1993-09

Automated interface for retrieving reusable software components

Dolgoff, Scott Joel

Monterey, California. Naval Postgraduate School

Approved for public release; distribution is unlimited

AUTOMATED INTERFACE FOR RETRIEVING REUSABLE SOFTWARE COMPONENTS

by

Scott Joel Dolgoff
Captain, United States Army
B.S., Lehigh University,1983
M.B.A., Lehigh University, 1984

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

September 1993

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1993	Master's Thesis, July 1991 - September 1993

4. TITLE AND SUBTITLE
Automated Interface For Retrieving Reusable Software Components

5. FUNDING NUMBERS

6. AUTHOR(S)
Dolgoff, Scott Joel

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-500

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

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

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Computer Aided Prototyping System (CAPS) software base contains software components described by formal specifications written in the Prototype System Description Language (PSDL). One problem addressed by this thesis is to develop a retrieval mechanism for extracting components that match user-provided PSDL specifications. Another problem addressed is the integration of a retrieved component into a software prototype. The approach taken was to match specifications by comparing operations and parameter types to include indirect subtype relations. Integrating a selected software base component required generating mappings to account for different operation and parameter orderings and, for generic components, automatic instantiation. The result was a tool which implements automated assistance for finding reusable components in a large software repository. Methods were developed for parameter and operator mapping, parameter type matching, and ensuring instantiation of a generic was possible. Upon receipt of a PSDL specification query, these methods are employed to automate the retrieval of all matching components and the integration of the selected component into the software prototype. This has been fully implemented for operator components and partially implemented for type components. The retrieval mechanism, a potential processing bottleneck, is extremely accurate and reasonably fast. A query on a 1,000 component repository retrieved all 50 possible matches in under 3 minutes.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Software Reuse, Component Retrieval, Type Matching, Generic Instantiation, Syntactic Matching, Type Hierarchy, Prototype System Description Language	410
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

ABSTRACT

The Computer Aided Prototyping System (CAPS) software base contains software components described by formal specifications written in the Prototype System Description Language (PSDL). One problem addressed by this thesis is to develop a retrieval mechanism for extracting components that match user-provided PSDL specifications. Another problem addressed is the integration of a retrieved component into a software prototype.

The approach taken was to match specifications by comparing operations and parameter types to include indirect subtype relations. Integrating a selected software base component required generating mappings to account for different operation and parameter orderings and, for generic components, automatic instantiation.

The result was a tool which implements automated assistance for finding reusable components in a large software repository. Methods were developed for parameter and operator mapping, parameter type matching, and ensuring instantiation of a generic was possible. Upon receipt of a PSDL specification query, these methods are employed to automate the retrieval of all matching components and the integration of the selected component into the software prototype. This has been fully implemented for operator components and partially implemented for type components. The retrieval mechanism, a potential processing bottleneck, is extremely accurate and reasonably fast. A query on a 1,000 component repository retrieved all 50 possible matches in under 3 minutes.

I. INTRODUCTION 1

A. THE SOFTWARE CRISIS 1

B. RAPID PROTOTYPING 3

C. COMPUTER AIDED PROTOTYPING SYSTEM (CAPS) 6

II. SOFTWARE REUSE 8

A. WHAT IS REUSE 8

B. REUSE ISSUES 9

1. Design For Reuse 9

2. Maintenance Issues 9

3. Legal Concerns 9

4. Contractor Incentives 9

5. Component Retrieval 10

a. Recall 10

b. Precision 10

c. Ranking 10

C. MODELING BEHAVIOR VERSUS CLASSIFICATION 11

1. Faceted Classification 11

2. Modeling Behavior 13

3. Comparing Syntactic Behavior and Faceted Classification 14

III. CAPS SOFTWARE BASE 17

A. COMPONENT STORAGE 17

1. Relevant PSDL Attributes 17

2. Storing a Component 19

B. COMPONENT RETRIEVAL 21

1. Browsing Through the Software Base 21

2. Keyword Query 21

3. PSDL Query 22

IV. EXTENDING SYNTACTIC MATCHING 23

A. DEFINITIONS 24

1. PSDL Specification 24

2. Software Base Component 24

3. Query Component 25

4. Component Signature 25

a. Parameter Types 25

b. Input Parameters 28

c. Output Parameters 28

d. States 29

e. Abstract Data Types 29

B. SYNTACTIC MATCHING RULES 29

1. Basic Rules for Operators 30

2. Basic Rules for Types 30

3. Extended Type Matching Rules for Operators 31

4. Extended Type Matching Rules for Types 32

C. A MECHANISM FOR SYNTACTIC MATCHING 33

1. What Didn't Work - The Initial Attempt at Developing a Mechanism 33

a. Signature Calculation 35

b. Guaranteeing the Uniqueness of Signatures 36

c. Problems With the Initial Mechanism 38

2. A Successful Syntactic Matching Mechanism 40

a. Parameters as Patterns 42
b. Pattern Matching 43
c. Signature Representation 46
d. Building the Signature 48
e. Signature Matching 48
f. Limitations with Composite Types 51
g. Eliminating False Matches 52
h. Measuring Signature Closeness 57
i. An Additional Syntactic Matching Filter For Type Components 61
j. An Additional Syntactic Matching Filter for Non-Generic Components 61
k. An Additional Syntactic Matching Filter for Generic Components 63
l. A Graphic Representation of the Syntactic Matching Filtering Mechanism 65
D. IMPLEMENTATION DETAILS 67
1. Ada Language Usage Limitations 67
2. Changes Made to the Original CAPS Software Base 68
a. The Physical Schema for the Software Base 68
b. Removal of the Generic Dictionary 69
c. Addition of Operator Component Input and Output Signature Dictionaries 70
d. Addition of Type Component Input and Output Signature Dictionaries 72

3. Using PSDL to Specify Components 74
a. General Assumptions 75
b. Generic, ADT, and User Defined Type Definitions 76
c. Input and Output Parameter Type Definitions 77
d. Query by PSDL Specification 77
4. Using the PSDL Grammar With User Defined Types 77
5. CAPS Interface Requirements 78
6. Program Flow, Source Code Files, and Data Files for Signature Encoding and Type Matching 78
a. Operator Component PSDL Query 79
b. Type Component PSDL Query 80
7. The PSDL Ada Data Structure 81
8. Ordering Retrieved Components 82
a. Operator Component Match Ordering 82
b. Type Component Match Ordering 82
9. Ada/C++ Interface Issues 83
10. Complexity Issues in Matching ADT Operators, Array Components, and Validating Generic Instantiations 83
11. Encoding Input and Output Signatures 84
V. COMPONENT INTEGRATION 87
A. THE PURPOSE OF COMPONENT INTEGRATION 87
B. THE COMPONENT TRANSFORMATION PROCESS 89
1. Mapping The PSDL Specifications 90
2. Generating the Wrapper Package 93
a. Incorporating the Ada With Clause 93

b. Parameter and Type Declarations 94
c. Procedure Calls to Invoke Software Base Component Operators 95
d. Excess Output Parameters in the Software Base Component 97
e. Generic Instantiation 98
VI. GRAPHICAL USER INTERFACE 99
A. SOFTWARE BASE MAINTENANCE ROUTINE REMOVAL 99
B. OPERATOR COMPONENT INTEGRATION 99
VII. CONCLUSIONS AND FUTURE RESEARCH 103
A. DEVELOP A FORMAL MODEL FOR SOFTWARE REUSE 104
1. User Defined Types 104
2. Type Constraints 105
3. Relating PSDL to Ada 105
B. POPULATE THE SOFTWARE BASE 106
C. DEVELOP A SOFTWARE BASE MANAGEMENT GUI 107
D. EXTEND USER DEFINED TYPES TO THE SOFTWARE BASE 107
E. REDUCE THE USER INTERACTION WITH PARAMETER MAPPING THROUGH A PRE-MAPPING FUNCTION 108
F. REMOVE PSDL SPECIFICATION LIMITATIONS 108
1. Allow Extra Generic Parameters 108
2. Make Secondary ADTs Visible to External Components 109
3. Expand Array Component Definition 109
4. Matching Composite Types 109
5. Matching Constrained Types 110

G. EXTEND ARRAY MATCHING, GENERIC INSTANTIATION VALIDATION AND INTEGRATION TO TYPE COMPONENTS 111
H. IMPROVE SPEED AND EFFICIENCY 112
LIST OF REFERENCES 114
BIBLIOGRAPHY 117
APPENDIX A - USER DEFINED TYPE EXAMPLE 119
APPENDIX B - PSDL SPECIFICATION EXAMPLES 121
A. PROTOTYPE PSDL SPECIFICATION 121
B. SOFTWARE BASE COMPONENT PSDL SPECIFICATION 123
APPENDIX C - WRAPPER PACKAGE EXAMPLES 126
APPENDIX D - ADA SOURCE CODE 136
APPENDIX E - C++ SOURCE CODE 271
APPENDIX F - MODIFICATIONS TO ORIGINAL CAPS SOFTWARE BASE C++ SOURCE CODE 315
APPENDIX G - MODIFICATIONS TO ORIGINAL CAPS GRAPHICAL USER INTERFACE TAE SOURCE CODE 357
INITIAL DISTRIBUTION LIST 394

ACKNOWLEDGMENT

I would like to thank Professor Luqi for bringing me into the CAPS team and for her guidance and many excellent suggestions that greatly improved the quality of this thesis. I would also like to thank Professor Shing for his constant support and patience, and the many hours we spent discussing a variety of approaches to take with this thesis.

Additional thanks to Professor Berzins, Professor Volpano, and Professor Shimeall for their willingness to provide me with the benefit of their technical expertise along the way.

This project owes a lot to the outstanding technical support available in the Naval Postgraduate School Computer Science Department. My thanks to Frank Palazzo, CAPS Lab Manager, for his terrific technical support and friendship during my graduate studies. Thanks also to Rosalie Johnson who could always find and eradicate the gremlins that inhabited the Unix operating system from time to time. I would also like to thank Al Wong and Raquel Kerol for their support and assistance.

I. INTRODUCTION

The goal of this thesis is to develop a software reuse tool that can serve as one of the building blocks of a rapid prototyping environment. The reasons why such a tool is desirable, the concept of rapid prototyping, and an on-going rapid prototyping research project at the Naval Postgraduate School are all discussed in this section.

Section II discusses the field of software reuse to include pertinent issues, retrieval techniques, and a comparison of two principal strategies for implementing reuse.

Section III describes the software reuse model, developed during prior research, that serves as the basis for this thesis.

Section IV describes a new model for software reuse within a rapid prototyping environment and discusses both the logical extension of the original model as well as the implementation-based changes made to the original software reuse model so that an accurate trace of system evolvement can be achieved.

Section V discusses the transformation of a software component selected for reuse into a component that is compatible with the rapid prototyping tool.

Section VI discusses the graphical user interface for the software reuse tool that allows software components to be searched, examined and selected.

Section VII provides concluding remarks about the utility of the new software reuse model and tool along with recommendations for improvements and related areas for future research.

A. THE SOFTWARE CRISIS

For over a decade now the term "software crisis" has been used to define the current status of software development practices. While an ever increasing backlog of requests

for new software can be considered a part of the crisis, it is just one of the symptoms of the underlying problem. Far more important is the fact that improvements made in the areas of software quality and productivity do not come close to matching the increased complexity of today's software requirements. This is truly the essence of our software crisis.

Here is a summary of costs versus utilization of nine Department of Defense software development contracts worth $6.8 million. It is an excellent example that illustrates the extent to which today's software engineering practices fail to manage the development of complex systems: [FFN91]

- Delivered software never used successfully ($3.2 million)
- Software paid for but not delivered ($1.95 million)
- Software delivered and used, but requiring extensive rework or later abandoned because rework could not be accomplished ($1.3 million)
- Software used as delivered ($0.119 million)

Grady Booch states that "It is our human inability to deal with complexity that lies at the root of the software crisis."[Booc87] This crisis is characterized by systems today that are delivered late, over cost, with low reliability and quality, and that don't meet the requirements. Some of these symptoms are related to software development tools. The desire to develop a new language in accordance with software engineering principles was the guiding force behind DoD's development of Ada from the late 1970's to the early 1980's. Other symptoms are unrelated to technical concerns but rather identify managerial problems with current software development practices. Meanwhile, the DoD software costs continue to rise from $3 billion in the early 1970's to over $32 billion in 1990. [Booc87]

One promising area to focus research efforts in appears to be the early stages of the software development life cycle model. These stages are composed primarily of defining requirements, analyzing those requirements, and developing specifications from the resulting analysis. This is where the first attempt to manage system complexity is made and is the focal point of interaction between system users and developers. This is also the basis on which all subsequent design and implementation decisions will take place. As described by Edward Yourdon, 50% of all errors are made during the systems analysis and the cost to remove these errors account for 75% of the total error removal costs [Your89]. The reason these errors are so costly is due the ripple-through effect they have on the rest of the system. The later they are caught, the greater the chance that they will have impacted other aspects of the system that may have to be modified. Rapid prototyping is a development technique designed to find these errors at the outset of the software development process.

B. RAPID PROTOTYPING

Rapid prototyping is a development technique that attempts to alleviate the sequential rigidity of the classical waterfall method of software development. The waterfall method is a sequential process that moves forward one phase at a time.[Royc70] The phases consist of requirements definition, functional specification, design, implementation, and testing. Errors made in one phase are propagated forward. No feedback mechanism exists between phases and so problems discovered at the end require a new start back at the beginning.

Rapid prototyping attacks the inefficiencies of the sequential waterfall model by following a spiral model that allows the different life cycle phases to progress in a more parallel fashion.[Boeh87] Each phase is worked through incrementally. Initially, the

basic requirements are determined, basic specifications are developed, a rough design is created and implemented, and some minor testing is performed. The goal is not to get everything right the first time, but to be able to come up with a quick skeleton of the desired system that can be shown to the user. Here is an model of the prototyping process [Luqi89]:

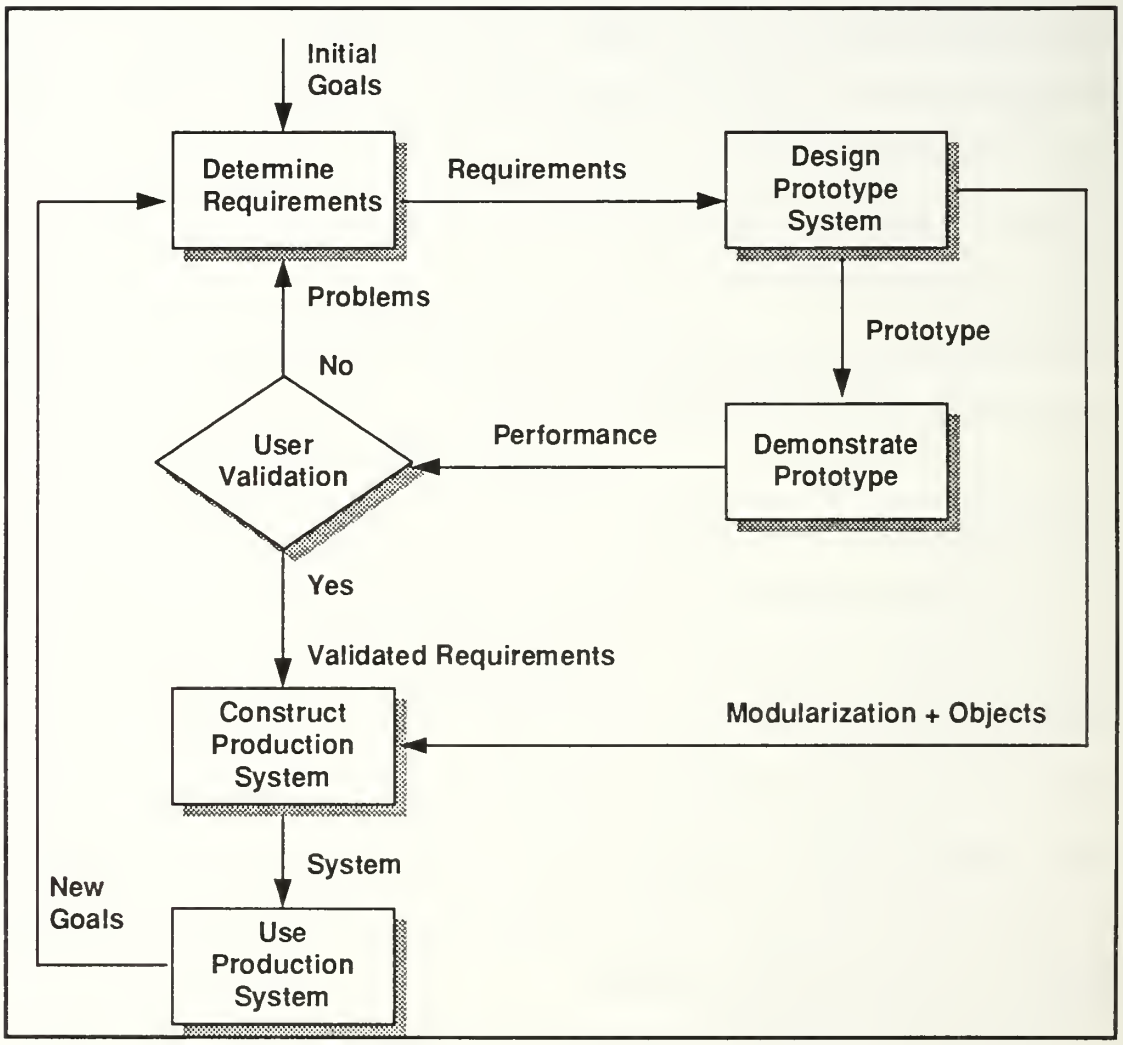

Figure 1 - The Prototype Process Model

The user looks at the "work in progress" and communicates what is right and wrong. Changes are made to requirements, specifications, design and implementation and then the next iteration of the "work in progress" is available for user examination. This is an iterative process and is designed to facilitate communication between developers and users. By maintaining the involvement of all interested parties throughout each iteration, maximum participation is achieved. Errors are caught early and ambiguities can be highlighted and resolved via group interaction. The end goal of this process is not a production system (normally). The end goal is an accurate and unambiguous set of requirements and specifications that form the basis of the subsequent production system. Because rapid prototyping addresses the potentially costly errors early in the development process, its contribution to software development can be significant.

The more functionality that can be incorporated into the prototype, the greater the likelihood that what the user is being shown is a close approximation of the desired system. However, the process of generating functionality is normally labor intensive (coding the implementation) and is in direct conflict with the prototyping goal of rapid turnaround. One solution to including more functionality into the prototype while at the same time supporting the rapid turnaround requirement is the concept of *software reuse*. A broad definition of 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."[BP89] A simple example of reusing code would be a requirement that the prototype be able to sort a list of names. Commonly the prototype development team would either write a sort routine or leave the requirement as an unimplemented stub in the prototype. If software reuse is employed, a library of software components would be accessed, a sort routine searched

for and, finally, integrated into the prototype. Thus, software reuse appears to have considerable potential as a supporting process for rapid prototyping.

C. COMPUTER AIDED PROTOTYPING SYSTEM (CAPS)

The Computer Aided Prototyping System (CAPS) is an ongoing research project in the Software Engineering Department at the Naval Postgraduate School. CAPS is used to prototype hard real-time systems. Its primary goal is to focus on the requirements, specification, and design phases of the software development life cycle. Errors and ambiguities in these phases are resolved during the many iterations of developer/customer interaction inherent in the prototyping methodology. This process provides increased productivity and reliability which in turn lead to better maintainability.

The ability to work at a very simple level is the heart of the CAPS design model. CAPS is built around the fact that all computer programs can be described or designed in terms of *operators* (functions or processes) and *types* (data streams or data structures) [LK88]. Concentrating on these two simple but fundamental components provides us with a limited set of representations which greatly aids the management of complexity in large systems. The Prototype System Description Language (PSDL) forms the basis for the CAPS computational model [LBY88]. All specifications for the prototype are written in PSDL.

As a prototyping tool, CAPS is actually a set of tools with a common user interface. These tools include a syntax-directed editor, graphics editor, design database, software base, and an execution support system [Luqi89]. This thesis focuses on the CAPS software base, a library of reusable software components. The expression "software base" will be used throughout this thesis to refer to both the physical entity that serves as a storage facility for reusable components and the abstract concept of a library of

software components that can be retrieved for reuse. The following figure provides an overview of the CAPS tools [Cumm90]]:

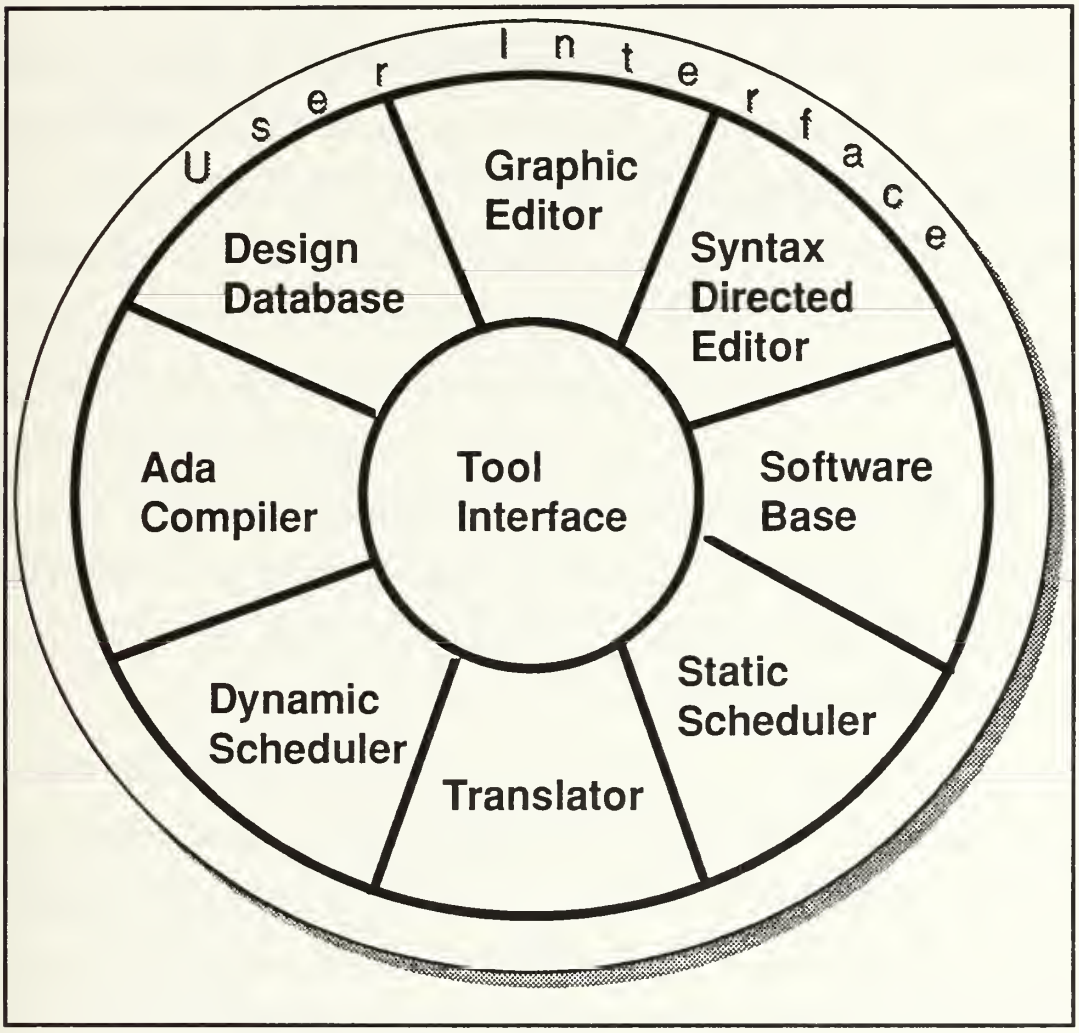

Figure 2 - CAPS Environment

II. SOFTWARE REUSE

A. WHAT IS REUSE

System source code is just one of the many deliverables upon completion of a large-scale system development effort. Requirements documents, functional specifications, architectural designs, test suites and designs, and many other forms of documentation and code are also essential components of a fully functional and maintainable system. Reuse, as defined earlier in Section I, seeks to capitalize on the knowledge and effort represented by *all* of these system deliverables. Most software under development today is not unique; that is, it has already been written in either the exact same or very similar form. However, despite the large chunks of commonality with already developed systems, new requirements are written, new specifications and designs are developed, and new code is written. The ability to reuse existing designs and code can significantly enhance productivity and reliability.

The greatest payoff appears to lie in the reuse of design level information. With the growing research into formal specification languages, automatic code generation will likely become a reality within the next decade or so thereby reducing the usefulness of reusing code. But the design process is another matter. As long as humans are involved with communicating ideas and creating requirements for systems, having an existing base of design templates to draw from will be a great enhancement to productivity. [BL91]

This thesis focuses on the reusability of code. Despite the fact that design reuse will ultimately be more important in the long run, code reuse can be of significant benefit in support of such techniques as rapid prototyping in the short run.

B. REUSE ISSUES

It is interesting that implementing software reuse is not primarily a technical challenge. While some aspects of reuse such as efficient and effective component retrieval rely on improved technological advances, other aspects are more managerial in nature. There are several issues to consider [Hoop89].

1. Design For Reuse

There is more effort and thus more time and cost involved in designing a software component so that it can be reused later. If systems are to incorporate future reusability, support must come from top management. Incentives must be built into the development process to motivate the program manager to design with reuse in mind.

2. Maintenance Issues

Regarding deliverable software to the government, some organization(s) must be responsible for maintaining a library of reusable components. Access to the library must be provided. A question to be resolved is whether the organization maintaining the library should be responsible for testing components before adding them to the library.

3. Legal Concerns

The question of who should be allowed access to government reusable component libraries must be resolved. In addition, should library access be provided as a service at a set fee? Should the government assess a charge for each retrieved component? Most importantly, who is liable if a retrieved component is responsible for a system failure? It could be the government, the original developer of the component, or even the component librarian.

4. Contractor Incentives

Arguments can be made on both sides of this issue. A contractor could lose money by retrieving reusable components from a software library because he would earn

more money from writing the code from scratch. On the other hand, it can be argued that reuse of components should be factored into the competitive bidding process for government contracts and ultimately lower the overall cost to the government. The solution probably lies in between these two extremes, but it will be up to the government to establish an environment conducive to contractor support of reuse.

5. Component Retrieval

The more components a reuse library is populated with, the greater the likelihood that a specific search will find something useful. However, there is a tradeoff. Even if a component is found that exactly meets your requirements, productivity will not have been enhanced if the search process took longer than it would have taken to simply code the component by hand. There are numerous techniques to retrieve reusable components from libraries. These include browsing, keyword searches, multi-attribute searches, syntactic matching, and semantic matching [Stei91, McDo91, Ozde92].

It is necessary for retrieval techniques to be fast and accurate. Three concepts applicable to retrieval issues are recall, precision, and ranking [WS88].

a. Recall

Recall defines the percentage of relevant components (i.e., components that match what you are searching for) that are retrieved from the set of relevant components available in the reusable component library.

b. Precision

Precision defines the percentage of retrieved components that are actually relevant.

c. Ranking

Ranking lists the retrieved components in order from best to worst match.

C. MODELING BEHAVIOR VERSUS CLASSIFICATION

It is useful to examine the methodology used today for software component retrieval and compare it against the retrieval technique this thesis employs.

1. Faceted Classification

The reuse strategy that is the most widely used and mature today is that of *faceted classification.*[PF87] Faceted classification is a methodology for software component retrieval developed by Dr. Ruben Prieto-Diaz in his 1985 dissertation and is modeled after techniques utilized in the library management sciences. The goal with faceted classification is to derive a scheme for grouping similar software components. The classification scheme is developed by defining a set of *facets*, or elemental classes, that adequately describe a software component. Examples of these facets for a software component are Functions (what the component does), Objects (what type of data the component works with), System Type (database, compiler, etc.) and Setting (business domain it will be used in). Within each of these facets are numerous elements called *terms* that define the valid member set of a facet. For example, Append, Encode, Create, and Format are all terms in the Function facet. When a software component is stored, an entry in a retrieval table is made of the most relevant term describing that component for each of the classification scheme's facets. These facets can be tailored to fit specific problem domains.

When searching for a software component, a query is made that consists of a n-tuple where n is the number of facets in the classification scheme. A term that best describes the desired component is entered into the n-tuple for each facet. Wild cards may be used that provide more flexibility in the kinds of components retrieved. When a wild card is used, no term is given for that particular facet. The retrieval mechanism then retrieves all components that have been described in the same manner. An evaluation

mechanism provides assistance during retrieval to provide components that are close to the requested description if no exact matches are found.

Customization of the classification scheme for a particular problem domain is a powerful feature of faceted classification. Domain analysis is a technique for defining reusable components and grouping them by common domain. [AP91] It is based on the premise that reusability has two basic cornerstones. First, that problems and their solutions are domain specific and, within those domains, share common attributes and environmental considerations. Second, the effort required to capture domain specific information that defines and organizes reusable items is worthwhile because we are assured of seeing the same types of problems many times in the future. In conjunction with domain analysis, faceted classification appears to be well suited for fixed domain software reuse.

The Department of Defense is heavily involved in establishing reuse libraries to boost software development quality, productivity and timeliness. Many of the more prominent libraries established to date such as the Asset Source for Software Engineering Technology (ASSET) and the Defense Software Repository System (DSRS) employ faceted classification and are attempting to utilize domain analysis to further specify their libraries. [Endo92] Commercial reuse library tools such as the Reusable Software Library and InQuisix also use a software classification approach for component retrieval [SPS93, BABKM87]. DSRS has over 2,000 components in its software library. Many of these were from the Army's Reusable Ada Products for Information Systems Development (RAPID) reuse library. Retrieving components from DSRS across the Internet in early 1993 proved to be cumbersome. The selection of facets was very slow. However, it is likely that these are interface problems that have been or will be easily cleared up. The ASSET repository has 142 resources consisting of

over 1,500 files [MM91]. Attempts to retrieve components from ASSET over the Internet in early 1993 went smoothly.

While faceted classification offers advantages over a simple keyword search mechanism due to a more rigorous and uniform component definition mandated by the classification scheme, it is still restricted to forcing the description of a component within a limited set of facets and terms. Ultimately what we would like is to be able to describe the *behavior* of a component and retrieve similar components on that basis.

2. Modeling Behavior

A modeling language is needed to define a wide class of models in a uniform manner. One critical aspect that this language would address should be component behavior. Attempts to create a behavioral model have focused on two levels. At the simpler level is *syntactic* behavior modeling that attempts to characterize a program's behavior in terms of its interface or number and types of input and output parameters [McDo91, SLB92]. The composition of input and output parameters form a "program signature" that partially characterizes the component's public behavior. That is, behavior that is made observable to the rest of the world in the program specification. The complexity of modeling syntactic behavior is greatly increased when the recognition of not only the number of input and output parameters, but their types as well, becomes a part of the behavioral model. Scanty literature is available on this subject. While the use of types as search keys for component retrieval has been described, those descriptions have been restricted to the field of functional programming [Ritt89, RT89]. Adding type information to the behavioral model introduces certain concerns specific to Ada. Ada packages whose parameters could be user defined types or even generics must be considered. In addition, subtype relations must be considered. Parameter types may not match exactly but can still logically map to an ancestor or descendant type in the Ada

type hierarchy, depending on whether you are working with input or output parameters respectively. For example, a query component input parameter of type Positive is compatible with and therefore matches its ancestor type Integer in the software base component. Thus, the process of encoding behavior into a signature that will be recognized and selected when an equivalent but inexact behavior is described has many intricacies.

The second and more complex level of behavioral modeling is *semantic* behavioral modeling. This is an attempt to characterize the implementation of the program in addition to its interface. A dissertation by Robert Steigerwald discusses semantic matching [Stei91]. The two different forms of behavioral modeling are most effective when used in combination. The simpler, and more importantly, *faster* syntactic matching serves as an initial filter eliminating all components that are not compatible with the specified input/output behavior. The subsequent reduced set of components is then passed on to the semantic matching algorithm which attempts to match and rank the correct component(s). If multiple components remain, the designer can browse through them to determine which, if any, will best satisfy his or her needs.

3. Comparing Syntactic Behavior and Faceted Classification

After examining two methods for component retrieval, faceted classification and behavioral modeling, the natural question is which technique is better. While concerns about efficiency are always important, the issues of precision and recall are even more so and in that context the two methodologies can be contrasted. We will use syntactic matching as the tool with which to consider behavioral modeling.

The appeal of faceted classification is that it casts a fairly wide net and is certain to retrieve components that are at least conceptually similar to the component being searched for. Because numerous facets can be employed, the component library is

substantially reduced on the first pass and may be judiciously reduced by subsequent narrowing of scope (few or no wild cards). Faceted classification also provides tools for measuring and defining "closeness" so that a search can be expanded if the qualifying facet terms prove too narrow [PF87]. Looking at the other alternative, the issue of "casting a wide net" is an area in which syntactic matching has to be very careful. It is at its best when retrieving an exact match; that is the input/output parameters of the retrieved component(s) exactly match the input/output parameters of the query component. However, as most reuse literature suggests, the predominant results of component retrieval will be inexact matches with necessary modifications subsequent to the retrieval. In this more likely scenario, syntactic matching looking for exact matches is very unforgiving and will likely exclude valuable components that match closely but not exactly with the query component. Therefore the design of the syntactic matching algorithm must be somewhat forgiving if it is to be effective. Despite these concerns, syntactic matching has benefits not found in faceted classification. One advantage of syntactic matching is that because it more closely models behavior, it is possible to locate reuse components whose behavior is analogous to the query component yet may not fall into the general conceptual/categorical context of the query component [MS92]. This is a case where a valid component would not have been retrieved through faceted classification. Because prototyping can often be used to explore new domains that do not have well developed domain models, cross-domain reuse is important in such a context. So in the category of *recall*, syntactic behavior retrieves more of the existing relevant components.

Neither faceted classification nor syntactic matching can do much to improve *precision,* the percentage of retrieved components that are actually relevant. However, syntactic matching has a big advantage here as well because it is only the first filter in the

behavioral model designed to weed out components that do not match and then pass the remaining candidates on to a semantic filter which, by its very nature, will improve the final retrieval *precision*. Thus, while behavior is harder to model, it is inherently more powerful.

This thesis builds on prior research to utilize the robust behavioral model for retrieving reuse components, providing important advantages over the reuse technology dominating the field today [McDo91]. These advantages are in the following areas:

- **component retrieval recall**
- **component retrieval precision**

High recall is ensured by the syntactic matching techniques described in this thesis. A high level of precision is realized through subsequent semantic filtering [Stei91].

III. CAPS SOFTWARE BASE

Initial implementation of the CAPS software base was first explored in a thesis by Daniel Galik [Gali88]. Actual implementation of the software base was accomplished by John McDowell [McDo91]. McDowell's implementation uses ONTOS, an object oriented data base management system that provides an interface to C++ for customization and flexibility. [Onto91] Good descriptions of the CAPS software base are provided by both McDowell and Ozdemir [McDo91, Ozde92]. As a repository for reusable software components, the CAPS software base supports two critical functions; component storage and component retrieval.

A. COMPONENT STORAGE

Each software component to be stored in the CAPS software base must have a Prototype System Description Language (PSDL) file.[LBY88] The PSDL file provides information about the software component that is used to determine how the component is stored. Although a PSDL file contains information about a variety of attributes of a real-time software component, we will only be concerned with the attributes that relate directly to the component retrieval mechanism developed by McDowell and extended in this thesis.

1. Relevant PSDL Attributes

A software component is stored on the basis of its signature or interface to the outside world. The PSDL attributes that comprise this interface include the generic, input, and output parameters of the component as well as whether or not it is a state machine. The role of the component, *type* (abstract data type) or *operator* (procedure), is also considered as part of the interface for storage purposes. All these PSDL

attributes can be combined to form a *multi-attribute key* that uniquely identifies all components that share a given set of attribute values [SLM91]. The following simple example shows an Ada package specification and its corresponding PSDL specification file:

Ada Package Specification:

```
package Example_Pkg  is
   procedure Example (Num1  : in       Integer;
                      Num2  : in       Integer;
                      Result :    out Integer);
end Example_Pkg;
```

PSDL Specification:

```
OPERATOR Example
SPECIFICATION
   INPUT   Num1 : Integer,
           Num2 : Integer
   OUTPUT  Result : Integer

   KEYWORDS  example, add
END
```

Figure 3 - Ada Package Specification with Corresponding PSDL Specification

Note that the keywords attribute is also listed in Figure 3. Keywords are used in one form of component retrieval and will be described shortly.

2. Storing a Component

The first determination of how a component is stored is dependent on whether the component is a *type* or an *operator*. References to *types* are stored in an ONTOS class called the SB_ADT_Component_Library and references to *operators* are stored in the SB_Operator_Component_Library. To illustrate the component storage process, we will concentrate on the storage of *operators*. Storage of both *types* and *operators* is described in detail by McDowell [McDo91].

The principal storage structure for all components is the ONTOS Dictionary class. A Dictionary is an object that stores key-element data pairs. The key can be used to order and retrieve the data in the dictionary. The element normally represents the specific data entity to be stored. Because ONTOS is built around an object oriented environment, the element is often an object. Figure 4 below provides a conceptual representation for how *operators* are stored in the CAPS software base. It has been reduced from its actual scope to simplify the explanation [McDo91]:

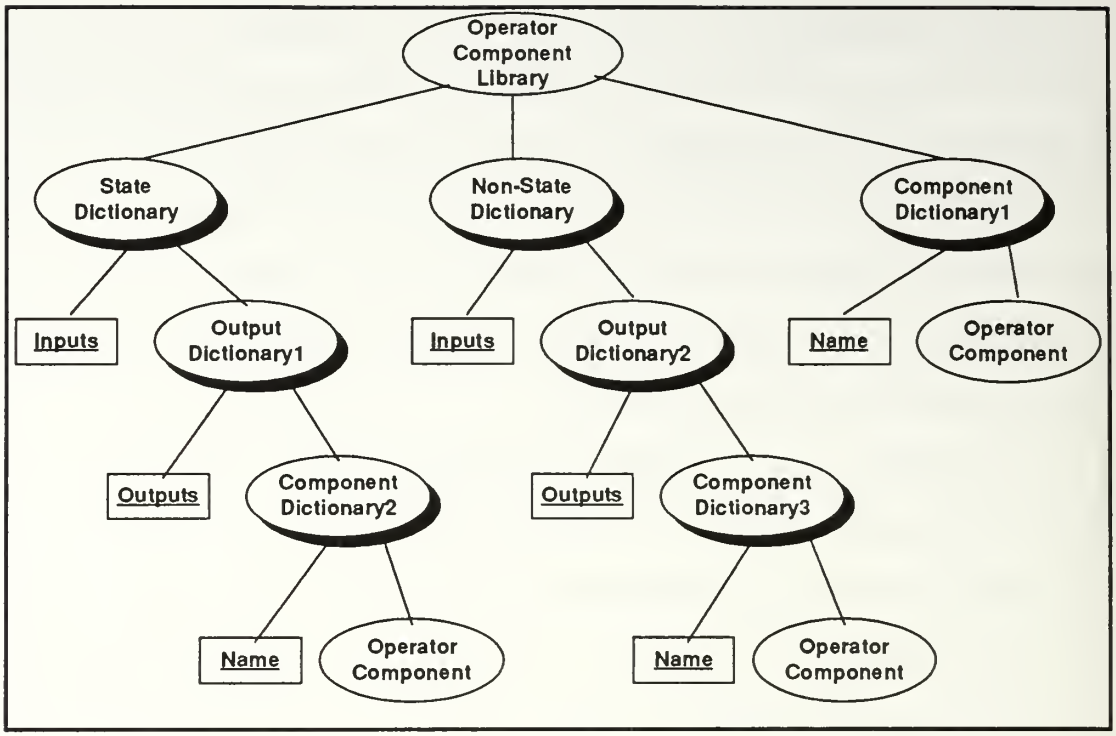

Figure 4 - Operator Component Library

Regular ellipses represent objects. Shadowed ellipses represent Dictionary objects. Rectangles represent Dictionary keys. Figure 4 depicts a storage hierarchy of objects. All objects are only stored once in a distinct physical location. References to those objects can then be stored within other objects. That is the role served by the above dictionaries. Their elements are actually references to other objects. When the *operator* software component Example from Figure 3 is stored in the CAPS software base, its PSDL specification is first parsed for information that will be stored to help aid subsequent retrieval requests. Because it is an *operator*, we start in the Operator_Component_Library. A reference to Example is placed in Component Dictionary1. This is used to provide quick access to a list of all the software base *operator* components which can then be used to browse through the software base. Next, because there is no state information for Example we move to the Non-State

Dictionary. Example has two input parameters, so we now move to the Output Dictionary2 object that corresponds to the key value 2 for the Non-State Dictionary. Example has one output parameter. In the Output Dictionary2 element, Component Dictionary3, that corresponds to a key value of 1, we store a reference to the software component Example. Thus, with the information about its state, number of input parameters, and number of output parameters, a reference to Example has been stored in a complex data hierarchy that will allow us to retrieve it by submitting the correct information in the form of a multi-attribute key [SLM91].

B. COMPONENT RETRIEVAL

Retrieving software components is an important part of the reuse process. There are currently three methods available to perform component retrieval from the CAPS software base. They are browsing, keyword query, and PSDL query.

1. Browsing Through the Software Base

Browsing through the CAPS software base is a very simple process. The user selects either the *type* or *operator* domain, and then a listing of all available components in the chosen domain is displayed. The user can skim through that list and select individual components for more detailed examination.

2. Keyword Query

For this retrieval mechanism, the user selects one or more keywords from a list of all keywords currently used in software components in the CAPS software base. An ONTOS Dictionary object called Keyword_Dictionary is then scanned. Its keys are the keywords and its elements are Component_Dictionary objects similar to the ones described in Figure 4 above. Each keyword maps to a Dictionary object that contains references to all components with that particular keyword in their PSDL specification.

When the scan is complete, the user is presented with all components that have at least one of the user's selected keywords. The components are listed in descending order with the components that have matched the most user keywords listed at the top.

3. PSDL Query

A PSDL query requires the user to provide a PSDL specification as the query. This specification is then compared against the PSDL specifications of the components in the CAPS software base to see if any are a valid match. Rather than having to compare the query against each software base component, the hierarchical methodology for storing information about the software base components serves as a very efficient filter for retrieving only the matching components [McDo91]. Using Figure 4 as an example, suppose we have a PSDL query specification that has two input parameters, two output parameters and no states. The retrieval mechanism, working from the information parsed from the PSDL query, immediately eliminates all components stored within the State_Dictionary. Next, the dictionary of components with two input parameters is extracted, eliminating all components with less than or more than two input parameters. Finally, a set of Component_Dictionary3 dictionaries is extracted. Each Component_Dictionary3 in the set contains all components with two (or three, or four, etc.) output parameters (and by the path of extraction, two input parameters and no state variables). This group of components is then presented to the user as all the valid matches for the particular PSDL query. The reason we might have more than one Component_Dictionary3 extracted is that a software base component with a greater or equal number of output parameters than a query component is considered to match the query component as long as both components have the same state and same number of input parameters.

IV. EXTENDING SYNTACTIC MATCHING

A PSDL specification defines the public view or interface of a software component while the actual implementation remains hidden. *Syntactic matching* is the process of comparing a query component's PSDL specification with a software base component's PSDL specification to determine if their interfaces are similar. Any component that satisfies a given specification must have a compatible interface, so we can quickly exclude from consideration all the components that do not meet this criterion. The syntactic matching process used in this thesis is an extension of the process used in a thesis by John McDowell.[McDo91] McDowell developed a theoretical formalization of syntactic matching that includes the component input and output parameter types. He also implemented a matrix scheme that provided a storage structure to address the matching of subtypes. However, McDowell's implemented syntactic matching process did not include matching parameter types. In this thesis *we extend McDowell's theoretical formalization of syntactic matching to include subtypes.* And where McDowell's implementation used multi-attribute keys that were limited to matching components based on attributes such as numbers of generic, input, and output parameters or number of operators (within an abstract data type) of a PSDL specification, *here we extend McDowell's implementation by including the type of the parameters in the match procedure.*

The goal of syntactic matching is to provide a fast method for selecting a subset of components from the software base that have a likelihood of providing the behavior required by the query component. This retrieved subset of candidate solutions to the query will then be passed to a semantic matching process. Semantic matching explores both the external and internal behavior of a component utilizing normalized algebraic

specifications [Stei91]. Thus the critical role syntactic matching plays is to shrink the list of candidate components in a very quick manner which can then be examined via the rigorous and consequently much slower semantic matching process. The power of syntactic matching is its ability to execute the match process rapidly with a high degree of *recall*. Semantic matching, on the other hand, matches much more slowly but with high *precision*.[SLB92]

The following guidelines describe aspects of the syntactic matching process for both PSDL *operators* and *types*. The matching process takes a PSDL specification supplied by the CAPS prototype designer and retrieves all software components from the software base that have a similarly defined PSDL specification. The expression "similarly defined" will be described formally along with the description of the matching process below.

A. DEFINITIONS

Before describing the syntactic matching process it is necessary to define all the terms and notation used in the process.

1. PSDL Specification

The PSDL specification for a component is denoted by **PS**.

2. Software Base Component

The software base component is denoted by **sbc**. The PSDL specification of a reusable software component stored in the software base would therefore by denoted by **PS(sbc)**.

3. Query Component

A query component refers to the component that the CAPS prototype designer is in the process of finding a software base match of and is denoted by **qc**. The PSDL specification for that query component would therefore be denoted by **PS(qc)**.

4. Component Signature

The component signature refers to the types of the component parameters. There is a separate signature for input and output parameters. A signature is encoded with information that describes each instance of all Ada types used by a component. For example, if an *operator* component has two input parameters of type Integer, an input parameter of type Boolean, and an input parameter of type Range, then the input signature for that *operator* would be encoded with two instances of Integer, one instance of Boolean, and one instance of Range. Signatures for *types* are treated a little differently and reflect the parameter type information contained in the aggregation of all the abstract data type's *operators*. So, for example, let us take a *type* that has two *operators,* type_operator1 and type_operator2. Type_operator1 has one Boolean input parameter and one Integer input parameter. Type_operator2 has one Integer input parameter. The input signature for this *type* would therefore be encoded with two instances of Integer and one instance of Boolean, reflecting the aggregation of the input parameters of the abstract data type's *operators*.

a. Parameter Types

Parameter types are easy to match if, for example, an input **PS(qc)** parameter is exactly the same type as an input **PS(sbc)** parameter. However, just because the parameter types do not match exactly does not mean they do not match. Because Ada employs a well-defined type hierarchy that is based on inheritance and subtyping, in some cases parameters of what appear to be different types can be matched. [Booc87] It is

also possible to include a parameter type in a specification that is not predefined by Ada. This is called a *user defined type* (UDT). A UDT must be defined as a *type* in the specification of the prototype if it is to be referenced by another *type* or *operator*. The UDT specification provides a critical link in the parameter type matching process by including a reference to the Ada type that defines the UDT. An example of this can be examined in Appendix A.

The types Private, Discrete, Array, Digits, Delta, Range, and Access can appear in specifications of generic parameters for generic Ada components. [Ada83] The types Private, Discrete, Integer, Range, Natural, Positive, Enumeration, Character, Boolean, Access, Record, Array, String, Digits, Float, Delta, and Fixed are predefined types in Ada. The following figure depicts a partial ordering of the Ada type hierarchy:

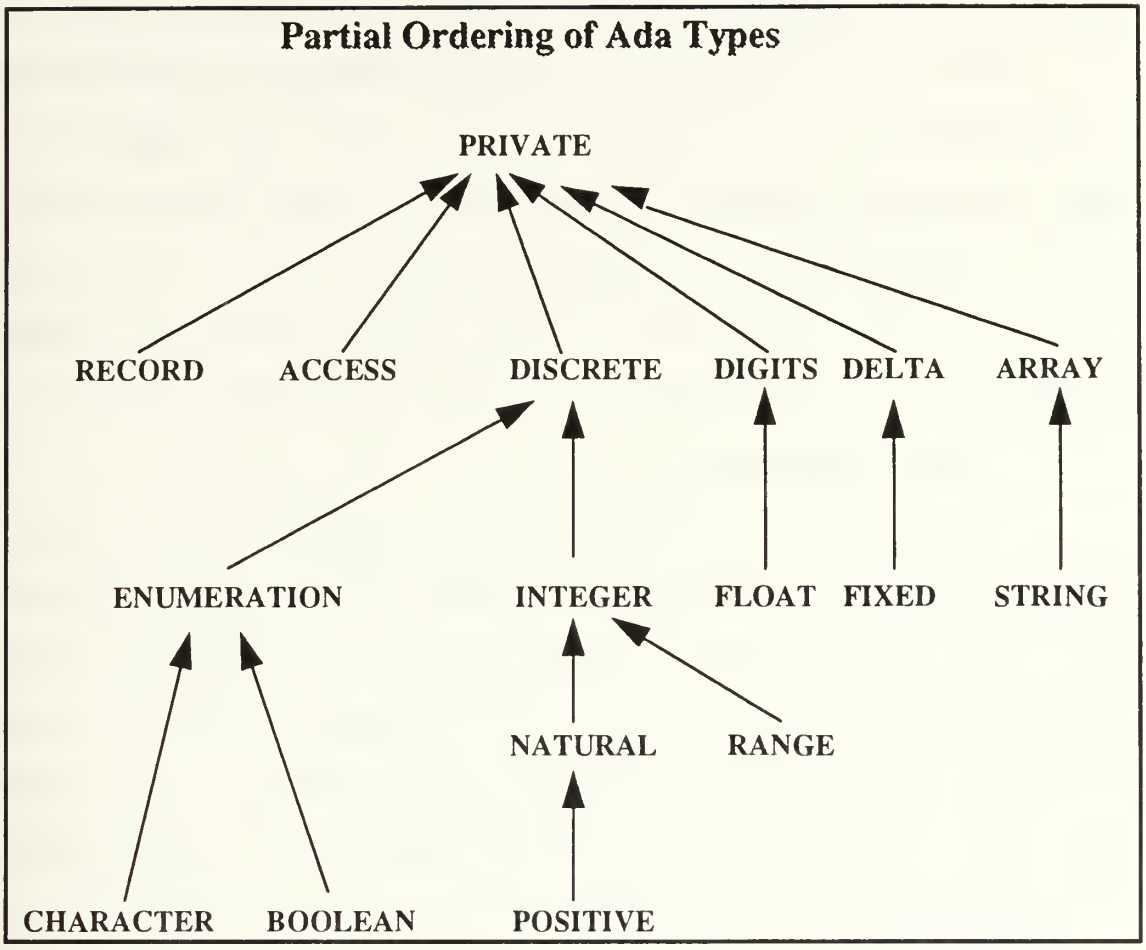

Figure 5 - Ada Subtype Hierarchy

In Figure 5 we see that, for example, an input **PS(qc)** parameter of type POSITIVE could be matched to an input **PS(sbc)** parameter of type INTEGER. It is important to note that this mapping is only allowed in one direction. The direction is dependent on whether you are working with input or output parameters. We could not take an input **PS(qc)** parameter of type INTEGER and match it to an input **PS(sbc)** parameter of type POSITIVE. This is because every input value that is consistent with the query specification must be a legal input of the software base component and every output value that can be produced by the software base must be consistent with the query

specification. Again, note that the required subtype relations go in opposite directions for input and output parameters. The type hierarchy follows a partial ordering scheme. As a partial ordering, this hierarchy is by definition reflexive, anti-symmetric, and transitive [Epp90]. This type hierarchy takes into account all type names that the **PS(qc)** can expect to reference. To illustrate type relationships within the hierarchy, suppose we have a *parameter type* **T** and another parameter type **t** such that **t** is a *subtype* of **T**. For example, Natural is a subtype of Integer. By definition of a subtype, **t** must be a descendant of a **T** with respect to Figure 5, or **t** must equal **T** [Booc87].

b. Input Parameters

Each input parameter has an identifier name and a corresponding type. The identifier name is represented by **p**. All identifiers in a PSDL specification must have unique names. The expression **input_type(p, sbc)** refers to the *parameter type* for a given input parameter **p** in the component **sbc**. And **input_type(p, qc)** refers to the *parameter type* for a given input parameter **p** in the component **qc**. The expression **In(sbc)** refers to the entire set of input parameter identifier names for a given software base component, and **In(qc)** refers to the entire set of input parameter identifier names for a given query component. As an example, suppose we have a query component with two Integer input parameters p1 and p2, and one Boolean input parameter p3. For **p** equal to p1 or p2, **input_type(p, qc)** would return Integer. For **p** equal to p3, **input_type(p, qc)** would return Boolean. And **In(qc)** would be the set {p1, p2, p3}.

c. Output Parameters

The definitions for the output parameters are very similar to the input parameters. The expression **output_type(p, sbc)** refers to the *parameter type* for a given output parameter **p** in the component **sbc**. And **output_type(p, qc)** refers to the *parameter type* for a given output parameter **p** in the component **qc**. The expression

Out(sbc) refers to the entire set of output parameter identifier names for a given software base component, and **Out(qc)** refers to the entire set of output parameter identifier names for a given query component.

d. States

The expression **ST(sbc)** is a boolean function that evaluates whether the software base component is a state machine or not. **ST(qc)** performs the identical function for the query component.

e. Abstract Data Types

Abstract data types (*types)* have certain definitions that *operators* do not. **ADT(sbc)** denotes the set of all abstract data types in a *type* software base component and **ADT(qc)** denotes the set of all abstract data types in a *type* query component. An abstract data type within a *type* component is a reference to a distinct type name that appears in the type declaration part of the *type* component's PSDL specification. **OPS(sbc)** denotes the set of all ADT *operators* in a *type* software base component and **OPS(qc)** denotes the set of all ADT *operators* in a *type* query component. Because we are dealing with aggregates, the expression **Tot_In(sbc)** refers to the entire set of input parameter identifier names over all operators of a *type* software base component and **Tot_In(qc)** refers to the entire set of input parameter identifier names over all operators of a *type* query component. **Tot_Out(sbc)** and **Tot_Out(qc)** are defined in the same manner but for output parameters.

B. SYNTACTIC MATCHING RULES

The initial set of syntactic matching rules with one modification are taken from McDowell's thesis and they serve as the basic rule set [McDo91]. The modification pertains to McDowell's treatment of user defined types. McDowell called user defined

types *unrecognized types* and created a rule between software base component generic parameters and query component unrecognized type parameters. However, this thesis does not permit unrecognized types and forces all query component parameters to be defined as Ada types. That is not to say that a query component parameter cannot be defined as a user defined type. It can. But ultimately by transitivity, a referenced user defined type must be defined in terms of an Ada type (see Appendix A).

The basic set of rules are used as the first filter in the syntactic matching process. These rules differ slightly between *operators* and *types*. This thesis augments those rules by including rules specific to parameter type matching. **NUM(X)** is defined as a function that returns the cardinality of the set represented by **X**.

1. Basic Rules for Operators

Basic rules for operators are primarily concerned with comparing number of parameters and are listed as follows [McDo91]:

- **NUM(In(sbc))** = **NUM(In(qc))**
- **NUM(Out(sbc))** ≥ **NUM(Out(qc))**
- **ST(sbc)** = **ST(qc)**

The number of software base component input parameters must equal those of the query component. The number of software base output parameters must be equal to or greater than those of the query components. And both components must either be state machines or not be state machines.

2. Basic Rules for *Types*

A PSDL *type* consists of one or more abstract data types (ADT) and zero or more operators (OPS). When initially matching PSDL *types* it is useful to aggregate operator input and output parameters. These aggregate values can be used to compare

two components. For example, a *type* component with three operators that each have two input parameters would have an aggregate input parameter value of six (three operators times two input parameters each). If the aggregate values do not meet the inequalities listed below then we can exclude the candidate component from any further consideration. If the aggregate signatures do match we have not confirmed a match, but must continue the matching process using more sophisticated filters. The basic rules for matching *types* are as follows [McDo91]:

- **NUM(ADT(sbc))** $\geq$ **NUM(ADT(qc))**
- **NUM(Tot_In(sbc))** $\geq$ **NUM(Tot_In(qc))**
- **NUM(Tot_Out(sbc))** $\geq$ **NUM(Tot_Out(qc))**
- **NUM(OPS(sbc))** $\geq$ **NUM(OPS(qc))**

The number of ADTs, operators, aggregate operator input and aggregate operator output parameters of the software base component must all be either greater than or equal to those of the query component.

3. Extended Type Matching Rules for Operators

The components that pass the basic rules are checked against the extended matching rules, which are more restrictive. The extended matching process includes comparison of parameter types between the query and software base components. The extended rules for matching *operators* are as follows:

- Property 1

  $\exists f : \mathbf{In(qc)} \rightarrow \mathbf{In(sbc)}$ such that

  [f is bijective $\wedge$

  $\forall \mathbf{p} \in \mathbf{In(qc)}$

  [**input_type(p, qc)** is_a_subtype_of **input_type**(f(**p**), **sbc**)]]

- Property 2

  $\exists f : \mathbf{Out(qc)} \rightarrow \mathbf{Out(sbc)}$ such that

  [f is one-to-one $\wedge$

  $\forall \mathbf{p} \in \mathbf{Out(qc)}$

  [**output_type(p, qc)** is_a_supertype_of **output_type**(f(**p**), **sbc**)]]

The first rule says that a) each input parameter of the query component must map to a distinct input parameter in the software base component and vice versa, and b) for each input parameter pair, the type of the query component input parameter must be equal to or a subtype of the software base input parameter. *Distinct* is defined to mean that no parameter in the function's range can be mapped to by more than one parameter in the function's domain. The second rule says that a) each output parameter of the query component must map to a distinct output parameter in the software base component, and b) for each output parameter pair, the type of the query component output parameter must be equal to or a supertype of the software base output parameter. The second property is not required to be *onto* because a software base component *operator* can have more output parameters than the query component.

4. Extended Type Matching Rules for *Types*

Let $\mathbf{OP_{qc}}$ denote a query *type* component operator and $\mathbf{OP_{sbc}}$ denote a software base *type* component operator. The extended rule for matching *types* is as follows:

- $\exists f : \mathbf{OP_{qc}} \rightarrow \mathbf{OP_{sbc}}$ such that

  [f is one-to-one $\wedge$

  $\forall \mathbf{OP_{qc}} \in \mathbf{OPS(qc)}$

  [Property 1 and Property 2 defined in Section IV.B.3

  hold true for each ($\mathbf{OP_{qc}}$, f($\mathbf{OPq_c}$)) pair.]]

This rule states that a) each operator in the query *type* component must map to a distinct operator in the software base *type* component, and b) each mapped operator pair must adhere to the rules defined in Sections IV.B.1 and IV.B.3.

C. A MECHANISM FOR SYNTACTIC MATCHING

Now that we have described the rules for what constitutes a valid match, we need a mechanism for calculating matches efficiently. Because a handwritten signature uniquely identifies an individual, the term *signature* has become synonymous in many domains with the concept of identification. A mechanism for generating signatures that represent a software component's parameter composition provides a utility that subsequently allows component matching to take place by component signature comparison.

1. What Didn't Work - The Initial Attempt at Developing a Mechanism

Prime numbers were explored as the basis for computing syntactic signatures that identify unique groupings of parameters. Each PSDL specification has a separate signature for its input and output parameters. The goal was to match PSDL specifications by comparing their signatures.

The following signature matching mechanism was developed. All types in the Ada type hierarchy of Figure 5 were assigned a *unique prime number* and a *representative value*. The root type has a unique prime number which also serves as its

representative value. The representative value for each descendant type is derived by multiplying the representative values of its parent type by its own unique prime number. So, for example, let us assign the following unique prime numbers to types: Private = 2 (root type), Discrete = 3, and Enumeration = 7. Looking at the Ada type hierarchy in Figure 5 we see that Private is a root type and therefore its representative value 2, equals its unique prime number. The representative value for Discrete is found by multiplying its unique prime number (3) by the representative value of its parent (Private = 2) which results in 6 (3×2). Likewise, the representative value for Enumeration is found by multiplying its unique prime number (5) by the representative value of its parent (Discrete = 6) which results in 30 (5×6). The unique prime numbers assigned to types recognized by the matching algorithm and their corresponding representative values are as follows:

					Unique Prime No.	Representative Value
private					2	2
	discrete				3	6
		enumeration			7	42
			character		17	714
			boolean		19	798
		integer			5	30
			range		23	690
			natural		11	330
				positive	13	4,290
	digit				37	74
		float			43	3,182
	delta				41	82
		fixed			47	3,854
	array				29	58
		string			53	3074
	record				31	62
	access				59	118

Figure 6 - Type Prime Number Hierarchy

a. Signature Calculation

A PSDL specification has separate signatures for its input and output parameters. A parameter signature is calculated by multiplying the representative values given above for all parameter types. For example, a PSDL specification with input parameters of type *integer* (30) and type *boolean* (798) would have an input parameter

signature of 23,940 (30×798). Multiple occurrences of a type cause its representative value to be used a corresponding number of times in the signature calculation.

b. Guaranteeing the Uniqueness of Signatures

The effect of this numbering scheme is to simplify the comparison of two parameter signatures. Using a unique integer value to represent a parameter signature allows for fast lookup in an ordered list in the simplest case of an exact match. In the more complicated case, a number of potential matches might not be exact but are guaranteed to apply due to the partial ordering of types. Therefore, a query component trying to match its input parameters of type *integer* and *boolean* can successfully map to a software base component with an input parameter X that is an ancestor of *integer* in the partially ordered Ada type hierarchy and an input parameter Y that is similarly an ancestor of *boolean*. If one component parameter signature can divide into another component parameter signature with no remainder, then a signature match has occurred (note that the determination of which component's signature is the dividend and which is the divisor is dependent on whether input parameter signatures or output parameter signatures are being matched). Informally, this is because the type hierarchy numbering scheme insures that the representative value of any descendant is a multiple of the representative values of all its ancestors. The value of a signature is unique. No other combination of parameters could result in the same value. To see why this is so, let us consider an example. Here are the input parameters for an *operator:*

Parameter Type	Rep. Value	Prime Factors
• Character	714	{17, 7, 3, 2}
• Integer	30	{5, 3, 2}
• Natural	330	{11, 5, 3, 2}

The input signature for this example would be 7,068,600 ($714 \times 30 \times 330$). What makes its value unique is the fact that the representational values for the parameter types are the result of prime factors, and each representational value has its own unique combination of prime factors. Thus each signature is also a unique combination of prime numbers. Only the exact same combination can represent a signature with the exact same parameters. The prime factors comprising a representational value have been deliberately chosen to provide the capability to perform subtype matching. For example, Integer is an ancestor of Natural. It is clear that dividing the representational value of Integer into the representational value of Natural would result in a zero remainder. That is because Natural has the same set of prime factors as Integer, in addition to the prime factor 11. This relationship was designed to hold true for all ancestors and their descendants in the Ada type hierarchy. To further illustrate this, let us divide input_signature1 into input_signature2. As long as the parameter types in input_signature1 can be mapped by a one-to-one correspondence with the parameter types of input_signature2 so that all input_signature1 parameter types are the same or an ancestor of the input_signature2 parameter type they are mapped to, then the result of the division must be a zero remainder. Any mapping that did not maintain an ancestor-descendant or equal-equal relationship between parameter types would guarantee that a set of prime factors (representing one parameter type) would be divided by a non-subset set of prime factors (representing the other parameter type in the mapping) resulting in a non-zero remainder. Another point to make with this prime factorization scheme is that, given an input signature, it is easy to derive the parameter types that make it up. Starting with the largest representational values and working down to the smallest, the representational value is divided into the signature. If the remainder is zero, then the new signature value becomes the result of the division and we have successfully

extracted one parameter type. We repeat this process with the same representational value until we get a non-zero remainder. If the remainder is non-zero then the value of the signature is not changed and we move to the next largest representational value and start the division process over again. This is continued until the value of the signature is zero which will happen once all parameter types are extracted. This process of deriving the parameter types that make up the signature is identical to the retrieval phase of the *knapsack algorithm* [Manb89].

Two factors aid the search process in a list of components ordered by parameter signature value for a match even when an exact parameter signature match is unavailable. First, all software base components with an input parameter signature greater than the query component input parameter signature can be skipped because the query component input parameter signature cannot be divided into by any of them with a remainder of zero. Second, all software base components that have a parameter signature less than or equal to the query component's parameter signature can quickly be checked by dividing the software base component signature into the query component signature to see if the result is a zero remainder (for input parameters). All results with a zero remainder indicate a match.

c. Problems With the Initial Mechanism

While outwardly fast and effective, this initial strategy for syntactic matching has several shortcomings. Array types, which are composite, are made up of two components; an element and an index. This necessitated separate and unique numbering schemes for the array index and array element in order to distinguish the array components from stand alone parameter types. For example, using the representational value of 30 for an Integer whether that Integer is a stand alone parameter type or the index component of an Array type cannot be allowed. Otherwise, dividing a signature

with one parameter of type Integer (signature = 30) into a signature with one parameter of type Array [element : Record, index : Integer] (signature = 107,880 = 58 × 62 × 30) would result in a zero remainder, falsely indicating that the signatures matched.

Trying to match software base components that had more input parameters than the query component (only allowable in the case of aggregate input signatures for PSDL *types*) necessitated another separate and unique numbering scheme similar to Figure 6 but reversed with the root representational value taking on the product of its children's representational values and a unique prime. This forced the representational values for ancestors to be greater than those of descendants. The reason for this is that it is important that the signature being divided into have the larger value and therefore be the one allowed to have extra parameters. Since it is always either an equal or larger value than that of the signature dividing into it, increasing its value by multiplying it by additional parameter representational values does not affect the outcome of the division (i.e., a zero or non-zero remainder). For example, if Signature1 with one Discrete parameter (signature = 6) is dividing into Signature2 with one Integer parameter (signature = 30), then we can arbitrarily add a Character parameter to Signature2 (new signature = 21,420 = 30 × 714) without affecting the fact that Signature2 divided by Signature1 will result in a zero remainder.

Finally, the greatest shortcoming of this initial mechanism proved to be technical. The numeric values of calculated signatures were simply too large to be handled by a computer dealing with 64 bit floating point values without losing precision. The maximum available precision of 15 digits was far too small and too limiting for the needs of this mechanism.

2. A Successful Syntactic Matching Mechanism

The mechanism developed in this thesis for syntactic matching that overcame the above problems is loosely based on the concepts of pattern recognition and set theory [Epp90, Manb89]. First, let us picture the subtype hierarchy of Figure 5 as a hierarchy of regions portrayed as follows:

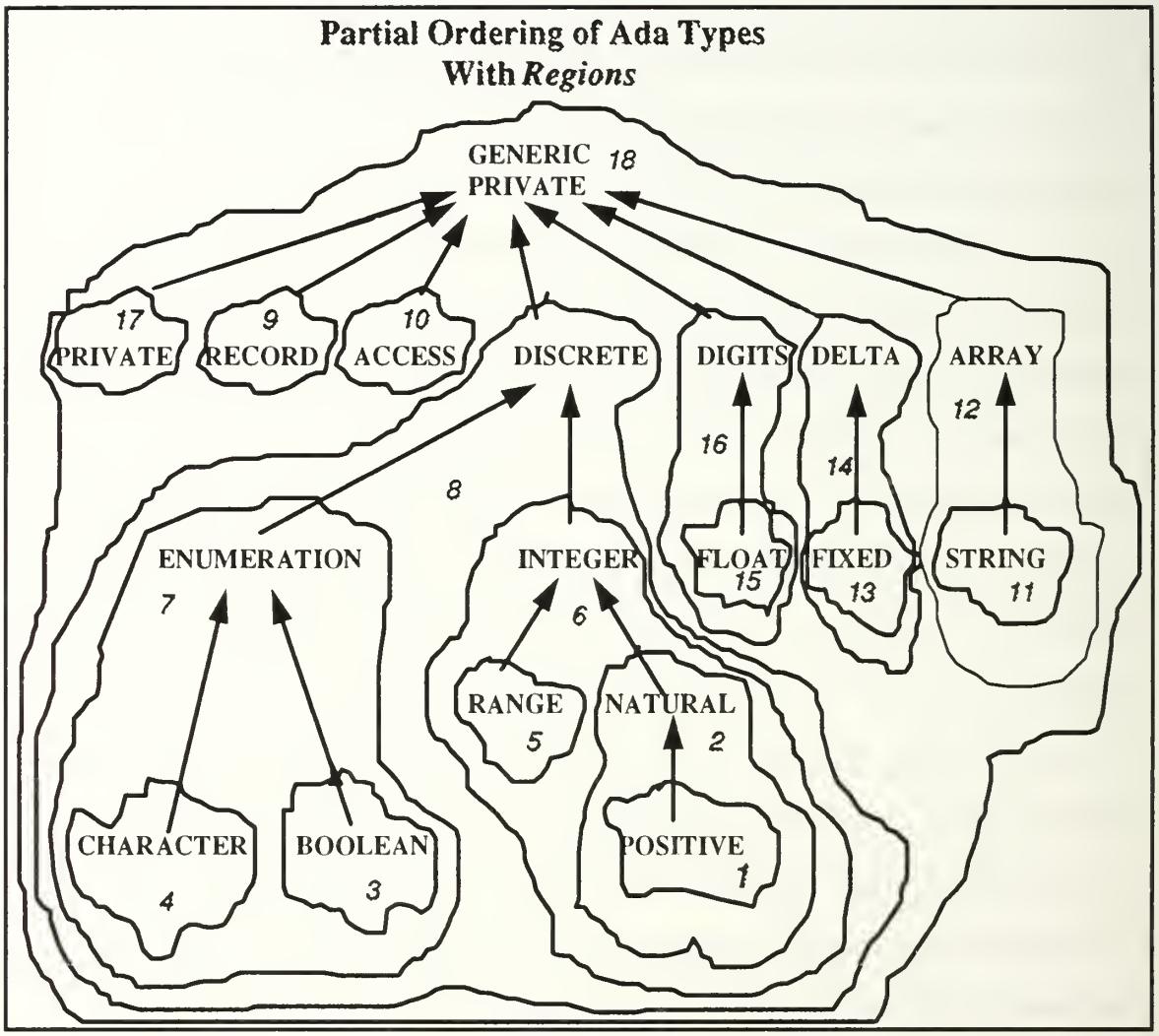

Figure 7 - Subtype Hierarchy with Regions (Input Parameters)

Figure 7 depicts a hierarchy of regions for input parameter types. The graphical depiction of one region within another represents the concept of *subtype* and *supertype* relationships in a type hierarchy; inner regions are the subtypes of their outer regions, and outer regions are the supertypes of their inner regions. Note that unlike Figure 5, a clear distinction is made between the generic and non-generic Private type. This is necessary because when matching input parameters, the non-generic Private type does not have any descendants. It must be matched exactly or to a generic Private type. The other generic types are not portrayed here because their role is not as crucial with input parameters. For example, a query component input parameter of type Boolean matches equally well to a software base component input parameter of type Discrete or generic Discrete.

