' will not define any *old* items already defined by M. With respect to these properties, the given importation mechanisms have the following characteristics:

Import Mechanism	Properties
protecting	no junk, no confusion
extending	no confusion
using	no guarantees at all

OBJ3 does not check whether these properties hold. The user must ensure that the chosen import method is appropriate for the object defined. In the object defined in Figure 3.4, we import another object BOOL using the *protecting* statement, which affords us the ability to use the operations *and*, *or* and *not* (among others) in Boolean expressions.

e. Why OBJ3?

Given the plethora of formal specification languages available today, we feel it is important to justify our selection of OBJ3. Since our particular implementation of the software base contains Ada [Ada83] reusable software components, we are concerned with how well-suited the chosen specification language is for describing Ada program units. One of the reasons we chose OBJ3 was because it corresponds well with Ada. It is easy to see parallels between OBJ3 objects and Ada packages. An OBJ3 signature is analogous to an Ada package specification and the axioms to a package body. Also, parameterized modules model the semantics of Ada generic software components that will be in the software base. The OBJ3 importation statements model the Ada *with*. Hence, OBJ3 specifications will have structures similar to the Ada modules they represent.

Given this close correspondence between OBJ3 and Ada, designers will be able to formulate their formal specifications more readily. Personnel familiar with Ada syntax and semantics will be able to easily identify the parallels between the two

languages, better understand the formal specifications, and more easily write specifications. OBJ3 provides a degree of consistency one would not find with other specification languages.

A further justification for the use of OBJ3 is its execution system. OBJ3 specifications have operational semantics when the axioms are treated as rewrite rules. In addition, the term rewriting system can be used as a theorem prover. These features are particularly important to our method of component retrieval and are therefore mandatory requirements for the chosen specification language.

f. Why not Predicate Logic?

Predicate logic is a solid candidate for use as a specification language. It has executable implementations (e.g., Prolog, Eql, etc.) and has been promoted as a formal specification language [Luqi87] , as a reusable component retrieval mechanism [RW90c], and as a basis for transformation from specification to executable code. What it lacks, however, is a close correspondence to Ada. We are already asking the designer to learn a formal specification language in order to express the semantics of modules. In the interest of regularity and syntactic consistency [MacL87], it is prudent to have the specification language be as close as possible to the implementation language without sacrificing necessary characteristics of the specification language. While predicate logic has executable implementations and theorem proving power, its syntax is an unnecessary inconvenience.

4. Component Retrieval Subsystem

Having described CAPS, PSDL, and OBJ3, we now focus on the component retrieval subsystem. This section provides a broad overview of the retrieval system and the general approach that lies beneath it. Finer details of the retrieval mechanisms may be found in Section D of this chapter.

a. Formal Specifications for Component Retrieval

The paradigm for rapid prototype construction in CAPS leads the designer from a graphical representation of the prototype, through specification with a prototyping language, and then on to code generation. Figure 3.5 shows the prototyping process supported by CAPS.

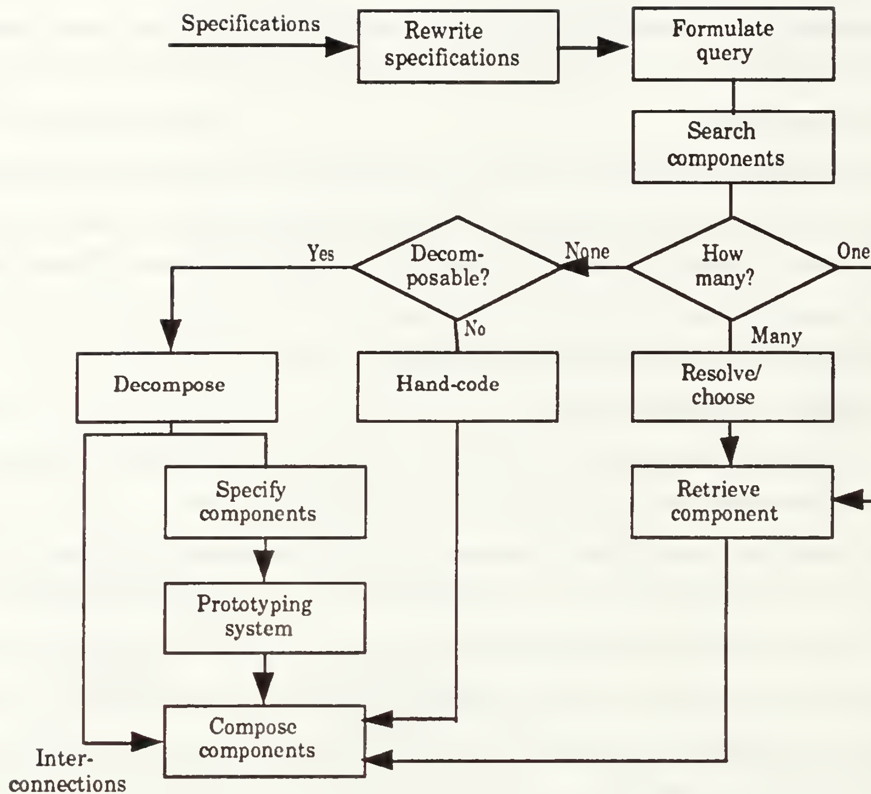


Figure 3.5 - The CAPS Prototyping Process

Note that CAPS is not designed to be a *code synthesis* system, which translates formal specifications into executable code (such as REFINE [Reas86]) . Instead, CAPS takes advantage of a library of reusable software components.

Since the prototype designer writes specifications for the operators and data streams to model system requirements, we use these specifications to locate components that will satisfy those requirements. A retrieval system that is automatic, efficient, and effective relieves the designer from having to use a browser or some other manual means to locate components. This is particularly beneficial when the software base contains thousands of components.

b. The Role of Normalization

The designer's specification for an operator serves as a key in the search for an appropriate component. Like most information retrieval mechanisms, we must modify the key in some way to improve the efficiency of the search. An analogy to this is *hashing*, a widely used technique for implementing table lookup algorithms [AHU83] where a given

key is manipulated mathematically to find an object's actual address within a data structure. The process of transforming or manipulating the specification for a reusable software component is called *normalization*.

The PSDL specification, augmented with OBJ3, describes both the interface (syntax) and the behavior (semantics) of an object. Hence, we perform two types of normalization: syntactic and semantic. Syntactic normalization standardizes the form of the query's interface characteristics to be used in syntactic matching. Semantic normalization transforms the signature and axioms of the OBJ3 portion of the specification to make them suitable for semantic matching. In both cases normalization is necessary based on the algorithm used for matching.

c. *System Structure*

The CAPS software base basically supports two activities: component storage and component retrieval. Figures 3.6 and 3.7 abstractly illustrate the storage and retrieval processes.

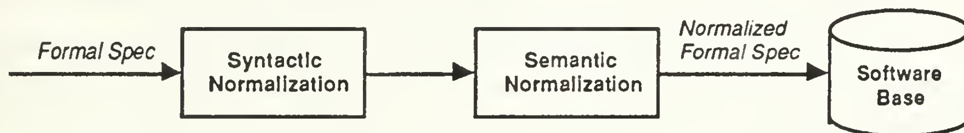


Figure 3.6 - Normalization for Component Storage

Components to be stored must first pass through syntactic and semantic normalization. The normalization processes transform the component's specification to facilitate later matching. The normalized specification is stored with the component in the software base.

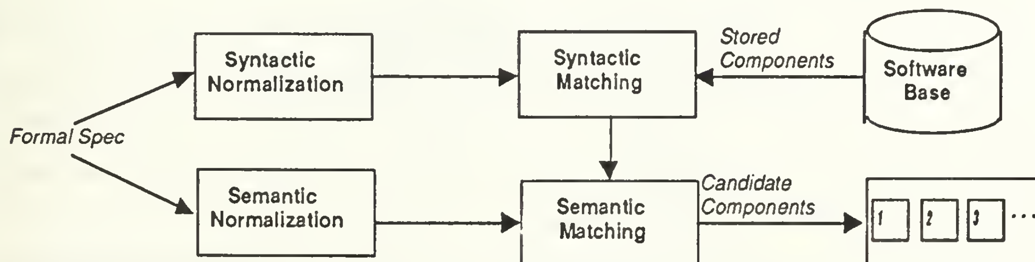


Figure 3.7 - Normalization for Component Retrieval

Figure 3.7 shows the abstract process for component retrieval. A query for a library component is a PSDL/OBJ3 specification. The query is syntactically and semantically normalized and then matched against stored specifications. Syntactic and semantic normalization may proceed in parallel but syntactic matching must take place before semantic matching. Syntactic matching is faster and partitions the software base quickly in order to narrow the list of possible candidates that the semantic matching algorithm must consider. Semantic matching may be time consuming and should be applied to as small a candidate list as possible without excluding potential matches. Semantic matching should provide an ordered list of candidate components.

Both syntactic and semantic normalization and matching are required to achieve the best performance from the system. The main benefits of syntactic matching are speed and recall, whereas the advantage of semantic matching is precision. We believe that this precision is required in order to reduce and rank order the reusable components that a designer will have to evaluate before making a selection.

This section provided a brief look at the component retrieval subsystem of CAPS. It serves as an introduction in order to better understand the following section on our initial assumptions and models. More detail on the retrieval mechanisms may be found in Section D of this chapter.

C. INITIAL ASSUMPTIONS AND MODELS

Semantic normalization and matching is the focus of this dissertation. We review the syntactic methods to some extent in Section III.D. The ensuing description of our initial assumptions and models relates to semantic normalization and matching techniques only.

1. Initial Assumptions

The search for a component is an information retrieval problem. It can be divided into two parts: representation and search. A representation is the model of the object sought and the search exploits the representation to find a desired object. A sophisticated representation technique should simplify the search problem. Conversely, a simple representation implies an involved search mechanism.

A tradeoff exists between representation and search. Increased sophistication in one area leads to simplification in the other. Looking at the two extremes, it would be profitable to find either a representation technique (normalization) that makes search trivial or a search technique that obviates normalization. For both of these extremes, we

can exploit an algebraic formalism (OBJ3) for specifying components. The preferred method proposed in this dissertation lies between the two extremes and has non-trivial components for both normalization and matching. Sections 2 and 3 explain the idealized extreme approaches and Section 4 describes the middle ground, that is, our modified assumptions.

2. Semantic Normalization

An ideal semantic normalization method would transform the axioms of two *semantically* equivalent objects into *syntactically* equivalent forms. Consider an ideal normalization algorithm. Figure 3.8 illustrates that, given two semantically equivalent specifications, A and B, the result of passing them through the ideal normalization procedure should yield the same specification, C. Ideally, any specification semantically equivalent to A or B should be transformed to C when passed through the procedure.

To implement the ideal normalization procedure we considered applying a set of *rewrite rules* to specifications to transform them. Since the axioms used to describe the semantics of a module are a *formal* language with a well-defined, regular structure, it is possible to automatically rewrite a set of axioms to an alternative form with the same meaning, that is, use *semantics preserving* transformations. A set of general purpose rewrite rules could be used to rewrite semantically similar axiom sets or *normalize* them to a common form. Thus, with respect to information retrieval, our representation technique becomes semantic normalization and our search is a simple matter of comparing axioms for syntactic equality. The following section shows an example of this approach.

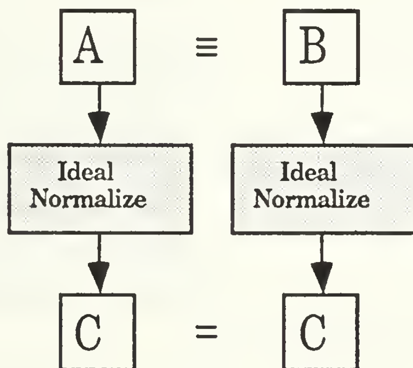


Figure 3.8 - Ideal Normalization of Axioms

a. Example

Consider the following example [Luqi87]. Given two specification fragments, we want to use rewrite rules to normalize them, reducing them to a syntactically equivalent form. We start with the following fragments:

1 <= i < j <= length(REPLY) → REPLY(i) <= REPLY[j] [1]

REPLY = a @ [x] @ b @ [y] @ c → x <= y [2]

Equation 1 uses indices and Equation 2 uses concatenation of subsequences (@) to specify that the elements of REPLY, the output of some software module, must be sorted in increasing order. The *solid* arrow used in the fragments (→) denotes an implication. Table 3.1 shows a set of rewrite rules that could be applied to Expression 1 to make it syntactically similar to Expression 2.

TABLE 3.1 - CONDITIONAL REWRITE RULES

#	Rule	Comment
R1	s = a @ [x] @ b → s[length(a) + 1] --> x	Relationship between the indices and data value at a given position in a sequence
R2	x < y + x --> 0 < y	Standard ordering on integers
R3	x <= y + x --> 0 < y	Standard ordering on integers
R4	0 <= length(s) --> true	Theorem about lengths of sequences
R5	true & p --> p	Absorbtion law of Boolean algebra
R6	p & true --> p	Absorbtion law of Boolean algebra
R7	x <= y < z --> x <= y & y < z	Definition of repeated inequalities
R8	x < y <= z --> x < y & y <= z	Definition of repeated inequalities
R9	REPLY --> c @ [y] @ d	Derived from Expression 1.2 in the hypothesis of the implication
R10	length(s @ t) --> length(s) + length(t)	Basic fact about the length of a sequence
R11	length([x]) --> 1	Basic fact about the length of a sequence
R12	x + y <= z + y --> x <= z	A standard inequality law
R13	length(s) < length(u) & s @ t = u @ v → u –> s @ w	Common prefix law for sequences

The *broken* arrow used in the rules denotes term rewriting, that is, if the expression on left-hand side can be matched and the conditions are met, then it can be

replaced with the expression on the right-hand side using a consistent binding for the variables.

We first apply R1 to Expression 1 under the substitution (s: REPLY, i: length(a) + 1) resulting in Expression 1.1.

```
REPLY = a @ [x] @ b & 1 <= length(a) + 1 < j <= length(REPLY)        [1.1]
        → x <= REPLY[j]
```

Applying R1 again with the substitution (s: REPLY, j: length(c) + 1) yields Expression 1.2.

```
REPLY = a @ [x] @ b                                                  [1.2]
      & REPLY = c @ [y] @ d
      & 1 <= length(a) + 1 < length(c) + 1 <= length(REPLY)
      → x <= y
```

Next, we can reduce to **true** the condition

```
1 <= length(a) + 1
```

using rules R2 and R4, and eliminate the truth value using R5 and R7. This yields:

```
REPLY = a @ [x] @ b                                                  [1.3]
      & REPLY = c @ [y] @ d
      & length(a) + 1 < length(c) + 1
      & length(c) + 1 <= length(REPLY)
      → x <= y
```

R12 is used to simplify

```
length(a) + 1 < length(c) + 1
```

to:

```
length(a) < length(c)
```

and the condition

```
length(c) + 1 <= length(REPLY)
```

can be reduced to **true** by applying R9, R10 (twice), R11, R12, R3 and R4. The truth value is eliminated using R6. The result is Expression 1.4.

```
REPLY = a @ [x] @ b                                                  [1.4]
      & REPLY = c @ [y] @ d
      & length(a)  < length(c)
      → x <= y
```

Further progress can be made using R13. Under the substitution (s: a @ [x], t: b, u: c, v: [y] @ d), the result is Expression 1.5.

```
REPLY = a @ [x] @ w @ [y] @ d → x <= y                               [1.5]
```

Expression 1.5 is the same as Expression 2, up to renaming of variables. If we rename the variables in a consistent manner, the two expressions are syntactically identical.

The above example is a powerful one that demonstrates that a set of rewrite rules, most of them standard laws, *can* be used to transform semantically equivalent expressions into syntactically equivalent forms. The question of whether this can be done *automatically*, however, raises some interesting issues.

b. Issues

If we refer to the rewriting example shown above as *normalization*, then we should contrast the process with our concept of ideal normalization. In the example above we started with two expressions and our goal was to rewrite one to look like the other. The application of rules was focussed on making Expression 1 identical to Expression 2. Hence, we could say that the process was *goal-driven*. This is analogous to manual theorem-proving, where we know what it is we want to prove and we select axioms that take us closer to our goal. Under ideal normalization used for component storage, however, there is no defined goal. Referring back to Figure 3.6, a component specification is normalized before it is stored. This normalization takes place in the absence of any corresponding specification with which to compare the specification being normalized. In essence, the normalization process has no defined goal toward which to work.

One approach to this problem is to simply apply rewrite rules until no more can be applied, that is, until the expression or expressions have reached *normal form* [Gogu88]. In order for the system to be automatic, the system of rewrite rules would have to be Church-Rosser and terminating (confluent and nœtherian) [HO80]. The Church-Rosser property is a completeness property that states, given terms M, N, and P, that if

$$P \to^{*} N \text{ and } P \to^{*} M,$$

then there must be a Q such that

$$M \to^{*} Q \text{ and } N \to^{*} Q,$$

where $\to^{*}$ is the symbol for successive application of rewrite rules [HO80]. The termination property states that there is no infinite chain of reductions (rewrite applications) for any term M. If our system of rewrite rules has the termination property, then the property of confluence is decidable [HO88]. In fact, the Knuth-Bendix [KB67] completion procedure can be used to augment the system of rules with additional rules to make the system Church-Rosser. Unfortunately, even if we did come up with a general set of rewrite rules that were Church-Rosser and terminating, additional problems relating to the structure axioms sets makes ideal normalization infeasible, leading us to conclude that some combination of non-ideal normalization and theorem proving is necessary.

For example, Figures 3.9 and 3.10 show two OBJ3 specifications for a *Set*. Both components define operations for constructing sets and testing membership, subset, and equality. The main difference lies in the way each component tests for equality.

In Figure 3.9, a *hidden*[5] "remove" operation is used to define the semantics of equality. It is considered hidden because it is not included in the list of exported operations defined in the ***(operations ...) comment. In Figure 3.10, the "subset" operation is used to define equality. This presents several problems. If we consider the "remove" operation in Figure 3.9 to be hidden, then the semantics of the specifications are equivalent. Suppose both of the specifications were passed through our ideal normalization procedure. To make either of these specifications look like the other would require the system to know the semantics of sets and set operations. We hypothesize that it *may* be possible to automatically synthesize a "remove" operation for the specification in Figure 3.10 or to eradicate the "remove" operation from the specification in Figure 3.9, but to do either would be extremely difficult.

The above example is a simple case. The main problem with ideal normalization using rewrite rules lies in the infinite variations possible in expressing component semantics. Even if we could expect to get two semantically equivalent specifications syntactically *close*, we would need additional help from the matching algorithm to determine how well one specification satisfies the semantics of another. We therefore turn to the other extreme, applying sophistication to the matching algorithm rather than the normalization algorithm.

3. Matching via Theorem Proving

The previous section shows that we cannot rely completely on normalization (the representation) to solve this information retrieval problem. This section focuses on the search mechanism in order to reduce the complexity required in normalization.

---

[5]The term *hidden* is derived from the software engineering concept of *information hiding* [Parn72] which states that the information contained within a module should be inaccessible to other modules that have no need for the information. In the case of an abstract data type (ADT), additional operations may be defined to support the function of the ADT's primary operations. It is not intended for the user of the ADT to access these auxiliary operations directly. Hence, they remain hidden.

```
***(operations  empty add member subset equal)
    obj SET1 is sort Set  .
    protecting NAT  .
    op empty : -> Set .
    op add : Nat Set -> Set .
    op member : Nat Set -> Bool .
    op subset : Set Set -> Bool .
    op equal : Set Set -> Bool .
    op remove : Nat Set -> Set .
    var E1 E2 : Nat .
    var S1 S2 : Set .
    cq add(E1,S1) = S1 if member(E1, S1) .                                  [1]
    eq member(E1,empty) = false .                                           [2]
    eq member(E1,add(E2,S1)) = E1 == E2 or member(E1,S1) .                  [3]
    eq subset(empty,S1) = true .                                            [4]
    eq subset(S1,empty) = false if S1 =/= empty .                           [5]
    eq subset(S1, S1) = true .                                              [6]
    eq subset(add(E1,S1),S2) = member(E1,S2) and subset(S1,S2) .            [7]
    eq equal(empty,empty) = true .                                          [8]
    eq equal(S1, S1) = true .                                               [9]
    eq equal(add(E1,S1),empty) = false .                                    [10]
    eq equal(empty,add(E1,S1)) = false .                                    [11]
    eq equal(add(E1,S1),add(E2,S2)) = member(E1,add(E2,S2))  and            [12]
                equal(S1,remove(E1,add(E2,S2))) .
    eq remove(E1,empty) = empty .                                           [13]
    eq remove(E1,add(E1,S1)) = S1 .                                         [14]
    cq remove(E1,add(E2,S1)) = add(E2,remove(E1,S1)) if E1 =/= E2 .         [15]
endo
```

Figure 3.9 - OBJ3 Specification for a Set

```
***(operations empty add member subset equal)
obj SET2 is sort set .
    op empty : -> Set .
    op add : Nat Set -> Set .
    op member : Nat Set -> Bool .
    op subset : Set Set -> Bool .
    op equal : Set Set -> Bool .
    var E1 E2 : Nat .
    var S1 S2 : Set .
    cq add(E1,S1) = S1 if member(E1, S1) .                                  [1]
    eq member(E1,empty) = false .                                           [2]
    eq member(E1,add(E2,S1)) = E1 == E2 or member(E1,S1) .                  [3]
    eq subset(empty,S1) = true .                                            [4]
    cq subset(S1,empty) = false if S1 =/= empty .                           [5]
    eq subset(S1, S1) = true .                                              [6]
    eq subset(add(E1,S1),S2) = member(E1,S2) and subset(S1,S2) .            [7]
    eq equal(S1,S2) = subset(S1,S2) and subset(S2,S1) .                     [8]
endo
```

Figure 3.10 - Alternative OBJ3 Specification for a Set

Because each formal specification contains a set of axioms that taken together constitute a theory, T, we can use theorem proving to show that the axioms of a query specification are satisfied by a component specification. Given a query specification, called a *presentation* $P_q(\Sigma_q, E_q)$ [Gogu88], its signature Σ_q, and its axioms E_q, we would like to determine if a candidate component specification $P_c(\Sigma_c, E_c)$ can satisfy the query. We assume that there is a Σ-homomorphism, $h: \Sigma_q \rightarrow \Sigma_c$, that maps the signature of the query to the signature of the component (determining this mapping is another problem in itself, described later). Given the homomorphism, we can prove that a candidate satisfies a query if we can show that each axiom, eq_i, of the query is satisfied in the theory of the stored component, E_c. Formally,

- Given: $P_q(\Sigma_q, E_q(eq_1 \ldots eq_n))$, $P_c(\Sigma_c, E_c(ec_1 \ldots ec_n))$, and $h: \Sigma_q \rightarrow \Sigma_c$
- Then: $P_c \models P_q$ iff $\forall i(1 \leq i \leq n)\ E_c \models h(eq_i)$

In other words, a stored component P_c satisfies a query P_q if and only if there is a homomorphism from Σ_q to Σ_c and each $e_q \in E_q$ is satisfied in E_c.

a. Example

As an example, we refer back to Figures 3.9 and 3.10. If the specification in Figure 3.9 were a query and the specification in Figure 3.10 corresponded to a stored component, we would first need to find a mapping between the two components. We seek an *injective* (one-to-one) mapping from the set of specified operations in the query to the specified operations in the component. If we do not consider the "remove" operation in the query (the designer must specify this), the mapping is trivial. Given the morphism, we must show that axioms [1] through [15] of the query are each satisfied by stored-component axioms [1] through [8]. The first seven axioms of the query are proven trivially since they are identical to those in the stored specification. Axioms [8] and [9] of the query are proven by first applying axiom [8] and then axiom [6] of the component. Axioms [10] and [11] of the query are proven by axioms [8] and [5] of the component. At this point all of the remaining axioms in the query make use of the "remove" operation. Since the designer specified that "remove" was a hidden operation (it was left out of the export list), it is not reasonable to expect the library component to satisfy the "remove" axioms ([13] through [15]). That leaves us with axiom [12] which uses the "remove" operation. Since there are no semantics for the remove operation in the stored component specification, axiom [12] cannot be proven without constructing the definition of the hidden remove operation, which can be very difficult to do automatically in the general case. Even though we *know* that the stored

component satisfies the requirements of the query, it is very hard to show it conclusively via theorem proving.

b. Issues

It is clear that theorem proving alone does not offer a complete solution to the specification matching problem. Besides the problem highlighted above relating to hidden operations, theorem proving has other drawbacks. In general, the process is slow and not guaranteed to terminate. To be practical, the axioms for each stored component would have to be *canonical*, but given our choice of specification language (OBJ3), it is not reasonable to expect or enforce this. OBJ3 does not have order-sorted Knuth-Bendix and unification algorithms and there does not exist a general method to check for termination [Gogu88].

4. Modified Assumptions

The difficulties inherent in both normalization using rewrite rules and theorem proving led us to modify our assumptions about what normalization should be and what constitutes a semantic match. We cannot rely on the rewrite rules to perfectly normalize axioms just as we cannot rely solely on theorem proving to perform perfect matching. But formal semantics *should* provide us with a means to compare components! A software designer who understands algebraic semantics can compare the behavioral properties of objects by analyzing the axioms. An automated matching system should be able to do the same. The next section describes the details of our overall schema and the method we have chosen to exploit formal semantics in the component retrieval problem.

D. SCHEMA FOR REUSABLE COMPONENT RETRIEVAL

Our proposed approach to reusable component retrieval is two-phased. The first phase focuses on the numbers and types of parameters within each operator in the PSDL portion of the query. This information is used to form a search key that partitions the software base, quickly ruling out those components that cannot possibly satisfy the query because of type incompatibilities. This phase, called the *syntactic search* phase, provides a set of components to the subsequent *semantic search* phases. Syntactic search requires syntactic normalization.

The second phase (semantic search), called *query by consistency*, relies on the formal OBJ3 specification for each component. Query by consistency formulates example terms from a query's algebra and passes the terms to its axioms for reduction. The set of outputs obtained is compared against the outputs from similar tests performed in the domain of a candidate. This phase reduces further the set of candidate components, eliminating

components that cannot possibly satisfy the query because of behavioral incompatibilities. Query by consistency requires normalization of OBJ3 specification signatures and axioms.

The following sections describe the details of syntactic normalization and matching, and semantic normalization and matching.

1. Syntactic Normalization and Matching[6]

The purpose of syntactic matching is to rapidly eliminate from consideration those modules in the software library that cannot match the query specification's interface. This matching process uses only the query module's PSDL interface specification. Once those modules with unsuitable interfaces have been removed, only a small subset of the software base needs to be semantically analyzed. The syntactic matching process reduces the number of candidate modules sufficiently to make semantic matching feasible. For small software bases, that is, "...where classes are contributed by a small number of people, and the total number of classes does not exceed a few tens or perhaps a few hundreds" [Meye88a, pp. 445-446], a browser is a practical alternative. As the software base grows beyond this, however, other means such as syntactic and semantic matching must be employed.

Before explaining syntactic normalization, we define what constitutes a syntactic match. PSDL allows the definition of both *type* and *operator* modules. Since a *type* module is a super-set of an *operator* module, the definition of an *operator* module match will be given in detail and then extended for use with *type* modules.

The components of a PSDL specification *p* for a software component *c*, that are important to the syntactic matching process are as follows:

S(p)= ({In(t,n) : there are n occurrences of type t as input parameters to c },
{Out(t,m) : there are m occurrences of type t as output parameters to c },
{E : E is an exception defined in c},
{St : St is a state variable in c})

S(p), a subset of the PSDL specification for module *c*, is the only part of the specification that pertains to the syntactic matching process. Given a software base module *m*, and a query module *q*, along with their respective PSDL interface specifications S(m) and S(q) then *m* is a syntactic match for *q* if and only if the following rules hold true:

---

[6]This section is abstracted from [McDo91] with permission.

- $\exists f_i : S(q) \rightarrow S(m)$ st $[(f_i\ (In(t',n)_q) = In(t',m)_i \Rightarrow (m=n \wedge (t=t' \vee t'$ is a generic match to t))) $\wedge f_i$ is bijective] [1]

- $\exists f_o : S(q) \rightarrow S(m)$ st$[(f_o(Out(t,n)q) = Out(t',m)_o \Rightarrow (m=n \wedge (t=t' \vee t'$ is a generic match to t))) $\wedge f_o$ is injective] [2]

- if $|\{ST_q\}| > 0$ then $|\{ST_m\}| > 0$ else $(|\{ST_q\}| = |\{ST_m\}| = 0)$ [3]

This definition of a syntactic match could be used directly to determine if a software base component could match a query specification's interface but would require the system to check every component in the software base. This type of implementation would be very inefficient. A better strategy uses matching rules to derive a set of module attributes that can be used to rapidly identify and reject modules with unsuitable interfaces. Some examples of these derived attributes include:

- If the number of input parameters in S(q) is not equal to the number input parameters in S(m), then there can be no function f_i to satisfy rule [1] without considering the semantics of parameters. Therefore S(m) can be eliminated from the search.

- If the number of output parameters in S(q) is greater than the number of output parameters in S(m), then there can be no function f_o to satisfy rule [2]. Therefore S(m) can be eliminated from the search.

- If S(q) has state variables defined (i.e. q defines a state machine) but S(m) has no state variables, then S(m) can be eliminated from the search.

Although passing these simple tests does not constitute a syntactic match, a failure does eliminate the module from further consideration because it cannot be a syntactic match. These attributes are derivable from the PSDL specification and can be used to form multi-attribute keys. These keys allow rapid reduction in the size of the viable subset of the software base via multi-attribute queries without the need to attempt to identify the individual mapping functions for each module. For those modules that are selected by the multi-attribute query, additional checks can be made to identify components that cannot meet rules [1] and [2]. These checks form a filtering mechanism that removes any unsuitable components from the query result.

The rules for syntactic matching of *type* modules are similar to those for *operator* modules with the addition of a mapping function to map the operators of S(q) to the operators

of S(m) and an additional check to ensure the generic parameter substitutions used for this mapping function are consistent for all operators in S(m). Multi-attribute keys can be formulated that incorporate these additional requirements. These keys can then used for the initial *type* module database query and additional checks only applied to those modules that are selected by the multi-attribute query.

2. Semantic Normalization and Matching

The task of the syntactic retrieval tool is to obtain a set of components from the software base that meet the syntactic requirements of a query, based on the *interface* of the query. The information about the interface is derived from the PSDL specification for the query. Syntactic search is efficient, quickly excluding components that cannot possibly match, resulting in a set of components that are passed to the semantic retrieval mechanism.

The technique used for semantic retrieval is called query by consistency. Query by consistency exploits the OBJ3 formal semantics in order to rule out components that are not good candidates and rank order components that are. The method generates sample terms from the term algebra of the query, performs reductions on those terms in both the query and the candidates and compares the results. Candidates whose outputs correspond more closely to the outputs of the query achieve a higher score and are deemed a better match. A threshold score can be used to eliminate some components from consideration. The details of query by consistency are covered in Chapter IV.

E. SUMMARY

In this chapter we described the model for reusable software component retrieval for the Computer Aided Prototyping System. The paradigm of CAPS is to build prototypes based on specification of requirements written in PSDL and OBJ3. Components to implement requirements are sought using the formal specifications as keys to search the software base. Efficient syntactic and semantic retrieval rely on normalization of the specification. Syntactic normalization and matching should be fast and provide high *recall*. Semantic normalization and matching improves *precision*. The remainder of this dissertation describes the theory and implementation of semantic normalization and retrieval.

IV. COMPARING SPECIFICATION SEMANTICS

A. INTRODUCTION

This chapter describes a method for reusable software component retrieval using normalized algebraic specifications. The method is called query by consistency (QBC). Given a query for a software component in the form of an algebraic specification, QBC automatically builds a set of example *terms* from the constructors provided in the signature of the specification, performs reduction on the terms using the axioms in both the query and stored components, and compares the results in order to eliminate some candidates and rank order the ones that remain.

The chapter begins by explaining some of the background theory behind QBC and then describes the techniques used for specification normalization, specification mapping, test set and I/O list construction, term reduction, and interpretation of results. The chapter ends with a formal explanation of the query/retrieval model that verifies its use as a semantic retrieval mechanism.

B. BACKGROUND

Query by consistency compares two specifications by evaluating the equivalence of algebraic terms reduced in the domains of the query specification and the specifications corresponding to candidate components. Term reduction means submitting a term to the specification axioms and performing term rewriting on the term until it has reached normal form, that is, a form wherein no further reductions are possible. The list of example terms (an *I/O list*) used in the QBC method is generated from a base set of terms called a *test set* . The test set is derived from the signature of the query.

The idea of using a test set stems from the work of Kapur and Zhang [KZ89] who developed a refinement to an inductionless induction procedure called *proof by consistency* [KM87]. In proof by consistency using test sets, a canonical algebraic theory is augmented by an axiom to be proven (a conjecture) and a new extended canonical theory is incrementally computed using the Knuth-Bendix completion algorithm. Whenever a new rule is generated during the process, the rule is checked against a test set to see if it reduces any of the irreducible ground constructor terms contained in the set. If the new rule can reduce a term in the test set, then the conjecture is not a theorem.

The test set is the key to proof by consistency. It is a finite set of terms that describes the equivalence classes of constructor ground terms. For example, a test set for integers with successor (*suc*) and predecessor (*pre*) constructors would be {0, *suc*(0), *suc*(*suc*(x)), *pre*(0), *pre*(*pre*(y))}. The test set used in QBC is similar to that used in proof by consistency. It is explained in more detail in Section IV.E.

The implementation of QBC is in the form of two executable programs. The first is a program to normalize the specifications that accompany components to be stored. The second program is used for matching a query specification to the specifications of candidate components. The following sections explain the processes. Implementation details are covered in Chapter V.

C. NORMALIZATION

Before a component is stored in the software base, its OBJ3 specification must be normalized. This normalization is performed when the component is stored to save time during the matching process. Just prior to matching, the query specification must be normalized. In both cases, expansion and instantiation are needed to make the specification an *atomic* unit. Interface normalization is also required for both specifications, but the result is different in each of the normalization routines. The following sections describe expansion, instantiation, and interface normalization.

1. Expansion and Instantiation

Expansion and instantiation in normalization was developed in the context of the Algebraic Specification Formalism (ASF) [BHK89]. In this approach, a normal form is achieved when all imports to a specification have been eliminated and as many parameters as possible have been eliminated. ASF's textual normalization *expands* a module by fully incorporating the sorts and functions of imports and by binding parameters to the greatest extent possible. The purpose of this normalization in ASF is to assign a semantics to the complete specification and to each module within the specification. In the process of normalizing, the algorithm *renames* sorts and functions to avoid conflicts; establishes the origin of each sort, function and variable, creating an attribute collocated with each definition; and binds formal with actual parameters.

In the system described in this dissertation, the normalization process also performs expansion and instantiation where necessary. The expansion is necessary because the module will be considered an atomic unit during the matching process.

Renaming is not performed in the system because OBJ3 allows operator overloading[1]. The following example illustrates this concept using a specification for a List (see Figure 4.1) and one for a BiTuple (see Figure 4.2). (Note: The ellipses that appear in many of the example specifications mean that there is more to the specification than is actually being shown.)

```
obj LIST[Item :: TRIV] is sort List .
    protecting NAT .
    protecting BOOL .
    op nil : -> List .
    op cons : Item List -> List .
    op length : List -> Nat .
    op head : List -> Item .
    op tail : List -> List .
    op append : List List -> List .
    op reverse : List -> List .
    op member : Item List -> Bool .
    ...
endo
```

Figure 4.1 - Signature for a List

```
obj BITUPLE[C1 :: TRIV, C2 :: TRIV] is
    sort BiTuple .
    op make : Elt.C1 Elt.C2 -> BiTuple .
    op first : BiTuple -> Elt.C1 .
    op second : BiTuple -> Elt.C2 .
    ...
endo
```

Figure 4.2 - Interface Description for a BiTuple

Suppose one used the List defined in Figure 4.1 in the following way:

```
obj LIST-OF-BITUPLE is
    protecting LIST[BITUPLE[NAT,NAT]] .
    op member : Nat List -> Nat .
    ...
endo
```

---

[1] In the current implementation of the system, it is assumed the designer has used unique names in specifying all operators, hence overloading is not supported. In Chapter VII a simple procedure is defined to remedy this situation and permit operator overloading.

The user has defined his own object which is composed of the List object and an object called BiTuple that defines a relation of 2 elements. The user has also defined a member function that returns the second argument of a tuple in the list given the first argument. The expanded version of the object is shown in Figure 4.3. It was necessary to instantiate the sort *Item* in object List as BiTuple and the elements of BiTuple as Nat.

```
obj LIST-OF-BITUPLE is sort List .
    sort BiTuple .
    protecting NAT .
    protecting BOOL .
    op nil : -> List .
    op cons : BiTuple List -> List .
    op make : Nat Nat -> BiTuple .
    op length : List -> Nat .
    op head : List -> BiTuple .
    op tail : List -> List .
    op append : List List -> List .
    op reverse : List -> List .
    op member : BiTuple List -> Bool .
    op first : BiTuple -> Nat .
    op second : BiTuple -> Nat .
    op member : Nat List -> Nat .
    ...
endo
```

Figure 4.3 - Interface Description for a List of BiTuple

The object in Figure 4.3 is expanded further by importing all operators and axioms defined in modules NAT and BOOL. The final step in this part of the normalization process is to store into a file the sorts, operators, and axioms defined in this atomic object. Interface normalization will add more information to this file.

2. Interface Normalization

Having performed expansion and renaming, the signature is now transformed to simplify mapping. Since Prolog is used as the tool to find the mappings between a query and a candidate component, each operator definition in the signature is transformed into a set of Prolog predicate expressions. To guide this transformation, it is necessary to have more information about the operators than is provided in the specification, that is, which of the operators the user wants considered.

For example, if the specification shown in Figure 4.3 were used as query to the software base, the user may not need all of the operators that come with the List object. A

more general query with fewer op definitions would certainly offer better recall from the software base. Also, the user may have defined hidden or local operators in his object that he does not require the stored component to provide. It is therefore left up to the user to specify the operators he wishes to have considered. A specification used for a query may have only a few of the operators identified, whereas a specification accompanying a component to be stored may have all operators identified. Figure 4.4 shows an example of the LIST-OF-BITUPLE used as a query and Figure 4.5 shows it used as part of a component to be stored.

```
***(operations nil cons make append length)
obj LIST-OF-BITUPLE is sort List .
    sort BiTuple .
    protecting NAT .
    protecting BOOL .
    ...
endo
```

Figure 4.4 - List of BiTuple as a Query

```
***(operations nil cons tail append reverse
    make length head first second member)
obj LIST-OF-BITUPLE is sort List .
    sort BiTuple .
    protecting NAT .
    protecting BOOL .
    ...
endo
```

Figure 4.5 - List of BiTuple for Storage

The specifications in Figures 4.4 and 4.5 have been augmented with OBJ3 comment blocks, "***(comment)", to indicate the operators the user wants considered. From this information and that contained in the signature, the necessary Prolog predicate expressions may be generated. For each operator specified in the signature, a corresponding *operator* predicate is defined, and for each input parameter in the operator an *argument* predicate is defined. The set of predicates for the specification in Figure 4.4 is:

```
operator(BITUPLE, 2, MAKE)
argument(MAKE, nat, MAKE1)
argument(MAKE, nat, MAKE2)
```

```
operator(LIST, 0, NIL),
operator(LIST, 2, CONS),
argument(CONS, BITUPLE, CONS1)
argument(CONS, LIST, CONS2)
operator(nat, 1, LENGTH)
argument(LENGTH, LIST, LENGTH1)
operator(LIST, 2, APPEND)
argument(APPEND, LIST, APPEND1)
argument(APPEND, LIST, APPEND2)
```

Each *operator* predicate expression has 3 arguments: a variable to bind to the range sort of a stored component's operator, the number of domain (input) parameters in the operator, and a variable to bind to the name of a stored component's operator. Each *argument* predicate expression has 3 arguments: a variable to bind to an operator name, the sort of this particular parameter, which may be a constant or a variable, and the position of the parameter in the domain of the operator. The example predicates above contain many variables (identifiers that are capitalized) because the specification in Figure 4.4 is meant to be a query and the query parameters must bind to the operator names and sorts of some stored component.

The choice of the arguments in the predicate expressions reflects some of the assumptions made about what constitutes a match between specifications. For instance, the number of parameters present in the operators must match precisely even though one can conceive of possibilities where an operator with two variable parameters, for example, could match to an operator with two variable parameters and a constant parameter. A rule used in finding a match is that all of the operators of the query must bind to unique operators in the component (the mapping is *injective*). This is based on the assumption that an engineer will not define identical semantics for any two operators in the same specification.

The order of the arguments in the predicate expressions is important for efficiency. Quintus Prolog© [Quin90] (the form of Prolog used for this portion of the implementation) hashes on the first argument of a predicate expression when that argument is bound. Using the range sort of an operator as the first argument of the operator predicate partitions the operators into smaller sets. Once a particular range sort variable has been bound, the search for subsequent matches will be very fast. The first argument of the argument predicate is the name of the operator because this variable is always bound in the operator predicate that precedes it. Thus, the search for appropriate arguments is also fast.

The set of predicate expressions for the specification in Figure 4.5 is:

```
operator(list, 0, nil)
operator(list, 2, cons)
argument(cons, bituple, 1)
argument(cons, list, 2)
operator( bituple, 2, make)
argument(make, nat, 1)
argument(make, nat, 2)
operator(nat, 1, length)
argument(length, list, 1)
operator(bituple,1, head)
argument(head, list, 1)
operator(list, 1, tail)
argument(tail, list, 1)
operator(list, 2, append)
argument(append, list, 1)
argument(append, list, 2)
operator(list, 1, reverse)
argument(reverse, list, 1)
operator(bool, 2, member)
argument(member, bituple, 1)
argument(member, list, 2)
operator(nat, 1, first)
argument(first, bituple, 1)
operator(nat, 1, second)
argument(second, bituple, 1)
operator(nat, 2, member)
argument(member, nat, 1)
argument(member, list, 2)
```

The predicate expressions derived from the specification in Figure 4.5 are treated as Prolog facts during the mapping phase. The predicate expressions from the specification in Figure 4.4 must be combined in some way to form a Prolog query. The next section covers the use of Prolog in the mapping process.

D. MAPPING QUERIES TO STORED COMPONENTS

1. Prolog as the Mapping Tool

Expansion and renaming are required to make a component an atomic unit for both storage in the software base and for comparison with the query by consistency algorithm. The Operator-definition to Prolog predicates transformation is necessary to provide the means to map a query to a candidate stored component using Prolog. To find a matching candidate in Prolog, the predicate expressions provided by the query are combined to form a Prolog rule. To that rule, additional predicate expressions are added to

ensure that all bound operator names are *unique* and that for each operator, all parameter positions are unique. The predicate expressions provided by a candidate component are used as a database of facts in an attempt to satisfy the query. Figure 4.6 shows an example of the Prolog query generated from the specification in Figure 4.4.

```
query(OutStream) :-
    operator(BITUPLE, 2, MAKE),
    argument(MAKE, nat, MAKE1),
    argument(MAKE, nat, MAKE2),
    unique([MAKE1, MAKE2]),
    operator(LIST, 0, NIL),
    operator(LIST, 2, CONS),
    argument(CONS, BITUPLE, CONS1),
    argument(CONS, LIST, CONS2),
    unique([CONS1, CONS2]),
    operator(nat, 1, LENGTH),
    argument(LENGTH, LIST, LENGTH1),
    unique([LENGTH1]),
    operator(LIST, 2, APPEND),
    argument(APPEND, LIST, APPEND1),
    argument(APPEND, LIST, APPEND2),
    unique([APPEND1, APPEND2]),
    unique([MAKE, NIL, CONS, LENGTH, APPEND]),
    store(OutStream, [MAKE, 2, BITUPLE, nat, MAKE1, nat,
        MAKE2, NIL, 0, LIST, CONS, 2, LIST, BITUPLE, CONS1,
        LIST, CONS2, LENGTH, 1, nat, LIST, LENGTH1,
        APPEND, 2, LIST, LIST, APPEND1, LIST, APPEND2,
        end]), fail.

query(OutStream) :- generic(G), store(OutStream, [generic, G]).
```

Figure 4.6 - Example Prolog Query

In the above example, the query in Figure 4.4 maps in four ways to the component of Figure 4.5. With some combinations, many mappings will be possible, but only one might be meaningful. This complicates the task of the overall query by consistency algorithm. For each candidate component, the algorithm must check every possible mapping. In the worst case, this task is worse than exponential in the number of operators with identical domain and range sorts. If one allows variables in stored components, which is the case when we store generic components, the problem is exacerbated. Chapter VII offers some suggestions to alleviate this problem. Figure 4.7 shows the mapping results

of having applied the query of Figure 4.6 to the Prolog facts listed above. The appendix lists the Prolog code that drives the mapping process.

```
[make,2,bituple,nat,1,nat,2,nil,0,list,cons,2,list,bituple,1,list,2,
    length,1,nat,list,1,append,2,list,list,1,list,2,end]
[make,2,bituple,nat,1,nat,2,nil,0,list,cons,2,list,bituple,1,list,2,
    length,1,nat,list,1,append,2,list,list,2,list,1,end]
[make,2,bituple,nat,2,nat,1,nil,0,list,cons,2,list,bituple,1,list,2,
    length,1,nat,list,1,append,2,list,list,1,list,2,end]
[make,2,bituple,nat,2,nat,1,nil,0,list,cons,2,list,bituple,1,list,2,
    length,1,nat,list,1,append,2,list,list,2,list,1,end]
[generic,[]]
```

Figure 4.7 - Mapping Results from Prolog Query

2. Checking Generic Consistency

A boon to the concept of reusable software is the *generic* component. The designers of CAPS expect the software base to contain a large number of generic components, although no predictions have been made as to what the percentage of generic components will be. It is therefore essential that the retrieval system have the capability to map queries to generic components. Figure 4.8 shows a specification for a generic component that models a *list* abstract data type.

```
***(operations nil cons car cdr)
obj GENERIC-LIST[X :: TRIV] is sort List .
 subsort Elt < List .
 op nil : -> List .
 op cons : Elt List -> List .
 op car  : List -> Elt .
 op cdr  : List -> List .
 var I, J : Elt .
 var L : List .
  eq car(cons(I,L)) = I .
  eq cdr(nil) = nil .
  eq cdr(cons(I,L)) = L .
endo
```

Figure 4.8 - OBJ3 Specification for a Generic List

Figure 4.9 shows the Prolog representation of the signature. Note that there are underscores (_) in some of the predicate expressions in Figure 4.9. The underscores represent Prolog variables that bind to any argument. Because of the flexibility inherent in this representation scheme, inconsistencies can arise during the mapping process, that

is, the variable that represents the single generic parameter (in this example) may bind to different sort values when the query is made[2]. If these bindings are inconsistent, the mapping is erroneous (a proper instantiation cannot be made) and that mapping must be discarded.

```
operator(list, 0, nil).
operator(list, 2, cons).
argument(cons, _,  1).
argument(cons, list, 2).
operator(_,  1, car).
argument(car, list, 1).
operator(list, 1, cdr).
argument(cdr, list, 1).
generic([[car, 0, x, 1], [cons, 1, x, 1]]).
```

Figure 4.9 - Prolog Predicate Expressions for an OBJ3 Specification of a Generic List

This check for generic consistency is made as the results of the Prolog query are scanned. In the current implementation, the generic parameters must map to *predefined* sorts. The system does not have the ability to extract features from a user query and use them to instantiate a stored generic component in order to perform QBC. This would be a useful extension and is examined in Chapter VII.