This, however, is not the case with output parameter types. The reason for this is that the mapping direction is reversed for output parameters. A query component output parameter of type Boolean does not match a software base component output parameter of type Discrete. However, it does match a software base component output parameter of type *generic* Discrete because that parameter could be instantiated to Boolean. So, for output parameters, we have a separate scheme depicting a hierarchy of output parameter regions as illustrated in the following figure:

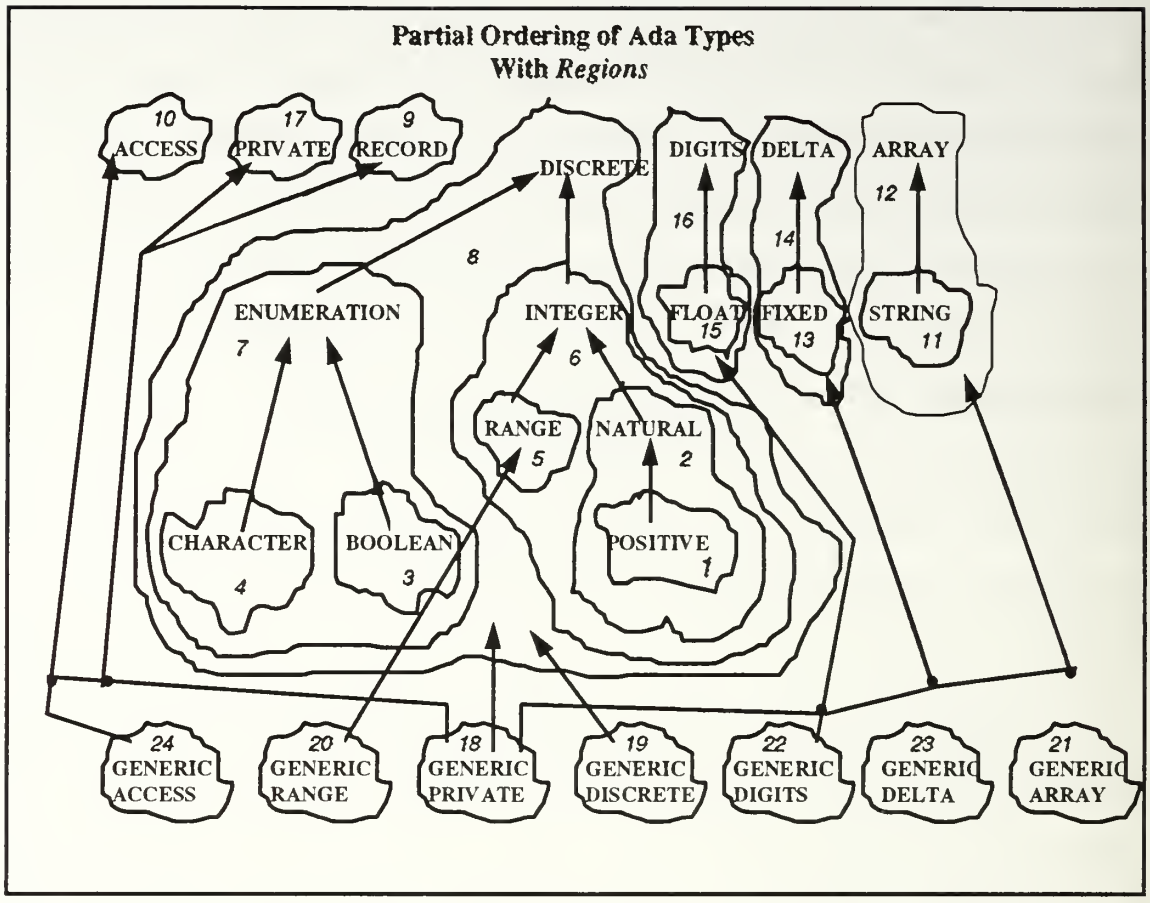

Figure 7a - Subtype Hierarchy with Regions (Output Parameters)

To simplify the drawing, connection dots are used to portray some of the mappings for the generic Private type.

a. Parameters as Patterns

The input parameters that make up a software component's input signature can be portrayed graphically by the number of times they appear in the various type hierarchy regions. A software component with one input parameter of type Integer and two of type Enumeration could be graphically depicted as follows:

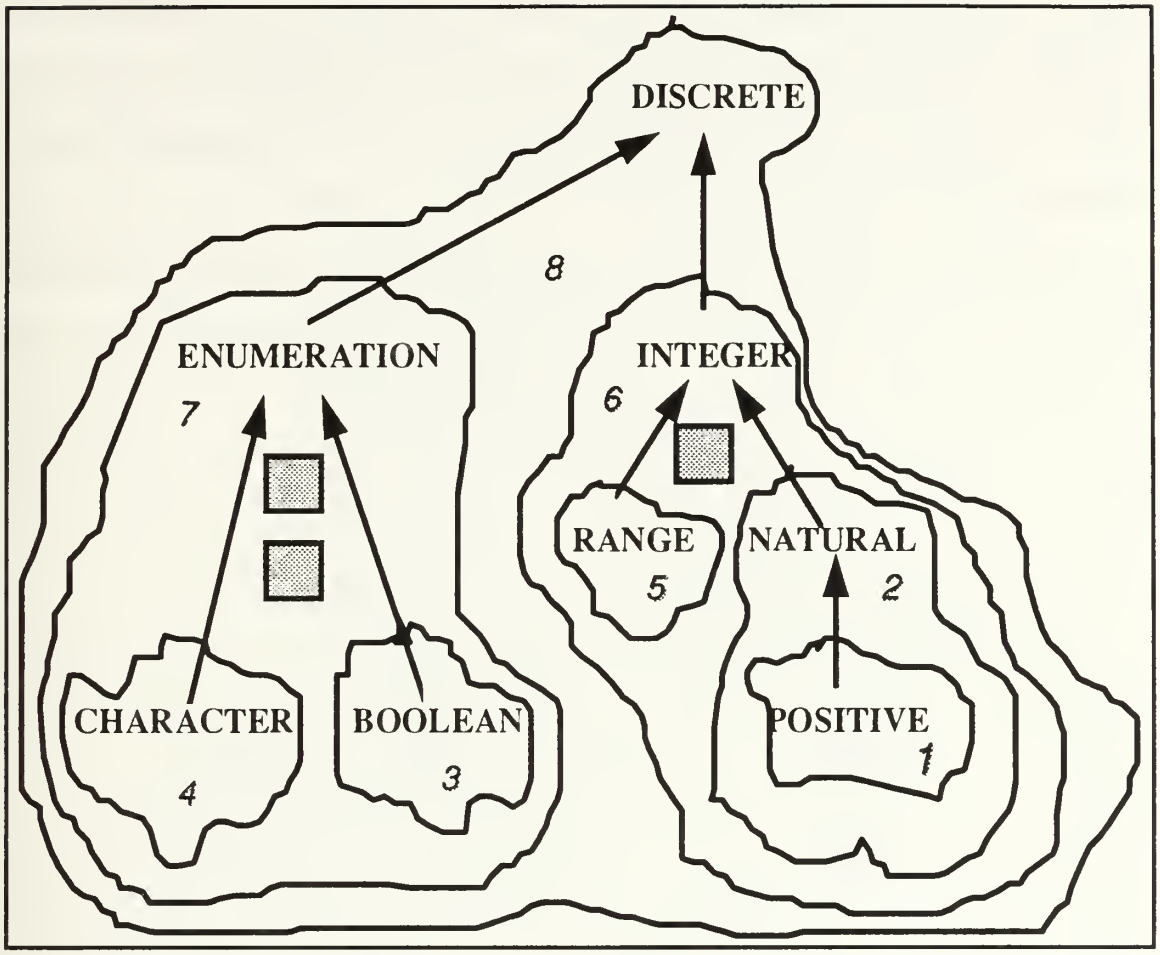

Figure 8 - Parameters as a Pattern

The layout of these input parameter types in their respective regions creates a *pattern* that graphically portrays the input signature of the software component. Another software component with the same pattern must have the same input parameter composition because a pattern is composed of the sub-patterns of individual parameter types (i.e., a shaded block in the Enumeration region, a second shaded block in the Enumeration region, and a shaded block in the Integer region).

b. Pattern Matching

The more difficult question, however is how to identify a pattern that is similar because it provides a correct mapping to the pattern in Figure 8. For example, suppose

we are trying to match the software component in Figure 8 (the software base component) with a query component that has input parameters of type Character, Boolean, and Positive. We will graphically portray these parameters as filled black circles as follows:

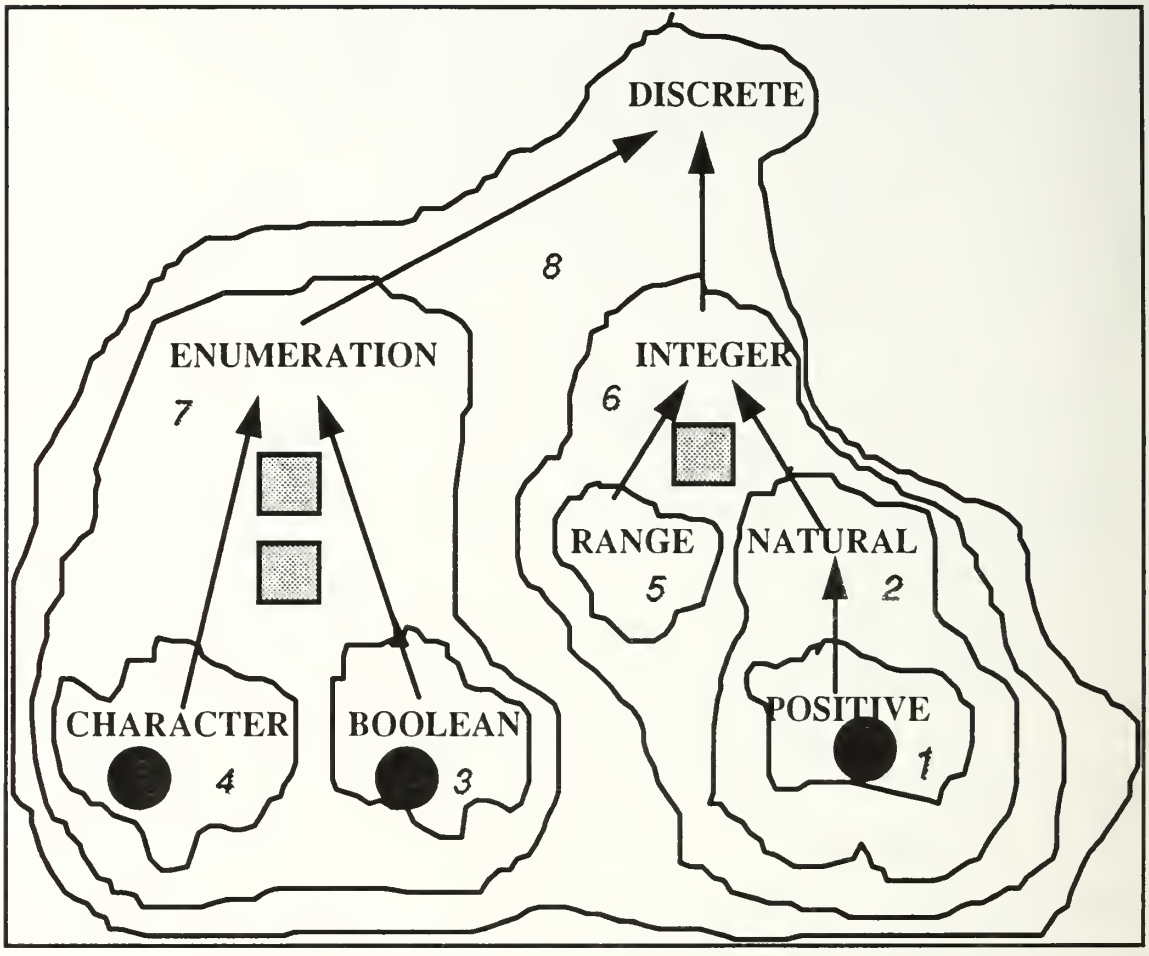

Figure 9 - Input Parameter Pattern Matching

The solution to matching similar patterns is solved by utilizing set theory and applying it to the hierarchy of regions. *Input parameter regions* are considered sets whose elements consist of all *subset* regions and themselves. Conversely, *output parameter regions* are considered sets whose elements consist of all *superset* regions and themselves. As an

input parameter example, an input parameter of type Integer is represented as a set containing the elements Regions 6, 5, 2, and 1, which can be written as Region_Set(Integer) = {Region 1, Region 2, Region 5, Region 6}. As an output parameter example, Region_Set(Integer) = {Region 6, Region 8}. A *pattern* is therefore defined as the multi-set containing all regions utilized by a component. A multi-set allows duplication so that each instance of region occupation by a parameter type appears in the set. Patterns are synonymous with signatures, and there is a separate pattern for the input and output signature. A query component pattern (**QCP**) matches a software base component pattern (**SBCP**) when $\mathbf{QCP} \subseteq \mathbf{SBCP}$. Thus, for our example in Figure 9, the query component pattern is {Region 1, Region 3, Region 4} which is a valid subset of the software base component pattern of {Region 1, Region 2, Region 5, Region 6, Region 3, Region 4, Region 7, Region 3, Region 4, Region 7}.

One benefit with matching a query pattern as a subset of a software base pattern is that extra input or output parameters in the software base component do not affect the matching process. They are simply irrelevant pieces of the software base component pattern. As long as the software base component's pattern includes the elements of the query component pattern, a match is recognized. The reason this is true is similar to the prime numbering scheme concept of getting match results by dividing signatures, discussed in Section IV.C.1. Let us use Figure 9 and an input signature or pattern as an example (we will ignore generic Privates for this example). The Region_Set for a query component input parameter of type Positive is {Region 1}. This input parameter can legally map to software base component input parameter types Natural, Integer, or Discrete which have Region_Sets of {Region 1, Region 2}, {Region 1, Region 2, Region 5, Region 6}, and {Region 1, Region 2, Region 5, Region 6, Region 8} respectively. The pattern for the query component input parameter Positive is a

subset of any of the software base component input parameter type patterns it can legally map to. Adding set elements representing additional input parameters to the software base component input signature pattern does not alter the subset relationship of the query component input signature (in this example comprised of the single input parameter type Positive).

c. Signature Representation

The ultimate size of a software reuse library could easily be on the order of tens of thousands of components or more. Because pattern matching can be a computationally intensive process, a representation for component input and output signatures is needed that allows signatures to be compared quickly. Signatures (or patterns) are represented as a series of 32 bit integers with each integer representing an individual region. Because all operations can be carried out by comparison of integer values, the matching of signatures can be accomplished fairly rapidly. Graphically, a signature is represented as follows:

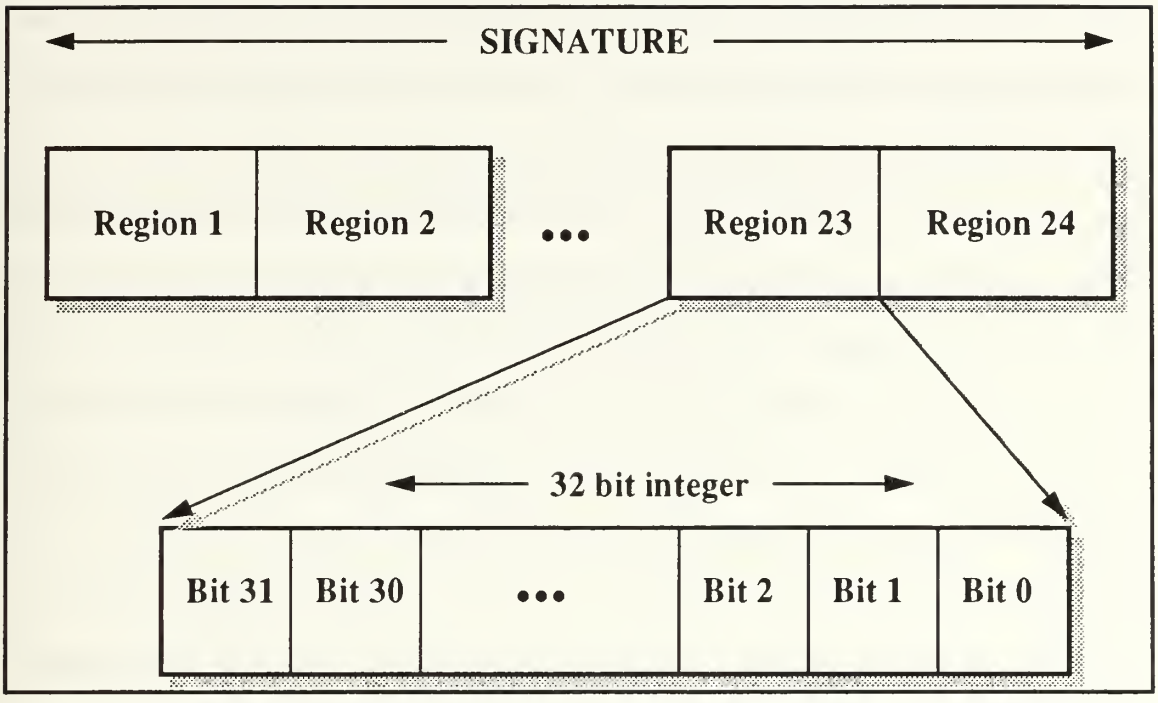

Figure 10 - Graphic Representation of a Signature

As Figure 10 shows, each region consists of a 32 bit integer. The value of the integer is equivalent to the number of type occurrences encoded in a region. Therefore, the upper limit on number of parameters of a particular type is $2^{32} - 1$. A signature is the logical concatenation of 24 regions. Comparison of two input signatures actually involves 18 comparisons (one for each non-generic region and one for generic Private). The generic regions other than generic Private need not be checked because the same result is returned by checking the non-generic region equivalent (see Section IV.C.2 for an example). However, comparison of two output signatures cannot necessarily ignore any of the generic regions. Seventeen comparisons are initially made for the non-generic regions. If any of those 17 comparisons fail to match, then a further check must be made of the corresponding generic regions. For example, suppose that the value of Region 6 for a query component output signature is three (three occurrences of type Integer) while the value of Region 6 for a software base component output signature is two (two

occurrences of type Integer). This initially indicates that the two signatures might not match, but is not conclusive. We must check the all relevant corresponding generic regions. In the case of Integer, the corresponding generic regions to check would be generic Discrete and generic Private. Looking at the generic Discrete region (Region 19) of the software base component output signature, if it is greater than zero (i.e., it has one or more occurrences of type generic Discrete) then we can match one of those occurrences with the query component output signature Region 6 instance that was previously unmatched. At this point, assuming all other regions from one to seventeen matched, we have a valid match. If the generic Discrete region did not have any type occurrences, or had fewer occurrences the number needed, we would continue by examining the occurrences of the generic Private region (Region 18) in the same manner.

d. Building the Signature

All integers representing regions in the signature are initially set to 0. Parameters are dealt with sequentially. The regions corresponding to a parameter's type are determined based on whether you are working with an input or output signature. The relevant regions for a parameter are encoded in the signature by incrementing the integer for each particular relevant region by 1. So, for example, when all input parameters have been encoded, the resulting series of 24 integers (regions) is the corresponding input signature.

e. Signature Matching

Let us look at a simplified example to see how signature matching is performed. Suppose we are comparing two input signatures. The software base component's input parameters are one Enumeration type, one Integer type, and one Natural type. Because we are working with input parameters, we can map in a downward direction in the type hierarchy and we are interested in subset regions.

Region_Sets for the software base component will be (Enumeration = {7, 4, 3}), (Integer = {6, 5, 2, 1}) and (Natural = {2, 1}). Its signature will have integer values of one for Regions 7, 6, 5, 4, and 3 and integer values of two for Regions 2 and 1. Our first query component's input parameters are one Boolean type, one Integer type, and one Positive type. Regions for the query component will be (Boolean = {3}), (Integer = {6, 5, 2, 1}) and (Positive = {1}). Its signature will have integer values of one for Regions 6, 5, 3 and 2 and an integer value of two for Region 1. The integer values for all other regions for both components remain set to zero.

Normally only the query component signature is encoded during matching because the signatures of the software base components are computed when the components are stored in the reuse library. Once the query signature is encoded, comparing signatures is a fast and simple process. The first query component described above is taken through this process in the following figure:

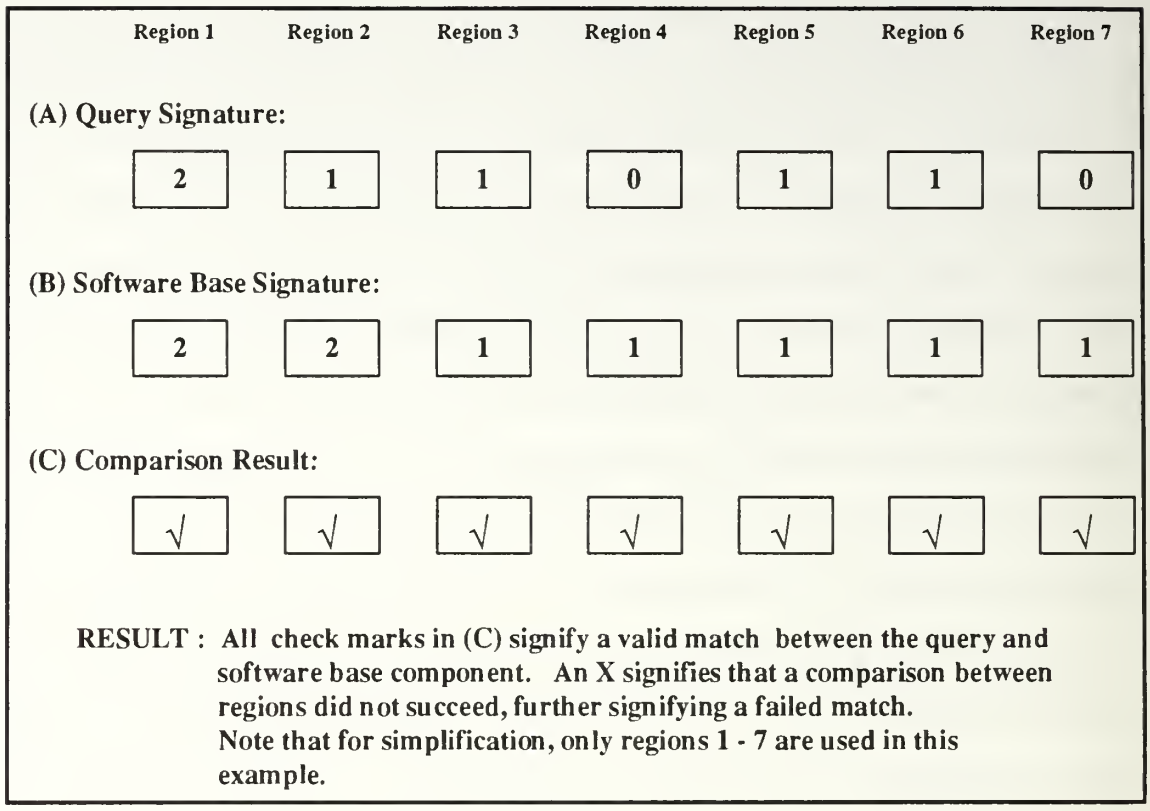

Figure 11 - Signature Matching Example 1

The first pair of regions to be compared in which the integer value of the query component region is greater than the region value of the software base component region value indicate that a match is not possible and the comparison with that particular software base component is done. If all 18 (only seven are shown in Figure 11) region comparisons succeed then a match has been found, as is the case in Figure 11.

Now let us take a look at a second example. The input parameters for the software base component will remain the same. Our second query component's input parameters are one Boolean type and two Integer types. Region_Sets for the query component will be (Boolean - {3}), (Integer - {6, 5, 2, 1}) and (Integer - {6, 5, 2, 1}). Its signature will have an integer value of one for Region 3 and an integer value of two for Regions 6, 5, 2, and 1. Integer values for all other regions for both components

remain set to zero. The following figure illustrates the signature matching process for the second query component:

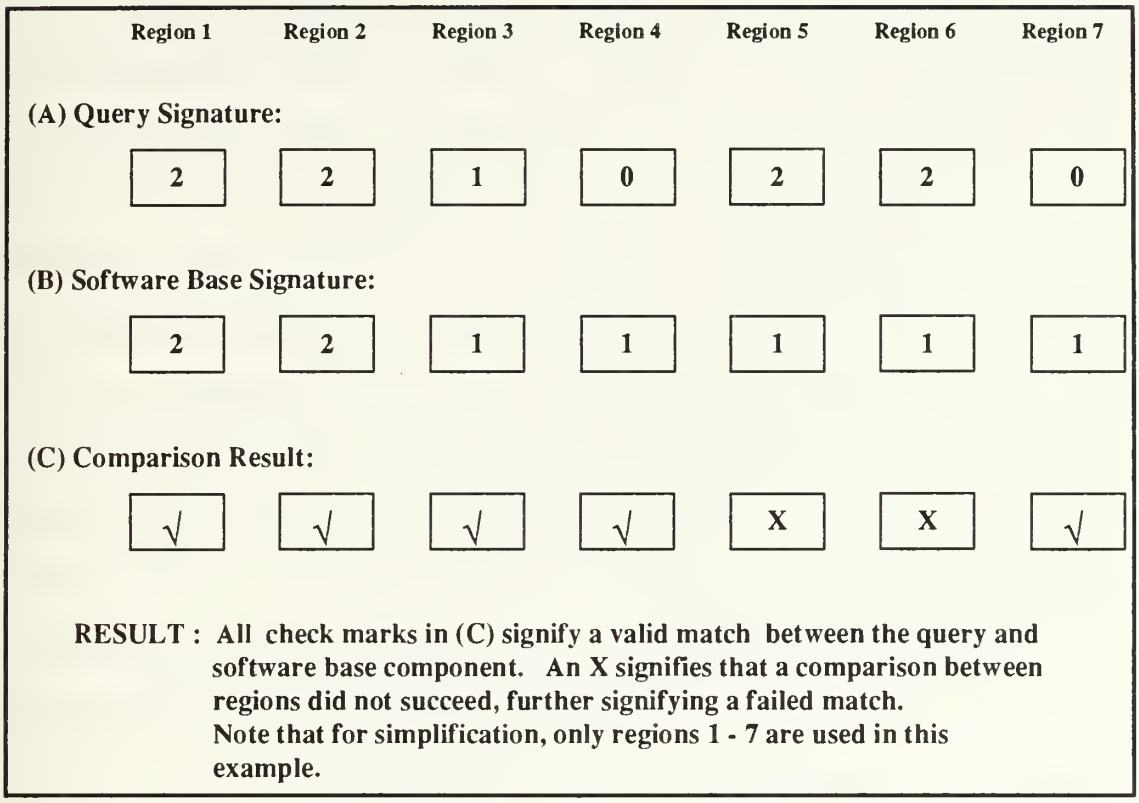

Figure 12 - Signature Matching Example 2

As expected, we did not get a match because the Integer parameter in the query component cannot map to the Natural parameter in the software base component when working with input signatures. The regions that wound up with X's were the two regions (Integer and Range) that were part of the query Integer parameter pattern but not part of the software base component Natural parameter pattern.

f. Limitations with Composite Types

The syntactic matching mechanism described in Sections IV.C.2(a-e) above has some limitations when dealing with composite types such as arrays and records. Composite types are defined as types that are logically made up of several components

[Booc87]. Arrays have two specific components, an index type and an element type. Matching arrays based on the makeup of their two components could be handled in the same manner as matching scalar types by breaking out an array type into three types; a scalarized array, an index, and an element. However, one limitation with this method is that false matches could be recorded due to confusion about which is the array element and which is the array index. For example, Array1[Array_Element : Integer, Array_Index : Positive] would be incorrectly matched with Array2[Array_Element : Positive, Array_Index : Integer]. A second concern is the fact that this could lead to an solution with unbounded nesting, because an Array element could itself be an Array which in turn would have to be described by its components. Records do not suffer from the problem of confusing their subcomponents. Record components, however, have the same potential for unbounded nesting.

One final potential limitation of arrays concerns the matching of an array as a query component input parameter against a generic Private type in a software base component input parameter. While the pattern of the Array type fits into the pattern of the Private type, the patterns of the Array plus its components do not.

The solution to these limitations is to not consider components of composite types during signature matching. Software components are evaluated for matches at the composite type component level during additional match processing subsequent to signature matching.

g. Eliminating False Matches

False matches are a problem with input parameters when using the pattern matching technique. The reason stems from the fact that additional information for each input parameter in the software base component is added to the pattern along multiple downward paths in the type hierarchy to ensure any valid subtype in the query

component will be recognized. For example, if Discrete is the type of one of the software base component input parameters, then not only is the integer value of the Discrete region incremented by one, but the integer values of the regions corresponding to Integer, Range, Natural, Positive, Enumeration, Boolean and Character are incremented by one as well. Because the added information represents several distinct downward paths in the type hierarchy rather than only one, it is possible that the pattern representing the one Discrete type might falsely match input parameter patterns representing several subtypes of discrete with *non-overlapping paths*. Two paths are defined as overlapping if they are equal or if one path contains all the nodes of the other. Any other path is considered non-overlapping. As an example, let us look at a software base component with one Discrete type and two Boolean type input parameters. The corresponding query component will have one Range type, one Natural type, and one Character type for its input parameters. Here is what the input signature matching process would result in for these two components:

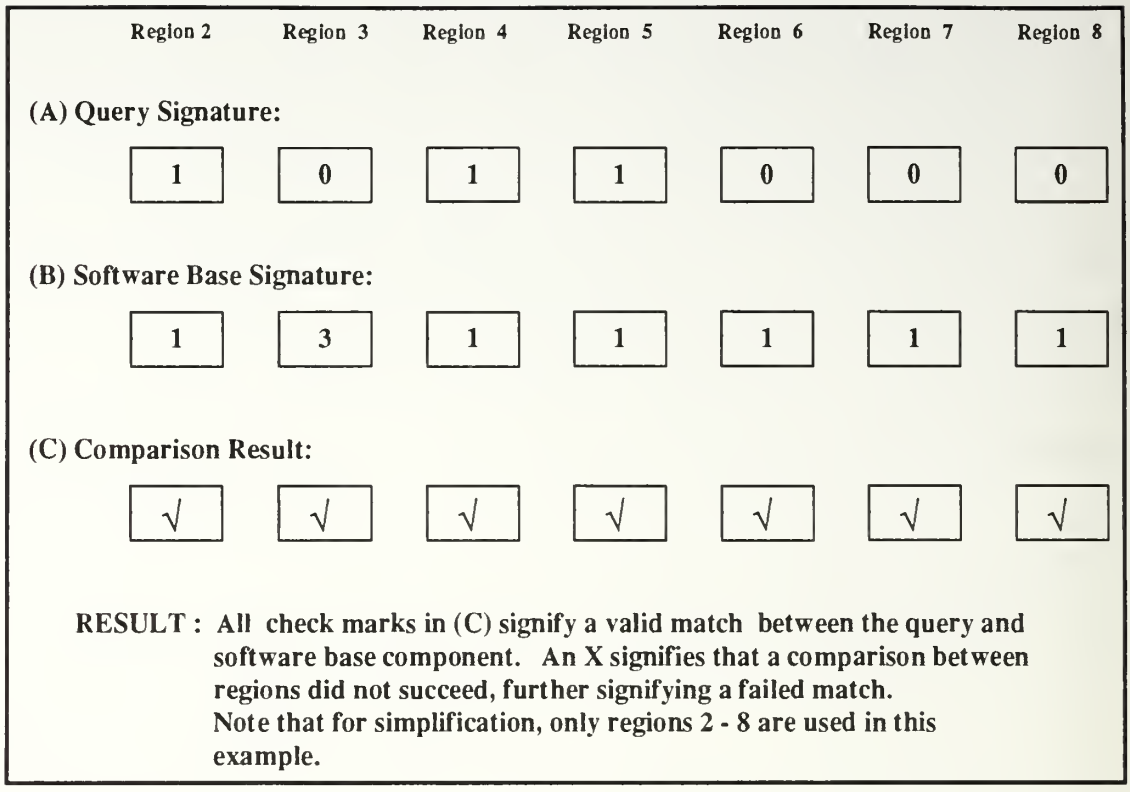

Figure 13 - Example 3

Although we know that the input signatures of these two components do not match, the pattern comparison technique generates a match. It is the non-overlapping paths that create the problem. The pattern for Discrete has several non-overlapping downward paths that can be seen in the type hierarchy of Figure 7. Examples are from Discrete down to Natural, Boolean, Character, and Range. Thus, the one parameter type Discrete could potentially signify a simultaneous match against four separate parameter types. Even if the number of input parameters are the same in both the software base component and query component, we still have false matches created by non-overlapping paths from the same root as we saw in Example 3 above.

Because output parameters deal with supertypes that by definition must have overlapping paths, the process of matching output signatures cannot generate false

matches. Therefore we now describe a method to eliminate false matches for input signatures.

We start with the query component and move from the bottom of the type hierarchy (Figure 7) to the top, working with the leaf nodes. Nodes are equivalent to the regions that comprise a signature. When processing of a leaf node is complete, it is discarded. When all the leaf nodes of a parent node have been processed and discarded, the parent then becomes a leaf node. Processing a leaf node occurs in several steps. If a leaf node has an integer value greater than zero (i.e., it has one or more occurrence of partial encoding information for a parameter type instance), then a parallel path is traced in the query and software base components from the particular leaf node up the subtype hierarchy. The trace in the query component stops when (1) the parent node of the current query node has an integer value of zero (i.e., no partial encoding information for a parameter type instance), (2) the current query node's parent has a smaller integer value than the current node (i.e., partial encoding information for fewer parameter type occurrences), (3) the current query node is the top of the subtype hierarchy, or (4) the software base current node has an integer value of zero and the current query node has an integer value greater than zero. In the first three cases we have completed a trace of a single query input parameter and it has a valid matching software base component input parameter. The fourth case represents a false match. At this point, we have a parallel trace established on the software base component. We now decrement the integer value of the query component current node and all its descendant nodes by one which has the effect of removing all the encoding of a parameter type instance from the query input signature. The trace up the type hierarchy continues for the software base component and halts under the same first three conditions described for stopping the query component trace. At this point we have a complete path defined in our software base

component. The root of the path represents the type of the corresponding software base component input parameter. Next, all downward paths from that type must be traversed and the integer value decremented by one at each node including the root.

To see how the fourth case described above would occur let us use the two components from Example 3 in Figure 13. Example 3 generated a valid match, so now we must test it to see whether a false match had been generated. Refer to the following figure as we check for a false match:

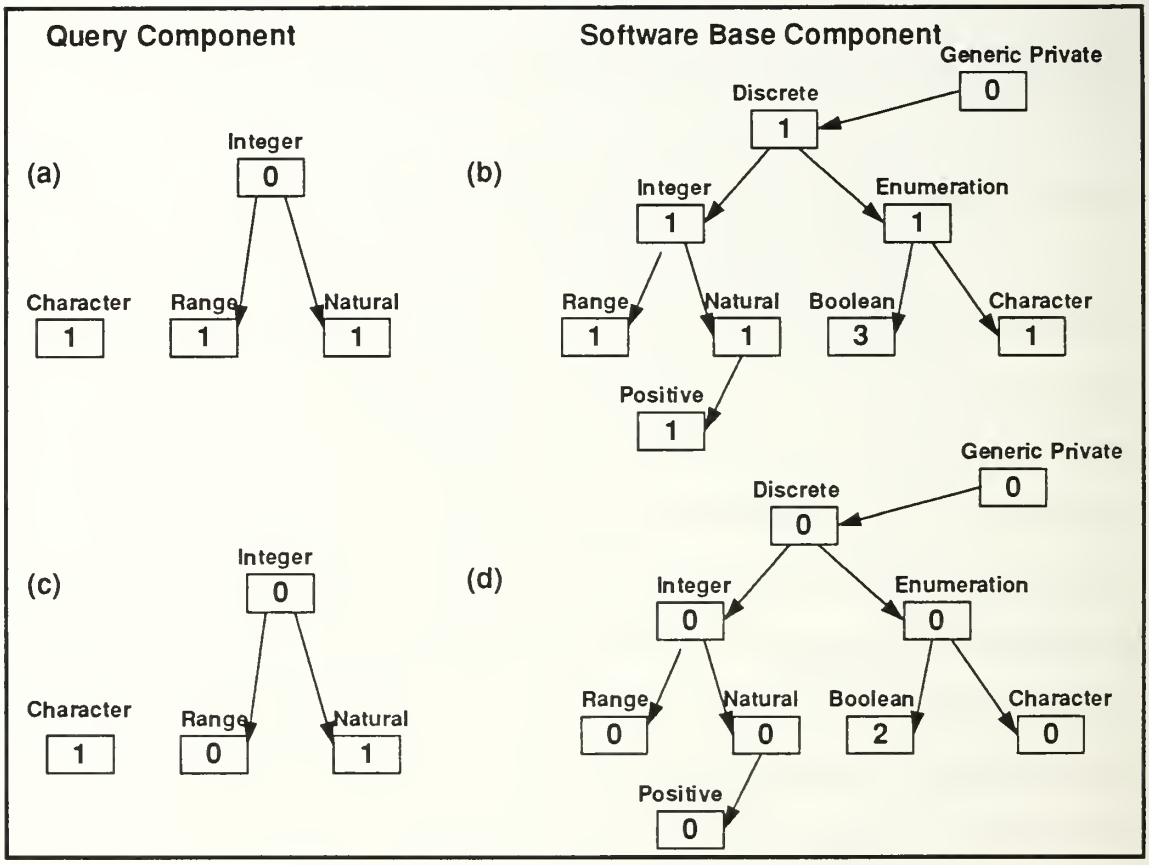

Figure 14- False Match Example

Since two of the three input parameters for the query component are leaf nodes in the Ada type hierarchy, we arbitrarily pick Range to start with. It has an integer value greater than zero and so we attempt to proceed in an upward path in both the query and

software base components. However, because the query component parent (Integer) has an integer value of zero, we go no further with the query component (Figure 14 (a)). We decrement the integer value of the query component Range node and all its descendants by one (in this case, the type Range has no descendants - Figure 14 (c)). We now continue with the software base component moving upward through the Ada type hierarchy from Range until we reach Discrete. At that point, the parent of Discrete has an integer value of zero (Figure 14 (b)). Since Discrete is the root of the path we found in the software base component, it must be an input parameter type. We must now decrement the integer values by one for all nodes along the paths Discrete-Range, Discrete-Positive, Discrete-Character, and Discrete-Boolean (Figure 14 (d)). Care must be taken to only decrement the integer value of a node by one the first time it is traversed. The only node with a non-zero integer values in the software base component input signature at this point is the Boolean region (Figure 14 (d)). We now turn back to the query component and proceed with the next leaf node, Character (Figure 14 (d)). Immediately we run into case four where the query component Character node has an integer value of one, but the corresponding software base component Character node has an integer value of zero indicating that we have a false match between these two components (Figure 14 (c) and (d)).

Thus we have a a process to detect false matches. In cases where two components do match, the check for false matches will confirm the valid match.

h. Measuring Signature Closeness

In a software reuse library with tens of thousands of components it is expected that many attempts to match a particular query component will yield multiple candidates. It is additionally desirable, therefore, not only to provide all matches but also to order

them from the closest match to least closest. From that point individual scrutiny on a component by component basis must be performed by the software prototype developer.

For *operator* components, using the difference in number of output parameters between the query and software base component is a good first criterion to base the ordering of candidate matches. It is possible to further improve the candidate ordering by using *signature closeness* as the second ordering criterion. Signature closeness is determined by evaluating an overall measure of closeness in type between two signature's parameters. *Closeness* is defined as the length of the path (i.e., number of edges) between two nodes in the type hierarchy. This is well defined because the type hierarchy is a tree, so there is exactly one path between any two nodes related by the subtype relation. Closeness is only relevant where a) the two nodes are the same, or b) one node is a descendant of the other. We have defined closeness between two parameter types. The closeness for a signature is an extension of that definition and is the summation of the individual closeness measurement between each query component parameter and its matching software base component parameter. Input signature closeness is the summation for input parameters, and the output signature closeness is the summation for output parameters. *Operator* closeness is the summation of the input signature and output signature closeness values.

When the software base component has more output parameters than the query component, these extraneous parameters must not be included when calculating the closeness degree. For one thing, the difference in the number of output parameters has already been taken into account by the first criterion described above for ordering matches. A second reason is that the results generated by those output parameters are not necessarily of interest to the prototype designer. Therefore, the closeness degree is a measurement describing type distance between all matched parameters. Input and output

parameters are processed in a similar manner with slight differences to arrive at an overall signature closeness measurement.

For input parameters, we piggyback onto the process that checks for false matches in Section IV.C.2.g. The false match check extracts the query and software base component parameter pairs which allows us to then calculate the closeness once the exact parameter types of the matched parameters are known. The crucial part of measuring closeness comes when the upward trace is halted in the query component while checking for a false match. At that point both the query and software base component have a closeness degree of 0 and we have a parallel trace established on the software base component. The trace up the type hierarchy continues for the software base component and halts under the same three conditions described for stopping the query component trace. At that point we have identified the software base parameter type that matches the query component parameter type we are trying to match. Each additional node reached in the software base component trace is counted as a degree of closeness. Therefore, for example, if the query component trace went from Positive through Integer and the software base component trace continued on to generic Private, the difference in nodes from Integer to generic Private is two which corresponds to the degree of closeness between the two input parameters Integer (query component) and generic Private (software base component).

With output parameters we also start with the leaf nodes of the query component. Because output signatures are encoded by adding one to the integer value of the node corresponding to the parameter type and all its ancestors, all query component original leaf nodes that have type occurrences encoded in them will have exact matches. It is when we exhaust all query component leaf nodes and start examining nodes at a higher level (also called leaf nodes since at that point their children

have been eliminated, but note that these are not original leaf nodes) that we may encounter a positive closeness factor. When a leaf node has an integer value greater than zero, we decrement by one the integer value of all nodes from the leaf node along the path to the top of the hierarchy in both the query and software base component. Then we must trace the software component from its corresponding leaf node down through the type hierarchy until we reach a descendant node that is a subtype hierarchy leaf node or whose integer value is greater than the sum of the integer values of its children. If, in order to continue a required downward trace, we must choose amongst the children of a node, the child is picked that has the shortest possible path to a leaf node. Each additional node reached in the software base component downward trace is counted as a degree of closeness. Therefore, for example, if the query component trace was based at Integer and the software base component trace continued from Integer down to Positive, the difference in nodes from Integer to Positive is two which corresponds to the degree of closeness between the two output parameters.

The methods described above for measuring input and output signature closeness apply equally to *type* components. The only difference is that where *operator* closeness is the summation of the input and output signature closeness values, *type* closeness is the summation of the *operator* closeness values for each of the *type's operators*. Despite the fact that the calculation of *type* closeness is a simple extension of the calculation of *operator* closeness, it is not used in this implementation of syntactic matching. The reason for this is due to the semantic uncertainty of matching the *operators* of a *type* component. There may be numerous valid mappings between the *operators* in the query and software base component, but there is no way to know in advance which of the valid mappings is the one desired by the user. For example, suppose the query *type* component has two *operators*, the first with a single input

parameter of type Positive and the second with a single input parameter of type Natural. Let us further suppose that a candidate software base *type* component has four *operators*. The first *operator* has a single input parameter of type Integer, the second has a single input parameter of type Integer, the third has a single input parameter of type Natural, and the fourth has a single input parameter of type Positive. There are six different possible mapping from the two query component *operators* to two of the four software base component *operators*, with a range in aggregate input signature closeness from 0 to 3.

i. An Additional Syntactic Matching Filter For Type Components

The aggregate input and output signatures provide a useful filter for matching *type* components but do not ensure that a match has been found. The next step in the process of matching *type* components is to determine whether a valid mapping can be derived for the *operators* of the query and software base *type* components as was briefly discussed in the above paragraph. All combinations of mappings pairing each query *operator* with one of the software base *operators* must be examined until a valid mapping is found. For each query/software base component *operator* pairings, the input and output signatures of the *operator* pair are compared. If they match, then that particular pairing is valid. If all pairings in a total mapping are valid, then the mapping is valid and the candidate software base *type* component successfully passes through this filter and remains a match candidate.

j. An Additional Syntactic Matching Filter for Non-Generic Components

Section IV.C.f discussed the limitations of composite types such as Arrays and Records and suggested that the components of these composite types be excluded from signature matching and examined during subsequent match processing. An additional filter has been developed for ensuring that parameter composite types match at the

component level. This process is very similar to the process for determining a valid mapping between *type* component *operators*. Only the components of Array types are examined in this implementation. Note that String types are included as special cases of Array types. In addition, this filter was implemented for *operators*, but not *types*.

All combinations of mappings between the query component *operator* input Array parameters and the software base component *operator* input Array parameters are explored until either a valid mapping is found or all combinations have been exhausted. A valid mapping has been found when each one of the query component *operator* input Array parameters can be matched to a distinct software base component *operator* input Array parameter at the Array's component level. This matching only goes to the first level. So, for example, if an Array element is an Array, that element is not subsequently matched based on the values of its components. If a valid mapping is found for input Array parameters, then the same process is carried out for output Array parameters. If a valid mapping is then found for output Array parameters, the candidate software base *operator* component successfully passes through this filter and remains a match candidate. The process for both input and output Array parameters can be summarized as follows:

- Go through the list of query component Array parameters. For each, go through the list of software base component Array parameters and record all matches.

- Start with the first query component Array parameter and select the first software base component Array parameter it has a recorded match with. That software base component Array parameter is now considered IN_USE.

- Go to the next query component Array parameter, and select the first software base component Array parameter that it has a recorded match with and that is not IN_USE. Mark its selection as IN_USE.

- Continue this process through all query component Array parameters. If the last query component Array parameter is reached and there is a software base component Array parameter that it has a recorded match with that is not IN_USE then we are done and a match was found for this filter. Otherwise, if a query component parameter is reached but all the software base component Array parameters that it has a recorded match with are in use, backtrack to the previous query component Array parameter, mark the current software base component Array parameter it is mapped to as not IN_USE, and attempt to find the next software base component Array parameter that it has a recorded match with to mark IN_USE.

- If we backtrack to the first query component Array parameter after trying the last software base Array parameter it recorded a match with, then the match fails.

Generic components do not use this filter for two reasons. First, the addition of generics complicates the process of mapping query to software base component Array parameters. With non-generic components, every query component Array parameter must have a corresponding software base component Array parameter because the input and output signatures of the two components have already been determined to match. However, with generic components, it is possible for the software base component to have fewer Array parameters than the query component with generic parameters making up the difference. The second reason is that generic components use a separate filter discussed in the next section that includes the matching of Array types at the Array component level.

k. An Additional Syntactic Matching Filter for Generic Components

An additional filter has been developed for ensuring that a query component can instantiate a candidate software base component and subsequently match that instantiation. This process is similar to the process for determining whether the Array parameters of two components match fully, but takes on an extra level of complexity because it deals not only with all query component parameters, but with the

instantiations of the corresponding generic parameters in the software base component. Two separate mappings must be determined. The first maps the generic parameters of the software base component to the query component parameters that will instantiate it. The second maps the query component input and output parameters to the software base component input and output parameters. These two mappings are related, because the instantiations determined by the first mapping affect the matching of input and output parameters of the second mapping. To reduce complexity, the possible generic instantiations are determined first. The reason is that the combinations for instantiation of the generic parameters can be accomplished without regard to the order in which the generic parameters are examined because they do not affect one another. This is not the case with input and output parameters. If input and output parameters are examined first, then every possible ordering of those parameters must be considered so that all possible instantiations are considered.

If one or more of the generic parameters in the software base component cannot be instantiated then the match fails. In addition, this filter was implemented for *operators,* but not *types*. The process for determining whether a valid instantiation of the generic software base component exists can be summarized as follows:

- First determine the first valid mapping from the software base component generic parameters to corresponding query component parameters that can instantiate them. This mapping determines the instantiations of the software base component input and output parameters that are defined by generic parameters.

- Next, attempt to find a valid mapping between the query component input parameters and the software base input parameters. If a valid mapping is found then continue by trying to find a valid mapping between the query component and software base component output parameters. If a valid mapping is found then we are done and a match was found for this filter.

- If a valid mapping could not be found for either the input or output parameters then we must backtrack to find the next valid generic instantiation mapping and proceed from there. If all generic instantiation mappings have been exhausted without finding valid mappings for the input and output parameters, then the match fails.

The detailed process for determining a valid mapping for both the generic instantiation and input/output mappings is identical to the process described in Section IV.C.2.j for mapping Array parameters. The only difference is what objects are mapped. For example, with generic instantiation mapping we are mapping software base component generic parameters with query component input and output parameters instead of query component Array parameters with software base component Array parameters. The process for arriving at a valid mapping is the same.

l. A Graphic Representation of the Syntactic Matching Filtering Mechanism

The syntactic matching mechanism can be summarized by the graphic representations of the component filtering process depicted below. The filters are represented by the conical shapes and have all been described previously. Here is the filtering process for *operator* components:

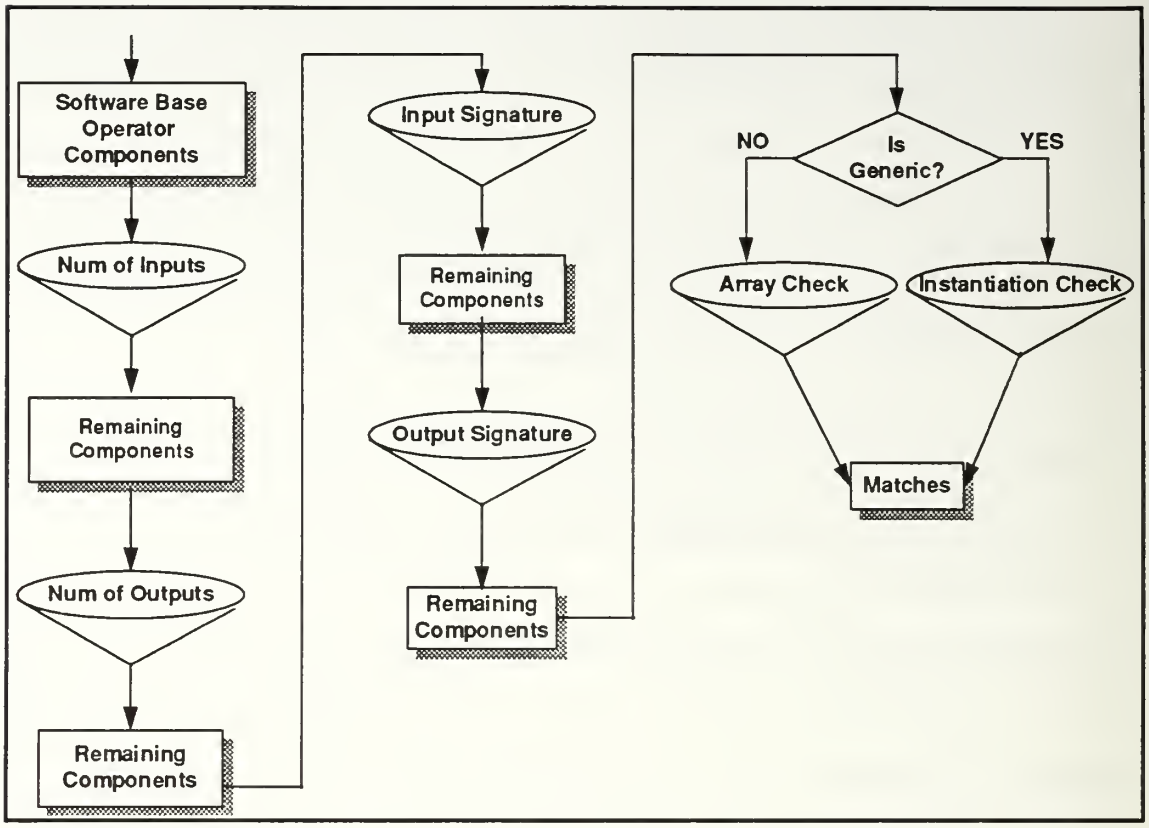

Figure 15 - Filter Process Flow for Operators

The next figure depicts the filtering process for *type* components. Note that total number of inputs and outputs reflect aggregate numbers for the *type* component's *operators* as do the input and output signatures.

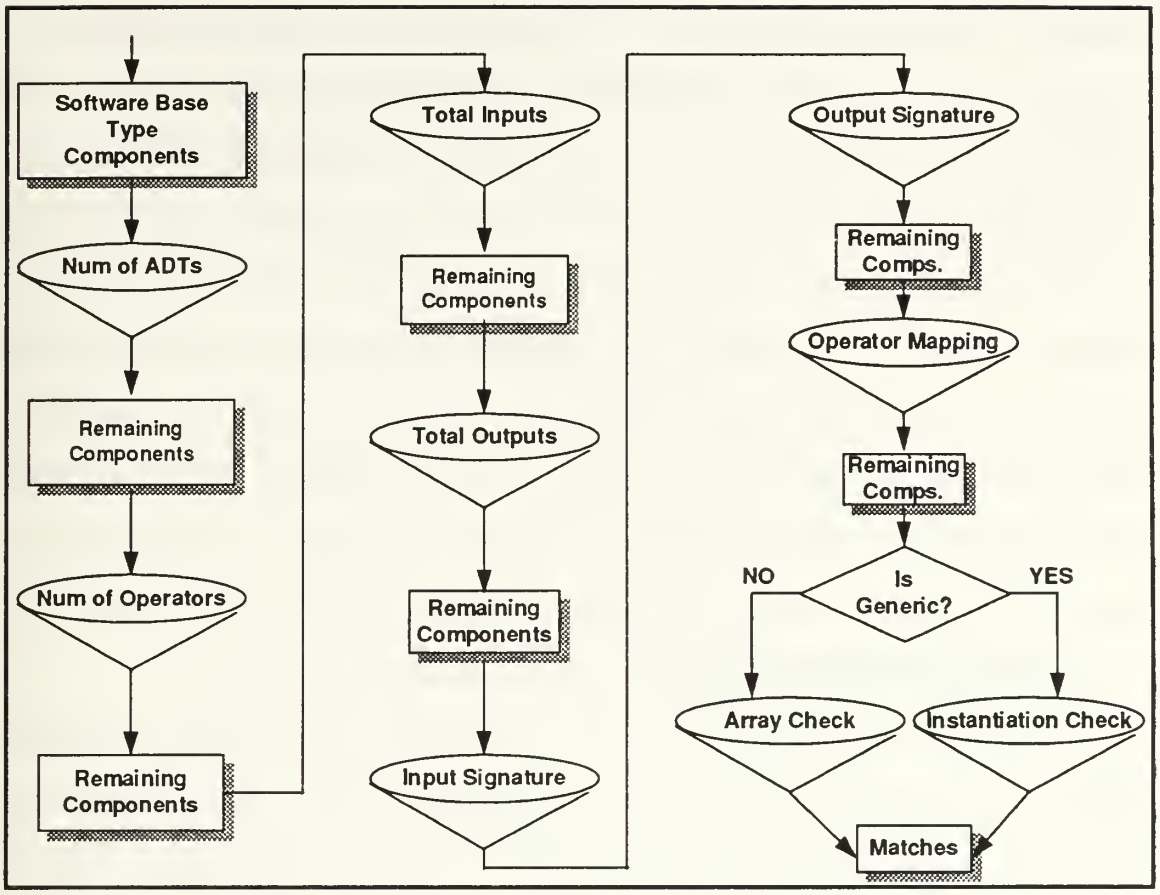

Figure 16 - Filter Process Flow for Types

D. IMPLEMENTATION DETAILS

This section describes many of the implementation details specific to realizing the syntactic matching mechanism described in Section IV.C.2 and also summarizes the changes made to the CAPS software base to extend its syntactic matching capability.

1. Ada Language Usage Limitations

In its present form, CAPS does not permit certain Ada language constructs to be used in the Ada specifications and bodies that make up the CAPS software base. Many of these limitations are due to the current programmed capabilities of the CAPS

translator. First, an *operator* cannot be implemented as a function; it must be a procedure. Second, procedure input and output parameters may only be defined as either "in" or "out" but not as "in out." This limitation has an important implication for naming PSDL specification input and output parameters. Specifically, it requires that no input parameter name be the same as an output parameter name within the same *operator* in a PSDL specification. Third, Limited Private types are not allowed because the translator relies on the ability to do assignments. Fourth, the only generic parameters allowed are generic types. No generic objects or values are allowed. Furthermore, there is no current means in PSDL for a prototype designer to define a procedure that a generic software base component may require as a generic parameter for instantiation.

2. Changes Made to the Original CAPS Software Base

The original CAPS software base was implemented by John Kelly McDowell. [McDo91] The primary change to the implementation of that software base by this thesis is the extension of its syntactic matching to include the ability to match components based on the types of their input and output parameters. This extension has resulted in new code in the form of Ada packages and programs for signature encoding and C++ classes and programs for type matching. This new code is described in more detail later.

a. The Physical Schema for the Software Base

To support our new method for syntactic matching, we have made some changes to the software base schema originally designed by McDowell.[McDo91] Before discussing those changes let us first review the original software base. Using states and the number of input, output, and generic parameters as multi-attribute keys, McDowell designed a *software base schema* that hierarchically partitions the search

space thereby limiting the number of components that must be searched for a valid match. Separate hierarchies were designed for *types* and *operators*.

Figure 4 provides a simplified view of the software base schema for *operator* components and is explained in Sections III.A and III.B. It does not include the use of the number of generic parameters as multi-key attributes. Ontos dictionary classes are used as the hierarchy nodes. These dictionaries use the multi-attribute keys as the dictionary *tag* (key) and a dictionary object as the dictionary *element*. The power of this software base schema is its ability to partition components into sets with similar attributes providing a very fast mechanism to find all software base components that match the attributes of the query component.

The changes we have made to McDowell's software base schema are (1) the removal of the number of generic parameters as a multi-attribute key and its corresponding generic dictionary in both the *operator* and *type* component libraries, (2) the addition of *operator* input and output signatures and their corresponding dictionaries to the *operator* component library, and (3) the addition of *type* input and output signatures and their corresponding dictionaries to the *type* component library. Each of these changes is explained in the following sections.

b. Removal of the Generic Dictionary

The first change made to the software base schema was the removal of the number of generic parameters from consideration as a multi-attribute key in the matching process. In his thesis, McDowell discussed but never actually implemented parameter type matching. [McDo91] He included some program code to support his parameter type matching methodology in the form of the SB_RECOGNIZED_TYPES C++ class, defined extensions to PSDL in the form of special identifiers, and created a rule matrix that defined valid type mappings or subtype relationships between all

recognized Ada types. McDowell treated user defined types as *unrecognized types* and created a rule between software base component generic parameters and query component unrecognized type parameters. An unrecognized type can be thought of as a user defined type.

This thesis, however, does not permit unrecognized types and forces all query component parameters to be defined as Ada types. That is not to say that a query component parameter cannot be defined as a user defined type. It can. But ultimately by transitivity, a referenced user defined type must be defined in terms of an Ada type. Therefore, given the difference in our approach to matching parameter types, the methodology used by this thesis can be employed without unrecognized types which subsequently eliminates the need to try and match unrecognized types in the query component with generic types in the software base component. Based on this, the number of generic parameters was removed as one of the multi-attribute keys in the software base schema. The removal of the number of generic parameters as a multi-attribute key eliminated the usefulness of maintaining a generic dictionary because the purpose of that dictionary was to partition components based on their number of generic parameters. Therefore, the generic dictionary was also removed.

Note that the removal of this multi-attribute key does not mean that generic parameters are not used in the matching process. They play an important part in both matching signatures and determining whether a valid generic instantiation is possible.

c. Addition of Operator Component Input and Output Signature Dictionaries

Additions made to the complex data hierarchy preserve the efficiency of the data structure while boosting its partitioning power. The first addition was to add the *operator* component input and output signatures as multi-attribute keys. They serve as additional partitioning mechanisms which further reduce the set of components that have

to be examined. *Operator* input and output signatures work a little differently than the other multi-attribute keys like the number of input parameters. Whereas an Ontos dictionary that uses the number of input parameters as a key can immediately retrieve the corresponding dictionary for the next lower level in the complex data hierarchy, input and output signatures must be examined one by one. The reason lies partly in how Ontos identifies objects. Signatures are used as Ontos dictionary keys and are stored in Array objects. Once stored, they are given a unique identifier. If a local Ontos Array is then loaded with an identical signature to one stored in the dictionary, the only way it can find its match is by examining each dictionary Array element by element. The attempt to go directly to the matching Array key in the dictionary will not work because at that level, Ontos is trying to match the unique object identifier value. Because we are concerned with inexact signature matches as well as exact ones, having to scan all signature keys in the dictionary turns out to be inevitable anyway, so nothing is really lost. The fact that signatures are represented as a series of integers with the matching mechanism implemented via simple integer comparison operations proves to be very efficient. So despite the fact that signatures must be examined one by one, the process of doing so is relatively quick because a) considerable partitioning has already taken place before reaching the signature (input or output) dictionary level in the complex data hierarchy and b) simple integer operations are very fast. The following diagram depicts the new *operator* component complex data hierarchy and is used by the SB_OPERATOR_COMPONENT_LIBRARY class:

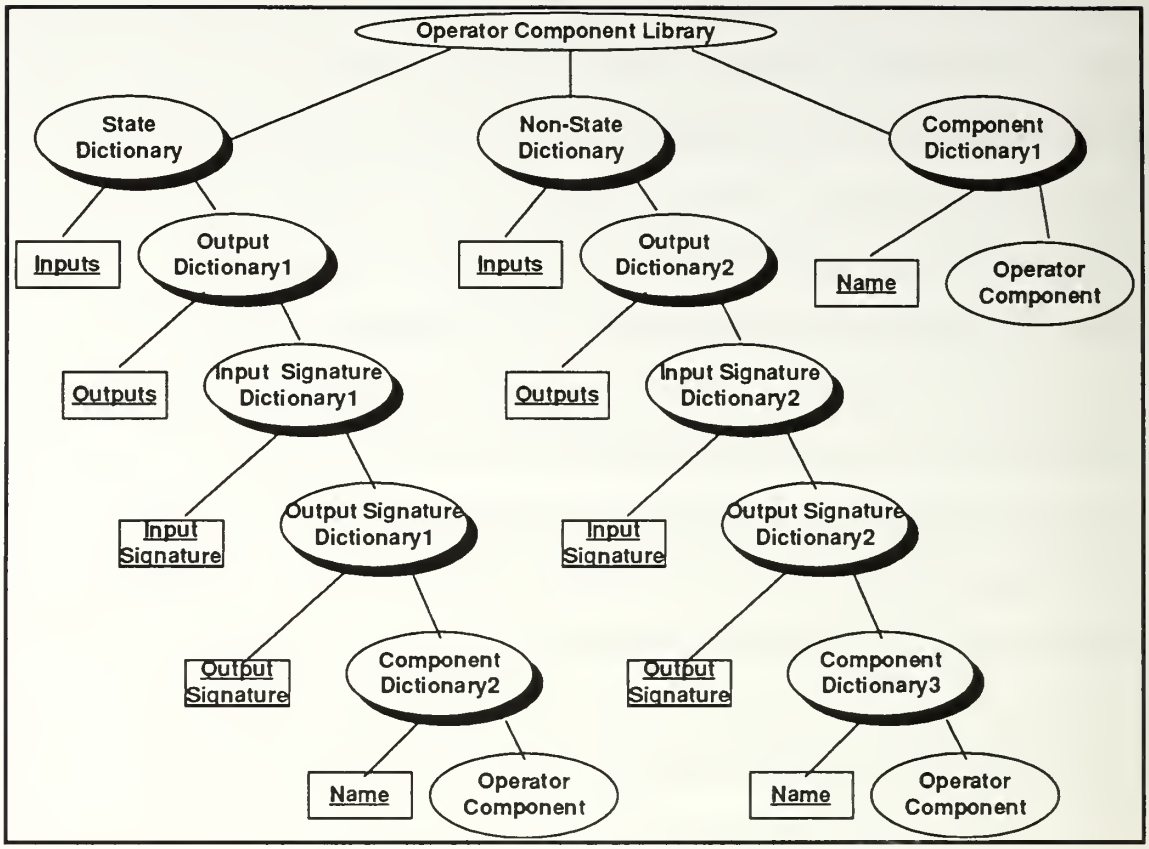

Figure 17 - New Software Base Operator Component Library

Regular ellipses represent objects. Shadowed ellipses represent Dictionary objects. And rectangles represent Dictionary keys. Figure 17 depicts a storage hierarchy of objects. All objects are only stored once in a distinct physical location. References to those objects can then be stored within other objects. That is the role the dictionaries serve above. Their elements are actually references to other objects.

d. Addition of Type Component Input and Output Signature Dictionaries

The second addition was to add the *type* component input and output signatures as multi-attribute keys. They provide the same partitioning and are stored and examined during matching in the same manner as the *operator* component signatures described above. One important difference that the *type* component signatures do possess,

however, is that they are actually aggregates of the individual signatures of the ADT *operators* that make up the *type* component. As aggregate signatures, they serve the useful role of allowing further partitioning without inadvertently overlooking any potential matches. Software base components that match aggregate input and output signatures with the query component have shown that they may ultimately match the query component. However, software base components that do not match aggregate input and output signatures with the query component are guaranteed not to match that query component, thus eliminating them from further consideration. Once the set of potential matches has been filtered via the use of aggregate input and output signatures, a more processing intensive mechanism is invoked that attempts to map individual ADT operators between the query and software base components and perform type matching at the individual operator level. It is in this process that the final determination is made whether a match exists or not. The following diagram depicts the new *type* component complex data hierarchy and is used by the SB_ADT_COMPONENT_LIBRARY class:

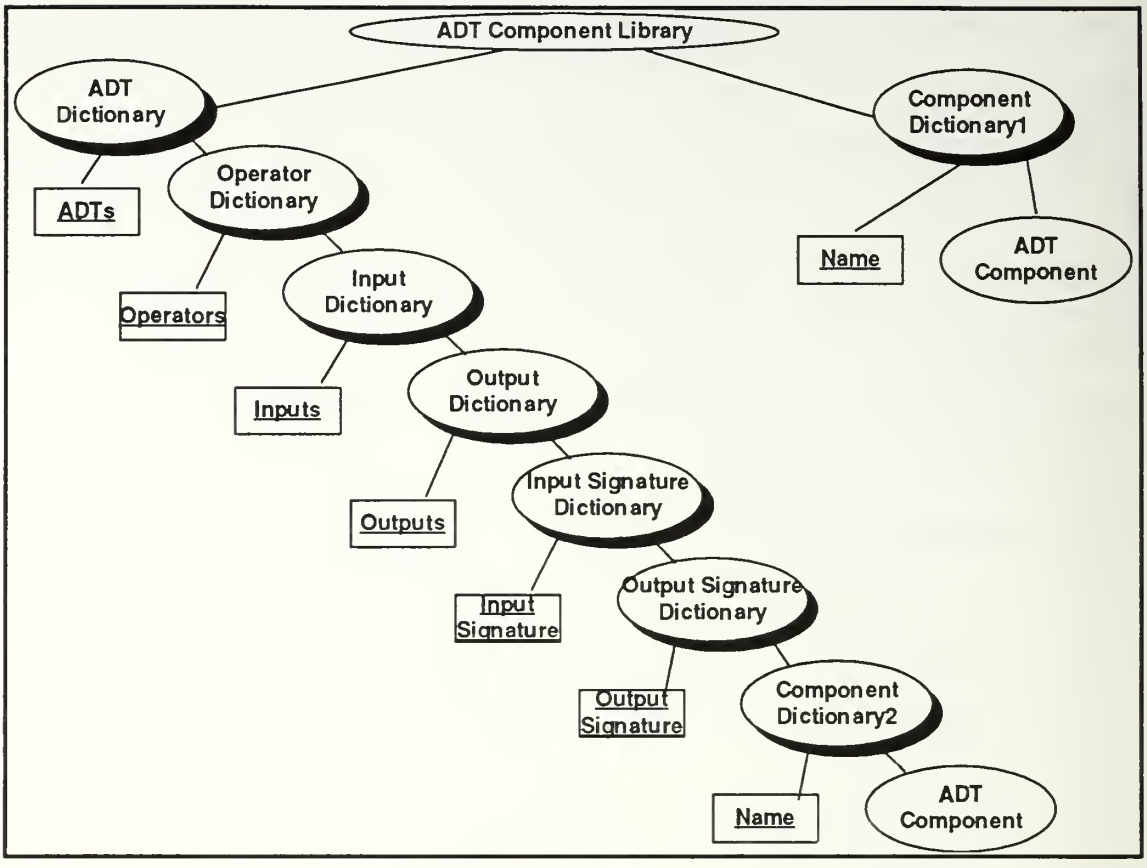

Figure 18 - New ADT Component Library

Note that the multi-attribute keys for the number of input and output parameters are also aggregate values derived from the entire set of ADT operators. For example, the total number of input parameters is obtained by adding the number of input parameters of each ADT *operator* together. The keys ADTs and Operators represent the number of ADTs and the number of operators in a *type* component. Refer to Section IV.A for more detailed definitions.

3. Using PSDL to Specify Components

Although the standards for what constitute a valid PSDL specification are strictly enforced by virtue of the PSDL grammar, rules for writing a PSDL specifications for software base components and for query components are necessary but not defined

anywhere. This is an initial attempt to clarify those rules through requirements and examples. See Appendix B for examples.

a. General Assumptions

Certain assumptions made by the syntactic matching process regarding the construction of PSDL specifications are listed below. These assumptions define constraints on writing PSDL specifications for components. Even though these constraints cannot all be automatically checked at the present time, failure to abide by them could invalidate the results of the matching process. Many of the assumptions are checked during signature encoding and most violations are revealed in the form of error messages. The general assumptions are as follows:

- Query components cannot have generic parameters.
- Software base components cannot have unrecognized types. This means that a software base component cannot reference an external user defined type.
- All Array types will use the identifiers ARRAY_ELEMENT and ARRAY_INDEX for their component parts. These identifiers are case sensitive just like any other identifier including the component names that are stored in the CAPS software base.
- The order of all *operator* or ADT *operator* input and output parameters will be exactly the same as the order of those parameters in the corresponding Ada package specifications and bodies. This requirement is necessary for not only the component transformation process once a component match is selected, but more importantly for the final generation of Ada code performed by the translator.

The last assumption specifically applies to the order Ada procedure parameters are declared in the Ada specification. That order must be identical to the order the corresponding parameters are declared in for the PSDL specification.

b. Generic, ADT, and User Defined Type Definitions

There are also several assumptions that pertain to how a generic, ADT or user defined type may be defined. These assumptions are outlined below and then discussed in more detail.

- All generic identifiers must be defined as Ada types. They may not reference user defined types or other generics.
- A user defined type also must be defined as an Ada type. Its one difference from generics is that if a user defined type is defined as an Ada array, the components of that array may be defined as generic types or other user defined types.
- All generic parameters must be referenced by at least one of the component's input or output parameters. Otherwise instantiation is not possible and any attempt to match that software base component will fail.
- Except for the abstract data type (ADT) with the same name as the *type* component that defines it, all of the other *type* component ADTs are visible only within that *type* component.

The only types definitions allowed for generic parameters are Private, Discrete, Array, Digits, Delta, Range, and Access. These type definitions must be specified explicitly. Neither another generic parameter nor a user defined type can be referenced by a generic parameter.

There are actually two categories of user defined types. A *type* component has a given name that identifies the *primary* abstract data type the component describes. However, it may also have some abstract data types, visible only to elements within the *type* component, that follow the same rules for definition as the primary ADT. For instance, the *type* component for Ring in Figure 23 makes Ring the primary ADT by using it to name the component. The component also has a *secondary* ADT called Direction that must be defined according to the rules established for any user defined type. ADT *operator* input and output parameters within Ring may be defined as the

type Direction, but parameters of operators outside the Ring component may not. This is a current limitation of this implementation. Ultimately, these secondary ADTs should be visible to any component with access to the *type* component containing secondary ADTs. These secondary ADTs are also discussed in Section IV.A.4.e. Section IV.A.4.a and Appendix A also discuss and provide examples of user defined types.

c. Input and Output Parameter Type Definitions

Certain assumptions have been made for specifying what is acceptable for input and output parameter type definitions. They are as follows:

- Input and output parameters in an *operator* component may be defined in terms of the component's generic types (software base components only), other user defined types in the prototype specification (this obviously excludes software base components because they are stored with their component PSDL specification and not the prototype PSDL specification), or Ada types.

- ADT operator input and output parameters in a *type* component may be defined in terms of the component or ADT operator's generic types (software base components only), other user defined types in the prototype specification (this again excludes software base components), the *type* component's primary ADT, secondary ADTs, or Ada types.

d. Query by PSDL Specification

Finally, the prototype PSDL specification for a query PSDL specification must include specifications of all user defined types referenced by the PSDL query specification.

4. Using the PSDL Grammar With User Defined Types

PSDL specifications for *type* components (also referred to as user defined types) can have one or more ADT in them. The ADT with the same identifier as the *type* component name is considered the primary ADT. Other PSDL specifications may use input or output parameters that reference that primary ADT. In order to be able to recognize the Ada type of parameters that reference user defined types, storing the Ada

type of an ADT became necessary. Within the PSDL grammar, the non-terminal *type_decl* is used to house this information. This was not the original use envisioned for *type_decl* by the original abstract model for the PSDL grammar, but seems to meet the current requirements of type matching without incurring negative side effects.

5. CAPS Interface Requirements

When a PSDL query is initiated from within the CAPS graphic or syntax-directed editor, two files must be present in the current working directory of the prototype designer. The first file is the PSDL specification for the entire prototype and its file name must be *prototype_psdl_spec.txt*. The second file is specific to the query component and its file name must be either *type_psdl_spec.txt* or *operator_psdl_spec.txt* depending on whether it is a *type* or *operator* component respectively. The reason the prototype PSDL specification file is necessary is to facilitate the identification of the Ada type of any user defined types in the prototype that the query component may reference. In order to do type matching, all query component type definitions must be resolved to their corresponding Ada type.

6. Program Flow, Source Code Files, and Data Files for Signature Encoding and Type Matching

The program flow is fairly similar for *operator* and *type* components, as well as much of the source code and data files. First, the query component input and output signatures are encoded by using the information in the prototype designer's current working directory, in the files *prototype_psdl_spec.txt* and either *type_psdl_spec.txt* or *operator_psdl_spec.txt*. The encoded signatures are then stored in the file *Signature.dat*. Next, the type matching algorithms read *Signature.dat* and search through the CAPS software base, looking only in the appropriate partitions and checking software base

component signatures against the query component. All final valid matches are returned in order of closest to farthest match.

a. Operator Component PSDL Query

Here is a diagram of the process flow and files involved with an *operator* component PSDL query:

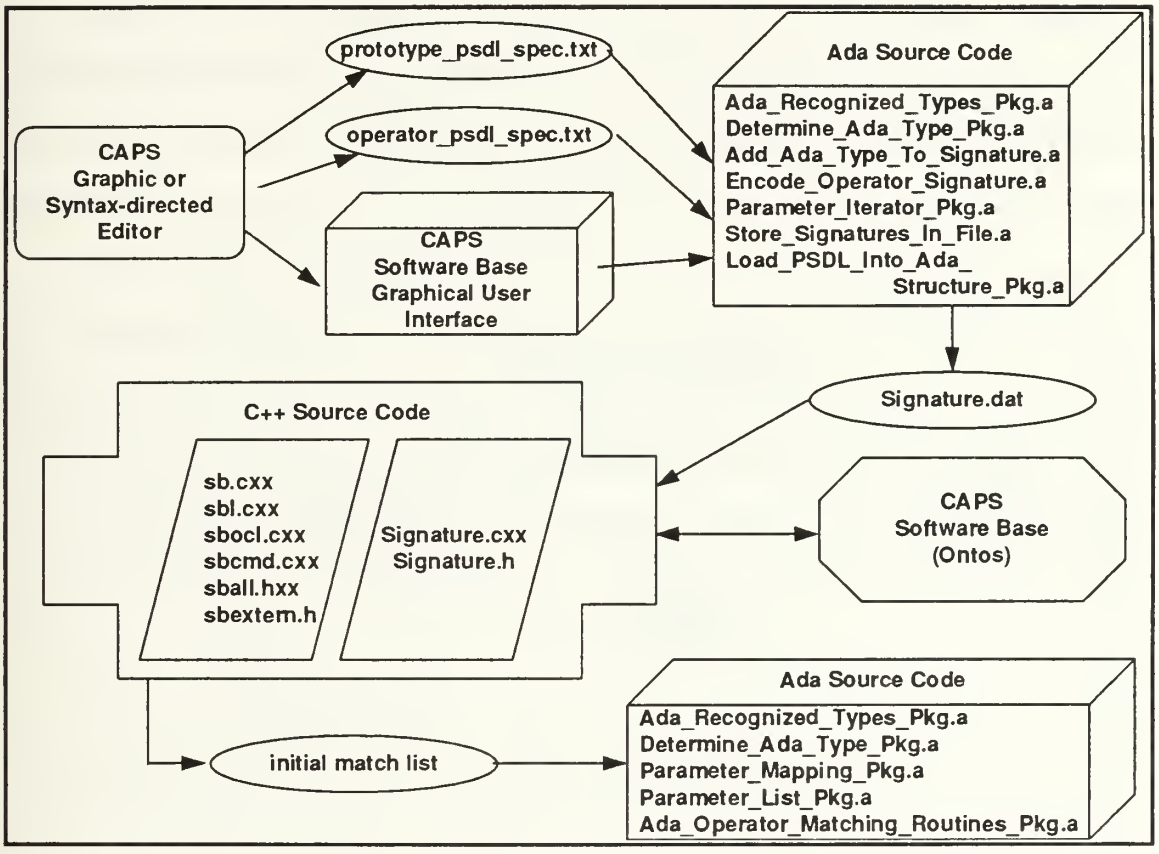

Figure 19 - Operator PSDL Query

Ellipses represent data files. Three dimensional rectangles and cubes represent Ada code and the object shaped like a plus represents C++ code. Although this diagram is intended to portray the process flow for an *operator* component PSDL query, the two way arrows between the C++ code and the CAPS Software Base illustrate that the

process is very much the same for component storage as it is for component retrieval. The only real difference occurs in the C++ code located in the left parallelogram.

b. Type Component PSDL Query

The *type* component code incorporates the code for the *operator* component query because *type* components can contain operators. It is, however, installed in different packages in order to incorporate the necessary changes and additions directly into the code. Here the *Signature.dat* file contains the aggregate signature of all the *type* component's individual operator signatures. Some additional Ada and C++ files are added to deal with the *type* specific signature encoding and matching routines. Here is a diagram of the process flow and files involved with a *type* component PSDL query:

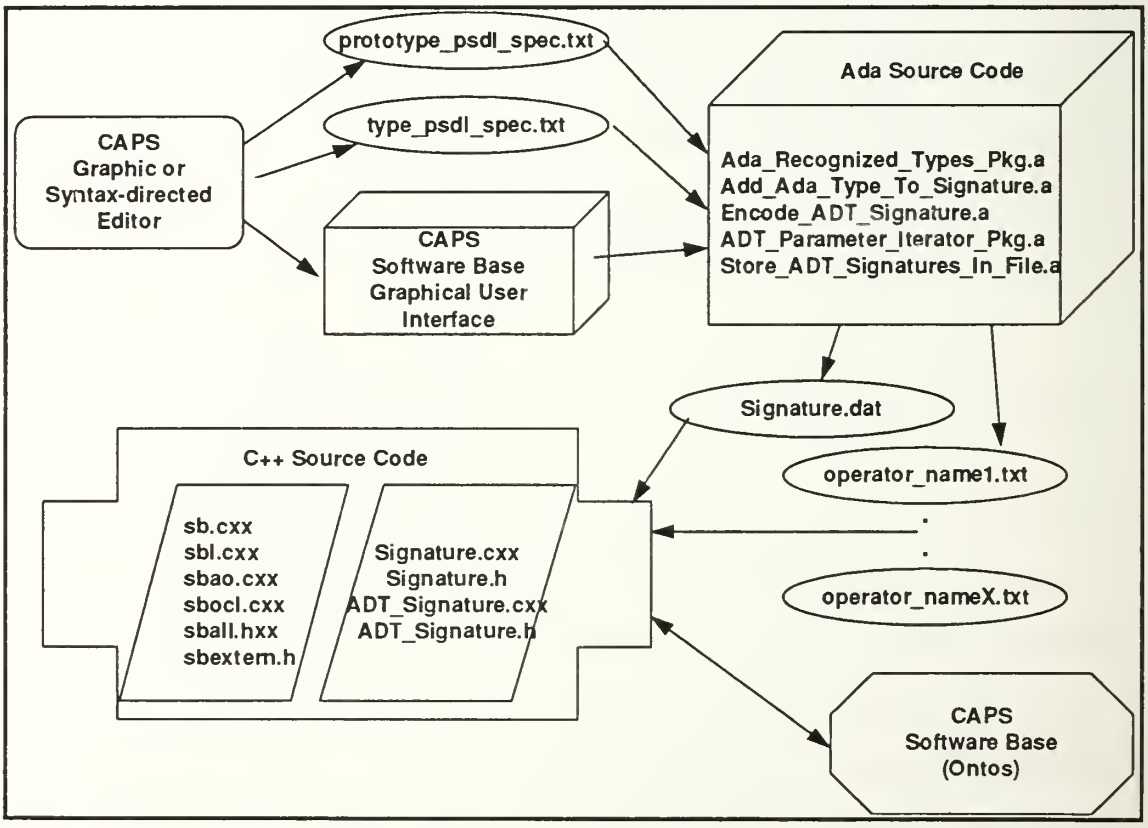

Figure 20 - Type PSDL Query

The major difference from the *operator* query is that the final routines for Array matching and generic instantiation validation are not available in this implementation. Another difference to note here is that separate signature files are created for the *type* component's individual operators during the signature encoding process. The names of these data files are the operator names concatenated with ".txt" as the file extension. During the component storage process these files are used to load the signatures of the *type* component individual operators into the software base. During component query and retrieval, these signature files are used to ensure that mappings of individual operators are valid matches.