After the mapping and check for generic consistency are completed, then, assuming there is a mapping between the query and a candidate, the next step is to create a test set.

E. GENRATING A TEST SET

A test set is a set of terms that represent the equivalence classes of constructor ground terms that can be generated by the signature defined within an object. The test set has also been referred to as a *signature of constructors* [Gogu88]. Formally, a *signature,* $\Sigma = (S, f)$, consists of a set, S, of sorts and a set f of function symbols. The set f is the union of pairwise disjoint subsets C_s and $f_{w,s}$ where C_s is a set of constant symbols of sorts $s \in S$ and $f_{w,s}$ is a set of operator symbols with domain sorts $w \in S^+$ (one or more domain sorts) and range $s \in$

---

[2]In Prolog, the scope of a variable is limited to a single rule, fact, or query. For example, using the same variable *A* in place of the two underscores in Figure 4.9 would make no difference. Both *A*'s would be treated as different variables.

S [EM85]. The test set Π is a set of terms with arities that correspond to a subset of the operators in f.

The reason for generating a test set is to have a collection of terms from which to build example terms to submit to the axioms for reduction. In a normalized object, the set f may contain a large number of functions due to importation and instantiation. Only a subset of these functions, the signature of constructors, is needed for the test set ($\Pi \subseteq \Sigma$). For the predefined sorts that appear in the object, there are standard, predefined test set terms that are read from a file. Because predefined terms are used, it is not necessary to consider any function in f whose range sort is one of the predefined sorts. For example, the predefined terms for sort Nat are its constructors, 0 and succ(N). These terms serve as an inductive definition of natural numbers. The constant term 0 represents an equivalence class containing one term, whereas the term succ(N) represents an equivalence class containing all natural numbers not including 0. Since these terms represent all natural numbers, it is not necessary to have any other terms in the test set whose range sort is Nat.

For user defined sorts, however, the test set must include terms corresponding to all operators in f whose range sort is one of the user defined sorts, but constrained by the list of export operators in the comment block. By including all of these functions, the process guarantees that there is a complete description of the classes of terms that can be composed for each user-defined sort. Figure 4.10 shows the test set generated from the expansion of the specification in Figure 4.4 (See the Appendix for the definitions of the objects NAT, NZNAT, and BOOL.).

```
Zero:      0
Nat:       0
Nat:       succ(natconst1)
NzNat:     1
NzNat:     succ(nznatconst1)
Bool:      true
Bool:      false
List:      nil
List:      cons(!!!, listconst1)
List:      append(listconst1, listconst2)
BiTuple:   make(!!!, !!!)
```

Figure 4.10 - Test Set for List of BiTuple

After expansion and instantiation, the sorts used in the query for a list of bituple are Zero, Nat, NzNat, Bool, List, and BiTuple. The sets of constructors for Zero, Nat, NzNat,

and Bool are minimal, that is, no more and no fewer constructors are required to define all of the ground terms for those sorts. The set of constructors for BiTuple is a minimal set since there is only one constructor for sort BiTuple. The set of constructors for List is not minimal since only *nil* and *cons* are required but *append* is also included. It must be included since the process that selects the operators cannot know (without possibly examining the axioms) which constructors for user-defined sorts make a minimal set.

The exclamation points in some of the test set terms are *placeholders*. They represent arguments that must be filled when using the term to build an I/O list input. A placeholder will be filled with a term having the appropriate sort. Some of the test set terms also contain constants such as *natconst1* and *listconst1*. Constants within the terms serve two purposes: to represent an inductive definition of the sort (as in the case of succ(natconst1)) and to help avoid infinite term expansion when building the I/O list (as with append(listconst1, listconst2)).

F. BUILDING THE INPUT TERMS OF THE I/O LIST

An I/O list, Ω, is a list of terms that will be used as sample inputs to query and candidate component axioms. The I/O list is built from terms in the test set. The process of building an I/O list starts with an initial I/O list or template defined by the user-specified export operators in the ***(operations ...) comment block. The process then *expands* the template with terms from the test set. During expansion, care must be taken to avoid circularities, which can occur when an operator's range sort is identical to one of its domain sorts.

1. Initial Template and Expansion

The initial I/O list is a template of the user-specified export operators. The initial I/O list for the specification in Figure 4.4 is:

```
nil
cons(!!!, !!!)
make(!!!, !!!)
append(!!!, !!!)
length(!!!)
```

Each operator exported by the user occupies one place in the list and each parameter for operators with parameters is filled with a placeholder. Just as in the test set, a placeholder represents an expansion slot that will be filled by a term of the appropriate sort.

To expand the I/O list, the process begins at the front of the list and scans for a term containing a placeholder. When the term and placeholder are found, n new terms are created, where n is equal to the number of terms in the test set whose range sort matches the sort of the placeholder. The new terms created are identical to the term containing the placeholder. In each of the new terms, the placeholder is replaced by a test set term having the appropriate range sort. These expanded terms are then appended to the end of the I/O list. The process then deletes the original term containing the placeholder from the I/O list and moves on to check the next term. The process continues until all terms containing placeholders have been expanded and all placeholders have been eliminated.

The result of this expansion process is a list of terms that collectively (and exhaustively) represent each export operator and the classes of arguments it may have. The following terms are a sample from the I/O list for the query in Figure 4.4.

```
nil
make(0, 0)
make(0, succ(natconst1))
make(succ(natconst1), 0)
make(succ(natconst1), succ(natconst1))
cons(make(0, 0), nil)
cons(make(0, 0), append(listconst1, listconst2))
cons(make(0, succ(natconst1)), nil)
cons(make(0, succ(natconst1)), append(listconst1, listconst2))
append(nil, nil)
append(nil, append(listconst1, listconst2))
append(append(listconst1, listconst2), nil)
append(append(listconst1, listconst2), append(listconst1, listconst2))
length(nil)
length(append(listconst1, listconst2))
length(cons(make(natconst1, natconst1), listconst1))
```

The entire I/O list contains 68 terms. Each term is comprised solely of operators or constant constructors (OBJ3 cannot perform reductions on terms containing variables). The number of terms in the I/O list depends on many factors including the number of operators in the export list, the number of parameters within the operators, the number of test set terms that correspond to the sorts of each parameters, and the rules for avoiding circularities during term expansion.

2. Checking for Circularities

In the process of expanding the I/O list, Ω, it is possible to encounter situations where expansion would continue ad infinitum. There is a single rule that is used to avoid

this situation. Suppose a term ω (from the I/O list) contains a placeholder and the parent of that placeholder is ω_p, that is, some operator within ω (In many cases $\omega = \omega_p$.). Then if a term π (from the test set) will be used to expand the placeholder in ω_p, then π must not contain a placeholder with the same range sort as ω_p or with the same range sort as ω. If either situation is encountered, the placeholder in π is replaced by a *constant* of the appropriate sort before π is used to expand ω. Any constants used in the terms in the I/O list must be declared as constant operators within the module. This task is accomplished in the next phase of the process, that of generating output terms in the query domain.

G. GENERATING OUTPUT TERMS IN THE QUERY

1. Reductions in the Query Domain

Having created the input half of the I/O list, we submit the terms to the axioms of the query using the OBJ3 environment to determine output results. OBJ3 uses term rewriting to reduce each input term to a normal form, that is, a form where no further reductions are possible. The corresponding outputs to the above list of inputs are:

```
nil
make(0, 0)
make(0, succ(natconst1))
make(succ(natconst1), 0)
make(succ(natconst1), succ(natconst1))
cons(make(0, 0), nil)
cons(make(0, 0), append(listconst1, listconst2))
cons(make(0, succ(natconst1)), nil)
cons(make(0, succ(natconst1)), append(listconst1, listconst2))
nil
append(listconst1, listconst2)
append(listconst1, listconst2)
append(append(listconst1, listconst2), append(listconst1, listconst2))
0
length(append(listconst1, listconst2))
sum(1, length(listconst1))
```

Note that many of the outputs are identical to the inputs. This will be the case when the input term is composed solely of constructor operators having no corresponding axioms, such as:

```
nil  and
cons(make(0, 0), nil).
```

This is also the case when the term contains constants that cannot be reduced by axioms, such as:

length(append(listconst1, listconst2)).

The fact that no rewriting was performed on those terms is just as important to the method as the knowledge obtained from a term reduction. No reduction means that in the domain of the query, the term is only syntactically defined. If, however, in the component domain, the same term *is* reduced then the process will have detected a dichotomy between the specifications.

2. Parsing the Results

As the terms are reduced by the OBJ3 rewrite system, the normal form of each term is written to a file. In order to read the terms from the file and store them in the I/O list, it is necessary to parse them. Since the terms are in prefix form, this task is simplified. A parser parses each output and stores it in the output half of the I/O list corresponding to the term's input. The I/O list is now *complete*, that is, both the inputs and outputs have been determined. The system may now perform semantic matching with the candidate component.

H. OUTPUTS IN THE CANDIDATE COMPONENT DOMAIN

Given a complete I/O list in the query domain and a set of mappings to the candidate component, the system performs (for each map) I/O list transformation, followed by term rewriting in the component domain, and inductionless induction to derive a score for the map.

1. I/O List Transformation

The names of the operators, the names of sorts, and the positions of parameters in the signature of the query will most likely be different than the corresponding operators, sorts, and parameters in the candidate component. Before rewriting of the I/O list terms can take place in the domain of the candidate component, the terms must be transformed to the domain of the candidate using one of the mapping functions. Since I/O list term output comparison will be performed in the domain of the candidate component, it is necessary to transform both the inputs and outputs to the component domain.

Formally, an I/O list is a set of terms Ω_q, where each ω_{qi} will be used as an input term to the axioms of the query. Reduction generates the term's normal form, ω'_{qi}. These

inputs and outputs must be mapped to the component domain using a mapping function h: $\Sigma_q \rightarrow \Sigma_c$ that maps terms derived from the signature of the query to terms from the signature of the stored component, yielding $h\omega_{q_i}$ and $h\omega'_{q_i}$. The reduction of the input, $h\omega_{q_i}$', and the comparison to the transformed query domain output, $h\omega'_{q_i}$', are performed simultaneously using a theorem proving method known as inductionless induction.

2. Inductionless Induction

Inductionless induction is a theorem proving method "...which uses purely equational reasoning (in the form of rewrite-rules) to prove theorems valid in an initial algebra that would normally have to be proved by induction." [MG85, p. 524] A Σ-algebra is *initial* in a class of Σ-algebras if and only if there is one and only one Σ-homomorphism from that algebra to all other Σ-algebras in the same class [MG85]. All instantiated object specifications in OBJ3, that is, those that are executable, are initial [Mese91].

An inductionless induction procedure is a built-in feature of OBJ3. Terms are compared by asking the system to reduce:

term1 == term2.

Since the system described in this dissertation uses prefix format for functions, the syntax actually used is:

==(term1, term2).

For each transformed I/O list pair, a term comparison is performed by substituting the transformed input for term1 and the transformed output for term2. OBJ3 then performs reductions on term1 to reduce it to normal form and then compares term1 and term2 for equivalence[3]. Operator attributes, such as associativity and commutativity, are applied in the check for equivalence.

The final result of a term comparison will be one of two terms: true or false. If the result is true, then the terms have been proven equivalent. This means that with respect to that term comparison, the two specifications are *behaviorally equivalent*. The component's behavior satisfies the query's requirement. A false result means that the terms could not be proven equivalent. This result suggests that the two specifications are

---

[3]OBJ3 actually attempts to reduce both term1 and term2 to canonical form before comparing the terms for equivalence. If, however, OBJ3 is allowed to reduce term2, which is the transformed normal form of term1 from the query domain, then term2 may be modified by the axioms of the component domain and would therefore no longer be a true representation of the semantics used to reduce it in the query domain. Thus, the comparison of term1 and term2 would be meaningless. The OBJ3 proof mechanism was altered to prevent reductions on term2. This change to OBJ3 is given in the Appendix.

not behaviorally equivalent with respect to that term. The proof process is a *semi-decision* procedure for determining the equivalence of two terms. The true and false results are used in the scoring method described in the next section.

3. Interpreting the Results

The result of submitting each transformed I/O pair to the inductionless induction procedure is a term with the value true or false. The semantic matching system uses a simple scoring mechanism, based on these true and false results, to select the best map for a given component and to ultimately rank order a set of components. The score given to a particular map is the ratio of the number of I/O pairs that reduce to true to the total number of I/O pairs reduced. For example, if 50 I/O pairs were reduced and the result was true for 40 of them, the score for that map would be 80%. Once all of the maps have been tried, the best score is used as the component's score in comparing against other candidate components.

There are other factors that could be used in scoring that have not been implemented. These are described in Chapter VII. Also, a threshold value could be assigned to eliminate some components from further consideration. The use of a threshold is not implemented, but is described as an extension to the system in Chapter VII.

I. VERIFICATION OF THE MODEL FOR RETRIEVAL

The system described in the preceding sections has been implemented. The implementation is described in Chapter V and examples are given in Chapter VI. In order to provide empirical results of system usage, a large software base would be required, but is not yet available. Therefore, this section presents a formal model of the system with respect to the forms of specifications, the test set, the I/O list, and the inductionless induction proof technique. Figure 4.11 illustrates the formal model of query by consistency. The numbers in the diagram of Figure 4.11 are explained in Table 4.1 which describes the diagram. The numbers are also referenced in the sections that follow.

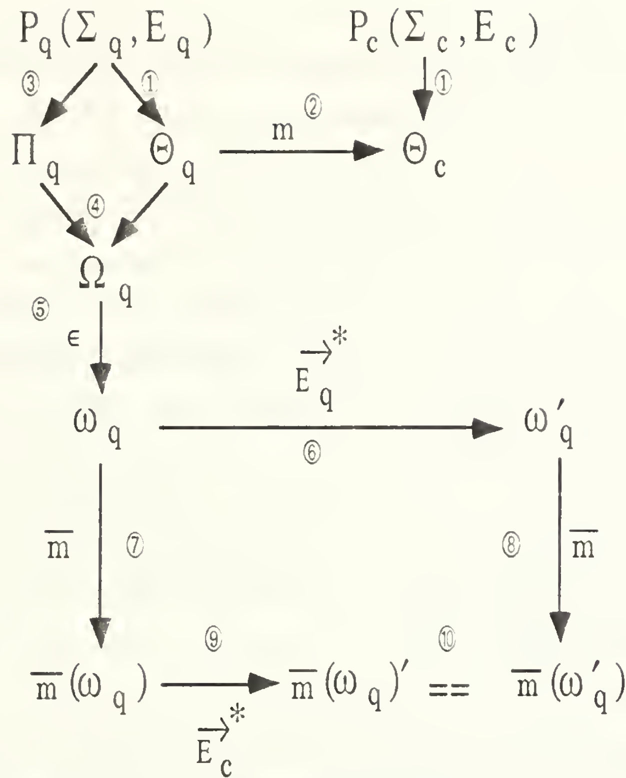


Figure 4.11 - Formal Model of Query by Consistency

TABLE 4.1 - EXPLANATION OF FIGURE 4.11

Step	Explanation of Function
①	Export signatures are derived from the query and stored component presentations.
②	A mapping, h, between the export signatures is determined.
③	A test set (signature of constructors) is derived from the query presentation.
④	An I/O List is generated from the export signature and the test set.
⑤	Each term in the I/O list will be processed in steps ⑥ through ⑩.
⑥	An I/O list term is reduced in the query domain.
⑦	An I/O list term is mapped to the stored component domain using an augmented mapping function.
⑧	The result of the reduction in step ⑥ is mapped to the component domain using an augmented mapping function.
⑨	The term mapped to the component domain in step ⑦ is reduced in the component domain.
⑩	The term resulting from the operations in steps ⑧ and ⑨ are compared for equivalence using inductionless induction.

1. The Specification Model

The formal specifications for both stored components and queries are written in OBJ3. Each object specification is considered a presentation $P(\Sigma, E)$. In this dissertation, a presentation for a *query* will be subscripted with a q, $P_q(\Sigma_q, E_q)$, and a presentation for a stored candidate *component* will be subscripted with a c, $P_c(\Sigma_c, E_c)$. A single presentation represents the query and a set of presentations represent the components that were retrieved by the syntactic search (see section III.D.1). Each presentation consists of a signature, Σ, and a set of axioms, E. The signature, Σ=(S, f), consists of a set, S, of sorts and a set, f, of function symbols. The set f is the union of pairwise disjoint subsets C_s and $f_{w,s}$ where C_s is a set of constant symbols of sorts $s \in S$ and $f_{w,s}$ is a set of operator symbols with domain sorts $w \in S^+$ (one or more domain sorts) and range $s \in S$ [EM85]. The axioms, E, define the abstract, denotational semantics for the object. The language is *executable* by treating the equations as rewrite rules, substituting matched instances of left-hand sides with corresponding right-hand sides.

2. Normalization

Normalization extends the presentation or definition of an algebra by adding another presentation, $P'(\Sigma', E')$, to the given presentation. It is assumed that all module importation is performed with the **protecting** statement (see Section 3.B.3.d) resulting in "no junk" and "no confusion." [MG85, p.464] Object *extension* then, is simply the union of two or more presentations, that is,

$$P \cup P' = P''(\Sigma \cup \Sigma', E \cup E')$$

where P" is the presentation of the new expanded object. In Figure 4.1, the presentations at the top of the figure are considered normalized. It is assumed, before normalization, that the specification to be normalized is syntactically correct and correctly models the behavior of some Ada software component (either sought or to be stored). It is also assumed that after normalization, a specification to be used for a query will be fully instantiated object. The system does not currently perform any checks to ensure these assumptions are satisfied.

3. The Export Signature

When storing a component in the software base or submitting a query, the user must augment the specification of the object with an OBJ3 comment block that specifies the operators that the object will export. For example, if the user queries for a stack abstract data type, the OBJ3 comment block might be:

```
***(operations empty push pop top)
```

The operators specified in the comment block must be identical to the symbols used in the signature of the object to define those operators. Hence, an *export signature* , Θ,is a subset of the signature Σ, $\Theta \subset \Sigma$, where each operator symbol in Θ is a member of the set of operators specified in the comment block.

Step ① in Figure 4.11 shows the derivation of the export signature for both presentations.

4. Mapping a Query to a Stored Component

The export signatures are used to determine the mappings from the query to the stored component. In order for a component to satisfy a query, there must be a Θ-homomorphism, m: $\Theta_q \rightarrow \Theta_c$, such that:

$$mf_q(\theta_1 ... \theta_n) = f_c(m(\theta_1)...m(\theta_n))$$

where θ_1 through θ_n are the individual operators in Θ. Furthermore, the homomorphism must be *injective*, that is, each operator of the query maps to a unique operator in the component. Research by Goguen and Meseguer [Gogu88, GM85] provide the definitions of mapping functions between many-sorted algebras. To identify a mapping function, it must be demonstrated that the correlation between sorts and operator symbols satisfy certain properties or rules. The rules for identifying a mapping function between two export signatures are:

1. There must be an injective mapping between the operator symbols in Θ_q and the operator symbols in Θ_c and a mapping between their respective domain and range arguments (using rules 2 and 3).

2. There must be a bijective mapping between the domain sorts of a query operator to the domain sorts of a candidate component operator (using rules 4 and 5).

3. The range sort of a given query operator must map to the range sort of a candidate component operator (using rules 4 and 5).

4. A predefined sort in the query (treated as a constant) must map to an identical predefined sort in the stored component.

5. A user-defined sort in the query (treated as a variable) may map to either a predefined sort or author-defined sort in a stored component.

6. All bindings of user-defined sorts in the query to sorts in the candidate component must be consistent.

Figure 4.12 shows the Prolog database generated from the specification in Figure 4.5. and Figure 4.13 repeats (for convenience) Figure 4.6, the Prolog query generated from the specification in Figure 4.4.

```
operator(list, 0, nil).
operator(list, 2, cons).
argument(cons, bituple, 1).
argument(cons, list, 2).
operator( bituple, 2, make).
argument(make, nat, 1).
argument(make, nat, 2).
operator(nat, 1, length).
argument(length, list, 1).
operator(bituple,1, head).
argument(head, list, 1).
operator(list, 1, tail).
argument(tail, list, 1).
operator(list, 2, append).
argument(append, list, 1).
argument(append, list, 2).
operator(list, 1, reverse).
argument(reverse, list, 1).
operator(bool, 2, member).
argument(member, bituple, 1).
argument(member, list, 2).
operator(nat, 1, first).
argument(first, bituple, 1).
operator(nat, 1, second).
argument(second, bituple, 1).
operator(nat, 2, member).
argument(member, nat, 1).
argument(member, list, 2).
```

Figure 4.12 - Example Prolog Database

```
query(OutStream) :-
    operator(BITUPLE, 2, MAKE),
    argument(MAKE, nat, MAKE1),
    argument(MAKE, nat, MAKE2),
    unique([MAKE1, MAKE2]),
    operator(LIST, 0, NIL),
    operator(LIST, 2, CONS),
    argument(CONS, BITUPLE, CONS1),
    argument(CONS, LIST, CONS2),
    unique([CONS1, CONS2]),
    operator(nat, 1, LENGTH),
    argument(LENGTH, LIST, LENGTH1),
    unique([LENGTH1]),
    operator(LIST, 2, APPEND),
    argument(APPEND, LIST, APPEND1),
    argument(APPEND, LIST, APPEND2),
    unique([APPEND1, APPEND2]),
    unique([MAKE, NIL, CONS, LENGTH, APPEND]),
    store(OutStream, [MAKE, 2, BITUPLE, nat, MAKE1, nat,
        MAKE2, NIL, 0, LIST, CONS, 2, LIST, BITUPLE, CONS1,
        LIST, CONS2, LENGTH, 1, nat, LIST, LENGTH1,
        APPEND, 2, LIST, LIST, APPEND1, LIST, APPEND2,
        end]), fail.

query(OutStream) :- generic(G), store(OutStream, [generic, G]).
```

Figure 4.13 - Example Prolog Query

Operator predicate expressions map range sorts, number of domain arguments, and operator names. *Argument* predicate expressions map argument sorts and positions given an operator name. The *unique* predicate expression ensures that the elements of a given list are all unique.

Mapping rule 1 is satisfied since the operator names in the query are variables (Prolog variables begin with a capital letter), the operator names in the candidate component are constants, and the query ensures, using the *unique* predicate, that all bindings to component operator symbols are unique. Mapping rule 2 is satisfied using the second argument of the *operator* predicate, the *argument* predicates, and the *unique* predicate. The operator predicate maps range sorts of the operators, satisfying mapping rule 3. For mapping rule 4, predefined sorts in the query and stored component are represented as Prolog constants and must be identical in order to map. For mapping rule 5, a user-defined sort in the query is represented as a Prolog variable and will map to either a predefined sort or author-defined sort in the stored component since they are represented as Prolog constants.

Mapping rule 6 is the final challenge. The Prolog query uses the same variable name throughout the query to represent user-defined sorts. If the query succeeds, then the binding to that variable must be consistent throughout. However, if the candidate component contains generic sorts, which are represented as anonymous Prolog variables (_), the mapping to these sorts may be inconsistent. In other words, two different sorts in the query could map to the same generic sort in the stored component. The bindings to the generic sorts must be checked after the Prolog query is complete. A procedure called Check_Generic_Consistency performs this task and discards the maps that are inconsistent.

The transformation of the export signatures to Prolog and the resulting Prolog query results correctly implement the requirement for an injective homomorphism between two export signatures. Step ② in Figure 4.11 models this process.

5. The Test Set

The test set, Π_q, is a subset of the query signature, Σ_q, and is called a *signature of constructors*. A signature of constructors for an algebra A "...is a subsignature $\Pi \subseteq \Sigma$ such that the unique Π-homomorphism $T_P \rightarrow A$ is surjective." [Gogu88, p.11] In other words, every unique term defined by the algebra A can be defined using a subset of the operators in A. For example, a signature of constructors for NAT, a sort representing the natural numbers, would be:

op zero : -> Nat .
op succ : Nat -> Nat .

All natural numbers can be represented with these two operators and no other constructors are required. “Every presentation has a signature of constructors.” [Gogu88, p. 11] To derive the signature of constructors from a query specification, the process must consider all sorts used in the specification. Each predefined sort has a predefined signature of constructors that is added to Π_q. For each user-defined sort in the query, all operators whose range sort is one of the user-defined sorts are also included in Π_q. Since Π_q is the union of signatures of constructors for all predefined sorts and all constructors of user defined sorts, it must therefore be a complete signature of constructors for the specification. Step ③ in Figure 4.11 identifies the test set construction task.

6. The I/O List

The I/O list, Ω_q, is a list of terms constructed from the export signature, Θ_q, and the test set, Π_q. The initial I/O list is modeled after the export signature, that is, for each operator defined in the export signature, a term is created with the exact same structure and added to the I/O list. For example, the operator

op cons : Nat List -> List .

from the export signature would take the following form in the I/O list:

$\text{cons}(!!!_{\text{Nat}}, !!!_{\text{List}})$

The *cons* term has two unbound arguments that are expanded later with subterms of sort Nat and List. The subterms used for expansion are modelled after operators in the test set. The I/O list expansion process is explained in Section IV.F.

After full expansion, the I/O list consists of terms whose outermost function is a member of the export signature and whose arguments are constructor ground terms derived from the test set. The process is complete in that every argument of every export operator uses every instance of the constructors for that sort. This affords the process the ability to thoroughly exercise the semantics of each export operator. The I/O list construction is identified at step ④ in Figure 4.11.

7. Reduction in the Query Domain

“OBJ3 does reduction, that is left-to-right deduction, by treating the equations in [a presentation] P as rewrite rules.” [Gogu88, p. 9] The purpose of the reduction step in the semantic matching process is to exercise the semantics of a specification by submitting the terms of the I/O list to the axioms of the specification for reduction. That is:

$$\forall i(1 \le i \le |\Omega|) : \omega_{qi} \in \Omega_q \Rightarrow \omega_{qi} \rightarrow^*_{E_q} \omega'_{qi}$$

The above expression states that for each term ω_q in Ω_q, rewriting using axioms E_q yields ω'_q. The symbol for rewriting, $\rightarrow^*$, indicates that the result is obtained with 0 or more rewrites. The theory behind term rewriting has been well-researched [HO80] and proofs for the term rewriting process in OBJ3 can be found in the work of Goguen [Gogu88]. It suffices to say here that if one or more rewrite rules are applied to the input term ω_{qi} to yield output term ω'_{qi}, then the structure of term ω_{qi} has been altered by the *semantics* of the specification. The significance of the transformation is that the input and output together model part of the *behavior* of the specification. This is precisely what the research that is the focus of this dissertation hopes to capture, that is, *concrete representations of specification semantics that can be compared to one another.* The reduction process for I/O list terms in the query domain is illustrated in steps ⑤ and ⑥ of Figure 4.11.

8. Mapping Terms

Section 4 above described the process for determining a mapping, m, between the query export signature Θ_q and the stored component export signature Θ_c. The mapping function m is sufficient to map the export operators of the two specifications. In the course of generating the I/O list, however, terms derived from the predefined operators (used in the test set) as well as auxiliary constants were used to expand the terms derived from the export signature. Therefore, the mapping function m may not be sufficient to map all terms in the I/O list from the query domain into the stored component domain. It is necessary, therefore, to augment the mapping to map constants as well as subterms derived from predefined operators in the test set. The augmented mapping function is $\overline{m}$.

The purpose of this step in the overall process is to use the function h' to transform the terms in the I/O list to the component domain. Hence:

$$\forall i(1 \le i \le |\Omega|): (\overline{m}\Omega_q(\omega_{qi}) = \overline{m}\omega_{qi}) \wedge (\overline{m}\Omega_q(\omega'_{qi}) = \overline{m}\omega'_{qi})$$

The mapping function $\overline{m}$ maps completely each ω_{qi} in the query domain to $\overline{m}\omega_{qi}$ in the component domain. The function is *not* complete, however, with respect to ω'_{qi}, that is, the reduced form of the input term. The result of term rewriting may be a term composed of hidden operators for which there is no map to the stored component specification. In this case the term is mapped "as is". When this term is compared to the component output, the result will be most likely be false. There is a slim possibility that the stored component uses an identical hidden operator in name and meaning, and that the comparison of the

two output terms will yield true. The process does not make the transformation in the hope that this occurs. On the contrary, the transformation is allowed because false results are important in scoring, which measures the extent of semantic similarity. Steps ⑦ and ⑧ in Figure 4.11 identify the mapping process.

9. Reduction in the Component Domain

Each input term, ω_{q_i}, from the I/O list that is mapped to the component domain is reduced by t he component axioms:

$$\forall i(1 \leq i \leq |\Omega|): \omega_{q_i} \in \Omega_q \Rightarrow \overline{m}\omega_{q_i} \rightarrow^{*}_{E_c} \overline{m}\,\omega_{q_i}'$$

If one or more rewrites are performed on a term, then the semantics of the specification has affected the structure of the term. The result is thus a concrete representation of a portion of the behavior of the specification. Step ⑨ in Figure 4.11 identified this process.

10. Comparing Terms and Scoring

a. Comparing Terms

The final step in the query by consistency process is to compare the output terms from the query and the candidate component specification. Herein lies the heart of the query by consistency method. Two sets of normalized terms must be compared for syntactic identity. The test for consistency checks for a property called *behavioral equivalence* . Behavioral equivalence ($\cong$) on terms is defined as follows:

$$(t_c \cong t_q) \leftrightarrow (t_c' == \overline{m}(t_q'))$$

The formula above states that two terms t_c and t_q are behaviorally equivalent if their normal forms are syntactically equivalent. Behavioral equivalence for specifications is defined as follows:

$$\forall t_q \in P_q (\exists t_c \in P_c \wedge t_c \cong t_q) \Rightarrow P_q \cong P_c$$

The formula above states that two specifications P_q and P_c, interpreted as sets of terms, are behaviorally equivalent if for every term in P_q there exists a behaviorally equivalent term in P_c. The query by consistency method searches for a candidate component that is behaviorally equivalent to a given query. Under certain circumstances, query by consistency *guarantees* that a stored component satisfies the requirements stated in a query. Given that all of the I/O list terms are behaviorally equivalent (using the

commuting diagram in Figure 4.12) and that the depth of the terms in the I/O list is sufficient to represent the depth of the terms used in the axioms, the proof must show that each axiom of the query is satisfied in the candidate specification. Each axiom in the query, L = R, must be satisfied in the axioms of the candidate specification.

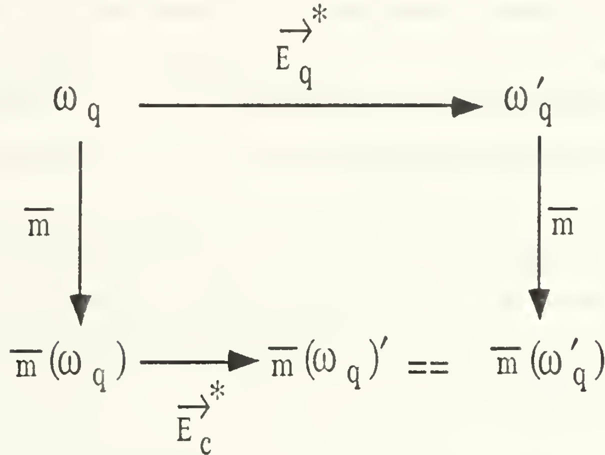


Figure 4.12 - Commuting Diagram

Given:

- All normalized terms in the I/O List of a query Pq are equivalent to corresponding (mapped) terms in the domain of a candidate Pc:

$$\forall t_q \in \Omega_q \ \exists t_c \in P_c \ [t_c = \overline{m}(t_q) \wedge (t'_q = \overline{m}(t_q)')]$$

- Query axioms of the form L = R
- The depth of the terms in the I/O list is sufficient to represent the terms used to define the axioms[4].

Prove: $\overline{m}(L) = \overline{m}(R)$

1.	$\overline{m}(L) = \overline{m}(L)'$	reduction of L in the candidate domain
2.	$\overline{m}(L)' = \overline{m}(L')$	by the commuting diagram
3.	$L' = R'$	reduction of L and R in the query domain
4.	$\overline{m}(L') = \overline{m}(R')$	by substitution of R' for L'

[4]The depth of the axioms in the I/O list is easily controlled by associating an attribute with each placeholder to monitor expansion depth. The placeholder attribute is not implemented in the current version of the system.

5. $\overline{m}(R') = \overline{m}(R)'$ by the commuting diagram

6. $\overline{m}(R)' = \overline{m}(R)$ reduction of R in the candidate domain

7. $\therefore \overline{m}(L) = \overline{m}(R)$ QED

The implications of the above proof are significant. If query by consistency reports a complete equivalence with respect to the terms in the I/O list, the user has a guarantee that the candidate component satisfies the stated requirements of the query. In addition, the result of the proof leads to the development of a scoring heuristic for comparing degree of behavioral equivalence.

b. Scoring

When two specifications do not have complete equivalence with respect to the I/O list, query by consistency may be used as a *heuristic* method to measure the *degree* of behavioral equivalence. The measure of behavioral equivalence is attained via a scoring mechanism that works as follows:

$$x := 0$$
$$\forall i(1 \leq i \leq |\Omega|): \overline{m}\omega_{q_i}' == \overline{m}\omega'_{q_i} \rightarrow true \Rightarrow x := x + 1$$
$$score := x/|\Omega|$$

Simply stated, the degree to which a stored component satisfies a query's requirement is the ratio of the number of successful term comparisons to the total number of term comparisons. The scores are used to select the best map from a number of possible mappings for a given candidate and to rank order candidates. Examples of the scoring are shown in the next chapter.

J. SUMMARY

This chapter describes a method of comparing normalized algebraic specifications for semantic similarity using a method called query by consistency (QBC). The implementation of the method consists of two executable programs, one to normalize specifications accompanying components to be stored in the software base, and one to match or compare a query specification with a candidate component specification.

The normalization process expands a specification and transforms the interface of the specification into a set of Prolog predicate expressions. The Prolog predicate expressions are then used to find a mapping between the export operators of the respective specifications.

The matching process creates a test set from the query signature, and an I/O list from the export signature and the test set. The terms in the I/O list are reduced in the domains of the query and candidate specifications and the results are compared using inductionless induction.

The fundamental premise of this dissertation is that the terms from the I/O list, when reduced in the domains of the query and the candidate, provide concrete representations of specification semantics that can be compared to one another for equivalence. From the set of comparisons a measure of semantic similarity may be computed and used to rank order candidate components based on how well they satisfy the semantic requirements of the query. The last section of this chapter formally describes the query by consistency model and offers a proof of the fundamental premise.

V. IMPLEMENTATION AND EXAMPLES

A. INTRODUCTION

This chapter describes the implementation details of the normalization and matching subsystems that make up the query by consistency method. The emphasis here is on showing that the data structures and processes presented are reasonable rather than showing that they are efficient. This chapter does not offer advice on whether to use any particular mechanisms in the implementation. There is also no comparison of the efficiency of this method to that of other methods. The primary intent is to provide information on the current implementation to lay a foundation for extending the research.

The body of the chapter is divided into five sections. The first section summarizes the programming languages and systems used to implement the programs. The second reviews the processes used for normalization and the third summarizes the processes used for semantic matching. The fourth section describes the primary data structures used in the implementation. The fifth section gives examples to demonstrate the capabilities of the system. The chapter ends with a summary.

B. IMPLEMENTATION LANGUAGES

A combination of four programming languages are used in this implementation of query by consistency: Ada, OBJ3, Lisp, and Prolog. The primary language used is Ada. The two executable programs, for normalization and matching, are Ada executables. The Ada compiler used is Verdix 6.0 [Verd91]. OBJ3 is used to write specifications, while the OBJ3 run-time system is used for expansion, term rewriting, and inductionless induction. OBJ3 is provided by SRI International [SRI88]. Since it was necessary to modify some of the OBJ3 source code, which is written in Common Lisp, some Lisp functions comprise a portion of the implementation. Quintus Prolog© [Quin90] is used to map specifications to one another.

C. NORMALIZATION

Figure 5.1 shows the basic structure of the normalization subsystem.The rectangular boxes represent processes. The names in the boxes are the actual names of the processes in

the Ada implementation except for blocks containing "OBJ3" or "Prolog", which represent a calls to those respective systems. See the Appendix for actual Ada the source code.

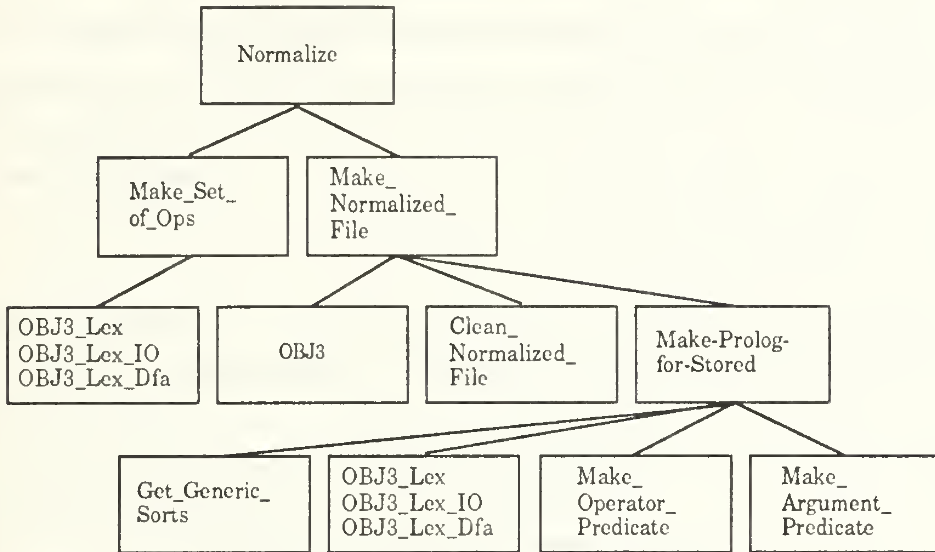


Figure 5.1 - Structure of the Normalization Subsystem

The normalization subsystem normalizes specifications that accompany components destined for storage in the software base. The process is called by the software base management system when a user wishes to *store* a reusable component. The process is an Ada executable invoked with the following command line:

normalize some_object.obj

where *some_object.obj* is the name of the file containing the specification. Any file name may be given but it must have the .obj extension. The process creates a file called *some_object.obj.norm* that is subsequently stored away with the reusable component by the software base management system.

The main functions of the normalization system are to expand the specification, transform its export signature into Prolog, and create the .norm file containing the normalized specification. The procedure Make_Set_of_Ops uses a lexical analyzer[1] to search the specification for the OBJ3 comment block containing the export operations and creates a *set* containing the operator names. Make_Set_of_Ops uses a lexical analyzer to

---

[1]All lexical analyzers and parsers used in the implementation were generated using AFLEX Version 1.1 [Self90] and AYACC Version 1.0 [TTS88].

process the OBJ3 specification. The procedure Make_Normalized_File invokes OBJ3 to expand the specification and write the details of the expansion to a file. The procedure Clean_Normalized_File removes extraneous OBJ3 output from the file.

The procedure Make_Prolog_for_Stored performs interface normalization. It uses the set of export operations (from Make_Set_of_Ops) and a lexical analyzer to process the signature of the specification, creating the operator and argument predicates that represent the export signature. These predicates and supporting information about generic parameters are written to the .norm file.

D. MATCHING

Figure 5.2 shows the top level structure of the matching subsystem.

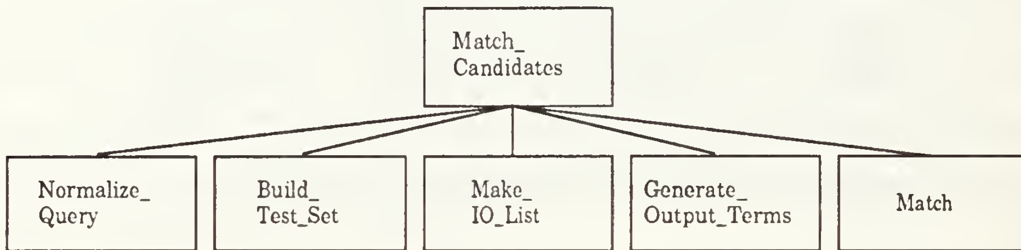


Figure 5.2 - Structure of the Matching Subsystem

Match_Candidates is an executable Ada process that is called by the software base management system when the user queries the software base. It is invoked with the following command line:

match-candidates my_query.obj candidates scores

The argument *my_query.obj* is the name of a file containing the query specification. The argument *candidates* is the name of the file containing a list of the file names of candidate component specifications. The argument *scores* is the name of the file to which the process writes the score received by each candidate.

The first four subprocesses beneath Match_Candidates (in Figure 5.2) are called only once. The last subprocess, Match, is called once for each candidate component in the file *candidates*. These five subprocesses are described in the following sections.

1. Normalize Query

The structure of the Normalize_Query subprocess is nearly identical to the structure of the normalization process shown in Figure 5.1, so it is not repeated here. The differences in the processes lie in the procedures that perform interface normalization, that is, generate the Prolog. In query normalization the Prolog created is a Prolog *query* rather than a Prolog *database*.

2. Build Test Set

Figure 5.3 shows the high level structure of the Build_Test_Set subprocess. In the course of building a test set, the process first creates a *set* of the sorts used in the query specification by scanning the .norm file with a lexical analyzer. For each *predefined* sort in the set, predefined test set terms are extracted from a file and added to the test set. The procedure Get_Predefined_Terms uses a lexical analyzer to scan a file containing the definitions of predefined term and uses a subprocedure called Make_Term to formulate terms from the stored definitions.

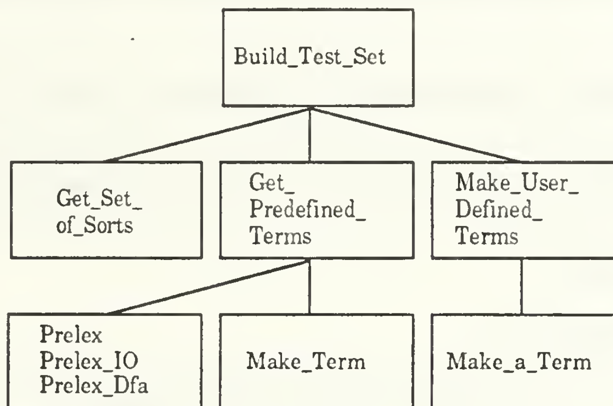


Figure 5.3 - Structure of the Build_Test_Set Subprocess

For *user-defined* sorts in the set of sorts, Build-Test-Set calls Make_User_Defined_Terms, which scans the query's operator definition sequence for query operators whose range sorts are among the user-defined sorts. The procedure Make_a_Term generates a term for each appropriate operator and adds it to the test set.

3. Make IO List

Figure 5.4 shows the high level structure of the Make_IO_List subprocess. Make_IO_List uses the test set, the set of export operations, and the sequence of operator

definitions to generate the list of input terms, which comprise the input side of the I/O list. Make_IO_List first calls Make_Template, which creates the initial I/O list. Make_IO_List then traverses the I/O list scanning for placeholders. When the process encounters a placeholder, it performs term expansion and then continues.

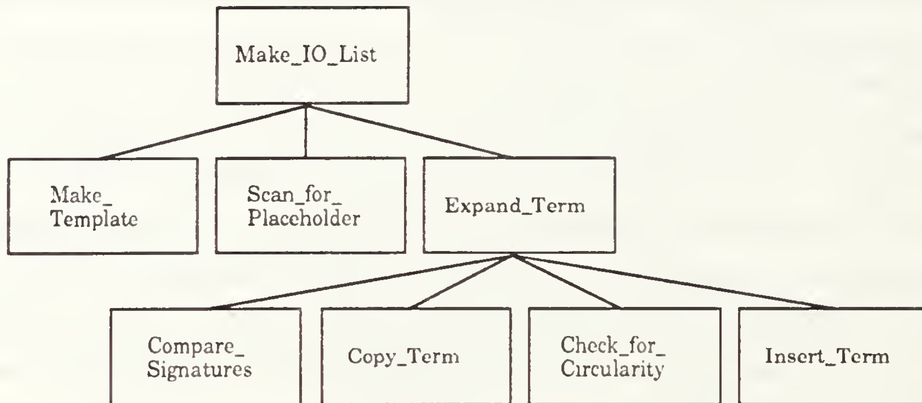


Figure 5.4 - Structure of the Make_IO_List Subprocess

Given two terms, **A** and **B**, the Expand_Term procedure inserts the expansion term, **B**, into the first placeholder position within term **A**, appends the new expanded term, **A'**, to the end of the I/O list, and deletes **A** from the I/O list. In performing this task, expand term uses utilities to compare term signatures, copy terms, check for circularities, and to insert one term into another.

4. Generate Output Terms

Figure 5.5 shows the high level structure of the Generate_Output_Terms subprocess. The Generate_Output_Terms subprocess invokes an OBJ3 process to reduce the input terms in the I/O list using the axioms in the query specification. The result of this process is a file containing the term reductions. Generate_Output_Terms then calls Clean_Output_File to remove extraneous OBJ3 output from the file. The Term_Parser procedure then parses the terms in the file using a lexical analyzer and parser. As the output terms are parsed, they are placed in the output side of the I/O list.

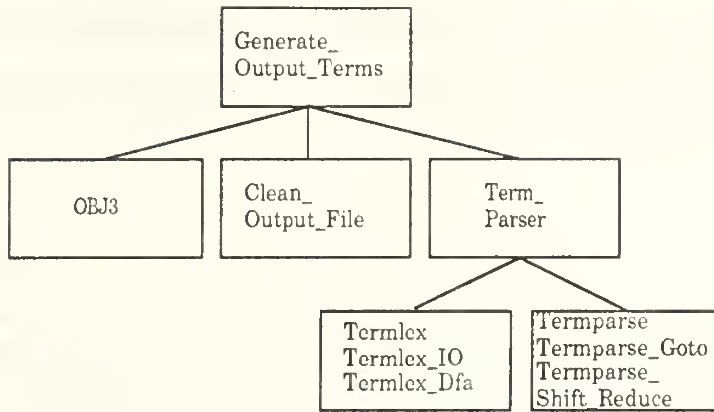


Figure 5.5 - Structure of the Generate_Output_Terms Subprocess

5. Match

Figure 5.6 shows the high level structure of the Match subprocess. Given an I/O list for the query and an operator sequence definition for the query, the Match subprocess must determine if the query will map to a given candidate component. The Extract_Prolog procedure copies the Prolog stored in the normalized query and candidate files and creates two new files containing the Prolog code. The Match subprocess then calls Find_Maps to find all of the mappings and Test_Maps to determine the best mapping. These are described in more detail below.

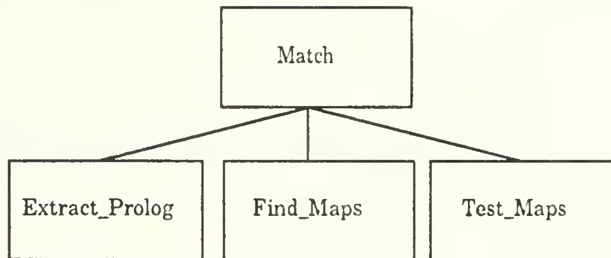


Figure 5.6 - Structure of the Match Subprocess

a. Find Maps

Figure 5.7 shows the structure of the Find_Maps subprocess. Find_Maps first calls the Prolog system using the Prolog extracted from the normalized specification files. Additional Prolog code used to drive the mapping process is shown in the Appendix. The Prolog process creates an output file that is examined by a lexical analyzer to read the

mapping information. If the candidate component is a generic object, then the maps are checked for consistent bindings to the generic parameters. If no maps are found, the candidate component receives a score of zero.

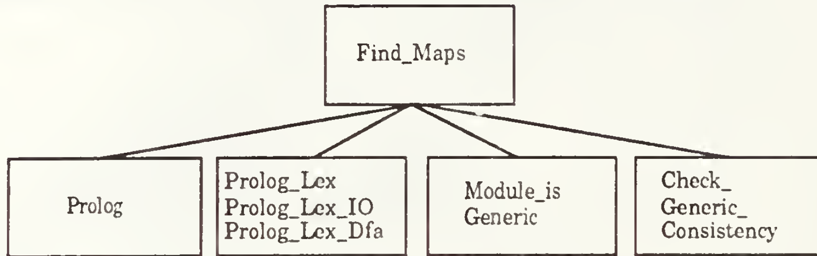


Figure 5.7 - Structure of the Find_Maps Subprocess

b. Test Maps

Figure 5.8 shows the structure of the Test_Maps subprocess. Given that there is a mapping between the query and candidate specifications, Test_Maps determines the correlation between the sorts in the two specifications and then calls Perform_Test. The Perform_Test procedure calls Transform_Term to transform the input and output terms in the I/O list from the query domain to the candidate component domain. It then creates a file to submit to OBJ3 to reduce the transformed input term and perform inductionless induction on the input/output pair.

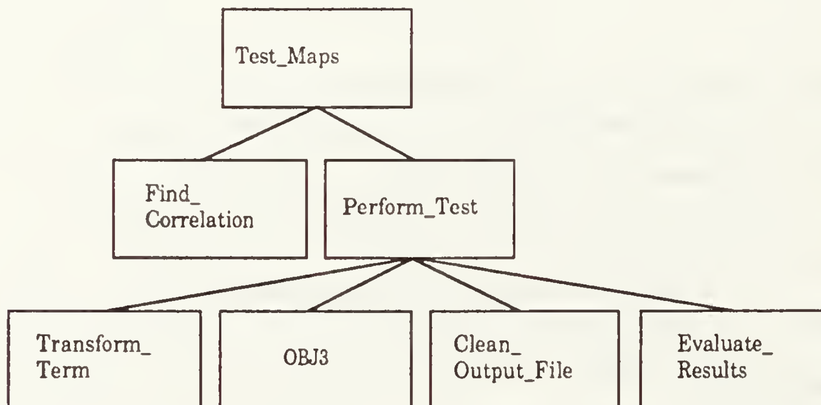


Figure 5.8 - Structure of the Test_Maps Subprocess

After the OBJ3 process completes, the Clean_Output_File procedure removes extraneous OBJ3 output from the file and the Evaluate_Results procedure calculates a score for the map. Test_Maps repeats this process for each map. The highest score obtained for any map is the overall score given to the component.

E. ABSTRACT DATA TYPES AND DATA STRUCTURES

1. Abstract Data Types

Several *reusable* abstract data types (ADT) are used extensively in the definition of the predominant data structures described in the previous sections. Their structures are shown here. An ADT called A_String, from the Verdix library [Verd91] bundled with the compiler, implements variable length strings and has the following form:

```
package A_Strings is
    type string_rec(len: natural) is
        record
            s   : string(1..len);
        end record;
    type A_String is access string_rec;
    ...
end A_Strings;
```

A_String provides the standard operations one would expect from a string package. A second ADT used frequently is Set. The Set ADT was provided by Berzins [Berz91] and has the following form:

```
generic
    type t is private;
    block_size: in natural:=128;
    with function eq(x,y: t) return boolean is "=";
package set_pkg is
    ...
private
    type link is access set;
    type elements_type is array(1..block_size) of  t;
    type set is
        record
            size        : natural:=0;       --The size of the set
            elements    : elements_type;    --The actual elements of the set
            next        : link:=null;       --The next node in the list
        end record;
end set_pkg;
```

The Set package is a generic package that provides the standard set operations plus additional operations for I/O. Another package provided by Berzins [Berz91] is the generic Sequence package, which implements a sequence ADT. The Sequence package provides standard sequence operations plus additional operations for I/O. It has the following form:

```
generic
    type t is private;
    block_size: in natural := 128;
package sequence_pkg is
    type sequence is private;
    . . .
private
    type link is access sequence;
    type elements_type is array(1 .. block_size) of t;
    type sequence is
        record
            length      : natural := 0;      -- The length of the sequence.
            elements    : elements_type;     -- A prefix of the sequence.
            next        : link := null;      -- The next node in the list.
        end record;
        -- Elements(1 .. min(length, block_size)) contains data.
end sequence_pkg;
```

2. Data Structures

The principle data structures used in the implementation are structures for terms, operator definitions, a test set, an I/O list, and maps. A term is an inherently recursive object so the data structure used to model it uses access types, as follows:

```
type Term;
type Term_Access is access Term;
Max_Arguments : constant natural := 10;
type Access_Array is array(1..Max_Arguments) of Term_Access;

type Term is
    record
        Op_Name      : A_Strings.A_String;
        Range_Sort   : A_Strings.A_String;
        Num_Args     : natural := 0;
        Signature    : natural := 0;
        Arguments    : Access_Array := (1..Max_Arguments => null);
    end record;
```

From the definition one can see that a term consists of an operator name, range sort, a certain number of arguments, and an array of arguments that are also terms. The

signature field in the record is used to point to the operator definition in a sequence of operator definitions that defines the structure of the term. For simplicity, the current implementation uses a constant array size (10) for term arguments rather than a discriminated record to implement variable length arrays.

A data structure is required to model the signature of a specification. The basis for this is the definition for an operator, as follows:

```
type Sort_Position_Pair is
    record
        Sort_Name       : A_Strings.A_String;
        Position        : natural;
    end record;

package Pair_Sequence_Pkg is new Sequence_Pkg(t => Sort_Position_Pair);

type Op_Defn_Type is
    record
        Op_Name                 : A_Strings.A_String;
        Num_Parameters          : natural;
        Range_Sort              : A_Strings.A_String;
        Domain_Sorts            : Pair_Sequence_Pkg.Sequence;
    end record;

package Op_Defn_Seq_Pkg is new Sequence_Pkg(t => Op_Defn_Type);
```

An operator definition consists of an operator name, a certain number of domain parameters, a range sort, and a sequence of domain sorts that each have a sort name and a position. A signature for a specification is a sequence of operator definitions. Note that the reusable sequence package was used twice here, once for the sequence of domain sorts and once for the sequence of operators.

A test set is implemented as a sequence of terms as follows[2]:

```
package Const_Seq_Pkg is new Sequence_Pkg(t => A_Strings.A_String);

type Sort_Index_Info is
    record
        Sort_Name       : A_Strings.A_String := A_Strings.to_a("!");
        Start           : Natural := 0;
        Stop            : Natural := 0;
```

---

[2]In the formal definition of query by consistency, the test set is treated as a set of operators since that is the logical interpretation of a test set. For implementation efficiency, the test set is treated as a list of terms , rather than translate an operator definition to a term every time one is needed.

```
        Constants       : Const_Seq_Pkg.Sequence := Const_Seq_Pkg.Empty;
    end record;

type Sort_Index_Array is array(Positive range <>) of Sort_Index_Info;

package Term_Sequence_Pkg is new Sequence_Pkg(t => Term_Access);

type Test_Set_Rec(Size : Natural := 10) is
    record
        Sort_Index      : Sort_Index_Array(1..Size);
        Term_List       : Term_Sequence_Pkg.Sequence :=
                              Term_Sequence_Pkg.Empty;
    end record;

type Test_Set_Def is access Test_Set_Rec;
```

The test set uses the sequence package in its definition. It also uses a variable length array (Sort_Index_Array) as an index into the term list to indicate where the terms associated with a particular sort begin and end. Sequences of constant identifiers are also maintained in the Sort_Index_Array. During term expansion it is sometimes necessary to add a constant to avoid a circularity. Constants must be declared before term rewriting begins, so the Sort_Index keeps track of all constants used in test set term definitions. Finally, the test set is implemented as an access type to avoid passing a large data structure around as a parameter.

The next principal data structure is the I/O list, implemented as follows:

```
type IO_Pair_Rec;

type IO_List_Def is access IO_Pair_Rec;

type IO_Pair_Rec is
    record
        Input       : Term_Access;
        Output      : Term_Access;
        Result      : A_Strings.A_String;
        Next        : IO_List_Def;
    end record;
```

The I/O list is a linked list of I/O pairs. An I/O pair is an input term, its corresponding output, the sort of the result, and a pointer to the next I/O pair. Since the I/O list is implemented as a linked list, it is only necessary to pass a pointer to the head of the list when passing the I/O list as a parameter.

The last of the principal data structures is the map structure, used to map one signature to another. It is implemented as follows:

```
type Generic_Binding is
    record
        Generic_Name     : A_Strings.A_String;
        Bound_To         : A_Strings.A_String;
    end record;

type Array_Type is array(Positive range <>) of Generic_Binding;

subtype Size_Range is integer range 0..100;

type Gen_Consis_Rec(Size : Size_Range := 10) is
    record
        Bindings   : Array_Type(1..Size);
        Length     : Size_Range := 0;
    end record;

type Correlation_Array is array(Positive range <>) of A_Strings.A_String;

type Correlation_Rec(Size : Size_Range := 10) is
    record
        Sort_Correlation   : Correlation_Array(1..Size);
    end record;

type Correlation_Access is access Correlation_Rec;

type Maps;

type Map_Access is access Maps;

type Maps is
    record
        Map                  : Op_Defn_Seq_Pkg.Sequence;
        Generic_Bindings     : Gen_Consis_Rec;
        Sort_Correlation     : Correlation_Access;
        Next                 : Map_Access := null;
    end record;
```

The list of maps from a query specification to a candidate component specification is implemented with a linked list. Each map in the linked list is implemented as a record containing a sequence of operator definitions, a generic consistency record, an array of sorts (sort correlation) corresponding to the query's sorts, and a pointer to the next map.

F. MATCHING EXAMPLES

This section provides three examples of the query by consistency method. The reason for including the examples is to demonstrate that the system works and to reinforce the concepts described earlier. Each of the examples presents a query specification, a candidate component specification, the test set generated from the query, the I/O list, the transformed terms submitted to OBJ for inductionless induction, and the results of the process. The first two examples match against a single candidate component, whereas the last example matches against a list of candidates.

1. List Matching Example

This first example matches a query for a *list* abstract data type (ADT) against a candidate that also models a list ADT. To illustrate a base case, the two components are identical up to renaming of the operators and sorts. There is only one possible mapping between them. The query for the list is as follows:

```
***(operations nil cons car cdr)
obj LIST-OF-NAT is sort List .
    protecting NAT .
    subsort Nat < List .
    op nil : -> List .
    op cons : Nat List -> List .
    op car  : List -> Nat .
    op cdr  : List -> List .
    var I, J : Nat .
    var L : List .
    eq car(cons(I,L)) = I .
    eq cdr(nil) = nil .
    eq cdr(cons(I,L)) = L .
endo
```

The specification for the stored component to which the query will be compared is as follows:

```
***(operations empty insert head tail)
obj ALIST-OF-NAT is sort Alist .
    protecting NAT .
    subsort Nat < Alist .
    op empty : -> Alist .
    op insert : Nat Alist -> Alist .
    op head  : Alist -> Nat .
    op tail  : Alist -> Alist .
    var I, J : Nat .
    var L : Alist .
```

```
    eq head(insert(I,L)) = I .
    eq tail(empty) = empty .
    eq tail(insert(I,L)) = L .
endo
```

The test set terms generated from the normalized query are shown below, ordered by sort:

```
Zero:     0
Nat:      0
Nat:      succ(natconst1)
NzNat:    1
NzNat:    succ(nznatconst1)
Bool:     true
Bool:     false
List:     cdr(listconst1)
List:     cons(!!!, listconst1)
List:     nil
```

The I/O list generated from the test set and the export signature contains 16 terms. Table 5.1 shows the the input terms and their corresponding outputs after reduction.

TABLE 5.1 - I/O LIST FOR LIST-OF-NAT

#	Input	Output
1	nil	nil
2	cdr(cdr(listconst1))	cdr(cdr(listconst1))
3	cdr(nil)	nil
4	car(cdr(listconst1))	car(cdr(listconst1))
5	car(cons(natconst1, listconst1))	natconst1
6	car(nil)	car(nil)
7	cdr(cons(0, listconst1))	listconst1
8	cdr(cons(succ(natconst1), listconst1))	listconst1
9	cons(0, cdr(listconst1))	cons(0, cdr(listconst1))
10	cons(0, nil)	cons(0, nil)
11	cons(succ(natconst1), cdr(listconst1))	cons(succ(natconst1), cdr(listconst1))
12	cons(succ(natconst1), nil)	cons(succ(natconst1), nil)

13	cons(0, cons(0, listconst1))	cons(0, cons(0, listconst1))
14	cons(0, cons(succ(natconst1), listconst1))	cons(0, cons(succ(natconst1), listconst1))
15	cons(succ(natconst1), cons(0, listconst1))	cons(succ(natconst1), cons(0, listconst1))
16	cons(succ(natconst1), cons(succ(natconst1), listconst1))	cons(succ(natconst1), cons(succ(natconst1), listconst1))

Given the I/O list, the next step is to map the query to the component. The only mapping is:

```
nil    -> empty
cons   -> insert
car    -> head
cdr    -> tail
```

Table 5.2 shows the check for term equivalence after transformation of the I/O list to the component domain. Each term in the *proof* column has the structure prove(term1, term2), where term1 is the transformed input and term2 is the transformed output. The *prove* function reduces term1 and then compares term1 and term2 using inductionless induction (==). The *result* column shows the result of the check for equivalence. The score for a sequence of checks is the ratio of the number of true results to the number tried, multiplied by 100 and truncated.

TABLE 5.2 - EQUIVALENCE CHECKS (LIST-OF-NAT TO ALIST-OF-NAT)

#	Proof LIST-OF-NAT to ALIST-OF-NAT Score: 100	Result
1	prove(empty, empty) .	true
2	prove(tail(tail(listconst1)), tail(tail(listconst1))) .	true
3	prove(tail(empty), empty) .	true
4	prove(head(tail(listconst1)), head(tail(listconst1))) .	true
5	prove(head(insert(natconst1, listconst1)), natconst1) .	true
6	prove(head(empty), head(empty)) .	true
7	prove(tail(insert(0, listconst1)), listconst1) .	true
8	prove(tail(insert(succ(natconst1), listconst1)), listconst1) .	true
9	prove(insert(0, tail(listconst1)), insert(0, tail(listconst1))) .	true

10	prove(insert(0, empty), insert(0, empty)) .	true
11	prove(insert(succ(natconst1), tail(listconst1)), insert(succ(natconst1),tail(listconst1))) .	true
12	prove(insert(succ(natconst1), empty), insert(succ(natconst1), empty)) .	true
13	prove(insert(0, insert(0, listconst1)), insert(0, insert(0, listconst1))) .	true
14	prove(insert(0, insert(succ(natconst1), listconst1)), insert(0, insert(succ(natconst1), listconst1))) .	true
15	prove(insert(succ(natconst1), insert(0, listconst1)), insert(succ(natconst1), insert(0, listconst1))) .	true
16	prove(insert(succ(natconst1), insert(succ(natconst1), listconst1)), insert(succ(natconst1), insert(succ(natconst1), listconst1))) .	true

It is not surprising that the each equivalence test was true and that the score is 100. The semantics (the axioms) of the two components are identical.

2. Set Matching Example

In this example the query is a specification for a *set* ADT and the component models a *set* ADT. The query is a requirement for a set of natural numbers. The query specification is as follows:

```
***(operations empty insert member subset equal)
obj SET-OF-NAT is
    sort Set .
    protecting NAT .
    op empty : -> Set .
    op insert : Nat Set -> Set .
    op member : Nat Set -> Bool .
    op subset : Set Set -> Bool .
    op equal  : Set Set -> Bool .
    vars S1 S2 : Set .
    vars E1 E2 : Nat .
    cq insert(E1, S1) = S1 if member(E1, S1) .
    eq member(E1, empty) = false .
    eq member(E1, insert(E2, S1)) = or(==(E1, E2), member(E1, S1)) .
    eq subset(empty, S1) = true .
    eq subset(S1, S1) = true .
    eq subset(insert(E1,S1), S2) = and(member(E1,S2), subset(S1, S2)) .
    eq equal(S1, S2) = and(subset(S1, S2), subset(S2, S1)) .
endo
```

The candidate component specification is shown below. The specification is generic. In order to perform the matching, the component specification is instantiated with NAT (a predefined object for natural numbers). Note that arguments for the add operator

are reversed. Also note that the definition of the equal operator is different from that in the query. A hidden remove operation is used (it is not exported). This will affect the scoring.

```
***(operations empty add member subset equal union)
obj GENERIC-SET[X :: TRIV] is
     sort Set .
     op empty : -> Set .
     op add : Set Elt.X -> Set .
     op member : Elt.X Set -> Bool .
     op subset : Set Set -> Bool .
     op equal : Set Set -> Bool .
     op union : Set Set -> Set .
     op remove : Elt.X Set -> Set .
     vars S1 S2 : Set .
     vars E1 E2 : Elt.X .
     cq add(S1, E1) = S1 if member(E1, S1) .
     eq member(E1, empty) = false .
     eq member(E1, add(S1, E2)) = or(==(E1, E2), member(E1, S1)) .
     eq subset(empty, S1) = true .
     eq subset(S1, S1) = true .
     eq subset(add(S1,E1), S2) = and(member(E1,S2), subset(S1, S2)) .
     eq equal(empty, empty) = true .
     eq equal(S1, S1) = true .
     eq equal(empty, add(S1, E1)) = false .
     eq equal(add(S1, E1), empty) = false .
     eq equal(add(S1,E1),add(S2,E2)) = and(member(E1,add(S2,E2)),
                    equal(S1,remove(E1,add(S2,E2)))) .
     eq union(S1, empty) = S1 .
     eq union(empty, S1) = S1 .
     eq union(add(S1, E1), S2) = if-then-else(member(E1, S2),
               union(S1, S2), union(S1, add(S2, E1))) .
     eq remove(E1, empty) = empty .
     eq remove(E1, add(S1, E1)) = S1 .
     cq remove(E1, add(S1, E2)) = add(remove(E1,S1), E2) if =/=(E1, E2) .
endo
```

The following test set was generated from the normalized query specification:

```
Zero:     0
Nat:      0
Nat:      succ(natconst1)
NzNat:    1
NzNat:    succ(nznatconst1)
Bool:     true
Bool:     false
Set:      empty
```

Set: insert(!!!, setconst1)

The I/O list generated from the test set and the export signature contains 31 terms. Table 5.3 shows the the input terms and their corresponding outputs after reduction.

TABLE 5.3 - I/O LIST FOR SET-OF-NAT

#	Input	Output
1	empty	empty
2	insert(0, empty)	insert(0, empty)
3	insert(succ(natconst1), empty)	insert(succ(natconst1), empty)
4	member(0, empty)	false
5	member(succ(natconst1), empty)	false
6	subset(empty, empty)	true
7	equal(empty, empty)	true
8	insert(0, insert(0, setconst1))	insert(0, setconst1)
9	insert(0, insert(succ(natconst1), setconst1))	insert(0, insert(succ(natconst1), setconst1))
10	insert(succ(natconst1), insert(0, setconst1))	insert(succ(natconst1), insert(0, setconst1))
11	insert(succ(natconst1), insert(succ(natconst1), setconst1))	insert(succ(natconst1), setconst1)
12	member(0, insert(0, setconst1))	true
13	member(0, insert(succ(natconst1), setconst1))	member(0, setconst1)
14	member(succ(natconst1), insert(0, setconst1))	member(succ(natconst1), setconst1)
15	member(succ(natconst1), insert(succ(natconst1), setconst1))	true
16	subset(empty, insert(0, setconst1))	true
17	subset(empty, insert(succ(natconst1), setconst1))	true
18	subset(insert(0, setconst1), empty)	false
19	subset(insert(succ(natconst1), setconst1), empty)	false
20	equal(empty, insert(0, setconst1))	false
21	equal(empty, insert(succ(natconst1), setconst1))	false

22	equal(insert(0, setconst1), empty)	false
23	equal(insert(succ(natconst1), setconst1), empty)	false
24	subset(insert(0, setconst1), insert(0, setconst1))	subset(setconst1, insert(0, setconst1))
25	subset(insert(0, setconst1), insert(succ(natconst1), setconst1))	and(member(0, setconst1), subset(setconst1, insert(succ(natconst1), setconst1)))
26	subset(insert(succ(natconst1), setconst1), insert(0, setconst1))	and(member(succ(natconst1), setconst1), subset(setconst1, insert(0, setconst1)))
27	subset(insert(succ(natconst1), setconst1), insert(succ(natconst1), setconst1))	subset(setconst1, insert(succ(natconst1), setconst1))
28	equal(insert(0, setconst1), insert(0, setconst1))	and(subset(setconst1, insert(0, setconst1)), subset(setconst1, insert(0, setconst1)))
29	equal(insert(0, setconst1), insert(succ(natconst1), setconst1))	and(member(0, setconst1), and(subset(setconst1, insert(succ(natconst1), setconst1)), and(member(succ(natconst1), setconst1), subset(setconst1, insert(0, setconst1)))))
30	equal(insert(succ(natconst1), setconst1), insert(0, setconst1))	and(member(succ(natconst1), setconst1), and(subset(setconst1, insert(0, setconst1)), and(member(0, setconst1), subset(setconst1, insert(succ(natconst1), setconst1)))))
31	equal(insert(succ(natconst1), setconst1), insert(succ(natconst1), setconst1))	and(subset(setconst1, insert(succ(natconst1), setconst1)), subset(setconst1, insert(succ(natconst1), setconst1)))

Given the I/O list, the next step is to map the query to the candidate component. There are eight possible ways to map the query to the candidate. The most obvious is:

```
empty     ->  empty
insert    ->  add
member    ->  member
subset    ->  subset
equal     ->  equal
```

The reason there are eight mappings is due to the identical domain and range sorts in the operators subset and equal. There are two possible mappings from $subset_q$ to $subset_c$. For

each of those there are two mappings from $equal_q$ to $equal_c$, which results in four mappings. Likewise $subset_q$ may map to $equal_c$ and $equal_q$ to $subset_c$, which produces another four.

When these maps are checked and scored, two receive a score of 87 and six receive a score of 61. The two maps with score 87 have the subset operators mapped correctly and the equal operators varying. The other six maps represent the other combinations, whose positive results come primarily from the empty, add (insert), and member operators.

Table 5.4 shows the check for term equivalence for one of the maps given a score of 87. The checks yield positive results for the first 27 pairs and negative results for the last 4. Note that even though the axioms for the *equal* operator are different in both specifications, many of the checks using *equal* yield positive results.

TABLE 5.4 - EQUIVALENCE CHECKS (SET-OF-NAT TO GENERIC-SET)

#	Proof SET-OF-NAT to GENERIC-SET Score: 87	Result
1	prove(empty, empty) .	true
2	prove(add(empty, 0), add(empty, 0)) .	true
3	prove(add(empty, succ(natconst1)), add(empty, succ(natconst1))) .	true
4	prove(member(0, empty), false) .	true
5	prove(member(succ(natconst1), empty), false) .	true
6	prove(subset(empty, empty), true) .	true
7	prove(equal(empty, empty), true) .	true
8	prove(add(add(setconst1, 0), 0), add(setconst1, 0)) .	true
9	prove(add(add(setconst1, succ(natconst1)), 0), add(add(setconst1, succ(natconst1)), 0)) .	true
10	prove(add(add(setconst1, 0), succ(natconst1)), add(add(setconst1, 0), succ(natconst1))) .	true
11	prove(add(add(setconst1, succ(natconst1)), succ(natconst1)), add(setconst1, succ(natconst1))) .	true
12	prove(member(0, add(setconst1, 0)), true) .	true
13	prove(member(0, add(setconst1,succ(natconst1))), member(0, setconst1)) .	true
14	prove(member(succ(natconst1), add(setconst1, 0)), member(succ(natconst1), setconst1)) .	true
15	prove(member(succ(natconst1), add(setconst1, succ(natconst1))), true) .	true
16	prove(subset(empty, add(setconst1, 0)), true) .	true

17	prove(subset(empty, add(setconst1, succ(natconst1))), true) .	true
18	prove(subset(add(setconst1, 0), empty), false) .	true
19	prove(subset(add(setconst1, succ(natconst1)), empty), false) .	true
20	prove(equal(add(setconst1, 0), empty), false) .	true
21	prove(equal(add(setconst1, succ(natconst1)), empty), false) .	true
22	prove(equal(empty, add(setconst1, 0)), false) .	true
23	prove(equal(empty, add(setconst1, succ(natconst1))), false) .	true
24	prove(subset(add(setconst1, 0), add(setconst1, 0)), subset(setconst1, add(setconst1, 0))) .	true
25	prove(subset(add(setconst1, 0), add(setconst1, succ(natconst1))), and(member(0, setconst1), subset(setconst1, add(setconst1, succ(natconst1))))) .	true
26	prove(subset(add(setconst1, succ(natconst1)), add(setconst1, 0)), and(member(succ(natconst1), setconst1), subset(setconst1, add(setconst1, 0)))) .	true
27	prove(subset(add(setconst1, succ(natconst1)), add(setconst1, succ(natconst1))),subset(setconst1, add(setconst1, succ(natconst1)))) .	true
28	prove(equal(add(setconst1, 0), add(setconst1, 0)), and(subset(setconst1, add(setconst1, 0)), subset(setconst1, add(setconst1, 0)))) .	false
29	prove(equal(add(setconst1, succ(natconst1)), add(setconst1, 0)), and(member(0, setconst1), and(subset(setconst1, add(setconst1, succ(natconst1))), and(member(succ(natconst1), setconst1), subset(setconst1, add(setconst1, 0)))))) .	false
30	prove(equal(add(setconst1, 0), add(setconst1, succ(natconst1))), and(member(succ(natconst1), setconst1), and(subset(setconst1, add(setconst1, 0)), and(member(0, setconst1), subset(setconst1, add(setconst1, succ(natconst1))))))) .	false
31	prove(equal(add(setconst1, succ(natconst1)), add(setconst1, succ(natconst1))), and(subset(setconst1, add(setconst1, succ(natconst1))),subset(setconst1,add(setconst1,succ(natconst1))))) .	false

3. Stack Matching Example

The final example matches a query for a stack of integers to three generic object specifications: a generic stack, a generic list, and a generic first-in-first-out queue. These three have been chosen because the query will map to each of them, but their behaviors are different. The query specification, which is a simple request for four stack operators, is:

```
***(operations empty push pop top)
obj STACK-OF-INT is sort Stack .
    protecting INT .
    op empty : -> Stack .
```

```
    op push : Int Stack -> Stack .
    op top : Stack -> Int .
    op pop : Stack -> Stack .
    var S : Stack .
    var X : Int .
    eq top(push(X, S)) = X .
    eq pop(push(X, S)) = S .
endo
```

The specification for the generic stack (below) is similar to the query but provides more functionality.

```
***(operations create isempty push pop top size)
obj GENERIC-STACK[X :: TRIV] is sort Stack .
    protecting NAT .
    op create : -> Stack .
    op isempty : Stack -> Bool .
    op push : Elt.X Stack -> Stack .
    op top : Stack -> Elt.X .
    op pop : Stack -> Stack .
    op underflow : -> Stack .
    op size : Stack -> Nat .
    var S : Stack .
    var X : Elt.X .
    eq size(create) = 0 .
    eq size(push(X, S)) = sum(1, size(S)) .
    eq top(push(X, S)) = X .
    eq pop(push(X, S)) = S .
    eq pop(create) = underflow .
    eq isempty(S) = if-then-else(==(S, create), true, false) .
endo
```

The specification for the generic list is:

```
***(operations nil cons car cdr length contains)
obj GENERIC-LIST[X :: TRIV] is sort List .
    protecting NAT .
    subsort Elt < List .
    op nil : -> List .
    op cons : Elt List -> List .
    op car  : List -> Elt .
    op cdr  : List -> List .
    op length : List -> Nat .
    op contains : List Elt -> Bool .
    var I, J : Elt .
    var L : List .
    eq length(nil) = 0 .
```

```
    eq length(cons(I, L)) = sum(1, length(L)) .
    eq car(nil) = nil .
    eq car(cons(I,L)) = I .
    eq cdr(nil) = nil .
    eq cdr(cons(I,L)) = L .
    eq contains(nil, I) = false .
    eq contains(cons(J, L), I) = if-then-else(==(J, I), true, contains(L, I)) .
endo
```

Finally, the specification for the generic queue is:

```
***(operations empty isempty add pop front length)
obj GENERIC-FIFO-QUEUE[X :: TRIV] is sort Queue .
    protecting NAT .
    op empty : -> Queue .
    op isempty : Queue -> Bool .
    op add : Elt.X Queue -> Queue .
    op front : Queue -> Elt.X .
    op pop : Queue -> Queue .
    op length : Queue -> Nat .
    var S : Queue .
    var X : Elt.X .
    eq length(empty) = 0 .
    eq length(add(X, S)) = sum(length(S), 1) .
    eq front(add(X, S)) = if-then-else(==(S, empty), X, front(S)) .
    eq pop(add(X, S)) = if-then-else(==(S, empty), empty, add(X, pop(S))) .
    eq isempty(S) = if-then-else(==(S, empty), true, false) .
endo
```

The following test set was generated from the normalized query specification for the stack of integers:

```
Zero:     0
Nat:      0
Nat:      succ(natconst1)
NzNat:    1
NzNat:    succ(nznatconst1)
Bool:     true
Bool:     false
Int:      0
Int:      succ(intconst1)
Int:      pred(intconst1)
NzInt:    succ(nzintconst1)
NzInt:    pred(nzintconst1)
Stack:    pop(stackconst1)
Stack:    push(!!!, stackconst1)
Stack:    empty
```

The I/O list generated from the test set and the export signature contains 24 terms. Table 5.5 shows the the input terms and their corresponding outputs after reduction.

TABLE 5.5 - I/O LIST FOR STACK-OF-INT

#	Input	Output
1	empty	empty
2	pop(pop(stackconst1))	pop(pop(stackconst1))
3	pop(empty)	pop(empty)
4	top(pop(stackconst1))	top(pop(stackconst1))
5	top(push(intconst1, stackconst1))	intconst1
6	top(empty)	top(empty)
7	pop(push(0, stackconst1))	stackconst1
8	pop(push(succ(intconst1), stackconst1))	stackconst1
9	pop(push(pred(intconst1), stackconst1))	stackconst1
10	push(0, pop(stackconst1))	push(0, pop(stackconst1))
11	push(0, empty)	push(0, empty)
12	push(succ(intconst1), pop(stackconst1))	push(sum(1, intconst1), pop(stackconst1))
13	push(succ(intconst1), empty)	push(sum(1, intconst1), empty)
14	push(pred(intconst1), pop(stackconst1))	push(sum(intconst1, -1), pop(stackconst1))
15	push(pred(intconst1), empty)	push(sum(intconst1, -1), empty)
16	push(0, push(0, stackconst1))	push(0, push(0, stackconst1))
17	push(0, push(succ(intconst1), stackconst1))	push(0, push(sum(1, intconst1), stackconst1))
18	push(0, push(pred(intconst1), stackconst1))	push(0, push(sum(intconst1, -1), stackconst1))
19	push(succ(intconst1), push(0, stackconst1))	push(sum(1, intconst1), push(0, stackconst1))
20	push(succ(intconst1), push(succ(intconst1), stackconst1))	push(sum(1, intconst1), push(sum(1, intconst1), stackconst1))
21	push(succ(intconst1), push(pred(intconst1), stackconst1))	push(sum(1, intconst1), push(sum(intconst1, -1), stackconst1))

22	push(pred(intconst1), push(0, stackconst1))	push(sum(intconst1, -1), push(0, stackconst1))
23	push(pred(intconst1), push(succ(intconst1), stackconst1))	push(sum(intconst1, -1), push(sum(1, intconst1), stackconst1))
24	push(pred(intconst1), push(pred(intconst1), stackconst1))	push(sum(intconst1, -1), push(sum(intconst1, -1), stackconst1))

Given the I/O list, the next step is to consider the mappings and the checks for equivalence in each of the three candidate specifications. The query maps to the generic stack in one way:

```
pop    -> pop
top    -> top
push   -> push
empty  -> create
```

Table 5.6 shows the comparison of terms from the query and the generic stack. Check #3 had a false result because the candidate specification reduced pop(create) to *underflow*, whereas the query did not.

TABLE 5.6 - EQUIVALENCE CHECKS (STACK-OF-INT TO GENERIC-STACK)

#	Proof STACK-OF-INT to GENERIC-STACK Score: 95	Result
1	prove(create, create) .	true
2	prove(pop(pop(stackconst1)), pop(pop(stackconst1))) .	true
3	prove(pop(create), pop(create)) .	false
4	prove(top(pop(stackconst1)), top(pop(stackconst1))) .	true
5	prove(top(push(intconst1, stackconst1)), intconst1) .	true
6	prove(top(create), top(create)) .	true
7	prove(pop(push(0, stackconst1)), stackconst1) .	true
8	prove(pop(push(succ(intconst1), stackconst1)), stackconst1) .	true
9	prove(pop(push(pred(intconst1), stackconst1)), stackconst1) .	true
10	prove(push(0, pop(stackconst1)), push(0, pop(stackconst1))) .	true
11	prove(push(0, create), push(0, create)) .	true
12	prove(push(succ(intconst1), pop(stackconst1)), push(sum(1, intconst1), pop(stackconst1))) .	true

13	prove(push(succ(intconst1), create), push(sum(1, intconst1), create)) .	true
14	prove(push(pred(intconst1), pop(stackconst1)), push(sum(intconst1, -1), pop(stackconst1))) .	true
15	prove(push(pred(intconst1), create), push(sum(intconst1, -1), create)) .	true
16	prove(push(0, push(0, stackconst1)), push(0, push(0, stackconst1))) .	true
17	prove(push(0, push(succ(intconst1), stackconst1)), push(0, push(sum(1, intconst1), stackconst1))) .	true
18	prove(push(0, push(pred(intconst1), stackconst1)), push(0, push(sum(intconst1, -1), stackconst1))) .	true
19	prove(push(succ(intconst1), push(0, stackconst1)), push(sum(1, intconst1), push(0, stackconst1))) .	true
20	prove(push(succ(intconst1), push(succ(intconst1), stackconst1)), push(sum(1, intconst1), push(sum(1, intconst1), stackconst1))) .	true
21	prove(push(succ(intconst1), push(pred(intconst1), stackconst1)), push(sum(1, intconst1), push(sum(intconst1, -1), stackconst1))) .	true
22	prove(push(pred(intconst1), push(0, stackconst1)), push(sum(intconst1, -1), push(0, stackconst1))) .	true
23	prove(push(pred(intconst1), push(succ(intconst1), stackconst1)), push(sum(intconst1, -1), push(sum(1, intconst1), stackconst1))) .	true
24	prove(push(pred(intconst1), push(pred(intconst1), stackconst1)), push(sum(intconst1, -1), push(sum(intconst1, -1), stackconst1))) .	true

The stack query maps to the generic list in only one way:

```
pop    -> cdr
top    -> car
push   -> cons
empty  -> nil
```

Table 5.7 shows the term equivalence checks for the query and the generic list. Checks #3 and 6 are false because the candidate reduces cdr(nil) and car(nil) to nil whereas the query does not.

TABLE 5.7 - EQUIVALENCE CHECKS (STACK-OF-INT TO GENERIC-LIST)

#	Proof STACK-OF-INT to GENERIC-LIST Score: 91	Result
1	prove(nil, nil) .	true
2	prove(cdr(cdr(stackconst1)), cdr(cdr(stackconst1))) .	true
3	prove(cdr(nil), cdr(nil)) .	false
4	prove(car(cdr(stackconst1)), car(cdr(stackconst1))) .	true

5	prove(car(cons(intconst1, stackconst1)), intconst1) .	true
6	prove(car(nil), car(nil)) .	false
7	prove(cdr(cons(0, stackconst1)), stackconst1) .	true
8	prove(cdr(cons(succ(intconst1), stackconst1)), stackconst1) .	true
9	prove(cdr(cons(pred(intconst1), stackconst1)), stackconst1) .	true
10	prove(cons(0, cdr(stackconst1)), cons(0, cdr(stackconst1))) .	true
11	prove(cons(0, nil), cons(0, nil)) .	true
12	prove(cons(succ(intconst1), cdr(stackconst1)), cons(sum(1, intconst1), cdr(stackconst1))) .	true
13	prove(cons(succ(intconst1), nil), cons(sum(1, intconst1), nil)) .	true
14	prove(cons(pred(intconst1), cdr(stackconst1)), cons(sum(intconst1, -1), cdr(stackconst1))) .	true
15	prove(cons(pred(intconst1), nil), cons(sum(intconst1, -1), nil)) .	true
16	prove(cons(0, cons(0, stackconst1)), cons(0, cons(0, stackconst1))) .	true
17	prove(cons(0, cons(succ(intconst1), stackconst1)), cons(0, cons(sum(1, intconst1), stackconst1))) .	true
18	prove(cons(0, cons(pred(intconst1), stackconst1)), cons(0, cons(sum(intconst1, -1), stackconst1))) .	true
19	prove(cons(succ(intconst1), cons(0, stackconst1)), cons(sum(1, intconst1), cons(0, stackconst1))) .	true
20	prove(cons(succ(intconst1), cons(succ(intconst1), stackconst1)), cons(sum(1, intconst1), cons(sum(1, intconst1), stackconst1))) .	true
21	prove(cons(succ(intconst1), cons(pred(intconst1), stackconst1)), cons(sum(1, intconst1), cons(sum(intconst1, -1), stackconst1))) .	true
22	prove(cons(pred(intconst1), cons(0, stackconst1)), cons(sum(intconst1, -1), cons(0, stackconst1))) .	true
23	prove(cons(pred(intconst1), cons(succ(intconst1), stackconst1)), cons(sum(intconst1, -1), cons(sum(1, intconst1), stackconst1))) .	true
24	prove(cons(pred(intconst1), cons(pred(intconst1), stackconst1)), cons(sum(intconst1, -1), cons(sum(intconst1, -1), stackconst1))) .	true

Finally, the query maps to the first-in-first-out-queue in only one way:

pop -> pop
top -> front
push -> add
empty -> empty

Table 5.8 shows the term equivalence checks for the query and the generic queue. Check #5 is false because the *front* operator in the candidate does not have the same

behavior as the *top* operator in the query. Checks #7, 8, and 9 are false because of the behavioral differences in the *pop* operators.

TABLE 5.8 - EQUIVALENCE CHECKS (STACK-OF-INT TO GENERIC-FIFO-QUEUE)

#	Proof STACK-OF-INT to GENERIC-FIFO-QUEUE Score: 83	Result
1	prove(empty, empty) .	true
2	prove(pop(pop(stackconst1)), pop(pop(stackconst1))) .	true
3	prove(pop(empty), pop(empty)) .	true
4	prove(front(pop(stackconst1)), front(pop(stackconst1))) .	true
5	prove(front(add(intconst1, stackconst1)), intconst1) .	false
6	prove(front(empty), front(empty)) .	true
7	prove(pop(add(0, stackconst1)), stackconst1) .	false
8	prove(pop(add(succ(intconst1), stackconst1)), stackconst1) .	false
9	prove(pop(add(pred(intconst1), stackconst1)), stackconst1) .	false
10	prove(add(0, pop(stackconst1)), add(0, pop(stackconst1))) .	true
11	prove(add(0, empty), add(0, empty)) .	true
12	prove(add(succ(intconst1), pop(stackconst1)), add(sum(1, intconst1), pop(stackconst1))) .	true
13	prove(add(succ(intconst1), empty), add(sum(1, intconst1), empty)) .	true
14	prove(add(pred(intconst1), pop(stackconst1)), add(sum(intconst1, -1), pop(stackconst1))) .	true
15	prove(add(pred(intconst1), empty), add(sum(intconst1, -1), empty)) .	true
16	prove(add(0, add(0, stackconst1)), add(0, add(0, stackconst1))) .	true
17	prove(add(0, add(succ(intconst1), stackconst1)), add(0, add(sum(1, intconst1), stackconst1))) .	true
18	prove(add(0, add(pred(intconst1), stackconst1)), add(0, add(sum(intconst1, -1), stackconst1))) .	true
19	prove(add(succ(intconst1), add(0, stackconst1)), add(sum(1, intconst1), add(0, stackconst1))) .	true
20	prove(add(succ(intconst1), add(succ(intconst1), stackconst1)), add(sum(1, intconst1), add(sum(1, intconst1), stackconst1))) .	true
21	prove(add(succ(intconst1), add(pred(intconst1), stackconst1)), add(sum(1, intconst1), add(sum(intconst1, -1), stackconst1))) .	true
22	prove(add(pred(intconst1), add(0, stackconst1)), add(sum(intconst1, -1), add(0,stackconst1))) .	true
23	prove(add(pred(intconst1), add(succ(intconst1), stackconst1)), add(sum(intconst1, -1), add(sum(1, intconst1), stackconst1))) .	true

24	prove(add(pred(intconst1), add(pred(intconst1), stackconst1)), add(sum(intconst1, -1), add(sum(intconst1, -1), stackconst1))) .	true

The scores obtained by the check for equivalence are 95 for the generic stack, 91 for the generic list, and 83 for the generic queue. These scores all appear high, as if any of the components would satisfy the requirement. It is important to remember, however, that the scoring is relative, not absolute. A high score does not necessarily mean a candidate is acceptable. The scores are all close, but in the final analysis, the rank order is as one would expect. The generic stack is the most appropriate candidate to meet the requirement expressed in the query. The generic list could be used to simulate a stack, but is not as desirable. Finally, the queue is probably not acceptable as a substitute for a stack.

G. SUMMARY

This chapter describes the implementation details of the normalization and matching subsystems that make up the query by consistency method and uses examples to reinforce the concepts described in Chapters III and IV. As mentioned in the introduction, the implementation is meant to be a proof of concept. The query by consistency method has limitations, which are described in Section VI.G. There are also inefficiencies in the data structures and algorithms, which could be improved to enhance system performance. Section VII.C describes suggested modifications to enhance performance and Section VII.D examines suggested extensions to this research.

VI. EVALUATION OF THE SOFTWARE RETRIEVAL MODEL

A. INTRODUCTION

This chapter evaluates the software retrieval model, first from a broad perspective and then more specifically using Salton and McGill's [SM83] six critical evaluation criteria for examining information retrieval systems: recall, precision, effort, time, presentation, and coverage. This chapter also addresses the limitations of the query by consistency method.

B. A FRAMEWORK FOR SOFTWARE RETRIEVAL SYSTEM EVALUATION

A general framework for the evaluation of software retrieval systems is composed of three components. The first is a set of all possible candidate software retrieval systems. The second is a cost-performance valuation function and the third is the integration of the first two components into a choice of the *optimal* cost-performance software retrieval system. [Jone91]

The first component, the set of all possible candidate software retrieval systems, is focused on the physical and technological feasibility of software retrieval. This dissertation has concentrated on just this by introducing a retrieval system called query by consistency. The physical and technical feasibility has been shown. The remaining task is to choose measures of performance and cost for the system. Sections C through G of this chapter are devoted to this task. For the purpose of this description of a general framework, it is assumed that the set of all possible technologically feasible software retrieval systems form a convex cost-performance space. Figure 6.1 shows a continuous curve representing the boundary of the set of all possible technologically feasible software retrieval systems.

The second component, the cost-performance valuation function, is focused on the overall valuation of performance and cost of a software retrieval system. The valuation function represents a complete, transitive, non-satiated ordering of the space of all possible measures of performance and cost. The ordering is labelled "at least as cost performance as". Implicitly the valuation function contains the pairwise tradeoff of each measure of performance and cost. Thus, the cost-performance valuation function trades off performance with performance and each measure of performance with cost. The

candidate software component retrieval system that is ordered the "highest", will be the one with the largest valuation function value, and should be the most desirable system.

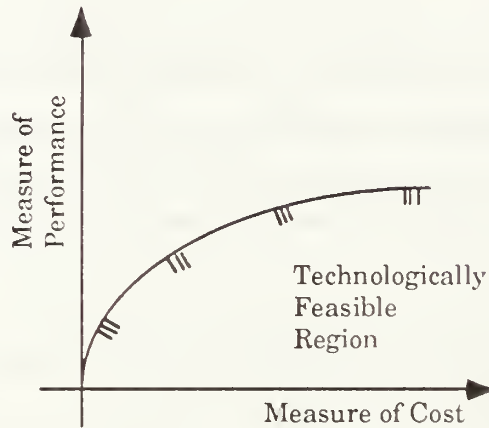


Figure 6.1 - Cost-Performance Curve

Assuming that increasing performance means increasing value, increasing cost means decreasing value, and assuming a convex space, a two dimensional picture of a valuation function (one measure of performance and one measure of cost) can be constructed. This is shown in Figure 6.2.

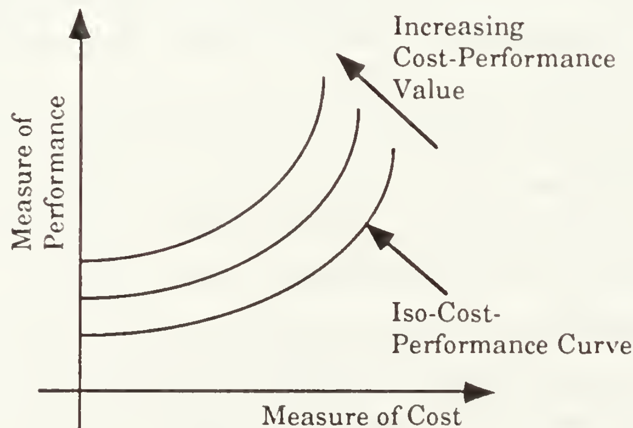


Figure 6.2 - Iso-Cost-Performance Curves

Each member of the family of curves shown is called an iso-cost-performance curve. The slope of the iso-cost-performance curve measures the tradeoff of performance and cost

at that point. The convexity assumption is interpreted as the willingness at low cost levels to trade off a relatively large increase in cost to gain a relatively small increase in performance. At high cost levels, the stated tradeoff is a small increase in cost to gain a relatively large increase in performance.

The third component in the general framework is the integration of the first two components into a choice of the optimal system. From the set of technologically feasible software component retrieval systems, the optimal system is the system that, by the valuation function, is the most valued. Figure 6.3 illustrates this by superimposing Figures 6.1 and 6.2.

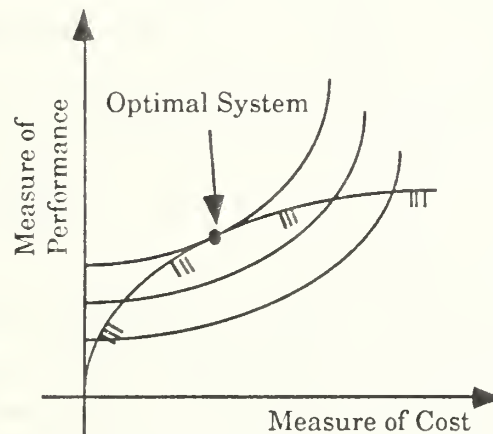


Figure 6.3 - Isolating the Optimal System

At this juncture, the field of reusable software component retrieval has not produced a large number of technologically feasible systems. When only discrete alternative systems are available, a complete valuation may not be needed. A simple application of vector dominance, appropriately adapted to the cost measure, may identify the optimal system. The following sections describe measures of performance that can be used for system evaluation and how query by consistency measures up to each.

C. RECALL AND PRECISION

This section examines CAPS' syntactic and semantic retrieval mechanisms with respect to recall and precision after presenting some background information.