7. The PSDL Ada Data Structure

An abstract data type (ADT) was developed in Ada that allows a PSDL specification to be represented as an Ada data structure. [Bayr91] This ADT is used extensively by the Ada packages that perform signature encoding. Occurrences of this ADT, also called a *PSDL type*, represent a particular PSDL program or specification. There are a host of Ada packages that make up this ADT including *psdl_io*, *psdl_id_pkg*, *psdl_program_pkg*, *psdl_component_pkg*, *psdl_concrete_type_pkg*, *type_name_pkg*, and *type_declaration_pkg*. The ADT provides a data structure in which to store all of the elements of a PSDL program (i.e., operators, data streams, and their associated attributes) along with the valid operations that allow you to access the ADT elements. The *psdl_io* package allows you to perform gets and puts on the data structure. Utilizing the "get" procedure you can pass a PSDL source file (i.e., the text of a PSDL specification) to the "get" procedure, it parses the file and subsequently loads the ADT data structure, and then specific PSDL information can be accessed via the ADT operations.

8. Ordering Retrieved Components

Whenever more than one software base component match is found for a query component, it is desirable to return a listing of matched software base components in an order that indicates which components were the better or closer match.

a. Operator Component Match Ordering

Because the number of input parameters must match exactly for a valid match, that attribute is not useful for ordering matches. The first criterion for ordering, therefore, is the difference in the number of output parameters between the query component and each software base component that is a valid match. The second ordering criterion is the type closeness difference. The type closeness for the software base components is the sum of the type closeness of each input and output parameter to the corresponding parameters in the query component. So, for example, all matched software based components with the same number of output parameters as the query component are listed first followed by all matched software based components that have one more output parameter than the query component, and so on. Within a group of software base components that have the same number of output parameters, the ordering is then based on type closeness with software components with a type closeness of 0 listed before those with a type closeness of 1 and so on.

b. Type Component Match Ordering

The first criterion for ordering *type* component matches is the difference in the number of ADTs between the query component and each software base component that is a valid match. The second ordering criterion is the difference in the number of operators. So, for example, all matched software based components with the same number of ADTs as the query component are listed first followed by all matched software based components that have one more ADT than the query component, and so

on. Within a group of software base components that have the same number of ADTs, the ordering is then based on number of operators with software components with the same number of operators as the query component listed before those with one more operator than the query component and so on. Attributes such as the total number of operator input and output parameters and individual operator signatures cannot be used in the ordering scheme. The reason for this is although an operator mapping has been found for the *type* component that guarantees a valid match, that mapping could be one of many possible mappings and the correct or best mapping is not known at this point.

9. Ada/C++ Interface Issues

Because considerable information is passed back and forth between Ada and C++ programs during signature encoding and matching, interface issues were explored that would allow a) Ada programs to call C++ programs, b) C++ programs to call Ada programs, and c) Ada and C++ programs to share parallel data structures. All the technical problems were resolved to accomplish these three goals. However, one technical obstacle proved too cumbersome to surmount. Although a C++ program can call an Ada program, Ada programs are really designed to be the main running program and so a dummy Ada program had to encapsulate the C++ program for everything to work. Rather than implement in this manner, we chose to have Ada and C++ programs pass information via text files.

10. Complexity Issues in Matching ADT Operators, Array Components, and Validating Generic Instantiations

While initial ADT matches are obtained by using the aggregate input and output signatures of their *operators*, final match confirmation cannot be done without finding a mapping between the query and software base component ADT *operators* in which the individual signatures of each query and software base component ADT *operator* pair

match. Given X query component and Y software base component ADT *operators*, the worst case number of combinations to explore before either finding a valid mapping or determining that none is possible would be on the order of $\binom{Y}{X}X!$ Although this is essentially a combinatorial problem which could have a big impact on processing time, the algorithm for finding a valid mapping is able to do considerable pruning and short circuiting of the search tree thus significantly reducing the actual order of complexity to within reasonable processing time limits. This same algorithm is used for matching Array components and during the validation of a generic instantiation.

11. Encoding Input and Output Signatures

Separate encoding schemes for input and output signatures have been developed that can be used with both *operators* and *types*. Input signatures employ a *downward* encoding scheme so that, for a particular parameter type, the region representing the parameter type and all its sub-regions are encoded into the signature. One disadvantage to *downward* encoding is the possibility of false matches, discussed in Section IV.C.g. Output signatures, on the other hand, employ an *upward* encoding scheme whereby, for a particular parameter type, the region representing the parameter type and all its super-regions are encoded into the signature.

One subtle point to highlight with *operator* input signatures is that there is never any concern about the software base component having more input parameters than the query component. The reason is because the initial syntactic matching filter described by the basic *operator* rules in Section IV.B.1 will eliminate any software base component with a different number of input parameters than the query component from consideration prior to the employment of signature matching filters [McDo91]. This is important because it provides an opportunity to use the *upward* encoding scheme for

both *operator* input and output signatures. The problem is that we still cannot use the *upward* encoding scheme for *type* input signatures.

Type input signatures are aggregates of all the input parameters of the *type* component's *operators*. Because a software base *type* component can have more *operators* than a query *type* component, it is possible for the software base *type* input signature to contain more input parameters than a corresponding query *type* input signature. If an *upward* encoding scheme is employed, we run into problems. The best way to demonstrate why is through the following examples using an *upward* encoding scheme where Region 1 is Positive, Region 2 is Natural, and Region 6 is Integer. The first example we look at is of two output parameters and the software base component output parameter is type Positive and the query component output parameter is type Integer:

	Region 1	Region 2	Region 6
sbc	1	1	1
qc	0	0	1

Here we see that the query component signature is a subset of or "fits inside" the software base signature. This is what we want because encoding any additional information (i.e., extra output parameters) into the software base component signature will not affect the match. Let us now look at input signatures using the *upward* encoding scheme for two components where we have a software base component input parameter of type Integer and a query component input parameter of type Positive:

	Region 1	Region 2	Region 6
sbc	0	0	1
qc	1	1	1

The first important thing we notice is that in order to compare input signatures when using the *upward* encoding scheme, we must now try and fit the software base component input signature into the query component input signature, the reverse of what occurs with output signatures. This is the root of the problem for *type* component input signatures encoded with the *upward* encoding scheme. The two input signatures in the above example do match, but as soon as an extra software base component input parameter is introduced, that software base component input signature can no longer be guaranteed to fit inside the query input signature. This is an undesirable situation in which a valid software base candidate component would be incorrectly eliminated from further consideration because of a failure to recognize that the input signatures match.

V. COMPONENT INTEGRATION

Once a user has selected a particular software base component that matches the provided PSDL query specification, the software base component must be integrated into the user's prototype working directory. An integration mechanism has been developed for *operator* components. The integration of *type* components is not implemented in this thesis.

A. THE PURPOSE OF COMPONENT INTEGRATION

The CAPS software base is a library of reusable software. It includes PSDL specifications, Ada specifications, and Ada implementations for all its stored components. The purpose of component integration is to treat the software base as a standard Ada library and use the Ada "with" clause to deal with software base components as library units [Ada83]. This way the software base source code is protected from any external modifications. Hooks from the user's prototype must be provided that reference and provide access to the Ada specification that corresponds to the selected software base PSDL specification, and these hooks are implemented by the Ada "with" clause. A *wrapper package* is built to incorporate these hooks. The file name of the wrapper package is always *operator_name_Pkg.a*.

In order to successfully integrate the selected component into the user's working directory, the generation of the wrapper package will normally require some degree of *transformation* of the software base component. That is because the query component and the software base component may have numerous differences. Parameter names may be different, parameter types may match but may not have the same ordering in the software base and query components, query component user defined types may need to

be referenced, and type declarations may need to be made for special cases like composite types. Once again, the transformation is effected inside the wrapper package. No changes are made to the software base component.

Component integration for the CAPS software base was first implemented by Dogan Ozdemir. [Ozde92] Ozdemir provided integration implementations for software base components selected by browsing, keyword search, and PSDL query. This thesis provides an implementation of component integration for *operator* PSDL queries only. In doing so, it removes some limitations imposed by Ozdemir's transformation process. Those limitations include (1) an assumption that the order of input and output parameters between the query and software base components was the same, (2) partially built generic wrapper packages for which the user would have to edit the wrapper and enter the instantiations once the transformation was finished, (3) use of the Ada renaming declaration to make a call to the software base component *operator* which cannot handle incompatible parameter ordering, and (4) an assumption that user defined types would not be used to define input or output parameters. The major reason these limitations can be removed is due to manner in which this implementation collects the information necessary for component transformation. Ozdemir parsed the query and software base PSDL specifications which provided him with a fast and automated method for obtaining critical component data, but with great difficulties in automating a mapping for that data. This thesis provides a graphical user interface that presents the key component data for both components to the user and allows him/her to select the appropriate mapping, greatly simplifying the subsequent transformation. This transformation process will be discussed in more detail in the following section.

B. THE COMPONENT TRANSFORMATION PROCESS

The component transformation process is provided to adapt the software base component to the format required by the query PSDL specification. It consists of two main steps. First, a mapping between query and software base input and output parameters must be determined. And second, a wrapper package is created based on the mapping information. Figure 21 below graphically illustrates the component transformation process:

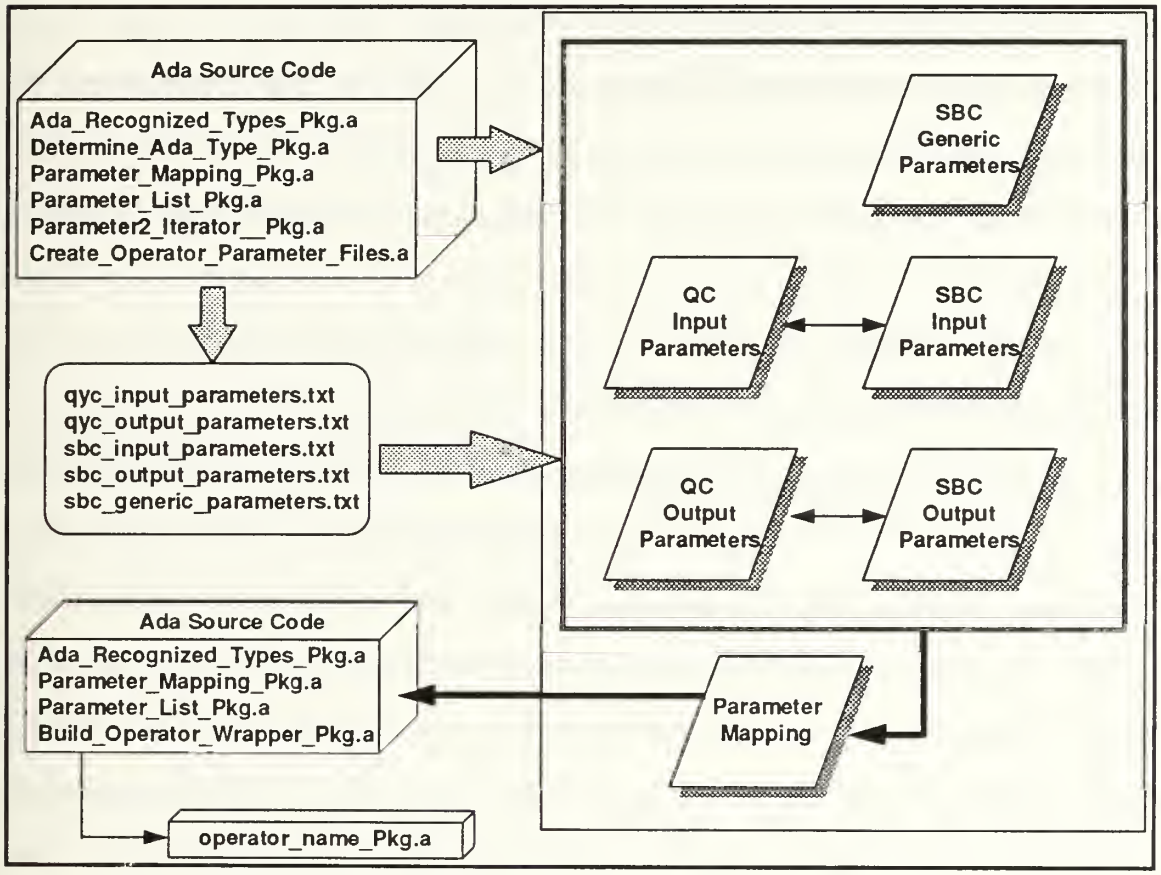

Figure 21 - Operator Component Transformation Process

The parallelograms depicted in Figure 21 represent TAE graphical user interface (GUI) display items [NASA91]. The GUI is discussed in detail in Section VI. User interaction with the GUI builds the necessary parameter map. The map is then used by an Ada

routine that generates a wrapper package that is stored in the user's prototype working directory. The *operator* component transformation process is now described in more detail.

1. Mapping The PSDL Specifications

As portrayed by Figure 21, Ada program parses both the query and software base component PSDL specifications, breaking the specifications down by parameter category (input, output, and generic) and storing that information in separate text files by category for each component. These text files serve as the basis for the user driven parameter mapping and are loaded into TAE display items for viewing and mapping selection. To simplify the mapping task for the user, all parameter types are defined in terms of their corresponding Ada type. So, for example, if the query component input parameter Q_In_Parameter1 was defined by the user defined type Index, and Index was defined as the Ada type Range, then the user would see "Q_In_Parameter1 : Range" as one of the selections in the query component input parameter TAE display item.

The query and software base component input and output parameter TAE display items are all interactive. The user creates the parameter map by selecting a parameter from a query input or output TAE display item and then selecting the corresponding parameter from the corresponding software base TAE display item. A type check is made to ensure that the two selected parameters match correctly. If they do not, an error message is generated and the user must make a new selection. If the two selected parameters match, then that selection is added to the parameter map and the two parameters are removed from the selection lists. A software base component parameter in the TAE display item that is preceded by the expression {G} informs the user that the parameter is defined by a generic parameter. When the user attempts to map a query component parameter to a software base component parameter, a check is made during

type matching to see whether the software base component parameter is defined by a generic parameter. If it is, then there are two possible cases that the match process must deal with. The first case is that the generic parameter has been instantiated and for that case the match process must ensure that the query component parameter matches the instantiation. The second case is that the generic parameter has not been instantiated and so, as a byproduct of the match process, the generic parameter must be instantiated by the query component parameter.

Unlike the input and output parameter TAE display items, the software base component generic parameter and parameter mapping TAE display items are passive. The user can view their contents, but does not interact with them. The parameter mapping TAE display item is updated as each selected parameter pair is validated as matching correctly.

Although automating the parameter mapping process to eliminate user interaction would speed up the integration and transformation of selected components, without a semantic matching capability it is an unrealistic goal. It is quite possible that a selected software base component may have numerous valid mappings with the query component. One of the limitations of syntactic matching is its inability to use semantics to determine the appropriateness of a particular mapping. For example, suppose we have the following PSDL specifications for input and output parameters of the query and software base component:

Query Component Parameters:

Input

Costs : INTEGER,

Revenues : INTEGER

Output

Profits : INTEGER

Software Base Component Parameters

Input

Sales : INTEGER,

Expenses : INTEGER

Output

Income : INTEGER

Figure 22 - Effect of Semantics on Automating Parameter Mapping

Examining the two sets of input parameters in Figure 22, we see that either of the following two mappings is syntactically correct:

Mapping 1:

Costs → Expenses

Revenues → Sales

Mapping 2:

Costs → Sales

Revenues → Expenses

Figure 23 - Possible Input Parameter Mappings

Figure 22 provides a simple example of specifications for the calculation of an organization's profits based on its costs and revenues. We know that the desired mapping from Figure 23 is Mapping 1, but that knowledge is based on understanding the semantics behind the parameter identifiers Costs, Expenses, Revenues, and Sales. Syntactic matching does not have access to this semantic knowledge and therefore views both mappings as equally valid. This problem is extended to the selection of the correct instantiation in the case of a generic software base component. Therefore, we have incorporated user interaction in the mapping process to provide the semantic reasoning that would otherwise be unavailable with automatic component integration.

2. Generating the Wrapper Package

The wrapper package serves as the bridge between the software base component and the user's PSDL specification. The transformation or translation of the software base component PSDL specification as prescribed by the query component PSDL specification is embedded into the wrapper package. Important aspects of the transformation process for generating a wrapper package are described below. Examples of wrapper packages generated for generic and non-generic software base components are provided in Appendix C.

a. Incorporating the Ada With Clause

The Ada "with" clause provides access to the software base component library unit. If user defined types are present in the query component PSDL specification, then other library units must also be referenced. A user defined type is defined in the prototype PSDL specification as a separate *type* with its own Ada implementation. The library unit that represents the Ada implementation of the user defined type is what must be accessed using a "with" clause in order to make the user defined type visible inside the wrapper package. Using Figure 27 as an example, let us suppose that the query

component PSDL specification for the *operator* Pick_A_Card has a parameter that is defined by a user defined type named Card_Deck. Based on implementation naming conventions, we know that the Ada implementation of Card_Deck must be called Card_Deck_Pkg. Therefore, during the generation of the wrapper package, the statement "with Card_Deck_Pkg;" will be included and any reference to the user defined type will be embedded as "Card_Deck_Pkg.Card_Deck" in the wrapper package. This is done for all user defined types referenced in the query component PSDL specification.

b. Parameter and Type Declarations

Parameter declarations are first made in the subprogram specification. [Ada83] The Ada language only allows predefined types to be used within parameter declarations. A predefined type may be either a type predefined by Ada or a user defined type. If a parameter type in the query PSDL specification is either a predefined Ada type or a user defined type then its incorporation into the wrapper package when creating the *operator* specification is very simple. Either it is used exactly as defined in the case of an Ada predefined type, or a reference is made to the package that defines it as described in Section V.B.2.a above in the case of a user defined type. However, in some cases, the query component parameter type may not be an Ada predefined type or a user defined type, but may actually be defined by the query PSDL specification. Array types provide a good example. Let us look at the *operator* Bubble_Sort in Figure 30. It provides a complete type definition for the parameter The_In_Array. However, this definition cannot be used in its exact form when writing the subprogram specification for Bubble_Sort into the wrapper package. Instead, a *type declaration* must be embedded in the wrapper package defining the type of The_In_Array before the Bubble_Sort subprogram specification is written. The Bubble_Sort subprogram specification will then reference the type declaration. As a naming convention, the identifier for the type

declaration will be the name of the relevant parameter concatenated with the base type of the type declaration. If the base type of the type declaration was Array, then "_ARRAY" would be concatenated to the name of the parameter that necessitated the type declaration. A snapshot of the wrapper package containing the necessary type declarations and subprogram specification for Bubble_Sort is provided in the following figure:

```
type The_In_Array_ARRAY is array (Index_Pkg.Index) of CHARACTER;
type The_Out_Array_ARRAY is array (Index_Pkg.Index) of CHARACTER;
procedure Bubble_Sort (The_In_Array  : in  The_In_Array_ARRAY;
                       The_Out_Array : out The_Out_Array_ARRAY);
```

Figure 24 - Example of Required Type Declaration

c. Procedure Calls to Invoke Software Base Component Operators

Ada provides two methods to call a procedure defined in the specification of a package. [Ada83] The first and more direct method is the *renaming declaration.* A renaming declaration allows you to give a procedure in a referenced external package a new name and then use the new name locally to access the external procedure. This could be employed in conjunction with generating a wrapper package by renaming the procedure corresponding to the software base component *operator* with the name of the query component *operator.* All local calls to the query component *operator* procedure invoke the software base *operator* procedure which is exactly the outcome desired. However, the renaming declaration is subject to stringent constraints that require the renaming declaration to use the pre-established ordering of parameter declarations in both the software base and query component *operators.* With this restriction, we run into two potential problems. First, the order of the two sets of parameter declarations

may not match up by type, in which case the Ada compiler will reject the renaming declaration. And second, the order of the two sets of parameter declarations may not match up semantically (as discussed in Section V.B.1), in which case the Ada compiler will accept the renaming declaration, but the logic of the generated wrapper will be faulty (i.e., we could have Costs mapped to Sales and Revenues mapped to Expenses as described in Figure 23).

Because we cannot assume that the order of parameters in the software base and query component *operators* will always be exactly what is required (in fact the ideal of a perfect match will likely be very rare), a second less direct method is employed to call the procedure associated with the software base component *operator*. This method requires implementing the procedure that defines the query component *operator*. Since an implementation is involved, the wrapper package must be extended from a package specification to the inclusion of a package body as well. It is within the package body that the implementation of the query component *operator* is defined. This definition consists of two primary parts; the subprogram implementation and, within that implementation, the call to the software base component *operator* procedure. Incorporating the subprogram implementation into the wrapper package body is identical to the previously described process for handling the subprogram specification with only minor formatting differences. Calling the software base component *operator* procedure is the second important part of the subprogram implementation. Because the parameter mappings have already been established by the user, its incorporation into the wrapper package is straightforward. If the software base component is non-generic then the software base component package name is used as a prefix to the procedure call. If the software base component is generic then the name of the instantiation of the software base component package is used as a prefix to the procedure call. The only factor that

complicates incorporating the procedure call to the software base component *operator* is the case in which the software base component has more output parameters then the query component. That situation is discussed in the next section.

d. Excess Output Parameters in the Software Base Component

The software base component is allowed to have a greater number of output parameters then the query component. If that is the case, then *dummy variables* must be created to be used as actual parameters when making the procedure call to the software base component *operator*. Incorporating the dummy variable declarations into the wrapper package body is straightforward unless the parameter type of the unused output parameter is not a predefined Ada type. If not, then a type declaration may have to precede the variable declaration. One of two cases is possible when the unused output parameter is not a predefined Ada type. In the first case the unused output parameter is defined by a generic parameter. If that is the case, then the generic parameter has been instantiated with a query component parameter type that is either a predefined Ada type, a user defined type, or was defined with a type declaration. The actual parameter used for the instantiation is used to define the unused output parameter for the first case. The second possible case is that the unused output parameter is defined by the software base component's PSDL specification. In that case, a type declaration must precede the variable declaration for that unused output parameter. The type declaration is created in the same manner described in Section V.B.2.b above. The name of each unused software base component output parameter is concatenated with "_DUMMY" to produce an identifier for its corresponding dummy variable.

e. Generic Instantiation

If a software base component is generic then it must be instantiated within the wrapper package. This is straightforward because the instantiations have already been defined during the parameter mapping process described in Section V.A. The actual parameters used in the instantiation will depend on how the type of the query component parameter responsible for the instantiation is defined. Thus, the actual parameter for the instantiation could be a predefined Ada type, defined by a type declaration, or a reference to a user defined type. The naming convention utilized for the instantiation attaches "TMP_" to the front of the query component *operator* name and "_PKG" to the end to produce the instantiation name. So, for example, if the query component *operator* name is Bubble_Sort then the name of the generic instantiation of the corresponding generic software base component would be TMP_Bubble_Sort_PKG. Examples of all aspects of generating a wrapper package are provided in Appendix C.

VI. GRAPHICAL USER INTERFACE

The original CAPS software base graphical user interface (GUI) was developed by Dogan Ozdemir. [Ozde92] This thesis makes two primary changes to Ozdemir's GUI. First, all software base maintenance routines are removed from the GUI. Second, a new interface is provided for the integration of software base *operator* components.

A. SOFTWARE BASE MAINTENANCE ROUTINE REMOVAL

Maintenance routines for the CAPS software base include adding, deleting and updating components and language libraries. While these are important functions necessary for the successful management of the CAPS software base, we do not believe that a user working to develop a prototype should have access to these capabilities. A user should only be concerned with acquiring reusable components from the software base. The actual administration of the software base should be a separate module with access limited to the software base manager. Therefore, all routines that relate to the management of the CAPS software base have been removed from the GUI that supports the prototype developer.

B. OPERATOR COMPONENT INTEGRATION

The only major addition to Ozdemir's GUI is the inclusion of a GUI panel for the parameter mapping necessary for the integration of a software base *operator* component. [Ozde92] All other GUI panels were designed and implemented by Ozdemir. Minor modifications were made to two of Ozdemir's GUI panels. The Component selection which allows a user to add or update software base components was removed from the Main Menu Panel. And the Delete selection that allows a user to delete software base components was removed from the Select Panel. Some modifications were also made to

some of the Ada code that Ozdemir added to the TAE generated Ada code [NASA91]. These modifications pertain to the addition of signatures, type matching, array matching, and generic instantiation validation to the PSDL query process.

The following figure illustrates the new parameter mapping panel:

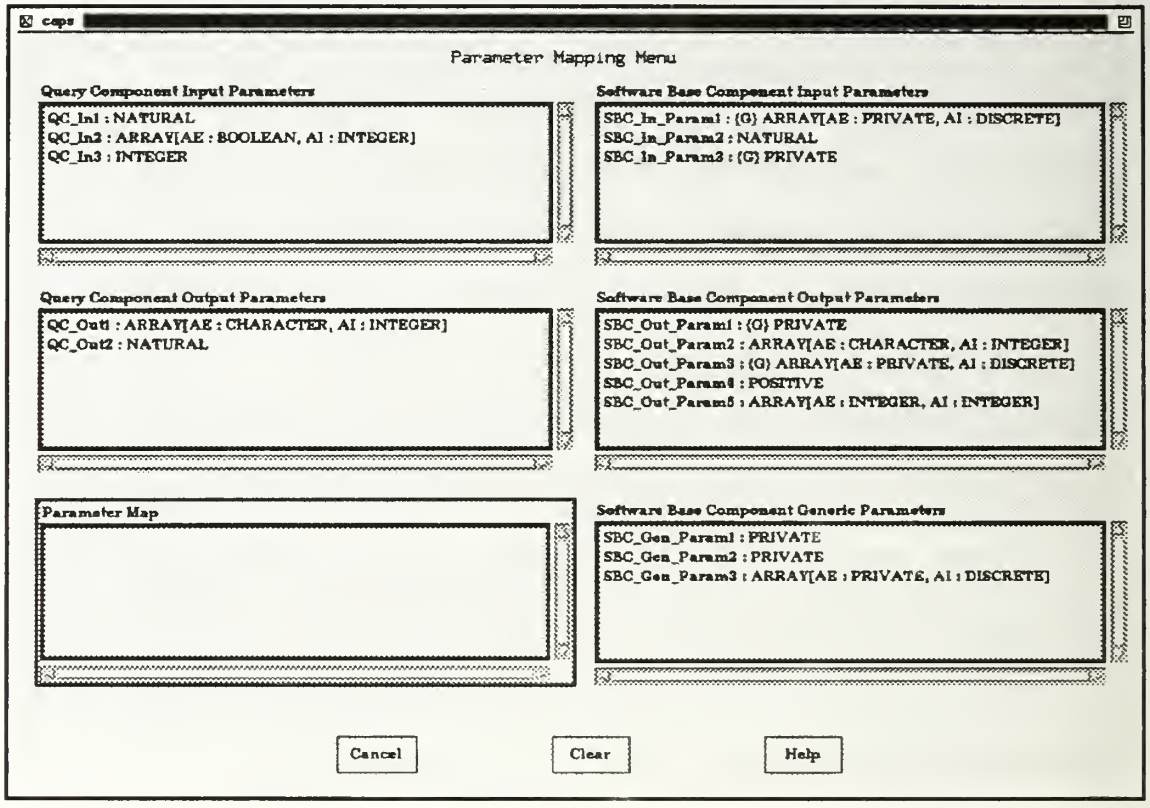

Figure 25 - Parameter Mapping GUI Panel

A detailed description of how a user performs parameter mapping in the above Parameter Mapping Panel is provided in Section V.B.1. To summarize, the user moves the mouse over a query component input or output parameter and then clicks the mouse button to select that parameter. The process is repeated for a corresponding software base parameter. Once the software base parameter is selected, the two selected parameters are checked to ensure they match. A valid match will cause both selections to be erased from their respective TAE display items and written to the Parameter Map

TAE display item. An invalid match will cause a warning message to be displayed and the user will then have to modify his/her selections. Once all query input or output parameters have been selected, that particular query TAE display item and corresponding software base TAE display item will be dimmed. It is important that the user be able to freely select from either the input or output parameters without having the order of selection imposed in any form. The reason for this is that order of selection is important for correct generic instantiation.

The figure below shows the Parameter Mapping Panel after a valid selection of two parameters from Figure 25 is made:

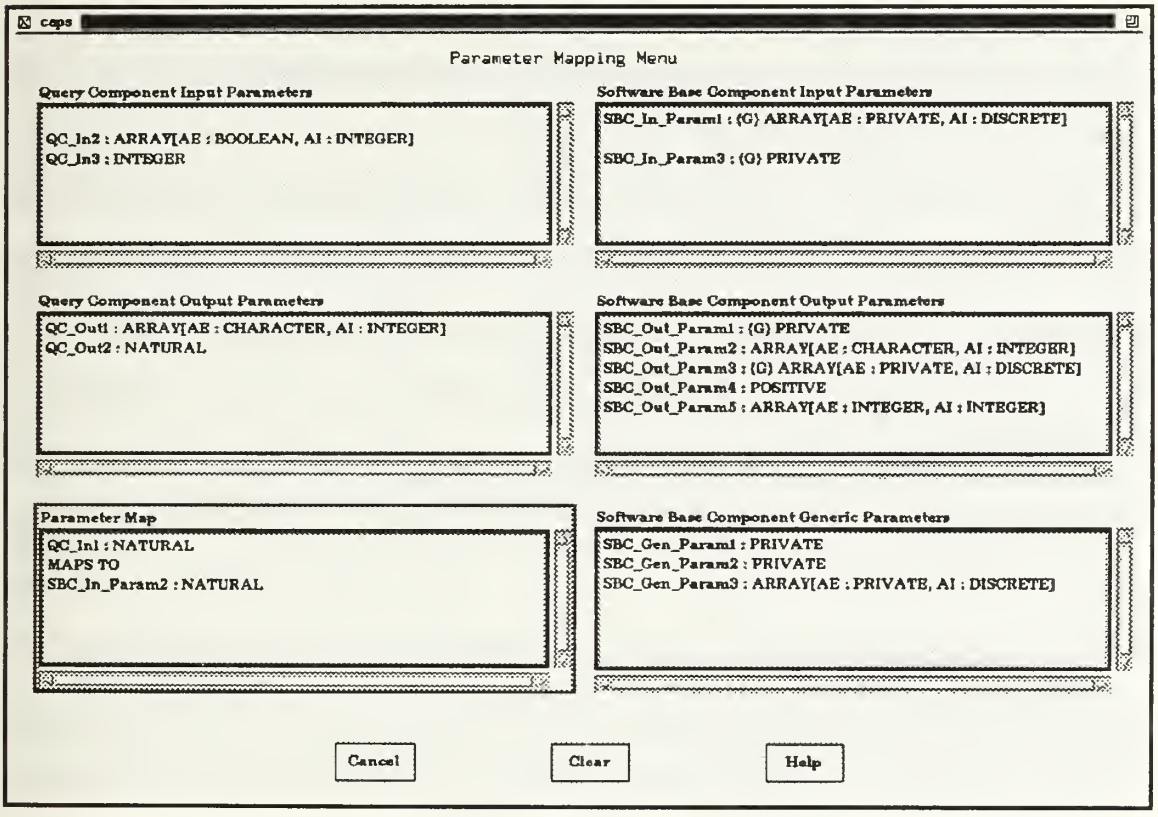

Figure 26 - Parameter Mapping Example

Note that two selections have been removed from their corresponding TAE display items in Figure 25 and their mapping has been recorded in the Parameter Mapping Tae display

item (Figure 26). When all query component parameters have been mapped successfully, generation of the wrapper package and wrapper package body is automatically initiated. The user is returned to the main menu upon completion of wrapper generation. If the selection of a particular mapping does not instantiate all the generics then an error message will be display and the user will be prompted to try another mapping. In some cases the user may select an incorrect partial mapping that will not allow any of the remaining parameters to be matched. In this case, the Clear button can be used to clear the partial mapping and allow the user to start the parameter mapping process from the beginning. The Cancel button allows a user to terminate parameter mapping at any point. The user will then be returned to the previous menu. Finally, the Help button provides information on the parameter mapping process and all selection buttons in the panel.

One limitation has been discovered with TAE selection lists. They are limited to a maximum of 80 characters. This could lead to problems when mapping input and output parameters. Some abbreviated naming conventions have been adopted to minimize this limitation. First, {G} is used to signify that a software base component parameter is defined by a generic. And second, **AE** and **AI** are used as abbreviations for ARRAY_ELEMENT and ARRAY_INDEX respectively. These abbreviations only apply to the information displayed in the TAE display items. PSDL specifications require the full identifier names. Another strategy to work around this limitation is to keep parameter identifiers short.

VII. CONCLUSIONS AND FUTURE RESEARCH

This thesis has described and implemented reliable automated assistance for finding reusable components from a large repository of existing software. Starting with the existing CAPS software base implementation originally developed by McDowell and improved by Ozdemir, this thesis has further improved the component retrieval and integration capabilities of the CAPS software base by (1) implementing parameter type matching, (2) extending the filtering process underlying the software base schema to further partition the component retrieval search space, (3) providing generic instantiation validation as part of the *operator* component matching process, and (4) removing many of the limitations from the previous *operator* component integration process [McDo91, Ozde92]. The task of automating the entire process from component retrieval through integration has been described and successfully implemented for *operator* components.

Perhaps the most significant lesson learned at the end of this thesis is the importance of a *formally defined model for software reuse*. A formal strategy for designing and engineering software base components specifically for reuse are crucial because the specifications of these components serve as the basis for component retrieval. Some of the issues that should be resolved and then formalized are discussed in Section VII.A below.

The development of this implementation has highlighted the important role semantics plays in the component parameter mapping process. Only by incorporating semantics can automatic component parameter mapping take place without the need for user interaction. While former limitations were removed, and accuracy and efficiency increased, this implementation is far from complete. There are still limitations imposed by this implementation as well as numerous areas in which the implementation can be

extended. The following sections identify those areas where improvements can be made and in some cases suggestions are provided for accomplishing those improvements.

A. DEVELOP A FORMAL MODEL FOR SOFTWARE REUSE

It is difficult to come up with a clean implementation of a component retrieval mechanism if there is no formal model defining the process of software reuse. That is because the software reuse model explicitly defines the boundaries of the problem. In the context of CAPS, this model must address the capabilities and limitations of the CAPS translator, the CAPS software base, PSDL, and the Ada language (for the current implementation). Issues such as the treatment of user defined types, composite types, constrained and unconstrained types, and what can and cannot appear in software base or query component specifications must be examined together, because decisions made for one issue will ultimately have some impact on most other issues. Suggestions for how to deal with some of these issues are outlined below and are further discussed later in Section VII.

1. User Defined Types

It appears that the cleanest treatment of user defined types (UDT) is to encapsulate them into abstract data types, declare them as Private types, and store them as *type* components in the software base. Thus, a data structure is provided as well as all operations required to gain access to and process the information embedded in the data structure. This will become powerful once this implementation is modified to allow software base components to reference UDTs within the software base (see Section VII.D). Perhaps a tool could be developed that would serve as a generic template for building common classes of UDTs.

2. Type Constraints

The issue of whether a parameter type is constrained or unconstrained becomes important when generating Ada code from PSDL specifications. Both the CAPS translator and the component integration process generate Ada code. One solution might be to provide constraint information within PSDL (see Section VII.F.5). User defined types could be used to provide prototype designer defined constraints. The safest solution is to require all constraints to be passed as instantiations of generic parameters. However, this approach may be more limiting.

3. Relating PSDL to Ada

Decisions need to be made regarding what kinds of Ada type declarations will be allowed in PSDL specifications. The rules will likely be different for software base and query components. For example, we have already specified that query components cannot have generic parameters. In some cases it may be helpful for PSDL parameter type names to be different than their corresponding Ada type names. Suggested conventions are made for the following types that should only appear in generic parameter declarations (i.e., not input or output parameter declarations):

- Ada type name = Private, PSDL type name = PRIVATE_TYPE
- Ada type name = Discrete, PSDL type name = DISCRETE_TYPE
- Ada type name = Enumeration, PSDL type name = ENUMERATION_TYPE
- Ada type name = Access, PSDL type name = ACCESS_TYPE

The following suggestions are made for other types that may or may not appear as generic parameter declarations:

- Ada type name = Array, PSDL type name = ARRAY_TYPE
- Ada type name = Digits, PSDL type name = DIGITS_TYPE
- Ada type name = Delta, PSDL type name = DELTA_TYPE
- Ada type name = Range, PSDL type name = RANGE_TYPE

Record types present their own unique concerns because Ada does not have any treatment for generic formal record types. The suggested solution is to allow query component Record parameter types to only be matched to a software base component generic Private type. In this manner, software base component Record structures can be encapsulated into abstract data types. This also eliminates a potential technical concern. The strong typing facilities of Ada treat two Records with identical structures as two different types. This could create problems with the compilation of automatically generated code.

B. POPULATE THE SOFTWARE BASE

Although commercial Ada software component libraries exist, and new software repositories within the Department of Defense are starting to come on-line, the CAPS software base is still very under-populated. Adding components to the CAPS software base is a labor intensive process. The primary reason for this is that a PSDL specification must be written for any Ada component added to the software base. In addition, certain limitations, like the inability to handle the "in out" parameter mode eliminate many candidate components.

One tool that would help resolve this problem is an automated translation tool that could take an Ada specification as input and generate the corresponding PSDL specification. Ideally, this translation tool could include some sort of conversion of Ada specifications that use "in out" parameters to an equivalent form that CAPS can use.

C. DEVELOP A SOFTWARE BASE MANAGEMENT GUI

It is necessary for a separate GUI to be developed for the management of the CAPS software base. Access to management functions such as language library creation or component add, delete, and update should be carefully controlled since these functions have a direct effect on the composition of the CAPS software base which in turn is utilized by a diverse group of users. The original GUI designed and implemented by Ozdemir already incorporates most of these functions, so the task of implementing a separate software base management GUI should not be a difficult one [Ozde92].

D. EXTEND USER DEFINED TYPES TO THE SOFTWARE BASE

Allowing components added to the software base to reference user defined types already in the software base will significantly increase the usefulness of the CAPS software base. So, for example, let us say that we are encoding the signature of an *operator* component as a precursor to adding that component to our software base. At some point we reach an input parameter In_Param of type Shipboard_Navigation. Shipboard_Navigation is not a predefined Ada type, and it is not defined as one of the components generic parameters. At this point, the current implementation would abort processing and declare a parameter type error. An improvement to this would be to search through the *type* components in the CAPS software base to see if Shipboard_Navigation has already been defined. Specifically, within the Operator_Component_Library, the Operator_Component_Dictionary could quickly be searched since it is indexed by component name. If Shipboard_Navigation was found then its corresponding Ada type could be extracted and provided to the signature encoding process. If Shipboard_Navigation is not found, then an error message should be generated and the attempt to add this component to the software base should be aborted.

E. REDUCE THE USER INTERACTION WITH PARAMETER MAPPING THROUGH A PRE-MAPPING FUNCTION

Automatic mapping selections should be performed, where possible, as an aid to the user's establishment of a parameter mapping scheme. These automatically selected mappings could be accomplished in cases where only one parameter mapping was possible for a particular query component parameter, or only one instantiation was possible for a particular generic parameter. This would reduce the number of mappings the user would have to select. It would also be helpful to provide a visual ripple-through effect for all instantiations. For example, suppose software base component parameters SBC_In1, SBC_In2, and SBC_Out1 are all defined by the generic parameter SBC_Gen_Param. As soon as the user makes a selection that instantiates SBC_Gen_Param, all software base component parameters defined by SBC_Gen_Param should be redisplayed with their parameter types updated to reflect the generic instantiation they are then locked into.

F. REMOVE PSDL SPECIFICATION LIMITATIONS

Several improvements can be realized by removing various requirements imposed by this implementation for writing PSDL specifications that are more restrictive than the PSDL language intended.

1. Allow Extra Generic Parameters

The current implementation will not process a PSDL specification that has uninstantiated generics during component integration. This includes generic procedures, which the current implementation does not recognize. Allowing generic procedures, even though they cannot be defined at present in PSDL, will expand the scope of the kinds of components admitted into the CAPS software base. Adjustments would have to be made during the integration process to allow a user to define any uninstantiated

generic within the generated wrapper so that the instantiation could be fully defined and implemented.

2. Make Secondary ADTs Visible to External Components

It is desirable for components to be able to reference a type declaration (secondary ADT - refer to Section IV.D.3.b) that was made in the specification of another component. This is similar to the concept of a user defined type. The difference is that with a user defined type, the name of the *type* component is the user defined type and can therefore be accessed directly. This may be a difficult concept to implement without having a big effect on processing time. The main reason for this is that there is no fast method for searching for secondary ADTs other than to iterate through all existing *type* components to see if the desired secondary ADT is defined there. One solution would be to create a separate dictionary of all secondary ADTs which would provide for direct access and greatly reduce search time. Referencing secondary ADTs is important because it imitates a capability inherent in the Ada language.

3. Expand Array Component Definition

The current implementation imposes restrictions on the definitions of Array components within a PSDL specification. Array components can be defined by any combination of user defined or predefined Ada types. However, for generic definitions, either both Array components must be defined by generic parameters, or else neither may be defined by generic parameters. Future implementations should remove this restriction.

4. Matching Composite Types

The current implementation does not go below one level when matching composite types. So, for example, if two Arrays are being matched, then their components (the first level) will be compared. However, if the element component is

itself a composite type (i.e., an Array of Arrays) then the Array components of the Array element component (second level) are not checked for a match. Future implementations should remove this restriction which will provide more accurate type matching.

The current implementation provides an algorithm for matching Array components down to one level (*Ada_Operator_Matching_Routines_Pkg.a*). That same algorithm can be utilized for any kind of composite type. The process of creating type declarations in the wrapper package has also been developed that allows for the predefinition of composite types before they are invoked in an Ada subprogram specification or implementation. This may prove to be useful when dealing with constrained types. A possible approach might be to specify constrained types as composite types with the sub-component(s) providing the necessary constraint information. See the following section for an example of this with the String type.

5. Matching Constrained Types

If an input parameter of a query component and a software base component is a constrained type like Range, the current implementation will validate that they match. However, it is possible that the constraint on the software base component input parameter Range is 1..50 while the constraint on the query component input parameter Range is 1..100. In reality, they do not match, nor is the query input parameter even a valid subtype. The key problem with going to a deeper level to match these types is that currently the information about the Range constraints is only contained in the Ada specifications. One possible solution to this would be to allow the Range constraints to be declared in the PSDL specification. For example, we could declare "X : Range [FROM : 1, TO : 50]" in the software base component PSDL specification. This information would not necessarily have to be encoded into the component input and output signature but could be handled by a filter that follows the signature matching

filter, much like Array matching at the component level is handled by this implementation.

A possibly better approach was alluded to in the previous section. A user defined type could be used as a composite type component to provide constraint information. Take String for example. It could be specified as

Input_Parameter : STRING [SIZE : some_UDT_name]

where SIZE would define the length of the String. The wrapper package would have direct access to this constraint information because it can access some_UDT_name_PKG.some_UDT_name.

G. EXTEND ARRAY MATCHING, GENERIC INSTANTIATION VALIDATION AND INTEGRATION TO TYPE COMPONENTS

Due to time limitations, Array type matching, generic validation, and integration of selected software base components was not implemented for *type* components. These capabilities should be included in future implementations. The underlying processes for their implementation will be very similar to the processes implemented for *operator* components. Because Array type matching does not deal with generics, that process can be incorporated exactly as it is currently implemented. Generic instantiation validation will have two extra steps to consider. The first is the mapping of the *type* component's operators. The algorithm for this is already provided in the implementation of a PSDL query for *type* components. The second additional step is that when iterating through all possible mapping combinations at the ADT operator level, a failure to find a successful match will potentially invalidate mappings found for previous operators and an alternate combination for the *type* component's generic parameters may have to be explored. This essentially contains one more level of complexity than does the process for *operator* components. *Operator* components had to deal with two levels of complexity. A

mapping for their generic parameters and a mapping for their non-generic parameters. *Type* components have both of those levels to deal with plus a third; the mapping of the *type* component's generic parameters. Similarly, the integration of a *type* component must be concerned with both this third level of complexity and having the user establish a mapping of the *type* component's operators prior to determining parameter mappings.

H. IMPROVE SPEED AND EFFICIENCY

Most of the effort that went into this thesis was to find a way to provide more accurate component retrieval. Little time was left to consider optimization. Initial tests were performed on a software base loaded with over 1,000 components. These components were automatically generated with varying numbers of input, output, and generic components. Many components were entered multiple times to force match processing on more significant numbers of matching components. One tested retrieval returned all 50 possible non-generic component matches from the 1,000 component software base in under a minute. Another retrieval designed to extract known generic components returned all 50 possible generic component matches in under three minutes. It is difficult to categorize these initial and limited results. For example, suppose we were looking for components that had four input parameters. Further suppose that of the next 5,000 components added to the software base, none will have four input parameters (i.e., all will have either more or less than four). In this case, due to the partitioning of the search base by the software base schema design, we will see no change in the retrieval time of the two test examples above even though they are searching a 5,000 component software base versus the original 1,000 component software base. More research needs to be done that can quantify retrieval patterns and expectations.

What is obvious, however, is where the processing bottleneck occurs in the retrieval mechanism implemented in this thesis. Signature matching is very fast. The aspect of the retrieval mechanism with the biggest payoff if optimized is *post-signature filtering*. This includes Array type matching at the Array component level and generic instantiation validation. One reason this is slow may be due to the need to access PSDL specifications for each software base component accessed by this filter. It also appears that there is unnecessary processing being performed in the generic instantiation validation. Currently, when examining all possible instantiation for a particular generic parameter, each query component parameter is included in the list of possibilities if it can instantiate the generic parameter. However, it may be the case that only each parameter type that is available to instantiate the generic parameter need be a part of the list of possible instantiations.

LIST OF REFERENCES

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

[AP91] Arango, Guillermo, and Prieto-Diaz, Ruben, *Part 1 - Introduction and Overview: Domain Analysis Concepts and Research Directions,* in Domain Analysis and Software Systems Modeling, IEEE Computer Society Press, 1991.

[Bayr91] Bayramoglu, Suleyman, *The Design and Implementation of an Expander for the Hierarchical Real-Time Constraints of Computer Aided Prototyping System (CAPS),* Master's Thesis, Naval Postgraduate School, September 1991.

[BBKM87] Burton, Bruce A., Aragon, Rhonda W., Bailey, Stephen A., Koehler, Kenneth D., and Mayes, Lauren A., *The Reusable Software Library,* IEEE Software, July 1987.

[BL91] Berzins, Valdis and Luqi, *Software Engineering with Abstractions,* Addison-Wesley Publishing Company, 1991

[Boeh87] Boehm, Barry W., "A Spiral Model of Software Development and Enhancement," in *Tutorial: Software Engineering Project Management,* IEEE Computer Society Press, 1987.

[Booc87] Booch, Grady, *Software Engineering With Ada,* Benjamin/Cummings Publishing Company, 1987.

[Booc87a] Booch, Grady, *Software Components with Ada*, The Benjamin/Cummings Publishing Company, 1987.

[BP89] Biggerstaff, Ted J. and Perlis, Alan J., *Software Reusability Volume I Concepts and Models,* Addison-Wesley Publishing Company, pg xv, 1989

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

[Endo92] Endoso, Joyce, "Business Issues Impede Software Reuse", *Government Computer News,* November 9, 1992.

[Epp90] Epp, Susanna S., *Discrete Mathematics with Applications*, Wadsworth Publishing Company, Belmont, California, 1990.

[FFN91] Frakes, W. B., Fox, C. J. and Nejmeh, B. A., *Software Engineering in the Unix/C Environment*, Prentice-Hall, 1991.

[Hoop89] Hooper, James W., *A Perspective of Software Reuse*, U.S. Army Institute for Research in Management Information, Communications, and Computer Science (AIRMICS), March 1989

[LBY88] Luqi, Berzins, Valdis and Yeh, Raymond T., "A Prototyping Language for Real-Time Software," *IEEE Transactions on Software Engineering*, October 1988.

[LK88] Luqi and Ketabachi, Mohammad, "A Computer-Aided Prototyping System," *IEEE Software*, March 1988.

[Luqi89] Luqi, *Software Evolution Through Rapid Prototyping*, IEEE Computer, May 1989.

[Manb89] Manber, Udi, *Introduction to Algorithms - A Creative Approach*, Addison-Wesley Publishing Company, 1989.

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

[MM91] McMullen, Barbara E., and McMullen, John F., "IBM & DARPA Set Up Reusable Software Library", *Newsbytes Inc.*, December 3, 1991.

[MS92] Maiden, Neil A., and Sutcliffe, Alistair G., "Exploiting Reusable Specifications Through Analogy," *Communications of the ACM*, April 1992.

[NASA91] NASA Goddard Space Flight Center, *Release Notes for TAE Plus*, ver. 5.1, April 1991.

[Onto91] ONTOS Inc., *ONTOS Object Database Documentation Release 2.1*, Burlington, MA, 1991.

[Ozde92] Ozdemir, Dogan, *The Design and Implementation of a Reusable Component Library and a Retrieval/Integration System*, Master's Thesis, Naval Postgraduate School, December 1992.

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

[Ritt89] Rittri, Mikael, "Using Types as Search Keys in Function Libraries," *The Fourth International Conference on Functional Programming Languages and Computer Architecture*, Imperial College, London, 1-13 September 1989.

[Royc70] Royce, Winston W., "Managing the Development of Large Software Systems," *Proceedings of IEEE WESCON*, August 1970.

[RT89] Runciman, Colan, and Toyn, Ian, "Retrieving Re-usable Software Components by Polymorphic Type," *The Fourth International Conference on Functional Programming Languages and Computer Architecture*, Imperial College, London, 1-13 September 1989.

[SLB92] Steigerwald, Robert A., Luqi, and Berzins, Valdis, "A Tool for Reusable Software Component Retrieval via Normalized Specifications," *Proceedings of the Hawaii Conference on System Sciences*, Koloa, Hawaii, January 7 - 10, 1992.

[SLM91] Steigerwald, Robert, Luqi, and McDowell, John K., "CASE Tool for Reusable Software Component Storage and Retrieval in Rapid Prototyping," *Information and Software Technology*, November 1991.

[SPS93] Software Productivity Solutions Inc., Melbourne Florida, 1993.

[Stei91] Steigerwald, Robert A., *Reusable Software Component Retrieval Via Normalized Algebraic Specifications*, Ph.D. Dissertation, Naval Postgraduate School, December 1991.

[WS88] Wood, M., and Sommerville, I., "An Information Retrieval System for Software Components," *Software Components and Reuse: special section of Software Engineering Journal*, Vol. 3, No. 5, September 1988.

[Your89] Yourdon, Edward, *Modern Structured Analysis*, Prentice-Hall, 1989.

BIBLIOGRAPHY

Booch, Grady, *Software Components with Ada,* The Benjamin/Cummings Publishing Company, 1987.

Dixon, Robert M., *The Design and Implementation of a User Interface for the Computer Aided Prototyping System,* Master's Thesis, Naval Postgraduate School, September 1992.

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

Gonzalez, Dean W., *Ada Programmer's Handbook and Language Reference Manual,* The Benjamin/Cummings Publishing Company, 1991.

Gough, K. J., *Syntax Analysis and Software Tools,* Addison-Wesley Publishing Company, 1988.

Huskins, James M., *Issues in Expanding the Software Base Management System Supporting the Computer Aided Prototyping System,* Master's Thesis, Naval Postgraduate School, June 1990.

Kelley, Al, and Pohl, Ira, *A Book On C,* Benjamin/Cummings Publishing Company, 1990.

Kernighan, Brian W., and Ritchie, Dennis M., *The C Programming Language,* Prentice-Hall, 1978.

Luqi and Berzins, Valdis, *Rapidly Prototyping Real-Time Systems,* IEEE Software, September 1988.

Pyle, I.C,. *The Ada Programming Language,* Prentice-Hall International, 1981.

Selby, Samule M., *Standard Mathematical Tables 23rd Edition,* CRC Press, 1974.

Self, J. *Aflex - An Ada Lexical Analyzer Generator,* ver. 1.1, Arcadia Document UCI-90-18, University of California, Irvine, CA, May 1990.

Sethi, Ravi, *Programming Languages - Concepts and Constructs,* Addison-Wesley Publishing Company, 1990.

Skansholm, Jan, *Ada From the Beginning,* Addison-Wesley Publishing Company, 1990.

Swan, Tom, *Learning C++*, SAMS, 1991.

Taback, D., Tolani, T., and Schmalz, R.J., *Ayacc User's Manual,* ver. 1.0, Arcadia Document UCI-85-10, University of California, Irvine, CA, May 1988.

APPENDIX A - USER DEFINED TYPE EXAMPLE

Here is an example of a simplified prototype PSDL specification that employs the user defined type (UDT) Card_Deck:

```
TYPE Card_Deck
SPECIFICATION
    Card_Deck  :  Range
END
IMPLEMENTATION ADA Card_Deck_Pkg END

OPERATOR Pick_A_Card
SPECIFICATION
    INPUT     The_In_Card    : Card_Deck
    OUTPUT    The_Out_Card   : Card_Deck
END
IMPLEMENTATION ADA Pick_A_Card_Pkg
```

Figure 27 - User Defined Type

The UDT Card_Deck is referenced by the *operator* Pick_A_Card. By checking the prototype specification for a *type* named Card_Deck, we see that Card_Deck is defined as the Ada type Range. The actual Ada implementation of Card_Deck, stored in Card_Deck_Pkg would look like this:

```
package Card_Deck_Pkg is
   type Card_Deck is Range 1..52;
end Card_Deck_Pkg;
```

Figure 28 - Ada Implementation of UDT

APPENDIX B - PSDL SPECIFICATION EXAMPLES

Many of the PSDL specifications in this section were written for Ada components developed by Grady Booch [Booc87a]. These PSDL specifications are not complete and only contain the information necessary to illustrate how to formulate PSDL specifications for software reuse. The examples in this appendix are designed to illustrate the assumptions and requirements described in Section IV.D.3.

A. PROTOTYPE PSDL SPECIFICATION

A prototype specification covers the entire system being prototyped. Thus it may have several *types* and *operators*. Here is an example of a prototype PSDL specification:

```
TYPE Ring
SPECIFICATION
        Ring      : PRIVATE,
        Direction : PRIVATE

        OPERATOR Rotate
        SPECIFICATION
            INPUT    The_Direction     : Direction,
                     The_In_Ring       : Ring
            OUTPUT   The_Out_Ring      : Ring
       END
END

OPERATOR Sort
 SPECIFICATION
        INPUT     The_In_List       : List_type,
        OUTPUT    The_Out_List      : List_type
END
```

Figure 29 - Partial Prototype PSDL Specification

Figure 29 depicts a *partial* prototype PSDL specification with just one *type* and one *operator* portrayed. It is used in conjunction with query components only. As explained

earlier, the prototype PSDL specification is important for all PSDL queries because the prototype contains all user defined types referenced by the PSDL query. Software base components are either *types* or *operators* and their corresponding PSDL specification is limited to a single *type* or *operator,* never an entire prototype. In Figure 29 the user defined type List_type is referenced. Although its definition is not provided, it would be defined as a *type*. Note that no generics are used because they are not allowed in query components. Note also that the primary ADT Ring is referenced by the ADT operator Rotate in its input and output parameters. The secondary ADT Direction is also referenced by an ADT operator input parameter. While other *operator* PSDL specifications within the prototype could validly reference Ring, they cannot reference the secondary ADT Direction because it is not visible outside the *type* Ring in this implementation. Because these secondary ADTs correspond to type declarations made in an Ada specification, ultimately they should become visible to any object that has access to the *type* component in which they are declared.

Here is another partially developed prototype PSDL specification:

```
TYPE Index
SPECIFICATION
      Index : RANGE
END


OPERATOR Bubble_Sort
SPECIFICATION
  INPUT    The_In_Array  : ARRAY [ ARRAY_ELEMENT : CHARACTER,
                                   ARRAY_INDEX   : Index]
  OUTPUT The_Out_Array : ARRAY [ ARRAY_ELEMENT : CHARACTER,
                                   ARRAY_INDEX   : Index]
END
```

Figure 30 - Partial Prototype PSDL Specification

Once again note that because it corresponds to query component PSDL specifications it contains no generics. Also note the use of the user defined type Index by the *operator* Bubble_Sort and the use of the mandatory identifiers ARRAY_ELEMENT and ARRAY_INDEX when specifying array components. One problem with trying to find a match for Bubble_Sort is that a generic component that could perform the required sorting normally requires a procedure passed in as one of its generic parameters that provides ordering information for the elements to be sorted. Because generic procedures cannot be handled by CAPS at this time, this limits the kinds of components available in the CAPS software base.

B. SOFTWARE BASE COMPONENT PSDL SPECIFICATION

We will look at two kinds of PSDL specifications for a software base component. One for a *type* component and one for an *operator* component. Because we are looking at software base components, generics can be included. Here is the PSDL specification for a software base *type* component:

```
TYPE Ring
SPECIFICATION
   GENERIC  Item : PRIVATE

        Ring      : PRIVATE,
        Direction : PRIVATE

        OPERATOR Insert
        SPECIFICATION
            INPUT     The_Item       : Item,
                      The_In_Ring    : Ring
            OUTPUT  The_Out_Ring  : Ring
        END

        OPERATOR Rotate
        SPECIFICATION
            INPUT     The_In_Ring      : Ring,
                      The_Direction  : Direction
            OUTPUT  The_Out_Ring       : Ring
        END
```

Figure 31 - PSDL Specification for *Type* Component

Note how Figure 31 is different from Figure 29 for the *type* Ring. Because it is a software base component, Figure 28 can use a generic type. Also note that the generic type as well as the ADTs are defined as Ada types. It would be illegal for either to try and reference another generic type or a user defined type. Here is the PSDL specification for a software base *operator* component:

```
OPERATOR Bubble_Sort
SPECIFICATION
      GENERIC Array_Type :   ARRAY [ ARRAY_ELEMENT : PRIVATE,
                                     ARRAY_INDEX    : DISCRETE]

      INPUT    The_In_Array   : Array_Type
      OUTPUT   The_Out_Array  : Array_Type
END
```

Figure 32 - PSDL Specification for Operator Component

We see in Figure 32 that the generic Array_Type defines itself and its component identifiers (the mandatory ARRAY_ELEMENT and ARRAY_INDEX) as Ada types which is the only option for a generic. Figure 32 also shows the *operator* inputs and outputs referencing the specification's generic. Note that for both Figure 31 and Figure 32, no unrecognized types are referenced (i.e., no references are made to user defined types that are declared and defined externally). There are two reasons for this. First, software base components do not have a parent prototype PSDL specification in which those external references could be looked up and resolved. Second, and more importantly, it is a limitation of this implementation. Future implementations should allow input and output parameters of a software base component to be defined in terms of *types* already defined in the software base.

APPENDIX C - WRAPPER PACKAGE EXAMPLES

This appendix provides examples of a query component prototype PSDL specification, query component *operator* PSDL specification, two software base *operator* component PSDL specifications (one generic, one non-generic), and the wrapper packages generated by integrating the software base components into the user's prototype. The following figure describes the query component prototype PSDL specification:

```
TYPE Index
  SPECIFICATION
    Index : RANGE
  END
IMPLEMENTATION ADA Index_PKG END

TYPE List
  SPECIFICATION
    List : ARRAY [ARRAY_ELEMENT : CHARACTER,
                  ARRAY_INDEX : INTEGER]
  END
IMPLEMENTATION ADA List_PKG END

OPERATOR QC_Op
  SPECIFICATION
    INPUT
          QC_In1 : NATURAL,
          QC_In2 : ARRAY [ARRAY_ELEMENT : BOOLEAN,
                          ARRAY_ELEMENT : Index],
          QC_In3 : Index
   OUTPUT
          QC_Out1 : List,
          QC_Out2 : NATURAL
  END
IMPLEMENTATION ADA QC_Op_PKG END
```

Figure 33 - Query Prototype PSDL Specification

The following figure describes the query *operator* component PSDL specification:

```
OPERATOR QC_Op
  SPECIFICATION
        INPUT
        QC_In1 : NATURAL,
        QC_In2 : ARRAY [ARRAY_ELEMENT : BOOLEAN,
                        ARRAY_INDEX : Index],
        QC_In3 : Index
    OUTPUT
        QC_Out1 : List,
        QC_Out2 : NATURAL
  END
IMPLEMENTATION ADA QC_Op_PKG END
```

Figure 34 - Query Operator Component PSDL Specification

The following figure describes a *generic* software base component that matches Figure 34:

```
OPERATOR SBC_Op
  SPECIFICATION
    GENERIC SBC_Gen_Param1 : PRIVATE,
            SBC_Gen_Param2 : PRIVATE,
            SBC_Gen_Param3 : ARRAY [ARRAY_ELEMENT : PRIVATE,
                                    ARRAY_INDEX : DISCRETE]
    INPUT
            SBC_In1 : SBC_Gen_Param3,
            SBC_In2 : NATURAL,
            SBC_In3 : SBC_Gen_Param2
    OUTPUT
            SBC_Out1 : SBC_Gen_Param1,
            SBC_Out2 : ARRAY [ARRAY_ELEMENT : CHARACTER,
                              ARRAY_INDEX : INTEGER],
            SBC_Out3 : SBC_Gen_Param3,
            SBC_Out4 : POSITIVE,
            SBC_Out5 : ARRAY [ARRAY_ELEMENT : INTEGER,
                              ARRAY_INDEX : INTEGER]
  END
IMPLEMENTATION ADA SBC_Op_SB END
```

Figure 35 - Software Base Generic Component PSDL Specification

Here is an example of a parameter mapping that matches all query PSDL specification parameters to the software base component PSDL specification parameters and additionally defines the generic instantiations:

```
QC_In1   →   SBC_In2
QC_In2   →   SBC_In1   → instantiated SBC_Gen_Param3
QC_In3   →   SBC_In3   → instantiated SBC_Gen_Param2
QC_Out1  →   SBC_Out2
QC_Out2  →   SBC_Out1  → instantiated SBC_Gen_Param1
```

Figure 36 - Parameter Mapping

The following figure lists the Ada source code for the wrapper package generated by the above parameter mapping to integrate the selected software base component into the user prototype:

```
with SBC_Op_SB;
use  SBC_Op_SB;
with Index_PKG;
with List_PKG;

package QC_Op_PKG is

  type QC_In2_ARRAY is array (Index_PKG.Index) of BOOLEAN;

  package TMP_QC_Op_PKG is new SBC_Op_SB(
    SBC_Gen_Param1 => NATURAL,
    SBC_Gen_Param2 => Index_PKG.Index,
    SBC_Gen_Param3 => QC_In2_ARRAY);

  procedure QC_Op(
    QC_In1          : in     NATURAL;
    QC_In2          : in     QC_In2_ARRAY;
    QC_In3          : in     Index_PKG.Index;
    QC_Out1         :   out List_PKG.List;
    QC_Out2         :   out NATURAL);

end QC_Op_PKG;
```

Figure 37 - Generic Software Base Component Wrapper Package

First, let us look at the *with* statements. The software base component package SBC_Op_SB is referenced ("_SB" is a naming convention for software base components). So too are the user defined type packages List_PKG and Index_PKG that were defined in the user prototype PSDL specification. Because the integrated software base component is generic, it must be instantiated. The prefix "TMP_" and the suffix "_PKG" are attached to the query *operator* name QC_Op to create the name of the instantiated package. Looking at the code in which the necessary generic parameter instantiations are made we see examples of three different kinds of generic instantiations. The instantiation of SBC_Gen_Param1 is accomplished with the predefined Ada type NATURAL. The instantiation of SBC_Gen_Param2 is accomplished with the user defined type List. And the instantiation of SBC_Gen_Param3 is accomplished by using an Array type that was predefined in the wrapper package. Looking at the subprogram specification for QC_Op we find the same three examples present for the type definitions of the parameters.

The wrapper package body is generated after the wrapper package. The following figure lists the Ada source code for the wrapper package *body* generated to integrate the selected software base component into the user prototype:

```
-----------------------------------------
---          PACKAGE BODY          ---
-----------------------------------------

package body QC_Op_PKG is

  procedure QC_Op(
    QC_In1              : in      NATURAL;
    QC_In2              : in      QC_In2_ARRAY;
    QC_In3              : in      Index_PKG.Index;
    QC_Out1             :   out List_PKG.List;
    QC_Out2             :   out NATURAL) is

    -- dummy output parameters
    type SBC_Out5_ARRAY is array (INTEGER) of INTEGER;
    SBC_Out3_DUMMY : QC_In2_ARRAY;
    SBC_Out4_DUMMY : POSITIVE;
    SBC_Out5_DUMMY : SBC_Out5_ARRAY;

    begin

      TMP_QC_Op_PKG.SBC_Op(
        QC_In2,
        QC_In1,
        QC_In3,
        QC_Out2,
        QC_Out1,
        SBC_Out3_DUMMY,
        SBC_Out4_DUMMY,
        SBC_Out5_DUMMY);

    end QC_Op;

end QC_Op_PKG;
```

Figure 38 - Generic Software Base Component Wrapper Package Body

In the wrapper package body we see a subprogram implementation for QC_Op nearly identical to the subprogram specification defined in Figure 37. Next, variables for any extra (i.e., unused) software base output parameters must be defined because they will be referenced in the procedure call to the software base component *operator*. We

see from the parameter mapping of Figure 36 that our three extra output parameters are SBC_Out3, SBC_Out4, and SBC_Out5. Each variable identifier is created by appending the suffix "_DUMMY" to the name of the extra output parameter. Two points need to be explained for defining extra software base component output parameters. First, for an output parameter that is not defined by a predefined Ada type (i.e., Array which must have its components defined), a type declaration must precede the variable definitions. This is the case for SBC_Out5. Second, in some cases an output parameter may not have been used, but it was defined by a generic parameter that was instantiated. Under these circumstances the type of the output parameter becomes the type that the corresponding generic parameter was instantiated with. That is the case for SBC_Out3.

Finally, a procedure call must be made to the software base *operator* component. In doing so, the new package TMP_QC_Op_PKG that defines the instantiation of the generic software base component must be referenced. Notice the ordering of the parameters in the call to SBC_Op. It follows the ordering of input and output parameters defined by the software base component PSDL specification of Figure 35 with the actual parameters (except for unused software base component output parameters) being the corresponding query component parameters defined by the parameter mapping in Figure 36.

Now that we have looked at an example of integrating a generic software base component that matched a query component we will look at an example of integrating a *non-generic* software base component. The query component remains the same one described by Figure 34 and has the same prototype PSDL specification of Figure 33. Here is a description of the non-generic software base component PSDL specification:

```
OPERATOR SBC_Op
  SPECIFICATION
    INPUT
          SBC_In1 : ARRAY [ARRAY_ELEMENT : ENUMERATION,
                           ARRAY_INDEX : DISCRETE],
          SBC_In2 : INTEGER,
          SBC_In3 : INTEGER
    OUTPUT
          SBC_Out1 : POSITIVE,
          SBC_Out2 : ARRAY [ARRAY_ELEMENT : CHARACTER,
                           ARRAY_INDEX : POSITIVE],
          SBC_Out3 : INTEGER,
          SBC_Out4 : ARRAY [ARRAY_ELEMENT : INTEGER,
                           ARRAY_INDEX : INTEGER]
  END
IMPLEMENTATION ADA SBC_Op_SB END
```

Figure 39 - Software Base Non-Generic Component PSDL Specification

Here is an example of a parameter mapping that matches all query PSDL specification parameters to the software base component PSDL specification parameters:

```
QC_In1   →   SBC_In3
QC_In2   →   SBC_In1
QC_In3   →   SBC_In2
QC_Out1  →   SBC_Out2
QC_Out2  →   SBC_Out1
```

Figure 40 - Parameter Mapping

The following figure lists the Ada source code for the wrapper package generated by the above parameter mapping to integrate the selected software base component into the user prototype:

```
with SBC_Op_SB;
use  SBC_Op_SB;
with Index_PKG;
with List_PKG;

package QC_Op_PKG is

  type QC_In2_ARRAY is array (Index_PKG.Index) of BOOLEAN;

  procedure QC_Op(
     QC_In1              : in     NATURAL;
     QC_In2              : in     QC_In2_ARRAY;
     QC_In3              : in      Index_PKG.Index;
     QC_Out1               :   out List_PKG.List;
     QC_Out2               :   out NATURAL);
end QC_Op_PKG;
```

Figure 41 - Non-Generic Software Base Component Wrapper Package

The wrapper package body is generated after the wrapper package. The following figure lists the Ada source code for the wrapper package *body* generated to integrate the selected software base component into the user prototype:

```
-----------------------------------------
---          PACKAGE BODY          ---
-----------------------------------------

package body QC_Op_PKG is

  procedure QC_Op(
     QC_In1              : in      NATURAL;
     QC_In2              : in      QC_In2_ARRAY;
     QC_In3              : in       Index_PKG.Index;
     QC_Out1              :    out List_PKG.List;
     QC_Out2              :    out NATURAL) is

    -- dummy output parameters
    type SBC_Out4_ARRAY is array (INTEGER) of INTEGER;
    SBC_Out3_DUMMY : INTEGER;
    SBC_Out4_DUMMY : SBC_Out4_ARRAY;

    begin

      SBC_Op_SB.SBC_Op(
         QC_In2,
         QC_In3,
         QC_In1,
         QC_Out2,
         QC_Out1,
         SBC_Out3_DUMMY,
         SBC_Out4_DUMMY);

    end QC_Op;
end QC_Op_PKG;
```

Figure 42 - Non-Generic Software Base Component Wrapper Package Body

The primary difference between (Figures 37 and 38) and (Figures 41 and 42) is that in the two figures immediately above no generic instantiation takes place and the call to the software base *operator* is made by referencing the software base component package SBC_Op_SB.