1. Background

Recall and precision, which are used as measures of performance for information retrieval systems, were introduced in Chapter II. Recall is the ratio of the

number of relevant items retrieved to the total number of relevant items in the database. Precision is the ratio of the number of relevant items retrieved to the number of items retrieved. "Recall measures the ability of the system to retrieve useful documents, while precision conversely measures the ability to reject useless materials." [SM83 : p. 160] High recall and high precision are desirable. The primary factors affecting the recall and precision measures are indexing and relevance.

a. Indexing

Indexing refers to the representation of the object sought, such as a list of keywords or a formal specification. The research by Salton and McGill [SM83] focuses on the use of keywords. Query by consistency requires formal specifications. An indexor is a person who formulates the representation for the purpose of storing or retrieving an object. Depending on the indexing method chosen, users may have some control over the values obtained for recall and precision. By providing a broad, general query, users can expect high recall and relatively low precision. Conversely, a detailed, specific query leads to lower recall and increased precision.

For example, in a keyword system, a query with just one or two keywords will usually provide high recall and low precision, whereas an increase in the number of keywords lowers recall but improves precision. The same effect can be achieved with formal specifications. A specification that defines only a few simple operations will map syntactically to many more candidate specifications than would a specification with many operators.

In many instances, a trained and experienced indexor makes the difference between good values and poor values for the recall and precision metrics. Meaningful measurements rely on indexor consistency and experience. For the purpose of evaluating the software base retrieval mechanisms, indexor consistency and experience are assumed.

b. Relevance

Of the six criteria listed in Section A, recall and precision are the most difficult to assess because of the ambiguity of *relevance*. According to Salton and McGill, relevance may be either objective or subjective.

Objective relevance considers relevance as a logical property between a pair of items. In other words, "...relevance is the correspondence in context between an information requirement statement (a query) and an article (a document), that is, the extent to which the article covers the material that is appropriate to the requirement

statement." [SM83 : p. 163] It does not consider the state of knowledge of the user submitting the query. Subjective relevance considers not only the items being compared, but also the knowledge of the user submitting the query. For example, a user may already be aware of a document that was retrieved. From his perspective, that document is not relevant.

In evaluating the recall and precision characteristics of the query by consistency method, only *objective* relevance is considered, that is, the user's state of knowledge at the time of the query is not considered. Therefore, any component that meets the user's requirements is considered relevant.

Another factor with respect to relevance is the subjective nature of deciding when a particular item is relevant, that is, users will vary in their opinion about whether an item in the database is relevant to a query. Salton and McGill report that if objective system evaluation is the goal, then relevance assessments should be available from some external and impartial source.

2. Syntax and Semantics

Starting with the entire collection of components, syntactic search quickly identifies a set of components that have PSDL interfaces consistent with the query. Semantic search begins by trying to map the query's OBJ3 export signature to the export signature of each candidate in the set, and then uses the I/O list and axioms to perform reductions and compare normalized terms. If there is no morphism between the query and a candidate, the candidate receives a score of 0, otherwise the score is the ratio of the number of positive term equivalence checks over the total tried. Using these scores, the candidates are *rank ordered* based on their semantics. This step in the process is not consistent with typical retrieval systems. Low-scoring candidates are not discarded, but retained and placed at the bottom of the list. This complicates the use of recall and precision metrics to compare this system's performance against others. It is desirable to have the recall and precision measurements that are consistent with the recall and precision measurements of other systems. Therefore, it is best to remain as faithful as possible to the model provided by Salton and McGill. The next two sections describe the processes for determining recall and precision measurements in the software base reusable component retrieval system.

a. Recall is Linked to Syntactic Search

For recall, the method recommended by Salton and McGill is suitable. The value for recall may be computed solely on the basis of *syntactic* search since the *semantic* search mechanism does not delete components from the set. In other words, since the size

of the set does not change, the process that created the set is responsible for the metrics derived from it. High values for recall are expected from the syntactic search mechanism. Assuming some uniformity in the way the designers fashion components and query for them, comparing interfaces is a promising way to locate potential reuse candidates [RT89]. Recall is not perfect however. There are many ways to implement a problem and other components with slightly different interfaces may still be relevant. Experience with indexing (query formulation) is also a factor that will lead to improved recall.

b. *Precision Requires a New Method*

Since semantic search does not reduce the set of components, the measure of precision proposed by Salton and McGill penalizes this method of search. What is required is a metric that scores standard metric but also takes into account the *ranking* of the components. The standard measure for precision (P) is P= R/Q, where R is the number of relevant component retrieved and Q is the total number of components retrieved. Seen in a different way, every component in the set Q is given a score. A relevant component receives a score of 1 and a non-relevant component a score of 0. The scores are totalled to compute R, the number of relevant components in Q. In other words:

$$R = \sum_{i=1}^{n} q_i \quad \text{such that } q_i = 1 \text{ if relevant and } q_i = 0 \text{ if not relevant}$$

A method is proposed to compute a metric, called *ranking precision*, where each component in Q receives a score *between* 1 and 0 inclusive, based on its ranking. Given an ordered list of n components[1], with the highest ranked components coming first in the list, each component receives an *initial* score (q_i) of 1 if it is relevant and 0 if it is not relevant. Then, based on its ranking, the initial score for each component is altered as follows.

- If q_i is relevant and there are m non-relevant components ahead of it in the ranking, then $q_i = 1 - m/n$, that is, q_i is penalized for being ranked below non-relevant components.

[1]Components are ranked in descending order by score. Two or more components with the same score are given the same rank, so that they are neither rewarded nor penalized for their rank relative to one another.

- If q_i is not relevant and there are m relevant components ahead of it in the ranking, then $q_i = 1 + m/n$, that is, q_i is rewarded for being ranked below relevant components.

Consider some examples. Given a list of components that are all relevant, the standard precision is 1. Since there are no non-relevant components, the ranking precision is also 1. For a list containing all non-relevant components, the standard precision is 0. Since there are no relevant components, the ranking precision is also 0. These are the extreme cases, which show that the scores for ranking precision lie within the bounds of the scores for standard precision. Table 6.1 shows a third example, where a list of eight components in rank order are scored for ranking precision.

TABLE 6.1 - COMPUTING RANKING PRECISION

Rank	Relevance	Penalty / Reward	Score
1	1	0/8 = 0	1.0
2	1	0/8 = 0	1.0
3	0	2/8 = .25	.25
4	0	2/8 = .25	.25
5	1	-2/8 = -.25	.75
6	1	-2/8 = -.25	.75
7	0	4/8 = .5	.5
8	0	4/8 = .5	.5
		Total:	R = 5.0

Since four of the components are relevant and four are not, the standard precision is .50. Ranking precision is 5/8 or .625. In this case the ranking precision is higher than the standard precision. The best ranking precision score possible is .75, when the four relevant components are ranked first through fourth. The worst ranking precision is .25, which occurs when the four relevant components are ranked fifth through eighth. There are actually 70 possible rankings since the mathematical combination of 8 items taken 4 at a time is 70. The average of the ranking precision values for the 70 possible combinations is equal to the standard precision. Therefore, if the components were ranked randomly, then on average, they would have the same ranking precision as

standard precision, which is the desired effect. In practice, however, the ranking process *should* improve the precision value, so one would expect the system to have higher precision, which is the purpose of having semantic matching.

Some may argue that the ranking precision should never be lower than the standard precision. This is a more liberal view of scoring (calling the above technique *conservative*). To achieve *liberal ranking precision*, simply ignore the rule that penalizes poor ranking of relevant components. Since the scores for relevant components will always be 1, the value for R can never be less than it is using standard precision and hence, the value for liberal ranking precision will always be greater than or equal to standard precision. The argument in favor liberal ranking precision is a valid one. After all, if there were no semantic matching mechanism at all, the precision would be the *same* as standard precision.

The choice to use conservative or liberal ranking precision is left to those who will populate the software base and exercise the retrieval mechanisms. A fundamental limitation of both ranking precision techniques is that the precision can never be perfect (1) unless all components retrieved by the *syntactic* retrieval mechanism are relevant. It is my recommendation that additional heuristics be used during semantic matching to further reduce the set of candidates (some are suggested in Chapter VII). If this is accomplished, then the standard precision metric will be adequate.

D. EFFORT

Effort is the physical or intellectual labor required to formulate queries, conduct the search, and screen the output. Formal specifications are difficult for most people to write. Thus the amount of intellectual labor required to write specifications as queries could be excessive. In the context of prototyping in CAPS or in the development of safety critical systems, however, the specifications are needed for other reasons, so there is no additional effort associated with using specifications for retrieving reusable components. Also, automated tools such as syntax directed editors that help the designer formulate specifications, can alleviate much of the burden by performing formatting, structuring, and even type checking [AFM90]. This allows the designer to focus on the semantics of the specification, rather than the syntax. It also improves the designer's productivity.

Little effort is required to *display* the identified candidate components. The user interface designed by McDowell [McDo91] presents the user with a scrollable list of candidate component file names. The user merely selects a file name from the list to view

the corresponding specification or source code. Since the files in the list were referenced in the search phase, their addresses are already known, so there is minimal computation required to retrieve file data.

E. TIME

This metric measures system response time, that is, the time elapsed between the submission of the query to the system and the presentation of system responses. System response time is closely related to the discussion of *effort* in Section C. The time required to conduct the search can be broken down into two parts: syntactic retrieval and semantic retrieval. Syntactic retrieval is described in detail in the research by McDowell [McDo91]. McDowell designed the syntactic retrieval system to search efficiently by using a series of indexes or *dictionaries,* which the object-oriented database implements with B-trees [Onto91]. A B-tree is a data structure known to provide good search efficiency [AHU83]. The current "bottleneck" in search efficiency is not syntactic search, but the semantic search mechanism. Since the software base currently contains only a few components, no meaningful measurements can be obtained. Performing measurements on a well populated software base is an area of future research. Section VII.C.5 describes techniques that can improve the performance of the current implementation.

F. PRESENTATION

Presentation is the form of the output displayed to the user. The CAPS environment is an interactive, windowing environment with keyboard and mouse interfaces. The software base interface is consistent with the overall CAPS interface. A designer composes a specification in a text editor window and then saves the specification to a file. When the designer queries the system, the interface displays the query results as a scrollable list of file names. This list of file names is an ordered list of candidates that satisfy the query. The designer may then select one of the candidate file names with the mouse and the system will open a scrollable window to display the contents of the file. The designer may "cut and paste" any or all of the file into his own application. The ability to automatically integrate a retrieved component into a design is not yet available and is an area of future study.

G. COVERAGE

Coverage is the extent to which relevant items are included in the database. Since the software base currently contains only a few components, coverage is low. As the software base grows coverage will improve. In the future, when software base coverage is assessed, it will be meaningful to make the evaluation based on *domains* or particular application areas. The software base retrieval mechanisms are designed to search for any component, regardless of its domain, but as projects are designed and components are added to the software base, some application areas will have more coverage. The application domains that will most likely receive attention are fundamental data structures, mathematical functions, command and control software, and autonomous underwater vehicle control software.

H. QBC LIMITATIONS

There are limitations to the query by consistency method. One is the problem of mathematical precision. At the lowest level of rewriting in OBJ3, Lisp is used to compute the answers to mathematical functions. When the normal forms of terms are compared, the answers must be exact or the terms are not equivalent. Consider the case when a user defines the constant π as 3.141 and the stored component uses the system defined π, which has much greater precision. The answers for computations in each domain will be different. This problem could be alleviated by modifying the Lisp code in OBJ3 which checks term equivalence, relaxing the constraints on numeric precision.

Another limitation with QBC is in the area of subtype mapping. For example, if a designer queries for a stack of natural numbers, the query would map to a generic stack, but would not map to a stack of integers. Since natural is a subtype of integer, one might expect a mapping. The limitation exists because the mapping subsystem treats predefined sorts as constants. A possible solution is to treat them as variables and then perform a check (similar to the check for generic consistency) after the mappings are determined. This check would ensure that the mapping is consistent and that the query sort is the same as or a subtype of the candidate sort.

A third limitation is matching what I call *deep semantics*. Deep semantics are attributed to functions whose behavior becomes apparent only after a significant amount of processing has taken place. Sorting a list is an example of deep semantics. In query by consistency, the term submitted for sorting would consist mainly of symbolic constants which cannot be meaningfully compared. Consequently, the rewriting cannot go very far,

possibly only to the point of comparing the first two elements of the list. In many cases this will be adequate to compare semantics, but not the true semantics of this type of function. One approach to alleviating this problem is to use a longer I/O list, that is, expand terms to a deeper level before the expansion is cut off by adding symbolic constants. Another solution to this problem is to query using examples, wherein the designer provides axioms which are concrete examples of the processing behavior required. Section VII.D.7 describes this technique.

I. SUMMARY

This chapter takes a broad look at evaluating component retrieval systems by showing how measures of performance and cost can be combined to select an optimal retrieval system. The objective of the research presented in this dissertation is to expand the technologically feasible region of component retrieval, thus making improvements in specific measures of performance, especially precision and recall.

This chapter examines the software base reusable component retrieval mechanism with respect to six evaluation criteria suggested by Salton and McGill [SM83]. Measurements of precision and recall are the standard for comparing information retrieval systems. Recall performance is tied to syntactic search and precision to semantic search. Ranking precision is introduced as an alternative to standard precision. Effort required to use the system is mostly for constructing formal specifications but automated tools can alleviate much of the burden. Time and coverage are difficult to assess without a sizable software base. Presentation, the form of output, is closely linked to the standard CAPS windowing interface.

Query by consistency has some limitations which affect its performance which are related to mathematical precision, subtype mapping, and deep semantics. Suggested enhancements for overcoming these limitations are given.

VII. SUMMARY, CONCLUSIONS, AND SUGGESTIONS FOR FUTURE RESEARCH

A. INTRODUCTION

This chapter summarizes the contents of the dissertation, identifying those areas that are contributions to the state of the art, and then offers suggestions for future research. The suggestions for future research are divided into two areas. The first area describes changes that could be made to the current system to enhance its performance. The second area describes enhancements to the system that could be added to improve flexibility and power.

B. DISSERTATION SUMMARY

This dissertation has described in detail a technique for retrieving reusable software components from a software base using normalized algebraic specifications as the search key. The implemented reusable software component retrieval tool is part of a Computer Aided Prototyping System (CAPS). The goal of CAPS is to provide software designers an integrated environment aimed at rapidly prototyping hard real-time embedded systems [LK88, Luqi91]. Fundamental to this rapid prototyping paradigm is the use of a prototyping language (PSDL) and formal specification language (OBJ3) to define module interfaces and behavior. Also fundamental is the use of reusable software components to realize the design requirements.

The reusable software component retrieval tool uses both PSDL and OBJ3 to search the software base for components. Two search phases, syntactic and semantic, improve performance with respect to recall and precision. Given a query in the form of a specification, syntactic search uses the PSDL description of the query module's interface to locate candidate software components. Semantic search normalizes the query's algebraic axioms to compare the behavior of the query against behaviors of the candidate components. Semantic search is performed using a method called Query by Consistency (QBC).

This research makes contributions to the state of the art in reusable software component retrieval. These contributions are:

- A theory (Query by Consistency) and scoring heuristic for comparing specification semantics based on the existence of a homomorphism between sets of normalized terms in two algebras

- A method and corresponding implementation that determines a set of mappings between the export signatures of two algebraic specifications

- A method and corresponding implementation to develop a set of terms derived from a specification's test set and export signature

- Evidence that large scale reuse is feasible, avoiding the limitations of informal methods

- Provides a new method of retrieval which can serve as the basis for future automated semantic retrieval and component integration

The implementation of Query by Consistency demonstrates the ability of the method to rank order candidate specifications based on the behavior defined by their axioms. The author believes that refinements to the implementation can make it an efficient and effective tool for locating reusable software components in the CAPS domain. In addition, the concept can be extended to any application where algebraic specifications are used to specify object semantics and a rewrite system exists to exercise the semantics.

C. SYSTEM MODIFICATIONS TO ENHANCE PERFORMANCE

This section describes modifications that can be made to the existing system to improve its performance or extend its capabilities slightly. The modifications suggested in this section should not be difficult to implement.

1. Operator Overloading

Overloading, or polymorphism, is not supported in the current system, although both Ada and OBJ3 allow it. The limitation is in the Prolog matching software which requires the mapping from query to candidate to be injective and the bound operator names to be unique. In other words, each operator of the query must bind to a unique operator in the candidate, but the check for uniqueness is done using operator names. The solution to this problem is to avoid using the real operator names to perform the mapping and uses aliases instead.

For each candidate, the Prolog predicates would be generated using alternative names for all operators and an alias list would be maintained to allow the use of actual names when required, such as during term transformation. A similar alias list would be

maintained for a query. Using this technique, the Prolog code would no longer be a barrier to polymorphic mapping and the restriction of using unique operator names could be lifted.

2. Adding Predefined Objects

The current system contains predefined objects that can be used in the definition of new specifications. The predefined objects offered by OBJ3 include BOOL (Boolean), NAT (Natural), NZNAT (Positive), INT (Integer), FLOAT, RAT (Rational), QID, QIDL, and ID (Identifiers). To extend the descriptive power of the language and the matching power of the system, more predefined objects could be added, such as set, list, stack, queue, tree, sequence, etc. This would make it easier for engineers to pose some queries, such as the follow specification for a list of integers:

```
***(operations nil cons car cdr length sum)
obj MYOBJECT is
    protecting LIST[INT] .
endo
```

The user did not have to write any axioms or define any operators. It also allows the user to query for more complex objects more easily[1]. For example, if the user wanted a sequence of sets of natural, the query might be:

```
***(operations nil cons car cdr empty add member union subset)
obj MYOBJECT is
    protecting SEQUENCE[SET[NAT]] .
endo
```

Adding more objects to the set of predefined objects requires only adding the object to the *new-objects.obj* file and adding constructor terms to the *predef-terms* file so that the object can be used in matching.

3. Syntax Checking

The current implementation assumes that the syntax of OBJ3 specifications is syntactically correct. A parser could be added to the front end of the normalization routines to ensure that the user's OBJ3 is in correct form. The parser would report errors to the user, allowing the user to fix the problem before performing the normalization.

---

[1]Whether to allow a user to search for complex objects is arguable, since a complex object could be decomposed and the search performed at a lower level. The system should not, however, restrict a user from performing this search. Experience will likely dictate the overall success of searches for complex objects based on the granularity of the objects in the database.

Alternatively, a syntax-directed editor could be generated for writing OBJ3 specifications. This tool would ensure that all specifications written by a user are syntactically correct.

4. Subsort Matching in Prolog

When performing the mapping task in Prolog, predefined sorts are treated as constants and must match exactly. This means that a query for a set of natural numbers, for example, would not match to a stored component that implements a set of integers. Intuitively, this component should be among the candidates presented to the user. One solution to this problem might be to use Prolog variables for predefined sorts in the query, rather than constants. In that case it would be necessary to check the consistency of the binding to that variable and to ensure that the binding is a supersort of the sort sought..

5. Improving Efficiency

The current implementation was designed as a "proof of concept." As such, there are many inefficiencies in the system that could be improved. One of the main inefficiencies in the current implementation is the rewriting process performed during the matching phase. For each map to a candidate component, the OBJ3 environment is loaded, initialized, and then asked to perform reductions. This is a slow process. Substantial time savings are possible if all maps are tested in OBJ3 at the same time, that is, one right after another, and then scored. This way, OBJ3 is called only one time for each candidate component. Adding this feature would require some modification to the mechanism that iterates through the maps and to the scoring system.

Another potential area of improvement is in space efficiency. The implementation make heavy use of access types, but is not diligent in deallocating used space. For very large problems, wasted space could lead to a storage error that would abnormally terminate the program.

There are other situations where the performance of the semantic search may be unacceptable, such as when there are a large number of candidates or when there are a large number of maps for any particular candidate. In these cases, heuristics can be used to reduce the processing time. For example, the system could check the number of maps for each candidate and attempt to match the candidates with the fewest maps first, reporting scores as it proceeds. The user could interrupt the remainder search if a candidate looked acceptable. Another approach would be to evaluate a few of the maps at random and if none of them look promising, discard that candidate.

These are only a few suggestions. There are many more possibilities, but these are left as a subject for future research.

6. Increasing the Number of Allowable Maps

For some combinations, it is possible for a query to map to a candidate in hundreds of ways. In the current implementation, the system reads from a file all of the maps found by the Prolog mapping algorithm. With a large number of maps, this can cause a stack overflow and abnormal termination of the program. For this reason, the number of maps allowed has been limited to 50. One solution to this problem is to read only one map at a time from the file, building the the OBJ3 input file as each map is processed.

Another related problem is that for each candidate component, the query by consistency algorithm must check every possible mapping. In the worst case, this task is worse than exponential in the number of operators with identical domain and range sorts. If one allows variables in stored components, which is the case when we store generic components, the problem is exacerbated. This problem could be alleviated by analyzing mapping information to discard maps that represent alternative combinations of operator arguments for an operator that has already been successfully mapped. Another approach is to retain successful mapping results so that the same combinations are not tried again for another map, that is, perform only the equivalence checks that have not already been tried.

7. Improving Retrieval Precision

Chapter VI describes two methods for computing *ranking precision*, which are required since the semantic matching mechanism does not currently discard any components that do not appear suitable. The standard measure of precision could be employed to provide more meaningful comparisons to other component retrieval systems if the semantic matching system used heuristics to discard some candidates.

One heuristic might use a threshold value to discard components based on score. One possibility is to average the scores of all candidates and discard those that are below the average. This would work well when there are many candidates with a wide scoring distribution. The system could ignore the threshold value when there are only a few candidates or when the deviation between scores is slight.

Another heuristic, which seems obvious, is to immediately discard components whose export signatures do not map to the query signature. Currently, they are merely given a score of 0 and ranked with the other candidates. Using this heuristic introduces a tradeoff. These components are among the candidates because they mapped to the query

via their PSDL descriptions, so there is a possibility that some of them are relevant. If they are all discarded, precision will likely increase but recall may suffer.

D. SYSTEM EXTENSIONS AND SUGGESTIONS FOR FUTURE RESEARCH

1. Knuth-Bendix Completion

The current implementation makes no assumptions about the axioms for a given specification. The query by consistency method would be most effective (in terms of the heuristic measure of semantic similarity) if the axiom sets in both the query and the candidate were Church-Rosser and terminating (see Section III.C.2.b). Checking for the termination property is undecidable in the general case, although partial procedures that can handle recognizable subsets could be added. The Knuth-Bendix [KB67] completion procedure can be used to augment the system of axioms with additional axioms to make the system Church-Rosser. This process could be added to the semantic normalization routines. Implementing this extension would require extensive knowledge of term rewriting theory, the OBJ3 environment, the Knuth-Bendix completion procedure, and the Lisp programming language.

2. Theorem Proving with Axioms

Section III.C.3 described a method for using the axioms of a candidate as a theory to prove the axioms of a query. While the impediments to this process described in Section III.C.3 still remain, some theorem proving can still be done to enhance the scoring and provide better differentiation between candidate components. To implement this extension, one would need to parse the axioms of the query, replace variables with constants of the appropriate sorts, transform the axioms to the candidate component domain and then perform the proofs. This process would be straightforward for **eq** axioms which use export operators, but more difficult for **cq** axioms which would require an additional transformation [Gogu88]. For axioms that use hidden operators, the problem is more difficult .

This process could be used as an additional filter and refined scoring mechanism. Each candidate would receive credit for the number of axioms from the query that it could satisfy.

3. Mixfix Syntax

In OBJ3 a user is allowed to use mixfix syntax to define operators and axioms. The current implementation of query by consistency allows only prefix form for operators. Allowing mixfix would not alter, for better or worse, the ability of the system to match the

semantics of specifications. It would, however, provide added flexibility for users writing specifications and would make the specifications more readable. A program which performs mixfix to prefix conversion would be a useful extension to the existing system.

4. Generalization Per Category - An Alternative Phase

As seen in Chapter II, many component retrieval mechanisms use classification schemes and component attributes as a basis for multi-attribute search. McDowell's [McDo91] syntactic search is faster than a multi-attribute search and has better recall but lacks precision. Semantic search should provide the precision but if the number of candidate components is large, the search may not be timely enough. Generalization per category could be used as a mechanism to reduce the number of candidate components presented for semantic matching or to ensure that the most likely possibilities are checked first. PSDL already contains a *keywords* section that could be structured to contain attributes for describing components. These attributes would be used to eliminate components that are not applicable before invoking the semantic search mechanism. The new schema, with this mechanism in place is shown in Figure 7.1.

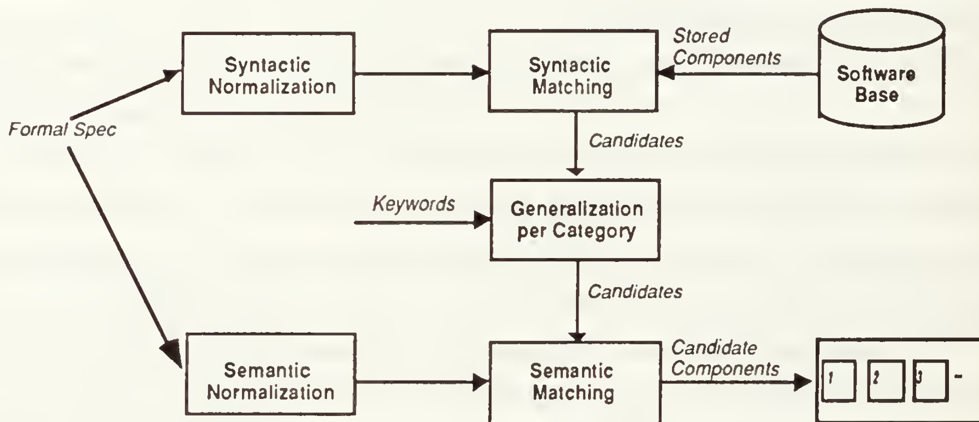


Figure 7.1 - Component Retrieval with Generalization per Category

Alternatively, generalization per category could be used after the semantic matching process to refine the scoring method. Candidates whose attributes match those of the query would be given a higher score, affecting the overall ranking of the candidate components.

5. Mapping Specifications using Ada

Prolog was chosen as the tool to map specification export signatures because the implementation was relatively fast and simple, although it did require a language transformation (from OBJ3 to Prolog predicates). Using another language and processing environment complicates the overall design of the query by consistency method and the overall design CAPS system. It also makes CAPS less portable. An alternative to using Prolog is to use Ada to perform the mapping. This does not mean that a programmer has to write a *general-purpose*, backtracking, unification algorithm in Ada. The implementation could be very specific to matching export signatures in algebraic specifications. The algorithm will need to consider all of the mapping rules described in Section IV.I.4.

6. Term Rewriting in Ada

As with the Prolog system, the requirement to have the OBJ3 system complicates the design and limits portability. A better design would have the term rewriting subsystem implemented in Ada. The foremost implementation options are to translate the OBJ3 system from Lisp to Ada, or to rewrite/redesign the system (and all hybrids in between). Another alternative is to select a different algebraic specification language whose syntax was comparable to OBJ3 and whose implementation might be more readily transformed to Ada. Any of these options would require substantial effort.

7. Query by Example

During the course of this research on query by consistency, several individuals (including myself) have questioned the practicality of requiring a user to write a formal specification for the object sought. Not all users are sophisticated enough to write formal specifications, much less correct ones. This is a valid question, for which there are several possible answers.

The first is to say that the users of CAPS are writing formal specifications in the course of defining a prototype. The QBC method simply takes advantage of that fact and uses the formal specifications that are being written anyway to locate reusable components. CAPS users must, therefore, be trained to write formal specifications.

The second answer assumes that there are trained system administrators that can help the users formulate queries for the components sought. This is, in fact, the way many organizations manage access to their large databases. If a system administrator is available to help the user, then he can use his experience to guide the user in writing a query that will lead to promising results.

A third answer to the question is to make it easier for the use to write the specification. The hardest part of a specification to write is the axioms. Instead of having the user write axioms to define behavior, he simply writes axioms that give *examples* of behavior, that is , *query by example*. For example, consider a query for a routine that sorts a list of integers. Assuming there is a predefined list object and the user knows about it, he generates the following query:

```
***(operations nil cons car cdr sort)
obj SORTFN is
    protecting LIST[INT] .
    op sort : List -> List .
    eq sort(cons(3, cons(2, cons(1, nil)))) = cons(1, cons(2, cons(3, nil))) .
    eq sort(nil) = nil .
endo
```

Assuming the user does not know about the existence of a list object, he generates the following:

```
***(operations nil cons sort)
obj SORTFN is
    sort List .
    protecting INT .
    op nil : -> List .
    op cons :  Int List -> List .
    op sort : List -> List .
    eq sort(cons(3, cons(2, cons(1, nil)))) = cons(1, cons(2, cons(3, nil))) .
    eq sort(nil) = nil .
endo
```

These are *simple* queries. This user does not need to know a lot about algebras; just the syntax for defining a signature, the way constructors recursively define terms, and what he wants in terms of inputs and outputs. In fact, this method of query is simpler for matching since it relieves the system from the burden of generating a test set.

Eichmann [Eich91] has also proposed using this method of querying with example to add semantic search capabilities to a faceted classification scheme.

E. CONCLUDING REMARKS

Automatically retrieving reusable components from a software base based on component specifications is an important factor in the meta-programming approach that is the basis of PSDL and CAPS. The use of syntactic information in a query specification can

help to filter through a large software base of components to quickly determine which subset of the components might be appropriate. The use of the semantic content of the specification further refines the search and can rank order the candidate components based on their semantic distance from the query.

The combination of formal methods, rapid prototyping, and reusable software components can vastly improve the productivity and reliability of software construction. As the software engineering discipline evolves and the demand for computer-aided software engineering tools grows, we expect to see increased emphasis in the area of reusable software component retrieval.

LIST OF REFERENCES

[Ada83] Department of Defense ANSI/MIL-STD-1815A, *Reference Manual for the Ada Programming Language*, Ada Joint Program Office, January 1983.

[AFM90] Antoy, S., Forcheri, P., and Molfino, M. T., "Specification-based Code Generation", *Proceedings of the 23rd Hawaii International Conference on System Sciences*, Kona, Hawaii, pp. 165-173, 3-5 January 1990.

[AHU83] Aho, A. V., Hopcroft, J. E., and Ullman, J.D., *Data Structures and Algorithms*, Addison-Wesley Publishing Company, 1983.

[Ande88] Air Force Armament Lab Technical Report, "The CAMP Approach: A Pragmatic Approach to Software Reuse", by C. Anderson, 1988.

[Berz91] Berzins, Valdis, Ada implementations of abstract data types for Set and Sequence, Naval Postgraduate School, Monterey, California, 1991.

[BG80] Burstall, R. M. and Goguen, J. A., "Semantics of CLEAR, a Specification Language", *Abstract Software Specifications*, D. Bjørner, ed, Springer-Verlag, 1980.

[BHK89] Bergstra, J.A., Heering, J. and Klint, P., "The Algebraic Specification Formalism ASF", *Algebraic Specification*, Addison-Wesley, 1989.

[BL91] Berzins, V. and Luqi, *Software Engineering with Abstractions*, Addison-Wesley, 1991.

[BLW90] Brown, James C., Lee, Taejae, and Werth, John, "Experimental Evaluation of a Reusability-Oriented Parallel Programming Environment", *IEEE Transactions on Software Engineering*, v. 16, pp. 111-120, February 1990.

[Booc86] Booch, Grady, *Software Components with Ada, Second Edition*, The Benjamin/Cummings Publishing Company, 1986.

[Booc87] Booch, Grady, *Software Engineering with Ada, Structures, Tools and Subsystems*, The Benjamin/Cummings Publishing Company, 1987.

[Boud85] Boudol, G., "Computational Semantics of Term Rewriting Systems", *Algebraic methods in semantics*, Maurice Nivat and John C. Reynolds, eds, Cambridge University Press, 1985.

[BP89a] Biggerstaff, T.J. and Perlis, A.J., *Software Reusability Volume I Concepts and Models*, Addison-Wesley, 1989.

[BP89b] Biggerstaff, T.J. and Perlis, A.J., *Software Reusability Volume II Applications and Experience*, Addison-Wesley, 1989.

[Burn90] Interview between Ms. Carla Burns, Rome Air Development Center, Rome, New York, and the author, 7 June 1990.

[BW87] Burton, Bruce A., Wienk, Rhonda, and others, "The Reusable Software Library", *IEEE Software*, v. 4, pp. 25-33, July 1987.

[CAMP89] Air Force Armament Laboratory, Contract F08635-88-C-0002, CDRL No. A009, *CAMP Parts Engineering System Catalog User's Guide*, McDonnell Douglas Missile Systems Company, 30 November 1989.

[CM84] Clocksin, W. and Mellish, C., *Programming in Prolog*, 2nd ed., Springer-Verlag, 1984.

[Cumm90] Cummings, Mary Ann, *The Development of User Interface Tools for the Computer Aided Prototyping System*, Master's Thesis, Naval Postgraduate School, December 1990.

[DL91] Dwyer, Andrew P., and Lewis, Gary W., *The Development of a Design Database for the Computer Aided Prototyping System*, Master's Thesis, Naval Postgraduate School, September 1991.

[Eich91] Eichmann, David A., "Selecting Reusable Components Using Algebraic Specifications", *Proceedings of the Second International Conference on Algebraic Methodology and Software Technology*, Iowa City, Iowa, pp. 37-40, 22-25 May 1991.

[EM85] Ehrig, H. and Mahr, B., *Fundamentals of Algebraic Specification 1 Equations and Initial Semantics*, Springer-Verlag, 1985.

[Fair85] Fairley, Richard, *Software Engineering Concepts*, McGraw-Hill Book Company, 1985.

[FD91] Fernandez, J.L., and De La Puente, J.A., "Constructing a Pilot Library of Components for Avionic Systems", *Proceedings of Ada-Europe International Conference*, Athens, 13-17-May 1991, Lecture Notes in Computer Science, v. 499, pp. 362-371, Springer-Verlag, 1991.

[FG90] Frakes, W. B., and Gandel, P.B., "Representing Reusable Software", *Information and Software Technology*, v. 32, pp.653-664, December 1990.

[GHM78] Guttag, J. V., Horowitz E., and Musser, D., "The Design of Data Type Specifications", *Current Trends in Programming Methodology Volume IV*, Raymond T. Yeh, ed, Prentice-Hall, Inc., 1978.

[GHW85] Guttag, J. V., Horning, J.J. and Wing, J.M., "The LARCH Family of Specification Languages", *IEEE Software*, v. 2, pp. 24-36, September 1985.

[Gold84] Goldberg, Adele, *Smalltalk-80 The Interactive Programming Environment*, Addison-Wesley Publishing Company, 1984.

[Gogu75] IBM Research Report RC5364, "Initial Algebraic Semantics", Goguen, J. A., and others, 1975.

[Gogu88] SRI International Report SRI-CSL-88-4R2, "OBJ as a Theorem Prover with Applications to Hardware Verification", Goguen, J. A., August 1988.

[GTW78] Goguen, J. A., Thatcher, J.W. and Wagner, E.G., "An Initial Approach to the Specification, Correctness, and Implementation of Abstract Data Types", *Current Trends in Programming Methodology Volume IV*, Raymond T. Yeh, ed, Prentice-Hall, Inc., 1978.

[GW88] SRI International Report SRI-CSL-88-9, "Introducing OBJ3", Goguen, J. A., and Winkler, T., August 1988.

[HHSU86] Honiden, S., and others, "Software Prototyping with Reusable Components", *Journal of Information Processing*, v. 9, pp. 123-129, March 1986.

[HO80] Huet, G. and Oppen, D., "Equations and Rewrite Rules A Survey", *Formal Language Theory Perspectives and Open Problems*, Ronald V. Book, ed, Academic Press, 1980.

[Huss85] Hussman, H., "Unification in Conditional-Equational Theories", MIP-8502, University of Passau, 1985.

[Inte88] *KEE Reference Manual*, Intellicorp, Inc., Mountain View, California, 1988.

[Inte90] *Software Through Pictures*, Interactive Development Environments, San Francisco, California, 1990.

[Jone89] Jones, O., *Introduction to the X Window System*, Prentice-Hall, 1989.

[Jone91] Jones, Carl R., "CC4001: C3 Systems: Structure, Processes and Dynamics", course notes, Naval Postgraduate School, 1991.

[KB67] Knuth, Donald E., and Bendix, Peter B., "Simple Word Problems in Universal Algebras", *Computational Problems in Abstract Algebras*, John Leech, ed., Pergamon Press, 1967.

[KM87] Kapur, D., and Musser, D., "Proof by Consistency", *Artificial Intelligence*, v. 31, pp. 125-157, February 1987.

[Kolo83] Kolodner, J. L., "Indexing and Retrieval Strategies for Natural Language Fact Retrieval", *ACM Transactions on Database Systems*, v. 8, pp. 434-464, September 1983.

[KZ89] Kapur, D., and Zhang, H., "RRL: Rewrite Rule Laboratory User's Manual", Department of Computer Science, State University of New York at Albany, May 1989.

[LVC89] Linton, M.A., Vlissides, J.M., and Calder, P.R., "Composing User Interfaces with InterViews", *IEEE Computer*, February 1989.

[LB88] Luqi, and Berzins, V., "Rapidly Prototyping Real Time Systems", *IEEE Software*, pp. 25-36, September 1988.

[LBY88] Luqi, Berzins, V., and Yeh, R. T., "A Prototyping Language for Real-Time Software", *IEEE Transactions on Software Engineering*, v. 14, pp. 1409-1423, October 1988.

[LK88] Luqi, and Ketabchi, M. A., "A Computer Aided Prototyping System", *IEEE Software*, pp. 66-72, March 1988.

[Luqi87] Luqi, "Normalized Specifications for Identifying Reusable Software", *IEEE Software*, pp. 46-49, July 1987.

[Luqi90] Luqi, "A Graph Model for Software Evolution", *IEEE Trans. on Software Engineering* , v. 16, pp. 917-927, August 1990.

[Luqi91] Luqi, "Computer-Aided Software Prototyping", *IEEE Computer,* v. 24, pp 111-112, September 1991.

[MacL87] MacLennan, Bruce J., *Principles of Programming Languages Design, Evaluation and Implementation*, Holt, Rinehart and Winston, 1987.

[MacL90] MacLennan, Bruce J., *Functional Programming Practice and Theory*, Addison-Wesley Publishing Company, 1990.

[McDo91] McDowell, J. K., *A Reusable Component Retrieval System for Prototyping*, Master's Thesis, Naval Postgraduate School, September 1991.

[MCT87] Mauldin, M., Carbonell, J., and Thomason, R., "Beyond the Keyword Barrier: Knowledge-based Information Retrieval", *Information Services and Use*, v. 7, pp. 103-117, 1987.

[Mese91] Telephone conversation between Dr. Jose Meseguer, Computer Science Laboratory, SRI International and the author, 7 March 1991.

[Meye87] Meyer, Bertrand, "Reusability: the Case for Object-Oriented Design", *IEEE Software*, v. 4, pp. 50-64, March 1987.

[Meye88a] Meyer, Bertrand, "Eiffel: Reusability and Reliability", *Software Reuse: Emerging Technology*, Will Tracz, ed, pp. 216-228, IEEE Computer Society Press, 1988.

[Meye88b] Meyer, Bertrand, *Object-Oriented Software Construction*, Prentice Hall, 1988.

[MG85] Meseguer, J. and Goguen, J. A., "Initiality, Induction, and Computability", *Algebraic methods in semantics*, Maurice Nivat and John C. Reynolds, eds, Cambridge University Press, 1985.

[MMG87] Moss, Lawrence S., Meseguer, Jose and Goguen, J. A., "Final Algebras, Cosemicomputable Algebras, and Degrees of Unsolvability", *Category Theory in Computer Science*, D. Pitt, A. Poigne, and D. Rydeheard, eds., Springer-Verlag, 1987.

[Neig84] Neighbors, James M., "The Draco Approach to Constructing Software from Reusable Components", *IEEE Transactions on Software Engineering*, v. 10, pp. 564-574, September 1984.

[Nest86] Nestor, J., "Toward a Persistent Object Base", *Advanced Programming Environments*, Springer Lecture Notes on Computer Science, 244, pp. 372-394, Springer-Verlag, 1986.

[Onto91] Ontologic, Inc., *Ontos Object Database Documentation Release 1.5*, Burlington, Massachusetts, 1991.

[Pala90] Palazzo, Frank V., *Integration of the Execution Support System for the Computer-Aided Prototyping System (CAPS)*, Master's Thesis, Naval Postgraduate School, September 1990.

[Parn72] Parnas, D. L., "On Criteria to be Used in Decomposing Systems into Modules", *Communications of the ACM*, v. 14, pp. 221-227, April 1972.

[PF87] Prieto-Diaz, Ruben, and Freeman, Peter, "Classifying Software for Reusability", *IEEE Software*, v. 4, pp. 6-16, January 1987.

[Prie85] Prieto-Diaz, Ruben, *A Software Classification Scheme*, Ph.D. Dissertation, University of California, Irvine, California, 1985.

[Prie91a] Prieto-Diaz, Ruben, "Making Software Reuse Work: An Implementation Model", *ACM Software Engineering Notes*, v. 16, pp.61-68, July 1991.

[Prie91b] Prieto-Diaz, Ruben, "Implementing Faceted Classification for Software Reuse", *Communications of the ACM* , v. 34, pp.88-97, May 1991.

[Quin90] *Quintus Prolog User's Guide*, Quintus Computer Systems, Inc., Mountain View, California, January, 1990.

[Reas86] *Refine User's Guide*, Reasoning Systems, Palo Alto, California, June 15, 1986.

[Reic87] Reichel, Horst, *Initial Computability, Algebraic Specifications and Partial Algebras*, Oxford University Press, New York, 1987.

[Reic89] Reichel, Horst, "Software Specifications by Behavioral Canons", *Algebraic Methods: Theory, Tools, and Applications*, Springer-Verlag, 1989.

[RG91] Naval Postgraduate School Report NPSCS-91-012, "Exploiting Captions in Multimedia Databases", by N. Rowe and G. Guglielmo, 1991.

[RH86] Roscoe, A.W., and Hoare, C.A.R., *The Laws of OCCAM Programming*, Technical Monograph PRG-53, Oxford University Computing Lab, Oxford, England, February 1986.

[Rott91] Telephone conversation between Sharon Rotter, Code 411, Naval Ocean Systems Center and the author, 13 March 1991.

[Rowe88] Rowe, Neil C., *Artificial Intelligence Through Prolog*, Prentice-Hall, 1988.

[RT89] Colin Runciman, and Ian Toyn, "Retrieving Reusable Software Components by Polymorphic Type", *Proceedings of the International Conference on Functional Programming and Computer Architecture (FPCA '89)*, New Orleans, pp. 166-173, 1989.

[RW88] Rich, Charles, and Waters, Richard C., "The Programmer's Apprentice: A Research Overview", *Computer*, v. 21, pp. 11-25, November 1988.

[RW89] Rich, Charles, and Waters, Richard C., "Formalizing Reusable Software Components in the Programmer's Apprentice", *Software Reusability Applications and Experience*, Addison-Wesley, 1989.

[RW90a] Rich, Charles, and Waters, Richard C., *The Programmer's Apprentice*, Addison-Wesley, 1990.

[RW90b] Rich, Charles, and Wills, Linda M., "Recognizing a Program's Design: A Graph-Parsing Approach", *IEEE Software*, v. 7, pp. 82-89, January 1990.

[RW90c] Rollins, Eugene J., and Wing, Jeanette M., "Specifications as Search Keys for SW Libraries: A Case Study using Lambda Prolog", CMU-CS-90-159, Carnegie Mellon University, 26 September 1990.

[Self90] Self, John, "Aflex - An Ada Lexical Analyzer Generator Version 1.1", UCI-90-18, Department of Information and Computer Science, University of California, Irvine, California, May 1990.

[SM83] Salton, G. and McGill, M., *Introduction to Modern Information Retrieval*, McGraw-Hill, 1983.

[SPS91] Software Productivity Solutions, "ARCS: Automated Reusable Software Toolset", Indialantic, Florida, 1991.

[SRI88] *Introducing OBJ3*, by J. Goguen and T. Winkler, SRI-CSL-88-9, SRI International, Menlo Park, California, August 1988.

[Stan84] Standish, T.A., "An Essay on Software Reuse", *IEEE Transactions on Software Engineering*, v. SE-10, p. 494, September 1984.

[Stei86] Steigerwald, R. A., *Reusable Software Components*, Master's Thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, January 1986.

[TTS88] Taback, D., Tolani, D., and Schmalz, R., "Ayacc User's Manual Version 1.0", UCI-88-16, Department of Information and Computer Science, University of California, Irvine, California, May 1988.

[Verd91] *Verdix Ada Development System 6.0*, Verdix Corporation, Chantilly, Virginia, 1991.

[VR90] U.S. Army Information Systems Software Development Center - Washington, "Reusable Ada Products for Information Systems Development (RAPID) Lessons Learned During Pilot Operations", Vogelsong, Terry, and Rothrock, Jack, 1990.

[Voge89] Vogelsong, Terry, "Reusable Ada Packages for Information Systems Development (RAPID): An Operational Center of Excellence for Software Reuse", *Proceedings of Tri-Ada '89*, Pittsburgh, PA., Association for Computing Machinery, 1989.

[Wink91] Timothy Winkler, "Introducing OBJ3's New Features", SRI International Report (preliminary version provided by the author), March 1991.

[Wirs88] Wirsing, M., "Algebraic Description of Reusable Software Components", *Proceedings of COMPEURO 88*, IEEE Computer Society Press, pp. 300-312, 1988.

[WS88] Wood, Murray, and Sommerville, Ian, "An Information Retrieval System for Software Components", *SIGIR Forum,* v. 22, pp. 11-28, Spring/Summer, 1988.

[Zill80] Zilles, S. N., "Introduction to Data Algebra", *Abstract Software Specifications*, D. Bjørner, ed, Springer-Verlag, 1980.

APPENDIX

I. SOURCE CODE

This appendix contains the source code for the implementation of the system described in this dissertation. Section A contains the Ada [Verd91] code for normalization processes. Section B contains the Ada code for matching processes. Section C contains the input source for the lexical analyzer [Self90] and parser [TTS88] generators. Section D contains the Prolog [Quin90] source code used for mapping. Section E contains the Lisp source code used to modify the processes of OBJ3. Section F contains definitions of the predefined OBJ3 objects used in query by consistency, which are simply prefix reformulations of the predefined objects provided by OBJ3 [SRI88]. Section G contains various support files.