APPENDIX D - ADA SOURCE CODE

The code below is listed by program in alphabetical order. It includes Ada code developed to encode signatures, perform array component level matching, generic instantiation validation, type matching, and component transformation.

```
-- Filename    / Ada_Operator_Matching_Routines_Pkg.a
-- Date        / 12 Aug 93
-- Author      / Scott Dolgoff
-- System      / Solbourne
-- Compiler    / Verdix Ada
-- Description / This package provides two kinds of matching routines
--             / that can be used by Operators. The first and simpler
--             / routine determines whether the Arrays of two Operators
--             / match at the array component level. The second routine
--             / determines, given a software base component with generics,
--             / whether an instantiation is possible with the specified
--             / query component.

with PARAMETER_LIST_PKG;
use  PARAMETER_LIST_PKG;

package OPERATOR_MATCH_ROUTINES is

-- This procedure examines the arrays of a query component and a
-- software base component and determines whether each query component
-- array can be matched against a distinct array in the software
-- base component. If so, NO_MATCH is returned as FALSE. If not, then
-- NO_MATCH is returned as TRUE.
 procedure MATCH_ARRAYS(QC_IN_PARAM_LIST  : in PARAMETERS;
                        QC_OUT_PARAM_LIST : in PARAMETERS;
                        SBC_IN_PARAM_LIST : in PARAMETERS;
                        SBC_OUT_PARAM_LIST: in PARAMETERS;
                        NO_MATCH          : in out BOOLEAN);


 -- This procedure attempts to find a valid mapping between a
 -- query component and a generic software base component. The
 -- valid mapping includes finding a possible instantiation of
```

```
-- the generic that leads to a valid mapping between parameters.
-- If a valid mapping is found, NO_MATCH is returned as TRUE. If
-- not, then NO_MATCH is returned as FALSE.
procedure FIND_INSTANTIATION(QC_IN_PARAM_LIST  : in out PARAMETERS;
                             QC_OUT_PARAM_LIST : in out PARAMETERS;
                             SBC_IN_PARAM_LIST : in out PARAMETERS;
                             SBC_OUT_PARAM_LIST : in out PARAMETERS;
                             SBC_GEN_PARAM_LIST : in out PARAMETERS;
                             NO_MATCH          : in out BOOLEAN);


end OPERATOR_MATCH_ROUTINES;


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



with TEXT_IO, ADA_RECOGNIZED_TYPES_PKG, PARAMETER_MAPPING_PKG, PSDL_ID_PKG;
use  TEXT_IO, ADA_RECOGNIZED_TYPES_PKG, PARAMETER_MAPPING_PKG, PSDL_ID_PKG;

package body OPERATOR_MATCH_ROUTINES is


-- create structures to build a dynamic array

 type POTENTIAL_MATCH_LIST(LIST_SIZE : NATURAL);
 type POTENTIAL_MATCH_PTR              is access POTENTIAL_MATCH_LIST;
 type ONE_TO_ONE_MAP                   is array(NATURAL range <>) of
 BOOLEAN;
 type PARAMS                           is array(NATURAL range <>) of
 PARAMETERS;

 type POTENTIAL_MATCH_LIST(LIST_SIZE : NATURAL) is record

  SIZE    : NATURAL                    := LIST_SIZE;

  -- Component A is trying to map to component B.
  -- each array cell corresponds to one of a component B's
  -- parameters (input, output, or combined).  This array is
  -- cycled through when a component A parameter attempts to find
  -- all parameters in component B that it can match.
  P_LIST   : PARAMS(1 .. LIST_SIZE)      := (others => null);

  -- each array cell corresponds to one of component B's parameters.
  -- When one of these is mapped to by component A, the cell is
  -- set to TRUE to indicate that no other component A parameter
  -- can now map to it.
  IS_MAPPED : ONE_TO_ONE_MAP(1 .. LIST_SIZE) := (others => FALSE);
```

```
end record;


type VALID_MATCH_LIST;
type VALID_MATCH_LIST_PTR              is access VALID_MATCH_LIST;

-- Again, component A is trying to map to component B.
-- this structure is used to build a linked list of all
-- component B's parameters that match a particular component A.
-- The reason THE_PARAMETER is an integer is because the integer
-- value represents the index position of the component B
-- parameter that is stored in the array
-- ONE_TO_ONE_MAP.
type VALID_MATCH_LIST is record
 THE_PARAMETER : INTEGER;
 NEXT       : VALID_MATCH_LIST_PTR;
end record;


type MAPPING_LIST(LIST_SIZE : NATURAL);
type MAPPING_LIST_PTR                  is access MAPPING_LIST;
type LIST_ENTRIES                      is array(NATURAL range <>) of
 VALID_MATCH_LIST_PTR;



type MAPPING_LIST(LIST_SIZE : NATURAL) is record

 SIZE       : NATURAL                  := LIST_SIZE;
 -- Again, component A is trying to map to component B.
 -- each array cell corresponds to one of the component A's
 -- parameters. The linked list attached to each array
 -- cell corresponds to all the component B parameters
 -- that match the component A parameter represented by the
 -- array cell.
 P_LIST     : LIST_ENTRIES(1 .. LIST_SIZE)  := (others => null);

end record;


type MATCH_SWITCH                      is (INPUT_MATCH, OUTPUT_MATCH);

MATCH_FOUND : BOOLEAN := FALSE;


------------------------------------------------------------------
---                                                            ---
---     ROUTINES COMMON TO BOTH ARRAY MATCHING AND             ---
---     FINDING A VALID GENERIC INSTANTIATION                  ---
---                                                            ---
```

```
-- The "tree traversal" treats the linked lists of matched software
-- base component parameters as a tree rooted at the first
-- query component parameter. Conceptually, the tree is a combinatoric
-- explosion of all possible mappings of one query parameter to
-- a valid software base parameter (so that no software base component
-- parameter is used more than once). The traversal does not explore
-- all paths. If a node is reached that selects a software base component
-- parameter that was already selected, no lower level nodes on that
-- path need to be visited. Also, the tree is never physically built,
-- just logically traversed. This helps alleviate potential memory
-- problems. Backtracking occurs when a query parameter (tree level)
-- is reached that cannot find an unused software base parameter from
-- its linked list of valid mappings. In the worst case it is
-- estimated that the number of paths to be checked is N factorial
-- where N is the total number of software base parameters.
procedure TREE_TRAVERSAL(QC_LIST       : in MAPPING_LIST_PTR;
                         SBC_PARAM_USAGE : in out POTENTIAL_MATCH_PTR;
                         LEVEL         : in INTEGER) is

-- set a pointer to the current software base parameter for the
-- current query parameter
 SBC_PTR : VALID_MATCH_LIST_PTR := QC_LIST.P_LIST(LEVEL);

begin


 while ((SBC_PTR /= null) and (not MATCH_FOUND)) loop

  if (not SBC_PARAM_USAGE.IS_MAPPED(SBC_PTR.THE_PARAMETER)) then
  -- query parameter maps to software base parameter that has not
  -- been claimed (mapped to) yet.

      SBC_PARAM_USAGE.IS_MAPPED((SBC_PTR.THE_PARAMETER)) := TRUE;
      if (LEVEL < QC_LIST.SIZE) then    -- haven't reached bottom

       TREE_TRAVERSAL(QC_LIST, SBC_PARAM_USAGE, LEVEL + 1);
    -- backtracked to this point. now will try next sbc parameter
    -- in the list so must reset current sbc parameter mapped to
    -- by this query operator to FALSE
       SBC_PARAM_USAGE.IS_MAPPED((SBC_PTR.THE_PARAMETER)) := FALSE;
       SBC_PTR := SBC_PTR.NEXT;
```

```
        else

          MATCH_FOUND := TRUE;

        end if;

    else
    -- try the next query component parameter in the list

        SBC_PTR := SBC_PTR.NEXT;

    end if;
  end loop;

end TREE_TRAVERSAL;
```

```
------------------------------------------------------------------
---                                                            ---
---          ROUTINES FOR ARRAY MATCHING                       ---
---                                                            ---
------------------------------------------------------------------
```

```
-- Step through parameter list and return the total number of
-- String and Array parameter types.
function GET_STRING_AND_ARRAY_COUNT(PARAM_LIST : PARAMETERS) return INTEGER is
  COUNT    : INTEGER    := 0;
  LIST_PTR : PARAMETERS := PARAM_LIST;

begin

  while LIST_PTR /= null loop

    if ((LIST_PTR.THE_TYPE.S = ARRAY_TYPE) or (LIST_PTR.THE_TYPE.S =
     STRING_TYPE)) then
        COUNT := COUNT + 1;
    end if;

    LIST_PTR := LIST_PTR.NEXT;

  end loop;

  return COUNT;
```

```
end GET_STRING_AND_ARRAY_COUNT;
```

```
-- initialize the dynamic arrays.  Set up the list of all
-- software base component Array and String parameters.
procedure INITIALIZE_SBC_PARAM_LIST(CONTAINER  : in out POTENTIAL_MATCH_PTR;
                                    PARAM_LIST : in PARAMETERS) is
 NUM_OF_ARRAYS : INTEGER;

begin

-- build dynamic arrays to store String and Array parameters
 NUM_OF_ARRAYS := GET_STRING_AND_ARRAY_COUNT(PARAM_LIST);
 if NUM_OF_ARRAYS > 0 then
  CONTAINER := new POTENTIAL_MATCH_LIST(NUM_OF_ARRAYS);
 end if;

end INITIALIZE_SBC_PARAM_LIST;
```

```
-- load the software base component dynamic arrays with
-- their parameter list of Array and String types
procedure LOAD_SBC_PARAMETERS(CONTAINER  : in out POTENTIAL_MATCH_PTR;
                              PARAM_LIST : in PARAMETERS) is

 PTR        : PARAMETERS := PARAM_LIST;
 NUM_OF_ARRAYS : INTEGER;

begin

-- load the dynamic arrays
 if CONTAINER /= null then
  NUM_OF_ARRAYS := 1;
  while ((PTR /= null) and (NUM_OF_ARRAYS <= CONTAINER.SIZE)) loop

        if ((PTR.THE_TYPE.S = ARRAY_TYPE) or (PTR.THE_TYPE.S =
         STRING_TYPE)) then
          CONTAINER.P_LIST(NUM_OF_ARRAYS) := PTR;
        end if;

        NUM_OF_ARRAYS := NUM_OF_ARRAYS + 1;
        PTR := PTR.NEXT;
```

```
   end loop;
  end if;

end LOAD_SBC_PARAMETERS;
```

```
-- initialize the dynamic arrays.  Becomes array of query
-- component parameters.  Each array cell has an attached list
-- of all software base component parameters that match the
-- particular query component parameter represented by the
-- array cell.  Initially the list is null.
procedure INITIALIZE_QC_MATCH_LIST(CONTAINER  : in out MAPPING_LIST_PTR;
                                   PARAM_LIST : in PARAMETERS) is
 NUM_OF_ARRAYS : INTEGER;

begin

-- build dynamic arrays to store String and Array parameters
 NUM_OF_ARRAYS := GET_STRING_AND_ARRAY_COUNT(PARAM_LIST);
 if NUM_OF_ARRAYS > 0 then
  CONTAINER := new MAPPING_LIST(NUM_OF_ARRAYS);
 end if;

end INITIALIZE_QC_MATCH_LIST;
```

```
-- Loads the query component parameters (not actual
-- parameters, but positional location in an array) and their
-- corresponding list of matched software base component
-- parameters. If a query component parameter has no corresponding
-- matches amongst the software base component parameters
-- then no overall match is possible and the process for this particular
-- software base component can be terminated.
procedure LOAD_QC_PARAMETERS(CONTAINER       : in out MAPPING_LIST_PTR;
                             QC_PARAM_LIST   : in PARAMETERS;
                             SBC_LIST        : in POTENTIAL_MATCH_PTR;
                             TYPE_OF_MATCH   : in MATCH_SWITCH;
                             NO_MATCH_POSSIBLE : in out BOOLEAN) is

 QC_PTR      : PARAMETERS        := QC_PARAM_LIST;
 SBC_PTR     : POTENTIAL_MATCH_PTR := SBC_LIST;
 PTR         : VALID_MATCH_LIST_PTR;
 NUM_OF_ARRAYS : INTEGER;
 MATCH       : BOOLEAN;
```

```
begin

 NO_MATCH_POSSIBLE := FALSE;
 if CONTAINER /= null then

 -- go through qc parameters
  NUM_OF_ARRAYS := 1;
  while ((QC_PTR /= null) and (NUM_OF_ARRAYS <= CONTAINER.SIZE) and
   (not NO_MATCH_POSSIBLE)) loop

        if ((QC_PTR.THE_TYPE.S = ARRAY_TYPE) or (QC_PTR.THE_TYPE.S =
         STRING_TYPE)) then

          PTR := null;
          NO_MATCH_POSSIBLE := TRUE;
          for INDEX in 1 .. SBC_PTR.SIZE loop   -- go through sbc parameters

           if ((SBC_PTR.P_LIST(INDEX).THE_TYPE.S = ARRAY_TYPE) or
            (SBC_PTR.P_LIST(INDEX).THE_TYPE.S = STRING_TYPE)) then

      -- see if the two parameters match
            if TYPE_OF_MATCH = INPUT_MATCH then
                  MATCH := INPUT_PARAMETER_TYPES_MAP_OK(QC_PTR,
                   SBC_PTR.P_LIST(INDEX));
            else
                  MATCH := OUTPUT_PARAMETER_TYPES_MAP_OK(QC_PTR,
                   SBC_PTR.P_LIST(INDEX));
            end if;

            if MATCH then

                  NO_MATCH_POSSIBLE := FALSE;
          -- add matched software base component parameter to
          -- linked list
                  if (CONTAINER.P_LIST(NUM_OF_ARRAYS) = null) then

          -- initialize list
                   PTR := new VALID_MATCH_LIST;
                   PTR.THE_PARAMETER := INDEX;
                   PTR.NEXT := null;
                   CONTAINER.P_LIST(NUM_OF_ARRAYS) := PTR;

                  else

          -- append to list
                   PTR.NEXT := new VALID_MATCH_LIST;
                   PTR := PTR.NEXT;
                   PTR.THE_PARAMETER := INDEX;
                   PTR.NEXT := null;
                  end if;
             end if;
            end if;
           end loop;
```

```
          NUM_OF_ARRAYS := NUM_OF_ARRAYS + 1;

        end if;

        QC_PTR := QC_PTR.NEXT;

    end loop;
  end if;

end LOAD_QC_PARAMETERS;
```

```
-- This procedure examines the arrays of a query component and a
-- software base component and determines whether each query component
-- array can be matched against a distinct array in the software
-- base component. If so, NO_MATCH is returned as FALSE. If not, then
-- NO_MATCH is returned as TRUE.
procedure MATCH_ARRAYS(QC_IN_PARAM_LIST  : in PARAMETERS;
                       QC_OUT_PARAM_LIST : in PARAMETERS;
                       SBC_IN_PARAM_LIST : in PARAMETERS;
                       SBC_OUT_PARAM_LIST : in PARAMETERS;
                       NO_MATCH          : in out BOOLEAN) is


 QC_IN, QC_OUT  : MAPPING_LIST_PTR   := null;
 SBC_IN, SBC_OUT : POTENTIAL_MATCH_PTR := null;


begin
 MATCH_FOUND := TRUE;

 -- build dynamic arrays to store String and Array parameters
 INITIALIZE_QC_MATCH_LIST(QC_IN, QC_IN_PARAM_LIST);
 INITIALIZE_QC_MATCH_LIST(QC_OUT, QC_OUT_PARAM_LIST);
 INITIALIZE_SBC_PARAM_LIST(SBC_IN, SBC_IN_PARAM_LIST);
 INITIALIZE_SBC_PARAM_LIST(SBC_OUT, SBC_OUT_PARAM_LIST);

 -- store all software base component Array and String
 -- parameters in list
 LOAD_SBC_PARAMETERS(SBC_IN, SBC_IN_PARAM_LIST);
 LOAD_SBC_PARAMETERS(SBC_OUT, SBC_OUT_PARAM_LIST);

 -- store all query component Array and String parameters in
 -- a list (not the actual values, just their representation by
 -- a list index position) and to each list index position, attach
```

```
-- a linked list of all software base component Array and String
-- parameters that match the query component parameter represented
-- by that list index position.
LOAD_QC_PARAMETERS(QC_IN, QC_IN_PARAM_LIST, SBC_IN, INPUT_MATCH, NO_MATCH);

if not NO_MATCH then     -- all query component Array and String input
                         -- parameters have at least one possible mapping

 LOAD_QC_PARAMETERS(QC_OUT, QC_OUT_PARAM_LIST, SBC_OUT, OUTPUT_MATCH,
  NO_MATCH);

 if not NO_MATCH then    -- all query component Array and String output
                         -- parameters have at least one possible mapping

       if (QC_IN /= null) then

   -- search for solution that provides valid mapping for entire
   -- set of query component Array and String input parameters
         MATCH_FOUND := FALSE;
         TREE_TRAVERSAL(QC_IN, SBC_IN, 1);

         if MATCH_FOUND then

          if (QC_OUT /= null) then
      -- search for solution that provides valid mapping for entire
      -- set of query component Array and String output parameters
            MATCH_FOUND := FALSE;
            TREE_TRAVERSAL(QC_OUT, SBC_OUT, 1);
          end if;
         end if;
        end if;

  else

       MATCH_FOUND := FALSE;
  end if;

 else

  MATCH_FOUND := FALSE;

 end if;

 if MATCH_FOUND then
  NO_MATCH := FALSE;
 else
  NO_MATCH := TRUE;
 end if;

end MATCH_ARRAYS;
```

```
------------------------------------------------------------
---                                                        ---
---          ROUTINES FOR                                  ---
---     FINDING A VALID GENERIC INSTANTIATION              ---
---                                                        ---
------------------------------------------------------------



-- Step through parameter list and return the total number of
-- parameter types.
function GET_PARAMETER_COUNT(PARAM_LIST : PARAMETERS) return INTEGER is
 COUNT    : INTEGER   := 0;
 LIST_PTR : PARAMETERS := PARAM_LIST;

begin

 while LIST_PTR /= null loop

  COUNT := COUNT + 1;

  LIST_PTR := LIST_PTR.NEXT;

 end loop;

 return COUNT;

end GET_PARAMETER_COUNT;




---------------------------------------------
---                                       ---
----------------------  Routines Particular to the Process ---
---     of Mapping a Query Component to a  ---
---     Software Base Component Once the   ---
---     Generic Instantiation has been     ---
---     Resolved.                          ---
---                                       ---
---------------------------------------------
```

```
-- initialize the dynamic arrays.  Set up the list of all
-- software base component parameters.
procedure INIT2_SBC_PARAM_LIST(CONTAINER  : in out POTENTIAL_MATCH_PTR;
                               PARAM_LIST : in PARAMETERS) is
 NUM : INTEGER;

begin

-- build dynamic arrays to store parameters
 NUM := GET_PARAMETER_COUNT(PARAM_LIST);
 if NUM > 0 then
  CONTAINER := new POTENTIAL_MATCH_LIST(NUM);
 end if;

end INIT2_SBC_PARAM_LIST;




-- load the software base component dynamic arrays with
-- their parameter list of types
procedure LOAD2_SBC_PARAMETERS(CONTAINER  : in out POTENTIAL_MATCH_PTR;
                               PARAM_LIST : in PARAMETERS) is

 PTR : PARAMETERS := PARAM_LIST;
 NUM : INTEGER;

begin

-- load the dynamic arrays
 if CONTAINER /= null then
  NUM := 1;
  while ((PTR /= null) and (NUM <= CONTAINER.SIZE)) loop

        CONTAINER.P_LIST(NUM) := PTR;

        NUM := NUM + 1;
        PTR := PTR.NEXT;

  end loop;
 end if;

end LOAD2_SBC_PARAMETERS;



-- initialize the dynamic arrays.  Becomes array of query
-- component parameters.  Each array cell has an attached list
```

```
-- of all software base component parameters that match the
-- particular query component parameter represented by the
-- array cell. Initially the list is null.
procedure INIT2_QC_MATCH_LIST(CONTAINER  : in out MAPPING_LIST_PTR;
                              PARAM_LIST : in PARAMETERS) is
 NUM : INTEGER;

begin

-- build dynamic arrays to store parameters
 NUM := GET_PARAMETER_COUNT(PARAM_LIST);
 if NUM > 0 then
  CONTAINER := new MAPPING_LIST(NUM);
 end if;

end INIT2_QC_MATCH_LIST;
```

```
-- Loads the query component parameters (not actual
-- parameters, but positional location in an array) and their
-- corresponding list of matched software base component
-- parameters. If a query component parameter has no corresponding
-- matches amongst the software base component parameters
-- then no overall match is possible and the process for this particular
-- software base component can be terminated. It is important to
-- check and see if the software base component parameter is defined
-- by a generic. If it is, then when matching it with the query
-- component parameter, use the generic instantiation value for the
-- software base parameter.
procedure LOAD2_QC_PARAMETERS(CONTAINER       : in out MAPPING_LIST_PTR;
                              QC_PARAM_LIST   : in PARAMETERS;
                              SBC_LIST        : in POTENTIAL_MATCH_PTR;
                              TYPE_OF_MATCH   : in MATCH_SWITCH;
                              NO_MATCH_POSSIBLE : in out BOOLEAN) is

 QC_PTR  : PARAMETERS       := QC_PARAM_LIST;
 SBC_PTR : POTENTIAL_MATCH_PTR := SBC_LIST;
 PTR     : VALID_MATCH_LIST_PTR;
 NUM     : INTEGER;
 MATCH   : BOOLEAN;

begin

 NO_MATCH_POSSIBLE := FALSE;
 if CONTAINER /= null then

 -- go through qc parameters
```

```
NUM := 1;
while ((QC_PTR /= null) and (NUM <= CONTAINER.SIZE) and
 (not NO_MATCH_POSSIBLE)) loop


      PTR := null;
      NO_MATCH_POSSIBLE := TRUE;
      for INDEX in 1 .. SBC_PTR.SIZE loop   -- go through sbc parameters

  -- see if the two parameters match
        if TYPE_OF_MATCH = INPUT_MATCH then

          if SBC_PTR.P_LIST(INDEX).DEFINED_BY_GENERIC_TYPE then
    -- need to use the instantiated value

            MATCH := INPUT_PARAMETER_TYPES_MAP_OK(QC_PTR,
             SBC_PTR.P_LIST(INDEX).GENERIC_LINK.MAPPING);

          else
            MATCH := INPUT_PARAMETER_TYPES_MAP_OK(QC_PTR,
             SBC_PTR.P_LIST(INDEX));

          end if;

        else                          -- output parameter match

          if SBC_PTR.P_LIST(INDEX).DEFINED_BY_GENERIC_TYPE then
    -- need to use the instantiated value

            MATCH := OUTPUT_PARAMETER_TYPES_MAP_OK(QC_PTR,
             SBC_PTR.P_LIST(INDEX).GENERIC_LINK.MAPPING);

          else
            MATCH := OUTPUT_PARAMETER_TYPES_MAP_OK(QC_PTR,
             SBC_PTR.P_LIST(INDEX));

          end if;

        end if;

        if MATCH then

          NO_MATCH_POSSIBLE := FALSE;
    -- add matched software base component parameter to
    -- linked list
          if (CONTAINER.P_LIST(NUM) = null) then

    -- initialize list
            PTR := new VALID_MATCH_LIST;
            PTR.THE_PARAMETER := INDEX;
            PTR.NEXT := null;
            CONTAINER.P_LIST(NUM) := PTR;

          else
```

```
        -- append to list
              PTR.NEXT := new VALID_MATCH_LIST;
              PTR := PTR.NEXT;
              PTR.THE_PARAMETER := INDEX;
              PTR.NEXT := null;
            end if;
          end if;
        end loop;

        NUM := NUM + 1;


        QC_PTR := QC_PTR.NEXT;

    end loop;
  end if;

end LOAD2_QC_PARAMETERS;
```

```
-- This procedure examines the arrays of a query component and a
-- software base component and determines whether each query component
-- array can be matched against a distinct array in the software
-- base component. If so, NO_MATCH is returned as FALSE. If not, then
-- NO_MATCH is returned as TRUE.
procedure COMPONENT_MATCH(QC_IN_PARAM_LIST  : in PARAMETERS;
                          QC_OUT_PARAM_LIST : in PARAMETERS;
                          SBC_IN_PARAM_LIST : in PARAMETERS;
                          SBC_OUT_PARAM_LIST : in PARAMETERS;
                          NO_MATCH          : in out BOOLEAN) is


 QC_IN, QC_OUT  : MAPPING_LIST_PTR   := null;
 SBC_IN, SBC_OUT : POTENTIAL_MATCH_PTR := null;


begin
 MATCH_FOUND := TRUE;

 -- build dynamic arrays to store String and Array parameters
 INIT2_QC_MATCH_LIST(QC_IN, QC_IN_PARAM_LIST);
 INIT2_QC_MATCH_LIST(QC_OUT, QC_OUT_PARAM_LIST);
 INIT2_SBC_PARAM_LIST(SBC_IN, SBC_IN_PARAM_LIST);
 INIT2_SBC_PARAM_LIST(SBC_OUT, SBC_OUT_PARAM_LIST);
```

```
-- store all software base component parameters in a list
LOAD2_SBC_PARAMETERS(SBC_IN, SBC_IN_PARAM_LIST);
LOAD2_SBC_PARAMETERS(SBC_OUT, SBC_OUT_PARAM_LIST);

-- store all query component parameters in a list
-- (not the actual values, just their representation by
-- a list index position) and to each list index position,
-- attach a linked list of all software base component
-- parameters that match the query component parameter
-- represented by that list index position.
LOAD2_QC_PARAMETERS(QC_IN, QC_IN_PARAM_LIST, SBC_IN, INPUT_MATCH, NO_MATCH);

if not NO_MATCH then     -- all query component input parameters
                         -- have at least one possible mapping

 LOAD2_QC_PARAMETERS(QC_OUT, QC_OUT_PARAM_LIST, SBC_OUT, OUTPUT_MATCH,
  NO_MATCH);

 if not NO_MATCH then    -- all query component output parameters
                         -- have at least one possible mapping

        if (QC_IN /= null) then

   -- search for solution that provides valid mapping for entire
   -- set of query component input parameters
          MATCH_FOUND := FALSE;
          TREE_TRAVERSAL(QC_IN, SBC_IN, 1);

          if MATCH_FOUND then

            if (QC_OUT /= null) then
      -- search for solution that provides valid mapping for
      -- entire set of query component output parameters
              MATCH_FOUND := FALSE;
              TREE_TRAVERSAL(QC_OUT, SBC_OUT, 1);
            end if;
          end if;
        end if;

 else

        MATCH_FOUND := FALSE;
 end if;

else

 MATCH_FOUND := FALSE;

end if;

if MATCH_FOUND then
 NO_MATCH := FALSE;
```

```
  else
   NO_MATCH := TRUE;
  end if;


end COMPONENT_MATCH;
```

```
-------------------------------------------------------
---                                                   ---
---   Routines Particular to the Process      ---
---    of Instantiating Generics                  ---
---                                                   ---
-------------------------------------------------------
```

```
-- determines whether the specified INDEX (representing a particular
-- query parameter) is already represented in the list
function NOT_MEMBER(ID   : INTEGER;
                    HEAD : VALID_MATCH_LIST_PTR) return BOOLEAN is

 PTR        : VALID_MATCH_LIST_PTR := HEAD;
 NOT_A_MEMBER : BOOLEAN            := TRUE;

begin

 while ((PTR /= null) and (NOT_A_MEMBER)) loop

  if (PTR.THE_PARAMETER = ID) then
  -- it's a duplicate
       NOT_A_MEMBER := FALSE;
  end if;

  PTR := PTR.NEXT;

 end loop;

 return NOT_A_MEMBER;

end NOT_MEMBER;
```

```
-- initialize the dynamic arrays.  Set up the list of all
-- query component parameters to be used when instantiating
-- the generics.
procedure INITIALIZE_QC_INSTANTIATION_LIST(CONTAINER  : in out POTENTIAL_MATCH_PTR;
                                           QC_IN      : in PARAMETERS;
                                           QC_OUT     : in PARAMETERS;
                                           NUM_QC_IN  : in out INTEGER;
                                           NUM_QC_OUT : in out INTEGER) is

 TOTAL_QC_PARAMS : INTEGER;

begin

-- build dynamic arrays to store query component parameters

 NUM_QC_IN := GET_PARAMETER_COUNT(QC_IN);
 NUM_QC_OUT := GET_PARAMETER_COUNT(QC_OUT);
 TOTAL_QC_PARAMS := NUM_QC_IN + NUM_QC_OUT;

 if TOTAL_QC_PARAMS > 0 then
  CONTAINER := new POTENTIAL_MATCH_LIST(TOTAL_QC_PARAMS);
 end if;

end INITIALIZE_QC_INSTANTIATION_LIST;




-- initialize the dynamic arrays.  Set up the list of all
-- software base component generic parameters.
procedure INITIALIZE_SBC_GEN_PARAM_LIST(CONTAINER : in out MAPPING_LIST_PTR;
                                        SBC_GEN   : in PARAMETERS) is

 NUM_OF_GENERICS : INTEGER;

begin

-- build dynamic arrays to generic parameters
 NUM_OF_GENERICS := GET_PARAMETER_COUNT(SBC_GEN);

 if NUM_OF_GENERICS > 0 then
  CONTAINER := new MAPPING_LIST(NUM_OF_GENERICS);
 end if;

end INITIALIZE_SBC_GEN_PARAM_LIST;
```

```
-- load the query component dynamic array with its parameter list
-- of input and output parameters.  Used for generic instantiation.
procedure GEN_LOAD_QC_PARAMETERS(CONTAINER : in out POTENTIAL_MATCH_PTR;
                                 QC_IN     : in PARAMETERS;
                                 QC_OUT    : in PARAMETERS) is

 NEW_ID        : PSDL_ID_PKG.PSDL_ID;
 PTR           : PARAMETERS;
 NUM_OF_GENERICS : INTEGER;

begin

-- load the dynamic arrays
 if CONTAINER /= null then
   NUM_OF_GENERICS := 1;
   PTR := QC_IN;
   -- load input parameters
   while ((PTR /= null) and (NUM_OF_GENERICS <= CONTAINER.SIZE)) loop

         CONTAINER.P_LIST(NUM_OF_GENERICS) := PTR;

         NUM_OF_GENERICS := NUM_OF_GENERICS + 1;
         PTR := PTR.NEXT;

   end loop;


   PTR := QC_OUT;
   -- load output parameters
   while ((PTR /= null) and (NUM_OF_GENERICS <= CONTAINER.SIZE)) loop

         CONTAINER.P_LIST(NUM_OF_GENERICS) := PTR;

         NUM_OF_GENERICS := NUM_OF_GENERICS + 1;
         PTR := PTR.NEXT;

   end loop;

 end if;

end GEN_LOAD_QC_PARAMETERS;




-- Loads the software base component generic parameters (not actual
-- parameters, but positional location in an array) and their
-- corresponding list of instantiation matched query component
```

```
-- parameters. If a software base component generic parameter has
-- no corresponding matches amongst the query component parameters
-- then no overall match is possible and the process for this particular
-- software base component can be terminated.
procedure LOAD_GEN_PARAMETERS(CONTAINER        : in out MAPPING_LIST_PTR;
                              SBC_GEN_PARAM_LIST : in PARAMETERS;
                              QC_INSTANTIATIONS  : in POTENTIAL_MATCH_PTR;
                              SBC_IN_PARAM_LIST  : in PARAMETERS;
                              SBC_OUT_PARAM_LIST : in PARAMETERS;
                              NUM_QC_IN          : in INTEGER;
                              NUM_QC_OUT         : in INTEGER;
                              NO_MATCH_POSSIBLE  : in out BOOLEAN) is

 GEN_PTR       : PARAMETERS := SBC_GEN_PARAM_LIST;
 QC_PTR        : POTENTIAL_MATCH_PTR;
 PTR           : VALID_MATCH_LIST_PTR;
 SBC_PTR       : PARAMETERS;
 NUM_OF_GENERICS : INTEGER;
 MATCH         : BOOLEAN;

begin

 NO_MATCH_POSSIBLE := FALSE;
 if CONTAINER /= null then

 -- go through generic parameters
  NUM_OF_GENERICS := 1;
  while ((GEN_PTR /= null) and (NUM_OF_GENERICS <= CONTAINER.SIZE) and
   (not NO_MATCH_POSSIBLE)) loop

        PTR := null;
        NO_MATCH_POSSIBLE := TRUE;

        SBC_PTR := SBC_IN_PARAM_LIST;
        while (SBC_PTR /= null) loop
         if ((SBC_PTR.DEFINED_BY_GENERIC_TYPE) and then (SBC_PTR.GENERIC_LINK =
          GEN_PTR)) then

           QC_PTR := QC_INSTANTIATIONS;
           for INDEX in 1 .. NUM_QC_IN loop   -- go through qc in parameters
      -- see if the two parameters match
             MATCH := INPUT_PARAMETER_TYPES_MAP_OK(QC_PTR.P_LIST(INDEX),
              SBC_PTR);

             if MATCH then
                   NO_MATCH_POSSIBLE := FALSE;
          -- add matched query component parameter to
          -- linked list
                   if (CONTAINER.P_LIST(NUM_OF_GENERICS) = null) then

          -- initialize list
                     PTR := new VALID_MATCH_LIST;
                     PTR.THE_PARAMETER := INDEX;
```

```
            PTR.NEXT := null;
            CONTAINER.P_LIST(NUM_OF_GENERICS) := PTR;

          else

  -- see if INDEX (representing a particular query
  -- component parameter) is already in the list
            if (NOT_MEMBER(INDEX, CONTAINER.P_LIST(NUM_OF_GENERICS))) then

   -- append to list
             PTR.NEXT := new VALID_MATCH_LIST;
             PTR := PTR.NEXT;
             PTR.THE_PARAMETER := INDEX;
             PTR.NEXT := null;
           end if;
         end if;
    end if;

   end loop;

   end if;

   SBC_PTR := SBC_PTR.NEXT;

 end loop;                    -- while SBC_PTR (input params)
                              -- /= null


 SBC_PTR := SBC_OUT_PARAM_LIST;
 while (SBC_PTR /= null) loop

  if ((SBC_PTR.DEFINED_BY_GENERIC_TYPE) and then (SBC_PTR.GENERIC_LINK =
   GEN_PTR)) then

   QC_PTR := QC_INSTANTIATIONS;

-- skip over qc in parameters by starting INDEX at
-- NUM_QC_IN + 1
    for INDEX in (NUM_QC_IN + 1) .. (NUM_QC_OUT + NUM_QC_IN) loop
-- go through qc out parameters

-- see if the two parameters match
      MATCH := OUTPUT_PARAMETER_TYPES_MAP_OK(QC_PTR.P_LIST(INDEX),
       SBC_PTR);

      if MATCH then

            NO_MATCH_POSSIBLE := FALSE;
  -- add matched query component parameter to
  -- linked list
            if (CONTAINER.P_LIST(NUM_OF_GENERICS) = null) then
```

```
        -- initialize list
                PTR := new VALID_MATCH_LIST;
                PTR.THE_PARAMETER := INDEX;
                PTR.NEXT := null;
                CONTAINER.P_LIST(NUM_OF_GENERICS) := PTR;

              else

        -- see if INDEX (representing a particular query
        -- component parameter) is already in the list
                if (NOT_MEMBER(INDEX, CONTAINER.P_LIST(NUM_OF_GENERICS))) then

         -- append to list
                 PTR.NEXT := new VALID_MATCH_LIST;
                 PTR := PTR.NEXT;
                 PTR.THE_PARAMETER := INDEX;
                 PTR.NEXT := null;

                end if;
               end if;
           end if;

          end loop;

         end if;

         SBC_PTR := SBC_PTR.NEXT;

       end loop;                    -- while SBC_PTR (output params)
                                    -- /= null

    -- next generic parameter
        NUM_OF_GENERICS := NUM_OF_GENERICS + 1;
        GEN_PTR := GEN_PTR.NEXT;

  end loop;
 end if;


end LOAD_GEN_PARAMETERS;



-- This function takes an index value from a created array of
-- generic parameters and returns a pointer to the actual
-- generic parameter referenced by the array index value
function GET_GENERIC_PARAM_TO_INSTANTIATE(COUNT     : INTEGER;
                                          PARAM_LIST : PARAMETERS)
                                          return PARAMETERS is
 INDEX   : INTEGER   := COUNT;
 GEN_PTR : PARAMETERS := PARAM_LIST;
```

```
begin

 if (INDEX > 0) then

   INDEX := INDEX - 1;
   while ((GEN_PTR /= null) and (INDEX > 0)) loop

         GEN_PTR := GEN_PTR.NEXT;
         INDEX := INDEX - 1;

   end loop;

   return GEN_PTR;

 else

   return null;

 end if;

exception

 when CONSTRAINT_ERROR =>
   PUT_LINE(" ***  ERROR occured in function ");
   PUT_LINE("     GET_GENERIC_PARAM_TO_INSTANTIATE.");
   raise ;

end GET_GENERIC_PARAM_TO_INSTANTIATE;
```

```
-- The "tree traversal" treats the linked lists of matched query
-- component parameters as a tree rooted at the first software base
-- component parameter. Conceptually, the tree is a combinatoric
-- explosion of all possible mappings of one generic parameter to
-- a valid query component parameter (so that no query component
-- parameter is used more than once). The traversal does not explore
-- all paths. If a node is reached that selects a query component
-- parameter that was already selected, no lower level nodes on that
-- path need to be visited. Also, the tree is never physically built,
-- just logically traversed. This helps alleviate potential memory
-- problems. Backtracking occurs when a generic parameter (tree level)
-- is reached that cannot find an unused query parameter from
-- its linked list of valid mappings.  In the worst case it is
-- estimated that the number of paths to be checked is N factorial
-- where N is the total number of query parameters.
procedure GEN_TREE_TRAVERSAL(GEN_LIST        : in MAPPING_LIST_PTR;
                             QC_PARAM_USAGE  : in out POTENTIAL_MATCH_PTR;
                             QC_IN_PARAM_LIST : in out PARAMETERS;
```

```
                                QC_OUT_PARAM_LIST  : in out PARAMETERS;
                                SBC_IN_PARAM_LIST  : in out PARAMETERS;
                                SBC_OUT_PARAM_LIST : in out PARAMETERS;
                                GEN_PARAM_LIST     : in out PARAMETERS;
                                LEVEL              : in INTEGER) is

 NO_MATCH : BOOLEAN           := TRUE;
 GEN_PTR  : PARAMETERS;
 -- set a pointer to the current query parameter for the
 -- current generic parameter
 QC_PTR   : VALID_MATCH_LIST_PTR := GEN_LIST.P_LIST(LEVEL);

begin


 while ((QC_PTR /= null) and (not MATCH_FOUND)) loop

  if (not QC_PARAM_USAGE.IS_MAPPED(QC_PTR.THE_PARAMETER)) then
  -- generic parameter maps to query parameter that has not
  -- been claimed (mapped to) yet.

        QC_PARAM_USAGE.IS_MAPPED((QC_PTR.THE_PARAMETER)) := TRUE;

    -- instantiate the generic
        GEN_PTR := GET_GENERIC_PARAM_TO_INSTANTIATE(LEVEL, GEN_PARAM_LIST);
        GEN_PTR.MAPPING := QC_PARAM_USAGE.P_LIST(QC_PTR.THE_PARAMETER);


        if (LEVEL < GEN_LIST.SIZE) then   -- haven't reached bottom

          GEN_TREE_TRAVERSAL(GEN_LIST, QC_PARAM_USAGE, QC_IN_PARAM_LIST,
           QC_OUT_PARAM_LIST, SBC_IN_PARAM_LIST, SBC_OUT_PARAM_LIST,
           GEN_PARAM_LIST, LEVEL + 1);
     -- backtracked to this point. now will try next qc parameter
     -- in the list so must reset current query parameter
     -- mapped to by this generic parameter  to FALSE
        QC_PARAM_USAGE.IS_MAPPED((QC_PTR.THE_PARAMETER)) := FALSE;
        QC_PTR := QC_PTR.NEXT;

     -- also need to de-instantiate the generic
        GEN_PTR := GET_GENERIC_PARAM_TO_INSTANTIATE(LEVEL, GEN_PARAM_LIST);
        GEN_PTR.MAPPING := null;

       else
    -- We have a scheme that successfully instantiates all
    -- generics. Now, based on these instantiations, see
    -- if a valid mapping between the query and software
    -- base components is possible. If so, COMPONENT_MATCH
    -- sets the package body global variable MATCH_FOUND to
    -- TRUE.
```

```
        COMPONENT_MATCH(QC_IN_PARAM_LIST, QC_OUT_PARAM_LIST, SBC_IN_PARAM_LIST,
         SBC_OUT_PARAM_LIST, NO_MATCH);

        if not MATCH_FOUND then

    -- we need to remove the hold on this qc parameter
         QC_PARAM_USAGE.IS_MAPPED((QC_PTR.THE_PARAMETER)) := FALSE;
         QC_PTR := QC_PTR.NEXT;

     -- also need to de-instantiate the generic
         GEN_PTR := GET_GENERIC_PARAM_TO_INSTANTIATE(LEVEL, GEN_PARAM_LIST);
         GEN_PTR.MAPPING := null;

        end if;

       end if;

  else
  -- try the next query component parameter in the list

       QC_PTR := QC_PTR.NEXT;

  end if;
 end loop;

end GEN_TREE_TRAVERSAL;




-- This procedure attempts to find a valid mapping between a
-- query component and a generic software base component. The
-- valid mapping includes finding a possible instantiation of
-- the generic that leads to a valid mapping between parameters.
-- If a valid mapping is found, NO_MATCH is returned as TRUE. If
-- not, then NO_MATCH is returned as FALSE.
procedure FIND_INSTANTIATION(QC_IN_PARAM_LIST  : in out PARAMETERS;
                             QC_OUT_PARAM_LIST : in out PARAMETERS;
                             SBC_IN_PARAM_LIST : in out PARAMETERS;
                             SBC_OUT_PARAM_LIST : in out PARAMETERS;
                             SBC_GEN_PARAM_LIST : in out PARAMETERS;
                             NO_MATCH       : in out BOOLEAN) is

 NUM_QC_IN, NUM_QC_OUT : INTEGER          := 0;
 SBC_GEN          : MAPPING_LIST_PTR   := null;
 QC_INSTANTIATIONS   : POTENTIAL_MATCH_PTR := null;


begin

 MATCH_FOUND := TRUE;
```

```
-- build dynamic arrays to store generic parameters
-- and their valid matches (i.e., instantiations)
INITIALIZE_SBC_GEN_PARAM_LIST(SBC_GEN, SBC_GEN_PARAM_LIST);
INITIALIZE_QC_INSTANTIATION_LIST(QC_INSTANTIATIONS, QC_IN_PARAM_LIST,
 QC_OUT_PARAM_LIST, NUM_QC_IN, NUM_QC_OUT);


-- store all qc parameters that could instantiate generic
-- parameters
GEN_LOAD_QC_PARAMETERS(QC_INSTANTIATIONS, QC_IN_PARAM_LIST,
 QC_OUT_PARAM_LIST);


-- store all software base component generic parameters in
-- a list (not the actual values, just their representation by
-- a list index position) and to each list index position, attach
-- a linked list of all query component parameters that match
-- the software base component generic parameter represented
-- by that list index position.
LOAD_GEN_PARAMETERS(SBC_GEN, SBC_GEN_PARAM_LIST, QC_INSTANTIATIONS,
 SBC_IN_PARAM_LIST, SBC_OUT_PARAM_LIST, NUM_QC_IN, NUM_QC_OUT, NO_MATCH);

if not NO_MATCH then   -- all software base component generic
                       -- parameters have at least one possible mapping

 if (SBC_GEN /= null) then

 -- search for solution that provides valid mapping for entire
 -- set of software base component generic parameters
       MATCH_FOUND := FALSE;
       GEN_TREE_TRAVERSAL(SBC_GEN, QC_INSTANTIATIONS, QC_IN_PARAM_LIST,
        QC_OUT_PARAM_LIST, SBC_IN_PARAM_LIST, SBC_OUT_PARAM_LIST,
        SBC_GEN_PARAM_LIST, 1);

 else
       MATCH_FOUND := FALSE;
 end if;

else

 MATCH_FOUND := FALSE;

end if;

if MATCH_FOUND then
 NO_MATCH := FALSE;
else
 NO_MATCH := TRUE;
end if;
```

```
 end FIND_INSTANTIATION;
end OPERATOR_MATCH_ROUTINES;
```

****
****
****

```
-- Filename    / Ada_Recognized_Types_Pkg.a
-- Date        / 6 June 93
-- Author      / Scott Dolgoff
-- System      / Sun SPARCstation
-- Compiler    / Verdix Ada
-- Description / This package defines the recognized Ada types and
--             / provides a function for determining if a specified
--             / type is an Ada type.


with PSDL_ID_PKG;
use  PSDL_ID_PKG;

package ADA_RECOGNIZED_TYPES_PKG is

-- function to determine if a type defined in a PSDL specification is
-- a recognized Ada type (if not then it must be a user defined type)
 function RECOGNIZED_ADA_TYPE(TYPE_NAME : in PSDL_ID_PKG.PSDL_ID)
                              return BOOLEAN;

 -- define recognized Ada types
 PRIVATE_TYPE                 : constant STRING := "PRIVATE";
 DISCRETE_TYPE                : constant STRING := "DISCRETE";
 ENUMERATION_TYPE                  : constant STRING := "ENUMERATION";
 BOOLEAN_TYPE                 : constant STRING := "BOOLEAN";
 CHARACTER_TYPE                  : constant STRING := "CHARACTER";
 INTEGER_TYPE                 : constant STRING := "INTEGER";
 RANGE_TYPE                   : constant STRING := "RANGE";
 NATURAL_TYPE                    : constant STRING := "NATURAL";
 POSITIVE_TYPE                : constant STRING := "POSITIVE";
 ARRAY_TYPE                   : constant STRING := "ARRAY";
 STRING_TYPE                  : constant STRING := "STRING";
 DIGITS_TYPE                  : constant STRING := "DIGITS";
 FLOAT_TYPE                   : constant STRING := "FLOAT";
 DELTA_TYPE                   : constant STRING := "DELTA";
 FIXED_TYPE                   : constant STRING := "FIXED";
 RECORD_TYPE : constant STRING := "RECORD";
 ACCESS_TYPE                  : constant STRING := "ACCESS";

 -- define special types treated as Ada types
 -- they are initialized in the package body {begin .. end}
```

```
 ARRAY_ELEMENT_TYPE, ARRAY_INDEX_TYPE : PSDL_ID_PKG.PSDL_ID;

end ADA_RECOGNIZED_TYPES_PKG;



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



with A_STRINGS, TEXT_IO;
use  A_STRINGS, TEXT_IO;

package body ADA_RECOGNIZED_TYPES_PKG is


-- function to determine if a type defined in a PSDL specification is
-- a recognized Ada type (if not then it must be a user defined type)
 function RECOGNIZED_ADA_TYPE(TYPE_NAME : in PSDL_ID_PKG.PSDL_ID)
                                  return BOOLEAN is
   THE_TYPE_NAME : A_STRINGS.A_STRING;

 begin

   THE_TYPE_NAME := A_STRINGS.LOWER_TO_UPPER(TYPE_NAME);


   if (THE_TYPE_NAME.S = PRIVATE_TYPE) or (THE_TYPE_NAME.S = DISCRETE_TYPE) or
    (THE_TYPE_NAME.S = ENUMERATION_TYPE) or (THE_TYPE_NAME.S = BOOLEAN_TYPE) or
    (THE_TYPE_NAME.S = CHARACTER_TYPE) or (THE_TYPE_NAME.S = INTEGER_TYPE) or
    (THE_TYPE_NAME.S = RANGE_TYPE) or (THE_TYPE_NAME.S = NATURAL_TYPE) or
    (THE_TYPE_NAME.S = POSITIVE_TYPE) or (THE_TYPE_NAME.S = ARRAY_TYPE) or
    (THE_TYPE_NAME.S = STRING_TYPE) or (THE_TYPE_NAME.S = DIGITS_TYPE) or
    (THE_TYPE_NAME.S = FLOAT_TYPE) or (THE_TYPE_NAME.S = DELTA_TYPE) or
    (THE_TYPE_NAME.S = FIXED_TYPE) or (THE_TYPE_NAME.S = RECORD_TYPE) or
    (THE_TYPE_NAME.S = ACCESS_TYPE) then
     return TRUE;
   else
    -- unrecognized Ada type
     return FALSE;
   end if;

 end RECOGNIZED_ADA_TYPE;


begin   -- package body initialization

-- initialize special types that are treated as Ada types
 ARRAY_ELEMENT_TYPE := new A_STRINGS.STRING_REC(13);
 ARRAY_ELEMENT_TYPE.S := "ARRAY_ELEMENT";
 ARRAY_INDEX_TYPE := new A_STRINGS.STRING_REC(11);
 ARRAY_INDEX_TYPE.S := "ARRAY_INDEX";
```

```
end ADA_RECOGNIZED_TYPES_PKG;

*****************************************************************************
*****************************************************************************
*****************************************************************************


-- Filename    / Add_Ada_Type_To_Signature.a
-- Date        / 29 August 93
-- Author      / Scott Dolgoff
-- System      / Sun SPARCstation
-- Compiler    / Verdix Ada
-- Description / This package encodes a specified
--             / Ada type into a software component signature.


with PSDL_ID_PKG;
use  PSDL_ID_PKG;

package ADD_ADA_TYPE_TO_SIGNATURE_PKG is

-- A signature has 24 total regions (7 are generic)
 type   ALL_REGIONS      is range 1 .. 24;

 -- For matching input signatures, the first 18 regions are the
 -- most important. They include all non-generic regions (1..17)
 -- and the generic Private region (Region 18).
 subtype INPUT_REGIONS    is ALL_REGIONS range 1 .. 18;

 -- For matching output signatures, the first 17 regions are the
 -- most important. They include all non-generic regions (1..17).
 -- The generic regions are examined only if the need arises.
 subtype OUTPUT_REGIONS   is ALL_REGIONS range 1 .. 17;

 -- Each region is represented by a 32 bit integer. Thus 2^32 - 1
 -- type instances can be represented by a single region.
 type   SIGNATURE        is array(ALL_REGIONS) of INTEGER;

 -- When encoding a signature, it is necessary to know which
 -- direction in the subtype hierarchy you must travel. In other
 -- words, are you concerned with subregions or super-regions.
 type   IO_SWITCH_CLASSES is (INPUT_PARAMETER, OUTPUT_PARAMETER);


 -- This procedure adds the
 -- Ada type passed in to the Signature passed in, and passes
```

```
-- out the newly updated Signature.
procedure ADD_ADA_TYPE_TO_SIGNATURE(TYPE_NAME    : in PSDL_ID_PKG.PSDL_ID;
                                    THE_SIGNATURE : in out SIGNATURE;
                                    IO_SWITCH    : in IO_SWITCH_CLASSES;
                                    GENERIC_TYPE  : in BOOLEAN);


-- *** This MUST be called prior to encoding an Operator or Type
-- Signature. Sets initial signatures to 0.
procedure INITIALIZE_SIGNATURES(THE_SIGNATURE : in out SIGNATURE);


-- exceptions
UNRECOGNIZED_GENERIC_TYPE, UNRECOGNIZED_TYPE : exception;

end ADD_ADA_TYPE_TO_SIGNATURE_PKG;
```

```
-------------------------------------------------------------------
-------------------------------------------------------------------

with A_STRINGS, TEXT_IO, ADA_RECOGNIZED_TYPES_PKG;
use  A_STRINGS, TEXT_IO, ADA_RECOGNIZED_TYPES_PKG;

package body ADD_ADA_TYPE_TO_SIGNATURE_PKG is


-- Sets initial signatures to 0.
 procedure INITIALIZE_SIGNATURES(THE_SIGNATURE : in out SIGNATURE) is

 begin

  for REGION in ALL_REGIONS'FIRST .. ALL_REGIONS'LAST loop

    THE_SIGNATURE(REGION) := 0;

  end loop;

 end INITIALIZE_SIGNATURES;


-- This procedure adds a type instance to the specified
-- signature region.
procedure ADD_1_TYPE_TO_REGION(REGION       : in ALL_REGIONS;
                               THE_SIGNATURE : in out SIGNATURE) is

 begin

-- increment number of type instances in the region by 1
  THE_SIGNATURE(REGION) := THE_SIGNATURE(REGION) + 1;

 end ADD_1_TYPE_TO_REGION;


-- This procedure adds the
-- Ada type passed in to the Signature passed in, and passes
-- out the newly updated Signature.
procedure ADD_ADA_TYPE_TO_SIGNATURE(TYPE_NAME    : in PSDL_ID_PKG.PSDL_ID;
                                    THE_SIGNATURE : in out SIGNATURE;
                                    IO_SWITCH    : in IO_SWITCH_CLASSES;
                                    GENERIC_TYPE  : in BOOLEAN) is

   REGION1     : constant ALL_REGIONS := 1;      -- Positive
   REGION2     : constant ALL_REGIONS := 2;      -- Natural
   REGION3     : constant ALL_REGIONS := 3;      -- Boolean
   REGION4     : constant ALL_REGIONS := 4;      -- Character
   REGION5     : constant ALL_REGIONS := 5;      -- Range
   REGION6     : constant ALL_REGIONS := 6;      -- Integer
```

```
REGION7     : constant ALL_REGIONS := 7;      -- Enumeration
REGION8     : constant ALL_REGIONS := 8;      -- Discrete
REGION9     : constant ALL_REGIONS := 9;      -- Record
REGION10    : constant ALL_REGIONS := 10;     -- Access
REGION11    : constant ALL_REGIONS := 11;     -- String
REGION12    : constant ALL_REGIONS := 12;     -- Array
REGION13    : constant ALL_REGIONS := 13;     -- Fixed
REGION14    : constant ALL_REGIONS := 14;     -- Delta
REGION15    : constant ALL_REGIONS := 15;     -- Float
REGION16    : constant ALL_REGIONS := 16;     -- Digits
REGION17    : constant ALL_REGIONS := 17;     -- Private
REGION18    : constant ALL_REGIONS := 18;     -- generic Private
REGION19    : constant ALL_REGIONS := 19;     -- generic Discrete
REGION20    : constant ALL_REGIONS := 20;     -- generic Range
REGION21    : constant ALL_REGIONS := 21;     -- generic Array
REGION22    : constant ALL_REGIONS := 22;     -- generic Digits
REGION23    : constant ALL_REGIONS := 23;     -- generic Delta
REGION24    : constant ALL_REGIONS := 24;     -- generic Float


THE_TYPE_NAME : A_STRINGS.A_STRING;


begin

THE_TYPE_NAME := A_STRINGS.LOWER_TO_UPPER(TYPE_NAME);

case IO_SWITCH is

 when INPUT_PARAMETER =>

      if ((PRIVATE_TYPE = THE_TYPE_NAME.S) and     -- generic Private
       (GENERIC_TYPE)) then
  -- all regions are included as the pattern for this type
       for REGION in INPUT_REGIONS'FIRST .. INPUT_REGIONS'LAST loop
        ADD_1_TYPE_TO_REGION(REGION, THE_SIGNATURE);
       end loop;

      elsif (PRIVATE_TYPE = THE_TYPE_NAME.S) then   -- Private
       ADD_1_TYPE_TO_REGION(REGION17, THE_SIGNATURE);

      elsif (DISCRETE_TYPE = THE_TYPE_NAME.S) then
  -- regions below discrete are numbered consecutively from
  -- 1 through 7.
       for REGION in REGION1 .. REGION8 loop
        ADD_1_TYPE_TO_REGION(REGION, THE_SIGNATURE);
       end loop;
```

```
    if GENERIC_TYPE then
-- add information to particular generic type
      ADD_1_TYPE_TO_REGION(REGION19, THE_SIGNATURE);
    end if;

   elsif (ENUMERATION_TYPE = THE_TYPE_NAME.S) then
    ADD_1_TYPE_TO_REGION(REGION7, THE_SIGNATURE);
-- also add Boolean and Character
    ADD_1_TYPE_TO_REGION(REGION3, THE_SIGNATURE);
    ADD_1_TYPE_TO_REGION(REGION4, THE_SIGNATURE);

   elsif (BOOLEAN_TYPE = THE_TYPE_NAME.S) then
    ADD_1_TYPE_TO_REGION(REGION3, THE_SIGNATURE);

   elsif (CHARACTER_TYPE = THE_TYPE_NAME.S) then
    ADD_1_TYPE_TO_REGION(REGION4, THE_SIGNATURE);

   elsif (INTEGER_TYPE = THE_TYPE_NAME.S) then
    ADD_1_TYPE_TO_REGION(REGION6, THE_SIGNATURE);
-- add Range, Natural, and Positive
    ADD_1_TYPE_TO_REGION(REGION5, THE_SIGNATURE);
    ADD_1_TYPE_TO_REGION(REGION2, THE_SIGNATURE);
    ADD_1_TYPE_TO_REGION(REGION1, THE_SIGNATURE);

   elsif (RANGE_TYPE = THE_TYPE_NAME.S) then
    ADD_1_TYPE_TO_REGION(REGION5, THE_SIGNATURE);

    if GENERIC_TYPE then
-- add information to particular generic type
      ADD_1_TYPE_TO_REGION(REGION20, THE_SIGNATURE);
    end if;

   elsif (NATURAL_TYPE = THE_TYPE_NAME.S) then
    ADD_1_TYPE_TO_REGION(REGION2, THE_SIGNATURE);
-- add Positive
    ADD_1_TYPE_TO_REGION(REGION1, THE_SIGNATURE);

   elsif (POSITIVE_TYPE = THE_TYPE_NAME.S) then
    ADD_1_TYPE_TO_REGION(REGION1, THE_SIGNATURE);

   elsif (ARRAY_TYPE = THE_TYPE_NAME.S) then
    ADD_1_TYPE_TO_REGION(REGION12, THE_SIGNATURE);
-- add String
    ADD_1_TYPE_TO_REGION(REGION11, THE_SIGNATURE);

    if GENERIC_TYPE then
-- add information to particular generic type
      ADD_1_TYPE_TO_REGION(REGION21, THE_SIGNATURE);
    end if;

   elsif (STRING_TYPE = THE_TYPE_NAME.S) then
    ADD_1_TYPE_TO_REGION(REGION11, THE_SIGNATURE);

   elsif (DIGITS_TYPE = THE_TYPE_NAME.S) then
```

```
        ADD_1_TYPE_TO_REGION(REGION16, THE_SIGNATURE);
    -- add Float
        ADD_1_TYPE_TO_REGION(REGION15, THE_SIGNATURE);

        if GENERIC_TYPE then
    -- add information to particular generic type
          ADD_1_TYPE_TO_REGION(REGION22, THE_SIGNATURE);
        end if;

       elsif (FLOAT_TYPE = THE_TYPE_NAME.S) then
        ADD_1_TYPE_TO_REGION(REGION15, THE_SIGNATURE);

       elsif (DELTA_TYPE = THE_TYPE_NAME.S) then
        ADD_1_TYPE_TO_REGION(REGION14, THE_SIGNATURE);
    -- add Fixed
        ADD_1_TYPE_TO_REGION(REGION13, THE_SIGNATURE);

        if GENERIC_TYPE then
    -- add information to particular generic type
          ADD_1_TYPE_TO_REGION(REGION23, THE_SIGNATURE);
        end if;

       elsif (FIXED_TYPE = THE_TYPE_NAME.S) then
        ADD_1_TYPE_TO_REGION(REGION13, THE_SIGNATURE);

       elsif (RECORD_TYPE = THE_TYPE_NAME.S) then
        ADD_1_TYPE_TO_REGION(REGION9, THE_SIGNATURE);

       elsif (ACCESS_TYPE = THE_TYPE_NAME.S) then
        ADD_1_TYPE_TO_REGION(REGION10, THE_SIGNATURE);

        if GENERIC_TYPE then
    -- add information to particular generic type
          ADD_1_TYPE_TO_REGION(REGION24, THE_SIGNATURE);
        end if;

       else
        raise UNRECOGNIZED_TYPE;
       end if;


when OUTPUT_PARAMETER =>

       if GENERIC_TYPE then

        if (PRIVATE_TYPE = THE_TYPE_NAME.S) then
         ADD_1_TYPE_TO_REGION(REGION18, THE_SIGNATURE);

        elsif (DISCRETE_TYPE = THE_TYPE_NAME.S) then
         ADD_1_TYPE_TO_REGION(REGION19, THE_SIGNATURE);

        elsif (RANGE_TYPE = THE_TYPE_NAME.S) then
         ADD_1_TYPE_TO_REGION(REGION20, THE_SIGNATURE);

        elsif (ARRAY_TYPE = THE_TYPE_NAME.S) then
```

```
      ADD_1_TYPE_TO_REGION(REGION21, THE_SIGNATURE);

     elsif (DIGITS_TYPE = THE_TYPE_NAME.S) then
      ADD_1_TYPE_TO_REGION(REGION22, THE_SIGNATURE);

     elsif (DELTA_TYPE = THE_TYPE_NAME.S) then
      ADD_1_TYPE_TO_REGION(REGION23, THE_SIGNATURE);

     elsif (ACCESS_TYPE = THE_TYPE_NAME.S) then
      ADD_1_TYPE_TO_REGION(REGION24, THE_SIGNATURE);

     else
      raise UNRECOGNIZED_GENERIC_TYPE;

     end if;

    elsif (PRIVATE_TYPE = THE_TYPE_NAME.S) then
     ADD_1_TYPE_TO_REGION(REGION17, THE_SIGNATURE);

    elsif (DISCRETE_TYPE = THE_TYPE_NAME.S) then
     ADD_1_TYPE_TO_REGION(REGION8, THE_SIGNATURE);

    elsif (ENUMERATION_TYPE = THE_TYPE_NAME.S) then
     ADD_1_TYPE_TO_REGION(REGION7, THE_SIGNATURE);
-- add Discrete
     ADD_1_TYPE_TO_REGION(REGION8, THE_SIGNATURE);

    elsif (BOOLEAN_TYPE = THE_TYPE_NAME.S) then
     ADD_1_TYPE_TO_REGION(REGION3, THE_SIGNATURE);
-- add Enumeration, Discrete
     ADD_1_TYPE_TO_REGION(REGION7, THE_SIGNATURE);
     ADD_1_TYPE_TO_REGION(REGION8, THE_SIGNATURE);

    elsif (CHARACTER_TYPE = THE_TYPE_NAME.S) then
     ADD_1_TYPE_TO_REGION(REGION4, THE_SIGNATURE);
-- add Enumeration, Discrete
     ADD_1_TYPE_TO_REGION(REGION7, THE_SIGNATURE);
     ADD_1_TYPE_TO_REGION(REGION8, THE_SIGNATURE);

    elsif (INTEGER_TYPE = THE_TYPE_NAME.S) then
     ADD_1_TYPE_TO_REGION(REGION6, THE_SIGNATURE);
-- add Discrete
     ADD_1_TYPE_TO_REGION(REGION8, THE_SIGNATURE);

    elsif (RANGE_TYPE = THE_TYPE_NAME.S) then
     ADD_1_TYPE_TO_REGION(REGION5, THE_SIGNATURE);
-- add Integer, Discrete
     ADD_1_TYPE_TO_REGION(REGION6, THE_SIGNATURE);
     ADD_1_TYPE_TO_REGION(REGION8, THE_SIGNATURE);

    elsif (NATURAL_TYPE = THE_TYPE_NAME.S) then
     ADD_1_TYPE_TO_REGION(REGION2, THE_SIGNATURE);
-- add Integer, Discrete
     ADD_1_TYPE_TO_REGION(REGION6, THE_SIGNATURE);
```

```
            ADD_1_TYPE_TO_REGION(REGION8, THE_SIGNATURE);

          elsif (POSITIVE_TYPE = THE_TYPE_NAME.S) then
            ADD_1_TYPE_TO_REGION(REGION1, THE_SIGNATURE);
        -- add Natural, Integer, Discrete
            ADD_1_TYPE_TO_REGION(REGION2, THE_SIGNATURE);
            ADD_1_TYPE_TO_REGION(REGION6, THE_SIGNATURE);
            ADD_1_TYPE_TO_REGION(REGION8, THE_SIGNATURE);

          elsif (ARRAY_TYPE = THE_TYPE_NAME.S) then
            ADD_1_TYPE_TO_REGION(REGION12, THE_SIGNATURE);

          elsif (STRING_TYPE = THE_TYPE_NAME.S) then
            ADD_1_TYPE_TO_REGION(REGION11, THE_SIGNATURE);
        -- add Array
            ADD_1_TYPE_TO_REGION(REGION12, THE_SIGNATURE);

          elsif (DIGITS_TYPE = THE_TYPE_NAME.S) then
            ADD_1_TYPE_TO_REGION(REGION16, THE_SIGNATURE);

          elsif (FLOAT_TYPE = THE_TYPE_NAME.S) then
            ADD_1_TYPE_TO_REGION(REGION15, THE_SIGNATURE);
        -- add Digits
            ADD_1_TYPE_TO_REGION(REGION16, THE_SIGNATURE);

          elsif (DELTA_TYPE = THE_TYPE_NAME.S) then
            ADD_1_TYPE_TO_REGION(REGION14, THE_SIGNATURE);

          elsif (FIXED_TYPE = THE_TYPE_NAME.S) then
            ADD_1_TYPE_TO_REGION(REGION13, THE_SIGNATURE);
        -- add Delta
            ADD_1_TYPE_TO_REGION(REGION14, THE_SIGNATURE);

          elsif (RECORD_TYPE = THE_TYPE_NAME.S) then
            ADD_1_TYPE_TO_REGION(REGION9, THE_SIGNATURE);

          elsif (ACCESS_TYPE = THE_TYPE_NAME.S) then
            ADD_1_TYPE_TO_REGION(REGION10, THE_SIGNATURE);

          else
           raise UNRECOGNIZED_TYPE;
          end if;


    end case;


exception
    when UNRECOGNIZED_TYPE =>
      PUT_LINE("*** ERROR ***   CANNOT ENCODE SIGNATURE WITH ");
      PUT_LINE(THE_TYPE_NAME.S & " BECAUSE IT IS AN");
      PUT_LINE("UNRECOGNIZED TYPE.");
      raise ;
```

```
   when UNRECOGNIZED_GENERIC_TYPE =>
     PUT_LINE("*** ERROR ***   CANNOT ENCODE SIGNATURE WITH ");
     PUT_LINE(THE_TYPE_NAME.S & " BECAUSE IT CANNOT BE");
     PUT_LINE("A GENERIC TYPE.");
     raise ;

   when others =>
     PUT_LINE("*** UNKNOWN ERROR *** FOUND IN procedure ");
     PUT_LINE("ADD_ADA_TYPE_TO_SIGNATURE.");
     raise ;

 end ADD_ADA_TYPE_TO_SIGNATURE;

end ADD_ADA_TYPE_TO_SIGNATURE_PKG;
```

```
*******************************************************************************
*******************************************************************************
*******************************************************************************
```

```
-- Filename    | ADT_Parameter_Iterator_Pkg.a
-- Date        | 30 August 93
-- Author      | Scott Dolgoff
-- System      | Sun SPARCstation
-- Compiler    | Verdix Ada
-- Description | This program encodes the signatures of software
--             | ADT (type) components from their Prototype System

with PSDL_ID_PKG, PSDL_CONCRETE_TYPE_PKG, ADD_ADA_TYPE_TO_SIGNATURE_PKG,
 PSDL_COMPONENT_PKG;
use  PSDL_ID_PKG, PSDL_CONCRETE_TYPE_PKG, ADD_ADA_TYPE_TO_SIGNATURE_PKG,
 PSDL_COMPONENT_PKG;

package ITERATE_THROUGH_ADT_OPERATOR_PARAMETERS_PKG is

-- procedure that allows this package additional access to the particular
-- ADT operator whose signature is being calculated.
 procedure PASS_ADT_OPERATOR(OP_COMPONENT : in PSDL_COMPONENT_PKG.OPERATOR);


 -- procedure passed to generic scan procedure in generic map package
 -- extracts the input parameters from the Type Declaration
 -- map
 procedure GET_IN_PARAMETER(ID       : in PSDL_ID_PKG.PSDL_ID;
                            TYPE_NAME : in PSDL_CONCRETE_TYPE_PKG.TYPE_NAME);

 -- iterate through the Type Declaration map to extract the set of input
```

```
 -- parameters.
 procedure ITERATE_THROUGH_INPUT_PARAMETERS is new
PSDL_CONCRETE_TYPE_PKG.TYPE_DECLARATION_PKG.GENERIC_SCAN(GENERATE=>GET_IN_PAR
AMETER);


 -- procedure passed to generic scan procedure in generic map package
 -- extracts the output parameters from the Type Declaration
 -- map
 procedure GET_OUT_PARAMETER(ID      : in PSDL_ID_PKG.PSDL_ID;
                             TYPE_NAME : in PSDL_CONCRETE_TYPE_PKG.TYPE_NAME);


 -- iterate through the Type Declaration map to extract the set of output
 -- parameters.
 procedure ITERATE_THROUGH_OUTPUT_PARAMETERS is new
PSDL_CONCRETE_TYPE_PKG.TYPE_DECLARATION_PKG.GENERIC_SCAN(GENERATE=>GET_OUT_PA
RAMETER);


 -- Encoded input and output signatures are returned by these
 -- procedures. NOTE: the procedure ITERATE_THROUGH_xxx_PARAMETERS
 -- must be invoked first which encodes the signature value.
 procedure GET_INPUT_SIGNATURE(IN_SIGNATURE : out SIGNATURE);
 procedure GET_OUTPUT_SIGNATURE(OUT_SIGNATURE : out SIGNATURE);


 -- *** This MUST be called prior to encoding an Operator or Type
 -- Signature. Sets initial signatures to all 0's. IO_SWITCH
 -- tells the procedure whether you need to initialize the input
 -- or output signature.
 procedure INITIALIZE_THE_SIGNATURES(IO_SWITCH : IO_SWITCH_CLASSES);


 -- exceptions
 -- the user defined type of an Operator input or output
 -- parameter was not defined in the Prototype PSDL specification
 UNDEFINED_USER_DEFINED_TYPE : exception;
 -- the type used to define a user defined type is not a
 -- valid Ada type
 INVALID_ADA_TYPE       : exception;

end ITERATE_THROUGH_ADT_OPERATOR_PARAMETERS_PKG;


------------------------------------------------------------
------------------------------------------------------------
```

```
with TEXT_IO, ADA_RECOGNIZED_TYPES_PKG, PSDL_PROGRAM_PKG,
 LOAD_PSDL_INTO_ADA_STRUCTURE_PKG, TYPE_NAME_PKG, A_STRINGS;
use  TEXT_IO, ADA_RECOGNIZED_TYPES_PKG, PSDL_PROGRAM_PKG,
 LOAD_PSDL_INTO_ADA_STRUCTURE_PKG, A_STRINGS;

package body ITERATE_THROUGH_ADT_OPERATOR_PARAMETERS_PKG is

 INPUT_SIGNATURE, OUTPUT_SIGNATURE   : SIGNATURE;
 GV_NAME, GV_UDT_NAME              : PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
 PROTOTYPE_SPEC                : PSDL_PROGRAM_PKG.PSDL_PROGRAM;
 MAIN_TYPE_COMPONENT, TYPE_COMPONENT : PSDL_COMPONENT_PKG.DATA_TYPE;
 OPERATOR_COMPONENT              : PSDL_COMPONENT_PKG.OPERATOR;


 -- procedure that allows this package additional access to the particular
 -- ADT operator whose signature is being calculated.
 procedure PASS_ADT_OPERATOR(OP_COMPONENT : in PSDL_COMPONENT_PKG.OPERATOR) is

 begin

  OPERATOR_COMPONENT := OP_COMPONENT;

 end PASS_ADT_OPERATOR;



 -- Encoded input and closeness signatures are returned by this
 -- procedure. NOTE:  the procedure ITERATE_THROUGH_INPUT_PARAMETERS
 -- must be invoked first which encodes the signature.
 procedure GET_INPUT_SIGNATURE(IN_SIGNATURE : out SIGNATURE) is

 begin

  IN_SIGNATURE := INPUT_SIGNATURE;

 end GET_INPUT_SIGNATURE;



 -- Encoded output signature is returned by this
 --          procedure.          NOTE:          the          procedure
ITERATE_THROUGH_OUTPUT_PARAMETERS
 -- must be invoked first which encodes the signature.
 procedure GET_OUTPUT_SIGNATURE(OUT_SIGNATURE : out SIGNATURE) is

 begin

  OUT_SIGNATURE := OUTPUT_SIGNATURE;

 end GET_OUTPUT_SIGNATURE;
```

```
-- initializes or resets signatures so new ones can be computed
procedure INITIALIZE_THE_SIGNATURES(IO_SWITCH : IO_SWITCH_CLASSES) is

begin

  case IO_SWITCH is

    when INPUT_PARAMETER =>
         INITIALIZE_SIGNATURES(INPUT_SIGNATURE);

    when OUTPUT_PARAMETER =>
         INITIALIZE_SIGNATURES(OUTPUT_SIGNATURE);

  end case;

end INITIALIZE_THE_SIGNATURES;



-- input parameter must be a user defined type so we must look
-- up its Ada type representation by extracting the
-- user defined type from the prototype PSDL specification
procedure GET_USER_DEFINED_TYPE(THE_TYPE_NAME : in
PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
                                UDT_TYPE_NAME : in out
PSDL_CONCRETE_TYPE_PKG.TYPE_NAME) is

  TYPE_MODEL : PSDL_CONCRETE_TYPE_PKG.TYPE_DECLARATION;

begin

-- retrieve the type component specification
  TYPE_COMPONENT := PSDL_PROGRAM_PKG.FETCH(PROTOTYPE_SPEC,
   THE_TYPE_NAME.NAME);
  if TYPE_COMPONENT = null then
    GV_NAME := THE_TYPE_NAME;
    raise UNDEFINED_USER_DEFINED_TYPE;
  end if;
  -- get the corresponding Ada type of the user defined type
  TYPE_MODEL := PSDL_COMPONENT_PKG.MODEL(TYPE_COMPONENT);

  UDT_TYPE_NAME :=
   PSDL_CONCRETE_TYPE_PKG.TYPE_DECLARATION_PKG.FETCH(TYPE_MODEL,
   THE_TYPE_NAME.NAME);

  -- Ensure type was defined.  If not, then nothing was fetched.
  if TYPE_NAME_PKG."=" (UDT_TYPE_NAME, TYPE_NAME_PKG.NULL_TYPE) then
    GV_NAME := THE_TYPE_NAME;
    raise UNDEFINED_USER_DEFINED_TYPE;
  end if;
```

```
 -- type of the UDT must be a valid Ada type
 if not RECOGNIZED_ADA_TYPE(UDT_TYPE_NAME.NAME) then
  GV_NAME := THE_TYPE_NAME;
  GV_UDT_NAME := UDT_TYPE_NAME;
  raise INVALID_ADA_TYPE;
 end if;

end GET_USER_DEFINED_TYPE;



-- Search through the type_declaration section of the ADT to see
-- if the Ada type we are looking for is defined as one of the
-- Type's ADTs.
procedure CHECK_ADT_ADTS(THE_TYPE_NAME : in PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
                         ADT_TYPE_NAME : in out PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
                         FOUND_TYPE    : out BOOLEAN) is

 TYPE_MODEL : PSDL_CONCRETE_TYPE_PKG.TYPE_DECLARATION;

begin

-- use the Type component
 TYPE_COMPONENT := MAIN_TYPE_COMPONENT;

 -- try to get the corresponding Ada type of the unknown type
 TYPE_MODEL := PSDL_COMPONENT_PKG.MODEL(TYPE_COMPONENT);

 ADT_TYPE_NAME :=
  PSDL_CONCRETE_TYPE_PKG.TYPE_DECLARATION_PKG.FETCH(TYPE_MODEL,
  THE_TYPE_NAME.NAME);

 -- Ensure type was defined.  If not, then nothing was fetched.
 if TYPE_NAME_PKG."=" (ADT_TYPE_NAME, TYPE_NAME_PKG.NULL_TYPE) then
  FOUND_TYPE := FALSE;
 else
  FOUND_TYPE := TRUE;


  -- type of the ADT must be a valid Ada type
  if not RECOGNIZED_ADA_TYPE(ADT_TYPE_NAME.NAME) then
        GV_NAME := THE_TYPE_NAME;
        GV_UDT_NAME := ADT_TYPE_NAME;
        raise INVALID_ADA_TYPE;

  end if;

 end if;

end CHECK_ADT_ADTS;
```

```
-- Looks through the generic parameters of a Type or Operator to
-- see if an unrecognized type is defined as an Ada type there.
procedure CHECK_GENERICS(COMPONENT       : in out PSDL_COMPONENT_PKG.PSDL_COMPONENT;
                                                  UNKNOWN_TYPE_NAME       :       in       out
PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
                                GENERIC_TYPE_NAME : in out PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
                                FOUND_TYPE      : out BOOLEAN) is

  GENERIC_MODEL : PSDL_CONCRETE_TYPE_PKG.TYPE_DECLARATION;
  COMP_NAME    : PSDL_ID_PKG.PSDL_ID;

 begin

 -- Get the generic parameters.
  GENERIC_MODEL := PSDL_COMPONENT_PKG.GENERIC_PARAMETERS(COMPONENT);

  if not PSDL_CONCRETE_TYPE_PKG.EQUAL(GENERIC_MODEL,
   PSDL_CONCRETE_TYPE_PKG.EMPTY_TYPE_DECLARATION) then

  -- get the corresponding Ada type of the corresponding unknown type
   GENERIC_TYPE_NAME :=
    PSDL_CONCRETE_TYPE_PKG.TYPE_DECLARATION_PKG.FETCH(GENERIC_MODEL,
    UNKNOWN_TYPE_NAME.NAME);

   -- Ensure unknown type was defined.  If not, then nothing was fetched.
   if TYPE_NAME_PKG."=" (GENERIC_TYPE_NAME, TYPE_NAME_PKG.NULL_TYPE) then
   -- will have to check something other then generics
         FOUND_TYPE := FALSE;
   else
   -- type of the GENERIC must be a valid Ada type
         if RECOGNIZED_ADA_TYPE(GENERIC_TYPE_NAME.NAME) then
          FOUND_TYPE := TRUE;
         else
          GV_NAME := UNKNOWN_TYPE_NAME;
          GV_UDT_NAME := GENERIC_TYPE_NAME;
          raise INVALID_ADA_TYPE;
         end if;
   end if;

  else
  -- will have to check something other then generics
   FOUND_TYPE := FALSE;
  end if;
end CHECK_GENERICS;



procedure ENCODE_ADA_TYPE_INTO_SIGNATURE(TYPE_NAME_ID : in PSDL_ID_PKG.PSDL_ID;
                                         IO_SWITCH    : in IO_SWITCH_CLASSES;
```

```
                                        GENERIC_TYPE : in BOOLEAN) is

begin
 case IO_SWITCH is

  when INPUT_PARAMETER =>
       ADD_ADA_TYPE_TO_SIGNATURE(TYPE_NAME_ID, INPUT_SIGNATURE, IO_SWITCH,
       GENERIC_TYPE);


  when OUTPUT_PARAMETER =>
       ADD_ADA_TYPE_TO_SIGNATURE(TYPE_NAME_ID, OUTPUT_SIGNATURE, IO_SWITCH,
       GENERIC_TYPE);

 end case;

end ENCODE_ADA_TYPE_INTO_SIGNATURE;
```

```
-- procedure passed to generic scan procedure in generic map package
-- extracts the input parameters from the Type Declaration
-- map.  NOTE that this procedure will be run once for each input
-- parameter as the entire set of input parameters is iterated through.
procedure GET_IN_PARAMETER(ID      : in PSDL_ID_PKG.PSDL_ID;
                           TYPE_NAME : in PSDL_CONCRETE_TYPE_PKG.TYPE_NAME) is

 ADT_TYPE_NAME, GENERIC_TYPE_NAME, THE_TYPE_NAME, UDT_TYPE_NAME :
 PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
 -- User Defined Type (UDT) type name
 IO_SWITCH                                :
 IO_SWITCH_CLASSES;
 GENERIC_TYPE, FOUND_TYPE                          : BOOLEAN;


begin
 IO_SWITCH := INPUT_PARAMETER;
 THE_TYPE_NAME := TYPE_NAME;
 GENERIC_TYPE := FALSE;

 if RECOGNIZED_ADA_TYPE(THE_TYPE_NAME.NAME) then
 -- Input parameter is a valid Ada type so go ahead and encode
 -- the current Input_Signature with this Ada type.
  THE_TYPE_NAME.NAME := A_STRINGS.LOWER_TO_UPPER(THE_TYPE_NAME.NAME);
  ENCODE_ADA_TYPE_INTO_SIGNATURE(THE_TYPE_NAME.NAME, IO_SWITCH,
   GENERIC_TYPE);

 else
 -- Maybe the parameter is defined in the generics of the
```

```
-- Operator component.
 CHECK_GENERICS(OPERATOR_COMPONENT, THE_TYPE_NAME, GENERIC_TYPE_NAME,
  FOUND_TYPE);

 if FOUND_TYPE then
       UDT_TYPE_NAME := GENERIC_TYPE_NAME;
       GENERIC_TYPE := TRUE;

 else
 -- Maybe the parameter is defined in the generics of the
 -- Type component.
       CHECK_GENERICS(MAIN_TYPE_COMPONENT, THE_TYPE_NAME, GENERIC_TYPE_NAME,
        FOUND_TYPE);

       if FOUND_TYPE then
        UDT_TYPE_NAME := GENERIC_TYPE_NAME;
        GENERIC_TYPE := TRUE;

       else
  -- Maybe the parameter is defined in the type_declaration
  -- block of the Type component (i.e. it is one of the Type
  -- component's ADTs.
        CHECK_ADT_ADTS(THE_TYPE_NAME, ADT_TYPE_NAME, FOUND_TYPE);

        if FOUND_TYPE then
         UDT_TYPE_NAME := ADT_TYPE_NAME;


        else   -- The parameter must be a user defined type so
         -- get its type.

         GET_USER_DEFINED_TYPE(THE_TYPE_NAME, UDT_TYPE_NAME);

        end if;
       end if;
 end if;

 UDT_TYPE_NAME.NAME := A_STRINGS.LOWER_TO_UPPER(UDT_TYPE_NAME.NAME);
 -- go ahead and encode into the current Input_Signature.
 ENCODE_ADA_TYPE_INTO_SIGNATURE(UDT_TYPE_NAME.NAME, IO_SWITCH,
  GENERIC_TYPE);

end if;

exception

 when UNDEFINED_USER_DEFINED_TYPE =>
  PUT_LINE("*** ERROR:  " & GV_NAME.NAME.S & " IS A USER DEFINED");
  PUT_LINE("TYPE THAT IS NOT DEFINED IN THE PROTOTYPE PSDL");
  PUT_LINE("SPECIFICATION.  THIS MEANS IT IS EITHER NOT INCLUDED");
  PUT_LINE("AS A SEPARATE TYPE SPECIFICATION, OR IF IT IS, IT IS");
  PUT_LINE("NOT DEFINED AS AN ADA TYPE WITHIN THE TYPE SPECIFICATION");
  PUT_LINE("IT IS ALSO POSSIBLE THAT EITHER 'ARRAY_ELEMENT' OR");
```

```
   PUT_LINE("'ARRAY_INDEX' WAS REQUIRED BUT NOT USED AS IDENTIFIERS.");
   NEW_LINE;
   raise ;

  when INVALID_ADA_TYPE =>
   PUT_LINE("*** ERROR:  INVALID ADA TYPE DISCOVERED FOR -");
   PUT_LINE("          " & GV_NAME.NAME.S & " : " & GV_UDT_NAME.NAME.S);
   PUT_LINE("A Type MUST BE DEFINED AS AN ADA TYPE.  IT CAN REFERENCE");
   PUT_LINE("INFORMATION IN ITS GENERIC PARAMETERS, BUT CANNOT");
   PUT_LINE("REFERENCE DEFINITION INFORMATION OUTSIDE ITS OWN");
   PUT_LINE("SPECIFICATION.  A Generic PARAMETER MUST BE FULLY ");
   PUT_LINE("DEFINED AS AN ADA TYPE.  IT CAN'T EVEN REFERENCE OTHER ");
   PUT_LINE("DEFINITIONS WITHIN ITS GENERICS.");
   NEW_LINE;
   raise ;

  when ADD_ADA_TYPE_TO_SIGNATURE_PKG.UNRECOGNIZED_TYPE =>
   raise ;

  when others =>
   PUT_LINE("*** UNKNOWN ERROR:  OCCURED IN procedure ");
   PUT_LINE("                GET_IN_PARAMETER.    ");
   NEW_LINE;
   raise ;

end GET_IN_PARAMETER;




-- procedure passed to generic scan procedure in generic map package
-- extracts the output parameters from the Type Declaration
-- map.  NOTE that this procedure will be run once for each output
-- parameter as the entire set of output parameters is iterated through.
procedure GET_OUT_PARAMETER(ID      : in PSDL_ID_PKG.PSDL_ID;
                            TYPE_NAME : in PSDL_CONCRETE_TYPE_PKG.TYPE_NAME) is

  ADT_TYPE_NAME, GENERIC_TYPE_NAME, THE_TYPE_NAME, UDT_TYPE_NAME :
   PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
  -- User Defined Type (UDT) type name
  IO_SWITCH                                :
   IO_SWITCH_CLASSES;
  GENERIC_TYPE, FOUND_TYPE                          : BOOLEAN;


begin
  IO_SWITCH := OUTPUT_PARAMETER;
  THE_TYPE_NAME := TYPE_NAME;
  GENERIC_TYPE := FALSE;

  if RECOGNIZED_ADA_TYPE(THE_TYPE_NAME.NAME) then
  -- Output parameter is a valid Ada type so go ahead and encode
  -- into the current Output_Signature.
```

```
  THE_TYPE_NAME.NAME := A_STRINGS.LOWER_TO_UPPER(THE_TYPE_NAME.NAME);
  ENCODE_ADA_TYPE_INTO_SIGNATURE(THE_TYPE_NAME.NAME, IO_SWITCH,
   GENERIC_TYPE);

 else
 -- Maybe the parameter is defined in the generics of the
 -- Operator component.
  CHECK_GENERICS(OPERATOR_COMPONENT, THE_TYPE_NAME, GENERIC_TYPE_NAME,
   FOUND_TYPE);

  if FOUND_TYPE then
        UDT_TYPE_NAME := GENERIC_TYPE_NAME;
        GENERIC_TYPE := TRUE;

  else
  -- Maybe the parameter is defined in the generics of the
  -- Type component.
        CHECK_GENERICS(MAIN_TYPE_COMPONENT, THE_TYPE_NAME, GENERIC_TYPE_NAME,
         FOUND_TYPE);

        if FOUND_TYPE then
         UDT_TYPE_NAME := GENERIC_TYPE_NAME;
         GENERIC_TYPE := TRUE;

        else
   -- Maybe the parameter is defined in the type_declaration
   -- block of the Type component (i.e. it is one of the Type
   -- component's ADTs.
         CHECK_ADT_ADTS(THE_TYPE_NAME, ADT_TYPE_NAME, FOUND_TYPE);

         if FOUND_TYPE then
          UDT_TYPE_NAME := ADT_TYPE_NAME;

         else   -- The parameter must be a user defined type so
          -- get its type.

          GET_USER_DEFINED_TYPE(THE_TYPE_NAME, UDT_TYPE_NAME);

         end if;
        end if;
  end if;


  UDT_TYPE_NAME.NAME := A_STRINGS.LOWER_TO_UPPER(UDT_TYPE_NAME.NAME);
  -- Go ahead and encode.
  ENCODE_ADA_TYPE_INTO_SIGNATURE(UDT_TYPE_NAME.NAME, IO_SWITCH,
   GENERIC_TYPE);
 end if;


exception
```

```
  when UNDEFINED_USER_DEFINED_TYPE =>
   PUT_LINE("*** ERROR:  " & GV_NAME.NAME.S & " IS A USER DEFINED");
   PUT_LINE("TYPE THAT IS NOT DEFINED IN THE PROTOTYPE PSDL");
   PUT_LINE("SPECIFICATION.  THIS MEANS IT IS EITHER NOT INCLUDED");
   PUT_LINE("AS A SEPARATE TYPE SPECIFICATION, OR IF IT IS, IT IS");
   PUT_LINE("NOT DEFINED AS AN ADA TYPE WITHIN THE TYPE SPECIFICATION");
   PUT_LINE("IT IS ALSO POSSIBLE THAT EITHER 'ARRAY_ELEMENT' OR");
   PUT_LINE("'ARRAY_INDEX' WAS REQUIRED BUT NOT USED AS IDENTIFIERS.");
   NEW_LINE;
   raise ;

  when INVALID_ADA_TYPE =>
   PUT_LINE("*** ERROR:  INVALID ADA TYPE DISCOVERED FOR -");
   PUT_LINE("          " & GV_NAME.NAME.S & " : " & GV_UDT_NAME.NAME.S);
   PUT_LINE("A Type MUST BE DEFINED AS AN ADA TYPE.  IT CAN REFERENCE");
   PUT_LINE("INFORMATION IN ITS GENERIC PARAMETERS, BUT CANNOT");
   PUT_LINE("REFERENCE DEFINITION INFORMATION OUTSIDE ITS OWN");
   PUT_LINE("SPECIFICATION.  A Generic PARAMETER MUST BE FULLY ");
   PUT_LINE("DEFINED AS AN ADA TYPE.  IT CAN'T EVEN REFERENCE OTHER ");
   PUT_LINE("DEFINITIONS WITHIN ITS GENERICS.");
   NEW_LINE;
   raise ;

  when ADD_ADA_TYPE_TO_SIGNATURE_PKG.UNRECOGNIZED_TYPE =>
   raise ;

  when others =>
   PUT_LINE("*** UNKNOWN ERROR:  OCCURED IN procedure ");
   PUT_LINE("                GET_OUT_PARAMETER.    ");
   NEW_LINE;
   raise ;

end GET_OUT_PARAMETER;




 ------------------------------------------------------------------------
 ---                    BEGIN MAIN                                    ---
 ------------------------------------------------------------------------

begin

-- Provides access to the stored Type component.
 RETRIEVE_TYPE_COMPONENT(MAIN_TYPE_COMPONENT);

 -- Provides access to the stored prototype specification.
 RETRIEVE_PROTOTYPE_SPEC(PROTOTYPE_SPEC);


end ITERATE_THROUGH_ADT_OPERATOR_PARAMETERS_PKG;
```

```
*******************************************************************************
*******************************************************************************
*******************************************************************************


-- Filename    / Build_Operator_Wrapper_Pkg.a
-- Date        / 3 Sep 93
-- Author      / Scott Dolgoff
-- System      / Sun SPARCstation
-- Compiler    / Verdix Ada
-- Description / This package creates a "wrapper" package for a
--             / software base component that allows the component
--             / to be integrated into a CAPS prototype system.


with PARAMETER_LIST_PKG;
use  PARAMETER_LIST_PKG;

package BUILD_OPERATOR_WRAPPER_PKG is

-- creates the transformation shell from the valid final mapping
-- between query and software base components. This shell becomes
-- the Ada package the prototype designer works with.
 procedure BUILD_OPERATOR_WRAPPER(QC_IN_PARAM_LIST  : in PARAMETERS;
                                  QC_OUT_PARAM_LIST : in PARAMETERS;
                                  SBC_IN_PARAM_LIST : in PARAMETERS;
                                  SBC_OUT_PARAM_LIST : in PARAMETERS;
                                  GEN_PARAM_LIST    : in PARAMETERS;
                                  QC_OPERATOR_NAME  : in STRING;
                                  SBC_OPERATOR_NAME : in STRING);


end BUILD_OPERATOR_WRAPPER_PKG;



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



with TEXT_IO, ADA_RECOGNIZED_TYPES_PKG, PSDL_ID_PKG;
use  TEXT_IO, ADA_RECOGNIZED_TYPES_PKG, PSDL_ID_PKG;

package body BUILD_OPERATOR_WRAPPER_PKG is

-- create structure for eliminating duplicate "with" statements
-- or duplicate type declarations
 type NODE;
```

```
type DUP_PTR is access NODE;

type NODE is record

 ID   : PSDL_ID_PKG.PSDL_ID;
 NEXT : DUP_PTR := null;

end record;


FILE_PREFIX       : STRING(1 .. 6)  := "proto.";
FILE_SUFFIX       : STRING(1 .. 2)  := ".a";
SBC_PKG_SUFFIX    : STRING(1 .. 3)  := "_SB";
PKG_SUFFIX        : STRING(1 .. 4)  := "_PKG";
GENERIC_PKG_PREFIX : STRING(1 .. 4)  := "TMP_";
DUMMY_SUFFIX      : STRING(1 .. 6)  := "_DUMMY";
ARRAY_SUFFIX      : STRING(1 .. 6)  := "_ARRAY";
THE_FILE          : FILE_TYPE;
SBC_IS_GENERIC    : BOOLEAN         := FALSE;
-- used to align parameters in procedure declarations
STD_ID_MAX_LENGTH  : constant INTEGER := 25;




-- print out blank spaces
procedure SPACES(COUNT : in INTEGER) is

begin

 if COUNT > 0 then

   for INDEX in 1 .. COUNT loop

         PUT(THE_FILE, " ");

   end loop;

 end if;

end SPACES;




-- determines whether the specified ID is already represented in
-- the list.  If it is not, then it is added to the list.
procedure ADD_WITH_MEMBER(NAME : in PSDL_ID_PKG.PSDL_ID;
                          HEAD : in out DUP_PTR) is

 PTR       : DUP_PTR := HEAD;
 INSERT    : DUP_PTR;
```

```
 NOT_A_MEMBER : BOOLEAN := TRUE;

begin

 if (HEAD = null) then
                                        -- initialize
  HEAD := new NODE;
  HEAD.ID := NAME;

 else
 -- see if ID is already in the list

  while ((PTR /= null) and (NOT_A_MEMBER)) loop

       if (PTR.ID.S = NAME.S) then
   -- it's a duplicate
        NOT_A_MEMBER := FALSE;
       end if;

       INSERT := PTR;
       PTR := PTR.NEXT;

  end loop;

  if NOT_A_MEMBER then
  -- add to list

       INSERT.NEXT := new NODE;
       INSERT.NEXT.ID := NAME;

  end if;

 end if;

end ADD_WITH_MEMBER;




-- builds a list of all input and output parameter identifiers
-- which is used to ensure no duplicate "with" statements are
-- made
function BUILD_WITH_LIST(IN_PARAM  : PARAMETERS;
                         OUT_PARAM : PARAMETERS) return DUP_PTR is

 IN_PTR   : PARAMETERS := IN_PARAM;
 OUT_PTR  : PARAMETERS := OUT_PARAM;
 HEAD_PTR : DUP_PTR    := null;

begin

 while (IN_PTR /= null) loop
```

```
 if IN_PTR.HAS_UDT then

       ADD_WITH_MEMBER(IN_PTR.UDT, HEAD_PTR);

 elsif (IN_PTR.THE_TYPE.S = ARRAY_TYPE) then
 -- not a UDT but maybe an array with components defined
 -- by UDTs

       if IN_PTR.ARRAY_ELEMENT_PTR.HAS_UDT then

        ADD_WITH_MEMBER(IN_PTR.ARRAY_ELEMENT_PTR.UDT, HEAD_PTR);

       end if;

       if IN_PTR.ARRAY_INDEX_PTR.HAS_UDT then

        ADD_WITH_MEMBER(IN_PTR.ARRAY_INDEX_PTR.UDT, HEAD_PTR);

       end if;
 end if;

 IN_PTR := IN_PTR.NEXT;

end loop;


while (OUT_PTR /= null) loop

 if OUT_PTR.HAS_UDT then

       ADD_WITH_MEMBER(OUT_PTR.UDT, HEAD_PTR);

 elsif (OUT_PTR.THE_TYPE.S = ARRAY_TYPE) then
 -- not a UDT but maybe an array with components defined
 -- by UDTs

       if OUT_PTR.ARRAY_ELEMENT_PTR.HAS_UDT then

        ADD_WITH_MEMBER(OUT_PTR.ARRAY_ELEMENT_PTR.UDT, HEAD_PTR);

       end if;

       if OUT_PTR.ARRAY_INDEX_PTR.HAS_UDT then

        ADD_WITH_MEMBER(OUT_PTR.ARRAY_INDEX_PTR.UDT, HEAD_PTR);

       end if;
 end if;


 OUT_PTR := OUT_PTR.NEXT;

end loop;
```

```
  return HEAD_PTR;

end BUILD_WITH_LIST;



-- write any "with" statements required due to user defined
-- types (UDTs) and software base component inclusion
procedure WRITE_WITH_STATEMENTS(SBC_OPERATOR_NAME : in STRING;
                                PARAM_LIST       : in DUP_PTR) is

  PTR : DUP_PTR := PARAM_LIST;

begin

  PUT_LINE(THE_FILE, "with " & SBC_OPERATOR_NAME & SBC_PKG_SUFFIX & ";");
  PUT_LINE(THE_FILE, "use  " & SBC_OPERATOR_NAME & SBC_PKG_SUFFIX & ";");

  while PTR /= null loop

    PUT_LINE(THE_FILE, "with " & PTR.ID.S & PKG_SUFFIX & ";");
    PTR := PTR.NEXT;

  end loop;

  NEW_LINE(THE_FILE, 2);

end WRITE_WITH_STATEMENTS;



-- write package declaration
procedure WRITE_PACKAGE_DECLARATION(QC_OPERATOR_NAME : in STRING) is

begin

  PUT_LINE(THE_FILE, "package " & QC_OPERATOR_NAME & PKG_SUFFIX & " is");
  NEW_LINE(THE_FILE);

end WRITE_PACKAGE_DECLARATION;



-- write package body declaration
procedure WRITE_PACKAGE_BODY_DECLARATION(QC_OPERATOR_NAME : in STRING) is
```

```
begin

 PUT_LINE(THE_FILE, "-------------------------------------------");
 PUT_LINE(THE_FILE, "---          PACKAGE BODY          ---");
 PUT_LINE(THE_FILE, "-------------------------------------------");
 NEW_LINE(THE_FILE, 3);
 PUT_LINE(THE_FILE, "package body " & QC_OPERATOR_NAME & PKG_SUFFIX & " is");
 NEW_LINE(THE_FILE);

end WRITE_PACKAGE_BODY_DECLARATION;




-- write the package end statement (for spec or body)
procedure WRITE_END_OF_PACKAGE(QC_OPERATOR_NAME : in STRING) is

begin

 PUT_LINE(THE_FILE, "end " & QC_OPERATOR_NAME & PKG_SUFFIX & ";");
 NEW_LINE(THE_FILE, 3);

end WRITE_END_OF_PACKAGE;




-- writes the new package that is the result of the generic
-- instantiation
procedure WRITE_GENERIC_INSTANTIATION(QC_OPERATOR_NAME  : in STRING;
                                      SBC_OPERATOR_NAME : in STRING;
                                      GEN_PARAM_LIST    : in PARAMETERS) is

 PTR : PARAMETERS := GEN_PARAM_LIST;

begin

 PUT_LINE(THE_FILE, "   package " & GENERIC_PKG_PREFIX & QC_OPERATOR_NAME &
  PKG_SUFFIX & " is new " & SBC_OPERATOR_NAME & SBC_PKG_SUFFIX & "(");
 while (PTR /= null) loop

  PUT(THE_FILE, "      ");
  PUT(THE_FILE, PTR.ID.S & " => ");
  if (PTR.MAPPING.HAS_UDT) then   -- qc parameter defined by UDT

       PUT(THE_FILE, PTR.MAPPING.UDT.S & PKG_SUFFIX & "." & PTR.MAPPING.UDT.S);
```

```
  elsif (PTR.MAPPING.THE_TYPE.S = ARRAY_TYPE) then

        PUT(THE_FILE, PTR.MAPPING.ID.S & ARRAY_SUFFIX);

  else

        PUT(THE_FILE, PTR.MAPPING.THE_TYPE.S);

  end if;

  PTR := PTR.NEXT;
  if (PTR /= null) then
        PUT_LINE(THE_FILE, ",");
  else
        PUT_LINE(THE_FILE, ");");
  end if;

 end loop;

end WRITE_GENERIC_INSTANTIATION;



-- define an array type declaration
procedure WRITE_ARRAY_TYPE_DECL(NAME : in STRING; PTR : in PARAMETERS) is

begin
 PUT(THE_FILE, "   type " & NAME & ARRAY_SUFFIX & " is array (");

 -- write the array index
 if PTR.DEFINED_BY_GENERIC_TYPE then
 -- refer to the qc parameter that instantiated the generic

  if PTR.GENERIC_LINK.MAPPING.ARRAY_INDEX_PTR.HAS_UDT then
        PUT(THE_FILE, PTR.GENERIC_LINK.MAPPING.ARRAY_INDEX_PTR.UDT.S & PKG_SUFFIX &
        "." & PTR.GENERIC_LINK.MAPPING.ARRAY_INDEX_PTR.UDT.S & ") of ");

  else
        PUT(THE_FILE, PTR.GENERIC_LINK.MAPPING.ARRAY_INDEX_PTR.THE_TYPE.S &
        ") of ");

  end if;

 else   -- array index not defined by a generic

  if PTR.ARRAY_INDEX_PTR.HAS_UDT then
        PUT(THE_FILE, PTR.ARRAY_INDEX_PTR.UDT.S & PKG_SUFFIX & "." &
        PTR.ARRAY_INDEX_PTR.UDT.S & ") of ");

  else
        PUT(THE_FILE, PTR.ARRAY_INDEX_PTR.THE_TYPE.S & ") of ");

  end if;
 end if;
```

```
  -- write the array element
  if PTR.DEFINED_BY_GENERIC_TYPE then
  -- refer to the qc parameter that instantiated the generic

   if PTR.GENERIC_LINK.MAPPING.ARRAY_ELEMENT_PTR.HAS_UDT then
         PUT(THE_FILE, PTR.GENERIC_LINK.MAPPING.ARRAY_ELEMENT_PTR.UDT.S & PKG_SUFFIX
&
          "." & PTR.GENERIC_LINK.MAPPING.ARRAY_ELEMENT_PTR.UDT.S & ";");

   else
         PUT(THE_FILE, PTR.GENERIC_LINK.MAPPING.ARRAY_ELEMENT_PTR.THE_TYPE.S &
          ";");

   end if;

  else   -- array element not defined by a generic

   if PTR.ARRAY_ELEMENT_PTR.HAS_UDT then
         PUT(THE_FILE, PTR.ARRAY_ELEMENT_PTR.UDT.S & PKG_SUFFIX & "." &
          PTR.ARRAY_ELEMENT_PTR.UDT.S & ";");

   else
         PUT(THE_FILE, PTR.ARRAY_ELEMENT_PTR.THE_TYPE.S & ";");

   end if;
  end if;

  NEW_LINE(THE_FILE);

end WRITE_ARRAY_TYPE_DECL;




-- write software base component type declaration
-- it is only needed for dummy output parameters
procedure WRITE_SBC_TYPE_DECLARATION(PTR_VAR : in PARAMETERS) is

  PTR : PARAMETERS := PTR_VAR;

begin

  while (PTR /= null) loop

   if ((PTR.MAPPING = null) and (not PTR.DEFINED_BY_GENERIC_TYPE)) then
   -- unmatched, therefore must be a dummy output parameter
   -- if it is defined by a generic, then we don't need to
   --   declare it because we will refer to the instantiation
   --   parameter in the query component
```

```
   -- see if it's an array
         if (PTR.THE_TYPE.S = ARRAY_TYPE) then
          PUT(THE_FILE, "  ");
          WRITE_ARRAY_TYPE_DECL(PTR.ID.S, PTR);
         end if;

   end if;

   PTR := PTR.NEXT;

  end loop;

end WRITE_SBC_TYPE_DECLARATION;
```

```
-- write query component type declaration
procedure WRITE_QC_TYPE_DECLARATION(PTR_VAR : in PARAMETERS) is

 PTR : PARAMETERS := PTR_VAR;

begin

 while (PTR /= null) loop

 -- see if it's an array
 -- however, if it is an array and defined by a UDT then
 -- the array type is already defined and we don't need
 -- a type declaration for it
  if ((PTR.THE_TYPE.S = ARRAY_TYPE) and (not PTR.HAS_UDT)) then
        WRITE_ARRAY_TYPE_DECL(PTR.ID.S, PTR);
  end if;

  PTR := PTR.NEXT;

 end loop;

end WRITE_QC_TYPE_DECLARATION;
```

```
-- Call the software base component Operator procedure.
-- The order of the parameters must exactly match the software
-- base component PSDL specification.  Dummy parameters have been
```

```
-- predeclared (outputs only are possible) using the formal name
-- used by the software base component.
procedure WRITE_CALL_SBC_PROCEDURE(QC_OPERATOR_NAME  : in STRING;
                                   SBC_OPERATOR_NAME  : in STRING;
                                   SBC_IN_PARAM_LIST  : in PARAMETERS;
                                   SBC_OUT_PARAM_LIST : in PARAMETERS) is

 PTR : PARAMETERS;

begin

 PUT(THE_FILE, "       ");
 if SBC_IS_GENERIC then
  PUT_LINE(THE_FILE, GENERIC_PKG_PREFIX & QC_OPERATOR_NAME & PKG_SUFFIX &
   "." & SBC_OPERATOR_NAME & "(");
 else
  PUT_LINE(THE_FILE, SBC_OPERATOR_NAME & SBC_PKG_SUFFIX & "." &
   SBC_OPERATOR_NAME & "(");
 end if;

 -- input parameters
 PTR := SBC_IN_PARAM_LIST;
 while (PTR /= null) loop

  PUT(THE_FILE, "           ");
  PUT(THE_FILE, PTR.MAPPING.ID.S);

  PTR := PTR.NEXT;
  if (PTR /= null) then
        PUT_LINE(THE_FILE, ",");
  else
        if (SBC_OUT_PARAM_LIST = null) then
         PUT_LINE(THE_FILE, ");");
        else
         PUT_LINE(THE_FILE, ",");
        end if;
  end if;

 end loop;


 -- output parameters
 PTR := SBC_OUT_PARAM_LIST;
 while (PTR /= null) loop

  PUT(THE_FILE, "           ");
  if (PTR.MAPPING = null) then
  -- dummy output parameter
        PUT(THE_FILE, PTR.ID.S & DUMMY_SUFFIX);
  else
        PUT(THE_FILE, PTR.MAPPING.ID.S);
  end if;

  PTR := PTR.NEXT;
  if (PTR /= null) then
```

```
        PUT_LINE(THE_FILE, ",");
    else
        PUT_LINE(THE_FILE, ");");
    end if;

  end loop;

  NEW_LINE(THE_FILE);

end WRITE_CALL_SBC_PROCEDURE;




-- define procedure specification
-- The order of the parameters only needs to match up with the
-- query PSDL specification since it's the implementation portion
-- of this procedure that will actually invoke the software base
-- component Operator procedure.
procedure WRITE_PROCEDURE_SPEC(QC_OPERATOR_NAME  : in STRING;
                               QC_IN_PARAM_LIST  : in PARAMETERS;
                               QC_OUT_PARAM_LIST : in PARAMETERS) is
  PTR : PARAMETERS;

begin

  NEW_LINE(THE_FILE);
  PUT_LINE(THE_FILE, "   procedure " & QC_OPERATOR_NAME & "(");

  -- input parameters
  PTR := QC_IN_PARAM_LIST;
  while (PTR /= null) loop

    PUT(THE_FILE, "      ");
    PUT(THE_FILE, PTR.ID.S);
    SPACES(STD_ID_MAX_LENGTH - PTR.ID.S'LENGTH);
    PUT(THE_FILE, " : in    ");
    if (PTR.HAS_UDT) then   -- qc parameter defined by UDT

        PUT(THE_FILE, PTR.UDT.S & PKG_SUFFIX & "." & PTR.UDT.S);

    elsif (PTR.THE_TYPE.S = ARRAY_TYPE) then
    -- use defined array type declaration
        PUT(THE_FILE, PTR.ID.S & ARRAY_SUFFIX);

    else

        PUT(THE_FILE, PTR.THE_TYPE.S);

    end if;

    PTR := PTR.NEXT;
```

```
  if (PTR /= null) then
        PUT_LINE(THE_FILE, ";");
  else
        if (QC_OUT_PARAM_LIST = null) then
          PUT_LINE(THE_FILE, ");");
        else
          PUT_LINE(THE_FILE, ";");
        end if;
  end if;

 end loop;


 -- output parameters
 PTR := QC_OUT_PARAM_LIST;
 while (PTR /= null) loop

  PUT(THE_FILE, "      ");
  PUT(THE_FILE, PTR.ID.S);
  SPACES(STD_ID_MAX_LENGTH - PTR.ID.S'LENGTH);
  PUT(THE_FILE, " :   out ");
  if (PTR.HAS_UDT) then   -- qc parameter defined by UDT

        PUT(THE_FILE, PTR.UDT.S & PKG_SUFFIX & "." & PTR.UDT.S);

  elsif (PTR.THE_TYPE.S = ARRAY_TYPE) then
  -- use defined array type declaration
        PUT(THE_FILE, PTR.ID.S & ARRAY_SUFFIX);

  else

        PUT(THE_FILE, PTR.THE_TYPE.S);

  end if;

  PTR := PTR.NEXT;
  if (PTR /= null) then
        PUT_LINE(THE_FILE, ";");
  else
        PUT_LINE(THE_FILE, ");");
  end if;

 end loop;

end WRITE_PROCEDURE_SPEC;




-- define procedure body
-- The order of the parameters only needs to match up with the
-- query PSDL specification since it's within the body
-- of this procedure that the software base component Operator
```

```
-- procedure is invoked.
procedure WRITE_PROCEDURE_BODY(QC_OPERATOR_NAME  : in STRING;
                               SBC_OPERATOR_NAME : in STRING;
                               QC_IN_PARAM_LIST  : in PARAMETERS;
                               QC_OUT_PARAM_LIST : in PARAMETERS;
                               SBC_IN_PARAM_LIST : in PARAMETERS;
                               SBC_OUT_PARAM_LIST : in PARAMETERS) is

 PTR  : PARAMETERS;
 PTR2 : DUP_PTR;

begin

 NEW_LINE(THE_FILE);
 PUT_LINE(THE_FILE, "   procedure " & QC_OPERATOR_NAME & "(");

 -- input parameters
 PTR := QC_IN_PARAM_LIST;
 while (PTR /= null) loop

  PUT(THE_FILE, "       ");
  PUT(THE_FILE, PTR.ID.S);
  SPACES(STD_ID_MAX_LENGTH - PTR.ID.S'LENGTH);
  PUT(THE_FILE, " : in     ");
  if (PTR.HAS_UDT) then   -- qc parameter defined by UDT

        PUT(THE_FILE, PTR.UDT.S & PKG_SUFFIX & "." & PTR.UDT.S);

  elsif (PTR.THE_TYPE.S = ARRAY_TYPE) then
  -- use defined array type declaration
        PUT(THE_FILE, PTR.ID.S & ARRAY_SUFFIX);

  else

        PUT(THE_FILE, PTR.THE_TYPE.S);

  end if;

  PTR := PTR.NEXT;
  if (PTR /= null) then
        PUT_LINE(THE_FILE, ";");
  else
        if (QC_OUT_PARAM_LIST = null) then
          PUT_LINE(THE_FILE, ") is");
        else
          PUT_LINE(THE_FILE, ";");
        end if;
  end if;

 end loop;


 -- output parameters
 PTR := QC_OUT_PARAM_LIST;
 while (PTR /= null) loop
```

```
PUT(THE_FILE, "      ");
PUT(THE_FILE, PTR.ID.S);
SPACES(STD_ID_MAX_LENGTH - PTR.ID.S'LENGTH);
PUT(THE_FILE, " :   out ");
if (PTR.HAS_UDT) then   -- qc parameter defined by UDT

      PUT(THE_FILE, PTR.UDT.S & PKG_SUFFIX & "." & PTR.UDT.S);

elsif (PTR.THE_TYPE.S = ARRAY_TYPE) then
-- use defined array type declaration
      PUT(THE_FILE, PTR.ID.S & ARRAY_SUFFIX);

else

      PUT(THE_FILE, PTR.THE_TYPE.S);

end if;

PTR := PTR.NEXT;
if (PTR /= null) then
      PUT_LINE(THE_FILE, ";");
else
      PUT_LINE(THE_FILE, ") is");
end if;

end loop;

NEW_LINE(THE_FILE);

-- see if dummy output parameters will be needed
if (PARAMETER_COUNT(QC_OUT_PARAM_LIST) <
 PARAMETER_COUNT(SBC_OUT_PARAM_LIST)) then
-- we must include dummy output parameters
-- create them and define them in procedure as variables

PUT_LINE(THE_FILE, "    -- dummy output parameters");

-- predefine any undefined array type declarations
WRITE_SBC_TYPE_DECLARATION(SBC_OUT_PARAM_LIST);

-- get dummy (unmatched) parameters
PTR := SBC_OUT_PARAM_LIST;
while (PTR /= null) loop

      if (PTR.MAPPING = null) then
  -- unmatched, therefore must be a dummy output parameter
  -- Note: for this implementation, no software base component
  --       parameter can be defined as a UDT unless it was
  --       a generic instantiated via a query UDT.

        PUT(THE_FILE, "    " & PTR.ID.S & DUMMY_SUFFIX & " : ");
```

```
        if PTR.DEFINED_BY_GENERIC_TYPE then
    -- get instantiated generic type

          if (PTR.GENERIC_LINK.MAPPING.THE_TYPE.S = ARRAY_TYPE) then
     -- see if the array is defined by a UDT
           if (PTR.GENERIC_LINK.MAPPING.HAS_UDT) then
                PUT(THE_FILE, PTR.GENERIC_LINK.MAPPING.UDT.S & PKG_SUFFIX & "." &
                 PTR.GENERIC_LINK.MAPPING.UDT.S);

           else
      -- use defined array type declaration
                PUT(THE_FILE, PTR.GENERIC_LINK.MAPPING.ID.S & ARRAY_SUFFIX);

           end if;

          else          -- not an array type

     -- see if the parameter is defined by a UDT
           if (PTR.GENERIC_LINK.MAPPING.HAS_UDT) then
                PUT(THE_FILE, PTR.GENERIC_LINK.MAPPING.UDT.S & PKG_SUFFIX & "." &
                 PTR.GENERIC_LINK.MAPPING.UDT.S);

           else
                PUT(THE_FILE, PTR.GENERIC_LINK.MAPPING.THE_TYPE.S);

           end if;
          end if;


        else            -- not defined by a generic

    -- see if its an array
          if (PTR.THE_TYPE.S = ARRAY_TYPE) then
     -- use defined array type declaration
           PUT(THE_FILE, PTR.ID.S & ARRAY_SUFFIX);

          else          -- not an array type

           PUT(THE_FILE, PTR.THE_TYPE.S);

          end if;
         end if;

        PUT_LINE(THE_FILE, ";");

       end if;

       PTR := PTR.NEXT;

  end loop;
end if;
```

```
 NEW_LINE(THE_FILE);

 -- procedure body begin statement
 NEW_LINE(THE_FILE);
 PUT_LINE(THE_FILE, "    begin");
 NEW_LINE(THE_FILE);

 -- call the software base component Operator procedure
 WRITE_CALL_SBC_PROCEDURE(QC_OPERATOR_NAME, SBC_OPERATOR_NAME,
  SBC_IN_PARAM_LIST, SBC_OUT_PARAM_LIST);

 -- write the end of the procedure body
 PUT_LINE(THE_FILE, "    end " & QC_OPERATOR_NAME & ";");
 NEW_LINE(THE_FILE);

end WRITE_PROCEDURE_BODY;




-- creates the "wrapper" package from the valid final mapping
-- between query and software base components.  This shell becomes
-- the Ada package the prototype designer works with.
procedure BUILD_OPERATOR_WRAPPER(QC_IN_PARAM_LIST  : in PARAMETERS;
                                 QC_OUT_PARAM_LIST : in PARAMETERS;
                                 SBC_IN_PARAM_LIST : in PARAMETERS;
                                 SBC_OUT_PARAM_LIST: in PARAMETERS;
                                 GEN_PARAM_LIST    : in PARAMETERS;
                                 QC_OPERATOR_NAME  : in STRING;
                                 SBC_OPERATOR_NAME : in STRING) is

 PTR : DUP_PTR;

begin

-- open the file in which the shell will be written to
 CREATE(THE_FILE, MODE => OUT_FILE, NAME => QC_OPERATOR_NAME & PKG_SUFFIX &
  FILE_SUFFIX);

 -- write any "with" statements required due to user defined
 -- types (UDTs) and software base component inclusion
 PTR := BUILD_WITH_LIST(QC_IN_PARAM_LIST, QC_OUT_PARAM_LIST);
 WRITE_WITH_STATEMENTS(SBC_OPERATOR_NAME, PTR);

 -- write package declaration
 WRITE_PACKAGE_DECLARATION(QC_OPERATOR_NAME);

 -- predefine all array type declarations
 WRITE_QC_TYPE_DECLARATION(QC_IN_PARAM_LIST);
 WRITE_QC_TYPE_DECLARATION(QC_OUT_PARAM_LIST);
```

```
   NEW_LINE(THE_FILE, 2);


   if (GEN_PARAM_LIST /= null) then   -- sbc has generic parameters

    SBC_IS_GENERIC := TRUE;
    WRITE_GENERIC_INSTANTIATION(QC_OPERATOR_NAME, SBC_OPERATOR_NAME,
     GEN_PARAM_LIST);
   end if;

   -- define procedure specification
   WRITE_PROCEDURE_SPEC(QC_OPERATOR_NAME, QC_IN_PARAM_LIST, QC_OUT_PARAM_LIST);

   -- write the package end statement
   WRITE_END_OF_PACKAGE(QC_OPERATOR_NAME);

   -- create a package body with procedure implementation
   WRITE_PACKAGE_BODY_DECLARATION(QC_OPERATOR_NAME);
   WRITE_PROCEDURE_BODY(QC_OPERATOR_NAME, SBC_OPERATOR_NAME, QC_IN_PARAM_LIST,
    QC_OUT_PARAM_LIST, SBC_IN_PARAM_LIST, SBC_OUT_PARAM_LIST);


   -- write the package body end statement
   WRITE_END_OF_PACKAGE(QC_OPERATOR_NAME);


   CLOSE(THE_FILE);

 end BUILD_OPERATOR_WRAPPER;

end BUILD_OPERATOR_WRAPPER_PKG;


*******************************************************************************
*******************************************************************************
*******************************************************************************


-- Filename    | Create_Operator_Parameter_Files.a
-- Date        | 10 Aug 93
-- Author      | Scott Dolgoff
-- System      | Sun SPARCstation
-- Compiler    | Verdix Ada
-- Description | This program drives the writing of the input and output
--             | parameters of software component's  Prototype System
--             | Description Language (PSDL) specifications into
--             | two separate files.  These input/output parameter
--             | files are later used by the graphical user interface
--             | written with TAE to help transform components selected
```

```
--          / through matching to be brought into the prototype
--          / working directory.  If the PSDL specification is of
--          / a software base component, then a check is made to
--          / see if it has generic parameters.  If it does
--          / then the generic parameters are written to a third
--          / file.


with PARAMETER_LIST_PKG, PSDL_ID_PKG;
use  PARAMETER_LIST_PKG, PSDL_ID_PKG;

package CREATE_OPERATOR_PARAMETER_FILES_PKG is


 type COMPONENT_STATUS is (QUERY_COMPONENT, SOFTWARE_BASE_COMPONENT);

 procedure CREATE_OPERATOR_PARAMETER_FILES(THE_COMPONENT  : in COMPONENT_STATUS;
                                           IN_PARAM_LIST  : in out PARAMETERS;
                                           IN_PARAM_COUNT : out INTEGER;
                                           OUT_PARAM_LIST : in out PARAMETERS;
                                           OUT_PARAM_COUNT : out INTEGER;
                                           GEN_PARAM_LIST : in out PARAMETERS;
                                           GEN_PARAM_COUNT : out INTEGER;
                                           HAS_GENERICS   : out BOOLEAN;
                                           OP_NAME        : out PSDL_ID);

end CREATE_OPERATOR_PARAMETER_FILES_PKG;



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


with TEXT_IO, PSDL_CONCRETE_TYPE_PKG, LOAD_PSDL_INTO_ADA_STRUCTURE_PKG,
 ITERATE2_THROUGH_OPERATOR_PARAMETERS_PKG,                    PSDL_PROGRAM_PKG,
PSDL_COMPONENT_PKG;

use  TEXT_IO, PSDL_CONCRETE_TYPE_PKG, LOAD_PSDL_INTO_ADA_STRUCTURE_PKG,
 ITERATE2_THROUGH_OPERATOR_PARAMETERS_PKG,                    PSDL_PROGRAM_PKG,
PSDL_COMPONENT_PKG;


package body CREATE_OPERATOR_PARAMETER_FILES_PKG is

 OPERATOR_COMPONENT : PSDL_COMPONENT_PKG.OPERATOR;


 procedure CREATE_OPERATOR_PARAMETER_FILES(THE_COMPONENT  : in COMPONENT_STATUS;
                                           IN_PARAM_LIST  : in out PARAMETERS;
                                           IN_PARAM_COUNT : out INTEGER;
                                           OUT_PARAM_LIST : in out PARAMETERS;
                                           OUT_PARAM_COUNT : out INTEGER;
```

```
                                        GEN_PARAM_LIST  : in out PARAMETERS;
                                        GEN_PARAM_COUNT : out INTEGER;
                                        HAS_GENERICS    : out BOOLEAN;
                                        OP_NAME         : out PSDL_ID_PKG.PSDL_ID) is

 IN_STANDARD_NAME                      : STRING(1 .. 20) :=
  "input_parameters.txt";
 OUT_STANDARD_NAME                      : STRING(1 .. 21) :=
  "output_parameters.txt";
 GEN_STANDARD_NAME                      : STRING(1 .. 22) :=
  "generic_parameters.txt";
 FILE_SUFFIX                        : STRING(1 .. 4);
 PTR, GEN_PTR                        : PARAMETERS;
 SET_LINK                           : BOOLEAN;
 GENERIC_MODEL, OUTPUT_PARAMETERS, INPUT_PARAMETERS :
  PSDL_CONCRETE_TYPE_PKG.TYPE_DECLARATION;
 PROTOTYPE_SPEC                          :
  PSDL_PROGRAM_PKG.PSDL_PROGRAM;


begin

-- get the Operator component
 RETRIEVE_OPERATOR_COMPONENT(OPERATOR_COMPONENT);

 -- get Operator name
 OP_NAME := PSDL_COMPONENT_PKG.NAME(OPERATOR_COMPONENT);

 -- get the Prototype Specification
 RETRIEVE_PROTOTYPE_SPEC(PROTOTYPE_SPEC);

 -- Pass the components to the packages that will need
 -- them
 SEND_COMPONENTS(OPERATOR_COMPONENT, PROTOTYPE_SPEC);


 -- Pass the file suffix so the correct files can be
 -- written to.
 if THE_COMPONENT = QUERY_COMPONENT then
  FILE_SUFFIX := "qyc_";
 else
  FILE_SUFFIX := "sbc_";
 end if;


 -- open the input parameters file
 OPEN_FILE(FILE_SUFFIX & IN_STANDARD_NAME);

 -- get the Operator's input parameters
 INPUT_PARAMETERS := PSDL_COMPONENT_PKG.INPUTS(OPERATOR_COMPONENT);

 -- iterate through the input parameters to store the
 -- input parameters in a text file
```

```
ITERATE_THROUGH_INPUT_PARAMETERS(INPUT_PARAMETERS);

CLOSE_FILE;

-- build list of all input parameters and how many there are
BUILD_PARAMETER_LIST(IN_PARAM_LIST, IN_PARAM_COUNT);


-- open the output parameters file
OPEN_FILE(FILE_SUFFIX & OUT_STANDARD_NAME);

-- get the Operator's output parameters
OUTPUT_PARAMETERS := PSDL_COMPONENT_PKG.OUTPUTS(OPERATOR_COMPONENT);

-- iterate through the output parameters to store the
-- output parameters in a text file
ITERATE_THROUGH_OUTPUT_PARAMETERS(OUTPUT_PARAMETERS);

CLOSE_FILE;

-- build list of all output parameters and how many there are
BUILD_PARAMETER_LIST(OUT_PARAM_LIST, OUT_PARAM_COUNT);



-- Check the component status. If it is a software base component
-- then it may have generic parameters.
HAS_GENERICS := FALSE;
GEN_PARAM_LIST := null;
GEN_PARAM_COUNT := 0;
if THE_COMPONENT = SOFTWARE_BASE_COMPONENT then

--  Get the generic parameters.
 GENERIC_MODEL :=
  PSDL_COMPONENT_PKG.GENERIC_PARAMETERS(OPERATOR_COMPONENT);

 if not PSDL_CONCRETE_TYPE_PKG.EQUAL(GENERIC_MODEL,
  PSDL_CONCRETE_TYPE_PKG.EMPTY_TYPE_DECLARATION) then

 -- it has generic parameters
       HAS_GENERICS := TRUE;

   -- open the input parameters file
       OPEN_FILE(FILE_SUFFIX & GEN_STANDARD_NAME);

   -- iterate through the generic parameters to store the
   -- generic parameters in a text file
       ITERATE_THROUGH_GENERIC_PARAMETERS(GENERIC_MODEL);

       CLOSE_FILE;
```

```
-- build list of all generic parameters and how many there are
    BUILD_PARAMETER_LIST(GEN_PARAM_LIST, GEN_PARAM_COUNT);

-- establish the links between the generic parameters and the
-- SBC input parameters defined in terms of those
-- generic parameters
    PTR := IN_PARAM_LIST;
    while (PTR /= null) loop

     if PTR.DEFINED_BY_GENERIC_TYPE then
 -- find the generic and set the link

       GEN_PTR := GEN_PARAM_LIST;
       SET_LINK := FALSE;
       while ((GEN_PTR /= null) and (not SET_LINK)) loop

        if (GEN_PTR.ID.S = PTR.GENERIC_ID.S) then
              SET_LINK := TRUE;
     -- link set
              PTR.GENERIC_LINK := GEN_PTR;
        end if;

        GEN_PTR := GEN_PTR.NEXT;

       end loop;

      end if;

      PTR := PTR.NEXT;

    end loop;


-- establish the links between the generic parameters and the
-- SBC output parameters defined in terms of those
-- generic parameters
    PTR := OUT_PARAM_LIST;
    while (PTR /= null) loop

     if PTR.DEFINED_BY_GENERIC_TYPE then
 -- find the generic and set the link

       GEN_PTR := GEN_PARAM_LIST;
       SET_LINK := FALSE;
       while ((GEN_PTR /= null) and (not SET_LINK)) loop

        if (GEN_PTR.ID.S = PTR.GENERIC_ID.S) then
              SET_LINK := TRUE;
              PTR.GENERIC_LINK := GEN_PTR;
        end if;

        GEN_PTR := GEN_PTR.NEXT;
```

```
            end loop;

          end if;

          PTR := PTR.NEXT;

        end loop;

    end if;
  end if;


 exception

  when others =>
   PUT_LINE("*** THIS PROGRAM CANNOT CONTINUE ***");
   raise ;

 end CREATE_OPERATOR_PARAMETER_FILES;

end CREATE_OPERATOR_PARAMETER_FILES_PKG;
```

```
*******************************************************************************
*******************************************************************************
*******************************************************************************
```

```
-- Filename    / Determine_Ada_Type_Pkg.a
-- Date        / 9 August 93
-- Author      / Scott Dolgoff
-- System      / Sun SPARCstation
-- Compiler    / Verdix Ada
-- Description / This program determines the Ada type of a parameter
--             / type that was not defined as an Ada type, but rather
--             / in terms of an ADT, generic, or user defined type.


with PSDL_PROGRAM_PKG, PSDL_COMPONENT_PKG, PSDL_CONCRETE_TYPE_PKG, PSDL_ID_PKG;
use  PSDL_PROGRAM_PKG, PSDL_COMPONENT_PKG, PSDL_CONCRETE_TYPE_PKG, PSDL_ID_PKG;
package DETERMINE_THE_ADA_TYPE_PKG is


 type COMPONENT_TYPE is (OPERATOR_COMP, TYPE_COMP);

 -- Input parameter is a user defined type so we must look
 -- up its Ada type representation by extracting the
 -- user defined type from the prototype PSDL specification.
 procedure                GET_USER_DEFINED_TYPE(THE_TYPE_NAME                :                in
PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
```

```
                                                UDT_TYPE_NAME        :        in        out
PSDL_CONCRETE_TYPE_PKG.TYPE_NAME);

 -- Looks through the generic parameters of a Type or Operator to
 -- see if an unrecognized type is defined as an Ada type there.
 procedure CHECK_GENERICS(THE_COMPONENT    : in COMPONENT_TYPE;
                                                UNKNOWN_TYPE_NAME        :        in        out
PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
                          GENERIC_TYPE_NAME : in out PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
                          FOUND_TYPE      : out BOOLEAN);


 -- Called if a type is an ARRAY which means that in order to
 -- correctly build the signature, not only must the ARRAY
 -- be encoded into the signature, but also the ARRAY components.
 procedure      GET_ARRAY_COMPONENTS(TYPE_NAME                                    :        in
PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
                                    ELEMENT_TYPE  : out PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
                                    ELEMENT_IS_UDT : out BOOLEAN;
                                    ELEMENT_UDT   : out PSDL_ID_PKG.PSDL_ID;
                                    INDEX_TYPE    : out PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
                                    INDEX_IS_UDT  : out BOOLEAN;
                                    INDEX_UDT     : out PSDL_ID_PKG.PSDL_ID;
                                    ARRAY_IN_UDT  : in BOOLEAN);


 -- make the operator or type component and the prototype component
 -- visible to this package
 procedure                PASS_COMPONENTS(MAIN_COMPONENT                    :                in
PSDL_COMPONENT_PKG.PSDL_COMPONENT;
                           PROTO_SPEC    : in PSDL_PROGRAM_PKG.PSDL_PROGRAM);


 -- exceptions
 --  the user defined type of an Operator input or output
 --  parameter was not defined in the Prototype PSDL specification
 UNDEFINED_USER_DEFINED_TYPE : exception;
 --  the type used to define a user defined type is not a
 --  valid Ada type
 INVALID_ADA_TYPE        : exception;

end DETERMINE_THE_ADA_TYPE_PKG;


------------------------------------------------------------
------------------------------------------------------------
```

```
with TEXT_IO, ADA_RECOGNIZED_TYPES_PKG, LOAD_PSDL_INTO_ADA_STRUCTURE_PKG,
 TYPE_NAME_PKG, A_STRINGS;

use  TEXT_IO, ADA_RECOGNIZED_TYPES_PKG, LOAD_PSDL_INTO_ADA_STRUCTURE_PKG,
 A_STRINGS;

package body DETERMINE_THE_ADA_TYPE_PKG is

 GV_NAME, GV_UDT_NAME                          :
  PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
 PROTOTYPE_SPEC                          :
  PSDL_PROGRAM_PKG.PSDL_PROGRAM;
 MAIN_TYPE_COMPONENT, TYPE_COMPONENT_WITH_ARRAY, TYPE_COMPONENT :
  PSDL_COMPONENT_PKG.DATA_TYPE;
 OPERATOR_COMPONENT                          :
  PSDL_COMPONENT_PKG.OPERATOR;
 KIND_OF_COMPONENT                          :
  PSDL_COMPONENT_PKG.COMPONENT_TYPE;




 -- make the operator or type component and the prototype component
 -- visible to this package
 procedure                PASS_COMPONENTS(MAIN_COMPONENT                :                in
PSDL_COMPONENT_PKG.PSDL_COMPONENT;
                              PROTO_SPEC    : in PSDL_PROGRAM_PKG.PSDL_PROGRAM) is

 begin
  KIND_OF_COMPONENT := COMPONENT_CATEGORY(MAIN_COMPONENT);
  case KIND_OF_COMPONENT is

   when PSDL_OPERATOR =>
        OPERATOR_COMPONENT := MAIN_COMPONENT;

   when PSDL_TYPE =>
        MAIN_TYPE_COMPONENT := MAIN_COMPONENT;

  end case;

  PROTOTYPE_SPEC := PROTO_SPEC;

 end PASS_COMPONENTS;




 -- Input parameter is a user defined type so we must look
 -- up its Ada type representation by extracting the
 -- user defined type from the prototype PSDL specification.
 procedure                GET_USER_DEFINED_TYPE(THE_TYPE_NAME                :                in
PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
```

```
                                        UDT_TYPE_NAME      :      in      out
PSDL_CONCRETE_TYPE_PKG.TYPE_NAME) is

  TYPE_MODEL : PSDL_CONCRETE_TYPE_PKG.TYPE_DECLARATION;

 begin

 -- retrieve the type component specification
  TYPE_COMPONENT := PSDL_PROGRAM_PKG.FETCH(PROTOTYPE_SPEC,
   THE_TYPE_NAME.NAME);
  if TYPE_COMPONENT = null then
    GV_NAME := THE_TYPE_NAME;
    raise UNDEFINED_USER_DEFINED_TYPE;
  end if;

  -- get the corresponding Ada type of the user defined type
  TYPE_MODEL := PSDL_COMPONENT_PKG.MODEL(TYPE_COMPONENT);

  UDT_TYPE_NAME :=
   PSDL_CONCRETE_TYPE_PKG.TYPE_DECLARATION_PKG.FETCH(TYPE_MODEL,
   THE_TYPE_NAME.NAME);

  -- Ensure type was defined.  If not, then nothing was fetched.
  if TYPE_NAME_PKG."=" (UDT_TYPE_NAME, TYPE_NAME_PKG.NULL_TYPE) then
    GV_NAME := THE_TYPE_NAME;
    raise UNDEFINED_USER_DEFINED_TYPE;
  end if;


  -- type of the UDT must be a valid Ada type
  if not RECOGNIZED_ADA_TYPE(UDT_TYPE_NAME.NAME) then
    GV_NAME := THE_TYPE_NAME;
    GV_UDT_NAME := UDT_TYPE_NAME;
    raise INVALID_ADA_TYPE;
  end if;

 exception

  when UNDEFINED_USER_DEFINED_TYPE =>
    PUT_LINE("*** ERROR:  " & GV_NAME.NAME.S & " IS A USER DEFINED");
    PUT_LINE("TYPE THAT IS NOT DEFINED IN THE PROTOTYPE PSDL");
    PUT_LINE("SPECIFICATION.  THIS MEANS IT IS EITHER NOT INCLUDED");
    PUT_LINE("AS A SEPARATE TYPE SPECIFICATION, OR IF IT IS, IT IS");
    PUT_LINE("NOT DEFINED AS AN ADA TYPE WITHIN THE TYPE SPECIFICATION");
    PUT_LINE("IT IS ALSO POSSIBLE THAT EITHER 'ARRAY_ELEMENT' OR");
    PUT_LINE("'ARRAY_INDEX' WAS REQUIRED BUT NOT USED AS IDENTIFIERS.");
    NEW_LINE;
    raise ;

  when INVALID_ADA_TYPE =>
    PUT_LINE("*** ERROR:  INVALID ADA TYPE DISCOVERED FOR -");
    PUT_LINE("          " & GV_NAME.NAME.S & " : " & GV_UDT_NAME.NAME.S);
    PUT_LINE("A Type MUST BE DEFINED AS AN ADA TYPE.  IT CAN REFERENCE");
    PUT_LINE("INFORMATION IN ITS GENERIC PARAMETERS, BUT CANNOT");
    PUT_LINE("REFERENCE DEFINITION INFORMATION OUTSIDE ITS OWN");
```

```
   PUT_LINE("SPECIFICATION.  A Generic PARAMETER MUST BE FULLY ");
   PUT_LINE("DEFINED AS AN ADA TYPE.  IT CAN'T EVEN REFERENCE OTHER ");
   PUT_LINE("DEFINITIONS WITHIN ITS GENERICS.");
   NEW_LINE;
   raise ;

  when others =>
   PUT_LINE("*** UNKNOWN ERROR:  OCCURED IN procedure ");
   PUT_LINE("              GET_USER)_DEFINED_TYPE.   ");
   NEW_LINE;
   raise ;


end GET_USER_DEFINED_TYPE;



-- Looks through the generic parameters of a Type or Operator to
-- see if an unrecognized type is defined as an Ada type there.
procedure CHECK_GENERICS(THE_COMPONENT    : in COMPONENT_TYPE;
                                                UNKNOWN_TYPE_NAME          :          in          out
PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
                         GENERIC_TYPE_NAME : in out PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
                         FOUND_TYPE      : out BOOLEAN) is

  COMPONENT    : PSDL_COMPONENT_PKG.PSDL_COMPONENT;
  GENERIC_MODEL : PSDL_CONCRETE_TYPE_PKG.TYPE_DECLARATION;
  COMP_NAME    : PSDL_ID_PKG.PSDL_ID;

 begin

 -- select the correct component type
  case THE_COMPONENT is

    when OPERATOR_COMP =>
         COMPONENT := OPERATOR_COMPONENT;

    when TYPE_COMP =>
         COMPONENT := TYPE_COMPONENT_WITH_ARRAY;

  end case;

  -- Get the generic parameters.
  GENERIC_MODEL := PSDL_COMPONENT_PKG.GENERIC_PARAMETERS(COMPONENT);

  if not PSDL_CONCRETE_TYPE_PKG.EQUAL(GENERIC_MODEL,
   PSDL_CONCRETE_TYPE_PKG.EMPTY_TYPE_DECLARATION) then

  -- get the corresponding Ada type of the corresponding unknown type
    GENERIC_TYPE_NAME :=
     PSDL_CONCRETE_TYPE_PKG.TYPE_DECLARATION_PKG.FETCH(GENERIC_MODEL,
     UNKNOWN_TYPE_NAME.NAME);
```

```
   -- Ensure unknown type was defined.  If not, then nothing was fetched.
   if TYPE_NAME_PKG."=" (GENERIC_TYPE_NAME, TYPE_NAME_PKG.NULL_TYPE) then
   -- will have to check something other then generics
         FOUND_TYPE := FALSE;
   else
   -- type of the GENERIC must be a valid Ada type
         if RECOGNIZED_ADA_TYPE(GENERIC_TYPE_NAME.NAME) then
          FOUND_TYPE := TRUE;
         else
          GV_NAME := UNKNOWN_TYPE_NAME;
          GV_UDT_NAME := GENERIC_TYPE_NAME;
          raise INVALID_ADA_TYPE;
         end if;
   end if;

  else
  -- will have to check something other then generics
   FOUND_TYPE := FALSE;
  end if;

 exception

  when INVALID_ADA_TYPE =>
   PUT_LINE("*** ERROR:  INVALID ADA TYPE DISCOVERED FOR -");
   PUT_LINE("         " & GV_NAME.NAME.S & " : " & GV_UDT_NAME.NAME.S);
   PUT_LINE("A Type MUST BE DEFINED AS AN ADA TYPE.  IT CAN REFERENCE");
   PUT_LINE("INFORMATION IN ITS GENERIC PARAMETERS, BUT CANNOT");
   PUT_LINE("REFERENCE DEFINITION INFORMATION OUTSIDE ITS OWN");
   PUT_LINE("SPECIFICATION.  A Generic PARAMETER MUST BE FULLY ");
   PUT_LINE("DEFINED AS AN ADA TYPE.  IT CAN'T EVEN REFERENCE OTHER ");
   PUT_LINE("DEFINITIONS WITHIN ITS GENERICS.");
   NEW_LINE;
   raise ;

  when others =>
   PUT_LINE("*** UNKNOWN ERROR:  OCCURED IN procedure ");
   PUT_LINE("                CHECK_GENERICS.    ");
   NEW_LINE;
   raise ;

 end CHECK_GENERICS;



 -- Called if a type is an ARRAY which means that in order to
 -- correctly build the signature, not only must the ARRAY
 -- be encoded into the signature, but also the ARRAY components.
 procedure       GET_ARRAY_COMPONENTS(TYPE_NAME                                  :       in
PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
                                      ELEMENT_TYPE   : out PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
                                      ELEMENT_IS_UDT : out BOOLEAN;
                                      ELEMENT_UDT    : out PSDL_ID_PKG.PSDL_ID;
                                      INDEX_TYPE     : out PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
```

```
                                    INDEX_IS_UDT   : out BOOLEAN;
                                    INDEX_UDT      : out PSDL_ID_PKG.PSDL_ID;
                                    ARRAY_IN_UDT   : in BOOLEAN) is

 FOUND_TYPE                                                    :
  BOOLEAN;
 -- User Defined Type (UDT) type name
 UDT_TYPE_NAME, GENERIC_TYPE_NAME, ARRAY_ELEMENT_TYPE_NAME,
  ARRAY_INDEX_TYPE_NAME, THE_TYPE_NAME : PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;


begin

-- Set to the last type component retrieved when looking for a
-- UDT.  Needed if the UDT was an Array type.  This will allow
-- the generic parameters of the Type Component to be searched
-- in case the array index or element is defined there.
 TYPE_COMPONENT_WITH_ARRAY := TYPE_COMPONENT;


 THE_TYPE_NAME := TYPE_NAME;

 -- Because "array" is a composite type, must get
 -- its components (element & index)

 -- Get the array element type.  ***NOTE:  this program expects
 -- the PSDL specification to use the identifier "ARRAY_ELEMENT"
 -- and it IS case sensitive.
 ARRAY_ELEMENT_TYPE_NAME :=
  PSDL_CONCRETE_TYPE_PKG.TYPE_DECLARATION_PKG.FETCH(THE_TYPE_NAME.GEN_PAR,
  ARRAY_ELEMENT_TYPE);
 -- Ensure type was defined.  If not, then nothing was fetched
 -- and that was due to a failure to use the correct identifier
 -- "ARRAY_ELEMENT".
 if TYPE_NAME_PKG."=" (ARRAY_ELEMENT_TYPE_NAME, TYPE_NAME_PKG.NULL_TYPE) then
   GV_NAME := THE_TYPE_NAME;
   raise UNDEFINED_USER_DEFINED_TYPE;
 end if;

 if RECOGNIZED_ADA_TYPE(ARRAY_ELEMENT_TYPE_NAME.NAME) then
 -- The parameter is a valid Ada type so go ahead and make
 -- it known to the procedure output parameter
   ELEMENT_TYPE := ARRAY_ELEMENT_TYPE_NAME;
   ELEMENT_IS_UDT := FALSE;

 else

 -- Maybe the parameter is defined in the generics of either the
 -- Operator component or a Type component (UDT) if that is where the
 -- Array identifier was defined.
```

```
    if ARRAY_IN_UDT then
          CHECK_GENERICS(TYPE_COMP, ARRAY_ELEMENT_TYPE_NAME, GENERIC_TYPE_NAME,
          FOUND_TYPE);
    else
          CHECK_GENERICS(OPERATOR_COMP,                    ARRAY_ELEMENT_TYPE_NAME,
GENERIC_TYPE_NAME,
          FOUND_TYPE);
    end if;

    if FOUND_TYPE then
    -- The type is either not composite or even if composite,
    -- will not be further specified by its components so
    -- go ahead and store it in element_type.
          ELEMENT_TYPE := GENERIC_TYPE_NAME;

    else   -- The parameter must be a user defined type so
          -- get its type.
          GET_USER_DEFINED_TYPE(ARRAY_ELEMENT_TYPE_NAME, UDT_TYPE_NAME);

     -- The type is either not composite or even if composite,
     -- will not be further specified by its components so
     -- go ahead and store it in element_type.
          ELEMENT_TYPE := UDT_TYPE_NAME;
          ELEMENT_IS_UDT := TRUE;
          ELEMENT_UDT := ARRAY_ELEMENT_TYPE_NAME.NAME;

    end if;
  end if;


  -- Get the array index type.  ***NOTE:  this program expects
  -- the PSDL specification to use the identifier "ARRAY_INDEX"
  --  and it IS case sensitive.
  ARRAY_INDEX_TYPE_NAME :=
   PSDL_CONCRETE_TYPE_PKG.TYPE_DECLARATION_PKG.FETCH(THE_TYPE_NAME.GEN_PAR,
   ARRAY_INDEX_TYPE);


  -- Ensure type was defined.  If not, then nothing was fetched
  -- and that was due to a failure to use the correct identifier
  -- "ARRAY_INDEX".
  if TYPE_NAME_PKG."=" (ARRAY_INDEX_TYPE_NAME, TYPE_NAME_PKG.NULL_TYPE) then
    GV_NAME := THE_TYPE_NAME;
    raise UNDEFINED_USER_DEFINED_TYPE;
  end if;


  if RECOGNIZED_ADA_TYPE(ARRAY_INDEX_TYPE_NAME.NAME) then
  -- The parameter is a valid Ada type so go ahead and store
  -- the array index into index_type.
    INDEX_TYPE := ARRAY_INDEX_TYPE_NAME;
```

```
    INDEX_IS_UDT := FALSE;

   else

   -- Maybe the parameter is defined in the generics of either the
   -- Operator component or a Type component (UDT) if that is where the
   -- Array identifier was defined.
    if ARRAY_IN_UDT then
          CHECK_GENERICS(TYPE_COMP, ARRAY_INDEX_TYPE_NAME, GENERIC_TYPE_NAME,
           FOUND_TYPE);
    else
          CHECK_GENERICS(OPERATOR_COMP,                    ARRAY_INDEX_TYPE_NAME,
GENERIC_TYPE_NAME,
           FOUND_TYPE);
    end if;

    if FOUND_TYPE then
    -- The type is either not composite or even if composite,
    -- will not be further specified by its components so
    -- go ahead and store.
          INDEX_TYPE := GENERIC_TYPE_NAME;

    else   -- The parameter must be a user defined type so
          -- get its type.

          GET_USER_DEFINED_TYPE(ARRAY_INDEX_TYPE_NAME, UDT_TYPE_NAME);

      -- The type is either not composite or even if composite,
      -- will not be further specified by its components so
      -- go ahead and store.
          INDEX_TYPE := UDT_TYPE_NAME;
          INDEX_IS_UDT := TRUE;
          INDEX_UDT := ARRAY_INDEX_TYPE_NAME.NAME;

    end if;
   end if;

 exception

   when UNDEFINED_USER_DEFINED_TYPE =>
    PUT_LINE("*** ERROR:  " & GV_NAME.NAME.S & " IS A USER DEFINED");
    PUT_LINE("TYPE THAT IS NOT DEFINED IN THE PROTOTYPE PSDL");
    PUT_LINE("SPECIFICATION.  THIS MEANS IT IS EITHER NOT INCLUDED");
    PUT_LINE("AS A SEPARATE TYPE SPECIFICATION, OR IF IT IS, IT IS");
    PUT_LINE("NOT DEFINED AS AN ADA TYPE WITHIN THE TYPE SPECIFICATION");
    PUT_LINE("IT IS ALSO POSSIBLE THAT EITHER 'ARRAY_ELEMENT' OR");
    PUT_LINE("'ARRAY_INDEX' WAS REQUIRED BUT NOT USED AS IDENTIFIERS.");
    NEW_LINE;
    raise ;

   when others =>
    PUT_LINE("*** UNKNOWN ERROR:  OCCURED IN procedure ");
    PUT_LINE("                ENCODE_ARRAY_AND_ITS_COMPONENTS.   ");
```

```
    NEW_LINE;
    raise ;

 end GET_ARRAY_COMPONENTS;


end DETERMINE_THE_ADA_TYPE_PKG;
```

****
****
****

```
-- Filename    / Encode_ADT_Signature.a
-- Date        / 30 August 93
-- Author      / Scott Dolgoff
-- System      / Sun SPARCstation
-- Compiler    / Verdix Ada
-- Description / This program encodes the signatures
--             / of software components from their Prototype System
--             / Description Language (PSDL) specifications.


with PSDL_COMPONENT_PKG, ADD_ADA_TYPE_TO_SIGNATURE_PKG;
use  PSDL_COMPONENT_PKG, ADD_ADA_TYPE_TO_SIGNATURE_PKG;

package ENCODE_ADT_SIGNATURE_PKG is

  procedure ENCODE_ADT_SIGNATURE(INPUT_SIGNATURE  : out SIGNATURE;
                                 OUTPUT_SIGNATURE : out SIGNATURE);

end ENCODE_ADT_SIGNATURE_PKG;



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


with TEXT_IO, PSDL_CONCRETE_TYPE_PKG, LOAD_PSDL_INTO_ADA_STRUCTURE_PKG,
 ITERATE_THROUGH_ADT_OPERATOR_PARAMETERS_PKG, PSDL_ID_PKG;
use  TEXT_IO, PSDL_CONCRETE_TYPE_PKG, LOAD_PSDL_INTO_ADA_STRUCTURE_PKG,
 ITERATE_THROUGH_ADT_OPERATOR_PARAMETERS_PKG, PSDL_ID_PKG;


package body ENCODE_ADT_SIGNATURE_PKG is

  INPUT_SWITCH                    :
   constant IO_SWITCH_CLASSES := INPUT_PARAMETER;
  OUTPUT_SWITCH                   :
```

```
 constant IO_SWITCH_CLASSES := OUTPUT_PARAMETER;
TYPE_COMPONENT                          :
 PSDL_COMPONENT_PKG.DATA_TYPE;
OPERATOR_COUNT                          :
 INTEGER            := 0;
-- WORKING_XXX_SIGNATURE is initialized to 0.  It
-- enables an aggregate signature (aggregate of all ADT Operator
-- signatures) to be built.
WORKING_INPUT_SIGNATURE, WORKING_OUTPUT_SIGNATURE  : SIGNATURE;
OPERATOR_INPUT_SIGNATURE, OPERATOR_OUTPUT_SIGNATURE : SIGNATURE;



-- procedure passed to generic scan procedure in generic map package
-- extracts the ADT Operators from the Type component
-- map
procedure GET_ADT_OPERATORS(ID             : in PSDL_ID_PKG.PSDL_ID;
                            OPERATOR_COMPONENT : PSDL_COMPONENT_PKG.OPERATOR);


-- iterate through the Operation map to extract the set of output
-- parameters.
 procedure                ITERATE_THROUGH_ADT_OPERATORS                  is                  new
PSDL_COMPONENT_PKG.OPERATION_MAP_PKG.GENERIC_SCAN(GENERATE=>GET_ADT_OPERATO
RS);



procedure STORE_ADT_OPERATOR_SIGNATURES_IN_FILE(OP_NAME : in STRING;
                                                TP_NAME : in STRING) is

  package INTEGER_INOUT is new INTEGER_IO(INTEGER);
  use  INTEGER_INOUT;

  -- ADT operator names are the catenation of the ADT component
  -- name with a "." and the ADT operator name.
  SIG_FILE_NAME : constant STRING := TP_NAME & "." & OP_NAME & ".txt";
  SIG_FILE : TEXT_IO.FILE_TYPE;


begin


-- Create the file using operator name to store the
-- Operator signatures in.
 TEXT_IO.CREATE(SIG_FILE, MODE => OUT_FILE, NAME => SIG_FILE_NAME);

  -- Store Operator input signature.
 for REGION in ALL_REGIONS'FIRST .. ALL_REGIONS'LAST loop

  -- Note:  Setting WIDTH to zero is critical because
```

```
  --      it left justifies the numbers.  This makes
  --      it easy on the C++ program that has to read
  --      the numbers from a file as characters and
  --      then convert them to integers.
   PUT(SIG_FILE, OPERATOR_INPUT_SIGNATURE(REGION), WIDTH => 0);
   NEW_LINE(SIG_FILE);

  end loop;


  -- Store Operator output signature.
  for REGION in ALL_REGIONS'FIRST .. ALL_REGIONS'LAST loop

   PUT(SIG_FILE, OPERATOR_OUTPUT_SIGNATURE(REGION), WIDTH => 0);
   NEW_LINE(SIG_FILE);

  end loop;

  TEXT_IO.CLOSE(SIG_FILE);

end STORE_ADT_OPERATOR_SIGNATURES_IN_FILE;



procedure UPDATE_WORKING_SIGNATURE(THE_SIGNATURE : in SIGNATURE;
                                   IO_SWITCH     : in IO_SWITCH_CLASSES) is


begin
 for REGION in ALL_REGIONS'FIRST .. ALL_REGIONS'LAST loop


   case IO_SWITCH is

        when INPUT_PARAMETER =>

         WORKING_INPUT_SIGNATURE(REGION) := WORKING_INPUT_SIGNATURE(REGION) +
         THE_SIGNATURE(REGION);

        when OUTPUT_PARAMETER =>

         WORKING_OUTPUT_SIGNATURE(REGION) := WORKING_OUTPUT_SIGNATURE(REGION) +
         THE_SIGNATURE(REGION);

   end case;

  end loop;

end UPDATE_WORKING_SIGNATURE;


-- procedure passed to generic scan procedure in generic map package
```

```
-- extracts the ADT Operators from the Type component
-- map
procedure GET_ADT_OPERATORS(ID              : in PSDL_ID_PKG.PSDL_ID;
                            OPERATOR_COMPONENT : PSDL_COMPONENT_PKG.OPERATOR) is
 COMPONENT_ID                : PSDL_ID_PKG.PSDL_ID;
 OUTPUT_PARAMETERS, INPUT_PARAMETERS :
  PSDL_CONCRETE_TYPE_PKG.TYPE_DECLARATION;


begin

-- provide necessary component to
-- ITERATE_THROUGH_ADT_OPERATOR_PARAMETERS_PKG
 PASS_ADT_OPERATOR(OPERATOR_COMPONENT);

 OPERATOR_COUNT := OPERATOR_COUNT + 1;

 -- get the Operator's input parameters
 INPUT_PARAMETERS := PSDL_COMPONENT_PKG.INPUTS(OPERATOR_COMPONENT);

 -- initialize signatures to empty
 INITIALIZE_THE_SIGNATURES(INPUT_SWITCH);

 -- iterate through the input parameters to get the
 -- input parameter signature for this Operator
 ITERATE_THROUGH_INPUT_PARAMETERS(INPUT_PARAMETERS);
 -- retrieve signature value from
ITERATE_THROUGH_OPERATOR_PARAMETERS_PKG
 GET_INPUT_SIGNATURE(OPERATOR_INPUT_SIGNATURE);


 -- get the Operator's output parameters
 OUTPUT_PARAMETERS := PSDL_COMPONENT_PKG.OUTPUTS(OPERATOR_COMPONENT);

 -- initialize signatures to empty
 INITIALIZE_THE_SIGNATURES(OUTPUT_SWITCH);

 -- iterate through the output parameters to get the
 -- output parameter signature for this Operator
 ITERATE_THROUGH_OUTPUT_PARAMETERS(OUTPUT_PARAMETERS);
 -- get signature value from
ITERATE_THROUGH_OPERATOR_PARAMETERS_PKG
 GET_OUTPUT_SIGNATURE(OPERATOR_OUTPUT_SIGNATURE);

 -- update the working signatures to aggregate the new type
 -- information
 UPDATE_WORKING_SIGNATURE(OPERATOR_INPUT_SIGNATURE, INPUT_SWITCH);
 UPDATE_WORKING_SIGNATURE(OPERATOR_OUTPUT_SIGNATURE, OUTPUT_SWITCH);
```

```
 -- Store the ADT_OPERATOR signature in a text file. We need
 -- to store each set of input and output signatures in two
 -- different files. One is used for loading an ADT into the
 -- CAPS software base. The other is information on the query
 -- component that is used during matching.
 COMPONENT_ID := PSDL_COMPONENT_PKG.NAME(TYPE_COMPONENT);
 STORE_ADT_OPERATOR_SIGNATURES_IN_FILE(ID.S, COMPONENT_ID.S);

end GET_ADT_OPERATORS;




procedure ENCODE_ADT_SIGNATURE(INPUT_SIGNATURE  : out SIGNATURE;
                               OUTPUT_SIGNATURE : out SIGNATURE) is

 ADT_OPERATORS : PSDL_COMPONENT_PKG.OPERATION_MAP;


begin

-- get the Type component
 RETRIEVE_TYPE_COMPONENT(TYPE_COMPONENT);

 -- initialize the signatures to 0
 for REGION in ALL_REGIONS'FIRST .. ALL_REGIONS'LAST loop

  INPUT_SIGNATURE(REGION) := 0;
  OUTPUT_SIGNATURE(REGION) := 0;
  WORKING_INPUT_SIGNATURE(REGION) := 0;
  WORKING_OUTPUT_SIGNATURE(REGION) := 0;

 end loop;

 -- retrieve the mapping of ADT Operators
 ADT_OPERATORS := PSDL_COMPONENT_PKG.OPERATIONS(TYPE_COMPONENT);

 -- check to see if the Type component has Operators
 if not (OPERATION_MAP_PKG.EQUAL(ADT_OPERATORS,
  PSDL_COMPONENT_PKG.EMPTY_OPERATION_MAP)) then

  ITERATE_THROUGH_ADT_OPERATORS(ADT_OPERATORS);

 end if;

 INPUT_SIGNATURE := WORKING_INPUT_SIGNATURE;
 OUTPUT_SIGNATURE := WORKING_OUTPUT_SIGNATURE;

exception

 when UNDEFINED_USER_DEFINED_TYPE =>
```

```
    PUT_LINE("Corrections may be needed to the Prototype PSDL spec.");
    PUT_LINE("*** THIS PROGRAM CANNOT CONTINUE ***");
    raise ;

   when INVALID_ADA_TYPE =>
    PUT_LINE("Corrections may be needed to the Prototype PSDL spec.");
    PUT_LINE("User defined types must be given a corresponding Ada");
    PUT_LINE("type in their PSDL specification.");
    PUT_LINE("*** THIS PROGRAM CANNOT CONTINUE ***");
    raise ;

   when PROTOTYPE_FILE_NOT_FOUND =>
    PUT_LINE("*** THIS PROGRAM CANNOT CONTINUE ***");
    raise ;

   when OPERATOR_FILE_NOT_FOUND =>
    PUT_LINE("*** THIS PROGRAM CANNOT CONTINUE ***");
    raise ;

   when others =>
    PUT_LINE("*** THIS PROGRAM CANNOT CONTINUE ***");
    raise ;

 end ENCODE_ADT_SIGNATURE;

end ENCODE_ADT_SIGNATURE_PKG;
```

****
****
****

```
-- Filename    | Encode_Operator_Signature.a
-- Date        | 29 August 93
-- Author      | Scott Dolgoff
-- System      | Sun SPARCstation
-- Compiler    | Verdix Ada
-- Description | This program encodes the signatures
--             | of software components from their Prototype System
--             | Description Language (PSDL) specifications.


with PSDL_COMPONENT_PKG, ADD_ADA_TYPE_TO_SIGNATURE_PKG;
use  PSDL_COMPONENT_PKG, ADD_ADA_TYPE_TO_SIGNATURE_PKG;

package ENCODE_OPERATOR_SIGNATURE_PKG is

 procedure ENCODE_OPERATOR_SIGNATURE(INPUT_SIGNATURE  : out SIGNATURE;
                                     OUTPUT_SIGNATURE : out SIGNATURE);

end ENCODE_OPERATOR_SIGNATURE_PKG;
```

```
-------------------------------------------------------------------
-------------------------------------------------------------------


with TEXT_IO, PSDL_CONCRETE_TYPE_PKG, LOAD_PSDL_INTO_ADA_STRUCTURE_PKG,
 ITERATE_THROUGH_OPERATOR_PARAMETERS_PKG, PSDL_PROGRAM_PKG;
use  TEXT_IO, PSDL_CONCRETE_TYPE_PKG, LOAD_PSDL_INTO_ADA_STRUCTURE_PKG,
 ITERATE_THROUGH_OPERATOR_PARAMETERS_PKG, PSDL_PROGRAM_PKG;


package body ENCODE_OPERATOR_SIGNATURE_PKG is

 INPUT_SWITCH      : constant IO_SWITCH_CLASSES := INPUT_PARAMETER;
 OUTPUT_SWITCH     : constant IO_SWITCH_CLASSES := OUTPUT_PARAMETER;
 PROTOTYPE_SPEC    : PSDL_PROGRAM_PKG.PSDL_PROGRAM;
 OPERATOR_COMPONENT : PSDL_COMPONENT_PKG.OPERATOR;


 procedure ENCODE_OPERATOR_SIGNATURE(INPUT_SIGNATURE  : out SIGNATURE;
                                     OUTPUT_SIGNATURE : out SIGNATURE) is


  OUTPUT_PARAMETERS, INPUT_PARAMETERS :
   PSDL_CONCRETE_TYPE_PKG.TYPE_DECLARATION;


 begin

 -- get Prototype specification
  RETRIEVE_PROTOTYPE_SPEC(PROTOTYPE_SPEC);

  -- get the Operator component
  RETRIEVE_OPERATOR_COMPONENT(OPERATOR_COMPONENT);

  -- Pass the components to the packages that will need
  -- them
  SEND_COMPONENTS(OPERATOR_COMPONENT, PROTOTYPE_SPEC);


  -- get the Operator's input parameters
  INPUT_PARAMETERS := PSDL_COMPONENT_PKG.INPUTS(OPERATOR_COMPONENT);

  -- initialize signatures to empty
  INITIALIZE_THE_SIGNATURES(INPUT_SWITCH);

  -- iterate through the input parameters to get the
  -- input parameter signature for this Operator
  ITERATE_THROUGH_INPUT_PARAMETERS(INPUT_PARAMETERS);
```

```
   -- retrieve signature value from
ITERATE_THROUGH_OPERATOR_PARAMETERS_PKG
   GET_INPUT_SIGNATURE(INPUT_SIGNATURE);


   -- get the Operator's output parameters
   OUTPUT_PARAMETERS := PSDL_COMPONENT_PKG.OUTPUTS(OPERATOR_COMPONENT);

   -- initialize signatures to empty
   INITIALIZE_THE_SIGNATURES(OUTPUT_SWITCH);

   -- iterate through the output parameters to get the
   -- output parameter signature for this Operator
   ITERATE_THROUGH_OUTPUT_PARAMETERS(OUTPUT_PARAMETERS);
   -- retrieve signature value from
ITERATE_THROUGH_OPERATOR_PARAMETERS_PKG
   GET_OUTPUT_SIGNATURE(OUTPUT_SIGNATURE);


  exception

   when PROTOTYPE_FILE_NOT_FOUND =>
    PUT_LINE("*** THIS PROGRAM CANNOT CONTINUE ***");
    raise ;

   when OPERATOR_FILE_NOT_FOUND =>
    PUT_LINE("*** THIS PROGRAM CANNOT CONTINUE ***");
    raise ;

   when others =>
    PUT_LINE("*** THIS PROGRAM CANNOT CONTINUE ***");
    raise ;

  end ENCODE_OPERATOR_SIGNATURE;

end ENCODE_OPERATOR_SIGNATURE_PKG;
```