A. ADA SOURCE CODE FOR NORMALIZATION

```
--------------------------------------------------------------------------------
-- Normalize is the main executable for the normalization process.
--------------------------------------------------------------------------------
with IO_Exceptions, A_Strings, Unix_Prcs, U_Env, Text_IO;
with Types_and_Constants, Formal_Spec_Object, Check_Spec_Syntax;
with Obj3_Tokens; use Obj3_Tokens;
with Obj3_Lex, Obj3_Lex_IO, Obj3_Lex_Dfa;

procedure Normalize is
    Spec_Filename  : A_Strings.A_String;
    Formal_Spec    : Formal_Spec_Object.Formal_Spec_Def;
    Error_Present : Boolean;
    No_Filename, No_OBJ_Extension : exception;

    procedure Make_Normalized_File is separate;

    procedure Make_Set_of_Ops is separate;

    begin
        if U_Env.argv'Last > 0 then
            Spec_Filename := U_Env.argv(1);
            if (Spec_Filename.s'Last > 4) then
                if  (Spec_Filename.s(Spec_Filename.s'Last-3..Spec_Filename.s'Last)
                    /= ".obj") then
                        raise No_OBJ_extension;
                end if;
            else
```

```
            raise No_OBJ_extension;
         end if;
      else
         raise No_Filename;
      end if;

      Text_IO.Put_Line("Normalizing: " & Spec_Filename.s);
      Check_Spec_Syntax(Spec_Filename, Error_Present);
      if Error_Present then
         Text_IO.Put_Line("Formal Spec contains an Error");
      else
         Make_Set_of_Ops; --(Spec_Filename, Formal_Spec);
         Make_Normalized_File;
      end if;

      exception
         when No_Filename =>
            Text_IO.Put_Line("Usage is:  normalize filename.obj");
            raise IO_Exceptions.Name_Error;
         when No_OBJ_extension =>
            Text_IO.Put_Line("Filename must have '.obj' extension!");
            raise IO_Exceptions.Name_Error;
         when IO_Exceptions.Name_Error =>
            Text_IO.Put_Line("Could not find file: " & Spec_Filename.s);
            raise IO_Exceptions.Name_Error;
end Normalize;

-----------------------------------------------------------------------------------------------
-- Check_Spec_Syntax is a stubbed process.  It should be expanded in future systems.
-----------------------------------------------------------------------------------------------
with Text_IO, Unix_Prcs, A_Strings;

procedure Check_Spec_Syntax(
       Spec_Filename   : in A_Strings.A_String;
      Error            : out Boolean) is

    Shell_Script_Cmd_Line   : A_Strings.A_String;
    Temp                    : Integer;

begin
    --Text_IO.Put_Line("Starting Procedure Check_Spec_Syntax");
    Error := False;
    --Text_IO.Put_Line("Completed Procedure Check_Spec_Syntax");
end Check_Spec_Syntax;

-----------------------------------------------------------------------------------------------
-- Make_Set_of_Ops performs lexical analysis of a given formal specification to find the
-- ops comment token and extract the export operator names, placing them in a set.
-----------------------------------------------------------------------------------------------
separate (Normalize)
procedure Make_Set_of_Ops is
```

```
    Atoken                : Obj3_Tokens.Token;
    Temp                  : A_Strings.A_String;
    No_Ops_Comment        : exception;
    Bad_Op_Name           : exception;

begin
    Obj3_Lex_IO.Open_Input(Spec_Filename.s);
    loop -- to look for the ops-comment which lists the export operations
        Atoken := Obj3_Lex.Yylex;
        exit when (Atoken = End_of_Input) or (Atoken = Ops_Comment_Token);
    end loop;
    if not (Atoken = Ops_Comment_Token) then
        raise No_Ops_Comment;
    end if;
    Types_and_Constants.Op_Set_Pkg.Empty(Formal_Spec.Set_of_Ops);
    loop
        Atoken := Obj3_Lex.yylex;
        Temp := A_Strings.to_a(Obj3_Lex_Dfa.yytext);
        if Temp.s /= ")" then
            if Temp.s(1) < 'a' or (Temp.s(1) > 'z') then
                raise Bad_Op_Name;
            end if;
            for i in Temp.s'First+1..Temp.s'Last loop
                if (Temp.s(i) < '0') or ((Temp.s(i) > '9') and (Temp.s(i) < 'a')) or
                        (Temp.s(i) > 'z') then
                    raise Bad_Op_Name;
                end if;
            end loop;
            Types_and_Constants.Op_Set_Pkg.Add(Temp, Formal_Spec.Set_of_Ops);
        end if;
        exit when Obj3_Lex_Dfa.yytext = ")";
    end loop;
    Obj3_Lex_IO.Close_Input;
    Text_IO.Put(Spec_Filename.s & " exports");
    Text_IO.Put(Integer'Image(Types_and_Constants.Op_Set_Pkg.
            Size(Formal_Spec.Set_of_Ops)));
    Text_IO.Put_Line(" operations.");

    exception
        when No_Ops_Comment =>
            Text_IO.Put("File to be normalized must contain an OBJ3 ");
            Text_IO.Put_Line("comment of the form:");
            Text_IO.Put_Line("     ***(operations op1 op2 op3");
            Text_IO.Put("Where op1, op2, etc are the names of the ");
            Text_IO.Put_Line("operations this module will export.");
            raise Constraint_Error;
        when Bad_Op_Name =>
            Text_IO.Put_Line(Temp.s & " is an illegal op name.");
            Text_IO.Put_Line("Op names must adhere to: [a-z][a-z0-9]*");
            raise Constraint_Error;
end Make_Set_of_Ops;
```

```
-----------------------------------------------------------------------------------------------
-- Make_Normalized_File invokes OBJ3 to expand a given specification, my_spec.obj,
-- extracts data from the specification, and stores it into a file, my_spec.obj.norm.
-----------------------------------------------------------------------------------------------
with Text_IO; use Text_IO;
with  Clean_Normalized_File;
with  Make_Prolog_For_Stored;

separate (Normalize)
procedure Make_Normalized_File is
    Temp_Script_Name,
    Temp_Shell_Name :  A_Strings.A_String;
    Obj_Temp_File,
    Obj_Shell_File :       Text_IO.File_Type;
    Command_Line,
    New_Name :             A_Strings.A_String;
    Temp :                 Integer;
begin
    Temp_Script_Name := A_Strings.to_a(Spec_Filename.s & ".script.obj");
    Text_IO.Create(Obj_Temp_File, Out_File, Temp_Script_Name.s);
    Command_Line := A_Strings.to_a("chmod 777 " & Temp_Script_Name.s);
    Temp := Unix_Prcs.Spawn(Command_Line);
    Text_IO.Put_Line(Obj_Temp_File, "in  newlisp.obj");
    Text_IO.Put(Obj_Temp_File, "in ");
    Text_IO.Put_Line(Obj_Temp_File, Spec_Filename.s);
    Text_IO.Put_Line(Obj_Temp_File, "ev (print-mod-name)");
    Text_IO.Put_Line(Obj_Temp_File, "ev (print-ps)");
    Text_IO.Put_Line(Obj_Temp_File, "ev (print-ops)");
    Text_IO.Put_Line(Obj_Temp_File, "ev (print-sorts)");
    -- axioms not used in normalization
    --Text_IO.Put_Line(Obj_Temp_File, "ev (print-axioms)");
    Text_IO.Put_Line(Obj_Temp_File, "ev (print-generics)");
    Text_IO.Put_Line(Obj_Temp_File, "q");
    Temp_Shell_Name := A_Strings.to_a(Spec_Filename.s & ".shell");
    Text_IO.Create(Obj_Shell_File, Out_File, Temp_Shell_Name.s);
    Text_IO.Put_Line(Obj_Shell_File, "obj <$1 >$2");
    Command_Line := A_Strings.to_a("chmod 777 " & Temp_Shell_Name.s);
    Temp := Unix_Prcs.Spawn(Command_Line);
    New_name := A_Strings."&"(Spec_Filename, ".norm");
    Command_Line := A_Strings.to_a(Temp_Shell_Name.s & " " &
                    Temp_Script_Name.s & " " & New_name.s);
    Text_IO.Put_Line("Running OBJ3 task to expand module");
    Temp := Unix_Prcs.Spawn(Command_Line);
    Text_IO.Put_Line("Finished OBJ3 task");
    Text_IO.Delete(Obj_Temp_File);
    Text_IO.Delete(Obj_Shell_File);
    Clean_Normalized_File(New_Name);
    Make_Prolog_for_Stored(New_Name, Formal_Spec.Set_of_Ops);
    Text_IO.Put_Line("File: " & New_Name.s & " created.");
end  Make_Normalized_File;
```

```
----------------------------------------------------------------------------------------
-- Clean_Normalized_File removes extraneous OBJ3 output from the .norm file
----------------------------------------------------------------------------------------

with Text_IO; use Text_IO;
with A_Strings;
with Unix_Prcs;

procedure Clean_Normalized_File
    (File_Name : in A_Strings.A_String) is

    Temp_File,
    Norm_File      : Text_IO.File_Type;
    Line           : String(1..1000);
    Cmd_Line       : A_Strings.A_String;
    Temp           : Integer;
    Line_length    : Natural;
begin
    Text_IO.Open(Norm_File, In_File, File_Name.s);
    Text_IO.Create(Temp_File, Out_File, File_Name.s & ".temp");
    while not End_of_File(Norm_File)
    loop
        while not End_of_File(Norm_File)
        loop
            Text_IO.Get_Line(Norm_File, Line, Line_Length);
            if Line_Length > 3 then
                if Line(1..3) = "!!!" then
                    Put_Line(Temp_File, Line(1..Line_Length));
                end if;
                exit when Line(1..3) = "!!!";
            end if;
        end loop;
        while not End_of_File(Norm_File)
        loop
            Text_IO.Get_Line(Norm_File, Line, Line_Length);
            Put_Line(Temp_File, Line(1..Line_Length));
            if Line_Length > 3 then
                exit when Line(1..3) = "!!!";
            end if;
        end loop;
    end loop;
    Text_IO.Close(Temp_File);
    Cmd_Line := A_Strings.to_a("mv " & File_Name.s & ".temp "
             & File_Name.s);
    Text_IO.Delete(Norm_File);
    Temp := Unix_Prcs.Spawn(Cmd_Line);
end Clean_Normalized_File;
```

```
-------------------------------------------------------------------------------------------
-- Make_Prolog_for_Stored transforms a specification's export signature into a Prolog
-- database of facts for use in mapping components.  The Prolog code is stored in the .norm
-- file.
-------------------------------------------------------------------------------------------
with Unix_Prcs, A_Strings;
with Text_IO;           use Text_IO;
with Obj3_Tokens;       use Obj3_Tokens;
with Obj3_Lex, Obj3_Lex_IO, Obj3_Lex_Dfa;
with Types_And_Constants, Get_Generic_Sorts;

procedure Make_Prolog_for_Stored
        (File_Name     : A_Strings.A_String;
         Set_Of_Ops    : Types_and_Constants.Op_Set_Pkg.Set) is

    Temp_File,
    Cat_Shell                  : Text_IO.File_Type;
    Sort_Name,
    Op_Name,
    Principal_Sort,
    Command_Line,
    Generic_Predicate          : A_Strings.A_String;
    Tok                        : Obj3_Tokens.Token;
    Bad_Op_Name                : exception;
    Position                   : Natural;
    Temp                       : Integer;
    Add_Comma                  : Boolean := False;
    Generic_Parameter_Seq      : Types_and_Constants.A_String_Seq_Pkg.Sequence;

    procedure Make_Argument_Predicate(Temp_File :Text_IO.File_Type) is separate;
    procedure Make_Op_Predicate(Temp_File :Text_IO.File_Type) is separate;

begin
    Text_IO.Create(Temp_File, Out_File, File_Name.s & ".temp");
    Principal_Sort := Get_Principal_Sort(File_Name);
    Generic_Parameter_Seq := Get_Generic_Sorts(File_Name);
    Obj3_Lex_IO.Open_Input(File_Name.s);
    Text_IO.Put_Line(Temp_File, "!!!prolog");
    Generic_Predicate := A_Strings.to_a("generic([");
    loop
        Tok := Obj3_Lex.yylex;
        exit when Tok = Ops_Start_Token;
    end loop;
    Tok := Obj3_Lex.yylex;    -- an op token
    loop
        Position := 0;        -- position of the domain arguments
        Tok := Obj3_Lex.yylex; -- an op-name token
        Op_Name := A_Strings.to_a(Obj3_Lex_Dfa.yytext);
        if Types_and_Constants.Op_Set_Pkg.Member(Op_Name, Set_of_Ops) then
            Tok := Obj3_Lex.yylex;    -- a colon token
```

```
            loop
                Tok := Obj3_Lex.yylex; -- Sort or Arrow token
                exit when Tok = Arrow_Token;
                Position := Position + 1;
                Sort_Name  := A_Strings.to_a(Obj3_Lex_Dfa.yytext);
                Make_Argument_Predicate(Temp_File);
            end loop;
            Tok := Obj3_Lex.yylex;    -- range sort token
            Sort_Name  := A_Strings.to_a(Obj3_Lex_Dfa.yytext);
            Make_Op_Predicate(Temp_File);
            Tok := Obj3_Lex.yylex;    -- end expression token
            Tok := Obj3_Lex.yylex;    -- next op or ops_end token
        else  -- the op-name was not a member of the export set
            loop -- skip this op definition
                Tok := Obj3_Lex.yylex; -- any token
                exit when (Tok = Op_Token) or (Tok = Ops_End_Token);
            end loop;
        end if;
        exit when Tok = Ops_End_Token;
    end loop;
    Generic_Predicate  := A_Strings."&"(Generic_Predicate,  "]).");
    Text_IO.Put_Line(Temp_File,  Generic_Predicate.s);
    Text_IO.Put_Line(Temp_File,  "!!!end-prolog");
    Obj3_Lex_IO.Close_Input;
    Text_IO.Create(Cat_Shell, Out_File, File_Name.s & ".shell");
    Text_IO.Put_Line(Cat_Shell, "cat $1 >> $2");
    Command_Line  := A_Strings.to_a("chmod 777 " & File_Name.s & ".shell");
    Temp  := Unix_Prcs.Spawn(Command_Line);
    Command_Line  := A_Strings."&"(A_Strings.to_a
        (File_Name.s & ".shell " & File_Name.s & ".temp "), File_name);
    Temp  := Unix_Prcs.Spawn(Command_Line);
    Text_IO.Delete(Cat_Shell);
    Text_IO.Delete(Temp_File);

    exception
        when Bad_Op_Name =>
            Text_IO.Put_Line("Processing aborted: Op names must be [a-z][a-z0-9]*");
end  Make_Prolog_for_Stored;

---------------------------------------------------------------------------------------------------
-- Get_Generic_Sorts performs lexical analysis on  a .norm file to extract the generic
--  parameter names from the specification and store them in a sequence.
---------------------------------------------------------------------------------------------------
with A_Strings;
with Text_IO;           use Text_IO;
with Obj3_Tokens;       use Obj3_Tokens;
with Obj3_Lex, Obj3_Lex_IO, Obj3_Lex_Dfa;
with Types_And_Constants;

function Get_Generic_Sorts
    (File_Name   : A_Strings.A_String)
```

```
    return Types_and_Constants.A_String_Seq_Pkg.Sequence is

    Tok                 : Obj3_Tokens.Token;
    A_Seq               : Types_and_Constants.A_String_Seq_Pkg.Sequence;
    Generics_Flag       : Boolean := false;

begin
    Obj3_Lex_IO.Open_Input(File_Name.s);
    --Text_IO.Put_Line("Opened file: " & File_Name.s);
    A_Seq := Types_and_Constants.A_String_Seq_Pkg.Empty;
    loop
        Tok := Obj3_Lex.yylex;
        exit when (Tok = Generics_Start_Token) or (Tok = End_of_Input);
    end loop;
    loop
        Tok := Obj3_Lex.yylex;
        exit when (Tok = Generics_End_Token) or (Tok = End_of_Input);
        Types_and_Constants.A_String_Seq_Pkg.Add
            (A_Strings.to_a(Obj3_Lex_Dfa.yytext), A_Seq);
        Generics_Flag := true;
    end loop;
    Obj3_Lex_IO.Close_Input;
    if Generics_Flag then
        Text_IO.Put("Generic parameters are: ");
        for i in 1..Types_and_Constants.A_String_Seq_Pkg.length(A_Seq) loop
            Text_IO.Put
                (Types_and_Constants.A_String_Seq_Pkg.Fetch(A_Seq, i).s & " ");
        end loop;
        Text_IO.New_Line;
    end if;
  return A_Seq;
end Get_Generic_Sorts;

----------------------------------------------------------------------------------------
-- Make_Op_Predicate makes an individual operator predicate for each export operator.

----------------------------------------------------------------------------------------

with Types_and_Constants;

separate (make_prolog_for_stored)
procedure Make_Op_Predicate
    (Temp_File : Text_IO.File_Type) is

    Generic_Predicate_Part   : A_Strings.A_String;
    Generic_Location         : Natural := 0;

    function Contains(Pattern, S: A_Strings.A_String; start: natural:=1)
            return Boolean is
        len_less_one: integer := Pattern.len - 1;
    begin
```

```
        for i in start .. S.len - len_less_one loop
            if S.s(i..i+len_less_one) = Pattern.s then
            return true;
        end if;
    end loop;
    return false;
    end;

    function Contains(Str : A_Strings.A_String; C : Character)
            return Boolean is
    begin
        for Counter in Str.s'First .. Str.s'Last loop
            if C = Str.s(Counter) then
                return true;
            end if;
        end loop;
        return false;
    end Contains;

begin
    Text_IO.Put(Temp_File, "operator(");

    -- if the sort is a qualified sort or starts with Elt then it is generic
    -- This is not true in the general case with OBJ3 but it is the case
    -- with our restricted grammar

    if Contains(Sort_Name, '.') or
            ((Sort_Name.s'Length >= 3) and Sort_Name.s(1..3) = "Elt") then
        Put(Temp_File, "_, "); -- generic; in Prolog will bind to anything
        for Counter in 1..Types_and_Constants.A_String_Seq_Pkg.
            Length(Generic_Parameter_Seq)
        loop
            if Contains(A_Strings."&"( Types_and_Constants.A_String_Seq_Pkg.
                    Fetch(Generic_Parameter_Seq, Counter), "::"), Sort_Name, 1) then
                Generic_Location := Counter;
                exit;
            end if;
        end loop;
        if (Sort_Name.s = "Elt") and Types_and_Constants.A_String_Seq_Pkg.
                Length(Generic_Parameter_Seq) = 1) then
            Generic_Location := 1;
        end if;
        Generic_Predicate_Part := A_Strings.to_a("");
        if Add_Comma then
            Generic_Predicate_Part := A_Strings."&"(Generic_Predicate_Part, ", ");
        end if;
        Add_Comma := True;
        Generic_Predicate_Part := A_Strings."&"(Generic_Predicate_Part, "[");
        Generic_Predicate_Part := A_Strings."&"(Generic_Predicate_Part, Op_Name);
        Generic_Predicate_Part := A_Strings."&"(Generic_Predicate_Part, ", 0, ");
        Generic_Predicate_Part := A_Strings."&"(Generic_Predicate_Part,
```

```
            A_Strings.Upper_to_Lower(Types_and_Constants.A_String_Seq_Pkg.
            Fetch(Generic_Parameter_Seq, Generic_Location)));
        Generic_Predicate_Part := A_Strings."&"(Generic_Predicate_Part, ",");
        Generic_Predicate_Part := A_Strings."&"(Generic_Predicate_Part,
            Natural'Image(Generic_Location) & "]");
        Generic_Predicate := A_Strings."&"(Generic_Predicate,
            Generic_Predicate_Part);
    else
        Put(Temp_File, A_Strings.Upper_To_Lower(Sort_Name).s & ",");
    end if;
    Text_IO.Put(Temp_File, Natural'Image(Position) & ", ");
    Text_IO.Put_Line(Temp_File, A_Strings.Upper_To_Lower(Op_Name).s & ").");

    exception
        when Constraint_Error =>
            Text_IO.Put_Line("Aborted in procedure: Make_Op_Predicate");
            Text_IO.Put_Line("Generic sort name is: " & Sort_Name.s);
end Make_Op_Predicate;

----------------------------------------------------------------------------------------------
-- Make_Argument_Predicate makes an individual argument predicate for an argument
-- of an export operator. It checks to see if the argument is generic or predefined.
----------------------------------------------------------------------------------------------
with Types_and_Constants;

separate (make_prolog_for_stored)
procedure Make_Argument_Predicate
    (Temp_File : Text_IO.File_Type) is

    Generic_Predicate_Part   : A_Strings.A_String;
    Generic_Location         : Natural := 0;

    function Contains(Pattern, S: A_Strings.A_String; start: natural:=1)
        return Boolean is
        len_less_one: integer := Pattern.len - 1;
    begin
        for i in start .. S.len - len_less_one loop
            if S.s(i..i+len_less_one) = Pattern.s then
                return true;
            end if;
        end loop;
        return false;
    end;

    function Contains(Str : A_Strings.A_String; C : Character) return Boolean is
    begin
        for Counter in Str.s'First .. Str.s'Last loop
            if C = Str.s(Counter) then
                return true;
            end if;
        end loop;
```

```
        return false;
    end Contains;

begin
    Text_IO.Put(Temp_File, "argument(");
    Text_IO.Put(Temp_File, A_Strings.Upper_To_Lower(Op_Name).s & ", ");

    -- if the sort is a qualified sort or starts with Elt then it is generic
    -- This is not true in the general case with OBJ3 but it is the case
    -- with our restricted grammar

    if Contains(Sort_Name, '.') or
            ((Sort_Name.s'Length >= 3) and Sort_Name.s(1..3) = "Elt") then
        Put(Temp_File, "_, "); -- generic; in Prolog, binds to anything
        for Counter in 1..Types_and_Constants.
                A_String_Seq_Pkg.Length(Generic_Parameter_Seq)
        loop
            if Contains(A_Strings."&"(Types_and_Constants.A_String_Seq_Pkg.
                    Fetch(Generic_Parameter_Seq,Counter), "::"), Sort_Name, 1) then
                Generic_Location := Counter;
                exit;
            end if;
        end loop;
        if (Sort_Name.s = "Elt") and (Types_and_Constants.A_String_Seq_Pkg.
                Length(Generic_Parameter_Seq) = 1) then
            Generic_Location := 1;
        end if;
        Generic_Predicate_Part := A_Strings.to_a("");
        if Add_Comma then
            Generic_Predicate_Part := A_Strings."&"(Generic_Predicate_Part, ", ");
        end if;
        Add_Comma := True;
        Generic_Predicate_Part := A_Strings."&"(Generic_Predicate_Part, "[");
        Generic_Predicate_Part := A_Strings."&"(Generic_Predicate_Part, Op_Name);
        Generic_Predicate_Part := A_Strings."&"(Generic_Predicate_Part,
            "," & Natural'Image(Position) & ", ");
        Generic_Predicate_Part := A_Strings."&"(Generic_Predicate_Part,
            A_Strings.Upper_to_Lower(Types_and_Constants.A_String_Seq_Pkg.
            Fetch(Generic_Parameter_Seq, Generic_Location)));
        Generic_Predicate_Part := A_Strings."&"(Generic_Predicate_Part, ",");
        Generic_Predicate_Part := A_Strings."&"(Generic_Predicate_Part,
            Natural'Image(Generic_Location) & "]");
        Generic_Predicate := A_Strings."&"(Generic_Predicate,
            Generic_Predicate_Part);
    else
        Put(Temp_File, A_Strings.Upper_To_Lower(Sort_Name).s & ",");
    end if;
    Text_IO.Put_Line(Temp_File, Natural'Image(Position) & ").");

    exception
        when Constraint_Error =>
```

```
            Text_IO.Put_Line("Aborted in procedure: Make_Argument_Predicate");
            Text_IO.Put_Line("Generic sort name is: " & Sort_Name.s);
end Make_Argument_Predicate;

------------------------------------------------------------------------------------------------
-- Types_And_Constants defines important structures for use through the normalization
--  process.  During instantiation, this package opens and reads a file called
-- "predefined-sorts". This file must be present.
------------------------------------------------------------------------------------------------
with Text_IO; use Text_IO;
with A_Strings, Set_Pkg, Sequence_Pkg;

package Types_And_Constants is

    Max_Maps                : constant := 50;
    Spec_Filename_Type      : A_Strings.A_String;
    Op_Name_Type            : A_Strings.A_String;

    function Equal(X, Y : A_Strings.A_String) return Boolean;

    package Predefined_Obj_Sorts_Pkg is new
        Set_Pk(t => A_Strings.A_String, eq => Equal);

    package Op_Set_Pkg is new
        Set_Pkg(t => A_Strings.A_String, eq => Equal);

    Predef_Obj_Sorts_Set : Predefined_Obj_Sorts_Pkg.Set;

    package A_String_Seq_Pkg is new
        Sequence_Pkg(t => A_Strings.A_String);

end Types_And_Constants;

package body Types_And_Constants is
    Sort_File          : File_Type;
    Sort_Name          : String(1..32);
    Name_Length        : Natural;

    function Equal(X, Y : A_Strings.A_String) return Boolean is
        Result : Boolean;
    begin
        Result := X.s = Y.s;
        return Result;
    end Equal;

    procedure Print_A_String(X : in A_Strings.A_String) is
    begin
        Text_IO.Put(X.s & " ");
    end Print_A_String;

    procedure Scan_Set is new
```

```
        Predefined_Obj_Sorts_Pkg.Scan(generate => Print_A_String);

begin
     Predefined_Obj_Sorts_Pkg.Empty(Predef_Obj_Sorts_Set);
     Text_IO.Open(Sort_File, In_File, "predefined-sorts");
     while not End_of_File(Sort_file) loop
        Text_IO.Get_Line(Sort_File, Sort_Name, Name_Length);
        Predefined_Obj_Sorts_Pkg.Add(A_Strings.to_a(Sort_name(1..Name_Length)),
                  Predef_Obj_Sorts_Set);
     end loop;
     Text_IO.Close(Sort_File);
     --Text_IO.Put("Predefined sorts are: ");
     --Scan_Set(Predef_Obj_Sorts_Set);
     --Text_IO.New_Line;
end Types_And_Constants;

with Types_And_Constants;
with A_Strings;

-----------------------------------------------------------------------------------------------
-- Formal_Spec_Object defines a set of a specifications export operators.
-----------------------------------------------------------------------------------------------
package Formal_Spec_Object is
     type Formal_Spec_Def is
     record
        Set_of_Ops : Types_and_Constants.Op_Set_Pkg.Set;
     end record;
end Formal_Spec_Object;
```

B. ADA SOURCE CODE FOR MATCHING

```
-----------------------------------------------------------------------------------------------
-- Match_Candidates is the main executable for the matching process.
-----------------------------------------------------------------------------------------------
with U_Env, A_Strings, Unchecked_Deallocation;
with Text_IO;                  use Text_IO;
with Formal_Spec_Object;       use Formal_Spec_Object;
with Term_Definition_Pkg;      use Term_Definition_Pkg;
with Op_Defns_Pkg;             use Op_Defns_Pkg;
with Match, Build_Test_Set, Make_IO_List, Generate_Output_Terms;

procedure Match_Candidates is

     Query_Filename,
     Candidates_Filename,
     Scores_Filename,
     Cand_Filename    : A_Strings.A_String;
     Formal_Spec      : Formal_Spec_Object.Formal_Spec_Def;
     Test_Set         : Term_Definition_Pkg.Test_Set_Def;
```

```
    IO_List             : Term_Definition_Pkg.IO_List_Def;
    Score, Length       : Natural := 0;
    Scores_File         : Text_IO.File_Type;
    Candidates_File     : Text_IO.File_Type;
    A_Candidate         : String(1..80);

    procedure Norm_Query
        (Query_Filename     : in A_Strings.A_String;
        Formal_Spec         : in out Formal_Spec_Def) is separate;

    procedure Free is new Unchecked_Deallocation
        (Object => Maps,
        Name   => Map_Access);

begin
    Query_Filename := U_env.argv(1);
    Candidates_Filename := U_env.argv(2);
    Scores_Filename := U_env.argv(3);

    Formal_Spec := new Formal_Spec_Record;
    Norm_Query(Query_Filename, Formal_Spec);
    Build_Test_Set(Query_Filename, Formal_Spec, Test_Set);
    Make_IO_List(Test_Set, Formal_Spec, IO_List);
    Generate_Output_Terms(Query_Filename, Formal_Spec, Test_Set, IO_List);

    Text_IO.Open(Candidates_File, In_File, Candidates_Filename.s);
    Text_IO.Create(Scores_File, Out_File, Scores_Filename.s);
    while not End_of_File(Candidates_File)
    loop
        Get_Line(Candidates_File, A_Candidate, Length);
        Cand_Filename := A_Strings.to_a(A_Candidate(1..Length));
        Free(Formal_Spec.Comp_Maps);
        Match(Query_Filename, Cand_Filename, Score,
                Formal_Spec, Test_Set, IO_List);
        Text_IO.Put_Line(Scores_File, Natural'Image(Score));
    end loop;
end Match_Candidates;
```

1. Normalize Query

```
-----------------------------------------------------------------------------------------------------
-- Norm_Query calls Create_from_Query.  This level of indirection should be removed.
-- Also, Checking specification syntax is not supported.
-----------------------------------------------------------------------------------------------------
separate (Match_Candidates)
procedure Norm_Query
    (Query_Filename : in A_Strings.A_String;
    Formal_Spec : in out Formal_Spec_Object.Formal_Spec_Def) is

    Error_Present : Boolean := False;
```

```
begin
    Text_IO.Put_Line("Normalizing Query: " & Query_Filename.s);
    --Check_Spec_Syntax(Query_Filename, Error_Present);
    if Error_Present then
        Text_IO.Put_Line("Formal Spec contains an Error");
    else
        Formal_Spec_Object.Create_from_Query(Query_Filename, Formal_Spec);
    end if;
end Norm_Query;

------------------------------------------------------------------------------------------------
-- The Formal_Spec_Object package defines a formal specification and a procedure
-- Create_from_Query which performs normalization.
------------------------------------------------------------------------------------------------
with Text_IO; use Text_IO;
with Types_And_Constants;
with A_Strings;
with Set_Pkg;
with Op_Defns_Pkg;

package Formal_Spec_Object is

    type Formal_Spec_Record is
    record
        Set_of_Ops      : Types_and_Constants.Op_Set_Pkg.Set;
        Op_Defns        : Op_Defns_Pkg.Op_Defn_Seq_Pkg.Sequence;
        Hidden_Ops      : Op_Defns_Pkg.Op_Defn_Seq_Pkg.Sequence;
        Comp_Maps       : Op_Defns_Pkg.Map_Access;
    end record;

    type Formal_Spec_Def is access Formal_Spec_Record;

    function Equal(X, Y : A_Strings.A_String) return Boolean;

    package Op_Name_Set_Pkg is new Set_Pkg
        (t => A_Strings.A_String, eq => Equal);

    procedure Create_from_Query
        (Spec_Filename  : in A_Strings.A_String;
        Formal_Spec     : in out Formal_Spec_Def);
end Formal_Spec_Object;

------------------------------------------------------------------------------------------------
-- The body for the Formal_Spec_Object package.
------------------------------------------------------------------------------------------------
with Obj3_Tokens; use Obj3_Tokens;
with Obj3_Lex, Obj3_Lex_IO, Obj3_Lex_Dfa;
with Unix_Prcs, Types_and_Constants;

package body Formal_Spec_Object is
```

```
  function Equal
     (X, Y : A_Strings.A_String) return Boolean is

     Result : Boolean;

     begin
         Result := X.s = Y.s;
         return Result;
     end Equal;

     procedure Create_from_Query(
         Spec_Filename     : in  A_Strings.A_String;
         Formal_Spec       : in out Formal_Spec_Def) is separate;
end Formal_Spec_Object;

----------------------------------------------------------------------------------------------------
-- Create from Query - a procedure to create a normalized query from
-- a given query specification.
----------------------------------------------------------------------------------------------------
separate (Formal_Spec_Object)
procedure Create_from_Query
     (Spec_Filename : in A_Strings.A_String;
     Formal_Spec : in out Formal_Spec_def) is

     Atoken                  :  Obj3_Tokens.Token;
     No_Ops_Comment          : exception;

     procedure Make_Normalized_Query_File is separate;

begin
       --Text_IO.Put_Line("Formal_Spec_Object.Create_from_Query  running");
     -- First make a set of the op-names that the query module exports
     Obj3_Lex_IO.Open_Input(Spec_Filename.s);
     Text_IO.Put_Line("Opened file: " & Spec_Filename.s);
     loop  --to look for the ops-comment which lists the export operations
         Atoken := Obj3_Lex.yylex;
         exit when (Atoken = End_of_Input) or (Atoken = Ops_Comment_Token);
     end loop;
     if not (Atoken = Ops_Comment_Token) then
         raise No_Ops_Comment;
     end if;
     --Text_IO.Put_Line("Creating empty Set-of-Ops");
     Types_and_Constants.Op_Set_Pkg.Empty(Formal_Spec.Set_of_Ops);
     loop
         Atoken := Obj3_Lex.yylex;
         if Obj3_Lex_Dfa.yytext /= ")" then
               --Text_IO.Put_Line("Adding: " & Obj3_Lex_Dfa.yytext);
             Types_and_Constants.Op_Set_Pkg.
             Add(A_Strings.to_a(Obj3_Lex_Dfa.yytext),   Formal_Spec.Set_of_Ops);
         end if;
         exit when Obj3_Lex_Dfa.yytext = ")";
```

```
    end loop;
    Obj3_Lex_IO.close_input;
    Text_IO.Put(Spec_Filename.s & " exports");
    Text_IO.Put(Integer'Image
        (Types_and_Constants.Op_Set_Pkg.size(Formal_Spec.Set_of_Ops)));
    Text_IO.Put_Line(" operations.");

    -- Now make the normalized query file
    Make_Normalized_Query_File;

    --Text_IO.Put_Line("Formal_Spec_Object.Create_from_Query finished");

    exception
        when No_Ops_Comment =>
        Text_IO.Put("File to be normalized must contain an OBJ3 ");
        Text_IO.Put_Line("comment of the form:");
        Text_IO.Put_Line("    ***(operations op1 op2 op3");
        Text_IO.Put("Where op1, op2, etc are the names of the ");
        Text_IO.Put_Line("operations this module will export.");
        raise Constraint_Error;
end Create_from_Query;

------------------------------------------------------------------------------------------------------------
-- Make_Normalized_Query_File invokes an OBJ3 process to expand the query spec
-- and store pertinent information in the normalized file
-- Calls Clean_Normalized_File and Make_Prolog_for_Query
------------------------------------------------------------------------------------------------------------
with Clean_Normalized_File;

separate (Formal_Spec_Object.create_from_query)
procedure Make_Normalized_Query_File is

    Temp_Script_Name,
    Temp_Shell_Name : A_Strings.A_String;
    Obj_Temp_File, Obj_Shell_File : Text_IO.File_Type;
    Command_Line, New_Name : A_Strings.A_String;
    Temp : Integer;

    procedure Make_Prolog_for_Query
        (File_Name    : in A_Strings.A_String;
        Formal_Spec   : in out Formal_Spec_Def) is separate;

begin
    Temp_Script_Name := A_Strings.to_a(Spec_Filename.s & ".script.obj");
    Text_IO.Create(Obj_Temp_File, Out_File, Temp_Script_Name.s);
    Command_Line := A_Strings.to_a("chmod 777 " & Temp_Script_Name.s);
    Temp := Unix_Prcs.Spawn(Command_Line);
    Text_IO.Put_Line(Obj_Temp_File, "in newlisp.obj");
    Text_IO.Put(Obj_Temp_File, "in ");
    Text_IO.Put_Line(Obj_Temp_File, Spec_Filename.s);
    Text_IO.Put_Line(Obj_Temp_File, "ev (print-ps)");
```

```
    Text_IO.Put_Line(Obj_Temp_File, "ev (print-ops)");
    Text_IO.Put_Line(Obj_Temp_File, "ev (print-sorts)");
    --Text_IO.Put_Line(Obj_Temp_File, "ev (print-axioms)");
    Text_IO.Put_Line(Obj_Temp_File, "ev (print-generics)");
    Text_IO.Put_Line(Obj_Temp_File, "q");
    Temp_Shell_Name := A_Strings.to_a(Spec_Filename.s & ".shell");
    Text_IO.Create(Obj_Shell_File, Out_File, Temp_Shell_Name.s);
    Text_IO.Put_Line(Obj_Shell_File, "obj <$1 >$2");
    Command_Line := A_Strings.to_a("chmod 777 " & Temp_Shell_Name.s);
    Temp := Unix_Prcs.Spawn(Command_Line);
    -- Append .norm to the spec filename
    New_name := A_Strings.to_a(Spec_Filename.s & ".norm");
    Command_Line := A_Strings.to_a(Temp_Shell_Name.s & " " &
                Temp_Script_Name.s & " " & New_name.s);
    Text_IO.New_Line;
    Text_IO.Put_Line("Running OBJ3 task to expand query module");
    Temp := Unix_Prcs.Spawn(Command_Line);
    Text_IO.Put_Line("Finished OBJ3 task");
    Text_IO.Delete(Obj_Temp_File);
    Text_IO.Delete(Obj_Shell_File);
    Clean_Normalized_File(New_Name);
    Make_Prolog_for_Query(New_Name, Formal_Spec);
end Make_Normalized_Query_File;

-----------------------------------------------------------------------------------------------
-- Clean_Normalized_File
-----------------------------------------------------------------------------------------------
with Text_IO; use Text_IO;
with A_Strings;
with Unix_Prcs;

--This procedure writes selected information from a given file to a
--new temporary file, deletes the given file, and then renames the
--temporary file as the given file.

procedure Clean_Normalized_File(File_name : in A_Strings.A_String) is

    Temp_File, Norm_File    : Text_IO.File_Type;
    Line                    : String(1..1000);
    Cmd_Line                : A_Strings.A_String;
    Temp                    : Integer;
    Line_length             : Natural;
begin
    Text_IO.Open(Norm_File, In_File, File_Name.s);
    Text_IO.Create(Temp_File, Out_File, File_name.s & ".temp");
    while not End_of_File(Norm_File)
    loop
        while not End_of_File(Norm_File)
        loop
            Text_IO.Get_Line(Norm_File, Line, Line_Length);
            if Line_Length > 3 then
```

```
            if Line(1..3) = "!!!" then
               Put_Line(Temp_File, Line(1..Line_Length));
            end if;
            exit when Line(1..3) = "!!!";
         end if;
      end loop;
      while not End_of_File(Norm_File)
      loop
         Text_IO.Get_Line(Norm_File, Line, Line_Length);
         Put_Line(Temp_File, Line(1..Line_Length));
         if Line_Length > 3 then
            exit when Line(1..3) = "!!!";
         end if;
      end loop;
   end loop;
   Text_IO.Close(Temp_File);
   Cmd_Line := A_Strings.to_a("mv " & File_Name.s & ".temp " & File_Name.s);
   Text_IO.Delete(Norm_File);
   Temp := Unix_Prcs.Spawn(Cmd_Line);
end Clean_Normalized_File;

-----------------------------------------------------------------------------------------------
-- Make_Prolog_for_Query parses the normalized query file and transform export
-- operator definitions into a Prolog query
-- Calls Store_Hidden_Op, Make_Operator_Predicate, Make_Argument_Predicates
-----------------------------------------------------------------------------------------------
separate (Formal_Spec_Object.create_from_query.make_normalized_query_file)
procedure Make_Prolog_for_Query
   (File_Name   : in A_Strings.A_String;
   Formal_Spec : in out Formal_Spec_Def) is

   Temp_File,
   Cat_Shell            : Text_IO.File_Type;
   Sort_Name,
   Op_Name,
   Final_Unique_Predicate,
   Range_Sort,
   Command_Line,
   Store_Predicate      : A_Strings.A_String;
   Tok                  : Obj3_Tokens.Token;
   Bad_Op_Name          : exception;
   Num_Args             : Natural;
   Temp                 : Integer;
   Domain_List          : Types_and_Constants.A_String_Seq_Pkg.Sequence;
   Comma_Flag           : Boolean := False;
   Op_Definition        : Op_Defns_Pkg.Op_Defn_Type;

   procedure Make_Operator_Predicate
      (Range_Sort        : in A_Strings.A_String;
      Length             : in Natural;
      Op_name            : in A_Strings.A_String;
```

```
        Temp_File           : in  Text_IO.File_Type;
        Store_Predicate     : in out A_Strings.A_String) is separate;

    procedure  Make_Argument_Predicates
        (Op_Name            : in  A_Strings.A_String;
        Domain_List         : in  Types_and_Constants.A_String_Seq_Pkg.Sequence;
        Temp_File           : in  Text_IO.File_Type;
        Store_Predicate     : in out A_Strings.A_String) is separate;

    procedure Store_Hidden_Op is separate;

begin
    Text_IO.Create(Temp_File, Out_File, File_Name.s & ".temp");
    Obj3_Lex_IO.Open_Input(File_Name.s);
    Text_IO.Put_Line(Temp_File,  "!!!prolog");
    Text_IO.Put_Line(Temp_File, "query(OutStream) :- ");
    Final_Unique_Predicate  := A_Strings.To_a("unique([");
    Store_Predicate := A_Strings.To_a("store(OutStream, [");
    Formal_Spec.Op_Defns  := Op_Defns_Pkg.Op_Defn_Seq_Pkg.Empty;
    Formal_Spec.Hidden_Ops  :=  Op_Defns_Pkg.Op_Defn_Seq_Pkg.Empty;
    loop
        Tok  := Obj3_Lex.yylex;
        exit when Tok = Ops_Start_Token;
    end loop;
    Tok := Obj3_Lex.yylex;    -- an op token
    loop
        Num_Args := 0;         -- number of domain arguments
        Domain_List  :=  Types_and_Constants.A_String_Seq_Pkg.Empty;
        Tok  := Obj3_Lex.yylex; -- an op-name token
        Op_Name  :=  A_Strings.to_a(Obj3_Lex_Dfa.yytext);
        if Types_and_Constants.Op_Set_Pkg.Member(Op_Name,
                                                  Formal_Spec.Set_of_Ops) then
            Tok := Obj3_Lex.yylex;    -- a colon token
            loop
                Tok := Obj3_Lex.yylex; -- Sort or Arrow token
                exit when Tok = Arrow_Token;
                Num_Args  := Num_args + 1;
                Sort_Name  := A_Strings.to_a(Obj3_Lex_Dfa.yytext);
                Types_and_Constants.A_String_Seq_Pkg.
                    Add(Sort_Name,  Domain_List);
            end loop;
             Tok := Obj3_Lex.yylex;    -- range sort token
            Range_Sort  :=  A_Strings.to_a(Obj3_Lex_Dfa.yytext);
            if Comma_Flag then
                Final_Unique_Predicate  :=
                    A_Strings."&"(Final_Unique_Predicate,  ",  ");
            end if;
            Comma_Flag  := True;
            Final_Unique_Predicate  :=  A_Strings."&"(Final_Unique_Predicate,
                A_Strings.Lower_to_Upper(Op_Name));
            Store_Predicate  :=  A_Strings."&"(Store_Predicate,
```

```
            A_Strings."&"(A_Strings.Lower_to_Upper(Op_Name),","));
        Make_Operator_Predicate(Range_Sort,
            Types_and_Constants.A_String_Seq_Pkg.Length(Domain_List),
            Op_Name, Temp_File, Store_Predicate);
        Op_Definition.Op_Name := Op_Name;
        Op_Definition.Num_Parameters := Num_Args;
        Op_Definition.Range_Sort := Range_Sort;
        Op_Definition.Domain_Sorts := Op_Defns_Pkg.Pair_Sequence_Pkg.Empty;
        if Num_args > 0 then
            Make_Argument_Predicates(Op_Name, Domain_List,
                Temp_File, Store_Predicate);
        end if;
         Op_Defns_Pkg.Op_Defn_Seq_Pkg.
            Add(Op_Definition, Formal_Spec.Op_Defns);
        Tok := Obj3_Lex.yylex;    -- end expression token
        Tok := Obj3_Lex.yylex;    -- next op or ops_end token
    else  -- the op-name was not a member of the export set
        Store_Hidden_Op;
    end if;
    exit when Tok = Ops_End_Token;
end loop;
-- Close off the query here
Final_Unique_Predicate := A_Strings."&"(Final_Unique_Predicate, "]),");
Text_IO.Put_Line(Temp_File, Final_Unique_predicate.s);
Text_IO.Put_Line(Temp_File, Store_Predicate.s & "end]), fail.");
Text_IO.Put_Line(Temp_File, "query(OutStream) :- generic(G), " &
            "store(OutStream, [generic, G]).");
Text_IO.Put_Line(Temp_File, "!!!end-prolog");
Obj3_Lex_IO.Close_Input;
Text_IO.Create(Cat_Shell, Out_File, File_Name.s & ".shell");
Text_IO.Put_Line(Cat_Shell, "cat $1 >> $2");
Command_Line := A_Strings.to_a("chmod 777 " & File_Name.s & ".shell");
Temp := Unix_Prcs.Spawn(Command_Line);
Command_Line := A_Strings."&"(A_Strings.to_a
    (File_Name.s & ".shell " & File_Name.s & ".temp "), File_name);
Temp := Unix_Prcs.Spawn(Command_Line);
Text_IO.Delete(Cat_Shell);
Text_IO.Delete(Temp_File);

exception
    when Bad_Op_Name =>
        Text_IO.Put_Line("Processing aborted: Op names must be [a-z][a-z0-9]*");
        raise Constraint_Error;

end Make_Prolog_for_Query;
```

```
--------------------------------------------------------------------------------------------------------
-- Store_Hidden_Op adds operator definitions of hidden operations to the sequence
-- of query op definitions.  These are used during parsing of rewrite results
--------------------------------------------------------------------------------------------------------
separate (Formal_Spec_Object.create_from_query.make_normalized_query_file.
make_prolog_for_query)
procedure Store_Hidden_Op is

     Pair : Op_Defns_Pkg.Sort_Position_Pair;

begin
     Tok := Obj3_Lex.yylex;    -- a colon token
     loop
         Tok := Obj3_Lex.yylex; -- Sort or Arrow token
         exit when Tok = Arrow_Token;
         Num_Args := Num_args + 1;
         Sort_Name := A_Strings.to_a(Obj3_Lex_Dfa.yytext);
         Types_and_Constants.A_String_Seq_Pkg.Add(Sort_Name,  Domain_List);
     end loop;
     Tok := Obj3_Lex.yylex;    -- range sort token
     Range_Sort := A_Strings.to_a(Obj3_Lex_Dfa.yytext);
     Op_Definition.Op_Name := Op_Name;
     Op_Definition.Num_Parameters := Num_Args;
     Op_Definition.Range_Sort := Range_Sort;
     Op_Definition.Domain_Sorts := Op_Defns_Pkg.Pair_Sequence_Pkg.Empty;
     for x in 1..Types_and_Constants.A_String_Seq_Pkg.Length(Domain_List) loop
         Pair.Sort_Name := A_Strings.to_a(
             Types_and_Constants.A_String_Seq_Pkg.Fetch(Domain_List,  x).s);
         Pair.Position := x;
         Op_Defns_Pkg.Pair_Sequence_Pkg.Add(Pair,  Op_Definition.Domain_Sorts);
     end loop;
     Op_Defns_Pkg.Op_Defn_Seq_Pkg.Add(Op_Definition,  Formal_Spec.Hidden_Ops);
     Tok := Obj3_Lex.yylex;    -- end expression token
     Tok := Obj3_Lex.yylex;    -- next op or ops_end token
end Store_Hidden_Op;

--------------------------------------------------------------------------------------------------------
-- Make_Operator_Predicate creates an operator predicate as part of the query Prolog
--------------------------------------------------------------------------------------------------------
separate (Formal_Spec_Object.create_from_query.
         make_normalized_query_file.make_prolog_for_query)
procedure Make_Operator_Predicate
     (Range_Sort : in A_Strings.A_String;
     Length : in Natural;
     Op_name : in A_Strings.A_String;
     Temp_File : in Text_IO.File_Type;
     Store_Predicate : in out A_Strings.A_String) is
begin
     Text_IO.Put(Temp_File, "operator(");
     Store_Predicate := A_Strings."&"(Store_Predicate, Natural'Image(Length) & ", ");
     if Types_and_Constants.Predefined_Obj_Sorts_Pkg.Member(Range_Sort,
```

```
            Types_and_Constants.Predef_Obj_Sorts_Set) then
        Text_IO.Put(Temp_File, A_Strings.Upper_to_Lower(Range_Sort).s);
        Store_Predicate := A_Strings."&"(Store_Predicate,
            A_Strings.Upper_to_Lower(Range_Sort).s & ", ");
    else
        Text_IO.Put(Temp_File, A_Strings.Lower_to_Upper(Range_Sort).s);
        Store_Predicate := A_Strings."&"(Store_Predicate,
        A_Strings.Lower_to_Upper(Range_Sort).s & ", ");
    end if;
    Text_IO.Put(Temp_File, "," & Natural'Image(Length) & ", ");
    Text_IO.Put(Temp_File, A_Strings.Lower_to_Upper(Op_Name).s);
    Text_IO.Put_Line(Temp_File, "),");
end Make_Operator_Predicate;

----------------------------------------------------------------------------------------------
-- Make_Argument_Predicates creates argument predicates as part of the query Prolog
----------------------------------------------------------------------------------------------
separate (Formal_Spec_Object.create_from_query.
        make_normalized_query_file.make_prolog_for_query)
procedure Make_Argument_Predicates
    (Op_Name          : in A_Strings.A_String;
    Domain_List       : in Types_and_Constants.A_String_Seq_Pkg.Sequence;
    Temp_File         : in Text_IO.File_Type;
    Store_Predicate   : in out A_Strings.A_String) is

    Position          : Natural;
    Append_Part, Unique_Part ,
    Domain_Sort,
    Position_Str      : A_Strings.A_String;
    Sort_Posn_Pair    : Op_Defns_Pkg.Sort_Position_Pair;

begin
    Position := 1;
    Unique_Part := A_Strings.to_a("unique([");
    loop
        Text_IO.Put(Temp_file, "argument(");
        Text_IO.Put(Temp_File, A_Strings.Lower_to_upper(Op_Name).s & ", ");
        Domain_Sort := Types_and_Constants.A_String_Seq_Pkg.
            Fetch(Domain_List, Position);
        if Types_and_Constants.Predefined_Obj_Sorts_Pkg.Member(Domain_Sort,
                Types_and_Constants.Predef_Obj_Sorts_Set) then
            Text_IO.Put(Temp_File,
                A_Strings.Upper_to_Lower(Domain_Sort).s & ", ");
            Store_Predicate := A_Strings."&"(Store_Predicate,
                A_Strings."&"(A_Strings.Upper_to_Lower(Domain_Sort),", "));
        else
            Text_IO.Put(Temp_File,
                A_Strings.Lower_to_Upper(Domain_Sort).s & ", ");
            Store_Predicate := A_Strings."&"(Store_Predicate,
                A_Strings."&"(A_Strings.Lower_to_Upper(Domain_Sort),", "));
        end if;
```

```
        Sort_Posn_Pair.Sort_Name := A_Strings.Upper_to_Lower(Domain_Sort);
        Sort_Posn_Pair.Position := Position;
        Op_Defns_Pkg.Pair_Sequence_Pkg.Add(Sort_Posn_Pair,
            Op_Definition.Domain_Sorts);
        Position_Str := A_Strings.to_a(Natural'Image(Position));
        Position_Str := A_Strings.Reverse_Order(A_Strings.Trim(
            A_Strings.Reverse_Order(Position_Str)));
        Append_Part := A_Strings."&"(A_Strings.Lower_to_Upper(Op_name),
            Position_Str);
        Text_IO.Put_Line(Temp_File, Append_Part.s & "),");
        Unique_Part := A_Strings."&"(Unique_Part, Append_Part);
        Store_Predicate := A_Strings."&"(Store_Predicate,
            A_Strings."&"(A_Strings.Lower_to_Upper(Append_Part),", "));
        if Position >= Types_and_Constants.A_String_Seq_Pkg.
                Length(Domain_List) then
            Unique_Part := A_Strings."&"(Unique_Part, "]),");
        else
            Unique_Part := A_Strings."&"(Unique_Part, ", ");
        end if;
        exit when Position >= Types_and_Constants.A_String_Seq_Pkg.
            Length(Domain_List);
        Position := Position + 1;
    end loop;
    Text_IO.Put_Line(Temp_File, Unique_Part.s);
end Make_Argument_Predicates;
```