****
****
****

-- *Filename* / **Load_PSDL_Into_Ada_Structure_Pkg.a**
-- *Date* / *29 August 93*
-- *Author* / *Scott Dolgoff*
-- *System* / *Sun SPARCstation*
-- *Compiler* / *Verdix Ada*
-- *Description* / *This program encodes provides the necessary routines*

```
--          | to load a PSDL specification into an Ada data structure.
--          | An Operator, Type, or a full blown Prototype PSDL
--          | specification can all be loaded.  Normally the
--          | Prototype Specification can then be searched to
--          | extract specific Types by name.


with PSDL_COMPONENT_PKG, PSDL_PROGRAM_PKG;
use  PSDL_COMPONENT_PKG, PSDL_PROGRAM_PKG;

package LOAD_PSDL_INTO_ADA_STRUCTURE_PKG is


-- Provides access to the stored Operator component.
 procedure              RETRIEVE_OPERATOR_COMPONENT(OP_COMPONENT              :              out
PSDL_COMPONENT_PKG.OPERATOR);


 -- Provides access to the stored Type component.
 procedure              RETRIEVE_TYPE_COMPONENT(TP_COMPONENT              :              out
PSDL_COMPONENT_PKG.DATA_TYPE);

 -- Provides access to the stored prototype specification.
 procedure              RETRIEVE_PROTOTYPE_SPEC(PROTOTYPE_SPEC              :              out
PSDL_PROGRAM_PKG.PSDL_PROGRAM);


 -- exceptions
 --  PSDL specification (text file) for the Operator is not in
 --  the current directory
 OPERATOR_FILE_NOT_FOUND  : exception;
 --  PSDL specification (text file) for the Type is not in
 --  the current directory
 TYPE_FILE_NOT_FOUND     : exception;
 --  PSDL specification (text file) for the prototype is not in
 --  the current directory
 PROTOTYPE_FILE_NOT_FOUND : exception;

end LOAD_PSDL_INTO_ADA_STRUCTURE_PKG;



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



with TEXT_IO, PSDL_IO, PSDL_ID_PKG;
use  TEXT_IO, PSDL_IO, PSDL_ID_PKG;
```

```
package body LOAD_PSDL_INTO_ADA_STRUCTURE_PKG is

-- procedure passed to generic scan procedure in generic map package
-- extracts the Operator from the PSDL specification (psdl_program)
 procedure GET_OPERATOR(ID       : in PSDL_ID_PKG.PSDL_ID;
                        COMPONENT : in PSDL_COMPONENT_PKG.PSDL_COMPONENT);

 -- iterate through the PSDL specification to extract the Operator.  Even
 -- though this PSDL specification only has one Operator (and no Types), it
 -- is initially stored as a psdl_program and we need it in the form of
 -- a psdl_component to later extract the input and output parameters.
 procedure ITERATE_THROUGH_OPERATOR_PSDL_SPECIFICATION is new
PSDL_PROGRAM_PKG.PSDL_PROGRAM_MAP_PKG.GENERIC_SCAN(GENERATE=>GET_OPERATOR);


 -- procedure passed to generic scan procedure in generic map package
 -- extracts the Type from the PSDL specification (psdl_program)
 procedure GET_TYPE(ID       : in PSDL_ID_PKG.PSDL_ID;
                    COMPONENT : in PSDL_COMPONENT_PKG.PSDL_COMPONENT);

 -- iterate through the PSDL specification to extract the Type.  Even
 -- though this PSDL specification is comprised of only one Type, it
 -- is initially stored as a psdl_program and we need it in the form of
 -- a psdl_component to later extract the input and output parameters of
 -- the Type's operators.
 procedure ITERATE_THROUGH_TYPE_PSDL_SPECIFICATION is new
PSDL_PROGRAM_PKG.PSDL_PROGRAM_MAP_PKG.GENERIC_SCAN(GENERATE=>GET_TYPE);


 PROTOTYPE_SPECIFICATION, OPERATOR_SPEC, TYPE_SPEC :
  PSDL_PROGRAM_PKG.PSDL_PROGRAM;
 OPERATOR_COMPONENT                 :
  PSDL_COMPONENT_PKG.OPERATOR;
 TYPE_COMPONENT                     :
  PSDL_COMPONENT_PKG.DATA_TYPE;
 TYPE_ID, OPERATOR_ID               : PSDL_ID_PKG.PSDL_ID;


 -- procedure passed to generic scan procedure in generic map package
 -- extracts the Operator from the PSDL specification (psdl_program)
 procedure GET_OPERATOR(ID       : in PSDL_ID_PKG.PSDL_ID;
                        COMPONENT : in PSDL_COMPONENT_PKG.PSDL_COMPONENT) is

 begin
  OPERATOR_ID := ID;
```

```
  OPERATOR_COMPONENT := COMPONENT;

 end GET_OPERATOR;


 -- Loads OPERATOR_SPEC data structure with the PSDL specification
 -- of the operator and then stores the operator in an OPERATOR_COMPONENT
 -- data structure.
 procedure GET_OPERATOR_COMPONENT is

  OPERATOR_FILE_NAME : constant STRING := "operator_psdl_spec.txt";
  OPERATOR_FILE     : TEXT_IO.FILE_TYPE;

 begin

 -- open the file that contains the operator PSDL specification
  TEXT_IO.OPEN(OPERATOR_FILE, MODE => IN_FILE, NAME => OPERATOR_FILE_NAME);

  -- parse the operator PSDL specification and insert it into
  -- the Ada data structure OPERATOR_SPEC
  PSDL_IO.GET(OPERATOR_FILE, OPERATOR_SPEC);

  TEXT_IO.CLOSE(OPERATOR_FILE);

  -- iterate through the PSDL specification to extract the Operator.
  -- Though this PSDL specification only has one Operator (and no Types),
  -- it's initially stored as a psdl_program and we need it in the form of
  -- a psdl_component to later extract the input and output parameters.
  -- The global variable OPERATOR_COMPONENT receives the data.
  ITERATE_THROUGH_OPERATOR_PSDL_SPECIFICATION(OPERATOR_SPEC);

 exception
  when NAME_ERROR =>
   raise OPERATOR_FILE_NOT_FOUND;

  when others =>
   PUT_LINE("*** UNKNOWN ERROR:  Occurred in procedure ");
   PUT_LINE("Get_Operator_Component.");
   raise ;

 end GET_OPERATOR_COMPONENT;



 -- Provides access to the stored operator component.
 procedure          RETRIEVE_OPERATOR_COMPONENT(OP_COMPONENT          :          out
PSDL_COMPONENT_PKG.OPERATOR) is

 begin

 -- Loads OPERATOR_SPEC data structure with the PSDL specification
 -- of the operator and then stores the operator in an OPERATOR_COMPONENT
```

```
-- data structure.
 GET_OPERATOR_COMPONENT;


 OP_COMPONENT := OPERATOR_COMPONENT;


exception
 when OPERATOR_FILE_NOT_FOUND =>
  PUT_LINE("The file 'operator_psdl_spec.txt' is missing");
  PUT_LINE("from the directory. It should contain the PSDL");
  PUT_LINE("specification for the operator whose signature");
  PUT_LINE("we are calculating.");
  raise ;

end RETRIEVE_OPERATOR_COMPONENT;




-- procedure passed to generic scan procedure in generic map package
-- extracts the Type from the PSDL specification (psdl_program)
procedure GET_TYPE(ID        : in PSDL_ID_PKG.PSDL_ID;
                   COMPONENT : in PSDL_COMPONENT_PKG.PSDL_COMPONENT) is

begin
 TYPE_ID := ID;
 TYPE_COMPONENT := COMPONENT;

end GET_TYPE;


-- Loads TYPE_SPEC data structure with the PSDL specification
-- of the Type and then stores the Type in a TYPE_COMPONENT
-- data structure.
procedure GET_TYPE_COMPONENT is

 TYPE_FILE_NAME : constant STRING := "type_psdl_spec.txt";
 TYPE_FILE      : TEXT_IO.FILE_TYPE;

begin

-- open the file that contains the Type PSDL specification
 TEXT_IO.OPEN(TYPE_FILE, MODE => IN_FILE, NAME => TYPE_FILE_NAME);

 -- parse the Type PSDL specification and insert it into
 -- the Ada data structure TYPE_SPEC
 PSDL_IO.GET(TYPE_FILE, TYPE_SPEC);

 TEXT_IO.CLOSE(TYPE_FILE);
```

```
  -- iterate through the PSDL specification to extract the Type.
  -- Though this PSDL specification only has one Type,
  -- it's initially stored as a psdl_program and we need it in the form of
  -- a psdl_component to later extract the input and output parameters
  -- of the Type's operators.
  -- The global variable TYPE_COMPONENT receives the data.
  ITERATE_THROUGH_TYPE_PSDL_SPECIFICATION(TYPE_SPEC);

 exception
  when NAME_ERROR =>
   raise TYPE_FILE_NOT_FOUND;

  when others =>
   PUT_LINE("*** UNKNOWN ERROR:  Occurred in procedure ");
   PUT_LINE("Get_Type_Component.");
   raise ;

 end GET_TYPE_COMPONENT;




 -- Provides access to the stored Type component.
 procedure                RETRIEVE_TYPE_COMPONENT(TP_COMPONENT                :                out
PSDL_COMPONENT_PKG.DATA_TYPE) is

 begin

 -- Loads TYPE_SPEC data structure with the PSDL specification
 -- of the Type and then stores the Type in a TYPE_COMPONENT
 -- data structure.
  GET_TYPE_COMPONENT;

  TP_COMPONENT := TYPE_COMPONENT;


 exception
  when TYPE_FILE_NOT_FOUND =>
   PUT_LINE("The file 'type_psdl_spec.txt' is missing");
   PUT_LINE("from the directory. It should contain the PSDL");
   PUT_LINE("specification for the Type whose signature");
   PUT_LINE("we are calculating.");
   raise ;

 end RETRIEVE_TYPE_COMPONENT;




 -- Loads PROTOTYPE_SPEC data structure with the PSDL specification
 -- of the prototype.
```

```
procedure GET_PROTOTYPE_SPEC is

  PROTOTYPE_FILE_NAME : constant STRING := "prototype_psdl_spec.txt";
  PROTOTYPE_FILE     : TEXT_IO.FILE_TYPE;

begin

-- open the file that contains the prototype PSDL specification
  TEXT_IO.OPEN(PROTOTYPE_FILE, MODE => IN_FILE, NAME => PROTOTYPE_FILE_NAME);


  -- parse the prototype PSDL specification and insert it into
  -- the Ada data structure PROTOTYPE_SPEC
  PSDL_IO.GET(PROTOTYPE_FILE, PROTOTYPE_SPECIFICATION);

  TEXT_IO.CLOSE(PROTOTYPE_FILE);

exception
  when NAME_ERROR =>
    raise PROTOTYPE_FILE_NOT_FOUND;

  when others =>
    PUT_LINE("*** UNKNOWN ERROR:  Occurred in procedure ");
    PUT_LINE("Get_Prototype_Spec.");
    raise ;

end GET_PROTOTYPE_SPEC;



-- Provides access to the stored prototype specification.
procedure RETRIEVE_PROTOTYPE_SPEC(PROTOTYPE_SPEC : out
PSDL_PROGRAM_PKG.PSDL_PROGRAM) is

begin

-- loads PROTOTYPE_SPEC data structure with the PSDL specification
-- of the prototype
  GET_PROTOTYPE_SPEC;


  PROTOTYPE_SPEC := PROTOTYPE_SPECIFICATION;

exception
  when PROTOTYPE_FILE_NOT_FOUND =>
    PUT_LINE("The file 'prototype_psdl_spec.txt' is missing");
    PUT_LINE("from the directory. It should contain the PSDL");
    PUT_LINE("specification for the prototype system that");
    PUT_LINE("the operator for which we are calculating a ");
    PUT_LINE("signature belongs.");
    raise ;


end RETRIEVE_PROTOTYPE_SPEC;
```

```
end LOAD_PSDL_INTO_ADA_STRUCTURE_PKG;


*******************************************************************************
*******************************************************************************
*******************************************************************************


-- Filename    / Parameter_Iterator_Pkg.a
-- Date        / 29 August 93
-- Author      / Scott Dolgoff
-- System      / Sun SPARCstation
-- Compiler    / Verdix Ada
-- Description / This program encodes the signatures
--             / of software components from their Prototype System
--             / Description Language (PSDL) specifications.


with PSDL_ID_PKG, PSDL_CONCRETE_TYPE_PKG, ADD_ADA_TYPE_TO_SIGNATURE_PKG,
 PSDL_COMPONENT_PKG, PSDL_PROGRAM_PKG;
use  PSDL_ID_PKG, PSDL_CONCRETE_TYPE_PKG, ADD_ADA_TYPE_TO_SIGNATURE_PKG,
 PSDL_COMPONENT_PKG, PSDL_PROGRAM_PKG;

package ITERATE_THROUGH_OPERATOR_PARAMETERS_PKG is


-- procedure passed to generic scan procedure in generic map package
-- extracts the input parameters from the Type Declaration
-- map
 procedure GET_IN_PARAMETER(ID      : in PSDL_ID_PKG.PSDL_ID;
                            TYPE_NAME : in PSDL_CONCRETE_TYPE_PKG.TYPE_NAME);

 -- iterate through the Type Declaration map to extract the set of input
 -- parameters.
 procedure                ITERATE_THROUGH_INPUT_PARAMETERS                is                new
PSDL_CONCRETE_TYPE_PKG.TYPE_DECLARATION_PKG.GENERIC_SCAN(GENERATE=>GET_IN_PAR
AMETER);



 -- procedure passed to generic scan procedure in generic map package
 -- extracts the output parameters from the Type Declaration
 -- map
 procedure GET_OUT_PARAMETER(ID      : in PSDL_ID_PKG.PSDL_ID;
                             TYPE_NAME : in PSDL_CONCRETE_TYPE_PKG.TYPE_NAME);
```

```
 -- iterate through the Type Declaration map to extract the set of output
 -- parameters.
 procedure ITERATE_THROUGH_OUTPUT_PARAMETERS is new
PSDL_CONCRETE_TYPE_PKG.TYPE_DECLARATION_PKG.GENERIC_SCAN(GENERATE=>GET_OUT_PA
RAMETER);


 -- Encoded input and output signatures are returned by these
 -- procedures. NOTE: the procedure ITERATE_THROUGH_xxx_PARAMETERS
 -- must be invoked first which encodes the signature value.
 procedure GET_INPUT_SIGNATURE(IN_SIGNATURE : out SIGNATURE);
 procedure GET_OUTPUT_SIGNATURE(OUT_SIGNATURE : out SIGNATURE);


 -- *** This MUST be called prior to encoding an Operator or Type
 -- Signature. Sets initial signatures to all 0's. IO_SWITCH
 -- tells the procedure whether you need to initialize the input
 -- or output signature.
 procedure INITIALIZE_THE_SIGNATURES(IO_SWITCH : IO_SWITCH_CLASSES);

 -- gets the necessary component values needed for looking up
 -- Ada type information
 procedure SEND_COMPONENTS(MAIN_COMPONENT : in
PSDL_COMPONENT_PKG.PSDL_COMPONENT;
                           PROTO_SPEC : in PSDL_PROGRAM_PKG.PSDL_PROGRAM);


end ITERATE_THROUGH_OPERATOR_PARAMETERS_PKG;



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



with TEXT_IO, ADA_RECOGNIZED_TYPES_PKG, DETERMINE_THE_ADA_TYPE_PKG,
 TYPE_NAME_PKG, A_STRINGS;
use TEXT_IO, ADA_RECOGNIZED_TYPES_PKG, DETERMINE_THE_ADA_TYPE_PKG, A_STRINGS;

package body ITERATE_THROUGH_OPERATOR_PARAMETERS_PKG is

 INPUT_SIGNATURE, OUTPUT_SIGNATURE : SIGNATURE;
 GV_NAME, GV_UDT_NAME : PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;


 -- gets the necessary component values needed for looking up
 -- Ada type information
 procedure SEND_COMPONENTS(MAIN_COMPONENT : in
PSDL_COMPONENT_PKG.PSDL_COMPONENT;
```

```
                    PROTO_SPEC    : in PSDL_PROGRAM_PKG.PSDL_PROGRAM) is

begin
 PASS_COMPONENTS(MAIN_COMPONENT, PROTO_SPEC);

end SEND_COMPONENTS;



-- Encoded input and closeness signatures are returned by this
-- procedure. NOTE:  the procedure ITERATE_THROUGH_INPUT_PARAMETERS
-- must be invoked first which encodes the signature.
procedure GET_INPUT_SIGNATURE(IN_SIGNATURE : out SIGNATURE) is

begin

 IN_SIGNATURE := INPUT_SIGNATURE;

end GET_INPUT_SIGNATURE;



-- Encoded output signature is returned by this
-- procedure.  NOTE:  the procedure
ITERATE_THROUGH_OUTPUT_PARAMETERS
-- must be invoked first which encodes the signature.
procedure GET_OUTPUT_SIGNATURE(OUT_SIGNATURE : out SIGNATURE) is

begin

 OUT_SIGNATURE := OUTPUT_SIGNATURE;

end GET_OUTPUT_SIGNATURE;



-- initializes or resets signatures so new ones can be computed
procedure INITIALIZE_THE_SIGNATURES(IO_SWITCH : IO_SWITCH_CLASSES) is

begin

 case IO_SWITCH is

  when INPUT_PARAMETER =>
       INITIALIZE_SIGNATURES(INPUT_SIGNATURE);

  when OUTPUT_PARAMETER =>
       INITIALIZE_SIGNATURES(OUTPUT_SIGNATURE);

 end case;

end INITIALIZE_THE_SIGNATURES;
```

```
procedure ENCODE_ADA_TYPE_INTO_SIGNATURE(TYPE_NAME_ID : in PSDL_ID_PKG.PSDL_ID;
                                         IO_SWITCH    : in IO_SWITCH_CLASSES;
                                         GENERIC_TYPE : in BOOLEAN) is

begin
 case IO_SWITCH is

  when INPUT_PARAMETER =>
       ADD_ADA_TYPE_TO_SIGNATURE(TYPE_NAME_ID, INPUT_SIGNATURE, IO_SWITCH,
       GENERIC_TYPE);


  when OUTPUT_PARAMETER =>
       ADD_ADA_TYPE_TO_SIGNATURE(TYPE_NAME_ID, OUTPUT_SIGNATURE, IO_SWITCH,
       GENERIC_TYPE);

 end case;

end ENCODE_ADA_TYPE_INTO_SIGNATURE;
```

-- procedure passed to generic scan procedure in generic map package
-- extracts the input parameters from the Type Declaration
-- map. NOTE that this procedure will be run once for each input
-- parameter as the entire set of input parameters is iterated through.

```
procedure GET_IN_PARAMETER(ID        : in PSDL_ID_PKG.PSDL_ID;
                           TYPE_NAME : in PSDL_CONCRETE_TYPE_PKG.TYPE_NAME) is
```

-- User Defined Type (UDT) type name

```
 UDT_TYPE_NAME, GENERIC_TYPE_NAME, ELEMENT_TYPE, INDEX_TYPE, THE_TYPE_NAME :
  PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
 IO_SWITCH                                    :
  IO_SWITCH_CLASSES;
 GENERIC_TYPE, FOUND_TYPE                     :
  BOOLEAN;
 OPERATOR_COMPONENT                           :
  DETERMINE_THE_ADA_TYPE_PKG.COMPONENT_TYPE := OPERATOR_COMP;

begin
 IO_SWITCH := INPUT_PARAMETER;
 THE_TYPE_NAME := TYPE_NAME;
 GENERIC_TYPE := FALSE;

 if RECOGNIZED_ADA_TYPE(THE_TYPE_NAME.NAME) then
```

```
  -- Input parameter is a valid Ada type so go ahead and encode
  -- the current Input_Signature with this Ada type.
   THE_TYPE_NAME.NAME := A_STRINGS.LOWER_TO_UPPER(THE_TYPE_NAME.NAME);
   ENCODE_ADA_TYPE_INTO_SIGNATURE(THE_TYPE_NAME.NAME, IO_SWITCH,
    GENERIC_TYPE);

  else
  -- Maybe the parameter is defined in the generics of the
  -- Operator component.
   CHECK_GENERICS(OPERATOR_COMPONENT, THE_TYPE_NAME, GENERIC_TYPE_NAME,
    FOUND_TYPE);

   if FOUND_TYPE then
   -- The type is either not composite or even if composite,
   -- will not be further specified by its components so
   -- go ahead and encode.
         UDT_TYPE_NAME := GENERIC_TYPE_NAME;
         GENERIC_TYPE := TRUE;

   else   -- The parameter must be a user defined type so
         -- get its type.

         GET_USER_DEFINED_TYPE(THE_TYPE_NAME, UDT_TYPE_NAME);

   end if;

   UDT_TYPE_NAME.NAME := A_STRINGS.LOWER_TO_UPPER(UDT_TYPE_NAME.NAME);
   -- go ahead and encode into the current Input_Signature.
   ENCODE_ADA_TYPE_INTO_SIGNATURE(UDT_TYPE_NAME.NAME, IO_SWITCH,
    GENERIC_TYPE);

  end if;

exception

  when others =>
   PUT_LINE("*** ERROR: OCCURED IN procedure GET_IN_PARAMETER. ");
   NEW_LINE;
   raise ;

end GET_IN_PARAMETER;




-- procedure passed to generic scan procedure in generic map package
-- extracts the output parameters from the Type Declaration
-- map.  NOTE that this procedure will be run once for each output
-- parameter as the entire set of output parameters is iterated through.
procedure GET_OUT_PARAMETER(ID       : in PSDL_ID_PKG.PSDL_ID;
                            TYPE_NAME : in PSDL_CONCRETE_TYPE_PKG.TYPE_NAME) is
```

```
-- User Defined Type (UDT) type name
 UDT_TYPE_NAME, GENERIC_TYPE_NAME, ELEMENT_TYPE, INDEX_TYPE, THE_TYPE_NAME :
  PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
 IO_SWITCH                                          :
  IO_SWITCH_CLASSES;
 GENERIC_TYPE, FOUND_TYPE                           :
  BOOLEAN;
 OPERATOR_COMPONENT                                 :
  DETERMINE_THE_ADA_TYPE_PKG.COMPONENT_TYPE := OPERATOR_COMP;



begin
 IO_SWITCH := OUTPUT_PARAMETER;
 THE_TYPE_NAME := TYPE_NAME;
 GENERIC_TYPE := FALSE;

 if RECOGNIZED_ADA_TYPE(THE_TYPE_NAME.NAME) then
 -- Output parameter is a valid Ada type so go ahead and encode
 -- into the current Output_Signature.

  THE_TYPE_NAME.NAME := A_STRINGS.LOWER_TO_UPPER(THE_TYPE_NAME.NAME);
  ENCODE_ADA_TYPE_INTO_SIGNATURE(THE_TYPE_NAME.NAME, IO_SWITCH,
   GENERIC_TYPE);

 else
 -- Maybe the parameter is defined in the generics of the
 -- Operator component.
  CHECK_GENERICS(OPERATOR_COMPONENT, THE_TYPE_NAME, GENERIC_TYPE_NAME,
   FOUND_TYPE);

  if FOUND_TYPE then
  -- The type is either not composite or even if composite,
  -- will not be further specified by its components so
  -- go ahead and encode.
        UDT_TYPE_NAME := GENERIC_TYPE_NAME;
        GENERIC_TYPE := TRUE;

  else   -- The parameter must be a user defined type so
       -- get its type.

        GET_USER_DEFINED_TYPE(THE_TYPE_NAME, UDT_TYPE_NAME);

  end if;

  UDT_TYPE_NAME.NAME := A_STRINGS.LOWER_TO_UPPER(UDT_TYPE_NAME.NAME);
  -- Go ahead and encode.
  ENCODE_ADA_TYPE_INTO_SIGNATURE(UDT_TYPE_NAME.NAME, IO_SWITCH,
   GENERIC_TYPE);
 end if;
```

```
 exception

  when others =>
   PUT_LINE("*** ERROR: OCCURED IN procedure GET_OUT_PARAMETER. ");
   NEW_LINE;
   raise ;

 end GET_OUT_PARAMETER;

end ITERATE_THROUGH_OPERATOR_PARAMETERS_PKG;



*********************************************************************************
*********************************************************************************
*********************************************************************************



-- Filename    / Parameter_List_Pkg.a
-- Date        / 12 Aug 93
-- Revised     / 15 Aug 93
-- Author      / Scott Dolgoff
-- System      / Solbourne
-- Compiler    / Verdix Ada
-- Description / This package provides a data structure for storing
--             / parameter lists and the operations necessary to
--             / be performed on the data structure. The structure
--             / contains some attributes that are particular to one of
--             / the following three categories - (a) query component
--             / input and output parameters, (b) software base
--             / component input and output parameters, and (c) software
--             / base component generic parameters.
--             / The parameter lists are used during the component
--             / transformation process to help establish a final
--             / mapping of parameter types between the query
--             / and software base components.


with PSDL_ID_PKG;
use  PSDL_ID_PKG;

package PARAMETER_LIST_PKG is


 type PARAMETER_LIST_NODE;
 type PARAMETERS       is access PARAMETER_LIST_NODE;
```

```
type PARAMETER_LIST_NODE is record
-- parameter identifier name
 ID                : PSDL_ID_PKG.PSDL_ID;

 -- parameter type name
 THE_TYPE          : PSDL_ID_PKG.PSDL_ID;

 -- array element type name (only used if THE_TYPE is an array)
 ARRAY_ELEMENT_PTR     : PARAMETERS;

 -- array index type name (only used if THE_TYPE is an array)
 ARRAY_INDEX_PTR       : PARAMETERS;

 -- pointer to the parameter in other component that this
 -- parameter is mapped to.  null indicates no mapping. For
 -- generic parameters this will also help locate the Ada
 -- type the generic is instantiated with.
 MAPPING           : PARAMETERS := null;

 -- link
 NEXT              : PARAMETERS := null;

 -- user defined type (UDT). It is necessary to record a
 -- UDT referenced by a parameter.  Certain UDTs will be
 -- of an Ada type that must be either constrained (i.e. like
 -- Range or better defined (i.e. like Record).  If these
 -- UDTs are used to instantiate a generic, then they must
 -- be used exactly as implemented in the protype to make
 -- use of their constraint or definition information.  Thus
 -- in the transformation shell their Ada specification
 -- packages must be "with"ed and their names referred to
 -- explicitly.
 UDT               : PSDL_ID_PKG.PSDL_ID;

 -- indicates whether THE_TYPE came from a UDT
 HAS_UDT           : BOOLEAN   := FALSE;


 -- This attribute is used only by sbc components.  It names
 -- he generic parameter that this sbc parameter is
 --  defined by (if any)
 GENERIC_ID        : PSDL_ID_PKG.PSDL_ID;

 -- This attribute is used only by sbc components.  It is a
 -- pointer to the generic parameter that this sbc parameter
 -- is defined by (if any)
```

```
  GENERIC_LINK          : PARAMETERS := null;

  -- Indicates whether the sbc component parameter is defined by
  -- a generic parameter in the sbc component.
  DEFINED_BY_GENERIC_TYPE : BOOLEAN   := FALSE;

 end record;


 -- takes an index position and returns the parameter in the list
 -- corresponding to that index position
 function PARAMETER_AT_INDEX(INDEX : INTEGER;
                             LIST  : PARAMETERS) return PARAMETERS;


 -- determines whether any parameters in the list are defined by
 -- generics
 function HAS_GENERIC_PARAMETERS(LIST : PARAMETERS) return BOOLEAN;



 --- determines how many parameters are in the list
 function PARAMETER_COUNT(LIST : PARAMETERS) return INTEGER;


end PARAMETER_LIST_PKG;


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

package body PARAMETER_LIST_PKG is


-- corresponding to that index position
 function PARAMETER_AT_INDEX(INDEX : INTEGER;
                             LIST  : PARAMETERS) return PARAMETERS is

  LIST_PTR : PARAMETERS := LIST;

 begin

  if LIST_PTR /= null then

   for POSITION in 1 .. (INDEX - 1) loop

        LIST_PTR := LIST_PTR.NEXT;
        if (LIST_PTR = null and POSITION < (INDEX - 1)) then
          return LIST_PTR;
        end if;

   end loop;
  end if;
```

```
  return LIST_PTR;

 end PARAMETER_AT_INDEX;

 -- determines whether any parameters in the list are defined by
 -- generics
 function HAS_GENERIC_PARAMETERS(LIST : PARAMETERS) return BOOLEAN is

  HAS_GENERICS : BOOLEAN   := FALSE;
  PTR          : PARAMETERS := LIST;

 begin

  while ((not HAS_GENERICS) and (PTR /= null)) loop

   if PTR.DEFINED_BY_GENERIC_TYPE then
         HAS_GENERICS := TRUE;
   else
         PTR := PTR.NEXT;
   end if;

  end loop;

  return HAS_GENERICS;

 end HAS_GENERIC_PARAMETERS;


 --- determines how many parameters are in the list
 function PARAMETER_COUNT(LIST : PARAMETERS) return INTEGER is

  PTR   : PARAMETERS := LIST;
  COUNT : INTEGER    := 0;

 begin

  while (PTR /= null) loop

   COUNT := COUNT + 1;
   PTR := PTR.NEXT;

  end loop;

  return COUNT;

 end PARAMETER_COUNT;

end PARAMETER_LIST_PKG;
```

```
*******************************************************************************
*******************************************************************************
*******************************************************************************


-- Filename    / Parameter_Mapping_Pkg.a
-- Date        / 12 Aug 93
-- Revised     / 18 Aug 93
-- Author      / Scott Dolgoff
-- System      / Solbourne
-- Compiler    / Verdix Ada
-- Description / This package provides routines that help in the
--             / process of determining whether a query component
--             / parameter type correctly maps to a software base
--             / component parameter type.


with PARAMETER_LIST_PKG;
use  PARAMETER_LIST_PKG;

package PARAMETER_MAPPING_PKG is


-- determines whether a query and software base component input
-- parameter validly map to each other based on their Ada type.
 function INPUT_PARAMETER_TYPES_MAP_OK(QC_PARAMETER  : PARAMETERS;
                                       SBC_PARAMETER : PARAMETERS)
                                       return BOOLEAN;

 -- determines whether a query and software base component output
 -- parameter validly map to each other based on their Ada type.
 function OUTPUT_PARAMETER_TYPES_MAP_OK(QC_PARAMETER  : PARAMETERS;
                                        SBC_PARAMETER : PARAMETERS)
                                        return BOOLEAN;


end PARAMETER_MAPPING_PKG;



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


with TEXT_IO, ADA_RECOGNIZED_TYPES_PKG, A_STRINGS, PSDL_ID_PKG;
use  TEXT_IO, ADA_RECOGNIZED_TYPES_PKG, A_STRINGS, PSDL_ID_PKG;

package body PARAMETER_MAPPING_PKG is
```

```
-- determines which array component, element or index, to deal with
type ARRAY_COMP_SWITCH is (ELEMENT, INDEX);

INVALID_ADA_TYPE : exception;


-- load the String array components into their respective
-- slots in the pointer node
function LOAD_STRING_COMPONENT(SWITCH : ARRAY_COMP_SWITCH)
                                    return PSDL_ID_PKG.PSDL_ID is

 NEW_NODE : PSDL_ID_PKG.PSDL_ID;

begin

 case SWITCH is

  when ELEMENT =>

       NEW_NODE := new A_STRINGS.STRING_REC(9);
       NEW_NODE.S := CHARACTER_TYPE;

  when INDEX =>

       NEW_NODE := new A_STRINGS.STRING_REC(8);
       NEW_NODE.S := POSITIVE_TYPE;

 end case;

 return NEW_NODE;

end LOAD_STRING_COMPONENT;




-- checks a mapping on the array component determined by the
-- switch
function CHECK_INPUT_ARRAY_COMPONENTS(QC_PARAMETER  : PARAMETERS;
                                      SBC_PARAMETER : PARAMETERS;
                                      SWITCH        : ARRAY_COMP_SWITCH)
                                      return BOOLEAN is

 SBC_TYPE_NAME, QC_TYPE_NAME : PSDL_ID_PKG.PSDL_ID;
 MAP_IS_VALID          : BOOLEAN := FALSE;

begin

 case SWITCH is
 -- special attention must be given to Strings since they
 -- do not have their components predefined ... so we must
 -- define the String components in order to compare them
```

```
-- with an array's components

 when ELEMENT =>

        if (QC_PARAMETER.THE_TYPE.S /= STRING_TYPE) then
         QC_TYPE_NAME :=
          A_STRINGS.LOWER_TO_UPPER(QC_PARAMETER.ARRAY_ELEMENT_PTR.THE_TYPE);

        else
         QC_TYPE_NAME := LOAD_STRING_COMPONENT(SWITCH);
        end if;

        if (SBC_PARAMETER.THE_TYPE.S /= STRING_TYPE) then

         SBC_TYPE_NAME :=
          A_STRINGS.LOWER_TO_UPPER(SBC_PARAMETER.ARRAY_ELEMENT_PTR.THE_TYPE);

        else
         SBC_TYPE_NAME := LOAD_STRING_COMPONENT(SWITCH);
        end if;


 when INDEX =>

        if (QC_PARAMETER.THE_TYPE.S /= STRING_TYPE) then
         QC_TYPE_NAME :=
          A_STRINGS.LOWER_TO_UPPER(QC_PARAMETER.ARRAY_INDEX_PTR.THE_TYPE);

        else
         QC_TYPE_NAME := LOAD_STRING_COMPONENT(SWITCH);
        end if;

        if (SBC_PARAMETER.THE_TYPE.S /= STRING_TYPE) then

         SBC_TYPE_NAME :=
          A_STRINGS.LOWER_TO_UPPER(SBC_PARAMETER.ARRAY_INDEX_PTR.THE_TYPE);

        else
         SBC_TYPE_NAME := LOAD_STRING_COMPONENT(SWITCH);
        end if;


end case;


if (PRIVATE_TYPE = QC_TYPE_NAME.S) then
 if (PRIVATE_TYPE = SBC_TYPE_NAME.S) then          -- regular
                                    -- or
                                    -- generic
                                    -- Private
        MAP_IS_VALID := TRUE;
 end if;

elsif (DISCRETE_TYPE = QC_TYPE_NAME.S) then
```

```
 if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                -- generic
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or             -- Private
  (DISCRETE_TYPE = SBC_TYPE_NAME.S)) then
       MAP_IS_VALID := TRUE;
 end if;

elsif (INTEGER_TYPE = QC_TYPE_NAME.S) then
 if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                -- generic
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or             -- Private
  (DISCRETE_TYPE = SBC_TYPE_NAME.S) or (INTEGER_TYPE =
  SBC_TYPE_NAME.S)) then
       MAP_IS_VALID := TRUE;
 end if;

elsif (RANGE_TYPE = QC_TYPE_NAME.S) then
 if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                -- generic
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or             -- Private
  (DISCRETE_TYPE = SBC_TYPE_NAME.S) or (INTEGER_TYPE =
  SBC_TYPE_NAME.S) or (RANGE_TYPE = SBC_TYPE_NAME.S)) then
       MAP_IS_VALID := TRUE;
 end if;

elsif (NATURAL_TYPE = QC_TYPE_NAME.S) then
 if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                -- generic
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or             -- Private
  (DISCRETE_TYPE = SBC_TYPE_NAME.S) or (INTEGER_TYPE =
  SBC_TYPE_NAME.S) or (RANGE_TYPE = SBC_TYPE_NAME.S) or
  (NATURAL_TYPE = SBC_TYPE_NAME.S)) then
       MAP_IS_VALID := TRUE;
 end if;

elsif (POSITIVE_TYPE = QC_TYPE_NAME.S) then
 if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                -- generic
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or             -- Private
  (DISCRETE_TYPE = SBC_TYPE_NAME.S) or (INTEGER_TYPE =
  SBC_TYPE_NAME.S) or (RANGE_TYPE = SBC_TYPE_NAME.S) or
  (NATURAL_TYPE = SBC_TYPE_NAME.S) or (POSITIVE_TYPE =
  SBC_TYPE_NAME.S)) then
       MAP_IS_VALID := TRUE;
 end if;

elsif (ENUMERATION_TYPE = QC_TYPE_NAME.S) then
 if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                -- generic
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or             -- Private
  (DISCRETE_TYPE = SBC_TYPE_NAME.S) or (ENUMERATION_TYPE =
  SBC_TYPE_NAME.S)) then
       MAP_IS_VALID := TRUE;
 end if;

elsif (CHARACTER_TYPE = QC_TYPE_NAME.S) then
 if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                -- generic
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or             -- Private
```

```
  (DISCRETE_TYPE = SBC_TYPE_NAME.S) or (ENUMERATION_TYPE =
  SBC_TYPE_NAME.S) or (CHARACTER_TYPE =
  SBC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (BOOLEAN_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                -- generic
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or              -- Private
  (DISCRETE_TYPE = SBC_TYPE_NAME.S) or (ENUMERATION_TYPE =
  SBC_TYPE_NAME.S) or (BOOLEAN_TYPE = SBC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (RECORD_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                -- generic
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or (RECORD_TYPE =   -- Private
  SBC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (ACCESS_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                -- generic
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or (ACCESS_TYPE =   -- Private
  SBC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (ARRAY_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                -- generic
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or (ARRAY_TYPE =    -- Private
  SBC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (STRING_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                -- generic
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or (ARRAY_TYPE =    -- Private
  SBC_TYPE_NAME.S) or (STRING_TYPE = SBC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (DIGITS_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                -- generic
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or (DIGITS_TYPE =   -- Private
  SBC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (FLOAT_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                -- generic
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or (DIGITS_TYPE =   -- Private
```

```
   SBC_TYPE_NAME.S) or (FLOAT_TYPE = SBC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
   end if;

  elsif (DELTA_TYPE = QC_TYPE_NAME.S) then
   if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                    -- generic
   (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or (DELTA_TYPE =    -- Private
   SBC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
   end if;

  elsif (FIXED_TYPE = QC_TYPE_NAME.S) then
   if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                    -- generic
   (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or (DELTA_TYPE =    -- Private
   SBC_TYPE_NAME.S) or (FIXED_TYPE = SBC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
   end if;

  else                                          -- unknown
                                                -- type
                                                -- passed
   raise INVALID_ADA_TYPE;

  end if;

  return MAP_IS_VALID;

 exception

  when INVALID_ADA_TYPE =>
   PUT_LINE("*** ERROR:  INVALID ADA TYPE DISCOVERED FOR -");
   PUT_LINE("          " & QC_PARAMETER.ID.S & " : " & QC_TYPE_NAME.S);
   PUT_LINE("A Type MUST BE DEFINED AS AN ADA TYPE.  IT CAN REFERENCE");
   PUT_LINE("INFORMATION IN ITS GENERIC PARAMETERS, BUT CANNOT");
   PUT_LINE("REFERENCE DEFINITION INFORMATION OUTSIDE ITS OWN");
   PUT_LINE("SPECIFICATION.  A Generic PARAMETER MUST BE FULLY ");
   PUT_LINE("DEFINED AS AN ADA TYPE.  IT CAN'T EVEN REFERENCE OTHER ");
   PUT_LINE("DEFINITIONS WITHIN ITS GENERICS.");
   NEW_LINE;
   raise ;

  when others =>
   PUT_LINE("*** UNKNOWN ERROR:  OCCURED IN procedure ");
   PUT_LINE("                CHECK_INPUT_ARRAY_COMPONENTS. ");
   NEW_LINE;
   raise ;

end CHECK_INPUT_ARRAY_COMPONENTS;
```

```
-- determines whether a query and software base component input
-- parameter validly map to each other based on their Ada type.
function INPUT_PARAMETER_TYPES_MAP_OK(QC_PARAMETER : PARAMETERS;
                                      SBC_PARAMETER : PARAMETERS)
                                      return BOOLEAN is
 SBC_TYPE_NAME, QC_TYPE_NAME : PSDL_ID_PKG.PSDL_ID;
 MAP_IS_VALID          : BOOLEAN        := FALSE;
 ELEMENT_SWITCH        : ARRAY_COMP_SWITCH := ELEMENT;
 INDEX_SWITCH          : ARRAY_COMP_SWITCH := INDEX;


begin


 QC_TYPE_NAME := A_STRINGS.LOWER_TO_UPPER(QC_PARAMETER.THE_TYPE);
 SBC_TYPE_NAME := A_STRINGS.LOWER_TO_UPPER(SBC_PARAMETER.THE_TYPE);


 if (PRIVATE_TYPE = QC_TYPE_NAME.S) then
  if (PRIVATE_TYPE = SBC_TYPE_NAME.S) then          -- regular
                                                    -- or
                                                    -- generic
                                                    -- Private
       MAP_IS_VALID := TRUE;
  end if;

 elsif (DISCRETE_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and         -- generic
   (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or      -- Private
   (DISCRETE_TYPE = SBC_TYPE_NAME.S)) then
       MAP_IS_VALID := TRUE;
  end if;

 elsif (INTEGER_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and         -- generic
   (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or      -- Private
   (DISCRETE_TYPE = SBC_TYPE_NAME.S) or (INTEGER_TYPE =
   SBC_TYPE_NAME.S)) then
       MAP_IS_VALID := TRUE;
  end if;

 elsif (RANGE_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and         -- generic
   (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or      -- Private
   (DISCRETE_TYPE = SBC_TYPE_NAME.S) or (INTEGER_TYPE =
   SBC_TYPE_NAME.S) or (RANGE_TYPE = SBC_TYPE_NAME.S)) then
       MAP_IS_VALID := TRUE;
  end if;

 elsif (NATURAL_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and         -- generic
```

```
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or              -- Private
  (DISCRETE_TYPE = SBC_TYPE_NAME.S) or (INTEGER_TYPE =
  SBC_TYPE_NAME.S) or (NATURAL_TYPE = SBC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (POSITIVE_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                -- generic
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or              -- Private
  (DISCRETE_TYPE = SBC_TYPE_NAME.S) or (INTEGER_TYPE =
  SBC_TYPE_NAME.S) or (NATURAL_TYPE = SBC_TYPE_NAME.S) or
  (POSITIVE_TYPE = SBC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (ENUMERATION_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                -- generic
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or              -- Private
  (DISCRETE_TYPE = SBC_TYPE_NAME.S) or (ENUMERATION_TYPE =
  SBC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (CHARACTER_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                -- generic
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or              -- Private
  (DISCRETE_TYPE = SBC_TYPE_NAME.S) or (ENUMERATION_TYPE =
  SBC_TYPE_NAME.S) or (CHARACTER_TYPE =
  SBC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (BOOLEAN_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                -- generic
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or              -- Private
  (DISCRETE_TYPE = SBC_TYPE_NAME.S) or (ENUMERATION_TYPE =
  SBC_TYPE_NAME.S) or (BOOLEAN_TYPE = SBC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (RECORD_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                -- generic
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or (RECORD_TYPE =   -- Private
  SBC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (ACCESS_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                -- generic
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or (ACCESS_TYPE =   -- Private
  SBC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
```

```
  end if;

elsif (ARRAY_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                    -- generic
   (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or                 -- Private
   (((STRING_TYPE = SBC_TYPE_NAME.S) or (ARRAY_TYPE =
   SBC_TYPE_NAME.S)) and then
   (CHECK_INPUT_ARRAY_COMPONENTS(QC_PARAMETER, SBC_PARAMETER,
   ELEMENT_SWITCH) and
   CHECK_INPUT_ARRAY_COMPONENTS(QC_PARAMETER, SBC_PARAMETER,
   INDEX_SWITCH)))) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (STRING_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                    -- generic
   (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or (STRING_TYPE =  -- Private
   SBC_TYPE_NAME.S) or ((ARRAY_TYPE =
   SBC_TYPE_NAME.S) and then
   (CHECK_INPUT_ARRAY_COMPONENTS(QC_PARAMETER, SBC_PARAMETER,
   ELEMENT_SWITCH)) and then
   (CHECK_INPUT_ARRAY_COMPONENTS(QC_PARAMETER, SBC_PARAMETER,
   INDEX_SWITCH))) or (STRING_TYPE = SBC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (DIGITS_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                    -- generic
   (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or (DIGITS_TYPE =  -- Private
   SBC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (FLOAT_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                    -- generic
   (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or (DIGITS_TYPE =  -- Private
   SBC_TYPE_NAME.S) or (FLOAT_TYPE = SBC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (DELTA_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                    -- generic
   (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or (DELTA_TYPE =   -- Private
   SBC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (FIXED_TYPE = QC_TYPE_NAME.S) then
  if (((PRIVATE_TYPE = SBC_TYPE_NAME.S) and                    -- generic
   (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) or (DELTA_TYPE =   -- Private
   SBC_TYPE_NAME.S) or (FIXED_TYPE = SBC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
```

```
  end if;

 else                                    -- unknown
                                         -- type
                                         -- passed
  raise INVALID_ADA_TYPE;

 end if;

 return MAP_IS_VALID;

exception

 when INVALID_ADA_TYPE =>
  PUT_LINE("*** ERROR:  INVALID ADA TYPE DISCOVERED FOR -");
  PUT_LINE("          " & QC_PARAMETER.ID.S & " : " & QC_TYPE_NAME.S);
  PUT_LINE("A Type MUST BE DEFINED AS AN ADA TYPE.  IT CAN REFERENCE");
  PUT_LINE("INFORMATION IN ITS GENERIC PARAMETERS, BUT CANNOT");
  PUT_LINE("REFERENCE DEFINITION INFORMATION OUTSIDE ITS OWN");
  PUT_LINE("SPECIFICATION.  A Generic PARAMETER MUST BE FULLY ");
  PUT_LINE("DEFINED AS AN ADA TYPE.  IT CAN'T EVEN REFERENCE OTHER ");
  PUT_LINE("DEFINITIONS WITHIN ITS GENERICS.");
  NEW_LINE;
  raise ;

 when others =>
  PUT_LINE("*** UNKNOWN ERROR:  OCCURED IN procedure ");
  PUT_LINE("                INPUT_PARAMETER_TYPES_MAP_OK. ");
  NEW_LINE;
  raise ;

end INPUT_PARAMETER_TYPES_MAP_OK;




-- checks a mapping on the array component determined by the
-- switch
function CHECK_OUTPUT_ARRAY_COMPONENTS(QC_PARAMETER : PARAMETERS;
                                       SBC_PARAMETER : PARAMETERS;
                                       SWITCH      : ARRAY_COMP_SWITCH)
                                       return BOOLEAN is

 SBC_TYPE_NAME, QC_TYPE_NAME : PSDL_ID_PKG.PSDL_ID;
 MAP_IS_VALID            : BOOLEAN := FALSE;
 INVALID_QC              : BOOLEAN := FALSE;

begin

 case SWITCH is

  when ELEMENT =>

       if (QC_PARAMETER.THE_TYPE.S /= STRING_TYPE) then
```

```
          QC_TYPE_NAME :=
           A_STRINGS.LOWER_TO_UPPER(QC_PARAMETER.ARRAY_ELEMENT_PTR.THE_TYPE);

         else
          QC_TYPE_NAME := LOAD_STRING_COMPONENT(SWITCH);
         end if;

         if (SBC_PARAMETER.THE_TYPE.S /= STRING_TYPE) then

          SBC_TYPE_NAME :=
           A_STRINGS.LOWER_TO_UPPER(SBC_PARAMETER.ARRAY_ELEMENT_PTR.THE_TYPE);

         else
          SBC_TYPE_NAME := LOAD_STRING_COMPONENT(SWITCH);
         end if;


  when INDEX =>

         if (QC_PARAMETER.THE_TYPE.S /= STRING_TYPE) then
          QC_TYPE_NAME :=
           A_STRINGS.LOWER_TO_UPPER(QC_PARAMETER.ARRAY_INDEX_PTR.THE_TYPE);

         else
          QC_TYPE_NAME := LOAD_STRING_COMPONENT(SWITCH);
         end if;

         if (SBC_PARAMETER.THE_TYPE.S /= STRING_TYPE) then

          SBC_TYPE_NAME :=
           A_STRINGS.LOWER_TO_UPPER(SBC_PARAMETER.ARRAY_INDEX_PTR.THE_TYPE);

         else
          SBC_TYPE_NAME := LOAD_STRING_COMPONENT(SWITCH);
         end if;


end case;

if ((PRIVATE_TYPE = SBC_TYPE_NAME.S) and        -- generic Private
 (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) then
-- it can map to any query component parameter
-- as long as that paramater is a valid Ada type
  if RECOGNIZED_ADA_TYPE(QC_TYPE_NAME) then
         MAP_IS_VALID := TRUE;

  else
         INVALID_QC := TRUE;
         raise INVALID_ADA_TYPE;
  end if;

elsif (PRIVATE_TYPE = SBC_TYPE_NAME.S) then      -- regular Private
  if (PRIVATE_TYPE = QC_TYPE_NAME.S) then
         MAP_IS_VALID := TRUE;
  end if;
```

```
elsif ((DISCRETE_TYPE = SBC_TYPE_NAME.S) and     -- generic Discrete
 (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) then
  if ((DISCRETE_TYPE = QC_TYPE_NAME.S) or (INTEGER_TYPE = QC_TYPE_NAME.S) or
   (RANGE_TYPE = QC_TYPE_NAME.S) or (NATURAL_TYPE = QC_TYPE_NAME.S) or
   (POSITIVE_TYPE = QC_TYPE_NAME.S) or (ENUMERATION_TYPE =
   QC_TYPE_NAME.S) or (CHARACTER_TYPE = QC_TYPE_NAME.S) or (BOOLEAN_TYPE =
   QC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (DISCRETE_TYPE = SBC_TYPE_NAME.S) then
  if (DISCRETE_TYPE = QC_TYPE_NAME.S) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (INTEGER_TYPE = SBC_TYPE_NAME.S) then
  if ((DISCRETE_TYPE = QC_TYPE_NAME.S) or (INTEGER_TYPE =
   QC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif ((RANGE_TYPE = SBC_TYPE_NAME.S) and        -- generic Range
 (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) then
  if ((INTEGER_TYPE = QC_TYPE_NAME.S) or (NATURAL_TYPE = QC_TYPE_NAME.S) or
   (POSITIVE_TYPE = QC_TYPE_NAME.S) or (RANGE_TYPE = QC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (RANGE_TYPE = SBC_TYPE_NAME.S) then
  if (RANGE_TYPE = QC_TYPE_NAME.S) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (NATURAL_TYPE = SBC_TYPE_NAME.S) then
  if ((DISCRETE_TYPE = QC_TYPE_NAME.S) or (INTEGER_TYPE = QC_TYPE_NAME.S) or
   (NATURAL_TYPE = QC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (POSITIVE_TYPE = SBC_TYPE_NAME.S) then
  if ((DISCRETE_TYPE = QC_TYPE_NAME.S) or (INTEGER_TYPE = QC_TYPE_NAME.S) or
   (NATURAL_TYPE = QC_TYPE_NAME.S) or (POSITIVE_TYPE = QC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (ENUMERATION_TYPE = SBC_TYPE_NAME.S) then
  if ((DISCRETE_TYPE = QC_TYPE_NAME.S) or (ENUMERATION_TYPE =
   QC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (CHARACTER_TYPE = SBC_TYPE_NAME.S) then
  if ((DISCRETE_TYPE = QC_TYPE_NAME.S) or (ENUMERATION_TYPE =
   QC_TYPE_NAME.S) or (CHARACTER_TYPE = QC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
```

```
  end if;

elsif (BOOLEAN_TYPE = SBC_TYPE_NAME.S) then
  if ((DISCRETE_TYPE = QC_TYPE_NAME.S) or (ENUMERATION_TYPE =
   QC_TYPE_NAME.S) or (BOOLEAN_TYPE = QC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (RECORD_TYPE = SBC_TYPE_NAME.S) then
  if (RECORD_TYPE = QC_TYPE_NAME.S) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (ACCESS_TYPE = SBC_TYPE_NAME.S) then
  if (ACCESS_TYPE = QC_TYPE_NAME.S) then
        MAP_IS_VALID := TRUE;
  end if;

elsif ((ARRAY_TYPE = SBC_TYPE_NAME.S) and       -- generic Array
 (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) then
  if ((ARRAY_TYPE = QC_TYPE_NAME.S) or (STRING_TYPE = QC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (ARRAY_TYPE = SBC_TYPE_NAME.S) then
-- could miss a potential match between an sbc array element of
-- type array which maps to a qc array element of type string,
-- but this implementation does not look at any level lower
-- than first level array components
  if (ARRAY_TYPE = QC_TYPE_NAME.S) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (STRING_TYPE = SBC_TYPE_NAME.S) then
  if ((ARRAY_TYPE = QC_TYPE_NAME.S) or (STRING_TYPE = QC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif ((DIGITS_TYPE = SBC_TYPE_NAME.S) and      -- generic Digits
 (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) then
  if ((DIGITS_TYPE = QC_TYPE_NAME.S) or (FLOAT_TYPE = QC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (DIGITS_TYPE = SBC_TYPE_NAME.S) then
  if (DIGITS_TYPE = QC_TYPE_NAME.S) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (FLOAT_TYPE = SBC_TYPE_NAME.S) then
  if ((DIGITS_TYPE = QC_TYPE_NAME.S) or (FLOAT_TYPE = QC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;
```

```
 elsif ((DELTA_TYPE = SBC_TYPE_NAME.S) and        -- generic Delta
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) then
   if ((DELTA_TYPE = QC_TYPE_NAME.S) or (FIXED_TYPE = QC_TYPE_NAME.S)) then
         MAP_IS_VALID := TRUE;
   end if;

 elsif (DELTA_TYPE = SBC_TYPE_NAME.S) then
   if (DELTA_TYPE = QC_TYPE_NAME.S) then
         MAP_IS_VALID := TRUE;
   end if;

 elsif (FIXED_TYPE = SBC_TYPE_NAME.S) then
   if ((DELTA_TYPE = QC_TYPE_NAME.S) or (FIXED_TYPE = QC_TYPE_NAME.S)) then
         MAP_IS_VALID := TRUE;
   end if;

 else                                  -- unknown type passed
   raise INVALID_ADA_TYPE;

 end if;

 return MAP_IS_VALID;

exception

 when INVALID_ADA_TYPE =>
   PUT_LINE("*** ERROR:  INVALID ADA TYPE DISCOVERED FOR -");
   if not INVALID_QC then
         PUT_LINE("         " & SBC_PARAMETER.ID.S & " : " & SBC_TYPE_NAME.S);
   else
         PUT_LINE("         " & QC_PARAMETER.ID.S & " : " & QC_TYPE_NAME.S);
   end if;
   PUT_LINE("A Type MUST BE DEFINED AS AN ADA TYPE. IT CAN REFERENCE");
   PUT_LINE("INFORMATION IN ITS GENERIC PARAMETERS, BUT CANNOT");
   PUT_LINE("REFERENCE DEFINITION INFORMATION OUTSIDE ITS OWN");
   PUT_LINE("SPECIFICATION.  A Generic PARAMETER MUST BE FULLY ");
   PUT_LINE("DEFINED AS AN ADA TYPE.  IT CAN'T EVEN REFERENCE OTHER ");
   PUT_LINE("DEFINITIONS WITHIN ITS GENERICS.");
   NEW_LINE;
   raise ;

 when others =>
   PUT_LINE("*** UNKNOWN ERROR:  OCCURED IN procedure ");
   PUT_LINE("                CHECK_OUTPUT_ARRAY_COMPONENTS. ");
   NEW_LINE;
   raise ;

end CHECK_OUTPUT_ARRAY_COMPONENTS;
```

-- determines whether a query and software base component output
-- parameter validly map to each other based on their Ada type.

```
-- Note that the direction of comparison is different here than
-- with INPUT_PARAMETER_TYPES_MAP_OK.
function OUTPUT_PARAMETER_TYPES_MAP_OK(QC_PARAMETER : PARAMETERS;
                                       SBC_PARAMETER : PARAMETERS)
                                       return BOOLEAN is
 SBC_TYPE_NAME, QC_TYPE_NAME : PSDL_ID_PKG.PSDL_ID;
 MAP_IS_VALID          : BOOLEAN        := FALSE;
 INVALID_QC            : BOOLEAN        := FALSE;
 ELEMENT_SWITCH        : ARRAY_COMP_SWITCH := ELEMENT;
 INDEX_SWITCH          : ARRAY_COMP_SWITCH := INDEX;


begin


 QC_TYPE_NAME := A_STRINGS.LOWER_TO_UPPER(QC_PARAMETER.THE_TYPE);
 SBC_TYPE_NAME := A_STRINGS.LOWER_TO_UPPER(SBC_PARAMETER.THE_TYPE);


 if ((PRIVATE_TYPE = SBC_TYPE_NAME.S) and       -- generic Private
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) then
 -- it can map to any query component parameter
 -- as long as that paramater is a valid Ada type
  if RECOGNIZED_ADA_TYPE(QC_TYPE_NAME) then
        MAP_IS_VALID := TRUE;

  else
        INVALID_QC := TRUE;
        raise INVALID_ADA_TYPE;
  end if;

 elsif (PRIVATE_TYPE = SBC_TYPE_NAME.S) then      -- regular Private
  if (PRIVATE_TYPE = QC_TYPE_NAME.S) then
        MAP_IS_VALID := TRUE;
  end if;

 elsif ((DISCRETE_TYPE = SBC_TYPE_NAME.S) and     -- generic Discrete
  (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) then
  if ((DISCRETE_TYPE = QC_TYPE_NAME.S) or (INTEGER_TYPE = QC_TYPE_NAME.S) or
   (RANGE_TYPE = QC_TYPE_NAME.S) or (NATURAL_TYPE = QC_TYPE_NAME.S) or
   (POSITIVE_TYPE = QC_TYPE_NAME.S) or (ENUMERATION_TYPE =
   QC_TYPE_NAME.S) or (CHARACTER_TYPE = QC_TYPE_NAME.S) or (BOOLEAN_TYPE =
   QC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

 elsif (DISCRETE_TYPE = SBC_TYPE_NAME.S) then
  if (DISCRETE_TYPE = QC_TYPE_NAME.S) then
        MAP_IS_VALID := TRUE;
  end if;

 elsif (INTEGER_TYPE = SBC_TYPE_NAME.S) then
  if ((DISCRETE_TYPE = QC_TYPE_NAME.S) or (INTEGER_TYPE =
   QC_TYPE_NAME.S)) then
```

```
        MAP_IS_VALID := TRUE;
  end if;

elsif ((RANGE_TYPE = SBC_TYPE_NAME.S) and        -- generic Range
 (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) then
  if ((INTEGER_TYPE = QC_TYPE_NAME.S) or (NATURAL_TYPE = QC_TYPE_NAME.S) or
   (POSITIVE_TYPE = QC_TYPE_NAME.S) or (RANGE_TYPE = QC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (RANGE_TYPE = SBC_TYPE_NAME.S) then
  if (RANGE_TYPE = QC_TYPE_NAME.S) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (NATURAL_TYPE = SBC_TYPE_NAME.S) then
  if ((DISCRETE_TYPE = QC_TYPE_NAME.S) or (INTEGER_TYPE = QC_TYPE_NAME.S) or
   (NATURAL_TYPE = QC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (POSITIVE_TYPE = SBC_TYPE_NAME.S) then
  if ((DISCRETE_TYPE = QC_TYPE_NAME.S) or (INTEGER_TYPE = QC_TYPE_NAME.S) or
   (NATURAL_TYPE = QC_TYPE_NAME.S) or (POSITIVE_TYPE = QC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (ENUMERATION_TYPE = SBC_TYPE_NAME.S) then
  if ((DISCRETE_TYPE = QC_TYPE_NAME.S) or (ENUMERATION_TYPE =
   QC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (CHARACTER_TYPE = SBC_TYPE_NAME.S) then
  if ((DISCRETE_TYPE = QC_TYPE_NAME.S) or (ENUMERATION_TYPE =
   QC_TYPE_NAME.S) or (CHARACTER_TYPE = QC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (BOOLEAN_TYPE = SBC_TYPE_NAME.S) then
  if ((DISCRETE_TYPE = QC_TYPE_NAME.S) or (ENUMERATION_TYPE =
   QC_TYPE_NAME.S) or (BOOLEAN_TYPE = QC_TYPE_NAME.S)) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (RECORD_TYPE = SBC_TYPE_NAME.S) then
  if (RECORD_TYPE = QC_TYPE_NAME.S) then
        MAP_IS_VALID := TRUE;
  end if;

elsif (ACCESS_TYPE = SBC_TYPE_NAME.S) then
  if (ACCESS_TYPE = QC_TYPE_NAME.S) then
        MAP_IS_VALID := TRUE;
  end if;

elsif ((ARRAY_TYPE = SBC_TYPE_NAME.S) and        -- generic Array
```

```
   (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) then
    if ((ARRAY_TYPE = QC_TYPE_NAME.S) or (STRING_TYPE = QC_TYPE_NAME.S)) then
          MAP_IS_VALID          :=          CHECK_OUTPUT_ARRAY_COMPONENTS(QC_PARAMETER,
SBC_PARAMETER,
           ELEMENT_SWITCH);
          if MAP_IS_VALID then                    -- array element mapped
                                    -- correctly
                    MAP_IS_VALID     :=     CHECK_OUTPUT_ARRAY_COMPONENTS(QC_PARAMETER,
SBC_PARAMETER,
            INDEX_SWITCH);
          end if;
    end if;

   elsif (ARRAY_TYPE = SBC_TYPE_NAME.S) then
    if ((ARRAY_TYPE = QC_TYPE_NAME.S) or (STRING_TYPE = QC_TYPE_NAME.S)) then
          MAP_IS_VALID          :=          CHECK_OUTPUT_ARRAY_COMPONENTS(QC_PARAMETER,
SBC_PARAMETER,
           ELEMENT_SWITCH);
          if MAP_IS_VALID then                    -- array element mapped
                                    -- correctly
                    MAP_IS_VALID     :=     CHECK_OUTPUT_ARRAY_COMPONENTS(QC_PARAMETER,
SBC_PARAMETER,
            INDEX_SWITCH);
          end if;
    end if;

   elsif (STRING_TYPE = SBC_TYPE_NAME.S) then
    if (((ARRAY_TYPE = QC_TYPE_NAME.S) and then
     (CHECK_OUTPUT_ARRAY_COMPONENTS(QC_PARAMETER, SBC_PARAMETER,
     ELEMENT_SWITCH)) and then (CHECK_OUTPUT_ARRAY_COMPONENTS(QC_PARAMETER,
     SBC_PARAMETER, INDEX_SWITCH))) or (STRING_TYPE = QC_TYPE_NAME.S)) then
          MAP_IS_VALID := TRUE;
    end if;

   elsif ((DIGITS_TYPE = SBC_TYPE_NAME.S) and       -- generic Digits
    (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) then
    if ((DIGITS_TYPE = QC_TYPE_NAME.S) or (FLOAT_TYPE = QC_TYPE_NAME.S)) then
          MAP_IS_VALID := TRUE;
    end if;

   elsif (DIGITS_TYPE = SBC_TYPE_NAME.S) then
    if (DIGITS_TYPE = QC_TYPE_NAME.S) then
          MAP_IS_VALID := TRUE;
    end if;

   elsif (FLOAT_TYPE = SBC_TYPE_NAME.S) then
    if ((DIGITS_TYPE = QC_TYPE_NAME.S) or (FLOAT_TYPE = QC_TYPE_NAME.S)) then
          MAP_IS_VALID := TRUE;
    end if;

   elsif ((DELTA_TYPE = SBC_TYPE_NAME.S) and       -- generic Delta
    (SBC_PARAMETER.DEFINED_BY_GENERIC_TYPE)) then
    if ((DELTA_TYPE = QC_TYPE_NAME.S) or (FIXED_TYPE = QC_TYPE_NAME.S)) then
          MAP_IS_VALID := TRUE;
```

```
    end if;

  elsif (DELTA_TYPE = SBC_TYPE_NAME.S) then
    if (DELTA_TYPE = QC_TYPE_NAME.S) then
          MAP_IS_VALID := TRUE;
    end if;

  elsif (FIXED_TYPE = SBC_TYPE_NAME.S) then
    if ((DELTA_TYPE = QC_TYPE_NAME.S) or (FIXED_TYPE = QC_TYPE_NAME.S)) then
          MAP_IS_VALID := TRUE;
    end if;

  else                                  -- unknown type passed
    raise INVALID_ADA_TYPE;

  end if;

  return MAP_IS_VALID;

exception

  when INVALID_ADA_TYPE =>
    PUT_LINE("*** ERROR:  INVALID ADA TYPE DISCOVERED FOR -");
    if not INVALID_QC then
          PUT_LINE("          " & SBC_PARAMETER.ID.S & " : " & SBC_TYPE_NAME.S);
    else
          PUT_LINE("          " & QC_PARAMETER.ID.S & " : " & QC_TYPE_NAME.S);
    end if;
    PUT_LINE("A Type MUST BE DEFINED AS AN ADA TYPE. IT CAN REFERENCE");
    PUT_LINE("INFORMATION IN ITS GENERIC PARAMETERS, BUT CANNOT");
    PUT_LINE("REFERENCE DEFINITION INFORMATION OUTSIDE ITS OWN");
    PUT_LINE("SPECIFICATION.  A Generic PARAMETER MUST BE FULLY ");
    PUT_LINE("DEFINED AS AN ADA TYPE.  IT CAN'T EVEN REFERENCE OTHER ");
    PUT_LINE("DEFINITIONS WITHIN ITS GENERICS.");
    NEW_LINE;
    raise ;

  when others =>
    PUT_LINE("*** UNKNOWN ERROR:  OCCURED IN procedure ");
    PUT_LINE("                OUTPUT_PARAMETER_TYPES_MAP_OK. ");
    NEW_LINE;
    raise ;

end OUTPUT_PARAMETER_TYPES_MAP_OK;




end PARAMETER_MAPPING_PKG;


**********
**********
**********
```

```
-- Filename    / Parameter2_Iterator_Pkg.a
-- Date        / 10 Aug 93
-- Revised     / 18 Aug 93
-- Author      / Scott Dolgoff
-- System      / Solbourne
-- Compiler    / Verdix Ada
-- Description / This program writes the input and output parameters
--             / of software component's  Prototype System
--             / Description Language (PSDL) specifications into
--             / two separate files. These input/output parameter
--             / files are later used by the graphical user interface
--             / written with TAE to help transform components selected
--             / through matching to be brought into the prototype
--             / working directory. If the PSDL specification is of
--             / a software base component, then a check is made to
--             / see if it has generic parameters.  If it does
--             / then the generic parameters are written to a third
--             / file.
--             / It also creates input, output, and (if appropriate),
--             / generic parameter lists that are used to help establish
--             / a final mapping of parameter types between the query
--             / and software base components.


with PSDL_ID_PKG, PSDL_CONCRETE_TYPE_PKG, PARAMETER_LIST_PKG, PSDL_PROGRAM_PKG,
 PSDL_COMPONENT_PKG;
use  PSDL_ID_PKG, PSDL_CONCRETE_TYPE_PKG, PARAMETER_LIST_PKG, PSDL_PROGRAM_PKG,
 PSDL_COMPONENT_PKG;

package ITERATE2_THROUGH_OPERATOR_PARAMETERS_PKG is


-- gets the necessary component values needed for looking up
-- Ada type information
 procedure                    SEND_COMPONENTS(MAIN_COMPONENT                    :                    in
PSDL_COMPONENT_PKG.PSDL_COMPONENT;
                                PROTO_SPEC    : in PSDL_PROGRAM_PKG.PSDL_PROGRAM);



 -- build list of all parameters and how many there are
 procedure BUILD_PARAMETER_LIST(PARAM_LIST  : out PARAMETERS;
                                      PARAM_COUNT : out INTEGER);
```

```
-- Procedure that opens the desired file (input, output, or generic
-- parameters file) for the desired component (query or software base).
-- files for a query component or software base component will
-- be created
procedure OPEN_FILE(FILE_NAME : in STRING);


-- Procedure that closes the file opened by OPEN_FILE;
procedure CLOSE_FILE;


-- procedure passed to generic scan procedure in generic map package
-- extracts the input parameters from the Type Declaration
-- map
procedure GET_IN_PARAMETER(ID      : in PSDL_ID_PKG.PSDL_ID;
                           TYPE_NAME : in PSDL_CONCRETE_TYPE_PKG.TYPE_NAME);

-- iterate through the Type Declaration map to extract the set of input
-- parameters.
procedure ITERATE_THROUGH_INPUT_PARAMETERS is new
PSDL_CONCRETE_TYPE_PKG.TYPE_DECLARATION_PKG.GENERIC_SCAN(GENERATE=>GET_IN_PAR
AMETER);



-- procedure passed to generic scan procedure in generic map package
-- extracts the output parameters from the Type Declaration
-- map
procedure GET_OUT_PARAMETER(ID      : in PSDL_ID_PKG.PSDL_ID;
                            TYPE_NAME : in PSDL_CONCRETE_TYPE_PKG.TYPE_NAME);


-- iterate through the Type Declaration map to extract the set of output
-- parameters.
procedure ITERATE_THROUGH_OUTPUT_PARAMETERS is new
PSDL_CONCRETE_TYPE_PKG.TYPE_DECLARATION_PKG.GENERIC_SCAN(GENERATE=>GET_OUT_PA
RAMETER);



-- procedure passed to generic scan procedure in generic map package
-- extracts the generic parameters from the Type Declaration
-- map
procedure GET_GEN_PARAMETER(ID      : in PSDL_ID_PKG.PSDL_ID;
                            TYPE_NAME : in PSDL_CONCRETE_TYPE_PKG.TYPE_NAME);

-- iterate through the Type Declaration map to extract the set of generic
-- parameters.
```

```
 procedure                ITERATE_THROUGH_GENERIC_PARAMETERS                is                new
PSDL_CONCRETE_TYPE_PKG.TYPE_DECLARATION_PKG.GENERIC_SCAN(GENERATE=>GET_GEN_PA
RAMETER);


end ITERATE2_THROUGH_OPERATOR_PARAMETERS_PKG;



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



with TEXT_IO, ADA_RECOGNIZED_TYPES_PKG, DETERMINE_THE_ADA_TYPE_PKG,
 TYPE_NAME_PKG, A_STRINGS;
use  TEXT_IO, ADA_RECOGNIZED_TYPES_PKG, DETERMINE_THE_ADA_TYPE_PKG, A_STRINGS;

package body ITERATE2_THROUGH_OPERATOR_PARAMETERS_PKG is

 GV_NAME, GV_UDT_NAME  : PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
 -- It is necessary to know where the Array type was declared when
 -- examining the generics of a component. You need to know whether
 -- you should be looking at the Operator generics or a Type's generics.
 ARRAY_REFERENCE_IN_UDT : BOOLEAN;
 THE_FILE          : FILE_TYPE;
 PARAMETER_LIST_HEAD   : PARAMETERS := null;
 PARAMETER_LIST_PTR    : PARAMETERS := null;
 PARAMETER_COUNT       : INTEGER   := 0;
 IS_GENERIC          : BOOLEAN   := FALSE;
 ARRAY_IS_GENERIC      : BOOLEAN   := FALSE;




 -- gets the necessary component values needed for looking up
 -- Ada type information
 procedure                SEND_COMPONENTS(MAIN_COMPONENT                :                in
PSDL_COMPONENT_PKG.PSDL_COMPONENT;
                              PROTO_SPEC    : in PSDL_PROGRAM_PKG.PSDL_PROGRAM) is

 begin
  PASS_COMPONENTS(MAIN_COMPONENT, PROTO_SPEC);

 end SEND_COMPONENTS;



 -- build list of all parameters and how many there are
 procedure BUILD_PARAMETER_LIST(PARAM_LIST  : out PARAMETERS;
                                PARAM_COUNT : out INTEGER) is
```

```
begin
 PARAM_LIST := PARAMETER_LIST_HEAD;
 PARAM_COUNT := PARAMETER_COUNT;


 -- reinitialize package memory
 PARAMETER_LIST_HEAD := null;
 PARAMETER_LIST_PTR := null;
 PARAMETER_COUNT := 0;

end BUILD_PARAMETER_LIST;



-- Procedure that opens the desired file (input, output, or generic
-- parameters file) for the desired component (query or software base).
-- files for a query component or software base component will
-- be created
procedure OPEN_FILE(FILE_NAME : in STRING) is

begin

-- create the (input, output, or generic) parameter text file
 CREATE(THE_FILE, MODE => OUT_FILE, NAME => FILE_NAME);

end OPEN_FILE;


-- Procedure that closes the file opened by OPEN_FILE;
procedure CLOSE_FILE is

begin

 CLOSE(THE_FILE);

end CLOSE_FILE;



-- Writes the input, output, or generic parameter to its respective
-- text file.
procedure WRITE_PARAMETER_TO_FILE(TYPE_NAME_ID : PSDL_ID_PKG.PSDL_ID) is

begin
 if IS_GENERIC then
   PUT(THE_FILE, " : {G} " & TYPE_NAME_ID.S);
   IS_GENERIC := FALSE;
 else
   PUT(THE_FILE, " : " & TYPE_NAME_ID.S);
 end if;

end WRITE_PARAMETER_TO_FILE;
```

```
-- procedure passed to generic scan procedure in generic map package
-- extracts the input parameters from the Type Declaration
-- map.  NOTE that this procedure will be run once for each input
-- parameter as the entire set of input parameters is iterated through.
procedure GET_IN_PARAMETER(ID       : in PSDL_ID_PKG.PSDL_ID;
                           TYPE_NAME : in PSDL_CONCRETE_TYPE_PKG.TYPE_NAME) is


-- User Defined Type (UDT) type name
 UDT_TYPE_NAME, GENERIC_TYPE_NAME, ELEMENT_TYPE, INDEX_TYPE, THE_TYPE_NAME :
  PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
 ELEMENT_UDT, INDEX_UDT                                    :
  PSDL_ID_PKG.PSDL_ID;
 ELEMENT_IS_UDT, INDEX_IS_UDT, FOUND_TYPE                  :
  BOOLEAN;
 OPERATOR_COMPONENT                                        :
  DETERMINE_THE_ADA_TYPE_PKG.COMPONENT_TYPE := OPERATOR_COMP;

begin
 THE_TYPE_NAME := TYPE_NAME;


 -- write the name of the identifier to the file
 PUT(THE_FILE, ID.S);

 -- write the name of the identifier into the parameter list
 if PARAMETER_LIST_HEAD = null then
 -- initialize list
  PARAMETER_LIST_HEAD := new PARAMETER_LIST_NODE;
  PARAMETER_LIST_HEAD.ID := ID;
  PARAMETER_LIST_PTR := PARAMETER_LIST_HEAD;

 else
  PARAMETER_LIST_PTR.NEXT := new PARAMETER_LIST_NODE;
  PARAMETER_LIST_PTR := PARAMETER_LIST_PTR.NEXT;
  PARAMETER_LIST_PTR.ID := ID;
 end if;
 PARAMETER_COUNT := PARAMETER_COUNT + 1;

 if RECOGNIZED_ADA_TYPE(THE_TYPE_NAME.NAME) then
 -- Input parameter is a valid Ada type so go ahead and append
 -- the current file with this Ada type.
  THE_TYPE_NAME.NAME := A_STRINGS.LOWER_TO_UPPER(THE_TYPE_NAME.NAME);
  WRITE_PARAMETER_TO_FILE(THE_TYPE_NAME.NAME);
  PARAMETER_LIST_PTR.THE_TYPE := THE_TYPE_NAME.NAME;
  if THE_TYPE_NAME.NAME.S = ARRAY_TYPE then
        ARRAY_REFERENCE_IN_UDT := FALSE;
```

```
        GET_ARRAY_COMPONENTS(THE_TYPE_NAME,  ELEMENT_TYPE,  ELEMENT_IS_UDT,
ELEMENT_UDT,
        INDEX_TYPE, INDEX_IS_UDT, INDEX_UDT, ARRAY_REFERENCE_IN_UDT);
    -- add array components to file
        PUT(THE_FILE, "[AE");
        WRITE_PARAMETER_TO_FILE(ELEMENT_TYPE.NAME);
        PUT(THE_FILE, ", AI");
        WRITE_PARAMETER_TO_FILE(INDEX_TYPE.NAME);
        PUT_LINE(THE_FILE, "]");
    -- add new array element
        PARAMETER_LIST_PTR.ARRAY_ELEMENT_PTR := new PARAMETER_LIST_NODE;
        PARAMETER_LIST_PTR.ARRAY_ELEMENT_PTR.THE_TYPE := ELEMENT_TYPE.NAME;
        if ELEMENT_IS_UDT then
         PARAMETER_LIST_PTR.ARRAY_ELEMENT_PTR.UDT := ELEMENT_UDT;
         PARAMETER_LIST_PTR.ARRAY_ELEMENT_PTR.HAS_UDT := TRUE;
        end if;

    -- add new array index
        PARAMETER_LIST_PTR.ARRAY_INDEX_PTR := new PARAMETER_LIST_NODE;
        PARAMETER_LIST_PTR.ARRAY_INDEX_PTR.THE_TYPE := INDEX_TYPE.NAME;
        if INDEX_IS_UDT then
         PARAMETER_LIST_PTR.ARRAY_INDEX_PTR.UDT := INDEX_UDT;
         PARAMETER_LIST_PTR.ARRAY_INDEX_PTR.HAS_UDT := TRUE;
        end if;


   else   -- not an Array type
        NEW_LINE(THE_FILE);
   end if;

  else
  -- Maybe the parameter is defined in the generics of the
  -- Operator component.
   CHECK_GENERICS(OPERATOR_COMPONENT, THE_TYPE_NAME, GENERIC_TYPE_NAME,
    FOUND_TYPE);

   if FOUND_TYPE then
   -- The type is either not composite or even if composite,
   -- will not be further specified by its components so
   -- go ahead and encode.
        UDT_TYPE_NAME := GENERIC_TYPE_NAME;
        PARAMETER_LIST_PTR.GENERIC_ID := THE_TYPE_NAME.NAME;
        PARAMETER_LIST_PTR.THE_TYPE := GENERIC_TYPE_NAME.NAME;
        PARAMETER_LIST_PTR.DEFINED_BY_GENERIC_TYPE := TRUE;
        IS_GENERIC := TRUE;
        ARRAY_IS_GENERIC := TRUE;

   else   -- The parameter must be a user defined type so
         -- get its type.

        GET_USER_DEFINED_TYPE(THE_TYPE_NAME, UDT_TYPE_NAME);
        PARAMETER_LIST_PTR.THE_TYPE := UDT_TYPE_NAME.NAME;
```

```
    -- record the UDT name
        PARAMETER_LIST_PTR.UDT := THE_TYPE_NAME.NAME;
        PARAMETER_LIST_PTR.HAS_UDT := TRUE;

  end if;

  UDT_TYPE_NAME.NAME := A_STRINGS.LOWER_TO_UPPER(UDT_TYPE_NAME.NAME);
  WRITE_PARAMETER_TO_FILE(UDT_TYPE_NAME.NAME);
  if UDT_TYPE_NAME.NAME.S = ARRAY_TYPE then
        PUT(THE_FILE, "[AE");
        ARRAY_REFERENCE_IN_UDT := TRUE;
        GET_ARRAY_COMPONENTS(UDT_TYPE_NAME,     ELEMENT_TYPE,     ELEMENT_IS_UDT,
ELEMENT_UDT,
         INDEX_TYPE, INDEX_IS_UDT, INDEX_UDT, ARRAY_REFERENCE_IN_UDT);

    -- add array components to file
        WRITE_PARAMETER_TO_FILE(ELEMENT_TYPE.NAME);
        PUT(THE_FILE, ", AI");
        WRITE_PARAMETER_TO_FILE(INDEX_TYPE.NAME);
        PUT_LINE(THE_FILE, "]");
    -- add new array element
        PARAMETER_LIST_PTR.ARRAY_ELEMENT_PTR := new PARAMETER_LIST_NODE;
        PARAMETER_LIST_PTR.ARRAY_ELEMENT_PTR.THE_TYPE := ELEMENT_TYPE.NAME;
        if ELEMENT_IS_UDT then
         PARAMETER_LIST_PTR.ARRAY_ELEMENT_PTR.UDT := ELEMENT_UDT;
         PARAMETER_LIST_PTR.ARRAY_ELEMENT_PTR.HAS_UDT := TRUE;
        end if;

    -- add new array index
        PARAMETER_LIST_PTR.ARRAY_INDEX_PTR := new PARAMETER_LIST_NODE;
        PARAMETER_LIST_PTR.ARRAY_INDEX_PTR.THE_TYPE := INDEX_TYPE.NAME;
        if INDEX_IS_UDT then
         PARAMETER_LIST_PTR.ARRAY_INDEX_PTR.UDT := INDEX_UDT;
         PARAMETER_LIST_PTR.ARRAY_INDEX_PTR.HAS_UDT := TRUE;
        end if;

    -- if array is generic then components must reflect that
        if ARRAY_IS_GENERIC then

         ARRAY_IS_GENERIC := FALSE;
         PARAMETER_LIST_PTR.ARRAY_ELEMENT_PTR.DEFINED_BY_GENERIC_TYPE := TRUE;
         PARAMETER_LIST_PTR.ARRAY_INDEX_PTR.DEFINED_BY_GENERIC_TYPE := TRUE;

        end if;

  else  -- The type is either not ARRAY_TYPE or even if some other
      -- composite, will not be further specified by its components.
  -- added already so go to next line in file.
        NEW_LINE(THE_FILE);

  end if;
 end if;
```

```
exception

  when others =>
    PUT_LINE("*** ERROR: OCCURED IN procedure GET_IN_PARAMETER. ");
    NEW_LINE;
    raise ;

end GET_IN_PARAMETER;




-- procedure passed to generic scan procedure in generic map package
-- extracts the output parameters from the Type Declaration
-- map.  NOTE that this procedure will be run once for each output
-- parameter as the entire set of output parameters is iterated through.
procedure GET_OUT_PARAMETER(ID       : in PSDL_ID_PKG.PSDL_ID;
                            TYPE_NAME : in PSDL_CONCRETE_TYPE_PKG.TYPE_NAME) is


-- User Defined Type (UDT) type name
  UDT_TYPE_NAME, GENERIC_TYPE_NAME, ELEMENT_TYPE, INDEX_TYPE, THE_TYPE_NAME :
    PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
  ELEMENT_UDT, INDEX_UDT                                    :
    PSDL_ID_PKG.PSDL_ID;
  ELEMENT_IS_UDT, INDEX_IS_UDT, FOUND_TYPE                  :
    BOOLEAN;
  OPERATOR_COMPONENT                                        :
    DETERMINE_THE_ADA_TYPE_PKG.COMPONENT_TYPE := OPERATOR_COMP;



begin
  THE_TYPE_NAME := TYPE_NAME;

  -- write the name of the identifier to the file
  PUT(THE_FILE, ID.S);

  -- write the name of the identifier into the parameter list
  if PARAMETER_LIST_HEAD = null then
  -- initialize list
    PARAMETER_LIST_HEAD := new PARAMETER_LIST_NODE;
    PARAMETER_LIST_HEAD.ID := ID;
    PARAMETER_LIST_PTR := PARAMETER_LIST_HEAD;

  else
    PARAMETER_LIST_PTR.NEXT := new PARAMETER_LIST_NODE;
    PARAMETER_LIST_PTR := PARAMETER_LIST_PTR.NEXT;
    PARAMETER_LIST_PTR.ID := ID;
  end if;
  PARAMETER_COUNT := PARAMETER_COUNT + 1;
```

```
  if RECOGNIZED_ADA_TYPE(THE_TYPE_NAME.NAME) then
  -- Output parameter is a valid Ada type so go ahead and add
  -- to the file.

   THE_TYPE_NAME.NAME := A_STRINGS.LOWER_TO_UPPER(THE_TYPE_NAME.NAME);
   WRITE_PARAMETER_TO_FILE(THE_TYPE_NAME.NAME);
   PARAMETER_LIST_PTR.THE_TYPE := THE_TYPE_NAME.NAME;
   if THE_TYPE_NAME.NAME.S = ARRAY_TYPE then

         PUT(THE_FILE, "[AE");
         ARRAY_REFERENCE_IN_UDT := FALSE;
         GET_ARRAY_COMPONENTS(THE_TYPE_NAME,      ELEMENT_TYPE,      ELEMENT_IS_UDT,
ELEMENT_UDT,
          INDEX_TYPE, INDEX_IS_UDT, INDEX_UDT, ARRAY_REFERENCE_IN_UDT);

     -- add array components to file
         WRITE_PARAMETER_TO_FILE(ELEMENT_TYPE.NAME);
         PUT(THE_FILE, ", AI");
         WRITE_PARAMETER_TO_FILE(INDEX_TYPE.NAME);
         PUT_LINE(THE_FILE, "]");
     -- add new array element
         PARAMETER_LIST_PTR.ARRAY_ELEMENT_PTR := new PARAMETER_LIST_NODE;
         PARAMETER_LIST_PTR.ARRAY_ELEMENT_PTR.THE_TYPE := ELEMENT_TYPE.NAME;
         if ELEMENT_IS_UDT then
          PARAMETER_LIST_PTR.ARRAY_ELEMENT_PTR.UDT := ELEMENT_UDT;
          PARAMETER_LIST_PTR.ARRAY_ELEMENT_PTR.HAS_UDT := TRUE;
         end if;

     -- add new array index
         PARAMETER_LIST_PTR.ARRAY_INDEX_PTR := new PARAMETER_LIST_NODE;
         PARAMETER_LIST_PTR.ARRAY_INDEX_PTR.THE_TYPE := INDEX_TYPE.NAME;
         if INDEX_IS_UDT then
          PARAMETER_LIST_PTR.ARRAY_INDEX_PTR.UDT := INDEX_UDT;
          PARAMETER_LIST_PTR.ARRAY_INDEX_PTR.HAS_UDT := TRUE;
         end if;

    else   -- not an Array type
         NEW_LINE(THE_FILE);
    end if;

  else
  -- Maybe the parameter is defined in the generics of the
  -- Operator component.
   CHECK_GENERICS(OPERATOR_COMPONENT, THE_TYPE_NAME, GENERIC_TYPE_NAME,
    FOUND_TYPE);

   if FOUND_TYPE then
   -- The type is either not composite or even if composite,
   -- will not be further specified by its components so
   -- go ahead and encode.
         UDT_TYPE_NAME := GENERIC_TYPE_NAME;
         PARAMETER_LIST_PTR.GENERIC_ID := THE_TYPE_NAME.NAME;
         PARAMETER_LIST_PTR.THE_TYPE := GENERIC_TYPE_NAME.NAME;
```

```
        PARAMETER_LIST_PTR.DEFINED_BY_GENERIC_TYPE := TRUE;
        IS_GENERIC := TRUE;
        ARRAY_IS_GENERIC := TRUE;

    else  -- The parameter must be a user defined type so
          -- get its type.

        GET_USER_DEFINED_TYPE(THE_TYPE_NAME, UDT_TYPE_NAME);
        PARAMETER_LIST_PTR.THE_TYPE := UDT_TYPE_NAME.NAME;
     -- record the UDT name
        PARAMETER_LIST_PTR.UDT := THE_TYPE_NAME.NAME;
        PARAMETER_LIST_PTR.HAS_UDT := TRUE;

    end if;

    UDT_TYPE_NAME.NAME := A_STRINGS.LOWER_TO_UPPER(UDT_TYPE_NAME.NAME);
    WRITE_PARAMETER_TO_FILE(UDT_TYPE_NAME.NAME);
    if UDT_TYPE_NAME.NAME.S = ARRAY_TYPE then

        PUT(THE_FILE, "[AE");
        ARRAY_REFERENCE_IN_UDT := TRUE;
        GET_ARRAY_COMPONENTS(UDT_TYPE_NAME,    ELEMENT_TYPE,    ELEMENT_IS_UDT,
ELEMENT_UDT,
         INDEX_TYPE, INDEX_IS_UDT, INDEX_UDT, ARRAY_REFERENCE_IN_UDT);

     -- add array components to file
        WRITE_PARAMETER_TO_FILE(ELEMENT_TYPE.NAME);
        PUT(THE_FILE, ", AI");
        WRITE_PARAMETER_TO_FILE(INDEX_TYPE.NAME);
        PUT_LINE(THE_FILE, "]");
     -- add new array element
        PARAMETER_LIST_PTR.ARRAY_ELEMENT_PTR := new PARAMETER_LIST_NODE;
        PARAMETER_LIST_PTR.ARRAY_ELEMENT_PTR.THE_TYPE := ELEMENT_TYPE.NAME;
        if ELEMENT_IS_UDT then
          PARAMETER_LIST_PTR.ARRAY_ELEMENT_PTR.UDT := ELEMENT_UDT;
          PARAMETER_LIST_PTR.ARRAY_ELEMENT_PTR.HAS_UDT := TRUE;
        end if;

     -- add new array index
        PARAMETER_LIST_PTR.ARRAY_INDEX_PTR := new PARAMETER_LIST_NODE;
        PARAMETER_LIST_PTR.ARRAY_INDEX_PTR.THE_TYPE := INDEX_TYPE.NAME;
        if INDEX_IS_UDT then
          PARAMETER_LIST_PTR.ARRAY_INDEX_PTR.UDT := INDEX_UDT;
          PARAMETER_LIST_PTR.ARRAY_INDEX_PTR.HAS_UDT := TRUE;
        end if;

     -- if array is generic then components must reflect that
        if ARRAY_IS_GENERIC then

          ARRAY_IS_GENERIC := FALSE;
          PARAMETER_LIST_PTR.ARRAY_ELEMENT_PTR.DEFINED_BY_GENERIC_TYPE := TRUE;
          PARAMETER_LIST_PTR.ARRAY_INDEX_PTR.DEFINED_BY_GENERIC_TYPE := TRUE;

        end if;
```

```
   else   -- The type is either not ARRAY_TYPE or even if some other
          -- composite, will not be further specified by its components.
   -- already added to file so go to next line in file
          NEW_LINE(THE_FILE);
   end if;
  end if;


exception

  when others =>
   PUT_LINE("*** ERROR: OCCURED IN procedure GET_OUT_PARAMETER. ");
   NEW_LINE;
   raise ;

end GET_OUT_PARAMETER;



-- procedure passed to generic scan procedure in generic map package
-- extracts the generic parameters from the Type Declaration
-- map.  NOTE that this procedure will be run once for each generic
-- parameter as the entire set of generic parameters is iterated through.
-- The file name is predetermined because only a software base component
-- can have generic parameters.
procedure GET_GEN_PARAMETER(ID       : in PSDL_ID_PKG.PSDL_ID;
                            TYPE_NAME : in PSDL_CONCRETE_TYPE_PKG.TYPE_NAME) is


-- User Defined Type (UDT) type name
 UDT_TYPE_NAME, GENERIC_TYPE_NAME, ELEMENT_TYPE, INDEX_TYPE, THE_TYPE_NAME :
  PSDL_CONCRETE_TYPE_PKG.TYPE_NAME;
 ELEMENT_UDT, INDEX_UDT                                    :
  PSDL_ID_PKG.PSDL_ID;
 ELEMENT_IS_UDT, INDEX_IS_UDT, FOUND_TYPE                  :
  BOOLEAN;
 OPERATOR_COMPONENT                                        :
  DETERMINE_THE_ADA_TYPE_PKG.COMPONENT_TYPE := OPERATOR_COMP;

begin
 THE_TYPE_NAME := TYPE_NAME;


 -- write the name of the identifier to the file
 PUT(THE_FILE, ID.S);

 -- write the name of the identifier into the parameter list
 if PARAMETER_LIST_HEAD = null then
 -- initialize list
  PARAMETER_LIST_HEAD := new PARAMETER_LIST_NODE;
  PARAMETER_LIST_HEAD.ID := ID;
```

```
   PARAMETER_LIST_PTR := PARAMETER_LIST_HEAD;

  else
   PARAMETER_LIST_PTR.NEXT := new PARAMETER_LIST_NODE;
   PARAMETER_LIST_PTR := PARAMETER_LIST_PTR.NEXT;
   PARAMETER_LIST_PTR.ID := ID;
  end if;
  PARAMETER_COUNT := PARAMETER_COUNT + 1;

  if RECOGNIZED_ADA_TYPE(THE_TYPE_NAME.NAME) then
  -- generic parameter is a valid Ada type so go ahead and append
  -- the current file with this Ada type.
   THE_TYPE_NAME.NAME := A_STRINGS.LOWER_TO_UPPER(THE_TYPE_NAME.NAME);
   WRITE_PARAMETER_TO_FILE(THE_TYPE_NAME.NAME);
   PARAMETER_LIST_PTR.THE_TYPE := THE_TYPE_NAME.NAME;
   if THE_TYPE_NAME.NAME.S = ARRAY_TYPE then
        ARRAY_REFERENCE_IN_UDT := FALSE;
    -- Note: even though Get_Array_Components will not restrict
    -- its search for the types of the array components to what
    -- is defined in the generics portion of the Operator component,
    -- that is OK. The reason is that by virtue of reaching this
    -- point (i.e. a software base component has been matched and
    -- selected), we are sure that the software base component
    -- is correctly formulated with regards to it PSDL specification.
    -- Otherwise, it could not have been stored in the software base.
        GET_ARRAY_COMPONENTS(THE_TYPE_NAME,      ELEMENT_TYPE,      ELEMENT_IS_UDT,
ELEMENT_UDT,
         INDEX_TYPE, INDEX_IS_UDT, INDEX_UDT, ARRAY_REFERENCE_IN_UDT);

    -- add array components to file
        PUT(THE_FILE, "[AE");
        WRITE_PARAMETER_TO_FILE(ELEMENT_TYPE.NAME);
        PUT(THE_FILE, ", AI");
        WRITE_PARAMETER_TO_FILE(INDEX_TYPE.NAME);
        PUT_LINE(THE_FILE, "]");
    -- add new array element
        PARAMETER_LIST_PTR.ARRAY_ELEMENT_PTR := new PARAMETER_LIST_NODE;
        PARAMETER_LIST_PTR.ARRAY_ELEMENT_PTR.THE_TYPE := ELEMENT_TYPE.NAME;
        if ELEMENT_IS_UDT then
         PARAMETER_LIST_PTR.ARRAY_ELEMENT_PTR.UDT := ELEMENT_UDT;
         PARAMETER_LIST_PTR.ARRAY_ELEMENT_PTR.HAS_UDT := TRUE;
        end if;

    -- add new array index
        PARAMETER_LIST_PTR.ARRAY_INDEX_PTR := new PARAMETER_LIST_NODE;
        PARAMETER_LIST_PTR.ARRAY_INDEX_PTR.THE_TYPE := INDEX_TYPE.NAME;
        if INDEX_IS_UDT then
         PARAMETER_LIST_PTR.ARRAY_INDEX_PTR.UDT := INDEX_UDT;
         PARAMETER_LIST_PTR.ARRAY_INDEX_PTR.HAS_UDT := TRUE;
        end if;

   else   -- not an Array type
```

```
            NEW_LINE(THE_FILE);
        end if;

      else
        raise INVALID_ADA_TYPE;
      end if;

    exception

      when INVALID_ADA_TYPE =>
        PUT_LINE("*** ERROR:  INVALID ADA TYPE DISCOVERED FOR -");
        PUT_LINE("          " & THE_TYPE_NAME.NAME.S);
        PUT_LINE("A Type MUST BE DEFINED AS AN ADA TYPE.  IT CAN REFERENCE");
        PUT_LINE("INFORMATION IN ITS GENERIC PARAMETERS, BUT CANNOT");
        PUT_LINE("REFERENCE DEFINITION INFORMATION OUTSIDE ITS OWN");
        PUT_LINE("SPECIFICATION.  A Generic PARAMETER MUST BE FULLY ");
        PUT_LINE("DEFINED AS AN ADA TYPE.  IT CAN'T EVEN REFERENCE OTHER ");
        PUT_LINE("DEFINITIONS WITHIN ITS GENERICS.");
        NEW_LINE;
        raise ;

      when others =>
        PUT_LINE("*** ERROR: OCCURED IN procedure GET_GEN_PARAMETER. ");
        NEW_LINE;
        raise ;

    end GET_GEN_PARAMETER;


end ITERATE2_THROUGH_OPERATOR_PARAMETERS_PKG;



*******************************************************************************
*******************************************************************************
*******************************************************************************



-- Filename    / Store_ADT_Signatures_In_File.a
-- Date        / 30 August 93
-- Author      / Scott Dolgoff
-- System      / Sun Sparcstation
-- Compiler    / Verdix Ada
-- Description / This program receives the encoded signatures and
--             / then stores them in the file "Signature.dat" for
--             / subsequent use by a C++ program.


with TEXT_IO, ENCODE_ADT_SIGNATURE_PKG, ADD_ADA_TYPE_TO_SIGNATURE_PKG;
use  TEXT_IO, ENCODE_ADT_SIGNATURE_PKG, ADD_ADA_TYPE_TO_SIGNATURE_PKG;

procedure STORE_ADT_SIGNATURES_IN_FILE is
```

```
 package INTEGER_INOUT is new INTEGER_IO(INTEGER);
 use  INTEGER_INOUT;

 IN_SIG, OUT_SIG : SIGNATURE;
 SIGNATURE_FILE_NAME : constant STRING := "Signature.dat";
 SIGNATURE_FILE : TEXT_IO.FILE_TYPE;


begin

-- Get the encoded ADT signatures.
 ENCODE_ADT_SIGNATURE(IN_SIG, OUT_SIG);

 -- Create the file to store the ADT signatures in.
 TEXT_IO.CREATE(SIGNATURE_FILE, MODE => OUT_FILE, NAME => SIGNATURE_FILE_NAME);

 -- Store ADT aggregate input signature.
 for REGION in ALL_REGIONS'FIRST .. ALL_REGIONS'LAST loop

 -- Note:  Setting WIDTH to zero is critical because
 --        it left justifies the numbers.  This makes
 --        it easy on the C++ program that has to read
 --        the numbers from a file as characters and
 --        then convert them to integers.
  PUT(SIGNATURE_FILE, IN_SIG(REGION), WIDTH => 0);
  NEW_LINE(SIGNATURE_FILE);

 end loop;


 -- Store ADT aggregate output signature.
 for REGION in ALL_REGIONS'FIRST .. ALL_REGIONS'LAST loop

  PUT(SIGNATURE_FILE, OUT_SIG(REGION), WIDTH => 0);
  NEW_LINE(SIGNATURE_FILE);

 end loop;

 TEXT_IO.CLOSE(SIGNATURE_FILE);

end STORE_ADT_SIGNATURES_IN_FILE;



*******************************************************************************
*******************************************************************************
*******************************************************************************



-- Filename    / Store_Signatures_In_File.a
-- Date        / 29 August 93
```

```
-- Author      / Scott Dolgoff
-- System      / Sun SPARCstation
-- Compiler    / Verdix Ada
-- Description / This program receives the encoded signatures and
--             / then stores them in the file "Signature.dat" for
--             / subsequent use by a C++ program.


with TEXT_IO, ENCODE_OPERATOR_SIGNATURE_PKG, ADD_ADA_TYPE_TO_SIGNATURE_PKG;
use  TEXT_IO, ENCODE_OPERATOR_SIGNATURE_PKG, ADD_ADA_TYPE_TO_SIGNATURE_PKG;

procedure STORE_SIGNATURES_IN_FILE is

 package INTEGER_INOUT is new INTEGER_IO(INTEGER);
 use  INTEGER_INOUT;

 IN_SIG, OUT_SIG : SIGNATURE;
 SIGNATURE_FILE_NAME : constant STRING := "Signature.dat";
 SIGNATURE_FILE : TEXT_IO.FILE_TYPE;


begin

-- Get the encoded Operator signatures.
 ENCODE_OPERATOR_SIGNATURE(IN_SIG, OUT_SIG);

 -- Create the file to store the Operator signatures in.
 TEXT_IO.CREATE(SIGNATURE_FILE, MODE => OUT_FILE, NAME => SIGNATURE_FILE_NAME);

 -- Store Operator input signature.
 for REGION in ALL_REGIONS'FIRST .. ALL_REGIONS'LAST loop

 -- Note:  Setting WIDTH to zero is critical because
 --        it left justifies the numbers.  This makes
 --        it easy on the C++ program that has to read
 --        the numbers from a file as characters and
 --        then convert them to integers.
  PUT(SIGNATURE_FILE, IN_SIG(REGION), WIDTH => 0);
  NEW_LINE(SIGNATURE_FILE);

 end loop;


 -- Store Operator output signature.
 for REGION in ALL_REGIONS'FIRST .. ALL_REGIONS'LAST loop

  PUT(SIGNATURE_FILE, OUT_SIG(REGION), WIDTH => 0);
  NEW_LINE(SIGNATURE_FILE);

 end loop;

 TEXT_IO.CLOSE(SIGNATURE_FILE);
```

```
end STORE_SIGNATURES_IN_FILE;
```

APPENDIX E - C++ SOURCE CODE

The code listed below is in alphabetical order and contains C++ routines for signature matching.

```
/*
 Filename   | ADT_Signature.h
 Date       | 31 August 93
 Author     | Scott Dolgoff
 System     | Sun SPARCstation
 Compiler   | Sun C++ 2.0
 Description | This is a header file for the ADT Signature class.
             | It contains the various functions required to
             | determine whether a query and software base
             | component match.
*/

#ifndef __ADT_SIGNATURE_H
#define __ADT_SIGNATURE_H    // prevent multiple #includes

#include "sball.hxx"
#include "sbextern.h"
#include "Signature.h"
#include <Primitives.h>


struct SIGNATURES
   {
    int INPUT_SIG[ALL_REGIONS];
    int OUTPUT_SIG[ALL_REGIONS];
   };

struct MATCH_LIST
  {
    int SB_OPERATOR;
    struct MATCH_LIST *NEXT;
  };

struct PARAMETERS
 {
    int NUM_OF_INPUT_PARAMETERS;
    int NUM_OF_OUTPUT_PARAMETERS;
 };

typedef struct MATCH_LIST ELEMENT;
typedef ELEMENT *LINK;


class ADT_SIGNATURE {
```

```
private:

  Boolean MATCH_FOUND;
  int MAX_LEVELS;

  int  GET_NUMBER(FILE *ifp);

  void TREE_TRAVERSAL (LINK QUERY_OPERATORS[],
                       Boolean ONE_TO_ONE_MAP[],
                       int LEVEL);

public:

  // Retrieves ADT operator signatures stored in text files.
  void GET_SIGNATURES_FROM_FILE (int IN_SIG[], int OUT_SIG[], char*);

  // Load ADT operator input and output signatures into an array whose cells
  // represent the ADT operators.  Also store the number of input and output
  // parameters each operator has in an array.
  void LOAD_ADT_OPERATOR_SIGNATURES (SIGNATURES THE_SIGNATURE[],
                                     PARAMETERS OPERATOR_IO[],
                                     SB_ADT_COMPONENT*);


   // Create an array of linked lists.  Each array cell corresponds to
   // a query component ADT operator.  Each linked list is made up of
   // all software base component ADT operators that match the query
   // component ADT operator.
  void BUILD_OPERATOR_MATCH_LIST (SIGNATURES QUERY_SIGNATURE[],
                                  LINK QUERY_OPERATORS[],
                                  PARAMETERS QUERY_OPERATOR_IO[],
                                  SB_ADT_COMPONENT *query_component,
                                  SB_ADT_COMPONENT *next_component);


  // This function attempts to find a valid mapping between query component
  // ADT operators and the software base component ADT operators.  A 1 is
  // returned if a valid match is found, otherwise a 0 is returned.
  int ADT_MATCH (LINK QUERY_OPERATORS[],
                 SB_ADT_COMPONENT *query_component,
                 SB_ADT_COMPONENT *next_component);

};

#endif   // __ADT_SIGNATURE_H


*****************************************************************************
*****************************************************************************
*****************************************************************************



/*
```

```
Filename   | ADT_Signature.cxx
Date       | 31 August 93
Author     | Scott Dolgoff
System     | Sun SPARCstation
Compiler   | Sun C++ 2.0
Description | This is the main module for the ADT_Signature class.
              | It contains the various functions required to
              | determine whether a query and software base
              | component match.

              | TECHNICAL NOTE:  In C++ arrays are treated as pointers to
              | the head of the array.  Therefore, when arrays are passed
              | as function arguments, it is automatically a call by
              | reference.  The only way to protect the actual argument
              | from being changed is to set a local array variable in the
              | function equal to the array that is passed by reference and
              | then work exclusively with the local array.
*/


// #include <stream.hxx>
// #include <stdlib.h>
// #include <Primitives.h>
#include "ADT_Signature.h"


 /*  This function reads a text file with a Component's ADT operator
    signatures that was created by the Ada program that encodes
    signatures.  A character-to-number conversion is necessary to
    load the actual numeric values for all the signature regions into
    the corresponding C++ signature arrays.  */
 int ADT_SIGNATURE::GET_NUMBER(FILE *ifp)
 {
  int NUM, C;

  do {
    NUM = 0;
    while ( ((C = getc(ifp)) != '\n') && (C != EOF) )
          NUM =  (10 * NUM) + (C - '0');

    if (C == '\n')
          return NUM;
    else
          { printf("*** Error:  Occurred while reading signatures from\n");
            printf("          ADT operator text file.\n");
          }
  } while (1);

 }




// Retrieves ADT operator signatures stored in text files.
void ADT_SIGNATURE::GET_SIGNATURES_FROM_FILE
```

```
            (int IN_SIG[], int OUT_SIG[], char *id)
{

  FILE *ifp;
  int INDEX;

  // load input and output signatures using the operator name
  // as the prefix and ".txt" as the suffix
  char *file_suffix = ".txt";
  char *file_name;

  file_name = new char[strlen(id) + strlen(file_suffix) + 1];
  strcpy(file_name, id);
  strcat(file_name, file_suffix);
  ifp = fopen(file_name, "r");

  // get input signature
  for (INDEX = 0; INDEX < ALL_REGIONS; ++INDEX)
        IN_SIG[INDEX] = GET_NUMBER(ifp);
  // get output signature
  for (INDEX = 0; INDEX < ALL_REGIONS; ++INDEX)
        OUT_SIG[INDEX] = GET_NUMBER(ifp);

  fclose(ifp);

}




// Load ADT operator input and output signatures into an array whose cells
// represent the ADT operators. Also store the number of input and output
// parameters each operator has in an array.
void ADT_SIGNATURE::LOAD_ADT_OPERATOR_SIGNATURES (SIGNATURES THE_SIGNATURE[],
                                                  PARAMETERS OPERATOR_IO[],
                                                SB_ADT_COMPONENT *THE_COMPONENT)
{
  int REGION;
  int OPERATOR_COUNT = 0;

  DictionaryIterator next_operator =
                                        THE_COMPONENT->adt_operator_iterator();
  while ( next_operator.moreData() )
        {
         SB_ADT_OPERATOR *THE_ADT_OPERATOR =
                              (SB_ADT_OPERATOR *) (Entity *) next_operator();

         // Load the number of input and output parameters the operator
         // has.
         OPERATOR_IO[OPERATOR_COUNT].NUM_OF_INPUT_PARAMETERS =
         THE_ADT_OPERATOR->num_inputs();


         OPERATOR_IO[OPERATOR_COUNT].NUM_OF_OUTPUT_PARAMETERS =
```

```
            THE_ADT_OPERATOR->num_outputs();


          // Load the operator input and output signatures.
          Array *IN_SIGNATURE = new Array(OC_integer, ALL_REGIONS, 1);
          IN_SIGNATURE = (Array *) (Entity *)
                                    THE_ADT_OPERATOR->get_input_signature();
          // Load the Array object into a C++ array.
          for (REGION=1; REGION <=ALL_REGIONS; ++REGION)
            {
             // Note:  C++ array goes from 0 .. ALL_REGIONS -1
             THE_SIGNATURE[OPERATOR_COUNT].INPUT_SIG[REGION - 1] =
                    * ( (Integer *) (Entity *) (*IN_SIGNATURE) [REGION]);
            }

          Array *OUT_SIGNATURE = new Array(OC_integer, ALL_REGIONS, 1);
          OUT_SIGNATURE =  (Array *) (Entity *)
                                    THE_ADT_OPERATOR->get_output_signature();

          // Load the Array object into a C++ array.
          for (REGION=1; REGION <=ALL_REGIONS; ++REGION)
            {
             // Note:  C++ array goes from 0 .. ALL_REGIONS -1
             THE_SIGNATURE[OPERATOR_COUNT].OUTPUT_SIG[REGION - 1] =
                    *( (Integer *) (Entity *) (*OUT_SIGNATURE) [REGION]);

            }


          OPERATOR_COUNT = OPERATOR_COUNT + 1;
        }

}



// Create an array of linked lists.  Each array cell corresponds to
// a query component ADT operator.  Each linked list is made up of
// all software base component ADT operators that match the query
// component ADT operator.
void ADT_SIGNATURE::BUILD_OPERATOR_MATCH_LIST(SIGNATURES QUERY_SIGNATURE[],
                                              LINK QUERY_OPERATORS[],
                                              PARAMETERS QUERY_OPERATOR_IO[],
                                             SB_ADT_COMPONENT *query_component,
                                             SB_ADT_COMPONENT *next_component)
{
 // class SIGNATURE comes from Signature.h
 SIGNATURE SIG;

 // Create an array that holds the input and output signature of
 // each software base component ADT operator.
 // We don't know the size in advance, so it must be a dynamic array.
 SIGNATURES *SB_SIGNATURE;
 SB_SIGNATURE = new
          SIGNATURES[next_component->num_adt_operators()];
```

```
// Create an array that holds the number of input and output
// parameters of each software base component ADT operator.
// We don't know the size in advance, so it must be a dynamic array.
PARAMETERS *SB_OPERATOR_IO;
SB_OPERATOR_IO = new
        PARAMETERS[next_component->num_adt_operators()];

// load software base component ADT operator signatures
LOAD_ADT_OPERATOR_SIGNATURES(SB_SIGNATURE, SB_OPERATOR_IO, next_component);


Boolean COMPONENTS_CANNOT_MATCH = FALSE;
int INDEX = 0;
while ( (INDEX < query_component->num_adt_operators()) &&
        (! COMPONENTS_CANNOT_MATCH) )
 {
  LINK PTR;
  int MATCH;

  // go through all software base component ADT operators and store
  // each one that matches into the query component ADT operator
  // linked list. Note:  if none match, then the two components
  // do not match so no further processing is necessary.
  int SB_INDEX = 0;
  Boolean QUERY_ADT_OPERATOR_HAS_MATCH = FALSE;
  while ( (SB_INDEX < next_component->num_adt_operators()) )
        {
         // First, ensure the number of input parameters for the query
         // component ADT operator is the same as the number of input
         // parameters for the software base component ADT operator.
         if (QUERY_OPERATOR_IO[INDEX].NUM_OF_INPUT_PARAMETERS ==
           SB_OPERATOR_IO[SB_INDEX].NUM_OF_INPUT_PARAMETERS)
          {
           MATCH = SIG.MATCH_INPUT_SIGNATURES(
                                    QUERY_SIGNATURE[INDEX].INPUT_SIG,
                                    SB_SIGNATURE[SB_INDEX].INPUT_SIG);

           if (MATCH == 1)
            {

            // Now check for a false match.
            MATCH = SIG.CHECK_FALSE_MATCH(
                                        QUERY_SIGNATURE[INDEX].INPUT_SIG,
                                        SB_SIGNATURE[SB_INDEX].INPUT_SIG);

            if (MATCH >= 0) // Not a false match
                  {


                  // Ensure the number of output parameters for the query
                  // component ADT operator is less than or equal the number
                  // of output parameters for the software base component
                  // ADT operator.
                  if (QUERY_OPERATOR_IO[INDEX].NUM_OF_OUTPUT_PARAMETERS <=
                    SB_OPERATOR_IO[SB_INDEX].NUM_OF_OUTPUT_PARAMETERS)
                   {
```

```
                MATCH = SIG.MATCH_OUTPUT_SIGNATURES(
                                QUERY_SIGNATURE[INDEX].OUTPUT_SIG,
                                SB_SIGNATURE[SB_INDEX].OUTPUT_SIG);

                if (MATCH == 1)
                 {
                 QUERY_ADT_OPERATOR_HAS_MATCH = TRUE;

          // add matched software base component ADT operator to linked list
                  if (QUERY_OPERATORS[INDEX] == NULL)
                   {
                        // initialize list
                        PTR = new ELEMENT;
                        PTR->SB_OPERATOR = SB_INDEX;
                        PTR->NEXT = NULL;
                        QUERY_OPERATORS[INDEX] = PTR;
                   }
                  else
                   {
                        // append to list
                        PTR->NEXT = new ELEMENT;
                        PTR = PTR->NEXT;
                        PTR->SB_OPERATOR = SB_INDEX;
                        PTR->NEXT = NULL;
                   }

                  };
                 };
                };
            };
           }; // if num input params

          SB_INDEX = SB_INDEX + 1;
         }


   if (! QUERY_ADT_OPERATOR_HAS_MATCH)
         // query component operator does not match any of the software base
         // component operators
         COMPONENTS_CANNOT_MATCH = TRUE;


   INDEX = INDEX + 1;
  }

}



// The "tree traversal" treats the linked lists of matched software
// base component ADT operators as a tree rooted at the first
// query component operator. Conceptually, the tree is a combinatoric
// explosion of all possible mappings of one query ADT operator to
// a valid software base ADT operator (so that no software base component
// ADT operator is used more than once). The traversal does not explore
```

```
// all paths. If a node is reached that selects a software base component
// ADT operator that was already selected, no lower level nodes on that
// path need to be visited. Also, the tree is never physically built,
// just logically traversed. This helps alleviate potential memory
// problems. Backtracking occurs when a query ADT operator (tree level)
// is reached that cannot find an unused software base ADT operator from
// its linked list of valid mappings. Finally, it was necessary to
// make the array QUERY_OPERATORS[] local at each level of recursion in
// order to provide "memory" for the backtracking. In the worst case
// it is estimated that the number of paths to be checked is N factorial
// where N is the total number of software base ADT operators.
void ADT_SIGNATURE::TREE_TRAVERSAL (LINK QUERY_OPERATORS[],
                                    Boolean ONE_TO_ONE_MAP[],
                                    int LEVEL)
 {
  int INDEX;
  LINK *Lv_QUERY_OPERATORS;

  // We don't know the size in advance, so it must be a dynamic array.
  Lv_QUERY_OPERATORS = new LINK[MAX_LEVELS];

  for (INDEX = 0; INDEX < MAX_LEVELS; ++INDEX)
       Lv_QUERY_OPERATORS[INDEX] = QUERY_OPERATORS[INDEX];

  while ( (Lv_QUERY_OPERATORS[LEVEL] != NULL) && (! MATCH_FOUND) )
       {
        if ( ! ONE_TO_ONE_MAP[Lv_QUERY_OPERATORS[LEVEL]->SB_OPERATOR] )
         // query operator maps to sb operator that has not been
         // claimed (mapped to) yet.
         {
          ONE_TO_ONE_MAP[(Lv_QUERY_OPERATORS[LEVEL]->SB_OPERATOR)] = TRUE;
          if ( (LEVEL + 1) < MAX_LEVELS )
           {
                TREE_TRAVERSAL (Lv_QUERY_OPERATORS, ONE_TO_ONE_MAP, LEVEL + 1);
                // backtracked to this point. now will try next sb operator
                // in the list so must reset current sb operator mapped to
                // by this query operator to FALSE
                ONE_TO_ONE_MAP[(Lv_QUERY_OPERATORS[LEVEL]->SB_OPERATOR)] =
                                                                    FALSE;
                Lv_QUERY_OPERATORS[LEVEL] = Lv_QUERY_OPERATORS[LEVEL]->NEXT;
           }
          else
                MATCH_FOUND = TRUE;
         }
        else
         // try the next query operator in the list
         Lv_QUERY_OPERATORS[LEVEL] = Lv_QUERY_OPERATORS[LEVEL]->NEXT;

       }
 }



// This function attempts to find a valid mapping between query component
// ADT operators and the software base component ADT operators. A 1 is
```

```
// returned if a valid match is found, otherwise a 0 is returned.
int ADT_SIGNATURE::ADT_MATCH( LINK QUERY_OPERATORS[],
                              SB_ADT_COMPONENT *query_component,
                              SB_ADT_COMPONENT *next_component)
 {
 int LEVEL = 0;
 int MATCH = 0;
 MATCH_FOUND = FALSE;
 MAX_LEVELS = query_component->num_adt_operators();

 // Create Boolean array corresponding to the software base component
 // set of ADT operators. Initially all ADT operators are false. As
 // a query component ADT operator is matched to a software base
 // component ADT operator, the corresponding Boolean array cell is
 // set to TRUE indicating that software base component ADT operator
 // cannot be selected by another query component ADT operator.
 // We don't know the size in advance, so it must be a dynamic array.
 Boolean *ONE_TO_ONE_MAP;
 ONE_TO_ONE_MAP = new Boolean[next_component->num_adt_operators()];
 int INDEX;
 for (INDEX = 0; INDEX < next_component->num_adt_operators(); ++INDEX)
  ONE_TO_ONE_MAP[INDEX] = FALSE;


 TREE_TRAVERSAL (QUERY_OPERATORS, ONE_TO_ONE_MAP, LEVEL);

 if (MATCH_FOUND)
  MATCH = 1;

 return MATCH;

 }
```

****
****
****

```
/*
Filename   | Signature.h
Date       | 30 August 93
Author     | Scott Dolgoff
System     | Sun SPARCstation
Compiler   | Sun C++ 2.0
Description | This is a header file for the Signature class.
            | It contains the various functions required to
            | determine whether a query and software base
            | component signature match. It also contains
            | functions that calculate the signature closeness
            | degree between the query and software base
            | component signatures.
*/
```

```
#ifndef __SIGNATURE_H
#define __SIGNATURE_H    // prevent multiple #includes

#include <stdio.h>

// ADDED
#define ALL_REGIONS 24
#define INPUT_REGIONS 18
#define OUTPUT_REGIONS 17

// ADDED
// set value for a null pointer
#define NULL 0


class SIGNATURE {

 private:

  int  BEST_CHILD(int REGION_NUMBER, int SB_SIG[]);
  int  NUM_OF_TYPES_IN_REGION(int REGION_NUMBER, int SB_SIG[]);
  int  SUM_OF_CHILDREN_TYPES(int REGION_NUMBER, int SB_SIG[]);
  int  PARENT (int REGION_NUMBER);
  int  GET_NUMBER(FILE *ifp);
  void ADD_INSTANTIATION_TO_SIGNATURE(int REGION_NUMBER,
                                      int SB_SIG[]);

  void REMOVE_A_TYPE_FROM_ALL_ANCESTORS(int REGION_NUMBER,
                                        int Q_SIG[],
                                        int SB_SIG[]);
  void REMOVE_A_TYPE_FROM_ALL_DESCENDANTS(int REGION_NUMBER,
                                        int SB_SIG[]);

 public:

  /* This function reads a text file with the Component signatures
     that was created by the Ada program responsible for encoding
     signatures.  A character-to-number conversion is necessary to
     load the actual numeric values for all the signature regions into
     the corresponding C++ signature arrays.  */
  void GET_SIGNATURES (int IN_SIG[], int OUT_SIG[]);

  /* This function determines whether a query component input
     signature and a software base component input signature
     match each other.  The function returns a 1 if a match was
     made and a 0 if the two signatures do not match.  */
  int  MATCH_INPUT_SIGNATURES (int Q_SIG[], int SB_SIG[]);

  /* This function determines whether a query component output
     signature and a software base component output signature
     match each other.  The function returns a 1 if a match was
     made and a 0 if the two signatures do not match.  */
  int  MATCH_OUTPUT_SIGNATURES (int Q_SIG[], int SB_SIG[]);

  /* This function is only necessary for input signatures.  It also
```

```
    calculates input signature closeness degree as a by product.  It
    is necessary to calculate the closeness degree in this manner
    when the number of input parameters in the query component are not
    equal to the number of input parameters in the software base component.
    This function returns the value of the closeness degree if a valid
    match was found, or a -1 if a False Match was found. */
  int  CHECK_FALSE_MATCH (int Q_SIG[], int SB_SIG[]);

  /*  Calculates the output signature closeness degree. */
  int  CALC_OUT_CLOSE_DEGREE (int Q_SIG[], int SB_SIG[]);

};

#endif   // __SIGNATURE_H
```

****
****
****

```
/*
 Filename   | Signature.cxx
 Date       | 30 August 93
 Author     | Scott Dolgoff
 System     | Sun SPARCstation
 Compiler   | Sun C++ 3.0
 Description | This is the main module for the Signature class.
             | It contains the various functions required to
             | determine whether a query and software base
             | component signature match.  It also contains
             | functions that calculate the signature closeness
             | degree between the query and software base
             | component signatures. Note that the terms "nodes"
             | and "regions" are used interchangeably in the
             | documentation below.

             | TECHNICAL NOTE:  In C++ arrays are treated as pointers to
             | the head of the array.  Therefore, when arrays are passed
             | as function arguments, it is automatically a call by
             | reference.  The only way to protect the actual argument
             | from being changed is to set a local array variable in the
             | function equal to the array that is passed by reference and
             | then work exclusively with the local array.
*/


#include <stream.hxx>
#include "Signature.h"

/* Define the region numbers for the Ada types.  Note that while
   all Ada programs refer to the regions as 1..24, C++ arrays
```

```
  start at 0 and therefore we need to decrement all regions by
  1 so we now have a set of regions from 0..23 that map to
  the original regions 1..24.  */
#define GENERIC_PRIVATE       17
#define GENERIC_DISCRETE      18
#define GENERIC_RANGE         19
#define GENERIC_ARRAY         20
#define GENERIC_DIGITS        21
#define GENERIC_DELTA         22
#define GENERIC_ACCESS        23

#define   PRIVATE           16
#define    DISCRETE          7
#define     INTEGER          5
#define       RANGE          4
#define       NATURAL        1
#define         POSITIVE     0
#define     ENUMERATION       6
#define       BOOLEAN         2
#define       CHARACTER       3
#define    DIGITS           15
#define     FLOAT           14
#define    DELTA            13
#define     FIXED           12
#define    ARRAY            11
#define     STRING          10
#define    ACCESS            9
#define    RECORD            8



/*  This function reads a text file with the Component signatures
    that was created by the Ada program responsible for encoding
    signatures.  A character-to-number conversion is necessary to
    load the actual numeric values for all the signature regions into
    the corresponding C++ signature arrays.  */
int SIGNATURE::GET_NUMBER(FILE *ifp)
{
  int NUM, C;

  do {
    NUM = 0;
    while ( ((C = getc(ifp)) != '\n') && (C != EOF) )
          NUM =  (10 * NUM) + (C - '0');

    if (C == '\n')
          return NUM;
    else
          { printf("*** Error:  Occurred while reading signatures from\n");
            printf("          Signature.dat text file.\n");
          }
  } while (1);

}
```

```
/*  This function selects the child that will provide the shortest
    downward path when trying to select the closest matching
    software base component output parameter. It returns the
    Region value of the child, or -1 if the current node is a leaf
    node and therefore has no children.  */
int  SIGNATURE::BEST_CHILD(int REGION_NUMBER, int SB_SIG[])
{
 const int NO_CHILDREN = -1;

 switch (REGION_NUMBER)
   {
        case POSITIVE:
                          return NO_CHILDREN;
                          break;

        case NATURAL:
                          return POSITIVE;
                          break;

        case RANGE:
                          return NO_CHILDREN;
                          break;

        case INTEGER:
                          if (SB_SIG[RANGE] > 0)
                           return RANGE;
                          else
                           return NATURAL;
                          break;

        case BOOLEAN:
                          return NO_CHILDREN;
                          break;

        case CHARACTER:
                          return NO_CHILDREN;
                          break;


        case ENUMERATION:
                          if (SB_SIG[CHARACTER] > 0)
                           return CHARACTER;
                          else
                           return BOOLEAN;
                          break;

        case DISCRETE:
                          if (SB_SIG[ENUMERATION] > 0)
                           return ENUMERATION;
                          else
                           return INTEGER;
                          break;

        case ACCESS:
```

```
                        return NO_CHILDREN;
                        break;

        case RECORD:
                        return NO_CHILDREN;
                        break;

        case STRING:
                        return NO_CHILDREN;
                        break;

        case ARRAY:
                        return STRING;
                        break;

        case FIXED:
                        return NO_CHILDREN;
                        break;

        case DELTA:
                        return FIXED;
                        break;

        case FLOAT:
                        return NO_CHILDREN;
                        break;

        case DIGITS:
                        return FLOAT;
                        break;

        case PRIVATE:
                        return NO_CHILDREN;
                        break;

        default:
                        printf("** Error in SIGNATURE::BEST_CHILD()\n");
                        return -2;
                        break;

   }

}



/*  This function returns the total number of type instances contained by
    all the children of a particular node.  */
int  SIGNATURE::SUM_OF_CHILDREN_TYPES(int REGION_NUMBER, int SB_SIG[])
{
 const int NO_CHILDREN = 0;

 switch (REGION_NUMBER)
   {
```

```
case POSITIVE:
                return NO_CHILDREN;
                break;

case NATURAL:
                return SB_SIG[POSITIVE];
                break;

case RANGE:
                return NO_CHILDREN;
                break;

case INTEGER:
                return (SB_SIG[RANGE] + SB_SIG[NATURAL]);
                break;

case BOOLEAN:
                return NO_CHILDREN;
                break;

case CHARACTER:
                return NO_CHILDREN;
                break;


case ENUMERATION:
                return (SB_SIG[BOOLEAN] + SB_SIG[CHARACTER]);
                break;

case DISCRETE:
                return (SB_SIG[INTEGER] + SB_SIG[ENUMERATION]);
                break;

case ACCESS:
                return NO_CHILDREN;
                break;

case RECORD:
                return NO_CHILDREN;
                break;

case STRING:
                return NO_CHILDREN;
                break;

case ARRAY:
                return SB_SIG[STRING];
                break;

case FIXED:
                return NO_CHILDREN;
                break;

case DELTA:
                return SB_SIG[FIXED];
                break;
```

```
        case FLOAT:
                        return NO_CHILDREN;
                        break;

        case DIGITS:
                        return SB_SIG[FLOAT];
                        break;

        case PRIVATE:
                        return NO_CHILDREN;
                        break;

        case GENERIC_PRIVATE:
                        return NO_CHILDREN;
                        break;

        case GENERIC_DISCRETE:
                        return NO_CHILDREN;
                        break;

        case GENERIC_RANGE:
                        return NO_CHILDREN;
                        break;

        case GENERIC_ARRAY:
                        return NO_CHILDREN;
                        break;

        case GENERIC_DIGITS:
                        return NO_CHILDREN;
                        break;

        case GENERIC_DELTA:
                        return NO_CHILDREN;
                        break;

        case GENERIC_ACCESS:
                        return NO_CHILDREN;
                        break;


        default:
                        printf("** Error in SIGNATURE::SUM_OF_CHILDREN..()\n");
                        return -2;
                        break;

    }

}
```

```
/* This function returns the parent of a specified Region. The
   Region corresponds to a subtype in the Ada subtype hierarchy. */
int  SIGNATURE::PARENT (int REGION_NUMBER)
{


 switch (REGION_NUMBER)
   {
        case POSITIVE:
                              return NATURAL;
                              break;

        case NATURAL:
                              return INTEGER;
                              break;

        case RANGE:
                              return INTEGER;
                              break;

        case INTEGER:
                              return DISCRETE;
                              break;

        case BOOLEAN:
                              return ENUMERATION;
                              break;

        case CHARACTER:
                              return ENUMERATION;
                              break;


        case ENUMERATION:
                              return DISCRETE;
                              break;

        case DISCRETE:
                              return GENERIC_PRIVATE;
                              break;

        case ACCESS:
                              return GENERIC_PRIVATE;
                              break;

        case RECORD:
                              return GENERIC_PRIVATE;
                              break;

        case PRIVATE:
                              return GENERIC_PRIVATE;
                              break;

        case STRING:
                              return ARRAY;
                              break;
```

```
        case ARRAY:
                        return GENERIC_PRIVATE;
                        break;

        case FIXED:
                        return DELTA;
                        break;

        case DELTA:
                        return GENERIC_PRIVATE;
                        break;

        case FLOAT:
                        return DIGITS;
                        break;

        case DIGITS:
                        return GENERIC_PRIVATE;
                        break;

        case GENERIC_PRIVATE:
                        return -1;  // -1 is delimeter that defines the
                        break;      // top of the subtype hierarchy

        default:
                        printf("** Error in SIGNATURE::PARENT()\n");
                        return -2;
                        break;

   }
}


/* This function adds a type instance to each of the ancestors
   of the specified region in the output type hierarchy.
   This is done when a query parameter is matched to a software
   base component generic parameter. By instantiating the
   generic, we make it easier to perform later type closeness
   calculations. */
void SIGNATURE::ADD_INSTANTIATION_TO_SIGNATURE(int REGION_NUMBER,
                                               int SB_SIG[])

{

 switch (REGION_NUMBER)
   {
        case POSITIVE:
                        SB_SIG[NATURAL] ++;
                        SB_SIG[INTEGER] ++;
                        SB_SIG[DISCRETE] ++;
                        break;

        case NATURAL:
                        SB_SIG[INTEGER] ++;
                        SB_SIG[DISCRETE] ++;
```

```
                    break;

case RANGE:
                    SB_SIG[INTEGER] ++;
                    SB_SIG[DISCRETE] ++;
                    break;

case INTEGER:
                    SB_SIG[DISCRETE] ++;
                    break;

case BOOLEAN:
                    SB_SIG[ENUMERATION] ++;
                    SB_SIG[DISCRETE] ++;
                    break;

case CHARACTER:
                    SB_SIG[ENUMERATION] ++;
                    SB_SIG[DISCRETE] ++;
                    break;

case ENUMERATION:
                    SB_SIG[DISCRETE] ++;
                    break;

case DISCRETE:
                    // top of hierarchy so no ancestors
                    break;

case ACCESS:
                    // top of hierarchy so no ancestors
                    break;

case RECORD:
                    // top of hierarchy so no ancestors
                    break;

case STRING:
                    SB_SIG[ARRAY] ++;
                    break;

case ARRAY:
                    // top of hierarchy so no ancestors
                    break;

case FIXED:
                    SB_SIG[DELTA] ++;
                    break;

case DELTA:
                    // top of hierarchy so no ancestors
                    break;

case FLOAT:
                    SB_SIG[DIGITS] ++;
                    break;
```

```
        case DIGITS:
                        // top of hierarchy so no ancestors
                        break;

        case PRIVATE:
                        // top of hierarchy so no ancestors
                        break;

        default:
         printf("** Error in SIGNATURE::ADD_INSTANTIATION_TO_SIGNATURE()\n");
         break;

   }

}




/*  This function removes a type instance from all ancestor nodes of the
    specified region in both the query and software base output
    signatures.  Each ancestor must have a type instance because output
    parameter patterns force a type instance in all ancestors of each
    output parameter type.  */
void SIGNATURE::REMOVE_A_TYPE_FROM_ALL_ANCESTORS(int REGION_NUMBER,
                                                 int Q_SIG[],
                                                 int SB_SIG[])
{

 switch (REGION_NUMBER)
   {
        case POSITIVE:
                        SB_SIG[NATURAL] --;
                        SB_SIG[INTEGER] --;
                        SB_SIG[DISCRETE] --;
                        Q_SIG[NATURAL] --;
                        Q_SIG[INTEGER] --;
                        Q_SIG[DISCRETE] --;
                        break;

        case NATURAL:
                        SB_SIG[INTEGER] --;
                        SB_SIG[DISCRETE] --;
                        Q_SIG[INTEGER] --;
                        Q_SIG[DISCRETE] --;
                        break;

        case RANGE:
                        SB_SIG[INTEGER] --;
                        SB_SIG[DISCRETE] --;
                        Q_SIG[INTEGER] --;
                        Q_SIG[DISCRETE] --;
                        break;
```

```
case INTEGER:
                SB_SIG[DISCRETE] --;
                Q_SIG[DISCRETE] --;
                break;

case BOOLEAN:
                SB_SIG[ENUMERATION] --;
                SB_SIG[DISCRETE] --;
                Q_SIG[ENUMERATION] --;
                Q_SIG[DISCRETE] --;
                break;

case CHARACTER:
                SB_SIG[ENUMERATION] --;
                SB_SIG[DISCRETE] --;
                Q_SIG[ENUMERATION] --;
                Q_SIG[DISCRETE] --;
                break;

case ENUMERATION:
                SB_SIG[DISCRETE] --;
                Q_SIG[DISCRETE] --;
                break;

case DISCRETE:
                // top of hierarchy so no ancestors
                break;

case ACCESS:
                // top of hierarchy so no ancestors
                break;

case RECORD:
                // top of hierarchy so no ancestors
                break;

case STRING:
                SB_SIG[ARRAY] --;
                Q_SIG[ARRAY] --;
                break;

case ARRAY:
                // top of hierarchy so no ancestors
                break;

case FIXED:
                SB_SIG[DELTA] --;
                Q_SIG[DELTA] --;
                break;

case DELTA:
                // top of hierarchy so no ancestors
                break;

case FLOAT:
```

```
                        SB_SIG[DIGITS] --;
                        Q_SIG[DIGITS] --;
                        break;

        case DIGITS:
                        // top of hierarchy so no ancestors
                        break;

        case PRIVATE:
                        // top of hierarchy so no ancestors
                        break;

        default:
         printf("** Error in SIGNATURE::REMOVE_A_TYPE_FROM_ALL_ANCESTORS()\n");
         break;

   }

}



/*  This function removes a type instance from all descendant nodes of the
    specified region in both the query and software base input
    signatures.  Each descendant must have a type instance because input
    parameter patterns force a type instance in all descendants of each
    input parameter type.  */
void SIGNATURE::REMOVE_A_TYPE_FROM_ALL_DESCENDANTS(int REGION_NUMBER,
                                                   int SB_SIG[])
{

 switch (REGION_NUMBER)
   {
        case POSITIVE:
                        // leaf node so no descendants
                        break;

        case NATURAL:
                        SB_SIG[POSITIVE] --;
                        break;

        case RANGE:
                        // leaf node so no descendants
                        break;

        case INTEGER:
                        SB_SIG[POSITIVE] --;
                        SB_SIG[NATURAL] --;
                        SB_SIG[RANGE] --;
                        break;

        case BOOLEAN:
                        // leaf node so no descendants
                        break;
```

```
case CHARACTER:
                    // leaf node so no descendants
                    break;

case ENUMERATION:
                    SB_SIG[BOOLEAN] --;
                    SB_SIG[CHARACTER] --;
                    break;

case DISCRETE:
                    SB_SIG[POSITIVE] --;
                    SB_SIG[NATURAL] --;
                    SB_SIG[RANGE] --;
                    SB_SIG[INTEGER] --;
                    SB_SIG[BOOLEAN] --;
                    SB_SIG[CHARACTER] --;
                    SB_SIG[ENUMERATION] --;
                    break;

case ACCESS:
                    // leaf node so no descendants;
                    break;

case RECORD:
                    // leaf node so no descendants;
                    break;

case PRIVATE:
                    // leaf node so no descendants;
                    break;

case STRING:
                    // leaf node so no descendants;
                    break;

case ARRAY:
                    SB_SIG[STRING] --;
                    break;

case FIXED:
                    // leaf node so no descendants;
                    break;

case DELTA:
                    SB_SIG[FIXED] --;
                    break;

case FLOAT:
                    // leaf node so no descendants;
                    break;

case DIGITS:
                    SB_SIG[FLOAT] --;
                    break;

case GENERIC_PRIVATE:
```

```
                                SB_SIG[POSITIVE] --;
                                SB_SIG[NATURAL] --;
                                SB_SIG[RANGE] --;
                                SB_SIG[INTEGER] --;
                                SB_SIG[BOOLEAN] --;
                                SB_SIG[CHARACTER] --;
                                SB_SIG[ENUMERATION] --;
                                SB_SIG[DISCRETE] --;
                                SB_SIG[ACCESS] --;
                                SB_SIG[RECORD] --;
                                SB_SIG[PRIVATE] --;
                                SB_SIG[ARRAY] --;
                                SB_SIG[STRING] --;
                                SB_SIG[DELTA] --;
                                SB_SIG[FIXED] --;
                                SB_SIG[DIGITS] --;
                                SB_SIG[FLOAT] --;
                                break;

            default:
             printf("** Error in SIGNATURE::REMOVE_A_TYPE_ALL_DESCENDANTS()\n");
             break;

    }

}



// ****************************************************************
// ****              PUBLIC FUNCTIONS                  ****
// ****************************************************************



 /*  This function reads a text file with the Component signatures
     that was created by the Ada program responsible for encoding
     signatures.  A character-to-number conversion is necessary to
     load the actual numeric values for all the signature regions into
     the corresponding C++ signature arrays.  */
 void SIGNATURE::GET_SIGNATURES
                        (int IN_SIG[], int OUT_SIG[])
 {
  FILE *ifp;
  int INDEX;

  ifp = fopen("Signature.dat", "r");

  // get input signature
  for (INDEX = 0; INDEX < ALL_REGIONS; ++INDEX)
    IN_SIG[INDEX] = GET_NUMBER(ifp);

  // get output signature
  for (INDEX = 0; INDEX < ALL_REGIONS; ++INDEX)
    OUT_SIG[INDEX] = GET_NUMBER(ifp);
```

```
fclose(ifp);

}

/*  This function determines whether a query component input
    signature and a software base component input signature
    match each other.  The function returns a 1 if a match was
    made and a 0 if the two signatures do not match.  */
int SIGNATURE::MATCH_INPUT_SIGNATURES (int Q_SIG[], int SB_SIG[])
{
 int INDEX, MATCH;


 MATCH = 1;

 INDEX = 0;
 while (MATCH == 1 && INDEX < INPUT_REGIONS)
  {
   if (Q_SIG[INDEX] <= SB_SIG[INDEX])
          {
           INDEX = INDEX + 1;
          }
   else
          {
           MATCH = 0;
          }
  }
  return MATCH;
}


/*  This function determines whether a query component output
    signature and a software base component output signature
    match each other.  The function returns a 1 if a match was
    made and a 0 if the two signatures do not match.  */
int SIGNATURE::MATCH_OUTPUT_SIGNATURES (int Q_SIG[], int SB_SIG[])
{
 int INDEX, MATCH, EXCESS;
 int GENERIC_PRIVATE_NUM, GENERIC_DISCRETE_NUM,
        GENERIC_RANGE_NUM, GENERIC_ARRAY_NUM, GENERIC_DIGITS_NUM,
        GENERIC_DELTA_NUM, GENERIC_ACCESS_NUM;


 MATCH = 1;

 GENERIC_PRIVATE_NUM = SB_SIG[GENERIC_PRIVATE];
 GENERIC_DISCRETE_NUM = SB_SIG[GENERIC_DISCRETE];
 GENERIC_RANGE_NUM = SB_SIG[GENERIC_RANGE];
```

```
GENERIC_ARRAY_NUM = SB_SIG[GENERIC_ARRAY];
GENERIC_DIGITS_NUM = SB_SIG[GENERIC_DIGITS];
GENERIC_DELTA_NUM = SB_SIG[GENERIC_DELTA];
GENERIC_ACCESS_NUM = SB_SIG[GENERIC_ACCESS];

INDEX = 0;
while (MATCH == 1 && INDEX < OUTPUT_REGIONS)
 {
  EXCESS = Q_SIG[INDEX] - SB_SIG[INDEX];
  if (Q_SIG[INDEX] <= SB_SIG[INDEX])
        {
         INDEX = INDEX + 1;
        }

  else
        {
         MATCH = 0;
         switch (INDEX)
           {
            case POSITIVE:
                  if (GENERIC_DISCRETE_NUM > 0)
                    {
                     // reduce number of generics that
                     // can now be instantiated
                     if (GENERIC_DISCRETE_NUM >= EXCESS)
                      {
                            MATCH = 1;
                            GENERIC_DISCRETE_NUM =
                              GENERIC_DISCRETE_NUM - EXCESS;
                            EXCESS = 0;
              }

                       else
                            {
                             EXCESS = EXCESS - GENERIC_DISCRETE_NUM;
                             GENERIC_DISCRETE_NUM = 0;
                            }
                    }


                  if ( (EXCESS <= GENERIC_PRIVATE_NUM) &&
                     (EXCESS > 0) )
                    {
                     // reduce number of generics that
                     // can now be instantiated
                            MATCH = 1;
                            GENERIC_PRIVATE_NUM =
                              GENERIC_PRIVATE_NUM - EXCESS;
                            EXCESS = 0;
                    }

                  if (MATCH == 1)
                    {
                     // need to "instantiate" the software base
                     // signature with the matched query parameter
                     // to use for later type closeness calculations
```

```
            SB_SIG[INDEX] = SB_SIG[INDEX] + 1;
            ADD_INSTANTIATION_TO_SIGNATURE(INDEX, SB_SIG);
            INDEX = INDEX + 1;
           }

        break;

case NATURAL:
        if (GENERIC_DISCRETE_NUM > 0)
          {
           // reduce number of generics that
           // can now be instantiated
           if (GENERIC_DISCRETE_NUM >= EXCESS)
            {
                  MATCH = 1;
                  GENERIC_DISCRETE_NUM =
                    GENERIC_DISCRETE_NUM - EXCESS;
                  EXCESS = 0;
  }

           else
                 {
                  EXCESS = EXCESS - GENERIC_DISCRETE_NUM;
                  GENERIC_DISCRETE_NUM = 0;
                 }
          }


        if ( (EXCESS <= GENERIC_PRIVATE_NUM) &&
             (EXCESS > 0) )
          {
           // reduce number of generics that
           // can now be instantiated
                  MATCH = 1;
                  GENERIC_PRIVATE_NUM =
                    GENERIC_PRIVATE_NUM - EXCESS;
                  EXCESS = 0;
          }

        if (MATCH == 1)
          {
           // need to "instantiate" the software base
           // signature with the matched query parameter
           // to use for later type closeness calculations
           SB_SIG[INDEX] = SB_SIG[INDEX] + 1;
           ADD_INSTANTIATION_TO_SIGNATURE(INDEX, SB_SIG);
           INDEX = INDEX + 1;
          }

        break;

case RANGE:
        if (GENERIC_RANGE_NUM > 0)
          {
           // reduce number of generics that
           // can now be instantiated
```

```
        if (GENERIC_RANGE_NUM >= EXCESS)
         {
               MATCH = 1;
               GENERIC_RANGE_NUM =
                 GENERIC_RANGE_NUM - EXCESS;
               EXCESS = 0;
}

        else
              {
               EXCESS = EXCESS - GENERIC_RANGE_NUM;
               GENERIC_RANGE_NUM = 0;
              }
        }

      if ( (GENERIC_DISCRETE_NUM > 0) &&
              (EXCESS > 0) )
       {
        // reduce number of generics that
        // can now be instantiated
        if (GENERIC_DISCRETE_NUM >= EXCESS)
         {
               MATCH = 1;
               GENERIC_DISCRETE_NUM =
                 GENERIC_DISCRETE_NUM - EXCESS;
               EXCESS = 0;
}

        else
              {
               EXCESS = EXCESS - GENERIC_DISCRETE_NUM;
               GENERIC_DISCRETE_NUM = 0;
              }
       }


      if ( (EXCESS <= GENERIC_PRIVATE_NUM) &&
         (EXCESS > 0) )
       {
        // reduce number of generics that
        // can now be instantiated
               MATCH = 1;
               GENERIC_PRIVATE_NUM =
                 GENERIC_PRIVATE_NUM - EXCESS;
               EXCESS = 0;
        }

      if (MATCH == 1)
       {
        // need to "instantiate" the software base
        // signature with the matched query parameter
        // to use for later type closeness calculations
        SB_SIG[INDEX] = SB_SIG[INDEX] + 1;
        ADD_INSTANTIATION_TO_SIGNATURE(INDEX, SB_SIG);
        INDEX = INDEX + 1;
       }
```

```
        break;


case INTEGER:
        if (GENERIC_DISCRETE_NUM > 0)
          {
           // reduce number of generics that
           // can now be instantiated
           if (GENERIC_DISCRETE_NUM >= EXCESS)
            {
                  MATCH = 1;
                  GENERIC_DISCRETE_NUM =
                    GENERIC_DISCRETE_NUM - EXCESS;
                  EXCESS = 0;
  }

           else
                 {
                  EXCESS = EXCESS - GENERIC_DISCRETE_NUM;
                  GENERIC_DISCRETE_NUM = 0;
                 }
          }


        if ( (EXCESS <= GENERIC_PRIVATE_NUM) &&
           (EXCESS > 0) )
          {
           // reduce number of generics that
           // can now be instantiated
                  MATCH = 1;
                  GENERIC_PRIVATE_NUM =
                    GENERIC_PRIVATE_NUM - EXCESS;
                  EXCESS = 0;
          }

        if (MATCH == 1)
          {
           // need to "instantiate" the software base
           // signature with the matched query parameter
           // to use for later type closeness calculations
           SB_SIG[INDEX] = SB_SIG[INDEX] + 1;
           ADD_INSTANTIATION_TO_SIGNATURE(INDEX, SB_SIG);
           INDEX = INDEX + 1;
          }

        break;

case BOOLEAN:
        if (GENERIC_DISCRETE_NUM > 0)
          {
           // reduce number of generics that
           // can now be instantiated
           if (GENERIC_DISCRETE_NUM >= EXCESS)
            {
                  MATCH = 1;
```

```
                GENERIC_DISCRETE_NUM =
                  GENERIC_DISCRETE_NUM - EXCESS;
                EXCESS = 0;
    }

          else
                {
                 EXCESS = EXCESS - GENERIC_DISCRETE_NUM;
                 GENERIC_DISCRETE_NUM = 0;
                }
         }


        if ( (EXCESS <= GENERIC_PRIVATE_NUM) &&
           (EXCESS > 0) )
         {
          // reduce number of generics that
          // can now be instantiated
                MATCH = 1;
                GENERIC_PRIVATE_NUM =
                  GENERIC_PRIVATE_NUM - EXCESS;
                EXCESS = 0;
         }

        if (MATCH == 1)
         {
          // need to "instantiate" the software base
          // signature with the matched query parameter
          // to use for later type closeness calculations
          SB_SIG[INDEX] = SB_SIG[INDEX] + 1;
          ADD_INSTANTIATION_TO_SIGNATURE(INDEX, SB_SIG);
          INDEX = INDEX + 1;
         }

        break;

case CHARACTER:
        if (GENERIC_DISCRETE_NUM > 0)
         {
          // reduce number of generics that
          // can now be instantiated
          if (GENERIC_DISCRETE_NUM >= EXCESS)
           {
                MATCH = 1;
                GENERIC_DISCRETE_NUM =
                  GENERIC_DISCRETE_NUM - EXCESS;
                EXCESS = 0;
    }

          else
                {
                 EXCESS = EXCESS - GENERIC_DISCRETE_NUM;
                 GENERIC_DISCRETE_NUM = 0;
                }
         }
```

```
        if ( (EXCESS <= GENERIC_PRIVATE_NUM) &&
             (EXCESS > 0) )
          {
           // reduce number of generics that
           // can now be instantiated
                  MATCH = 1;
                  GENERIC_PRIVATE_NUM =
                    GENERIC_PRIVATE_NUM - EXCESS;
                  EXCESS = 0;
          }

        if (MATCH == 1)
          {
           // need to "instantiate" the software base
           // signature with the matched query parameter
           // to use for later type closeness calculations
           SB_SIG[INDEX] = SB_SIG[INDEX] + 1;
           ADD_INSTANTIATION_TO_SIGNATURE(INDEX, SB_SIG);
           INDEX = INDEX + 1;
          }

        break;

case ENUMERATION:
        if (GENERIC_DISCRETE_NUM > 0)
          {
           // reduce number of generics that
           // can now be instantiated
           if (GENERIC_DISCRETE_NUM >= EXCESS)
            {
                  MATCH = 1;
                  GENERIC_DISCRETE_NUM =
                    GENERIC_DISCRETE_NUM - EXCESS;
                  EXCESS = 0;
  }

            else
                 {
                  EXCESS = EXCESS - GENERIC_DISCRETE_NUM;
                  GENERIC_DISCRETE_NUM = 0;
                 }
          }


        if ( (EXCESS <= GENERIC_PRIVATE_NUM) &&
             (EXCESS > 0) )
          {
           // reduce number of generics that
           // can now be instantiated
                  MATCH = 1;
                  GENERIC_PRIVATE_NUM =
                    GENERIC_PRIVATE_NUM - EXCESS;
                  EXCESS = 0;
          }
```

```
        if (MATCH == 1)
          {
           // need to "instantiate" the software base
           // signature with the matched query parameter
           // to use for later type closeness calculations
           SB_SIG[INDEX] = SB_SIG[INDEX] + 1;
           ADD_INSTANTIATION_TO_SIGNATURE(INDEX, SB_SIG);
           INDEX = INDEX + 1;
          }

        break;

case DISCRETE:
        if (GENERIC_RANGE_NUM > 0)
          {
           // reduce number of generics that
           // can now be instantiated
           if (GENERIC_RANGE_NUM >= EXCESS)
            {
                  MATCH = 1;
                  GENERIC_RANGE_NUM =
                    GENERIC_RANGE_NUM - EXCESS;
                  EXCESS = 0;
}

           else
                 {
                  EXCESS = EXCESS - GENERIC_RANGE_NUM;
                  GENERIC_RANGE_NUM = 0;
                 }
          }

        if ( (GENERIC_DISCRETE_NUM > 0) &&
                (EXCESS > 0) )
          {
           // reduce number of generics that
           // can now be instantiated
           if (GENERIC_DISCRETE_NUM >= EXCESS)
            {
                  MATCH = 1;
                  GENERIC_DISCRETE_NUM =
                    GENERIC_DISCRETE_NUM - EXCESS;
                  EXCESS = 0;
}

           else
                 {
                  EXCESS = EXCESS - GENERIC_DISCRETE_NUM;
                  GENERIC_DISCRETE_NUM = 0;
                 }
          }


        if ( (EXCESS <= GENERIC_PRIVATE_NUM) &&
           (EXCESS > 0) )
          {
```

```
          // reduce number of generics that
          // can now be instantiated
                MATCH = 1;
                GENERIC_PRIVATE_NUM =
                  GENERIC_PRIVATE_NUM - EXCESS;
                EXCESS = 0;
          }

        if (MATCH == 1)
          {
          // need to "instantiate" the software base
          // signature with the matched query parameter
          // to use for later type closeness calculations
          SB_SIG[INDEX] = SB_SIG[INDEX] + 1;
          ADD_INSTANTIATION_TO_SIGNATURE(INDEX, SB_SIG);
          INDEX = INDEX + 1;
          }

        break;

case ACCESS:
        if (GENERIC_ACCESS_NUM > 0)
          {
          // reduce number of generics that
          // can now be instantiated
          if (GENERIC_ACCESS_NUM >= EXCESS)
           {
                MATCH = 1;
                GENERIC_ACCESS_NUM =
                  GENERIC_ACCESS_NUM - EXCESS;
                EXCESS = 0;
  }

          else
                {
                 EXCESS = EXCESS - GENERIC_ACCESS_NUM;
                 GENERIC_ACCESS_NUM = 0;
                }
          }


        if ( (EXCESS <= GENERIC_PRIVATE_NUM) &&
             (EXCESS > 0) )
          {
          // reduce number of generics that
          // can now be instantiated
                MATCH = 1;
                GENERIC_PRIVATE_NUM =
                  GENERIC_PRIVATE_NUM - EXCESS;
                EXCESS = 0;
          }

        if (MATCH == 1)
          {
          // need to "instantiate" the software base
          // signature with the matched query parameter
```

```
            // to use for later type closeness calculations
            SB_SIG[INDEX] = SB_SIG[INDEX] + 1;
            ADD_INSTANTIATION_TO_SIGNATURE(INDEX, SB_SIG);
            INDEX = INDEX + 1;
           }

          break;

case RECORD:
          if ( (EXCESS <= GENERIC_PRIVATE_NUM) &&
             (EXCESS > 0) )
           {
            // reduce number of generics that
            // can now be instantiated
                  MATCH = 1;
                  GENERIC_PRIVATE_NUM =
                    GENERIC_PRIVATE_NUM - EXCESS;
                  EXCESS = 0;
           }

          if (MATCH == 1)
           {
            // need to "instantiate" the software base
            // signature with the matched query parameter
            // to use for later type closeness calculations
            SB_SIG[INDEX] = SB_SIG[INDEX] + 1;
            ADD_INSTANTIATION_TO_SIGNATURE(INDEX, SB_SIG);
            INDEX = INDEX + 1;
           }

          break;

case PRIVATE:
          if (GENERIC_PRIVATE_NUM > 0)
           {
            // reduce number of generics that
            // can now be instantiated
            if (GENERIC_PRIVATE_NUM >= EXCESS)
             {
                  MATCH = 1;
                  GENERIC_PRIVATE_NUM =
                    GENERIC_PRIVATE_NUM - EXCESS;
                  EXCESS = 0;
  }

            else
                 {
                  EXCESS = EXCESS - GENERIC_PRIVATE_NUM;
                  GENERIC_PRIVATE_NUM = 0;
                 }
           }

          if (MATCH == 1)
           {
            // need to "instantiate" the software base
            // signature with the matched query parameter
```

```
          // to use for later type closeness calculations
          SB_SIG[INDEX] = SB_SIG[INDEX] + 1;
          ADD_INSTANTIATION_TO_SIGNATURE(INDEX, SB_SIG);
          INDEX = INDEX + 1;
         }

        break;

case STRING:
        if (GENERIC_ARRAY_NUM > 0)
          {
          // reduce number of generics that
          // can now be instantiated
          if (GENERIC_ARRAY_NUM >= EXCESS)
           {
                MATCH = 1;
                GENERIC_ARRAY_NUM =
                  GENERIC_ARRAY_NUM - EXCESS;
                EXCESS = 0;
  }

           else
                {
                 EXCESS = EXCESS - GENERIC_ARRAY_NUM;
                 GENERIC_ARRAY_NUM = 0;
                }
          }


        if ( (EXCESS <= GENERIC_PRIVATE_NUM) &&
            (EXCESS > 0) )
          {
          // reduce number of generics that
          // can now be instantiated
                MATCH = 1;
                GENERIC_PRIVATE_NUM =
                  GENERIC_PRIVATE_NUM - EXCESS;
                EXCESS = 0;
          }

        if (MATCH == 1)
          {
          // need to "instantiate" the software base
          // signature with the matched query parameter
          // to use for later type closeness calculations
          SB_SIG[INDEX] = SB_SIG[INDEX] + 1;
          ADD_INSTANTIATION_TO_SIGNATURE(INDEX, SB_SIG);
          INDEX = INDEX + 1;
          }

        break;

case ARRAY:
        if (GENERIC_ARRAY_NUM > 0)
          {
          // reduce number of generics that
```

```
            // can now be instantiated
            if (GENERIC_ARRAY_NUM >= EXCESS)
             {
                    MATCH = 1;
                    GENERIC_ARRAY_NUM =
                      GENERIC_ARRAY_NUM - EXCESS;
                    EXCESS = 0;
  }

            else
                   {
                    EXCESS = EXCESS - GENERIC_ARRAY_NUM;
                    GENERIC_ARRAY_NUM = 0;
                   }
           }


          if ( (EXCESS <= GENERIC_PRIVATE_NUM) &&
             (EXCESS > 0) )
           {
            // reduce number of generics that
            // can now be instantiated
                    MATCH = 1;
                    GENERIC_PRIVATE_NUM =
                      GENERIC_PRIVATE_NUM - EXCESS;
                    EXCESS = 0;
           }

          if (MATCH == 1)
           {
            // need to "instantiate" the software base
            // signature with the matched query parameter
            // to use for later type closeness calculations
            SB_SIG[INDEX] = SB_SIG[INDEX] + 1;
            ADD_INSTANTIATION_TO_SIGNATURE(INDEX, SB_SIG);
            INDEX = INDEX + 1;
           }

          break;

case FIXED:
          if (GENERIC_DELTA_NUM > 0)
           {
            // reduce number of generics that
            // can now be instantiated
            if (GENERIC_DELTA_NUM >= EXCESS)
             {
                    MATCH = 1;
                    GENERIC_DELTA_NUM =
                      GENERIC_DELTA_NUM - EXCESS;
                    EXCESS = 0;
  }

            else
                   {
                    EXCESS = EXCESS - GENERIC_DELTA_NUM;
```

```
                GENERIC_DELTA_NUM = 0;
               }
        }


      if ( (EXCESS <= GENERIC_PRIVATE_NUM) &&
         (EXCESS > 0) )
        {
         // reduce number of generics that
         // can now be instantiated
               MATCH = 1;
               GENERIC_PRIVATE_NUM =
                 GENERIC_PRIVATE_NUM - EXCESS;
               EXCESS = 0;
        }

      if (MATCH == 1)
        {
         // need to "instantiate" the software base
         // signature with the matched query parameter
         // to use for later type closeness calculations
         SB_SIG[INDEX] = SB_SIG[INDEX] + 1;
         ADD_INSTANTIATION_TO_SIGNATURE(INDEX, SB_SIG);
         INDEX = INDEX + 1;
        }

      break;

case DELTA:
      if (GENERIC_DELTA_NUM > 0)
        {
         // reduce number of generics that
         // can now be instantiated
         if (GENERIC_DELTA_NUM >= EXCESS)
          {
               MATCH = 1;
               GENERIC_DELTA_NUM =
                 GENERIC_DELTA_NUM - EXCESS;
               EXCESS = 0;
}

          else
               {
                EXCESS = EXCESS - GENERIC_DELTA_NUM;
                GENERIC_DELTA_NUM = 0;
               }
        }


      if ( (EXCESS <= GENERIC_PRIVATE_NUM) &&
         (EXCESS > 0) )
        {
         // reduce number of generics that
         // can now be instantiated
               MATCH = 1;
               GENERIC_PRIVATE_NUM =
```

```
                GENERIC_PRIVATE_NUM - EXCESS;
              EXCESS = 0;
          }

        if (MATCH == 1)
          {
          // need to "instantiate" the software base
          // signature with the matched query parameter
          // to use for later type closeness calculations
          SB_SIG[INDEX] = SB_SIG[INDEX] + 1;
          ADD_INSTANTIATION_TO_SIGNATURE(INDEX, SB_SIG);
          INDEX = INDEX + 1;
          }

        break;

case FLOAT:
        if (GENERIC_DIGITS_NUM > 0)
          {
          // reduce number of generics that
          // can now be instantiated
          if (GENERIC_DIGITS_NUM >= EXCESS)
           {
                MATCH = 1;
                GENERIC_DIGITS_NUM =
                  GENERIC_DIGITS_NUM - EXCESS;
                EXCESS = 0;
 }

           else
                {
                 EXCESS = EXCESS - GENERIC_DIGITS_NUM;
                 GENERIC_DIGITS_NUM = 0;
                }
          }


        if ( (EXCESS <= GENERIC_PRIVATE_NUM) &&
           (EXCESS > 0) )
          {
          // reduce number of generics that
          // can now be instantiated
                MATCH = 1;
                GENERIC_PRIVATE_NUM =
                  GENERIC_PRIVATE_NUM - EXCESS;
                EXCESS = 0;
          }

        if (MATCH == 1)
          {
          // need to "instantiate" the software base
          // signature with the matched query parameter
          // to use for later type closeness calculations
          SB_SIG[INDEX] = SB_SIG[INDEX] + 1;
          ADD_INSTANTIATION_TO_SIGNATURE(INDEX, SB_SIG);
          INDEX = INDEX + 1;
```

```
            }

            break;

    case DIGITS:
            if (GENERIC_DIGITS_NUM > 0)
              {
               // reduce number of generics that
               // can now be instantiated
               if (GENERIC_DIGITS_NUM >= EXCESS)
                {
                      MATCH = 1;
                      GENERIC_DIGITS_NUM =
                        GENERIC_DIGITS_NUM - EXCESS;
                      EXCESS = 0;
      }

               else
                     {
                      EXCESS = EXCESS - GENERIC_DIGITS_NUM;
                      GENERIC_DIGITS_NUM = 0;
                     }
              }


            if ( (EXCESS <= GENERIC_PRIVATE_NUM) &&
                 (EXCESS > 0) )
              {
               // reduce number of generics that
               // can now be instantiated
                      MATCH = 1;
                      GENERIC_PRIVATE_NUM =
                        GENERIC_PRIVATE_NUM - EXCESS;
                      EXCESS = 0;
              }

            if (MATCH == 1)
              {
               // need to "instantiate" the software base
               // signature with the matched query parameter
               // to use for later type closeness calculations
               SB_SIG[INDEX] = SB_SIG[INDEX] + 1;
               ADD_INSTANTIATION_TO_SIGNATURE(INDEX, SB_SIG);
               INDEX = INDEX + 1;
              }

            break;

    default:
            printf("** Error in SIGNATURE::MATCH_OUTPUT_SIGNATURES()\n");
            break;

   }

}
```

```
  }
 return MATCH;
}



/*  This function is only necessary for input signatures.  It also
    calculates input signature closeness degree as a by product.  It
    is necessary to calculate the closeness degree in this manner
    when the number of input parameters in the query component are not
    equal to the number of input parameters in the software base
    component (which could occur with ADT aggregate input signatures).
    This function returns the value of the closeness degree if a valid
    match was found, or a -1 if a False Match was found. */
int SIGNATURE::CHECK_FALSE_MATCH (int Q_SIG[], int SB_SIG[])
{

 int REGION, CLOSENESS, NOT_DONE, INDEX, MATCH;
 int Lv_Q_SIG[ALL_REGIONS], Lv_SB_SIG[ALL_REGIONS];

 /* Copy contents of query and software base signature arrays into
    local variable (Lv) arrays. */
 for (INDEX = 0; INDEX < ALL_REGIONS; ++INDEX)
   {
   Lv_Q_SIG[INDEX] = Q_SIG[INDEX];
   Lv_SB_SIG[INDEX] = SB_SIG[INDEX];
   }
 MATCH = 0;
 CLOSENESS = 0;

 /* Loop through the leaf nodes.  Once all a node's children are
    processed, it becomes a leaf node.  That is why we can loop
    from Regions 1 (0) through Region 18 (17).  The order ensures
    we are always at a leaf node.  */
for (INDEX = 0; INDEX < INPUT_REGIONS; ++INDEX)
  {

  // The while loop continues as long as the leaf region has
  // type instances.
  while (Lv_Q_SIG[INDEX] > 0)
        {
        REGION = INDEX;
        NOT_DONE = 1;
        /* There are four cases that can change the status of the
          NOT_DONE boolean variable:
          case 1 - The parent node of the current query node has zero
                   type instances.
          case 2 - The current query node's parent has fewer type
                   instances then the current query node.
          case 3 - The current query node is at the top of the Ada subtype
                   hierarchy (i.e., generic Private)
          case 4 - The software base current node has zero type instances
                   and the current query node has one or more type
                   instances (FALSE MATCH).*/
```

```
while (NOT_DONE)

 {

        /* Put Case 3 first to "short circuit" the other cases from
           being evaluated if Region = 17.  */
  if ( (REGION == GENERIC_PRIVATE) ||                  // Case 3
        (Lv_Q_SIG[PARENT(REGION)] == 0) ||             // Case 1
        (Lv_Q_SIG[PARENT(REGION)] < Lv_Q_SIG[REGION]) ||   // Case 2
        ((Lv_SB_SIG[REGION] == 0) && (Lv_Q_SIG[REGION] > 0)) )
                                                              // Case 4

    if  ((Lv_SB_SIG[REGION] == 0) && (Lv_Q_SIG[REGION] > 0)) // Case 4
        { // We have a False Match ... No further processing needed.
  MATCH = -1;
  return MATCH;
        }
    else
        { /* We have completed the trace of a single query component
             input parameter and it has a valid matching parameter
             in the software base component. */
          NOT_DONE = 0;
         /* We have found the node that represents the type of
           the query component input parameter. Remove a type
           instance from this node and all descendants. */
         Lv_Q_SIG[REGION] --;
         REMOVE_A_TYPE_FROM_ALL_DESCENDANTS(REGION, Lv_Q_SIG);

         }
   else // continue up the Ada subhierarchy
     {
     // Remove a type instance from the query region.
//       Lv_Q_SIG[REGION] --;
     // move up to the parent region
     REGION = PARENT(REGION);
     }
  } // end while NOT_DONE

  /* Proceed up the Ada subtype hierarchy in the software base
     component.  We stop if Cases 1 - 3 described above are
     encountered. Each move up the hierarchy corresponds to
     a difference of 1 degree of closeness between the query
     component and software base component parameters. */
  NOT_DONE = 1;
  while (NOT_DONE)
  {

        /* Put Case 3 first to "short circuit" the other cases from
           being evaluated if Region = 17. */
  if ( (REGION == GENERIC_PRIVATE) ||                  // Case 3
        (Lv_SB_SIG[PARENT(REGION)] == 0) ||            // Case 1
        (Lv_SB_SIG[PARENT(REGION)] < Lv_SB_SIG[REGION]) )  // Case 2
    {
        // Halt the upward trace.
        NOT_DONE = 0;
        /* We have found the node that represents the type of
```

```
                    the software base input parameter matched to the query
                    component input parameter.  Remove a type instance from
                    this node and all descendants. */
                  Lv_SB_SIG[REGION] --;
                  REMOVE_A_TYPE_FROM_ALL_DESCENDANTS(REGION, Lv_SB_SIG);

              }

            else // continue up the Ada subhierarchy
              {
              // add last upward move to closeness degree
              CLOSENESS = CLOSENESS + 1;
              // move up to the parent region
              REGION = PARENT(REGION);
              }
            } // end while NOT_DONE


        }

  }

// Return the value of the degree of closeness which, since it must
// be 0 or greater implies that a False Match was not found.
MATCH = CLOSENESS;
return MATCH;

}



/* Calculates the output signature closeness degree. */
int SIGNATURE::CALC_OUT_CLOSE_DEGREE (int Q_SIG[], int SB_SIG[])
{


 int REGION, CLOSENESS, NOT_DONE, INDEX;
 int Lv_Q_SIG[ALL_REGIONS], Lv_SB_SIG[ALL_REGIONS];

 /* Copy contents of query and software base signature arrays into
    local variable (Lv) arrays. */
 for (INDEX = 0; INDEX < ALL_REGIONS; ++INDEX)
   {
   Lv_Q_SIG[INDEX] = Q_SIG[INDEX];
   Lv_SB_SIG[INDEX] = SB_SIG[INDEX];
   }
 CLOSENESS = 0;

 /* Loop through the leaf nodes.  Once all a node's children are
    processed, it becomes a leaf node.  That is why we can loop
    from Regions 1 (0) through Region 17 (16).  The order ensures
    we are always at a leaf node.  */
for (INDEX = 0; INDEX < OUTPUT_REGIONS; ++INDEX)
  {
```

```
// The while loop continues as long as the leaf region has type
// instances.
while (Lv_Q_SIG[INDEX] > 0)
        {

        REGION = INDEX;
        /* Remove a type instance from this Region (query only) and all
           ancestor Regions in both the query and software
           base component. */
        Lv_Q_SIG[REGION] --;
        REMOVE_A_TYPE_FROM_ALL_ANCESTORS(REGION, Lv_Q_SIG, Lv_SB_SIG);


        /* We must now continue down through the Ada subtype hierarchy
           with the software base component.  Since there are multiple
           downward paths (a node can have more than 1 child) we must
           choose the shortest possible path.  */

        NOT_DONE = 1;
        /* There are two cases that can change the status of the
           NOT_DONE boolean variable:
           case 1 - The current software base node is a leaf node.
           case 2 - The current software base node has more type
                        instances than the total number of type instances
                        in its children. This means we have found the
                        closest possible matching output parameter.
           However, because we couldn't have reached a leaf node
           unless it had one or more type instances, and because it then
           follows that the number of type instances in a leaf node is
           greater than the number of type instances of its children
           (which must be zero), we only need to apply Case 2 as a test. */
        while (NOT_DONE)
          {

          if ( Lv_SB_SIG[REGION] >
                  SUM_OF_CHILDREN_TYPES(REGION, Lv_SB_SIG) )   // Case 2

                 {
                  /* We have completed the trace of a single software base
                     component output parameter that matches the parameter
                     in the query component. */
                  NOT_DONE = 0;
                  // Remove a type instance from the current region.
                  Lv_SB_SIG[REGION] --;
                  }
          else // continue down the Ada subhierarchy
            {
            // Remove a type instance from the current region.
            Lv_SB_SIG[REGION] --;

            // add last upward move to closeness degree
            CLOSENESS = CLOSENESS + 1;
            // move down to the best child region
            REGION = BEST_CHILD(REGION, Lv_SB_SIG);
            }
          } // end while NOT_DONE
```

```
        }

    }

// Return the value of the degree of closeness.
return CLOSENESS;

}
```