2. Build Test Set

```
---------------------------------------------------------------------------------------------------
-- Build_Test_Set creates the test set from the query signature
-- Calls Get_Predefined_Terms, Make_User_Defined_Terms, Print_Term
---------------------------------------------------------------------------------------------------
with Text_IO, Formal_Spec_Object, A_Strings, Term_Definition_Pkg;
with Get_Set_of_Sorts, Types_and_Constants, Get_Predefined_Terms;
with Print_Term, Make_User_Defined_Terms;

procedure Build_Test_Set
    (Query_Filename     : in A_Strings.A_String;
    Formal_Spec         : in out Formal_Spec_Object.Formal_Spec_Def;
    Test_Set            : in out Term_Definition_Pkg.Test_Set_Def) is

    Sort_Set : Types_and_Constants.Op_Set_Pkg.Set;
    Norm_Filename : A_Strings.A_String;
    Num_Sorts, Buffer : Natural;
    Spaces : String(1..15) := "               ";

begin
    Text_IO.New_Line;
    Text_IO.Put_Line("Building a Test-Set.");
    Norm_Filename := A_Strings.to_a(Query_Filename.s & ".norm");
```

```
    Sort_Set := Get_Set_of_Sorts(Norm_Filename);
    Num_Sorts := Types_and_Constants.Op_Set_Pkg.Size(Sort_Set);
     Test_Set := new Term_Definition_Pkg.Test_Set_Rec(Size => Num_Sorts);
    --Text_IO.Put_Line("Made a test-set with" & Natural'Image(
    --    Types_and_Constants.Op_Set_Pkg.Size(Sort_Set)) & " sorts.");
    Get_Predefined_Terms(Test_Set, Sort_Set);
    Make_User_Defined_Terms(Test_Set, Sort_Set, Formal_Spec);
    Text_IO.Put_Line("The terms in the test set are:");
    for i in 1..Term_Definition_Pkg.
              Term_Sequence_Pkg.Length(Test_Set.Term_List)
    loop
        Buffer := Spaces'Length - Term_Definition_Pkg.Term_Sequence_Pkg.
                  Fetch(Test_Set.Term_List, i).Range_Sort.s'Length;
        Text_IO.Put(Spaces(1..Buffer));
        Text_IO.Put(Term_Definition_Pkg.Term_Sequence_Pkg.
            Fetch(Test_Set.Term_List, i).Range_Sort.s & ": ");
        Print_Term(Text_IO.Standard_Output,
            Term_Definition_Pkg.Term_Sequence_Pkg.Fetch(Test_Set.Term_List, i));
        Text_IO.New_Line;
    end loop;
end Build_Test_Set;

-------------------------------------------------------------------------------------------
-- Get_Set_of_Sorts creates a set composed of the names of the sorts used in the query
-- specification.  Uses auxiliary procedures for diagnostics.
-------------------------------------------------------------------------------------------
with A_Strings, Set_Pkg, Types_And_Constants;
with Obj3_Lex_IO, Obj3_Lex, Obj3_Lex_Dfa;
with Text_IO;       use Text_IO;
with Obj3_Tokens;      use Obj3_Tokens;

function Get_Set_of_Sorts(File_Name  : in A_Strings.A_String)
    return Types_and_Constants.Op_Set_Pkg.Set is

    Tok     : Obj3_Tokens.token;
    A_Set   : Types_and_Constants.Op_Set_Pkg.Set;

    procedure Print_Name(Name : in A_Strings.A_String) is
    begin
        Text_IO.Put(Name.s & " ");
    end Print_Name;

    procedure Print_Set is new Types_and_Constants.Op_Set_Pkg.Scan(Print_Name);

begin
    Obj3_Lex_IO.Open_Input(File_Name.s);
    --Text_IO.Put_Line("Opened file: " & File_Name.s);
    Types_and_Constants.Op_Set_Pkg.Empty(A_Set);
    loop
        Tok := Obj3_Lex.yylex;
        exit when (Tok = Sorts_Start_Token) or (Tok = End_of_Input);
```

```
    end loop;
    loop
        Tok := Obj3_Lex.yylex; -- sort_token or sorts_end_token
        exit when (Tok = Sorts_End_Token) or (Tok = End_of_Input);
        Tok := Obj3_Lex.yylex; -- sort_id_token
        while (Tok /= Endexpr_Token)
        loop
            if Tok /= '<' then
                Types_and_Constants.Op_Set_Pkg.
                    Add(A_Strings.to_a(Obj3_Lex_Dfa.yytext), A_Set);
            end if;
            Tok := Obj3_Lex.yylex;
        end loop;
    end loop;
    Obj3_Lex_IO.Close_Input;
    --Text_IO.Put("Sorts used in this module are: ");
    --Print_Set(A_Set);
     --Text_IO.New_Line;
    return A_Set;
end Get_Set_of_Sorts;

------------------------------------------------------------------------------------------
-- Get_Predefined_Terms reads predefined terms from a file and adds them to the
-- test set.  Uses a recursive procedure Make_Term.
------------------------------------------------------------------------------------------
with Text_IO,  A_Strings, Types_and_Constants;
with Term_Definition_Pkg;   use  Term_Definition_Pkg;
with Predef_Lex_IO;
with Predef_Lex_Dfa;        use Predef_Lex_Dfa;
with Predef_Lex;            use Predef_Lex;

procedure Get_Predefined_Terms
    (Test_Set   : in out Term_Definition_Pkg.Test_Set_Def;
    Sort_Set    : in Types_and_Constants.Op_Set_Pkg.Set) is

    Num_Terms,
    Term_Group_Start_Position,
    Sort_Index_Count        : Natural;
    Tok                     : Predef_Lex.Token;
    Predef_Sort             : A_Strings.A_String;
    A_Term                  : Term_Definition_Pkg.Term_Access;

    procedure Make_Term(A_Term : in out Term_Definition_Pkg.Term_Access) is

    Another_Term      : Term_Definition_Pkg.Term_Access;
    Args              : Natural;

    begin
       Tok := yylex; -- Name:
       Tok := yylex; -- Op_Name
       A_Term.Op_Name := A_Strings.to_a(yytext);
```

```
        --Text_IO.Put_Line("Term or subterm name is: " & yytext);
        A_Term.Range_Sort := Predef_Sort;
        Tok := yylex; -- numargs:
        Tok := yylex; -- Number of arguments
        Args := Natural'Value(yytext);
        A_Term.Num_Args := Args;
        for k in 1..Args
        loop
            Another_Term := new Term_Definition_Pkg.Term;
            Make_Term(Another_Term);
            A_Term.Arguments(K) := Another_Term;
        end loop;
    end Make_Term;

begin
    --Text_IO.Put_Line("Starting Get-Predefined-Terms.");
    Term_Group_Start_Position := 1;
    Sort_Index_Count := 0;
    predef_lex_io.Open_Input("predef-terms");
    loop
        loop
            Tok := yylex;
            exit when (Tok = Predef) or (Tok = End_of_Input);
        end loop;
        exit when Tok = End_of_Input;
        Tok := yylex;  -- a predefined sort
        Predef_Sort := A_Strings.to_a(yytext);
        if Types_and_Constants.Op_Set_Pkg.Member(Predef_Sort, Sort_Set) then
            --Text_IO.Put_Line("Adding terms for: " & Predef_Sort.s);
            Sort_Index_Count := Sort_Index_Count + 1;
            --Text_IO.Put_Line("Sort index is:" & Natural'Image(Sort_Index_Count));
            Test_Set.Sort_Index(Sort_Index_Count).Sort_Name :=
                 A_Strings.to_a(Predef_Sort.s);
             --Text_IO.Put_Line("Added " & Predef_Sort.s & " to Index.");
            Test_Set.Sort_Index(Sort_Index_Count).Start :=
                 Term_Group_Start_Position;
            --Text_IO.Put_Line("Set start position to:" & Natural'Image(
            --    Term_Group_Start_Position));
            Tok := yylex;  -- constants
            loop
                Tok := yylex; -- a constant or numterms:
                exit when Tok = Numterms;
                Const_Seq_Pkg.Add(A_Strings.to_a(yytext),
                    Test_Set.Sort_Index(Sort_Index_Count).Constants);
            end loop;
            Tok := yylex;  -- the number of terms to follow
            Num_Terms := Natural'Value(yytext);
             --Text_IO.Put_Line(Predef_sort.s & " will add" &
            --    Natural'Image(Num_Terms) & " term(s).");
            Test_Set.Sort_Index(Sort_Index_Count).Stop :=
            Term_Group_Start_Position + Num_Terms - 1;
```

```
            for j in 1..Num_Terms
            loop
                A_Term := new Term_Definition_Pkg.Term;
                Make_Term(A_Term);
                Term_Definition_Pkg.Term_Sequence_Pkg.Add(A_Term,
                            Test_Set.Term_List);
            end loop;
            Term_Group_Start_Position := Term_Group_Start_Position + Num_Terms;
        end if;
    end loop;
    predef_lex_io.Close_Input;
    --Text_IO.Put_Line("Finished  Get_Predefined_Terms");
end  Get_Predefined_Terms;

----------------------------------------------------------------------------------------------------
-- Make_User_Defined_Terms makes test set terms from the op definitions sequence
-- Contains many diagnostics
----------------------------------------------------------------------------------------------------
with Text_IO, A_Strings;
with Term_Definition_Pkg;    use  Term_Definition_Pkg;
with Types_and_Constants;    use  Types_and_Constants;
with Formal_Spec_Object;     use  Formal_Spec_Object;
with Op_Defns_Pkg;           use  Op_Defns_Pkg;

procedure  Make_User_Defined_Terms
    (Test_Set        : in out Test_Set_Def;
    Sort_Set         : in out Op_Set_Pkg.Set;
    Formal_Spec      : in out Formal_Spec_Def) is

    Rem_Sort_Seq                : A_String_Seq_Pkg.Sequence;
    Sort_of_Interest            : A_Strings.A_String;
    Op_Definition               : Op_Defn_Type;
    Test_Set_Start_Position     : Natural;
    Num_Terms_Added,
    Sort_Index_Location,
    Sig_Location                : Natural;
    A_Term                      : Term_Access;

    procedure Make_A_Term is separate;

    procedure Generator(X : in A_Strings.A_String) is
    begin
        if not Predefined_Obj_Sorts_Pkg.Member(X, Predef_Obj_Sorts_Set) then
            A_String_Seq_Pkg.Add(X,  Rem_Sort_Seq);
         end if;
    end Generator;

    procedure Scan_Set is new Op_Set_Pkg.Scan(Generator);

begin -- Make_User_Defined_Terms
    --Text_IO.Put_Line("Making user defined terms.");
```

```
Test_Set_Start_Position := Term_Sequence_Pkg.Length(Test_Set.Term_List) + 1;
Rem_Sort_Seq := A_String_Seq_Pkg.Empty;
Scan_Set(Sort_Set);
for i in 1..A_String_Seq_Pkg.Length(Rem_Sort_Seq) loop
    Sort_Index_Location := 0;
    Sort_of_Interest := A_Strings.to_a(A_String_Seq_Pkg.Fetch(
          Rem_Sort_Seq, i).s);
--Text_IO.Put_Line("Making terms for: " & Sort_of_Interest.s);
Num_Terms_Added := 0;
for j in 1..Op_Defn_Seq_Pkg.Length(Formal_Spec.Op_Defns) loop
        if Op_Defn_Seq_Pkg.Fetch(Formal_Spec.Op_Defns, j).Range_Sort.s =
                Sort_of_Interest.s then
            Num_Terms_Added := Num_Terms_Added + 1;
            Op_Definition := Op_Defn_Seq_Pkg.Fetch(Formal_Spec.Op_Defns, j);
            Sig_Location := j;
            --Text_IO.Put_Line("Op " & Op_Definition.Op_Name.s & " has range "
            --     & Sort_of_Interest.s & ".");

            -- update the sort_index for this sort if it has not already
            -- been done.
            if Sort_Index_Location = 0 then
                --Text_IO.Put_Line("The sort Index has" & Natural'Image(
                --   Test_Set.Sort_Index'Length) & " cells.");
                --Text_IO.Put("The contents are:");
                for x in Test_Set.Sort_Index'Range
                loop
                     --Text_IO.Put(" " & Test_Set.Sort_Index(x).Sort_Name.s);
                     if Test_Set.Sort_Index(x).Sort_Name.s = "!" then
                        Test_Set.Sort_Index(x).Start := Test_Set_Start_Position;
                        Test_Set.Sort_Index(x).Sort_Name :=
                            A_Strings.to_A(Sort_of_Interest.s);
                        Sort_Index_Location := x;
                        --Text_IO.New_Line;
                         --Text_IO.Put_Line("The index location for " &
                        --   Test_Set.Sort_Index(x).Sort_Name.s & " is" &
                        --    Natural'Image(x) & ".");
                        exit;
                        else if Test_Set.Sort_Index(x).Sort_Name.s =
                                  Sort_of_Interest.s then
                            Test_Set.Sort_Index(x).Start :=
                                Test_Set_Start_Position;
                            Sort_Index_Location := x;
                            --Text_IO.New_Line;
                            --Text_IO.Put_Line("The index location for " &
                            --             Sort_of_Interest.s & " is" &
                            --              Natural'Image(x) & ".");
                            exit;
                        end if;
                    end if;
                end loop;
                --Text_IO.New_Line;
```

```
                end if;

                -- Now make the term for the test set
                Make_A_Term;
                Test_Set_Start_Position := Test_Set_Start_Position + 1;
            end if;
        end loop;
        Test_Set.Sort_Index(Sort_Index_Location).Stop :=
        Test_Set.Sort_Index(Sort_Index_Location).Start + Num_Terms_Added - 1;
    end loop;
end  Make_User_Defined_Terms;

--------------------------------------------------------------------------------------------
-- Make_a_Term creates a term from an op definition to be added to the test set
--------------------------------------------------------------------------------------------
with A_Strings; use A_Strings;

separate (make_user_defined_terms)
procedure Make_A_Term is

    Dom_Sort, New_Constant      : A_String;
    Const_Count                 : Natural := 1;
    Another_Term                :  Term_Definition_Pkg.Term_Access;

begin
    --Text_IO.Put_Line("Making a term for: " & Op_Definition.Op_Name.s & ".");
    A_Term := new Term_Definition_Pkg.Term;
    A_Term.Op_Name := Op_Definition.Op_Name;
    A_Term.Range_Sort := Op_Definition.Range_Sort;
    A_Term.Num_Args := Op_Definition.Num_Parameters;
    A_Term.Signature := Sig_Location; -- the location in the map of
                    -- this term's signature
    --Text_IO.Put_Line("Checking its parameters.");
    for y in 1..Op_Definition.Num_Parameters loop
        Another_Term := new Term_Definition_Pkg.Term;
        Dom_Sort := Op_Defns_Pkg.Pair_Sequence_Pkg.Fetch(
            Op_Definition.Domain_Sorts, y).Sort_Name;
        --Text_IO.Put_Line("Argument" & Natural'Image(y) & " is " & Dom_Sort.s);
        Another_Term.Range_Sort := A_Strings.to_a(Dom_Sort.s);

        -- if the domain sort of this term is the same as the range sort, then
        -- we will make that argument a constant of that sort
        if Upper_to_Lower(Dom_Sort).s = Upper_to_Lower(A_Term.Range_Sort).s then
            New_Constant := A_Strings.to_a(
            A_Strings.Upper_to_Lower(A_Term.Range_Sort).s & "const" &
            Reverse_Order(Trim(Reverse_Order(to_a(
                Natural'Image(Const_Count))))).s);
            Another_Term.Op_Name := A_Strings.to_a(New_Constant.s);

            -- if this new constant is not already in the list of constants then
            -- we must add it to the sort index info
```

```
            if Const_Count > Term_Definition_Pkg.Const_Seq_Pkg.Length(
                    Test_Set.Sort_Index(Sort_Index_Location).Constants) then
                Term_Definition_Pkg.Const_Seq_Pkg.Add(New_Constant,
                Test_Set.Sort_Index(Sort_Index_Location).Constants);
            end if;
             Const_Count := Const_Count + 1;
        else -- not the same sort, so fill with a placeholder
            Another_Term.Op_Name  := A_Strings.to_a("!!!");
        end if;
        A_Term.Arguments(y) := Another_Term;
    end loop;
    Term_Definition_Pkg.Term_Sequence_Pkg.Add(A_Term,  Test_Set.Term_List);
end  Make_A_Term;
```

3. Make IO List

```
----------------------------------------------------------------------------------------------------
-- Make_IO_List creates the IO List from the export signature and the test set.
-- Calls Make_Template, Scan_For_Placeholder, and Expand Term
----------------------------------------------------------------------------------------------------
with Term_Definition_Pkg;   use  Term_Definition_Pkg;
with Formal_Spec_Object;    use  Formal_Spec_Object;
with Op_Defns_Pkg;          use  Op_Defns_Pkg;
with A_Strings;             use  A_Strings;
with Make_Template, Scan_for_Placeholder, Unchecked_Deallocation;
with Text_IO, Print_Term;

procedure Make_IO_List
    (Test_Set       : in out Test_Set_Def;
    Formal_Spec     : in  Formal_Spec_Def;
    IO_List         : in out IO_List_Def) is

    IO_Pair,
    Next_IO_Pair,
    Temp, Head,
    Tail, Previous      : IO_List_Def;
    Expansion           : Boolean := False;

    procedure Deallocate is new Unchecked_Deallocation(
        Object => IO_Pair_Rec,
        Name => IO_List_Def);

    procedure Expand_Term(
        Whole_Term,
        A_Term          : in out Term_Access;
        Expansion       : in out Boolean) is separate;

begin
    Text_IO.New_Line;
    Text_IO.Put_Line("Making  an  IO_List.");
```

```
IO_Pair := new IO_Pair_Rec;
Head := IO_Pair;
Tail := IO_Pair;

-- for every op-defintion make an initial template of sample terms
-- to be used for later tests. The IO-List is a linked list
--Text_IO.Put_Line("Making templates for the export ops.");
for i in 1..Op_Defn_Seq_Pkg.Length(Formal_Spec.Op_Defns)
loop
    Make_Template(IO_Pair.Input, Op_Defn_Seq_Pkg.Fetch(
        Formal_Spec.Op_Defns, i), i);
    exit when i = Op_Defn_Seq_Pkg.Length(Formal_Spec.Op_Defns);
    Next_IO_Pair := new IO_Pair_Rec;
    IO_Pair.Next := Next_IO_Pair;
    Tail := Next_IO_Pair;
    IO_Pair := Next_IO_Pair;
end loop;

-- Now scan the IO_List looking for terms containing !!! placeholders.
 -- If a term contains a placeholder, expand the term by creating copies
-- of it, filling the placeholder with a suitable subterm taken from
-- the Test_Set. Continue to scan until all placeholders have been removed
IO_Pair := Head;
Previous := Head;
loop
    --Text_IO.Put("Scanning for placeholders in ");
    --Print_Term(Text_IO.Standard_Output, IO_Pair.Input);
    --Text_IO.New_Line;
    if Scan_for_Placeholder(IO_Pair.Input) then
        --Text_IO.Put_Line("Placeholder found in: " & IO_Pair.Input.Op_Name.s);
        Expand_Term(IO_Pair.Input, IO_Pair.Input, Expansion);
        --Text_IO.Put_Line("Term expansion completed.");
        Temp := IO_Pair.Next;
        if IO_Pair = Head then -- if deleting the head of the linked list
            Head := IO_Pair.Next;
            Previous := IO_Pair.Next;
            --Text_IO.Put_Line("Changing the head of the IO_List.");
        else -- deleting a node in the middle of the linked list
            Previous.Next := IO_Pair.Next;
             --Text_IO.Put_Line("Dereferencing a middle node in the IO_List.");
        end if;
        --Text_IO.Put_Line("Pointers have been updated.");
        Deallocate(IO_Pair); -- Garbage collection
        --Text_IO.Put_Line("Deallocated the IO_Pair.");
         IO_Pair := Temp;     -- let's consider the next term
    else -- the term did not have a placeholder, skip it
        Previous := IO_Pair;
        IO_Pair := IO_Pair.Next; -- let's consider the next term
    end if;
    exit when IO_Pair = null;
end loop;
```

```
     IO_List := Head;
     Text_IO.Put_Line("The input terms in the IO_List are:");
     Temp := Head;
     loop
          Text_IO.Put("    ");
          Print_Term(Text_IO.Standard_Output, Temp.Input);
          Text_IO.New_Line;
          Temp := Temp.Next;
          exit when Temp = null;
     end loop;
end Make_IO_List;

------------------------------------------------------------------------------------
-- Make_Template makes a template for a given op definition so that it can be added
-- to the IO List
------------------------------------------------------------------------------------
with Term_Definition_Pkg;   use  Term_Definition_Pkg;
with Op_Defns_Pkg;          use  Op_Defns_Pkg;
with A_Strings, Text_IO;

procedure Make_Template
     (A_Term              : in out Term_Access;
     Op_Def               : in Op_Defn_Type;
     Signature_Loc        : in Natural) is

     Subterm : Term_Access;

begin
     --Text_IO.Put_Line("Making a template for: " & Op_Def.Op_Name.s);
     A_Term := new Term;
     A_Term.Op_Name := A_Strings.to_a(Op_Def.Op_Name.s);
     A_Term.Range_Sort := A_Strings.to_a(Op_Def.Range_Sort.s);
     A_Term.Num_Args := Op_Def.Num_Parameters;
     A_Term.Signature := Signature_Loc;
     for i in 1..Op_Def.Num_Parameters
     loop
          Subterm := new Term;
          Subterm.Op_Name := A_Strings.to_a("!!!");
          Subterm.Range_Sort := A_Strings.to_a(Pair_Sequence_Pkg.Fetch
               (Op_Def.Domain_Sorts, i).Sort_Name.s);
          Subterm.Num_Args := 0;
          A_Term.Arguments(i) := Subterm;
     end loop;
end Make_Template;

------------------------------------------------------------------------------------
-- Scan_for_Placeholder is a recursive function that checks to see if a term contains a
-- placeholder
------------------------------------------------------------------------------------
with Term_Definition_Pkg; use Term_Definition_Pkg;
```

```
function Scan_for_Placeholder(A_Term : in Term_Access) return Boolean is

begin
     if A_Term = null then
         return false;
     else
         if A_Term.Op_Name.s = "!!!" then
             return true;
         else
             for i in 1..A_Term.Num_Args loop
                 if Scan_for_Placeholder(A_Term.Arguments(i)) then
                     return true;
                 end if;
             end loop;
             return false;
         end if;
     end if;
end  Scan_for_Placeholder;

------------------------------------------------------------------------------------------------
-- Expand_Term expands a term containing a placeholder, adding the newly expanded
-- terms to the end of the IO List.
-- Makes use of  Compare_Signatures, Copy_Term, Insert_Term, Print_Term, and
--  Check_for_Circularity
------------------------------------------------------------------------------------------------
with Copy_Term, Print_Term, Insert_Term;
with Check_for_Circularity, Compare_Signatures;

separate (make_io_list)
procedure Expand_Term
     (Whole_Term, A_Term    : in out Term_Access;
     Expansion              : in out Boolean) is

     Op_Defn                 :  Op_Defns_Pkg.Op_Defn_Type;
     Expansion_Sort          :  A_Strings.A_String;
     A_Copy, Subterm         :  Term_Access;
     New_IO_Pair             :  IO_List_Def;
     Flag                    :  Boolean;
     Test_Set_Pointer        :  Term_Access;
     Term_List_Start_Posn,
     Term_List_Stop_Posn,
     Term_List_Pointer,
     Sort_Index_Position     :  Natural;

begin
     --Text_IO.Put_Line("Expanding term: " & A_Term.Op_Name.s);
     Expansion := False;
     for k in 1..A_Term.Num_Args loop  --for each argument in the term
         if A_Term.Arguments(k).Op_Name.s = "!!!" then   --a placeholder
             --Text_IO.Put("Placeholder in position:" & Natural'Image(k) & ".");
             -- Now search the op definitions to find the sort of the argument.
```

```
-- It is possible that the placeholder to be filled is a
 -- predefined generic whose sort is unknown by the term.
-- The user must! export the constructors for predefined generics!
for i in 1..Op_Defns_Pkg.Op_Defn_Seq_Pkg.Length
        (Formal_Spec.Op_Defns)
loop
    Op_Defn := Op_Defns_Pkg.Op_Defn_Seq_Pkg.Fetch
        (Formal_Spec.Op_Defns, i);
    if Compare_Signatures(A_Term, Op_Defn) then
        Expansion_Sort := A_Strings.to_a(Pair_Sequence_Pkg.Fetch
            (Op_Defn.Domain_Sorts, k).Sort_Name.s);
        --Text_IO.Put_Line("  Its sort is " & Expansion_Sort.s & ".");
        -- Check the index to find out where in the test_set are the
         -- terms we will use to expand the given term
        Expansion := True;
        for j in Test_Set.Sort_Index'Range
        loop
            Sort_Index_Position := j;
            Term_List_Start_Posn := Test_Set.Sort_Index(j).Start;
            Term_List_Stop_Posn := Test_Set.Sort_Index(j).Stop;
            exit when Lower_to_Upper(Test_Set.Sort_Index(j).Sort_Name).s
                = Lower_to_Upper(Expansion_Sort).s;
        end loop;
        --Text_IO.Put_Line("Expansion with Test_Set terms" &
        --        Natural'Image(Term_List_Start_Posn) & " through" &
        --        Natural'Image(Term_List_Stop_Posn) & ".");
        Term_List_Pointer := Term_List_Start_Posn;
        loop
            Copy_Term(Whole_Term, A_Copy);
            --Text_IO.Put_Line("Made a copy of: "
            --            & Whole_Term.Op_Name.s & ".");
            --Text_IO.Put("From:  ");
            --Print_Term(Text_IO.Standard_Output, A_Copy);
                --Text_IO.Put("  to  ");
            -- We must avoid circularities here! If the term to be added
            -- has an argument whose sort is the same as the range sort
            -- of the given term, that will create a cycle.
            Test_Set_Pointer := Term_Sequence_Pkg.Fetch
                (Test_Set.Term_List, Term_List_Pointer);
            Copy_Term(Test_Set_Pointer, Subterm);
            Check_for_Circularity(Whole_Term, Test_Set, Subterm);
            if A_Term /= Whole_Term then
                Check_for_Circularity(A_Term, Test_Set, Subterm);
            end if;
            Flag := False;
             Insert_Term(A_Copy, Subterm, Flag);
            --Print_Term(Text_IO.Standard_Output, A_Copy);
              --Text_IO.New_Line;
            New_IO_Pair := new IO_Pair_Rec;
            New_IO_Pair.Input := A_Copy;
            Tail.Next := New_IO_Pair;
```

```
                    Tail := New_IO_Pair;
                    Term_List_Pointer := Term_List_Pointer + 1;
                    exit when Term_List_Pointer > Term_List_Stop_Posn;
                end loop;
                exit;     -- there should be only *one* signature to match the export
                          -- signature.
            end if;
        end loop;  -- to check the op definitions
        exit;     -- We only allow one expansion per pass. Other placeholders
                  -- in this term will be expanded when the copies are examined.
    else     -- this op-name is not a placeholder
        -- depth first search
        Expand_Term(Whole_Term,  A_Term.Arguments(k),  Expansion);
        exit when Expansion;
    end if;
  end loop;  -- to check the arguments of a given term
end  Expand_Term;

------------------------------------------------------------------------------------------
-- Compare_Signatures checks to see if the structure of a given term matches that
-- o f a given signature
------------------------------------------------------------------------------------------
with Term_Definition_Pkg;   use  Term_Definition_Pkg;
with Op_Defns_Pkg;          use  Op_Defns_Pkg;
with A_Strings;             use  A_Strings;

function Compare_Signatures
    (A_Term  :  Term_Access;
    Op_Defn : Op_Defn_Type) return Boolean is

    Result : Boolean := True;

begin
    if (A_Term.Op_Name.s = Op_Defn.Op_Name.s) and
            (A_Term.Num_Args   = Op_Defn.Num_Parameters) and
            (Lower_to_Upper(A_Term.Range_Sort).s =
                Lower_to_Upper(Op_Defn.Range_Sort).s) then
        for x in 1..A_Term.Num_Args loop
            if Lower_to_Upper(A_Term.Arguments(x).Range_Sort).s /=
                    Lower_to_Upper(Pair_Sequence_Pkg.Fetch
                    (Op_Defn.Domain_Sorts,  x).Sort_Name).s then
                Result := False;
            end if;
        end loop;
    else
        Result := False;
    end if;
    return Result;
end Compare_Signatures;
```

```
-----------------------------------------------------------------------------------------------
-- Copy_Term creates a term identical to a given term
-----------------------------------------------------------------------------------------------
with Term_Definition_Pkg;   use  Term_Definition_Pkg;
with A_Strings;             use  A_Strings;
with Text_IO;

procedure Copy_Term
    (A_Term         : in Term_Access;
     New_Term       : in out Term_Access) is

begin
    --Text_IO.Put_Line("Entered copy term");
    --Text_IO.Put("Copy ");
    if A_Term = null then
        New_Term := null;
    else
        --Text_IO.Put_Line("The term was not null.");
        New_Term := new Term;
        New_Term.Op_Name :=   A_Strings.to_a(A_Term.Op_Name.s);
        New_Term.Range_Sort := A_Strings.to_a(A_Term.Range_Sort.s);
        New_Term.Num_Args := A_Term.Num_Args;
        New_Term.Signature := A_Term.Signature;
        --Text_IO.Put_Line("Base term copied.  Now for the subterms.");
        for i in 1..A_Term.Num_Args loop
            Copy_Term(A_Term.Arguments(i),  New_Term.Arguments(i));
        end loop;
    end if;
end Copy_Term;

-----------------------------------------------------------------------------------------------
-- Insert_Term inserts a term into the first (depth-first) placeholder position
-- of a given term
-----------------------------------------------------------------------------------------------
with Term_Definition_Pkg; use Term_Definition_Pkg;
with Text_IO;


procedure Insert_Term
    (A_Term    : in out Term_Access;
     Subterm   : in Term_Access;
     Flag      : in out Boolean) is
begin
    --Text_IO.Put("Insert ");
    for x in 1..A_Term.Num_Args loop
        if A_Term.Arguments(x).Op_Name.s = "!!!" then
            A_Term.Arguments(x) := Subterm;
            Flag := True;
            else if A_Term.Arguments(x).Num_Args > 0 then
                Insert_Term(A_Term.Arguments(x), Subterm, Flag);
            end if;
```

```
        end if;
        exit when Flag;
    end loop;
end  Insert_Term;

--------------------------------------------------------------------------------
-- Check_for_Circularity checks each argument of a given subterm to determine if the
--  argument is a placeholder and has the same sort as the whole term's range sort
-- If so, it replaces the placeholder with a constant
--------------------------------------------------------------------------------
with Term_Definition_Pkg;    use  Term_Definition_Pkg;
with A_Strings;              use  A_Strings;
with Text_IO;

procedure  Check_for_Circularity(
    Whole_Term        : in  Term_Access;
    Test_Set          : in out Test_Set_Def;
    Subterm           : in out Term_Access) is

    New_Constant          :  Term_Access;
    New_Name              :  A_Strings.A_String;
    Sort_Index_Position   : Natural;

begin
    -- search the sort_index to find the position of the expansion sort
    for y in Test_Set.Sort_Index'Range
    loop
        Sort_Index_Position := y;
        exit when Lower_to_Upper(Test_Set.Sort_Index(y).Sort_Name).s =
            Lower_to_Upper(Whole_Term.Range_Sort).s;
    end loop;
    for x in 1..Subterm.Num_Args loop
        if (Lower_to_Upper(Subterm.Arguments(x).Range_Sort).s =
                Lower_to_Upper(Whole_Term.Range_Sort).s)  and
                (Subterm.Arguments(x).Op_Name.s  =  "!!!")  then
            --Text_IO.Put_Line("Circularity  Detected.");
            New_Constant := new Term;
            New_Constant.Range_Sort  := A_Strings.to_a(Whole_Term.Range_Sort.s);
            New_Constant.Num_Args := 0;
            if Const_Seq_Pkg.Length(Test_Set.Sort_Index(Sort_Index_Position).
                    Constants) > 0 then
                New_Constant.Op_Name  := A_Strings.to_a(Const_Seq_Pkg.Fetch(
                Test_Set.Sort_Index(Sort_Index_Position).Constants,  1).s);
            else
                New_Name  := A_Strings.to_a(Upper_to_Lower
                                (Whole_Term.Range_Sort).s  &  "const1");
                Const_Seq_Pkg.Add(New_Name,
                    Test_Set.Sort_Index(Sort_Index_Position).Constants);
                New_Constant.Op_Name  := A_Strings.to_a(New_Name.s);
            end if;
            Subterm.Arguments(x)  := New_Constant;
```

```
      end if;
    end loop;
end  Check_for_Circularity;
```

4. Generate Output Terms

```
---------------------------------------------------------------------------------------------------
-- Generate_Output_Terms builds an OBJ3 input file which will be used to reduce the
-- IO List inputs.  After the reductions are complete, the output file is cleaned and the
-- canonical terms are parsed and stored into the output side of the IO List
---------------------------------------------------------------------------------------------------
with Types_and_Constants, Unix_Prcs;
with Text_IO;                    use Text_IO;
with Formal_Spec_Object;         use Formal_Spec_Object;
with Term_Definition_Pkg;        use  Term_Definition_Pkg;
with A_Strings;                  use A_Strings;
with Op_Defns_Pkg;               use Op_Defns_Pkg;
with Print_Term, Clean_Output_File, Term_Parser;

procedure Generate_Output_Terms
    (Query_Filename : in A_String;
    Formal_Spec       : in Formal_Spec_Def;
    Test_Set          : in Test_Set_Def;
    IO_List           : in out IO_List_Def) is

    Command_Line,
    New_Name,
    Temp_Script_Name,
    Temp_Shell_Name     : A_Strings.A_String;
    Obj_Temp_File,
    Obj_Shell_File      : Text_IO.File_Type;
    Temp                : Integer;
    Num_Constants       : Natural;
    An_IO_Rec           : IO_List_Def;
    An_Input            : Term_Access;

begin
    Temp_Script_Name := A_Strings.to_a(Query_Filename.s & ".script.obj");
    Text_IO.Create(Obj_Temp_File, Out_File, Temp_Script_Name.s);
    Command_Line := A_Strings.to_a("chmod 777 " & Temp_Script_Name.s);
    Temp := Unix_Prcs.Spawn(Command_Line);
    Text_IO.Put_Line(Obj_Temp_File, "in  newlisp.obj");
    Text_IO.Put_Line(Obj_Temp_File, "in  new-objects.obj");
    Text_IO.Put_Line(Obj_Temp_File, "in " & Query_Filename.s);
     -- need to "openr ."  and declare some constants
    Text_IO.Put_Line(Obj_Temp_File, "openr .");
    -- declare constants here
    for i in Test_Set.Sort_Index'Range
    loop
        Num_Constants := Const_Seq_Pkg.Length(Test_Set.Sort_Index(i).Constants);
```

```
          if Num_Constants > 0 then
               for j in 1..Num_Constants
               loop
                    Text_IO.Put_Line(Obj_Temp_File, "op " &
                    Const_Seq_Pkg.Fetch(Test_Set.Sort_Index(i).Constants, j).s &
                         " : -> " & Test_Set.Sort_Index(i).Sort_Name.s & " .");
               end loop;
          end if;
     end loop;
     Text_IO.Put_Line(Obj_Temp_File, "close");
     -- now enter the reduction loop
     Text_IO.Put_Line(Obj_Temp_File, "ev (do-red-loop)");
     -- now submit the terms from the IO_List, end each with a " ."
     An_IO_Rec := new IO_Pair_Rec;
     An_Input := new Term;
     An_IO_Rec := IO_List;
     loop
          exit when An_IO_Rec = null;
          An_Input := An_IO_Rec.Input;
          Print_Term(Obj_Temp_File, An_Input);
          Text_IO.Put_Line(Obj_Temp_File, " .");
          An_IO_Rec := An_IO_Rec.Next;
     end loop;
     -- end the reduction loop and quit
     Text_IO.Put_Line(Obj_Temp_File, ".");
     Text_IO.Put_Line(Obj_Temp_File, "q");
     Temp_Shell_Name := A_Strings.to_a(Query_Filename.s & ".shell");
     Text_IO.Create(Obj_Shell_File, Out_File, Temp_Shell_Name.s);
     Text_IO.Put_Line(Obj_Shell_File, "obj <$1 >$2");
     Command_Line := A_Strings.to_a("chmod 777 " & Temp_Shell_Name.s);
     Temp := Unix_Prcs.Spawn(Command_Line);
     -- Add ".output" to the file name
     New_name := A_Strings.to_a(Query_Filename.s & ".output");
     Command_Line := A_Strings.to_a(Temp_Shell_Name.s & " "
         & Temp_Script_Name.s & " " & New_name.s);
     Text_IO.New_Line;
     Text_IO.Put_Line("Running OBJ3 task to determine IO_List outputs.");
     Temp := Unix_Prcs.Spawn(Command_Line);
     Text_IO.Put_Line("Finished OBJ3 task.");
     Text_IO.Delete(Obj_Temp_File);
     Text_IO.Delete(Obj_Shell_File);
     Clean_Output_File(New_Name);
     Term_Parser.Parse_Output_Terms(New_Name, Formal_Spec, Test_Set, IO_List);
     -- temporary stuff
     Text_IO.New_line;
     Text_IO.Put_Line("Here are the I/O List outputs...");
     An_IO_Rec := IO_List;
     loop
         Text_IO.Put("    ");
         Print_Term(Text_IO.Standard_Output, An_IO_Rec.Output);
         Text_IO.New_Line;
```

```
          An_IO_Rec := An_IO_Rec.Next;
          exit when An_IO_Rec = null;
     end loop;
end Generate_Output_Terms;

----------------------------------------------------------------------------------------
-- Clean_Output_File removes extraneous OBJ3 output from the file containing the
-- reductions of the IO List terms
----------------------------------------------------------------------------------------
with Text_IO; use Text_IO;
with A_Strings, Unix_Prcs;

procedure Clean_Output_File
     (File_name : in A_Strings.A_String) is

     Temp_File, Output_File   : Text_IO.File_Type;
     Line                     : String(1..1000);
     Cmd_Line                 : A_Strings.A_String;
     Temp                     : Integer;
     Line_length              : Natural;

begin
     Text_IO.Open(Output_File, In_File, File_Name.s);
     Text_IO.Create(Temp_File, Out_File, File_name.s & ".temp");
     while not End_of_File(Output_File)
     loop
          Text_IO.Get_Line(Output_File, Line, Line_Length);
          if Line_Length >= 11 then
               exit when Line(1..11) = "!!!red-loop";
          end if;
     end loop;
     while not End_of_File(Output_File)
     loop
          Text_IO.Get_Line(Output_File, Line, Line_Length);
          if Line_Length >= 15 then
               exit when Line(1..15) = "!!!end-red-loop";
          end if;
          while not End_of_File(Output_File)
          loop
               if Line_Length >= 9 then
                    exit when Line(1..9) = "!!!result";
               end if;
               Text_IO.Get_Line(Output_File, Line, Line_Length);
          end loop;
          if not End_of_File(Output_File) then
               Put_Line(Temp_File, Line(1..Line_Length));
          end if;
          while not End_of_File(Output_File)
          loop
               Text_IO.Get_Line(Output_File, Line, Line_Length);
               Put_Line(Temp_File, Line(1..Line_Length));
```

```
            if Line_Length >= 13 then
                exit when Line(1..13) = "!!!end-result";
            end if;
        end loop;
    end loop;
    Text_IO.Close(Temp_File);
    Cmd_Line := A_Strings.to_a("mv " & File_Name.s & ".temp " & File_Name.s);
    --Text_IO.Put_Line(Cmd_Line.s);
    Text_IO.Delete(Output_File);
    --Text_IO.Put_Line("Command Line is " & Cmd_Line.s);
    Temp := Unix_Prcs.Spawn(Cmd_Line);
end Clean_Output_File;

-----------------------------------------------------------------------------------------
-- Parse_Output_Terms may be found in the file Termparse.y in Appendix Section C.
-----------------------------------------------------------------------------------------
```

5. Match

```
-----------------------------------------------------------------------------------------
-- Match attempts to match a query spec with a candidate spec. This procedure calls
-- Extract_Prolog, then Find_Maps, and finally, Test_Maps to determine the best
--score for the given candidate
-----------------------------------------------------------------------------------------
with A_Strings;
with Text_IO;                  use Text_IO;
with Types_and_Constants;      use Types_and_Constants;
with Formal_Spec_Object;       use Formal_Spec_Object;
with Term_Definition_Pkg;      use Term_Definition_Pkg;
with Op_Defns_Pkg;             use Op_Defns_Pkg;
with Check_Spec_Syntax, Extract_Prolog, Find_Maps, Test_Maps;

procedure Match
    (Query_Filename            : in     A_Strings.A_String;
    Candidate_Filename         : in     A_Strings.A_String;
    Score                      : in out Natural;
    Formal_Spec                : in out Formal_Spec_Def;
    Test_Set                   : in out Test_Set_Def;
    IO_List                    : in out IO_List_Def)  is

    Query_Prolog,
    Candidate_Prolog       :  A_Strings.A_String;
    Num_Maps               : Natural := 0;

begin
    Text_IO.New_Line;
    Text_IO.Put_Line("########################################
################");
    Text_IO.Put_Line("Matching " & Query_Filename.s & " with " &
            Candidate_Filename.s & ".");
```

```
    Extract_Prolog(Query_Filename, Candidate_Filename,
                   Query_Prolog, Candidate_Prolog);
    Find_Maps(Query_Prolog, Candidate_Prolog, Formal_Spec, Num_Maps);
    if Num_Maps = 0 then
        Score := 0;
        else if Num_Maps <= Types_and_Constants.Max_Maps then
            Test_Maps(Test_Set, Formal_Spec, IO_List, Query_Filename,
                      Candidate_Filename, Score);
            else  -- too many maps to consider
                Score := 1;
            end if;
    end if;

    exception
        when constraint_error =>
            Text_IO.Put("Usage is:  normalize_query ");
            Text_IO.Put_Line("<queryfile.obj>  <candidatefile.obj>");
end Match;

--------------------------------------------------------------------------------------------
-- Extract_Prolog extracts the Prolog code from the normalized query and candidate files
-- for use in mapping
--------------------------------------------------------------------------------------------
with A_Strings;
with Text_IO; use Text_IO;

procedure Extract_Prolog
    (Query_Filename,
    Candidate_Filename       : in  A_Strings.A_String;
    Query_Prolog,
    Candidate_Prolog         : in out A_Strings.A_String) is

    Query_File,
    Query_Prolog_File,
    Candidate_File,
    Candidate_Prolog_File    : Text_IO.File_Type;
    Line                     : String(1..1000);
    Norm_Query_Filename,
    Norm_Cand_Filename       : A_Strings.A_String;
    Line_length              : Natural;

begin
    -- The code below extracts the prolog statements from query.norm
    -- and puts them in query.prolog
    Text_IO.New_Line;
    Text_IO.Put_Line("Extracting Prolog from normalized files.");
    Text_IO.Put_Line("Query file is: " & Query_Filename.s);
    Norm_Query_Filename := A_Strings."&"(Query_Filename, ".norm");
    Query_Prolog := A_Strings."&"(Query_Filename, ".prolog");
    Text_IO.Open(Query_File, In_File, Norm_Query_Filename.s);
    Text_IO.Create(Query_Prolog_File, Out_File, Query_Prolog.s);
```

```
    while not End_of_File(Query_File)
    loop
        Text_IO.Get_Line(Query_File, Line, Line_Length);
        if Line_Length >= 9 then
            exit when Line(1..9) = "!!!prolog";
        end if;
    end loop;
    Text_IO.Get_Line(Query_File, Line, Line_Length);
    while not End_of_File(Query_File)
    loop
        Put_Line(Query_Prolog_File, Line(1..Line_Length));
        Text_IO.Get_Line(Query_File, Line, Line_Length);
        if Line_Length > 3 then
            exit when Line(1..3) = "!!!";
        end if;
    end loop;
    Text_IO.Close(Query_Prolog_File);
    Text_IO.Close(Query_File);

    -- The code below extracts the prolog statements from candidate.norm
    -- and puts them in candidate.prolog
    Text_IO.Put_Line("The candidate is: " & Candidate_Filename.s);
    Norm_Cand_Filename := A_Strings."&"(Candidate_Filename, ".norm");
    Candidate_Prolog := A_Strings."&"(Candidate_Filename, ".prolog");
    Text_IO.Open(Candidate_File, In_File, Norm_Cand_Filename.s);
    Text_IO.Create(Candidate_Prolog_File, Out_File, Candidate_Prolog.s);
    while not End_of_File(Candidate_File)
    loop
         Text_IO.Get_Line(Candidate_File, Line, Line_Length);
        if Line_Length >= 9 then
            exit when Line(1..9) = "!!!prolog";
        end if;
    end loop;
    Text_IO.Get_Line(Candidate_File, Line, Line_Length);
    while not End_of_File(Candidate_File)
    loop
        Put_Line(Candidate_Prolog_File, Line(1..Line_Length));
        Text_IO.Get_Line(Candidate_File, Line, Line_Length);
        if Line_Length > 3 then
            exit when Line(1..3) = "!!!";
        end if;
    end loop;
    Text_IO.Close(Candidate_Prolog_File);
    Text_IO.Close(Candidate_File);

    --Text_IO.Put_Line("Extracted Prolog statements from " &
     --              Norm_Query_filename.s & " and " &
     --              Norm_Cand_Filename.s & ".");
end Extract_Prolog;
```

```
------------------------------------------------------------------------------------------------
-- Find_Maps invokes Prolog to determine if the query spec maps to the candidate
-- component spec, and lexically analyzes the Prolog results, storing the maps
-- in  a linked-list map structure
------------------------------------------------------------------------------------------------
with Types_and_Constants,  Unix_Prcs, A_Strings;
with Text_IO;            use Text_IO;
with op_defns_pkg;       use op_defns_pkg;
with prolog_lex;         use prolog_lex;
with prolog_lex_dfa;     use prolog_lex_dfa;
with prolog_lex_io;
with Formal_Spec_Object, Module_is_Generic, Unchecked_Deallocation;

procedure Find_Maps
    (Query_Prolog, Candidate_Prolog    : in A_Strings.A_String;
    Formal_Spec                        : in out Formal_Spec_Object.Formal_Spec_Def;
    Number_of_Maps                     : in out Natural) is

    Command_Line,
    Candidate_File,
    Maps_Filename           : A_Strings.A_String;
    Query_Prolog_File,
    Maps_File ,
    Candidate_Prolog_File   : Text_IO.File_Type;
    Temp                    : Integer;
    Tok                     : Prolog_Lex.Token;
    Op_Definition           : Op_Defn_Type;
    Sort_Pos_Pair           : Sort_Position_Pair;
    A_Map, Head,
    Next_Map                : Op_Defns_Pkg.Map_Access;
    Counter                 : Natural;
    Candidate_is_Generic    : Boolean;
    Too_Many_Maps           : exception;

    procedure Check_Generic_Consistency is separate;

    procedure Free is new Unchecked_Deallocation(
        Object => A_Strings.String_rec,
        Name => A_Strings.A_String);

begin
    Maps_Filename := A_Strings.to_a(Candidate_Prolog.s & ".maps");
    Command_Line := A_Strings.to_a("findmappings " & Query_Prolog.s &
            " " & Candidate_Prolog.s & " " & Maps_Filename.s);
    Text_IO.Put_Line("Running Prolog Executable to find mappings.");
    Temp := Unix_Prcs.Spawn(Command_Line);
    Free(Command_Line);
    Candidate_File := A_Strings.Change(Candidate_Prolog,
        Candidate_Prolog.s'Length-6, Candidate_Prolog.s'Length, ".norm");
    Candidate_is_Generic := Module_is_Generic(Candidate_File);
```

```
-- Need a test here to determine if there are too many possibilities
-- to consider.
Text_IO.Open(Maps_File, In_File, Maps_Filename.s);
Number_of_Maps := 0;
while not End_of_File(Maps_File)
loop
    Text_IO.Skip_Line(Maps_File);
    Number_of_Maps := Number_of_Maps + 1;
end loop;
Text_IO.Close(Maps_File);
Number_of_Maps := Number_of_Maps - 1; -- The last line is not a map
--Text_IO.Put_Line("The number of maps found in the file was:" &
--        Natural'Image(Number_of_Maps) & ".");
If Number_of_Maps > Types_and_Constants.Max_Maps then
    Text_IO.Put_Line("The number of maps found was:" &
        Natural'Image(Number_of_Maps));
    raise Too_Many_Maps;
end if;

Number_of_Maps := 0;
Prolog_Lex_IO.Open_Input(Maps_Filename.s);
Text_IO.New_Line;
Text_IO.Put_Line("Scanning Prolog results.");
A_Map := new Maps;
A_Map.Map := Op_Defn_Seq_Pkg.Empty;
Head := A_Map;
Next_Map := A_Map;
Tok := yylex; -- Left bracket or Start_Generics
if Tok = Start_Generics then
    Text_IO.Put_Line(Query_Prolog.s & " does not map to " &
        Candidate_Prolog.s & ".");
else
    loop
        Number_of_Maps := Number_of_Maps + 1;
        --Text_IO.Put_Line("Scanning map:" &
                Natural'Image(Number_of_Maps));
        Tok := yylex; -- an op name
        loop
            Op_Definition.Op_Name := A_Strings.to_a(yytext);
            Tok := yylex; -- a comma
            Tok := yylex; -- number of domain parameters
            Op_Definition.Num_Parameters := Natural'Value(yytext);
             Tok := yylex; -- a comma
            Tok := yylex; -- Range sort
            Op_Definition.Range_Sort := A_Strings.to_a(yytext);
            Op_Definition.Domain_Sorts := Pair_Sequence_Pkg.Empty;
            Tok := yylex; -- a comma
            Counter := 1;
            while Counter <= Op_Definition.Num_Parameters
            loop
                Tok := yylex; -- a domain sort
```

```
                    Sort_Pos_Pair.Sort_Name := A_Strings.to_a(yytext);
                    Tok := yylex; -- a comma
                    Tok := yylex; -- the domain sort's position
                    Sort_Pos_Pair.Position := Natural'Value(yytext);
                    Tok := yylex; -- a comma
                    Pair_Sequence_Pkg.Add(Sort_Pos_Pair,
                            Op_Definition.Domain_Sorts);
                    Counter := Counter + 1;
                end loop; -- no more parameters for this operation
                Op_Defn_Seq_Pkg.Add(Op_Definition, A_Map.Map);
                Tok := yylex; -- End_of_Map token or an op-name
                exit when Tok = End_of_Map;
            end loop; -- this map is finished
            Tok := yylex; -- Generics_Start or another Map (left bracket)
            if Tok = Start_Generics then
                exit;
            else
                A_Map := new Maps;    -- create a new map structure
                Next_Map.Next := A_Map;  -- link the last structure to the new one
                Next_Map := A_Map;  -- Position the pointer to the current map
                A_Map.Map := Op_Defn_Seq_Pkg.Empty; --initialize the sequence
            end if;
        end loop;
        Text_IO.Put_Line("The number of maps found was:" &
        Natural'Image(Number_of_Maps));

        if Candidate_is_Generic then
            Check_Generic_Consistency;
        else
            Text_IO.Put_Line("Candidate component is not generic.");
        end if;
    end if;
    prolog_lex_io.close_input;
    Formal_Spec.Comp_Maps := Head;
    Text_IO.Put_Line("Number of maps remaining:" &
            Natural'Image(Number_of_Maps));

    -- discard the prolog files and the maps file
    Text_IO.Open(Query_Prolog_File, In_File, Query_Prolog.s);
    Text_IO.Delete(Query_Prolog_File);
    Text_IO.Open(Candidate_Prolog_File, In_File, Candidate_Prolog.s);
    Text_IO.Delete(Candidate_Prolog_File);
    Text_IO.Open(Maps_File, In_File, Maps_Filename.s);
    Text_IO.Delete(Maps_File);

    exception
        when Too_Many_Maps =>
            Text_IO.Put_Line("There are too many maps to consider.");
            Text_IO.Put_Line("Evaluate the candidate component manually.");
            raise Too_Many_Maps;
end Find_Maps;
```

```
-----------------------------------------------------------------------------------------------
-- Module_is_Generic is a function that returns true if a given specification is generic
-----------------------------------------------------------------------------------------------
with Obj3_Tokens; use Obj3_Tokens;
with Obj3_Lex, Obj3_Lex_IO;
with A_Strings;

function Module_is_Generic
    (File_Name : in A_Strings.A_String) return Boolean is

    Tok : Token;

begin
    Obj3_Lex_IO.Open_Input(File_name.s);
    loop  -- to look for generics-start-token
        Tok := Obj3_Lex.yylex;
        exit when (Tok = Generics_Start_Token) or (Tok = End_of_Input);
    end loop;
    if tok = End_of_Input then
        return False;
    end if;
    Tok := Obj3_Lex.yylex;
    if Tok /= Generics_End_Token then
        return True;
    else
        return False;
    end if;
end Module_is_Generic;

with A_Strings;
with Sequence_Pkg;
with Op_Defns_Pkg; use Op_Defns_Pkg;
with Get_Generic_Sorts, Modify_Sort;
with Types_and_Constants; use Types_and_Constants;

-----------------------------------------------------------------------------------------------
-- Check_Generic_Consistency is a rather complex procedure to determine if the bindings
-- in a given map are consistent with the generic parameters/sorts of a candidate
-- component specification
-----------------------------------------------------------------------------------------------
separate (Find_Maps)
procedure Check_Generic_Consistency is

    type Generic_Association_Rec is
    record
        Op_Name         : A_Strings.A_String;
        Position        : Natural;
        Generic_Name    : A_Strings.A_String;
        Gen_Posn        : Natural;
    end record;
```

```
    package Gen_Assoc_Seq_Pkg is new Sequence_Pkg(t => Generic_Association_Rec);

    Generic_Association_Seq      : Gen_Assoc_Seq_Pkg.Sequence;
    Generic_Consis_Seq           : Op_Defns_Pkg.Gen_Consis_Rec;
    Sort_of_Interest,
    The_Binding,
    Check_Op                     : A_Strings.A_String;
    Dom_Sorts                    : Op_Defns_Pkg.Pair_Sequence_Pkg.Sequence;
    Generic_Formal_Position,
    Check_Position,
    Number_of_Maps_Removed,
    Op_Location,
    Generic_Position,
    Num_Generics,
    Map_Count                    : Natural;
    Generic_Param_Seq            : Types_and_Constants.A_String_Seq_Pkg.Sequence;
    Generic_Assoc                : Generic_Association_Rec;
    A_Binding                    : Op_Defns_Pkg.Generic_Binding;
    Last_Map                     : Op_Defns_Pkg.Map_Access;
    Num_Associations             : Natural := 0;
    Inconsistent,
    Incomplete,
    Binding_Found,
    Impossible                   : Boolean;

begin --check_generic_consistency
    Text_IO.Put_Line("Checking generic consistency.");
    Generic_Param_Seq := Get_Generic_Sorts(Candidate_File);
    Num_Generics := Types_and_Constants.A_String_Seq_Pkg.
    Length(Generic_Param_Seq);
    --Text_IO.Put_Line("The candidate file has" & Natural'Image(Num_Generics)
    --            & " generic parameter(s).");
    Generic_Association_Seq := Gen_Assoc_Seq_Pkg.Empty;
    Tok := yylex; -- a comma
    Tok := yylex; -- a left bracket
    --Text_IO.Put("Lexing Prolog associations ");
    loop
        Num_Associations := Num_Associations + 1;
        Tok := yylex; -- a left bracket
        Tok := yylex; -- an op name
        Generic_Assoc.Op_Name := A_Strings.to_a(yytext);
        Tok := yylex; -- a comma
        Tok := yylex; -- position of the parameter
        Generic_Assoc.Position := Natural'Value(yytext);
        Tok := yylex; -- a comma
        Tok := yylex; -- A generic parameter name
        Generic_Assoc.Generic_Name :=
            A_Strings.Lower_to_Upper(A_Strings.to_a(yytext));
        Tok := yylex; -- a comma
        Tok := yylex; -- generic position
```

```
        Generic_Assoc.Gen_Posn := Natural'Value(yytext);
        Gen_Assoc_Seq_Pkg.Add(Generic_Assoc, Generic_Association_Seq);
        Tok := yylex; -- a right bracket
        Tok := yylex; -- comma or right bracket
        --Text_IO.Put(".  ");
        exit when Tok = Right_Bracket;
    end loop;  -- no more generic uses
    --Text_IO.New_Line;
    --Text_IO.Put_Line("There were" & Natural'Image(Num_Associations) &
    --              " uses of generic parameters.");
    A_Map := Head;
    Last_Map := Head;
    Number_of_Maps_Removed := 0;
    Map_Count := 0;
    loop  -- to examine each Map
        Map_Count := Map_Count + 1;
        --Text_IO.Put_Line("Examining Map:" & Natural'Image(Map_Count));
        Incomplete := False;
        Inconsistent := False;
        Impossible := False;
        Generic_Consis_Seq := (
                Size        => Num_Generics,
                Bindings    => (1..Num_Generics =>
                                (Generic_Name    => A_Strings.to_a(" "),
                                 Bound_To        => A_Strings.to_a("!!!"))),
                Length      => Num_Generics);
        -- Initialize the consistency sequence with the actual names of the
        -- generic parameters and their sorts to !!! unbound.
        for i in 1..Num_Generics
        loop
            A_Binding.Generic_Name :=
            Types_and_Constants.A_String_Seq_Pkg.Fetch(Generic_Param_Seq, i);
            A_Binding.Bound_To := A_Strings.to_a("!!!");   --unbound
            Generic_Consis_Seq.Bindings(i) := A_Binding;
        end loop;  -- to initialize generic consistency sequence

        -- Now check each generic use in the stored component, filling in the
        -- bindings for the generic formal parameters as we go
        for i in 1..Num_Associations loop  --for each generic use in the spec
            Check_Op := Gen_Assoc_Seq_Pkg.  -- Get Op_name that uses this generic
            Fetch(Generic_Association_Seq, i).Op_Name;
            --Text_IO.Put_Line("Checking generic consistency for: " &
            --              Check_Op.s);

            -- Was the op that uses this generic used in mapping
            -- to the query? If so, what is its location in the map?
            for j in 1..Op_Defn_Seq_Pkg.Length(A_Map.Map)
            loop
                Op_Location := j;
                Binding_Found := False;
                -- If the op was used in the mapping, to what sort was the generic
```

```
    -- parameter bound?
    if Check_Op.s = Op_Defn_Seq_Pkg.Fetch(A_Map.Map,
            Op_Location).Op_name.s then
        -- Get the position of the generic parameter in the op
        -- definition from the association list
        Check_Position := Gen_Assoc_Seq_Pkg.
        Fetch(Generic_Association_Seq, i).Position;
        Generic_Formal_Position := Gen_Assoc_Seq_Pkg.
            Fetch(Generic_Association_Seq, i).Gen_Posn;
        if Check_Position = 0 then -- its Range sort is generic
            Sort_of_Interest := Op_Defn_Seq_Pkg.
            Fetch(A_Map.Map, Op_Location).Range_Sort;
        else -- one of the domain sorts was generic
            Dom_Sorts := Op_Defn_Seq_Pkg.
            Fetch(A_Map.Map, Op_Location).Domain_Sorts;
            for c in 1..Op_Defn_Seq_Pkg.
                Fetch(A_Map.Map, Op_Location).Num_Parameters
            loop
                if Check_Position = Op_Defns_Pkg.Pair_Sequence_Pkg.
                        Fetch(Dom_Sorts, c).Position then
                    Sort_of_Interest := Op_Defns_Pkg.Pair_Sequence_Pkg.
                        Fetch(Dom_Sorts, c).Sort_Name;
                end if;
            end loop;
        end if;
        Sort_of_Interest := A_Strings.Lower_to_Upper(Sort_of_Interest);

        Generic_Position := Generic_Formal_Position;
        The_Binding := Generic_Consis_Seq.
                Bindings(Generic_Position).Bound_To;
        if The_Binding.s = "!!!" then -- it is currently unbound
            Generic_Consis_Seq.Bindings(Generic_Position).Bound_To :=
                Sort_of_Interest;
            Binding_Found := True;
        else    --it is bound, but is it consistent
                --with the current binding?
            if The_Binding.s /= Sort_of_Interest.s then
                Inconsistent := True;
                  --Text_IO.Put_Line(Generic_Consis_Seq.Bindings
                --    (Generic_Position).Generic_Name.s &
                --    " is currently bound to " & The_Binding.s & ".");
                --Text_IO.Put_Line("That is inconsistent with: " &
                --            Sort_of_Interest.s & ".");
            else
                Inconsistent := False;
                Binding_Found := True;
            end if;
        end if;
    end if;
    exit when Binding_Found;
end loop;
```

```
          exit when Inconsistent;
     end loop; -- for checking each generic use in the candidate spec

     -- Now check for completeness
     for j in 1..Num_Generics
     loop
          if Generic_Consis_Seq.Bindings(j).Bound_To.s = "!!!" then
               Incomplete := True;
               --Text_IO.Put_Line("No binding for generic parameter: "
               --    & Generic_Consis_Seq.Bindings(j).Generic_Name.s);
               --Text_IO.Put_Line("This map is incomplete.");
          end if;
     end loop;

     -- Now check that each instantiation is with a predefined Sort
     -- We cannot instantiate a generic candidate with something other
     -- than a predefined sort - but that would be a nice extension
     for j in 1..Num_Generics
     loop
          if not Predefined_Obj_Sorts_Pkg.Member(Modify_Sort(
                    Generic_Consis_Seq.Bindings(j).Bound_To),
                    Predef_Obj_Sorts_Set) then
               Impossible := True;
               --Text_IO.Put_Line("No instantiation possible for: "
               --    & Generic_Consis_Seq.Bindings(j).Generic_Name.s);
               --Text_IO.Put_Line("This map cannot be used.");
          end if;
     end loop;

     if Incomplete or Inconsistent or Impossible then
          if A_Map = Head then
               Head := A_Map.Next;  -- discard the Map at Head position
          else
               Last_Map := A_Map.Next;  -- discard A_Map
          end if;
          Number_of_Maps_Removed := Number_of_Maps_Removed + 1;
          Number_of_Maps := Number_of_Maps - 1;
     else  -- complete and consistent so let's save the bindings
          A_Map.Generic_Bindings  :=  Generic_Consis_Seq;
          Last_Map := A_Map;  -- update the last_map pointer
     end if;
     A_Map := A_Map.Next;  -- Let's try the next Map
     --Text_IO.Put(".  ");
     exit when A_Map = null;
end loop;  -- to check each Map for generic consistency and completeness
Text_IO.New_Line;
Text_IO.Put_Line("Number of maps discarded:" &
          Natural'Image(Number_of_Maps_Removed));

exception
     when Constraint_Error =>
```

```
            Text_IO.Put_Line("Aborted in Check_Generic_Consistency");
end Check_Generic_Consistency;

-----------------------------------------------------------------------------------------------
-- Get_Generic_Sorts extracts the names of generic parameters from a normalized file
-----------------------------------------------------------------------------------------------
with A_Strings;
with Text_IO;       use Text_IO;
with Obj3_Lex_IO, Obj3_Lex, Obj3_Lex_Dfa;
with Obj3_Tokens;     use Obj3_Tokens;
with Types_And_Constants;

function Get_Generic_Sorts
     (File_Name   : A_Strings.A_String)
        return Types_and_Constants.A_String_Seq_Pkg.Sequence is

     Tok                : Obj3_Tokens.token;
     A_Seq              : Types_and_Constants.A_String_Seq_Pkg.Sequence;
     Generics_Flag      : Boolean := false;

begin
     --Text_IO.Put_Line("Entered Get_Generic_Sorts!");
     Obj3_Lex_IO.Open_Input(File_Name.s);
     --Text_IO.Put_Line("Opened file: " & File_Name.s);
     A_Seq := Types_and_Constants.A_String_Seq_Pkg.Empty;
     loop
        Tok := Obj3_Lex.yylex;
        exit when (Tok = Generics_Start_Token) or (Tok = End_of_Input);
     end loop;
     loop
        Tok := Obj3_Lex.yylex;
        exit when (Tok = Generics_End_Token) or (Tok = End_of_Input);
        Types_and_Constants.A_String_Seq_Pkg.Add(
        A_Strings.to_a(Obj3_Lex_Dfa.yytext), A_Seq);
        Generics_Flag := true;
     end loop;
     Obj3_Lex_IO.Close_Input;
     if Generics_Flag then
        Text_IO.Put("Generic parameters are: ");
        for i in 1..Types_and_Constants.A_String_Seq_Pkg.length(A_Seq)
        loop
           Text_IO.Put(Types_and_Constants.A_String_Seq_Pkg.
              Fetch(A_Seq, i).s & " ");
        end loop;
        Text_IO.New_Line;
     end if;
     return A_Seq;
end Get_Generic_Sorts;
```

```
-----------------------------------------------------------------------------------------
-- Test_Maps
-----------------------------------------------------------------------------------------
with Term_definition_Pkg;          use  Term_definition_Pkg;
with Op_Defns_Pkg;                 use Op_Defns_Pkg;
with Formal_Spec_Object;           use Formal_Spec_Object;
with A_Strings;                    use A_Strings;
with Find_Correlation, Perform_Test, Get_Component_Name, Show_Map;
with Text_IO;

procedure Test_Maps
     (Test_Set                   : in Test_Set_Def;
     Formal_Spec                 : in Formal_Spec_Def;
     IO_List                     : in IO_List_Def;
     Query_Filename              : in  A_Strings.A_String;
     Candidate_Filename          : in  A_Strings.A_String;
     Best_Score                  : in out Natural) is

     A_Map                       : Map_Access;
     A_Term, New_Term            : Term_Access;
     IO_Pair                     : IO_List_Def;
     Component_Name              :  A_Strings.A_String;
     Score,
     Best_Map,
     Map_Count                   : Natural;

begin
     -- Let's start by getting a correlation between the sorts in the query
     -- and the sorts in the component. The correlation could be different
     -- for each map so we must have a separate one for each map
     Text_IO.New_Line;
     Text_IO.Put_Line("Correlating Sorts between Query and Maps.");
     A_Map := new Maps;
     A_Map := Formal_Spec.Comp_Maps;
     loop
          exit when A_Map = null;
          Find_Correlation(Test_Set, Formal_Spec.Op_Defns, A_Map);
          A_Map := A_Map.Next;
     end loop;

     -- Now we must get the name of the component defined in the candidate file
     -- in case there are generic parameters to instantiate
     if Formal_Spec.Comp_Maps.Generic_Bindings.Length > 0 then
          Component_Name := Get_Component_Name(Candidate_Filename);
     else
          Component_Name := A_Strings.Empty;
     end if;

     -- Here we must loop through each of the possible maps of a given
     -- component, invoking OBJ to check the similarity of the query outputs
     -- vs the component outputs
```

```
    Text_IO.New_Line;
    Text_IO.Put_Line("Testing the maps...");
    Text_IO.New_Line;
    A_Map := Formal_Spec.Comp_Maps;
    Best_Score := 0;
    Map_Count := 0;
    Best_Map := 0;
    loop
        exit when A_Map = null;
        Map_Count := Map_Count + 1;
        Text_IO.Put("Map:" & Natural'Image(Map_Count) & " Score:");
        Perform_Test(Candidate_Filename, Component_Name, Formal_Spec,
            A_Map, Test_Set, IO_List, Score);
        Text_IO.Put_Line(Natural'Image(Score) & ".");
        if Score >= Best_Score then
            Best_Score := Score;
            Best_Map := Map_Count;
        end if;
        A_Map := A_Map.Next;
    end loop;
    Text_IO.New_Line;
    Text_IO.Put_Line("Best map is #" & Natural'Image(Best_Map));
    Show_Map(Best_Map, Formal_Spec);
end Test_Maps;

-----------------------------------------------------------------------------------------------
-- Find_Correlation determines a correlation between the sorts of a query and the sorts of
-- a candidate component
-----------------------------------------------------------------------------------------------
with Term_definition_Pkg;   use Term_definition_Pkg;
with Op_Defns_Pkg;          use Op_Defns_Pkg;
with Types_and_Constants;   use Types_and_Constants;
with A_Strings;             use A_Strings;

procedure Find_Correlation
    (Test_Set       : in Test_Set_Def;
    Query_Ops       : in Op_Defn_Seq_Pkg.Sequence;
    A_Map           : in out Map_Access) is

    A_Range_Sort     : A_Strings.A_String;
    Location         : Natural;

begin
    -- make a list of sorts for this map like the one in the test set
    A_Map.Sort_Correlation := new Correlation_Rec
        (Size => Test_Set.Sort_Index'Last);

    -- first fill the array with the same sorts as the test set
    -- this takes care of all of the predefined sorts
    for i in Test_Set.Sort_Index'Range
    loop
```

```
            A_Map.Sort_Correlation.Sort_Correlation(i) :=
                A_Strings.to_a(Test_Set.Sort_Index(i).Sort_Name.s);
        end loop;

        -- Now check the range sorts of each op-definition in the query
        -- and find the corresponding sort in the candidate map
        for i in 1..Op_Defn_Seq_Pkg.Length(Query_Ops)
        loop
            A_Range_Sort := Op_Defn_Seq_Pkg.Fetch(Query_Ops, i).Range_Sort;
            if not Predefined_Obj_Sorts_Pkg.Member(A_Range_Sort,
                            Predef_Obj_Sorts_Set) then
                for x in Test_Set.Sort_Index'Range
                loop
                    if A_Range_Sort.s = Test_Set.Sort_Index(x).Sort_Name.s then
                        Location := x;
                        exit;
                    end if;
                end loop;
                A_Map.Sort_Correlation.Sort_Correlation(Location) := A_Strings.to_a(
                Op_Defn_Seq_Pkg.Fetch(A_Map.Map, i).Range_Sort.s);
            end if;
        end loop;
    end Find_Correlation;

    ----------------------------------------------------------------------------------------------------
    -- Show_Map shows the correlation between operators of two specifications given a map
    ----------------------------------------------------------------------------------------------------
    with Text_IO, Unchecked_Deallocation;
    with Op_Defns_Pkg, Formal_Spec_Object;
    use  Op_Defns_Pkg, Formal_Spec_Object;

    procedure Show_Map
        (Map_Count    : in Natural;
        Formal_Spec   : in Formal_Spec_Def) is

        A_Map : Map_Access;

        procedure Free is new Unchecked_Deallocation
            (Object     => Maps,
            Name        => Map_Access);

    begin
        Text_IO.New_Line;
        if Map_Count = 0 then
            Text_IO.Put_Line("No  Correlation");
         else
            A_Map := new Maps;
            A_Map := Formal_Spec.Comp_Maps;
            for x in 1..Map_Count-1
            loop
                A_Map := A_Map.Next;
```

```
        end loop;
        for x in 1..Op_Defn_Seq_Pkg.Length(Formal_Spec.Op_Defns)
        loop
            Text_IO.Put("    " & Op_Defn_Seq_Pkg.Fetch
                (Formal_Spec.Op_Defns, x).Op_Name.s & " -> ");
            Text_IO.Put_Line(Op_Defn_Seq_Pkg.Fetch(A_Map.Map, x).Op_Name.s);
        end loop;
        Free(A_Map);
    end if;
end Show_Map;

-----------------------------------------------------------------------------------------------
-- Perform_Test invokes OBJ3 to compare the output terms of the query with reduced
-- terms in the candidate component
-----------------------------------------------------------------------------------------------
with Types_and_Constants, Unix_Prcs;
with Text_IO;                 use Text_IO;
with Formal_Spec_Object;      use Formal_Spec_Object;
with Term_Definition_Pkg;     use Term_Definition_Pkg;
with A_Strings;               use A_Strings;
with Op_Defns_Pkg;            use Op_Defns_Pkg;
with Print_Term, Clean_Output_File, Evaluate_Results;
with Modify_Sort, Transform_Term;

procedure Perform_Test
    (Candidate_Filename      : in A_String;
    Component_Name           : in A_String;
    Formal_Spec              : in Formal_Spec_Def;
    Test_Map                 : in Map_Access;
    Test_Set                 : in Test_Set_Def;
    IO_List                  : in IO_List_Def;
    Score                    : out Natural) is

    Temp_Script_Name,
    Temp_Shell_Name      : A_Strings.A_String;
    Obj_Temp_File,
    Obj_Shell_File,
    New_File             : Text_IO.File_Type;
    Command_Line,
    New_Name             : A_Strings.A_String;
    Temp                 : Integer;
    Num_Constants        : Natural;
    An_IO_Rec            : IO_List_Def;
    Domain_Result,
    Candidate_Input      : Term_Access;
    Comma                : Boolean;

begin
    Temp_Script_Name := A_Strings.to_a(Candidate_Filename.s & ".script.obj");
    Text_IO.Create(Obj_Temp_File, Out_File, Temp_Script_Name.s);
    Command_Line := A_Strings.to_a("chmod 777 " & Temp_Script_Name.s);
```

```
Temp := Unix_Prcs.Spawn(Command_Line);
Text_IO.Put_Line(Obj_Temp_File, "in newlisp.obj");
Text_IO.Put_Line(Obj_Temp_File, "in " & Candidate_Filename.s);
-- must instantiate generic here.
if Test_Map.Generic_Bindings.Length > 0 then
    Text_IO.Put(Obj_Temp_File, "make " & Component_Name.s & "-NEW is " &
        Component_Name.s & "[");
    Comma := False;
    for i in 1..Test_Map.Generic_Bindings.Length
    loop
        if Comma then
            Text_IO.Put(Obj_Temp_File, ", ");
        end if;
        Comma := True;
        Text_IO.Put(Obj_Temp_File,
            Test_Map.Generic_Bindings.Bindings(i).Bound_to.s);
    end loop;
    Text_IO.Put_Line(Obj_Temp_File, "] endm");
end if;

-- need to "openr ." and declare some constants
Text_IO.Put_Line(Obj_Temp_File, "openr .");
-- declare constants here
for i in Test_Set.Sort_Index'Range
loop
    Num_Constants := Const_Seq_Pkg.Length(Test_Set.Sort_Index(i).Constants);
    if Num_Constants > 0 then
        for j in 1..Num_Constants
        loop
            Text_IO.Put_Line(Obj_Temp_File, "op " &
                Const_Seq_Pkg.Fetch(Test_Set.Sort_Index(i).Constants, j).s &
                " : -> " & Modify_Sort(Test_Map.Sort_Correlation.
                Sort_Correlation(i)).s & " .");
        end loop;
    end if;
end loop;
Text_IO.Put_Line(Obj_Temp_File, "close");
-- now enter the reduction loop
Text_IO.Put_Line(Obj_Temp_File, "ev (do-red-loop)");
-- now submit the terms from the IO_List, end each with a " ."
An_IO_Rec := new IO_Pair_Rec;
An_IO_Rec := IO_List;
loop
    exit when An_IO_Rec = null;
    Transform_Term(An_IO_Rec.Output, Domain_Result,
            Formal_Spec.Op_Defns, Test_Map, Test_Set);
    Transform_Term(An_IO_Rec.Input, Candidate_Input,
            Formal_Spec.Op_Defns, Test_Map, Test_Set);
    Text_IO.Put(Obj_Temp_File, "prove(");
    Print_Term(Obj_Temp_File, Candidate_Input);
    Text_IO.Put(Obj_Temp_File, ", ");
```

```
        Print_Term(Obj_Temp_File, Domain_Result);
        Text_IO.Put_Line(Obj_Temp_File, ") .");
        An_IO_Rec := An_IO_Rec.Next;
    end loop;
    -- end the reduction loop and quit
    Text_IO.Put_Line(Obj_Temp_File, ".");
    Text_IO.Put_Line(Obj_Temp_File, "q");
    Temp_Shell_Name := A_Strings.to_a(Candidate_Filename.s & ".shell");
    Text_IO.Create(Obj_Shell_File, Out_File, Temp_Shell_Name.s);
    Text_IO.Put_Line(Obj_Shell_File, "obj <$1 >$2");
    Command_Line := A_Strings.to_a("chmod 777 " & Temp_Shell_Name.s);
    Temp := Unix_Prcs.Spawn(Command_Line);
    -- Add ".output" to the file name
    New_name := A_Strings.to_a(Candidate_Filename.s & ".output");
    Command_Line := A_Strings.to_a(Temp_Shell_Name.s & " " &
        Temp_Script_Name.s & " " & New_name.s);
    --  Text_IO.New_Line;
    --  Text_IO.Put_Line("Running OBJ3 task to compare results.");
    Temp := Unix_Prcs.Spawn(Command_Line);
    Temp := Unix_Prcs.Spawn((A_Strings.to_a("cat " & New_name.s)));
    --  Text_IO.Put_Line("Finished OBJ3 task.");
    Text_IO.Delete(Obj_Temp_File);
    Text_IO.Delete(Obj_Shell_File);
    Clean_Output_File(New_Name);
    -- must evaluate the results here
    Evaluate_Results(New_Name, Score);
    Text_IO.Open(New_File, In_File, New_Name.s);
    Text_IO.Delete(New_File);
end Perform_Test;

---------------------------------------------------------------------------------------------------
-- Modify_Sort changes some special case sort names to their internal (to OBJ3) form
---------------------------------------------------------------------------------------------------
with A_Strings; use A_Strings;

function Modify_Sort
    (Sort : A_String) return A_String is

    Sort1 : A_String;

begin
    if Upper_to_Lower(Sort).s = "nznat" then
        return to_a("NzNat");
    else
        if Upper_to_Lower(Sort).s = "nzint" then
            return to_a("NzInt");
        else
            Sort1 := upper_to_lower(Sort);
            Sort1.s(1) := to_upper(Sort1.s(1));
            return Sort1;
        end if;
```

```
    end if;
end Modify_Sort;

-------------------------------------------------------------------------------------------
-- Get_Component_Name extracts the name of a component (object) from a normalized
-- specification file, for the purpose of generic instantiation
-------------------------------------------------------------------------------------------
with Obj3_Tokens;     use Obj3_Tokens;
with Obj3_Lex, Obj3_Lex_IO, Obj3_Lex_Dfa;
with A_Strings;

function Get_Component_Name
    (File_Name : in A_Strings.A_String) return A_Strings.A_String is

    Tok                  : Token;
    Norm_Filename,
    Component_Name       : A_Strings.A_String;

begin
    Norm_Filename := A_Strings."&"(File_name, ".norm");
    Obj3_Lex_IO.Open_Input(Norm_Filename.s);
    loop  -- to look for MOD_NAME_START_TOKEN
        Tok := Obj3_Lex.yylex;
        exit when (Tok = MOD_NAME_START_TOKEN) or (Tok = End_of_Input);
    end loop;
    Tok := Obj3_Lex.yylex;
    Component_Name := A_Strings.to_a(Obj3_Lex_Dfa.yytext);
    Obj3_Lex_IO.close_input;
    return Component_Name;
end Get_Component_Name;

-------------------------------------------------------------------------------------------
-- Transform_Term transforms the term from the I/O list to the domain of the candidate
-- component
-------------------------------------------------------------------------------------------
with Text_IO;
with Term_definition_Pkg;  use Term_definition_Pkg;
with Op_Defns_Pkg;         use Op_Defns_Pkg;
with Types_and_Constants;  use Types_and_Constants;
with A_Strings;            use A_Strings;
with Subsort;

procedure Transform_Term
    (From_Term   : in Term_Access;
    To_Term      : in out Term_Access;
    From_Map     : in Op_Defn_Seq_Pkg.Sequence;
    To_Map       : in Map_Access;
    Test_Set     : in Test_Set_Def) is

    Signature_Match      : Boolean := False;
    Domain_Match         : Boolean;
```

```
    From_Op_Def          : Op_Defn_Type;
    Location,
    Sort_Loc,
    Z                    : Natural;
    Subterm              : Term_Access;
    First,
    Second               : A_Strings.A_String;

begin
    --Text_IO.Put_Line("Transforming: " & From_Term.Op_Name.s & ":" &
    --             From_Term.Range_Sort.s & " with" &
    --               Natural'Image(From_Term.Num_Args) & " args.");
    -- Make a new empty term
    To_Term := new Term;

    if From_Term.Signature > 0 then  -- we know its signature already
        Signature_Match := True;
        Location := From_Term.Signature;
    else
         -- Let's look for it among the op-definitions
        for x in 1..Op_Defn_Seq_Pkg.Length(From_Map)
        loop
            From_Op_Def := Op_Defn_Seq_Pkg.Fetch(From_Map, x);
            if (From_Term.Op_Name.s = From_Op_Def.Op_Name.s) and
                    (From_Term.Range_Sort.s = From_Op_Def.Range_Sort.s) and
                    (From_Term.Num_Args = From_Op_Def.Num_Parameters) then
                Domain_Match := True;
                for y in 1..From_Term.Num_Args
                loop
                    First  := Upper_to_Lower(From_Term.Arguments(y).Range_Sort);
                    Second := Upper_to_Lower(Pair_Sequence_Pkg.Fetch
                                (From_Op_Def.Domain_Sorts, y).Sort_Name);
                    if (First.s /= Second.s) and then not Subsort(First, Second) then
                        Domain_Match := False;
                    end if;
                    A_Strings.Free(First);
                    A_Strings.Free(Second);
                end loop;
                if Domain_Match then
                    Signature_Match := True;
                    Location := x;
                    exit;
                end if;
            end if;
        end loop;
    end if;

    -- Maybe the term is one of the predefined terms
    if (not Signature_Match) and (From_Term.Num_Args > 0) then
        -- this is a predefined term
        To_Term.Op_Name  := A_Strings.to_a(From_Term.Op_Name.s);
```

```
    To_Term.Range_Sort := A_Strings.to_a(From_Term.Range_Sort.s);
    To_Term.Num_Args := From_Term.Num_Args;
    for x in 1..From_Term.Num_Args
    loop
        Transform_Term(From_Term.Arguments(x), Subterm,
            From_Map, To_Map, Test_Set);
        To_Term.Arguments(x) := new Term;
        To_Term.Arguments(x) := Subterm;
    end loop;
 end if;

-- Let's check if it's a constant
if (not Signature_Match) and (From_Term.Num_Args = 0) then
    -- this is a constant
    --Text_IO.Put_line(From_Term.Op_Name.s & " is a constant with sort "
    --            & From_Term.Range_Sort.s);
    To_Term.Op_Name := From_Term.Op_Name;
    To_Term.Num_Args := 0;
    for i in Test_Set.Sort_Index'Range
    loop
        if Upper_to_Lower(From_Term.Range_Sort).s =
                Upper_to_Lower(Test_Set.Sort_Index(i).Sort_Name).s then
            Sort_Loc := i;
            exit;
        end if;
    end loop;
    To_Term.Range_Sort := A_Strings.to_a(To_Map.Sort_Correlation.
            Sort_Correlation(Sort_loc).s);
end if;

-- Perhaps we found the map
if Signature_Match then
     --Text_IO.Put_Line("Found the signature for: " & From_Term.Op_Name.s);
    To_Term.Op_Name := A_Strings.to_a(Op_Defn_Seq_Pkg.Fetch(To_Map.Map,
                                        Location).Op_Name.s);
    --Text_IO.Put_Line("Corresponding Op_Name is: " & To_Term.Op_Name.s);
    To_Term.Range_Sort :=
        A_Strings.to_a(Op_Defn_Seq_Pkg.Fetch(To_Map.Map,
                        Location).Range_Sort.s);
    To_Term.Num_Args := From_Term.Num_Args;
    To_Term.Signature := From_Term.Signature;
    for i in 1..From_Term.Num_Args
    loop
        Z := Pair_Sequence_Pkg.Fetch(Op_Defn_Seq_Pkg.Fetch(To_Map.Map,
            Location).Domain_Sorts, i).Position;
        Transform_Term(From_Term.Arguments(Z), Subterm,
                        From_Map, To_Map, Test_Set);
        To_Term.Arguments(i) := new Term;
        To_Term.Arguments(i) := Subterm;
    end loop;
end if;
```

```
end Transform_Term;

------------------------------------------------------------------------------------------
-- Subsort checks predefined subsort relationships to support term transformation
------------------------------------------------------------------------------------------
with Text_IO;
with A_Strings; use A_Strings;

function Subsort
    (A, B : A_Strings.A_String) return Boolean is

    Result : Boolean := false;

begin
    --Text_IO.Put_Line("Is " & A.s & " a subsort of " & B.s & "?");

    if (Upper_to_Lower(A).s = "nznat") and (Upper_to_Lower(B).s = "nat") then
        Result := true;
    end if;
    if (Upper_to_Lower(A).s = "nat") and (Upper_to_Lower(B).s = "int") then
        Result := true;
    end if;
    if (Upper_to_Lower(A).s = "nznat") and (Upper_to_Lower(B).s = "int") then
        Result := true;
    end if;
    if (Upper_to_Lower(A).s = "nzint") and (Upper_to_Lower(B).s = "int") then
        Result := true;
    end if;
    if (Upper_to_Lower(A).s = "nznat") and (Upper_to_Lower(B).s = "nzint") then
        Result := true;
    end if;
    if (Upper_to_Lower(A).s = "zero") and (Upper_to_Lower(B).s = "nat") then
        Result := true;
    end if;
    if (Upper_to_Lower(A).s = "zero") and (Upper_to_Lower(B).s = "int") then
        Result := true;
    end if;
    return Result;
end Subsort;

------------------------------------------------------------------------------------------
-- Evaluate Results determines how many of he equivalence checks were positive out
-- of the number tried
------------------------------------------------------------------------------------------
with Term_Lex, Term_Lex_IO, Term_Lex_Dfa, Termparse_Tokens, A_Strings;
use  Term_Lex, Term_Lex_IO, Term_Lex_Dfa, Termparse_Tokens, A_Strings;

procedure Evaluate_Results
    (Result_File    : in A_String;
    Score           : out Natural) is
```

```
    Tok                 : Termparse_Tokens.Token;
    Num_Tests,
    Num_Successful   : Natural := 0;

begin
    Term_Lex_IO.Open_Input(Result_File.s);
    Tok := yylex;  -- result token
    while Tok /= End_of_Input
    loop
        Num_Tests := Num_Tests + 1;
        Tok := yylex;  -- Sort token
        if yytext = "Bool" then
            Tok := yylex; -- Term head
            if yytext = "true" then
                Num_Successful := Num_Successful + 1;
            end if;
        end if;
        loop
            Tok := yylex;
            exit when (Tok = Result_Start_Token) or (Tok = End_of_Input);
        end loop;
    end loop;
    Term_Lex_IO.Close_input;
    Score := (Num_Successful * 100) / Num_Tests;
end Evaluate_Results;
```

6. Support Code

```
-----------------------------------------------------------------------------------------------------
-- Print_Term prints a term to the specified location
-----------------------------------------------------------------------------------------------------
with Text_IO;                    use Text_IO;
with Term_Definition_Pkg;   use  Term_Definition_Pkg;

procedure Print_Term
    (Out_File : in File_Type;
    A_Term :   in Term_Access) is

begin
    if A_Term /= null then
        Put(Out_File,  A_Term.Op_Name.s);
        if A_Term.Num_Args > 0 then
            Put(Out_File,  "(");
            for i in 1..A_Term.Num_Args
            loop
                Print_Term(Out_File,  A_Term.Arguments(i));
                if i /= A_Term.Num_Args then
                    put(Out_File, ", ");
                end if;
            end loop;
```

```
            Put(Out_File, ")");
        end if;
    end if;
end Print_Term;

--------------------------------------------------------------------------------
-- Types_And_Constants defines useful constants and data structures
--------------------------------------------------------------------------------
with A_Strings;
with Set_Pkg;
with Sequence_Pkg;
with Text_IO; use Text_IO;

package Types_And_Constants is

    Max_Maps                : constant := 50;
    Spec_Filename_Type      : A_Strings.A_String;
    Op_Name_Type            : A_Strings.A_String;

    function Equal(X, Y : A_Strings.A_String) return Boolean;

    package Predefined_Obj_Sorts_Pkg is new Set_Pkg
        (t   => A_Strings.A_String,
        eq  => Equal);

    package Op_Set_Pkg is new Set_Pkg
        (t   => A_Strings.A_String,
        eq  => Equal);

    Predef_Obj_Sorts_Set : Predefined_Obj_Sorts_Pkg.Set;

    package A_String_Seq_Pkg is new Sequence_Pkg
        (t => A_Strings.A_String);

end Types_And_Constants;

package body Types_And_Constants is

    Sort_File           : File_Type;
    Sort_Name           : String(1..32);
    Name_Length         : Natural;

    function Equal(X, Y : A_Strings.A_String) return Boolean is
        Result : Boolean;
    begin
        Result := X.s = Y.s;
        return Result;
    end Equal;

    procedure Print_A_String(X : in A_Strings.A_String) is
    begin
```

```
        Text_IO.Put(X.s & " ");
    end Print_A_String;

    procedure Scan_Set is new
        Predefined_Obj_Sorts_Pkg.Scan(generate => Print_A_String);

begin
    Predefined_Obj_Sorts_Pkg.Empty(Predef_Obj_Sorts_Set);
    Text_IO.Open(Sort_File, In_File, "predefined-sorts");
    while not End_of_File(Sort_file)
    loop
        Text_IO.Get_Line(Sort_File, Sort_Name, Name_Length);
        Predefined_Obj_Sorts_Pkg.Add(A_Strings.to_a(Sort_name(1..Name_Length)),
                Predef_Obj_Sorts_Set);
    end loop;
    Text_IO.Close(Sort_File);
    --Text_IO.Put("Predefined sorts are: ");
    --Scan_Set(Predef_Obj_Sorts_Set);
    --Text_IO.New_Line;
end Types_And_Constants;

----------------------------------------------------------------------------------------------------
-- Op_Defns_Pkg defines op definition structure and Maps structure
----------------------------------------------------------------------------------------------------
with A_Strings, Sequence_Pkg;

package Op_Defns_Pkg is

    type Sort_Position_Pair is
        record
            Sort_Name     : A_Strings.A_String;
            Position      : Natural;
        end record;

    package Pair_Sequence_Pkg is new Sequence_Pkg(t => Sort_Position_Pair);

    type Op_Defn_Type is
        record
            Op_Name            : A_Strings.A_String;
            Num_Parameters     : Natural;
            Range_Sort         : A_Strings.A_String;
            Domain_Sorts       : Pair_Sequence_Pkg.Sequence;
        end record;

    package Op_Defn_Seq_Pkg is new Sequence_Pkg(t => Op_Defn_Type);

    type Generic_Binding is
        record
            Generic_Name    : A_Strings.A_String;
            Bound_To        : A_Strings.A_String;
        end record;
```

```
    type Array_Type is array(Positive range <>) of Generic_Binding;

    subtype Size_Range is integer range 0..100;

    type Gen_Consis_Rec(Size : Size_Range := 10) is
        record
            Bindings  : Array_Type(1..Size);
            Length    : Size_Range := 0;
        end record;

    type Correlation_Array is array(Positive range <>) of A_Strings.A_String;

    type Correlation_Rec(Size : Size_Range := 10) is
        record
            Sort_Correlation : Correlation_Array(1..Size);
        end record;

    type Correlation_Access is access Correlation_Rec;

    type Maps;

    type Map_Access is access Maps;

    type Maps is
        record
            Map                 : Op_Defns_Pkg.Op_Defn_Seq_Pkg.Sequence;
            Generic_Bindings    : Gen_Consis_Rec;
            Sort_Correlation    : Correlation_Access;
            Next                : Map_Access := null;
        end record;
end Op_Defns_Pkg;

----------------------------------------------------------------------------------------------------
-- Term_Definition_Pkg defines terms , test set, and I/O list
----------------------------------------------------------------------------------------------------
with A_Strings, Sequence_Pkg;

package Term_Definition_Pkg is

    Max_Arguments : constant Natural := 10;

    type Term;
    type Term_Access is access Term;
    type Access_Array is array(1..Max_Arguments) of Term_Access;

    type Term is
        record
            Op_Name      : A_Strings.A_String;
            Range_Sort   : A_Strings.A_String;
            Num_Args     : Natural := 0;
```

```
            Signature       : Natural := 0;
            Arguments       : Access_Array := (1..Max_Arguments => null);
        end record;

    package Const_Seq_Pkg is new Sequence_Pkg(t => A_Strings.A_String);

    type Sort_Index_Info is
        record
            Sort_Name       : A_Strings.A_String := A_Strings.to_a("!");
            Start           : Natural := 0;
            Stop            : Natural := 0;
            Constants       : Const_Seq_Pkg.Sequence := Const_Seq_Pkg.Empty;
        end record;

    type Sort_Index_Array is array(Positive range <>) of Sort_Index_Info;

    package Term_Sequence_Pkg is new Sequence_Pkg(t => Term_Access);

    type Test_Set_Rec(Size : Natural := 10) is
        record
            Sort_Index  : Sort_Index_Array(1..Size);
            Term_List   : Term_Sequence_Pkg.Sequence :=
                Term_Sequence_Pkg.Empty;
        end record;

    type Test_Set_Def is access Test_Set_Rec;

    type IO_Pair_Rec;
    type IO_List_Def is access IO_Pair_Rec;

    type IO_Pair_Rec is
        record
            Input       : Term_Access;
            Output      : Term_Access;
            Result      : A_Strings.A_String;
            Next        : IO_List_Def;
        end record;
end Term_Definition_Pkg;
```

C. INPUT SOURCE FOR ANALYZERS AND PARSERS

1. OBJ3 Lexical Analysis

```
-- Definitions of lexical classes
-- NOTE: Changes to standard OBJ3 are:
--        A Module ID must be all capitals and may contain digits or
--          the minus sign.
--        A Sort ID must start with a capital letter, followed by lower
--          case letters, digits, or the minus sign.
```

```
--      Identifiers such as variable names and operation names will
--      be recognized as either Module_ID, Sort_ID, or Symbol.

Digit           [0-9]
Int             [-+]?{Digit}+
IntList         {Int}[ ]*
Nat             {Digit}+
Letter          [a-zA-Z]
Alpha           {Letter}|{Digit}
Blank           [ \t]
Sym             [-_=+|:;/'<>?!@#$&*%~a-zA-Z0-9]+
EndExpr         {Blank}"."
ModuleID        [A-Z][-A-Z0-9]*
SortID          [A-Z][-a-z0-9]*
GenericSort     "Elt.("[^)]*")"
OpnameID        [a-z][-a-z0-9]*
OpsComment      "***(operations"
ShortComment    "***"[^(].*
StartComment    "***("
Attribute       assoc|comm|idem|memo|intrinsic
IdAttribute     id:|idr:
GatherAttr      gather[ ]*"("([eE&][ ]*)+")"
StratAttr       strat[ ]*"("{IntList}+")"
PrecAttr        prec[ ]*{Nat}

%%

obj                     return(OBJ_TOKEN);
is                      return(IS_TOKEN);
endo                    return(ENDO_TOKEN);
th                      return(TH_TOKEN);
endth                   return(ENDTH_TOKEN);
sort                    return(SORT_TOKEN);
op                      return(OP_TOKEN);
ops                     return(OPS_TOKEN);
op-as                   return(OP_AS_TOKEN);
protecting              return(PROTECTING_TOKEN);
extending               return(EXTENDING_TOKEN);
using                   return(USING_TOKEN);
with                    return(WITH_TOKEN);
and                     return(AND_TOKEN);
dfn                     return(DFN_TOKEN);
subsort                 return(SUBSORT_TOKEN);
subsorts                return(SUBSORTS_TOKEN);
var                     return(VAR_TOKEN);
vars                    return(VARS_TOKEN);
for                     return(FOR_TOKEN);
if                      return(IF_TOKEN);
eq                      return(EQ_TOKEN);
cq                      return(CQ_TOKEN);
bq                      return(BQ_TOKEN);
```

```
beq                          return(BEQ_TOKEN);
cbeq                         return(CBEQ_TOKEN);
cbq                          return(CBQ_TOKEN);
of                           return(OF_TOKEN);
as                           return(AS_TOKEN);
view                         return(VIEW_TOKEN);
from                         return(FROM_TOKEN);
to                           return(TO_TOKEN);
endv                         return(ENDV_TOKEN);
bsort                        return(BSORT_TOKEN);
poly                         return(POLY_TOKEN);
"!!!ops"                     return(OPS_START_TOKEN);
"!!!end-ops"                 return(OPS_END_TOKEN);
"!!!axioms"                  return(AXIOMS_START_TOKEN);
"!!!end-axioms"              return(AXIOMS_END_TOKEN);
"!!!sorts"                   return(SORTS_START_TOKEN);
"!!!end-sorts"               return(SORTS_END_TOKEN);
"!!!principal-sort"          return(PRINCIPAL_SORT_START_TOKEN);
"!!!end-principal-sort"      return(PRINCIPAL_SORT_END_TOKEN);
"!!!prolog"                  return(PROLOG_START_TOKEN);
"!!!end-prolog"              return(PROLOG_END_TOKEN);
"!!!generics"                return(GENERICS_START_TOKEN);
"!!!end-generics"            return(GENERICS_END_TOKEN);
"!!!module-name"             return(MOD_NAME_START_TOKEN);
"!!!end-module-name"         return(MOD_NAME_END_TOKEN);
{OpsComment}                 return(OPS_COMMENT_TOKEN);
{Attribute}                  return(ATTRIBUTE_TOKEN);
{IdAttribute}                return(ID_ATTRIBUTE_TOKEN);
{GatherAttr}                 return(GATHER_ATTR_TOKEN);
{StratAttr}                  return(STRAT_ATTR_TOKEN);
{PrecAttr}                   return(PREC_ATTR_TOKEN);
{GenericSort}                return(GENERIC_SORT_TOKEN);
"["                          return('[');
"]"                          return(']');
"("                          return('(');
")"                          return(')');
"{"                          return('{');
"}"                          return('}');
","                          return(',');
":"                          return(':');
"="                          return('=');
"."                          return('.');
"+"                          return('+');
"*"                          return('*');
"<"                          return('<');
"->"                         return(ARROW_TOKEN);
"::"                         return(DOUBLE_COLON_TOKEN);
{ShortComment}               return(SHORT_COMMENT_TOKEN);
{StartComment}               return(START_COMMENT_TOKEN);
{ModuleID}                   return(MODULE_ID_TOKEN);
{SortID}                     return(SORT_ID_TOKEN);
```

```
{OpnameID}                      return(OP_ID_TOKEN);
{Sym}                           return(SYMBOL_TOKEN);
{EndExpr}                       return(ENDEXPR_TOKEN);
[\n]                            {linenum;}
{Blank}*                        null;

%%

with Text_IO; use Text_IO;
with u_env;
with Obj3_Tokens; use Obj3_Tokens;

package Obj3_Lex is

    procedure lexit;
    function yylex return token;

end Obj3_Lex;

package body Obj3_Lex is

    procedure lexit is

        Tok : Token;

    begin  -- lexit
        Obj3_Lex_IO.Open_Input(U_Env.argv(1).s);
        loop
            Tok := YYLex;
            Text_IO.Put(Obj3_Lex_Dfa.YYText);
            Text_IO.New_Line;
            exit when Tok = End_of_Input;
        end;

 end lexit;

# #

end Obj3_Lex;
```

2. Predefined Term Lexical Analysis

```
-- Definition for a lexical analyzer for the predefined terms used in the test-set generator

Ident       [a-zA-Z][a-zA-Z0-9]*
Digit       [0-9]+
Blank       [ \t\n]

%%
```

```
"predef:"           return(Predef);
"constants:"        return(Constants);
"numterms:"         return(Numterms);
"name:"             return(Name);
"numargs:"          return(Numargs);
"!!!"               return(Generic_Op);
{Ident}             return(Identifier);
{Digit}             return(Number);
{Blank}*            null;

%%

with Text_IO; use Text_IO;
with u_env;

package predef_lex is

    subtype YYSType is integer;
    YYLVal, YYVal : YYSType;

    Syntax_Error : exception;

    type Token is
        (End_Of_Input, Error, Predef, Numterms, Name, Numargs, Constants,
        Generic_Op, Identifier, Number);

    procedure predeflex;
    function yylex return token;

end predef_lex;

package body predef_lex is

    procedure predeflex is
        Tok : Token;

    begin  -- predeflex
        predef_lex_io.Open_Input(U_Env.argv(1).s);
        loop
            Tok := YYLex;
            Text_IO.Put(predef_lex_dfa.YYText);
            Text_IO.New_Line;
            exit when Tok = End_of_Input;
        end loop;
    end predeflex;

##

end predef_lex;
```

3. Prolog Output Lexical Analysis

```
-- A lexical analyzer for the prolog output generated by the Findmappings
-- Prolog executable.

Ident        [a-zA-Z][a-zA-Z0-9]*
Digit        [0-9]+
Blank        [ \t\n]

%%

"[generic"   return(Start_Generics);
"end]"       return(End_of_Map);
"["          return(Left_Bracket);
","          return(Comma);
"]"          return(Right_Bracket);
{Ident}      return(Identifier);
{Digit}      return(Number);
{Blank}*     null;

%%

with Text_IO; use Text_IO;
with u_env;

package prolog_lex is

    subtype YYSType is integer;
    YYLVal, YYVal : YYSType;
    Syntax_Error : exception;

    type Token is
        (End_Of_Input, Error, Start_Generics, End_of_Map,
        Left_Bracket, Right_Bracket, Comma, Identifier, Number);

    procedure plex;
    function yylex return token;

end prolog_lex;

package body prolog_lex is

    procedure plex is

        Tok : Token;

    begin  -- plex
        prolog_lex_io.Open_Input(U_Env.argv(1).s);
        loop
            Tok := YYLex;
```

```
            Text_IO.Put(prolog_lex_dfa.YYText);
            Text_IO.New_Line;
            exit when Tok = End_of_Input;
        end loop;
    end plex;

# #

end prolog_lex;
```

4. Term Lexical Analysis

```
-- A lexical analyzer for obj output terms
-- NOTE: Changes to standard OBJ3 are:
--      Op names must begin with a lower case letter, i.e.
--        [a-z][a-z1-9]*

Digit           [0-9]
NegInt          "-"{Digit}+
Nat             {Digit}+
Float           [-]?{Digit}+"."{Digit}+
Blank           [ \t\n]
EndExpr         {Blank}"."
SortID          [A-Z][-a-zA-Z0-9]*
Qualified       "."{SortID}
OpnameID        [a-z][-a-z0-9]*

%%

if                  {YYText_Val := A_Strings.to_a(term_lex_dfa.yytext);
                        return(IF_TOKEN); }
then                {YYText_Val := A_Strings.to_a(term_lex_dfa.yytext);
                        return(THEN_TOKEN); }
else                {YYText_Val := A_Strings.to_a(term_lex_dfa.yytext);
                        return(ELSE_TOKEN); }
fi                  {YYText_Val := A_Strings.to_a(term_lex_dfa.yytext);
                        return(FI_TOKEN); }
"!!!result"         {return(RESULT_START_TOKEN);}
"!!!end-result"     {return(RESULT_END_TOKEN);}
"err!!"             {return(ERROR_TOKEN);}
"("                 {return('(');}
")"                 {return(')');}
","                 {return(',');}
"=="                {YYText_Val := A_Strings.to_a(term_lex_dfa.yytext);
                        return(EQUIV_TOKEN); }
"=/="               {YYText_Val := A_Strings.to_a(term_lex_dfa.yytext);
                        return(NOT_EQUIV_TOKEN); }
{SortID}            {YYText_Val := A_Strings.to_a(term_lex_dfa.yytext);
                        return(SORT_ID_TOKEN); }
{Qualified}         null;
```

```
{NegInt}             {YYText_Val := A_Strings.to_a(term_lex_dfa.yytext);
                         return(NEG_INT_TOKEN); }
{Float}              {YYText_Val := A_Strings.to_a(term_lex_dfa.yytext);
                         return(FLOAT_TOKEN); }
{Nat}                {YYText_Val := A_Strings.to_a(term_lex_dfa.yytext);
                         return(NAT_TOKEN); }
{OpnameID}           {YYText_Val := A_Strings.to_a(term_lex_dfa.yytext);
                         return(OP_ID_TOKEN); }
{EndExpr}            {return(ENDEXPR_TOKEN);}
{Blank}*                 null;

%%

with Text_IO; use Text_IO;
with u_env;
with termparse_tokens; use termparse_tokens;
with A_Strings;
with term_lex_dfa; use term_lex_dfa;

package term_lex is

    YYText_Val : A_Strings.A_String;
    procedure lex_term;
    function yylex return token;

end term_lex;

package body term_lex is

    procedure lex_term is

        Tok : Token;

    begin  -- lex_term
        term_lex_io.Open_Input(U_Env.argv(1).s);
        loop
            Tok := YYLex;
            Text_IO.Put(term_lex_dfa.YYText);
            Text_IO.New_Line;
            exit when Tok = End_of_Input;
        end loop;
  end lex_term;

# #

end term_lex;
```

5. Term Parser

```
--AYacc definitions for OBJ3  term parsing
```

```
%token      IF_TOKEN;
%token      THEN_TOKEN;
%token      ELSE_TOKEN;
%token      FI_TOKEN;
%token      RESULT_START_TOKEN;
%token      RESULT_END_TOKEN;
%token      ERROR_TOKEN;
%token      ENDEXPR_TOKEN;
%token      OP_ID_TOKEN;
%token      NAT_TOKEN;
%token      FLOAT_TOKEN;
%token      NEG_INT_TOKEN;
%token      EQUIV_TOKEN;
%token      NOT_EQUIV_TOKEN;
%token      SORT_ID_TOKEN

%with  Term_Definition_Pkg
%use   Term_Definition_Pkg
%with  A_Strings;
%use  A_Strings;

{
    type key_type is (Rterm, Rterm_List, Op, Empty);

    type YYSType(Key : Key_Type := Empty) is
        record
            case Key is
                when Rterm =>
                    Term_Val        : Term_Definition_Pkg.Term_Access;
                when Rterm_List =>
                    Count       : Natural;
                    Term_List_Val  :  Term_Definition_Pkg.Access_Array;
                when Op =>
                    Op_Name       : A_Strings.A_String;
                when Empty =>
                    null;
            end case;
        end record;
}

%%
 results :  results result
        | result
       ;

  result : RESULT_START_TOKEN SORT_ID_TOKEN
     {
         --Text_IO.Put_Line("Parsing term with sort: " & YYText_Val.s);
         Term_Range_Sort := A_Strings.to_a(YYtext_Val.s);
     }
```

```
term
{
  --Text_IO.New_Line;
  IO_List_Ptr.Output := new Term;
   --Text_IO.Put_Line("Made a new Term for the Output field");
     --Text_IO.Put_Line("It's Op_Name is: " & $4.Term_Val.Op_Name.s);
  IO_List_Ptr.Output := $4.Term_Val;
  --Text_IO.Put_Line("Assigned the term to the Output field");
    IO_List_Ptr.Output.Range_Sort := A_Strings.to_a(Term_Range_Sort.s);
  IO_List_Ptr := IO_List_Ptr.Next;
   --Text_IO.Put_Line("IO_List has been updated.");
  Term_Count := Term_Count + 1;}

  RESULT_END_TOKEN
;

term :  simple_term
   {$$ := (key => Rterm, Term_Val => new Term);
    $$.Term_Val := $1.Term_Val;
    --Text_IO.Put_Line("Parsed a Simple Term: " & $$.Term_Val.Op_Name.s);
  }

  | if_then_else
   {$$ := (key => Rterm, Term_Val => new Term);
    $$.Term_Val := $1.Term_Val;}

  | term_with_args
   {$$ := (key => Rterm, Term_Val => new Term);
    $$.Term_Val := $1.Term_Val;
  --Text_IO.Put_Line("Assigned: " & $1.Term_Val.Op_Name.s & " to 'term'.");
  }
 ;

if_then_else : IF_TOKEN term THEN_TOKEN term ELSE_TOKEN term FI_TOKEN
      {$$ := (key => Rterm, Term_Val => new Term);
         $$.Term_Val.Op_name := A_Strings.to_a("if-then-else");
        $$.Term_Val.Range_Sort := A_Strings.to_a("Bool");
      $$.Term_Val.Num_args := 3;
       $$.Term_Val.Arguments(1) := new Term;
        $$.Term_Val.Arguments(1) := $2.Term_Val;
       $$.Term_Val.Arguments(2) := new Term;
        $$.Term_Val.Arguments(2) := $4.Term_Val;
       $$.Term_Val.Arguments(3) := new Term;
        $$.Term_Val.Arguments(3) := $6.Term_Val;}
    ;

term_with_args : OP_ID_TOKEN
      {$1 := (key => Op, Op_Name => YYText_Val);}

      '(' term_list ')'
```

```
           {$$ := (key => Rterm, Term_Val => new Term);
            $$.Term_Val.Op_Name := $1.Op_Name;
            $$.Term_Val.Num_Args := $4.Count;
             $$.Term_Val.Arguments := $4.Term_List_Val;
             $$.Term_Val.Range_Sort := Get_Range_Sort($$.Term_Val,
            Test_Set, Formal_Spec);
             --Text_IO.Put_Line("Finished parse (term-with-args) of: " &
          --              $$.Term_Val.Op_Name.s);
        }
        ;

term_list :   term_list ',' term
          {$$ := (key => Rterm_List, Count => $1.Count + 1,
                 Term_List_Val => $1.Term_List_Val);
           $$.Term_List_Val($$.Count) := new Term;
            $$.Term_List_Val($$.Count) := $3.Term_Val;}

       | term
          {$$ := (key => Rterm_List, Count => 1,
               Term_List_Val =>
                     (1..Term_Definition_Pkg.Max_Arguments => null));
           $$.Term_List_Val(1) := new Term;
            $$.Term_List_Val(1) := $1.Term_Val;}
       ;

simple_term :
            NEG_INT_TOKEN
             {$$ := (key => Rterm, Term_Val => new Term);
              $$.Term_Val.Op_Name := YYText_Val;
               $$.Term_Val.Range_Sort := A_Strings.to_a("Int");
               --Text_IO.Put_Line("The Op id is: " & YYText_Val.s);
          }
          | FLOAT_TOKEN
             {$$ := (key => Rterm, Term_Val => new Term);
              $$.Term_Val.Op_Name := YYText_Val;
               $$.Term_Val.Range_Sort := A_Strings.to_a("Float");
               --Text_IO.Put_Line("The Op id is: " & YYText_Val.s);
          }
          | NAT_TOKEN
             {$$ := (key => Rterm, Term_Val => new Term);
              $$.Term_Val.Op_Name := YYText_Val;
             if YYText_Val.s = "0" then
                  $$.Term_Val.Range_Sort := A_Strings.to_a("Zero");
          else
                  $$.Term_Val.Range_Sort := A_Strings.to_a("Nat");
           end if;
               --Text_IO.Put_Line("The Op id is: " & YYText_Val.s);
         }
         | OP_ID_TOKEN
            {$$ := (key => Rterm, Term_Val => new Term);
             $$.Term_Val.Op_Name := YYText_Val;
```

```
            $$.Term_Val.Range_Sort := Get_Range_Sort($$.Term_Val,
            Test_Set, Formal_Spec);
            --Text_IO.Put_Line("The Op id is: " & YYText_Val.s);
        }
      ;


%%

with Term_Definition_Pkg, Op_Defns_Pkg, Formal_Spec_Object, A_Strings;
use  Term_Definition_Pkg, Op_Defns_Pkg, Formal_Spec_Object;

package term_parser is

    echo : boolean := false;
    number_of_errors : natural := 0;

    procedure Parse_Output_Terms
        (New_Name    : in A_Strings.A_String;
        Formal_Spec  : in Formal_Spec_Def;
        Test_Set     : in Test_Set_Def;
        IO_List      : in out IO_List_Def);

end term_parser;

with termparse_tokens, termparse_goto, termparse_shift_reduce;
with term_lex, Text_IO, term_lex_dfa, term_lex_io, A_Strings;
with Term_Definition_Pkg, Op_Defns_Pkg, Formal_Spec_Object;

use  termparse_tokens, termparse_goto, termparse_shift_reduce;
use  term_lex, Text_IO, A_Strings;
use  Term_Definition_Pkg, Op_Defns_Pkg, Formal_Spec_Object;

package body term_parser is

    procedure yyerror ( s: in string := "syntax error") is
        space : integer;
    begin
        number_of_errors := number_of_errors + 1;
        Text_IO.new_line;
        Text_IO.put("Line" & integer'image(lines-1) &": ");
        Text_IO.put_line(term_lex_dfa.yytext);
        space:=integer(term_lex_dfa.yytext'length)+integer'image(lines)'length+5;
        for i in 1 .. space loop
            put("-");
        end loop;
        put_line("^ syntax error");
    end yyerror;

    function Get_Range_Sort
        (A_Term: in Term_Access;
```

```
        Test_Set : in Test_Set_Def;
        Formal_Spec : in Formal_Spec_Def)  return A_Strings.A_String is separate;

    procedure Parse_Output_Terms
        (New_Name     : in A_Strings.A_String;
        Formal_Spec : in Formal_Spec_Def;
        Test_Set    : in Test_Set_Def;
        IO_List     : in out IO_List_Def) is

        IO_List_Ptr             : IO_List_Def;
        Term_Range_Sort         :  A_Strings.A_String;
        Term_Count              : Natural := 0;

##%procedure_parse

    begin --Parse_Output_Terms
        IO_List_Ptr := IO_List;
        Term_Lex_IO.Open_Input(New_Name.s);
        yyparse;
        Term_Lex_IO.Close_Input;
        Text_IO.New_Line;
        Text_IO.Put_Line("Parsed" & Natural'Image(Term_Count) & " terms.");
    end Parse_Output_Terms;
end term_parser;

-----------------------------------------------------------------------------------------
-- Get_Range_Sort tries to determine the range sort of a term being parsed
-----------------------------------------------------------------------------------------
with Text_IO;
with Subsort;

separate (Term_Parser)
function Get_Range_Sort
    (A_Term       : in Term_Access;
    Test_Set    : in Test_Set_Def;
    Formal_Spec : in Formal_Spec_Def) return A_Strings.A_String is

    Result              : A_Strings.A_String := A_Strings.to_a("Unknown");
    Signature_Found,
    Domain_Match        : Boolean := false;
    Op_Def              : Op_Defn_Type;

     procedure Check_Ops(Op_Sequence : Op_Defn_Seq_Pkg.Sequence) is separate;

begin
    --Text_IO.Put_Line("Checking range sort for: " & A_Term.Op_Name.s & ".");
    if A_Term.Num_Args = 0 then -- check true, false, and constants
        if (A_Term.Op_Name.s = "true") or (A_Term.Op_Name.s = "false") then
            Signature_Found := true;
            Result := A_Strings.to_a("Bool");
        else -- check constants
```

```
            for x in Test_Set.Sort_Index'Range loop
                for y in 1..Const_Seq_Pkg.Length(Test_Set.
                        Sort_Index(x).Constants) loop
                    if A_Term.Op_Name.s = Const_Seq_Pkg.Fetch
                            (Test_Set.Sort_Index(x).Constants, y).s then
                        Signature_Found := true;
                        Result := A_Strings.to_a(Test_Set.Sort_Index(x).Sort_Name.s);
                        exit;
                    end if;
                end loop;
                exit when Signature_Found;
            end loop;
        end if;
    end if;

    -- May need to check the op_definitions
    if not Signature_Found then
        --Text_IO.Put_Line("Checking the export ops.");
        Check_Ops(Formal_Spec.Op_Defns);
    end if;

    -- May need to check the hidden ops
    if not Signature_Found then
        --Text_IO.Put_Line("Checking the hidden ops.");
        Check_Ops(Formal_Spec.Hidden_Ops);
    end if;

    return Result;
end Get_Range_Sort;

----------------------------------------------------------------------------------------------
-- Check_Ops compares Op definition structures to term structures to support the
-- Get_Range_Sort procedure
----------------------------------------------------------------------------------------------
with Unchecked_Deallocation;

separate (Term_Parser.Get_Range_Sort)
procedure Check_Ops
    (Op_Sequence : Op_Defn_Seq_Pkg.Sequence) is

    First, Second : A_Strings.A_String;

    procedure Free is new Unchecked_Deallocation(
                    Object => A_Strings.String_rec,
                    Name => A_Strings.A_String);

begin
    --Text_IO.Put_Line("Is" & Natural'Image(A_Term.Num_Args) &
    --              "-" & A_Term.Op_Name.s & " among: ");
    for x in 1..Op_Defn_Seq_Pkg.Length(Op_Sequence)
    loop
```

```
        Op_Def := Op_Defn_Seq_Pkg.Fetch(Op_Sequence, x);
        --Text_IO.Put_Line(Natural'Image(Op_Def.Num_Parameters) & "-"
        --        & Op_Def.Op_Name.s & " ");
        if (A_Term.Op_Name.s = Op_Def.Op_Name.s) and
              (A_Term.Num_Args = Op_Def.Num_Parameters) then
            Domain_Match := true;
            for y in 1..A_Term.Num_Args
            loop
                First := A_Strings.Upper_to_Lower
                    (A_Term.Arguments(y).Range_Sort);
                Second := A_Strings.Upper_to_Lower(Pair_Sequence_Pkg.Fetch
                    (Op_Def.Domain_Sorts, y).Sort_Name);
                if (First.s /= Second.s) and then not Subsort(First, Second) then
                    Domain_Match := False;
                    exit;
                end if;
                Free(First);
                Free(Second);
            end loop;
            if Domain_Match then
                 Signature_Found := true;
                --Text_IO.Put_Line("Found the right signature!");
                Result := A_Strings.to_a(Op_Def.Range_Sort.s);
                exit;
            end if;
        end if;
    end loop;
end Check_Ops;
```

D. PROLOG SOURCE CODE

```
-- The following Prolog code, from the file maprules, will determine the mappings
-- between two formal specifications given their transformed signatures.

startup :- compile(library(basics)), unix(argv([A,B,C])), [A], [B],
       open(C, write, OutStream), query(OutStream), close(OutStream), halt.

startup : - halt.

store(OutStream, L) :- write(OutStream, L), nl(OutStream).

unique([_]).
unique([X|T]) :- \+member(X, T), unique(T).

--Notes
--Findmappings finds a correspondence between two specification
--signatures represented as Prolog predicate expressions.

--To create a findmappings executable:
     Enter Prolog.
```

Load maprules file with |?- [maprules].
Save state with |?- save_program(findmappings, startup).
Halt prolog with |? halt.

E. LISP SOURCE CODE

The Lisp source code contained in this section is *modified* Lisp code extracted from the OBJ3 environment [SRI88]. It is intended to be imported into each OBJ3 session to provide increased functionality.

```
in new-objects

-- myprint$op_brief  prints operator definitions in a simple format
eval-quiet
  (defun myprint$op_brief (op)
     (princ "op ")
     (print$simple_princ_open (operator$name op))
     (princ " : ")
     (when (operator$arity op)
     (print$sort_list_open obj$current_module (operator$arity op))
     (princ " "))
     (princ "-> ")
     (print$sort_name obj$current_module (operator$coarity op)))

-- print-ops prints a series of op definitions
eval-quiet
  (defun print-ops ()
     (print$next)
     (princ  "!!!ops")
     (let ((mod *mod_eval$$last_module*) (omit *print$ignore_mods*))
     (when (module$operators mod)
         (let ((obj$current_module mod))
              (dolist (op (module$operators mod))
              (unless (let ((opmod (operator$module op)))
              (and (not (eq mod opmod)) (member opmod omit)))
              (print$next)
              (myprint$op_brief op)
              (princ  "  ."))))))
     (print$next)
     (princ  "!!!end-ops")
     (print$next))

-- print-axioms prints all axioms defined for an object
eval-quiet
  (defun print-axioms ()
     (print$next)
     (princ  "!!!axioms")
     (let ((mod *mod_eval$$last_module*) (omit *print$ignore_mods*))
         (if (module$is_compiled  mod)
```

```
            (dolist (op (reverse (module$operators mod)))
                (unless (let ((opmod (operator$module op)))
                        (and (not (eq mod opmod)) (member opmod omit)))
                    (dolist (r (module$all_rules mod op))
                        (when (or *print$all_eqns* (null (rule$kind r)))
                            (print$next)
                            (print$rule_brief r)
                            (princ " ."))
                        (when  *print$all_eqns*
                            (dolist (er (rulex$a_extensions r))
                                (when er
                                    (print$next)
                                    (print$rule_brief er)
                                    (princ " .")))
                            (dolist (er (rulex$ac_extension r))
                                (when er
                                    (print$next)
                                    (print$rule_brief er)
                                    (princ " ."))))))))
            (dolist (r (reverse (module$equations mod)))
                (print$next)
                (print$rule_brief r)
                (princ " ."))))
    (print$next)
    (princ  "!!!end-axioms")
    (print$next))

-- print-sorts prints all of the sorts defined for an object
eval-quiet
  (defun print-sorts ()
    (print$next)
    (princ  "!!!sorts")
    (let ((mod  *mod_eval$$last_module*))
        (when (module$sorts mod)
            (let ((so (module$sort_order mod))
                (modprs (module$principal_sort  mod)))
            (when modprs
                (print$next)
                (princ "sort ")
                (print$sort_info mod so modprs)
                (princ " ."))
            (dolist (s (reverse (module$sorts mod)))
                (unless (or (eq s *obj$sort_Universal*) (eq s modprs))
                (print$next)
                (princ "sort ")
                (print$sort_info mod so s)
                (princ " .")))))))
    (print$next)
    (princ  "!!!end-sorts")
    (print$next))
```

```
-- print-ps prints the principal sort for an object
eval-quiet
  (defun print-ps ()
     (print$next)
     (princ  "!!!principal-sort")
     (print$next)
     (let ((mod  *mod_eval$$last_module*))
         (let ((obj$current_module mod))
             (print$module_sort_info mod (module$sort_order mod)
                 (module$principal_sort  mod))
             (princ  "!!!end-principal-sort")
             (print$next))))

-- print-generics prints the name of the generic parameters defined for a module
eval-quiet
  (defun print-generics ()
     (print$next)
     (princ  "!!!generics")
     (print$next)
     (when (module$parameters *mod_eval$$last_module*)
         (dolist (x (module$parameters *mod_eval$$last_module*))
             (princ (car (module$name (cdr x))))
             (print$next)))
     (princ  "!!!end-generics")
     (print$next))

--print-mod-name prints the name of a module
eval-quiet
   (defun print-mod-name ()
     (print$next)
     (princ  "!!!module-name")
     (print$next)
     (let ((mod  *mod_eval$$last_module*))
         (let ((obj$current_module mod))
             (print$mod_name  mod)
             (print$next)
             (princ  "!!!end-module-name")
             (print$next))))

-- do-red-loop invokes a reduction loop to reduce a series of  terms
eval-quiet
  (defun do-red-loop ()
     (print$next)
     (princ  "!!!red-loop")
     (print$next)
     (ci$red_loop  *mod_eval$$last_module*)
     (princ  "!!!end-red-loop")
     (print$next))

-- ci$red performs a reduction on a single term
eval-quiet
```

```
(defun ci$red (mod preterm)
  (let ((obj$current_module (if (consp mod) (modexp_eval$eval mod) mod)))
      (let ((res
          (rew$!normalize (parse$parse mod preterm *obj$sort_Universal*))))
          (princ  "!!!result")
          (terpri)
          (print$short_sort_name (term$sort res))
          (terpri)
          (let ((*show-retracts* nil)) (term$print res))
              (that$set res)
              (terpri)
              (princ  "!!!end-result")
              (terpri))))
```

E. OBJ3 PREDEFINED OBJECTS

This section contains definitions of the predefined OBJ3 objects used in query by consistency, which are simply prefix reformulations of the predefined objects provided by OBJ3 [SRI88].

```
obj TRUTH is
  protecting TRUTH-VALUE .
  protecting UNIVERSAL .
  op if-then-else : Bool Universal Universal -> Universal
       [polymorphic obj_BOOL$if_resolver intrinsic strategy (1 0)
      gather (& & &) prec 0] .
  op == : Universal Universal -> Bool [strategy (1 2 0) prec 51] .
  op =/= : Universal Universal -> Bool [strategy (1 2 0) prec 51] .
  var XU : Universal .
  var YU : Universal .
  eq if-then-else(true, XU, YU) = XU .
  eq if-then-else(false, XU, YU) = YU .
  beq ==(XU, YU) = (obj_bool$coerce_to_bool (term$equational_equal xu
   yu)) .
  beq =/=(XU, YU) = (obj_bool$coerce_to_bool (not (term$equational_equal
   xu yu))) .
endo
```

*** Note that the object BOOL contains a new operator *prove* which is used to make equivalence checks between terms.

```
obj BOOL is
  protecting TRUTH .
  op and : Bool Bool -> Bool [assoc comm idr: true
                 strat (1 2 0)
                  gather (e E) prec 55] .
  op or  : Bool Bool -> Bool [assoc comm idr: false
                 strat (1 2 0)
```

```
                 gather (e E) prec 59] .
  op xor : Bool Bool -> Bool [assoc comm idr: false
                 strat (1 2 0)
                 gather (e E) prec 57] .
  op not : Bool -> Bool [prec 53] .
  op implies : Bool Bool -> Bool [gather (e E) prec 61] .
  op prove : Universal Universal -> Bool [strat (1 0)] .
  var A : Bool .
  var B : Bool .
  eq and(false, A) = false .
  eq or(true, A) = true .
  eq xor(true, true) = false .
  eq not(true) = false .
  eq not(false) = true .
  eq implies(A, B) = or(not(A), B) .
  eq prove(XU,YU) = ==(quote(XU), quote(YU)) .
endo

obj NZNAT is
  bsort NzNat
     (obj_NZNAT$is_NzNat_token obj_NZNAT$create_NzNat prin1
      obj_NZNAT$is_NzNat) .
  protecting BOOL .
  op sum : NzNat NzNat -> NzNat [assoc comm prec 33] .
  op diff : NzNat NzNat -> NzNat [comm] .
  op quot : NzNat NzNat -> NzNat [gather (E e) prec 31] .
  op less : NzNat NzNat -> Bool [prec 51] .
  op lesseq : NzNat NzNat -> Bool [prec 51] .
  op gtr : NzNat NzNat -> Bool [prec 51] .
  op gtreq : NzNat NzNat -> Bool [prec 51] .
  op succ : NzNat -> NzNat [prec 15] .
  op mult : NzNat NzNat -> NzNat [assoc comm prec 31 idr: 1] .
  var NN : NzNat .
  var NM : NzNat .
  bq sum(NN, NM) = (+ NN NM) .
  bq diff(NN, NM) = (if (= NN NM) 1 (abs (- NN NM))) .
  bq mult(NN, NM) = (* NN NM) .
  bq quot(NN, NM) = (if (> NN NM) (truncate NN NM) 1) .
  bq less(NN, NM) = (< NN NM) .
  bq lesseq(NN, NM) = (<= NN NM) .
  bq gtr(NN, NM) = (> NN NM) .
  bq gtreq(NN, NM) = (>= NN NM) .
  bq succ(NN) = (1+ NN) .
endo

obj NAT is
  bsort Nat
     (obj_NAT$is_Nat_token obj_NAT$create_Nat prin1
      obj_NAT$is_Nat) .
  protecting NZNAT .
  bsort Zero
```

```
    (obj_NAT$is_Zero_token obj_NAT$create_Zero prin1
    obj_NAT$is_Zero) .
  subsorts NzNat < Nat .
  subsorts Zero < Nat .
  op sum  : Nat Nat -> Nat [assoc comm idr: 0 prec 33] .
  op sd : Nat Nat -> Nat [comm] .
  op mult : Nat Nat -> Nat [assoc comm idr: 1 prec 31] .
  op quo : Nat NzNat -> Nat [gather (E e) prec 31] .
  op rem : Nat NzNat -> Nat [gather (E e) prec 31] .
  op divides : NzNat Nat -> Bool [prec 51] .
  op less  : Nat Nat -> Bool [prec 51] .
  op lesseq : Nat Nat -> Bool [prec 51] .
  op gtr  : Nat Nat -> Bool [prec 51] .
  op gtreq  : Nat Nat -> Bool [prec 51] .
  op succ : Nat -> NzNat [prec 15] .
  op pred : NzNat -> Nat [prec 15] .
  var M : Nat .
  var N : Nat .
  var NN : NzNat .
  bq sd(M, N) = (abs (- M N)) .
  eq mult(N, 0) = 0 .
  bq quo(M, NN) = (truncate M NN) .
  bq rem(M, NN) = (rem M NN) .
  bq divides(NN, M) = (= 0 (rem M NN)) .
  eq less(N, 0) = false .
  eq less(0, NN) = true .
  eq lesseq(NN, 0) = false .
  eq lesseq(0, N) = true .
  eq gtr(0, N) = false .
  eq gtr(NN, 0) = true .
  eq gtreq(0, NN) = false .
  eq gtreq(N, 0) = true .
  eq succ(0) = 1 .
  bq pred(NN) = (- NN 1) .
endo

obj TUPLE2[C1 :: TRIV, C2 :: TRIV] is
  sort Tuple2 .
  op make : Elt.C1 Elt.C2 -> Tuple2 .
  op first : Tuple2 -> Elt.C1 .
  op second : Tuple2 -> Elt.C2 .
  var e1 : Elt.C1 .
  var e2 : Elt.C2 .
  eq first(make(e1, e2)) = e1 .
  eq second(make(e1, e2)) = e2 .
endo

obj TUPLE3[C1 :: TRIV, C2 :: TRIV, C3 :: TRIV] is
  sort Tuple3 .
  op make : Elt.C1 Elt.C2 Elt.C3 -> Tuple3 .
  op first : Tuple3 -> Elt.C1 .
```

```
  op second : Tuple3 -> Elt.C2 .
  op third : Tuple3 -> Elt.C3 .
  var e1 : Elt.C1 .
  var e2 : Elt.C2 .
  var e3 : Elt.C3 .
  eq first(make(e1, e2, e3)) = e1 .
  eq second(make(e1, e2, e3)) = e2 .
  eq third(make(e1, e2, e3)) = e3 .
endo

obj TUPLE4[C1 :: TRIV, C2 :: TRIV, C3 :: TRIV, C4 :: TRIV] is
  sort Tuple4 .
  op make : Elt.C1 Elt.C2 Elt.C3 Elt.C4 -> Tuple4 .
  op first : Tuple4 -> Elt.C1 .
  op second : Tuple4 -> Elt.C2 .
  op third : Tuple4 -> Elt.C3 .
  op fourth : Tuple4 -> Elt.C4 .
  var e1 : Elt.C1 .
  var e2 : Elt.C2 .
  var e3 : Elt.C3 .
  var e4 : Elt.C4 .
  eq first(make(e1, e2, e3, e4)) = e1 .
  eq second(make(e1, e2, e3, e4)) = e2 .
  eq third(make(e1, e2, e3, e4)) = e3 .
  eq fourth(make(e1, e2, e3, e4)) = e4 .
endo

obj INT is
  bsort Int
    (obj_INT$is_Int_token obj_INT$create_Int prin1
    obj_INT$is_Int) .
  bsort NzInt
    (obj_INT$is_NzInt_token obj_INT$create_NzInt prin1
    obj_INT$is_NzInt) .
  protecting NAT .
  subsorts Nat < Int .
  subsorts NzNat < NzInt < Int .
  op inverse : Int -> Int [prec 15] .
  op inverse : NzInt -> NzInt [prec 15] .
  op sum : Int Int -> Int [assoc comm idr: 0 prec 33] .
  op diff : Int Int -> Int [gather (E e) prec 33] .
  op mult : Int Int -> Int [assoc comm idr: 1 prec 31] .
  op mult : NzInt NzInt -> NzInt [assoc comm prec 31] .
  op quo : Int NzInt -> Int [gather (E e) prec 31] .
  op rem : Int NzInt -> Int [gather (E e) prec 31].
  op divides : NzInt Int -> Bool [prec 51] .
  op less : Int Int -> Bool [prec 51] .
  op lesseq : Int Int -> Bool [prec 51] .
  op gtr : Int Int -> Bool [prec 51] .
  op gtreq : Int Int -> Bool [prec 51] .
  op succ : Int -> Int [prec 15] .
```

```
 op pred : Int -> Int [prec 15] .
 vars I J : Int .
 var NJ : NzInt .
 bq inverse(I) = (- I) .
 bq sum(I,J) = (+ I J) .
*** bq I - J = (- I J) .
 eq diff(I,J) = sum(I, inverse(J)) .
 bq mult(I,J) = (* I J) .
 bq quo(I,NJ) = (truncate I NJ) .
 bq rem(I,NJ) = (rem I NJ) .
 bq divides(NJ,I) = (= 0 (rem I NJ)) .
 bq less(I,J) = (< I J) .
 bq lesseq(I,J) = (<= I J) .
 bq gtr(I,J) = (> I J) .
 bq gtreq(I,J) = (>= I J) .
 eq succ(I) = sum(1,I) .
 eq pred(I) = diff(I, 1) .
endo

obj FLOAT is
 bsort Float
   (obj_FLOAT$is_Float_token obj_FLOAT$create_Float obj_FLOAT$print_Float
   obj_FLOAT$is_Float) .
 pr BOOL .
 op inverse : Float -> Float [prec 15] .
 op sum : Float Float -> Float [assoc comm prec 33] .
 op diff : Float Float -> Float [gather (E e) prec 33] .
 op mult : Float Float -> Float [assoc comm prec 31] .
 op div : Float Float -> Float [gather (E e) prec 31] .
 op rem : Float Float -> Float [gather (E e) prec 31] .
 op exp : Float -> Float .
 op log : Float -> Float .
 op sqrt : Float -> Float .
 op abs : Float -> Float .
 op sin : Float -> Float .
 op cos : Float -> Float .
 op atan : Float -> Float .
 op pi : -> Float .
 op less : Float Float -> Bool [prec 51] .
 op lesseq : Float Float -> Bool [prec 51] .
 op gtr : Float Float -> Bool [prec 51] .
 op gtreq : Float Float -> Bool [prec 51] .
 vars X Y Z : Float .
 bq sum(X, Y) = (+ X Y) .
 bq inverse(X) = (- X) .
 bq diff(X, Y) = (- X Y) .
 bq mult(X, Y) = (* X Y) .
 bq div(X, Y) = (/ X Y) .
 bq rem(X, Y) = (rem X Y) .
 bq exp(X) = (exp X) .
 bq log(X) = (log X) .
```

```
 bq sqrt(X) = (sqrt X) .
 bq abs(X) = (abs X) .
 bq sin(X) = (sin X) .
 bq cos(X) = (cos X) .
 bq atan(X) = (atan X) .
 bq pi = pi .
 bq less(X, Y) = (< X Y) .
 bq lesseq(X, Y) = (<= X Y) .
 bq gtr(X, Y) = (> X Y) .
 bq gtreq(X, Y) = (>= X Y) .
endo

obj ID is
 bsort Id (obj_ID$is_Id_token obj_ID$create_Id obj_ID$print_Id
        obj_ID$is_Id) .
 pr BOOL .
 op less : Id Id -> Bool [prec 51] .
 var !X !Y : Id .
 *** the variable names have been chosen so that they are not Id's
 bq less(!X, !Y) = (string< !x !y) .
endo

obj QID is
 --- Quoted IDentifier
 --- symbols starting with ' character
 bsort Id (obj_QID$is_Id_token obj_QID$create_Id obj_QID$print_Id
        obj_QID$is_Id) .
endo

obj QIDL is
 protecting QID .
 pr BOOL .
 op less : Id Id -> Bool [prec 51] .
 var X Y : Id .
 bq less(X, Y) = (string< X Y) .
endo
```

G. SUPPORT FILES

The file *predef-sorts* is a list of all the predefined sorts supported by the current implementation. This file must be visible for the program to work properly. It is intended that this file will expand as predefined sorts are added to the environment.

```
Universal
Nat
Float
Bool
Int
Id
```

NzNat
Zero
NzInt

The file *predef-terms* defines term for predefined sorts, used in building a test-set. This file must be visible for the system to work properly. It is intended that this file will expand as predefined sorts and terms are added to the environment. This file is processed by a lexical analyzer called *predef-lex*. See Section C.

```
predef: Zero
constants:
numterms: 1
  name: 0
  numargs: 0

predef: Nat
constants: natconst1
numterms: 2
  name: 0
  numargs: 0
  name: succ
  numargs: 1
    name: natconst1
    numargs: 0

predef: NzNat
constants: nznatconst1
numterms: 2
  name: 1
  numargs: 0
  name: succ
  numargs: 1
    name: nznatconst1
    numargs: 0

predef: Bool
constants:
numterms: 2
  name: true
  numargs: 0
  name: false
  numargs: 0

predef: Int
constants: intconst1
numterms: 3
  name: 0
  numargs: 0
  name: succ
```

```
  numargs: 1
    name: intconst1
    numargs: 0
  name: pred
  numargs: 1
    name: intconst1
    numargs: 0

predef: NzInt
constants: nzintconst1
numterms: 2
  name: succ
  numargs: 1
    name: nzintconst1
    numargs: 0
  name: pred
  numargs: 1
    name: nzintconst1
    numargs: 0

predef: Float
constants: floatconst1 floatconst2
numterms: 2
  name: floatconst1
  numargs: 0
  name: floatconst2
  numargs: 0

predef: Tuple2
constants:
numterms: 1
  name: make
  numargs: 2
    name: !!!
    numargs: 0
    name: !!!
    numargs: 0

predef: Tuple3
constants:
numterms: 1
  name: make
  numargs: 3
    name: !!!
    numargs: 0
    name: !!!
    numargs: 0
    name: !!!
    numargs: 0

predef: Tuple4
```

```
constants:
numterms: 1
  name: make
  numargs: 4
    name: !!!
    numargs: 0
    name: !!!
    numargs: 0
    name: !!!
    numargs: 0
    name: !!!
    numargs: 0

predef: Id
constants: idconst1 idconst2
numterms: 2
  name: idconst1
  numargs: 0
  name: idconst2
  numargs: 0
```

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 052 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Dr. Luqi Code CS/Lq Department of Computer Science Naval Postgraduate School Monterey, California 93943	2
4.	Dr. Michael Nelson Code CS/Ne Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
5.	Dr. Timothy Shimeall Code CS/Sm Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
6.	Dr. Tarek Abdel-Hamid Code AS/AH Department of Administrative Sciences Naval Postgraduate School Monterey, California 93943	1
7.	Dr. Robert McGhee Code CS/Mz Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
8.	Dr. Carl Jones Code C3/Jo Department of Command & Control Naval Postgraduate School Monterey, California 93943	1

9. Capt. Robert Steigerwald 1
USAFA/DFCS
Colorado Springs, Colorado 80841

10. Sharon Rotter 1
Naval Ocean Systems Center, Code 411
San Diego, California 92152

11. LtC Mark Kindl 1
115 O'Keefe Bldg
Georgia Institute of Technology
Atlanta, Georgia 30332-0800

12. Dr. David Eichmann 1
Department of Statistics and Computer Science
Knapp Hall
West Virginia University
Morgantown, West Virginia 26506

Thesis
S68253 Steigerwald
c.1 Reusable software
component retrieval via
normalized algebraic
specifications.

3 2768 00036903 7