APPENDIX F - MODIFICATIONS TO ORIGINAL CAPS SOFTWARE BASE C++ SOURCE CODE

The files listed below contain the C++ source code written by John McDowell in which the more extensive modifications were made to extend the CAPS software base capabilities.

filename: sbacl.cxx

```
/*  Original software for CAPS software base developed by
    John Kelly McDowell.


    Modified - 31 August 1993 by Scott Dolgoff
      Because modifications were fairly extensive in this module,
      refer to McDowell's thesis to see what the original code
      looked like.  Essentially, his by_num_generics dictionary has
      been removed from the complex data hierarchy.  In its place are
      two new dictionaries;  input_signature and output_signature.
      I have put my variables in UPPER CASE to help differentiate. */

// ADDED
// note:  ADT_Signature.h includes sball.hxx, sbextern.h and Signature.h
#include "ADT_Signature.h"
#include <Set.h>
// #include <Primitives.h>



SB_ADT_COMPONENT_LIBRARY::SB_ADT_COMPONENT_LIBRARY(APL *theAPL) :
Object(theAPL)
{
};


SB_ADT_COMPONENT_LIBRARY::SB_ADT_COMPONENT_LIBRARY() : Object()
{

 SB_COMPONENT_DICTIONARY *new_adt_component_dictionary=
   new SB_COMPONENT_DICTIONARY();

 the_adt_component_dictionary=
   new_adt_component_dictionary->findTRef();

 Dictionary *new_main_library=new Dictionary(OC_integer,
                                             OC_dictionary,
                                             TRUE,
                                             FALSE);
 the_main_library=new_main_library->findTRef();
```

```
};

void SB_ADT_COMPONENT_LIBRARY::Destroy(Boolean aborted)
{
 adt_component_dictionary()->Destroy(aborted);

 // now must iterate through the multi-attribute tree of
 // dictionaryies to destroy each one of them

 Dictionary *by_num_adts;
 Dictionary *by_num_operators;
 Dictionary *by_num_total_inputs;
 Dictionary *by_num_generics;
 Dictionary *by_num_total_outputs;
 Dictionary *leaf_dictionary;

 by_num_adts=main_library();

 DictionaryIterator next_by_num_adt(by_num_adts);

 while(next_by_num_adt.moreData())
  {
   by_num_operators=(Dictionary *)(Entity *)next_by_num_adt();

   // components must have at least as many operators as the query

   DictionaryIterator next_by_num_operators(by_num_operators);

   while(next_by_num_operators.moreData())
         {
          by_num_generics=(Dictionary *)(Entity *)
           next_by_num_operators();

          DictionaryIterator next_by_num_generics(by_num_generics);

          while(next_by_num_generics.moreData())
           {

            by_num_total_outputs=(Dictionary *)(Entity *)
                  next_by_num_generics();

            DictionaryIterator
                  next_by_num_total_outputs(by_num_total_outputs);

            while(next_by_num_total_outputs.moreData())
                  {

                   by_num_total_inputs=(Dictionary *)(Entity *)
                    next_by_num_total_outputs();

                   DictionaryIterator
                    next_by_num_total_inputs(by_num_total_inputs);

                   while(next_by_num_total_inputs.moreData())
                    {
```

```
                        leaf_dictionary=(Dictionary *)(Entity *)
                              next_by_num_total_inputs();

                        leaf_dictionary->Destroy(aborted);

                       };
                     by_num_total_inputs->Destroy(aborted);
                    };
              by_num_total_outputs->Destroy(aborted);

             };
       by_num_generics->Destroy(aborted);
          };
    by_num_adts->Destroy(aborted);
   };

 delete the_adt_component_dictionary;
 delete the_main_library;

 Object::Destroy(aborted);

};

void SB_ADT_COMPONENT_LIBRARY::deleteObject(Boolean deallocate)
{
 adt_component_dictionary()->deleteObject(deallocate);

 // now must iterate through the multi-attribute tree of
 // dictionaryies to destroy each one of them

 Dictionary *by_num_adts;
 Dictionary *by_num_operators;
 Dictionary *by_num_total_inputs;
 Dictionary *by_num_generics;
 Dictionary *by_num_total_outputs;
 Dictionary *leaf_dictionary;

 by_num_adts=main_library();

 DictionaryIterator next_by_num_adt(by_num_adts);

 while(next_by_num_adt.moreData())
  {
   by_num_operators=(Dictionary *)(Entity *)next_by_num_adt();

   // components must have at least as many operators as the query

   DictionaryIterator next_by_num_operators(by_num_operators);

   while(next_by_num_operators.moreData())
        {
         by_num_generics=(Dictionary *)(Entity *)
          next_by_num_operators();

         DictionaryIterator next_by_num_generics(by_num_generics);
```

```
            while(next_by_num_generics.moreData())
              {

               by_num_total_outputs=(Dictionary *)(Entity *)
                     next_by_num_generics();

               DictionaryIterator
                     next_by_num_total_outputs(by_num_total_outputs);

               while(next_by_num_total_outputs.moreData())
                     {

                      by_num_total_inputs=(Dictionary *)(Entity *)
                       next_by_num_total_outputs();

                      DictionaryIterator
                       next_by_num_total_inputs(by_num_total_inputs);

                      while(next_by_num_total_inputs.moreData())
                        {
                         leaf_dictionary=(Dictionary *)(Entity *)
                               next_by_num_total_inputs();

                         leaf_dictionary->deleteObject(FALSE);

                        };
                      by_num_total_inputs->deleteObject(FALSE);
                     };
               by_num_total_outputs->deleteObject(FALSE);

              };
        by_num_generics->deleteObject(FALSE);
           };
     by_num_adts->deleteObject(FALSE);
    };

  Object::deleteObject(deallocate);

};

void SB_ADT_COMPONENT_LIBRARY::putObject(Boolean deallocate)
{
 adt_component_dictionary()->Dictionary::putObject(deallocate);
 main_library()->putObject(deallocate);

 Object::putObject(deallocate);

};


Type *SB_ADT_COMPONENT_LIBRARY::getDirectType()
{
 return SB_ADT_COMPONENT_LIBRARY_OType;
};

SB_COMPONENT_DICTIONARY *SB_ADT_COMPONENT_LIBRARY::adt_component_dictionary()
```

```
{
 return (SB_COMPONENT_DICTIONARY *)(Entity *)
   the_adt_component_dictionary->Binding();
};

Dictionary *SB_ADT_COMPONENT_LIBRARY::main_library()
{
 return (Dictionary *)(Entity *)the_main_library->Binding();
};


// ADDED sig_file as a parameter to "add" function
Boolean SB_ADT_COMPONENT_LIBRARY::add(SB_ADT_COMPONENT *new_component,
                                      char *SIG_FILE)
{
 Boolean return_flag=TRUE;


// ADDED
 Boolean NEW_INPUT_SIGNATURE_ARRAY = FALSE;
 Boolean NEW_OUTPUT_SIGNATURE_ARRAY = FALSE;
 Dictionary *BY_INPUT_SIGNATURE_DICTIONARY;
 Dictionary *BY_OUTPUT_SIGNATURE_DICTIONARY;


 Dictionary *by_num_adts;
 Dictionary *by_num_operators;
 Dictionary *by_num_total_inputs;
 Dictionary *by_num_total_outputs;
 Dictionary *leaf_dictionary;

 adt_component_dictionary()->add(new_component);
 adt_component_dictionary()->Dictionary::putObject();

 // insert into the component dictionary was successfull
 // so insert it into the library

 // get the dictionary for the number of adt's

 by_num_adts=main_library();

 // now find the dictionary for adt_operators

 if(by_num_adts->isIndex(new_component->num_adts()))
   {
    by_num_operators=(Dictionary *)(Entity *)(*by_num_adts)
          [new_component->num_adts()];
   }
 else
   {
    by_num_operators=new Dictionary(OC_integer,
                                    OC_dictionary,
                                    TRUE,
                                    FALSE);

    by_num_adts->Insert(new_component->
```

```
                                num_adts(),
                                by_num_operators);

  };


// have correct by_num_operator dictionary so get the
// total number of inputs dictionary

if(by_num_operators->isIndex(new_component->
                                num_adt_operators()))
  {
   by_num_total_inputs=(Dictionary *)(Entity *)
         (*by_num_operators)
          [ new_component->
           num_adt_operators()];
  }
else
  {
   by_num_total_inputs=new Dictionary(OC_integer,
                                      OC_dictionary,
                                      TRUE,
                                      FALSE);


   by_num_operators->Insert(new_component->
                                num_adt_operators(),
                                by_num_total_inputs);

  };


// got the total number of inputs dictionary so now get the total
// number of outputs dictionary


if(by_num_total_inputs->
  isIndex(new_component->total_inputs())==TRUE)
  {
   by_num_total_outputs=(Dictionary *)(Entity *)
         (*by_num_total_inputs)
          [new_component->total_inputs()];
  }
else
  {
   by_num_total_outputs=new Dictionary(OC_integer,
                                    OC_dictionary,
                                        TRUE,
                                        FALSE);


   by_num_total_inputs->Insert(new_component->
                                    total_inputs(),
                                    by_num_total_outputs);

  };
```

```
// got the total number of outputs dictionary so now get the
// INPUT SIGNATURE dictionary

if(by_num_total_outputs->
  isIndex(new_component->total_outputs())==TRUE)
  {
   BY_INPUT_SIGNATURE_DICTIONARY=(Dictionary *)(Entity *)
         (*by_num_total_outputs)
          [new_component->total_outputs()];
  }
else
  {
   BY_INPUT_SIGNATURE_DICTIONARY=new Dictionary(OC_array,
                                                OC_dictionary,
                                                FALSE,
                                                FALSE);


   by_num_total_outputs->Insert(new_component->
                                total_outputs(),
                                BY_INPUT_SIGNATURE_DICTIONARY);

  };

// Have correct  dictionary for the component's total ADT operator outputs.
// Now get correct INPUT_SIGNATURE dictionary.


// first get the SIGNATURE value from SIG_FILE

ifstream SIGNATURE(SIG_FILE);

int REGION, REGION_VALUE;

Array *INPUT_SIGNATURE_ARRAY = new Array (OC_integer, ALL_REGIONS, 1);

for (REGION=1; REGION <= ALL_REGIONS; ++REGION)
  {
   SIGNATURE >>  REGION_VALUE;
   INPUT_SIGNATURE_ARRAY->setElement(REGION, REGION_VALUE);
  }

//  Can't use isIndex to see if the new signature is already an
//  index in the dictionary because each index is an object
//  with a unique id.  Therefore, must iterate through indices
//  and see if one isSimilar (i.e. of equal value) to the
// INPUT_SIGNATURE_ARRAY.
Array *FOUND_INDEX_ARRAY;
int NOT_DONE = 1;
int INDEX_FOUND = 0;

DictionaryIterator NEXT_INPUT_SIGNATURE =
         DictionaryIterator(BY_INPUT_SIGNATURE_DICTIONARY, TRUE);
```

```
while( NEXT_INPUT_SIGNATURE.moreData() && NOT_DONE )
 {
  FOUND_INDEX_ARRAY = (Array *)(Entity *)
                                        NEXT_INPUT_SIGNATURE();

  // compare the two signature arrays
  if (FOUND_INDEX_ARRAY->isSimilar(INPUT_SIGNATURE_ARRAY))
    {
          NOT_DONE = 0;
          INDEX_FOUND = 1;
    };
 }

if(INDEX_FOUND)
  {

   BY_OUTPUT_SIGNATURE_DICTIONARY=(Dictionary *)(Entity *)
         (*BY_INPUT_SIGNATURE_DICTIONARY) [FOUND_INDEX_ARRAY];

  }
else
  {

   BY_OUTPUT_SIGNATURE_DICTIONARY=new Dictionary(OC_array,
                                                 OC_dictionary,
                                                 FALSE,
                                                 FALSE);

   NEW_INPUT_SIGNATURE_ARRAY = TRUE;

   BY_INPUT_SIGNATURE_DICTIONARY->Insert(INPUT_SIGNATURE_ARRAY,
                                         BY_OUTPUT_SIGNATURE_DICTIONARY);

  };



// Have correct  dictionary for the component's INPUT SIGNATURE.
// Now get correct OUTPUT_SIGNATURE dictionary.


// first get OUTPUT_SIGNATURE from SIG_FILE

Array *OUTPUT_SIGNATURE_ARRAY = new Array (OC_integer, ALL_REGIONS, 1);

for (REGION=1; REGION <= ALL_REGIONS; ++REGION)
  {
   SIGNATURE >>  REGION_VALUE;
   OUTPUT_SIGNATURE_ARRAY->setElement(REGION, REGION_VALUE);
  }


//////////////////
//  Can't use isIndex to see if the new signature is already an
//  index in the dictionary because each index is an object
//  with a unique id.  Therefore, must iterate through indices
```

```
// and see if one isSimilar (i.e. of equal value) to the
// OUTPUT_SIGNATURE_ARRAY.
NOT_DONE = 1;
INDEX_FOUND = 0;

DictionaryIterator NEXT_OUTPUT_SIGNATURE =
        DictionaryIterator(BY_OUTPUT_SIGNATURE_DICTIONARY, TRUE);

while( NEXT_OUTPUT_SIGNATURE.moreData() && NOT_DONE )
 {
  FOUND_INDEX_ARRAY = (Array *)(Entity *)
                                    NEXT_OUTPUT_SIGNATURE();

  // compare the two signature arrays
  if (FOUND_INDEX_ARRAY->isSimilar(OUTPUT_SIGNATURE_ARRAY))
    {
        NOT_DONE = 0;
        INDEX_FOUND = 1;
    };
 }

if(INDEX_FOUND)
  {

   leaf_dictionary=(Dictionary *)(Entity *)
        (*BY_OUTPUT_SIGNATURE_DICTIONARY) [FOUND_INDEX_ARRAY];

  }

else
  {
   leaf_dictionary=new Dictionary(OC_string,
                                    SB_COMPONENT_OType,
                                    FALSE,
                                    FALSE);

   NEW_OUTPUT_SIGNATURE_ARRAY = TRUE;

   BY_OUTPUT_SIGNATURE_DICTIONARY->Insert(OUTPUT_SIGNATURE_ARRAY,
                                        leaf_dictionary);


  };



// have the leaf dictionary so now insert the component into it
leaf_dictionary->Insert(new_component->component_name(),
                        new_component);


by_num_adts->putObject();
by_num_operators->putObject();
by_num_total_inputs->putObject();
by_num_total_outputs->putObject();
leaf_dictionary->putObject();;
```

```
 BY_INPUT_SIGNATURE_DICTIONARY->putObject();
 BY_OUTPUT_SIGNATURE_DICTIONARY->putObject();
 if (NEW_INPUT_SIGNATURE_ARRAY)
  INPUT_SIGNATURE_ARRAY->putObject();
 if (NEW_OUTPUT_SIGNATURE_ARRAY)
  OUTPUT_SIGNATURE_ARRAY->putObject();


 return return_flag;
};

void SB_ADT_COMPONENT_LIBRARY::delete_component(SB_ADT_COMPONENT *the_component)
{

 Dictionary *by_num_adts;
 Dictionary *by_num_operators;
 Dictionary *by_num_total_inputs;
 Dictionary *by_num_generics;
 Dictionary *by_num_total_outputs;
 Dictionary *leaf_dictionary;


 adt_component_dictionary()->Remove(the_component->component_name());

 adt_component_dictionary()->Dictionary::putObject();


 by_num_adts=main_library();

 // now find the dictionary for adt_operators

 if(by_num_adts->isIndex(the_component->num_adts()))
  {
   by_num_operators=(Dictionary *)(Entity *)(*by_num_adts)
        [the_component->num_adts()];
   // have correct by_num_operator dictionary so get the
   // generic types dict.

   if(by_num_operators->isIndex(the_component->
                                        num_adt_operators()))
        {
         by_num_generics=(Dictionary *)(Entity *)
          (*by_num_operators)
           [the_component->
            num_adt_operators()];

         // got the generics dictionary so get the total base
         // types dictionary

         if(by_num_generics->isIndex(the_component->
                                          num_generic_types())==TRUE)
          {
           by_num_total_outputs=(Dictionary *)(Entity *)
                (*by_num_generics)
                 [the_component->num_generic_types()];
```

```
if(by_num_total_outputs->
        isIndex(the_component->total_outputs())==TRUE)
       {
   by_num_total_inputs=(Dictionary *)(Entity *)
    (*by_num_total_outputs)
          [the_component->total_outputs()];


       if(by_num_total_inputs->
     isIndex(the_component->total_inputs())==TRUE)
         {
          leaf_dictionary=(Dictionary *)(Entity *)
                (*by_num_total_inputs)
                 [the_component->total_inputs()];

          // have to leaf dictionary

          leaf_dictionary->Remove(the_component->component_name());
          leaf_dictionary->putObject();
          if(leaf_dictionary->Cardinality()==0)
                {
                 by_num_total_inputs->
                  Remove(the_component->total_inputs());
                 by_num_total_inputs->putObject();
                 leaf_dictionary->deleteObject(TRUE);

                 if(by_num_total_inputs->Cardinality()==0)
                   {
                    by_num_total_outputs->
                          Remove(the_component->total_outputs());
                    by_num_total_outputs->putObject();
                    by_num_total_inputs->deleteObject(TRUE);

                    if(by_num_total_outputs->Cardinality()==0)
                          {
                           by_num_generics->
                            Remove(the_component->num_generic_types());
                           by_num_generics->putObject();
                           by_num_total_outputs->deleteObject(TRUE);

                           if(by_num_generics->Cardinality()==0)
                             {
                              by_num_operators->
                                    Remove(the_component->num_adt_operators());
                              by_num_operators->putObject();
                              by_num_generics->deleteObject();

                              if(by_num_operators->Cardinality()==0)
                                    {
                                     by_num_adts->
                                      Remove(the_component->num_adts());
                                     by_num_adts->putObject();
                                     by_num_operators->deleteObject();

                                    };
```

```
                                        };

                                    };

                                };


                            };

                        };
                    };
                };

            };

        };


};

SB_COMPONENT_MATCHES_DICTIONARY *SB_ADT_COMPONENT_LIBRARY::query(
                                                SB_ADT_COMPONENT *query_component)
{

  Dictionary *by_num_adts;
  Dictionary *by_num_operators;
  Dictionary *by_num_total_inputs;
  Dictionary *by_num_total_outputs;
  Dictionary *leaf_dictionary;

// ADDED
  Dictionary *BY_INPUT_SIGNATURE_DICTIONARY;
  Dictionary *BY_OUTPUT_SIGNATURE_DICTIONARY;

// ADDED -- class SIGNATURE comes from Signature.h
  SIGNATURE SIG;
  // class ADT_SIGNATURE comes from ADT_Signature.h
  ADT_SIGNATURE ADT_SIG;
  int QUERY_IN_SIG[ALL_REGIONS];
  int QUERY_OUT_SIG[ALL_REGIONS];
  int SB_IN_SIG[ALL_REGIONS];
  int SB_OUT_SIG[ALL_REGIONS];
  int REGION;
  int MATCH;



// MODIFIED
  SB_COMPONENT_MATCHES_DICTIONARY *query_result = new
     SB_COMPONENT_MATCHES_DICTIONARY();

// ADDED
  SB_COMPONENT_DICTIONARY *intermediate_query_result = new
      SB_COMPONENT_DICTIONARY();
```

```
// ADDED
  // Get the signatures from the query component.
  SIG.GET_SIGNATURES(QUERY_IN_SIG, QUERY_OUT_SIG);

  // in order for a match library must have at least as many adt's as
  // being requested

  by_num_adts=main_library();

  DictionaryIterator next_by_num_adt(by_num_adts,
                                     FALSE,
                                     query_component->num_adts());

  while(next_by_num_adt.moreData())
   {
    by_num_operators=(Dictionary *)(Entity *)next_by_num_adt();

    // components must have at least as many operators as the query
    DictionaryIterator next_by_num_operators(by_num_operators,
                              FALSE,
                 query_component->num_adt_operators());

    while(next_by_num_operators.moreData())
          {
           by_num_total_inputs=(Dictionary *)(Entity *)
                       next_by_num_operators();


           DictionaryIterator next_by_num_total_inputs(by_num_total_inputs,
                                                       FALSE,
                                              query_component->total_inputs());

           while(next_by_num_total_inputs.moreData())
            {
             by_num_total_outputs =
                           (Dictionary *)(Entity *)next_by_num_total_inputs();

             // Got the corresponding output dictionary so now go through
             // and get the INPUT SIGNATURES that are in dictionaries with
             // number of total output parameters greater than or equal
             // to the query component.

             DictionaryIterator NEXT_INPUT_SIGNATURE_DICTIONARY =
               DictionaryIterator(by_num_total_outputs,
                                  FALSE,
                                  query_component->
                                  total_outputs());


             while(NEXT_INPUT_SIGNATURE_DICTIONARY.moreData())
               {

                   BY_INPUT_SIGNATURE_DICTIONARY = (Dictionary *)(Entity *)
                     NEXT_INPUT_SIGNATURE_DICTIONARY();

               // Got an Input Signature dictionary.
```

```
// got an INPUT SIGNATURE dictionary so iterate over it
// for the OUTPUT SIGNATURE dictionaries. The dictionary indices
// (Array objects corresponding to Input Signatures) are returned.
// Only those output dictionaries will be examined where the
// INPUT SIGNATURE of the stored component matches the INPUT
// SIGNATURE of the query component.
DictionaryIterator NEXT_INPUT_SIGNATURE =
      DictionaryIterator(BY_INPUT_SIGNATURE_DICTIONARY, TRUE);

// Create a temporary set to store all matched signatures in
// for that particular INPUT SIGNATURE dictionary.
// Loop through all of the INPUT SIGNATURES for that dictionary.
Set *MATCHED_INPUT_SIGNATURES;
MATCHED_INPUT_SIGNATURES = new Set(OC_array);

while(NEXT_INPUT_SIGNATURE.moreData())
      {
       Array *INPUT_SIGNATURE = (Array *)(Entity *)
                                        NEXT_INPUT_SIGNATURE();

       // Load the Array object into a C++ array.
       for (REGION=1; REGION <=ALL_REGIONS; ++REGION)
        {
        // Note:  C++ array goes from 0 .. ALL_REGIONS -1
        SB_IN_SIG[REGION - 1] =
                *( (Integer *) (Entity *) (*INPUT_SIGNATURE) [REGION]);
        }
       // Check to see if we have matching input signatures.
       MATCH = SIG.MATCH_INPUT_SIGNATURES(QUERY_IN_SIG, SB_IN_SIG);
       if (MATCH == 1)
        {

         // Check for False Match.
         // If MATCH >= 0 then we have a valid match and the
         // value of MATCH is the type closeness degree.
         MATCH = SIG.CHECK_FALSE_MATCH(QUERY_IN_SIG, SB_IN_SIG);
         if (MATCH >= 0)
                {

                  // Add the software base component signature to the
                  // set of matched signatures.
                  MATCHED_INPUT_SIGNATURES->Insert(INPUT_SIGNATURE);

                };
        };
      }; // end iterate through Input Signatures


// Now, go through the set of matched signatures, and check
// the corresponding OUTPUT SIGNATURE dictionaries.
SetIterator NEXT_OUTPUT_SIGNATURE_DICTIONARY =
      SetIterator(MATCHED_INPUT_SIGNATURES);
```

```
while (NEXT_OUTPUT_SIGNATURE_DICTIONARY.moreData())
    {
    Array *SIGNATURE_DICTIONARY_INDEX = (Array *) (Entity *)
                                NEXT_OUTPUT_SIGNATURE_DICTIONARY();

    BY_OUTPUT_SIGNATURE_DICTIONARY =  (Dictionary *) (Entity *)
                        (*BY_INPUT_SIGNATURE_DICTIONARY)
                            [SIGNATURE_DICTIONARY_INDEX];


      // Got an Output Signature dictionary.

      // Create a temporary set to store all matched signatures in for
      // the specific OUTPUT SIGNATURE dictionary.
      // Loop through all of the OUTPUT SIGNATURES in that dictionary.
      Set *MATCHED_OUTPUT_SIGNATURES;
      MATCHED_OUTPUT_SIGNATURES = new Set(OC_array);

      // Only those output dictionaries will be examined where the
      // OUTPUT SIGNATURE of the stored component matches the
      // OUTPUT SIGNATURE of the query component.
      DictionaryIterator NEXT_OUTPUT_SIGNATURE =
            DictionaryIterator(BY_OUTPUT_SIGNATURE_DICTIONARY, TRUE);

      while(NEXT_OUTPUT_SIGNATURE.moreData())
        {
            Array *OUTPUT_SIGNATURE = (Array *)(Entity *)
                                        NEXT_OUTPUT_SIGNATURE();

            // Load the Array object into a C++ array.
            for (REGION=1; REGION <=ALL_REGIONS; ++REGION)
              // Note: C++ array goes from 0 .. ALL_REGIONS - 1
              SB_OUT_SIG[REGION - 1] = *( (Integer *) (Entity *)
                    (*OUTPUT_SIGNATURE) [REGION] );

            // Check to see if we have matching OUTPUT signatures.
            MATCH = SIG.MATCH_OUTPUT_SIGNATURES(
                                        QUERY_OUT_SIG, SB_OUT_SIG);
            if (MATCH == 1)
              {

                // Add the software base component signature to the
                // set of matched signatures.
                MATCHED_OUTPUT_SIGNATURES->Insert(OUTPUT_SIGNATURE);

              };
        } // end iterate through OUTPUT Signatures


      // Now, go through the set of matched signatures, and get
      // the corresponding leaf dictionaries.
      SetIterator next_leafs_dict =
            SetIterator(MATCHED_OUTPUT_SIGNATURES);

      while (next_leafs_dict.moreData())
            {
```

```
                        Array *SIGNATURE_DICT_INDEX = (Array *) (Entity *)
                                                        next_leafs_dict();

                        leaf_dictionary = (Dictionary *) (Entity *)
                          (*BY_OUTPUT_SIGNATURE_DICTIONARY) [SIGNATURE_DICT_INDEX];

                        // Got the corresponding leaf dictionary so now go through
                        // and get the components that are in the dictionary.

                        DictionaryIterator next_component=
                          DictionaryIterator(leaf_dictionary);

                        while(next_component.moreData())
                         {
                          SB_ADT_COMPONENT *the_component=
                           (SB_ADT_COMPONENT *)(Entity *)next_component();
                          // currently always true
                          if(query_component->filter(the_component)==TRUE)
                          {
                                  // put the components in the intermediate
                                  // query result dictionary

                                 intermediate_query_result->add(the_component);
                          };

                         };
                        };
                    };
                   };
            };
          };
  };
// Verify that the components match.  So far we have only matched
// aggregate signatures.  We now need to ensure that we can map
// individual query component ADT operators to matching software
// base component ADT operators.

// Create an array that holds the input and output signature of
// each query ADT operator.  Note:  it must be dynamic since we
// don't know its size until runtime.
SIGNATURES *QUERY_SIGNATURE;
QUERY_SIGNATURE = new
          SIGNATURES[query_component->num_adt_operators()];

// Create array to hold the number of input and output parameters of
// the query component's ADT operators.
PARAMETERS *QUERY_OPERATOR_IO;
QUERY_OPERATOR_IO = new
          PARAMETERS[query_component->num_adt_operators()];

// load query ADT operator signatures
ADT_SIG.LOAD_ADT_OPERATOR_SIGNATURES (QUERY_SIGNATURE,
              QUERY_OPERATOR_IO, query_component);

// iterate through the dictionary of components that matched on the
// basis of aggregate input and output signatures
```

```
DictionaryIterator next_component = intermediate_query_result->iterator();

while ( next_component.moreData() )
 {
  SB_ADT_COMPONENT *the_component =
    (SB_ADT_COMPONENT *) (Entity *) next_component();

  // Create an array of linked lists. Each array cell corresponds to
  // a query component ADT operator. Each linked list is made up of
  // all software base component ADT operators that match the query
  // component ADT operator. Note - it must be dynamic because we
  // don't know its size until runtime.

  LINK *QUERY_OPERATORS;
  QUERY_OPERATORS = new
            LINK[query_component->num_adt_operators()];

  // initialize all lists to null
  int INDEX;
  for (INDEX = 0; INDEX <query_component->num_adt_operators(); ++INDEX)
        QUERY_OPERATORS[INDEX] = NULL;

  ADT_SIG.BUILD_OPERATOR_MATCH_LIST (QUERY_SIGNATURE, QUERY_OPERATORS,
                        QUERY_OPERATOR_IO,
                                query_component, the_component);

  // find a valid mapping of query operators to sofware base component
  // operators
  MATCH = ADT_SIG.ADT_MATCH( QUERY_OPERATORS, query_component,
                                the_component);

  if (MATCH == 1)
   {
        // match was found. Store next_component in query_result dictionary.
        // Use its number of ADTs and ADT operators (Vs the number for the
        // query component) to order it in the dictionary. Because the
        // match found may actually be desired (semantics) or optimal,
        // signature type closeness is not used when ordering matched
        // ADT components.

        // put the components in the return result dictionary

        long TOTAL_CLOSENESS = 10000;

        TOTAL_CLOSENESS = TOTAL_CLOSENESS  +
                        ( (the_component->num_adts() -
                         query_component->num_adts()) * 1000)  +
                          (the_component->num_adt_operators() -
                           query_component->num_adt_operators());

        query_result->Insert(TOTAL_CLOSENESS, the_component);
  };

 };
// add code here for semantic matching interface
```

```
 return query_result;
};

void SB_ADT_COMPONENT_LIBRARY::list(ofstream& outstream)
{
 adt_component_dictionary()->printOn(outstream);
};
```

****
****
****

filename: sbao.cxx

```
//-------------------------------------------------------------------
//
//          J. K. MCDOWELL 23 AUG 91
//
//     Modified by Scott Dolgoff - 29 July 1993
//       Additions are preceded by "// ADDED".  Modifications are
//       preceded by "// MODIFIED".
//
//-------------------------------------------------------------------

// MODIFIED
//# include "sball.hxx"
//# include "sbextern.h"

// ADDED
#include "ADT_Signature.h"


SB_ADT_OPERATOR::SB_ADT_OPERATOR(APL *theAPL) :
SB_OPERATOR(theAPL)
{
};


SB_ADT_OPERATOR::SB_ADT_OPERATOR(char *id) :
SB_OPERATOR(id)
{

// ADDED (all code in constructor)
 Array *new_input_signature=new Array(OC_integer, ALL_REGIONS, 1);

 the_input_signature=new_input_signature->findTRef();


 Array *new_output_signature=new Array(OC_integer, ALL_REGIONS, 1);

 the_output_signature=new_output_signature->findTRef();
```

```
  // ADDED -- from the class ADT_SIGNATURE
  ADT_SIGNATURE ADT_SIG;
  int REGION;
  int REGION_VALUE;
  int IN_SIG[ALL_REGIONS];
  int OUT_SIG[ALL_REGIONS];

  ADT_SIG.GET_SIGNATURES_FROM_FILE (IN_SIG,  OUT_SIG, id);
  // read in input signature
  for (REGION = 1; REGION <= ALL_REGIONS; ++REGION)
    {
     REGION_VALUE = IN_SIG[REGION - 1];
    input_signature()->setElement(REGION, REGION_VALUE);
    }

  // read in output signature
  for (REGION = 1; REGION <= ALL_REGIONS; ++REGION)
    {
    REGION_VALUE = OUT_SIG[REGION - 1];
    output_signature()->setElement(REGION, REGION_VALUE);
    }

};


void SB_ADT_OPERATOR::Destroy(Boolean aborted)
{
// ADDED
  input_signature()->Destroy(aborted);
  output_signature()->Destroy(aborted);

  SB_OPERATOR::Destroy(aborted);
};

void SB_ADT_OPERATOR::deleteObject(Boolean deallocate)
{
// ADDED
  input_signature()->deleteObject(deallocate);
  output_signature()->deleteObject(deallocate);

  SB_OPERATOR::deleteObject(deallocate);
};

void SB_ADT_OPERATOR::putObject(Boolean deallocate)
{
// ADDED
  input_signature()->putObject(deallocate);
  output_signature()->putObject(deallocate);

  SB_OPERATOR::putObject(deallocate);
};

Type *SB_ADT_OPERATOR::getDirectType()
{
  return SB_ADT_OPERATOR_OType;
```

```
};


// ADDED
Array *SB_ADT_OPERATOR::input_signature()
{
 return (Array *)(Entity *)the_input_signature->Binding();
};


// ADDED
Array *SB_ADT_OPERATOR::output_signature()
{
 return (Array *)(Entity *)the_output_signature->Binding();
};



// ADDED
Array *SB_ADT_OPERATOR::get_input_signature()
{
 return input_signature();
};

// ADDED
Array *SB_ADT_OPERATOR::get_output_signature()
{
 return output_signature();
};

Boolean SB_ADT_OPERATOR::process_type_info(SB_ADT_COMPONENT *adt)
{
 // process types by checking local generic then adt.adt usage then
 // adt.generic usage before making it unrecognized.  This will update
 // the adt usage dictionaries as well

 // update all usage dictionaryies for inputs and outputs
 //
 // first go through all of the inputs
 DictionaryIterator next_input=input_attributes()->id_iterator();
 while(next_input.moreData())
   {
    SB_ID_DECL *this_decl=(SB_ID_DECL *)(Entity *)next_input();
    SB_TYPE_NAME *this_type_name=this_decl->type_name();
    // first see if this id_decl type is a generic
    if(generic_usage()->update(this_type_name)==FALSE)
          {
           // was not a generic type check the ADT generic list

           if(adt->generic_usage()->update(this_type_name)==FALSE)
             {
              // was not an adt generic so check the adt list
              if(adt->adt_usage()->update(this_type_name)==FALSE)
                    {
                     // was not an adt adt so put it in its local list
```

```
                    // based on whether or not it is recognized
                    if(this_type_name->recognized()==FALSE)
                      {
                       // was unrecognized so try to update
                       // the unrecognized list or add it to
                       // the list
                       if(unrecognized_type_usage()->
                              update(this_type_name)==FALSE)
                              {
                               // not yet in list so add it
                               unrecognized_type_usage()->
                                 add_type(this_type_name->id(),
                                          this_type_name);
                               // now update it for being used once
                               unrecognized_type_usage()->
                                 update(this_type_name);
                              };
                      }
                    else
                      {
                       // this type name is recognized so update
                       // or add it
                       if(recognized_type_usage()->
                              update(this_type_name)==FALSE)
                              {
                               // not yet in list so add it
                               recognized_type_usage()->
                                 add_type(this_type_name->id(),
                                          this_type_name);
                               // now update it for being used once
                               recognized_type_usage()->
                                 update(this_type_name);
                              };
                      };

                  };
            };
        };
 };
DictionaryIterator next_output=output_attributes()->id_iterator();
while(next_output.moreData())
 {
  SB_ID_DECL *this_decl=(SB_ID_DECL *)(Entity *)next_output();
  SB_TYPE_NAME *this_type_name=this_decl->type_name();

  // first see if this id_decl type is a generic
  if(generic_usage()->update(this_type_name)==FALSE)
        {

         // was not a generic type check the ADT generic list

         if(adt->generic_usage()->update(this_type_name)==FALSE)
           {
            // was not an adt generic so check the adt list
            if(adt->adt_usage()->update(this_type_name)==FALSE)
                  {
```

```
// was not an adt adt
// so put it in its local list
// based on whether or not it is recognized
if(this_type_name->recognized()==FALSE)
  {
   // was unrecognized so try to update
   // the unrecognized list or add it to
   // the list
   if(unrecognized_type_usage()->
          update(this_type_name)==FALSE)
          {
           // not yet in list so add it
           unrecognized_type_usage()->
             add_type(this_type_name->id(),
                      this_type_name);
           // now update it for being used once
           unrecognized_type_usage()->
             update(this_type_name);
          };

   // now update the adt list as well

   if(adt->unrecognized_type_usage()->
          update(this_type_name)==FALSE)
          {
           // not yet in list so add it
           adt->unrecognized_type_usage()->
             add_type(this_type_name->id(),
                      this_type_name);
           // now update it for being used once
           adt->unrecognized_type_usage()->
             update(this_type_name);
          };


  }
else
  {
   // this type name is recognized so update
   // or add it
   if(recognized_type_usage()->
          update(this_type_name)==FALSE)
          {
           // not yet in list so add it
           recognized_type_usage()->
             add_type(this_type_name->id(),
                      this_type_name);
           // now update for being used onece
           recognized_type_usage()->
             update(this_type_name);
          };
   // now update the adt usage list
   if(adt->recognized_type_usage()->
          update(this_type_name)==FALSE)
          {
```

```
                        // not yet in list so add it
                        adt->recognized_type_usage()->
                          add_type(this_type_name->id(),
                                   this_type_name);
                        // now update for being used onece
                        adt->recognized_type_usage()->
                          update(this_type_name);
                       };

                  };

               };
            };
         };
      };
   return TRUE;

};

*****************************************************************************
*****************************************************************************
*****************************************************************************
```

filename: sbcmd.cxx

```
// Added to the CAPS software base classes (developed by
// John Kelly McDowell in August, 1991) by Scott Dolgoff.
// SB_COMPONENT_MATCHES_DICTIONARY is only used as a temporary
// variable, so no puts, destroys or deletes are expected to
// be used.


# include "sball.hxx"
# include "sbextern.h"



SB_COMPONENT_MATCHES_DICTIONARY::SB_COMPONENT_MATCHES_DICTIONARY(APL *theAPL)
                                                    : Dictionary(theAPL)
{
};

SB_COMPONENT_MATCHES_DICTIONARY::SB_COMPONENT_MATCHES_DICTIONARY()
                                        : Dictionary (OC_integer, // KEY
                                                      SB_COMPONENT_OType,
                                                      TRUE,
                                                      TRUE)
{
  // Index values (keys) represent the closeness of the match with the
  // smaller index values representing the closest matches.


};

Type *SB_COMPONENT_MATCHES_DICTIONARY::getDirectType()
{
```

```
 return SB_COMPONENT_MATCHES_DICTIONARY_OType;
};

void SB_COMPONENT_MATCHES_DICTIONARY::Destroy(Boolean aborted)
{
 // first destroy all of the references in the dictionary

 DictionaryIterator next_component(this);

 while(next_component.moreData())
  {
   ((SB_COMPONENT *)(Entity *)next_component())->Destroy(aborted);
  };

 Dictionary::Destroy(aborted);

};

void SB_COMPONENT_MATCHES_DICTIONARY::deleteObject(Boolean deallocate)
{
 // first delete all of the references in the dictionary

 DictionaryIterator next_component(this);

 while(next_component.moreData())
  {
   ((SB_COMPONENT *)(Entity *)next_component())->deleteObject(FALSE);
  };

 Dictionary::deleteObject(deallocate);

};

void SB_COMPONENT_MATCHES_DICTIONARY::putObject(Boolean deallocate)
{
 // first put all of the references in the dictionary

 DictionaryIterator next_component(this);

 while(next_component.moreData())
  {
   ((SB_COMPONENT *)(Entity *)next_component())->putObject(deallocate);
  };

 Dictionary::putObject(deallocate);

};


void SB_COMPONENT_MATCHES_DICTIONARY::printOn(ofstream& outstream)
{
// Because the components are stored in the order of their closeness
// value (closeness = input type closeness + output type closeness +
// excess output parameters) they are automatically retrieved by
// the iterator in the desired order from closest match to farthest.
 DictionaryIterator next_component=
```

```
  DictionaryIterator(this);

 while(next_component.moreData())
  {
   int i;
   SB_COMPONENT *the_component=(SB_COMPONENT *)(Entity *)next_component();
   outstream << the_component->component_name();
   for(i=strlen(the_component->component_name());i < DEFAULT_NAME_SIZE; i++)
        {
         outstream << " ";
        };
   outstream << " ";
   char *informal_desc=(the_component->informal_description())->text();
   i=0;
   while(informal_desc[i]!=NULL && informal_desc[i]!='\n')
        {
         outstream << informal_desc[i];
         i++;
        };
   outstream << "\n";
  };
};

DictionaryIterator SB_COMPONENT_MATCHES_DICTIONARY::iterator()
{
        return DictionaryIterator(this);
};
```

****
****
****

filename: sbextern.h

```
//-------------------------------------------------------------
//
//          J. K. MCDOWELL 23 AUG 91
//
//    Modified by Scott Dolgoff - 29 July 1993
//      Additions are preceded by "// ADDED".  Modifications are
//      preceded by "// MODIFIED".
//
//-------------------------------------------------------------
//
// this file contains all of the external references to the
// ontos type schemas
//



extern Type *SB_LIBRARY_OType;
extern Type *SB_ADT_COMPONENT_LIBRARY_OType;
```

```
extern Type *SB_OPERATOR_COMPONENT_LIBRARY_OType;
extern Type *SB_COMPONENT_OType;
extern Type *SB_COMPONENT_DICTIONARY_OType;
extern Type *SB_KEYWORD_DICTIONARY_OType;
extern Type *SB_TEXT_OBJECT_OType;
extern Type *SB_ADT_COMPONENT_OType;
extern Type *SB_OPERATOR_COMPONENT_OType;
extern Type *SB_ADT_OPERATOR_OType;
extern Type *SB_ID_DECL_DICTIONARY_OType;
extern Type *SB_ID_DECL_OType;
extern Type *SB_TYPE_NAME_OType;
extern Type *SB_ADT_OPERATOR_DICTIONARY_OType;
extern Type *SB_EXCEPTION_DICTIONARY_OType;
extern Type *SB_TYPE_USAGE_OType;
extern Type *SB_TYPE_USAGE_DICTIONARY_OType;
extern Type *SB_RECOGNIZED_TYPES_OType;

extern SB_LIBRARY *SB_MAIN_LIBRARY;

// ADDED
extern Type *SB_COMPONENT_MATCHES_DICTIONARY_OType;

# define DEFAULT_NAME_SIZE 21

#define SB_UNRECOGNIZED_TYPE 1

#define SB_BASE_TYPE 2

#define SB_GENERIC_TYPE 3

#define SB_GENERIC_SUBPROGRAM 4

#define SB_GENERIC_VALUE 5

#define SB_ABSTRACT_DATA_TYPE 6

#define SB_ARRAY 7

#define SB_ARRAY_INDEX 8

#define SB_ARRAY_ELEMENT 9

// ADDED
#define ALL_REGIONS 24
#define INPUT_REGIONS 18
#define OUTPUT_REGIONS 17

// ADDED
// set value for a null pointer
#define NULL 0
```

****

****

****

filename: sbocl.cxx

```
/*  Original software for CAPS software base developed by
    John Kelly McDowell.


    Modified - 31 August 1993 by Scott Dolgoff
      Because modifications were fairly extensive in this module,
      refer to McDowell's thesis to see what the original code
      looked like.  Essentially, his by_num_unrecognized_dictionary has
      been removed from the complex data hierarchy.  In its place are
      two new dictionaries;  input_signature and output_signature.
      I have put my variables in UPPER CASE to help differentiate. */




# include "sball.hxx"
# include "sbextern.h"

// ADDED
#include "Signature.h"
#include <Set.h>
#include <Primitives.h>


SB_OPERATOR_COMPONENT_LIBRARY::SB_OPERATOR_COMPONENT_LIBRARY(APL *theAPL) :
Object(theAPL)
{
};


SB_OPERATOR_COMPONENT_LIBRARY::SB_OPERATOR_COMPONENT_LIBRARY() : Object()
{

 SB_COMPONENT_DICTIONARY *new_operator_component_dictionary=
  new SB_COMPONENT_DICTIONARY();

 the_operator_component_dictionary=
  new_operator_component_dictionary->findTRef();

 Dictionary *new_state_dictionary=new Dictionary(OC_integer,
                                                 OC_dictionary,
                                                 TRUE,
                                                 FALSE);

 the_state_dictionary=new_state_dictionary->findTRef();

 Dictionary *new_non_state_dictionary=new Dictionary(OC_integer,
                                                     OC_dictionary,
                                                     TRUE,
                                                     FALSE);
```

```
 the_non_state_dictionary=new_non_state_dictionary->findTRef();

};

void SB_OPERATOR_COMPONENT_LIBRARY::Destroy(Boolean aborted)
{
 operator_component_dictionary()->Destroy(aborted);

 // now must iterate through the multi-attribute query
 // dictionary tree

 Dictionary *leaf_dictionary;
 Dictionary *by_num_inputs_dictionary;
 Dictionary *by_num_unrecognized_dictionary;
 Dictionary *by_num_outputs_dictionary;

 by_num_inputs_dictionary=state_dictionary();

 DictionaryIterator next_input_dictionary=
  DictionaryIterator(by_num_inputs_dictionary);

 while(next_input_dictionary.moreData())
  {
   by_num_unrecognized_dictionary=
         (Dictionary *)(Entity *)next_input_dictionary();


   DictionaryIterator next_outputs_dict(by_num_unrecognized_dictionary);

   while(next_outputs_dict.moreData())
         {
          by_num_outputs_dictionary=(Dictionary *)(Entity *)
           next_outputs_dict();


          DictionaryIterator next_leaf_dict(by_num_outputs_dictionary);

          while(next_leaf_dict.moreData())
           {
            leaf_dictionary=(Dictionary *)(Entity *)
                  next_leaf_dict();

            DictionaryIterator next_component(leaf_dictionary);


            while(next_component.moreData())
                  {
                   ((SB_OPERATOR_COMPONENT *)(Entity *)next_component())->
                    Destroy(aborted);
                  };
            leaf_dictionary->Destroy(aborted);
           };
          by_num_outputs_dictionary->Destroy(aborted);
         };
   by_num_unrecognized_dictionary->Destroy(aborted);
```

```
  };
 by_num_inputs_dictionary->Destroy(aborted);


 by_num_inputs_dictionary=non_state_dictionary();


 next_input_dictionary=
  DictionaryIterator(by_num_inputs_dictionary);

 while(next_input_dictionary.moreData())
  {
   by_num_unrecognized_dictionary=
        (Dictionary *)(Entity *)next_input_dictionary();


   DictionaryIterator next_outputs_dict(by_num_unrecognized_dictionary);
   while(next_outputs_dict.moreData())
        {
         by_num_outputs_dictionary=(Dictionary *)(Entity *)
          next_outputs_dict();


         DictionaryIterator next_leaf_dict(by_num_outputs_dictionary);

         while(next_leaf_dict.moreData())
          {
           leaf_dictionary=(Dictionary *)(Entity *)
                next_leaf_dict();

           DictionaryIterator next_component(leaf_dictionary);


           while(next_component.moreData())
                {
                 ((SB_OPERATOR_COMPONENT *)(Entity *)next_component())->
                   Destroy(aborted);
                };
           leaf_dictionary->Destroy(aborted);
          };
         by_num_outputs_dictionary->Destroy(aborted);
        };
   by_num_unrecognized_dictionary->Destroy(aborted);
  };
 by_num_inputs_dictionary->Destroy(aborted);

 delete the_operator_component_dictionary;
 delete the_state_dictionary;
 delete the_non_state_dictionary;

};

void SB_OPERATOR_COMPONENT_LIBRARY::deleteObject(Boolean deallocate)
{
 operator_component_dictionary()->deleteObject(deallocate);
```

```
Dictionary *leaf_dictionary;
Dictionary *by_num_inputs_dictionary;
Dictionary *by_num_unrecognized_dictionary;
Dictionary *by_num_outputs_dictionary;

by_num_inputs_dictionary=state_dictionary();



DictionaryIterator next_input_dictionary(by_num_inputs_dictionary);

while(next_input_dictionary.moreData())
 {
  by_num_unrecognized_dictionary=
        (Dictionary *)(Entity *)next_input_dictionary();


  DictionaryIterator next_outputs_dict(by_num_unrecognized_dictionary);

  while(next_outputs_dict.moreData())
        {
         by_num_outputs_dictionary=(Dictionary *)(Entity *)
          next_outputs_dict();


         DictionaryIterator next_leaf_dict(by_num_outputs_dictionary);

         while(next_leaf_dict.moreData())
          {
           leaf_dictionary=(Dictionary *)(Entity *)
                 next_leaf_dict();

           DictionaryIterator next_component(leaf_dictionary);


           while(next_component.moreData())
                 {
                  ((SB_OPERATOR_COMPONENT *)(Entity *)next_component())->
                    deleteObject(FALSE);
                 };
           leaf_dictionary->deleteObject(FALSE);
          };
         by_num_outputs_dictionary->deleteObject(FALSE);
        };
  by_num_unrecognized_dictionary->deleteObject(FALSE);
 };
by_num_inputs_dictionary->deleteObject(FALSE);


by_num_inputs_dictionary=non_state_dictionary();


next_input_dictionary=
 DictionaryIterator(by_num_inputs_dictionary);

while(next_input_dictionary.moreData())
```

```
    {
     by_num_unrecognized_dictionary=
          (Dictionary *)(Entity *)next_input_dictionary();


     DictionaryIterator next_outputs_dict(by_num_unrecognized_dictionary);
     while(next_outputs_dict.moreData())
          {
           by_num_outputs_dictionary=(Dictionary *)(Entity *)
            next_outputs_dict();


           DictionaryIterator next_leaf_dict(by_num_outputs_dictionary);

           while(next_leaf_dict.moreData())
            {
             leaf_dictionary=(Dictionary *)(Entity *)
                  next_leaf_dict();

             DictionaryIterator next_component(leaf_dictionary);


             while(next_component.moreData())
                  {
                   ((SB_OPERATOR_COMPONENT *)(Entity *)next_component())->
                    deleteObject(FALSE);
                  };
             leaf_dictionary->deleteObject(FALSE);
            };
           by_num_outputs_dictionary->deleteObject(FALSE);
          };
     by_num_unrecognized_dictionary->deleteObject(FALSE);
    };
  by_num_inputs_dictionary->deleteObject(FALSE);

  Object::deleteObject(deallocate);

};

void SB_OPERATOR_COMPONENT_LIBRARY::putObject(Boolean deallocate)
{
  operator_component_dictionary()->putObject(deallocate);

  state_dictionary()->putObject(deallocate);
  non_state_dictionary()->putObject(deallocate);
  Object::putObject(deallocate);

};



Type *SB_OPERATOR_COMPONENT_LIBRARY::getDirectType()
{
  return SB_OPERATOR_COMPONENT_LIBRARY_OType;
};
```

```
SB_COMPONENT_DICTIONARY
*SB_OPERATOR_COMPONENT_LIBRARY::operator_component_dictionary()
{
 return (SB_COMPONENT_DICTIONARY *)(Entity *)
   the_operator_component_dictionary->Binding();
};

Dictionary *SB_OPERATOR_COMPONENT_LIBRARY::state_dictionary()
{
 return (Dictionary *)(Entity *)the_state_dictionary->Binding();
};

Dictionary *SB_OPERATOR_COMPONENT_LIBRARY::non_state_dictionary()
{
 return (Dictionary *)(Entity *)the_non_state_dictionary->Binding();
};

// ADDED sig_file as a parameter to "add" function
Boolean SB_OPERATOR_COMPONENT_LIBRARY::add(SB_OPERATOR_COMPONENT *new_component,
                                           char *SIG_FILE)
{


 Boolean return_flag=TRUE;
 Dictionary *leaf_dictionary;
 Dictionary *by_num_inputs_dictionary;

// ADDED
 Boolean NEW_INPUT_SIGNATURE_ARRAY = FALSE;
 Boolean NEW_OUTPUT_SIGNATURE_ARRAY = FALSE;
 Dictionary *BY_INPUT_SIGNATURE_DICTIONARY;
 Dictionary *BY_OUTPUT_SIGNATURE_DICTIONARY;

// ADDED -- class SIGNATURE comes from Signature.h
 SIGNATURE SIG;

 Dictionary *by_num_outputs_dictionary;

 operator_component_dictionary()->add(new_component);

 operator_component_dictionary()->Dictionary::putObject();

 // insert into the component dictionary was successfull
 // so insert it into the library

 if(new_component->states()==TRUE)
   {
    by_num_inputs_dictionary=state_dictionary();
   }
 else
   {
    by_num_inputs_dictionary=non_state_dictionary();
   };

 // have correct state dictionary so now find correct
 // input dictionary
```

```
if(by_num_inputs_dictionary->isIndex(new_component->num_inputs())==TRUE)
  {
    by_num_outputs_dictionary=(Dictionary *)
          (Entity *)(*by_num_inputs_dictionary)
           [new_component->num_inputs()];
  }
else
  {
    by_num_outputs_dictionary=new Dictionary(OC_integer,
                                             OC_dictionary,
                                             TRUE,
                                             FALSE);


    by_num_inputs_dictionary->Insert(new_component->
                                     num_inputs(),
                                     by_num_outputs_dictionary);

  };



// Have correct  dictionary for the component's number of outputs.
// Now get correct INPUT_SIGNATURE dictionary.

if(by_num_outputs_dictionary->isIndex(new_component->num_outputs())==TRUE)
  {
   BY_INPUT_SIGNATURE_DICTIONARY=(Dictionary *)
          (Entity *)(*by_num_outputs_dictionary)
           [new_component->num_outputs()];
  }
else
  {

    BY_INPUT_SIGNATURE_DICTIONARY=new Dictionary(OC_array,
                                                 OC_dictionary,
                                                 FALSE,
                                                 FALSE);

    by_num_outputs_dictionary->Insert(new_component->
                                     num_outputs(),
                                     BY_INPUT_SIGNATURE_DICTIONARY);

  };


// first get the SIGNATURE value from SIG_FILE

ifstream SIGNATURE(SIG_FILE);

int REGION, REGION_VALUE;

Array *INPUT_SIGNATURE_ARRAY = new Array (OC_integer, ALL_REGIONS, 1);

for (REGION=1; REGION <= ALL_REGIONS; ++REGION)
```

```
  {
   SIGNATURE >> REGION_VALUE;
   INPUT_SIGNATURE_ARRAY->setElement(REGION, REGION_VALUE);
  }

// Can't use isIndex to see if the new signature is already an
// index in the dictionary because each index is an object
// with a unique id. Therefore, must iterate through indices
// and see if one isSimilar (i.e. of equal value) to the
// INPUT_SIGNATURE_ARRAY.
Array *FOUND_INDEX_ARRAY;
int NOT_DONE = 1;
int INDEX_FOUND = 0;

DictionaryIterator NEXT_INPUT_SIGNATURE =
        DictionaryIterator(BY_INPUT_SIGNATURE_DICTIONARY, TRUE);

while( NEXT_INPUT_SIGNATURE.moreData() && NOT_DONE )
 {
  FOUND_INDEX_ARRAY = (Array *)(Entity *)
                                    NEXT_INPUT_SIGNATURE();

  // compare the two signature arrays
  if (FOUND_INDEX_ARRAY->isSimilar(INPUT_SIGNATURE_ARRAY))
    {
        NOT_DONE = 0;
        INDEX_FOUND = 1;
    };
 }

if(INDEX_FOUND)
  {

   BY_OUTPUT_SIGNATURE_DICTIONARY=(Dictionary *)(Entity *)
        (*BY_INPUT_SIGNATURE_DICTIONARY) [FOUND_INDEX_ARRAY];

  }
 else
  {

   BY_OUTPUT_SIGNATURE_DICTIONARY=new Dictionary(OC_array,
                                                 OC_dictionary,
                                                 FALSE,
                                                 FALSE);

   NEW_INPUT_SIGNATURE_ARRAY = TRUE;

   BY_INPUT_SIGNATURE_DICTIONARY->Insert(INPUT_SIGNATURE_ARRAY,
                                         BY_OUTPUT_SIGNATURE_DICTIONARY);

  };


// get OUTPUT_SIGNATURE from SIG_FILE
```

```
Array *OUTPUT_SIGNATURE_ARRAY = new Array (OC_integer, ALL_REGIONS, 1);

for (REGION=1; REGION <= ALL_REGIONS; ++REGION)
  {
   SIGNATURE >> REGION_VALUE;
   OUTPUT_SIGNATURE_ARRAY->setElement(REGION, REGION_VALUE);
  }


////////////////
// Can't use isIndex to see if the new signature is already an
// index in the dictionary because each index is an object
// with a unique id.  Therefore, must iterate through indices
// and see if one isSimilar (i.e. of equal value) to the
// OUTPUT_SIGNATURE_ARRAY.
NOT_DONE = 1;
INDEX_FOUND = 0;

DictionaryIterator NEXT_OUTPUT_SIGNATURE =
        DictionaryIterator(BY_OUTPUT_SIGNATURE_DICTIONARY, TRUE);

while( NEXT_OUTPUT_SIGNATURE.moreData() && NOT_DONE )
 {
  FOUND_INDEX_ARRAY = (Array *)(Entity *)
                                        NEXT_OUTPUT_SIGNATURE();

  // compare the two signature arrays
  if (FOUND_INDEX_ARRAY->isSimilar(OUTPUT_SIGNATURE_ARRAY))
    {
          NOT_DONE = 0;
          INDEX_FOUND = 1;
    };
 }

if(INDEX_FOUND)
  {

   leaf_dictionary=(Dictionary *)(Entity *)
         (*BY_OUTPUT_SIGNATURE_DICTIONARY) [FOUND_INDEX_ARRAY];

  }

else
  {
   leaf_dictionary=new Dictionary(OC_string,
                                        SB_COMPONENT_OType,
                                        FALSE,
                                        FALSE);

   NEW_OUTPUT_SIGNATURE_ARRAY = TRUE;

   BY_OUTPUT_SIGNATURE_DICTIONARY->Insert(OUTPUT_SIGNATURE_ARRAY,
                                                leaf_dictionary);


  };
```

```
 // have the leaf dictionary so now insert the component into it
 leaf_dictionary->Insert(new_component->component_name(),
                                 new_component);


 leaf_dictionary->putObject();
 by_num_inputs_dictionary->putObject();
 by_num_outputs_dictionary->putObject();
 BY_INPUT_SIGNATURE_DICTIONARY->putObject();
 BY_OUTPUT_SIGNATURE_DICTIONARY->putObject();
 if (NEW_INPUT_SIGNATURE_ARRAY)
  INPUT_SIGNATURE_ARRAY->putObject();
 if (NEW_OUTPUT_SIGNATURE_ARRAY)
  OUTPUT_SIGNATURE_ARRAY->putObject();

 return return_flag;
};



void SB_OPERATOR_COMPONENT_LIBRARY::
delete_component(SB_OPERATOR_COMPONENT *the_component)
{

 Dictionary *leaf_dictionary;
 Dictionary *by_num_inputs_dictionary;
 Dictionary *by_num_unrecognized_dictionary;
 Dictionary *by_num_outputs_dictionary;

 operator_component_dictionary()->
  Remove(the_component->component_name());
 operator_component_dictionary()->putObject();


 if(the_component->states()==TRUE)
  {
   by_num_inputs_dictionary=state_dictionary();
  }
 else
  {
   by_num_inputs_dictionary=non_state_dictionary();
  };

 // have correct state dictionary so now find correct
 // input dictionary

 if(by_num_inputs_dictionary->isIndex(the_component->num_inputs())==TRUE)
  {
   by_num_unrecognized_dictionary=(Dictionary *)
          (Entity *)(*by_num_inputs_dictionary)
           [the_component->num_inputs()];
```

```
    // got the unrecognized dictionary
    // use num generics since for a library unit all unrecognized types
    // must be generics

    if(by_num_unrecognized_dictionary->
            isIndex(the_component->num_generic_types())==TRUE)
           {

            by_num_outputs_dictionary=(Dictionary *)(Entity *)
             (*by_num_unrecognized_dictionary)
              [the_component->num_generic_types()];

       if(by_num_outputs_dictionary->
              isIndex(the_component->num_outputs())==TRUE)
              {
               leaf_dictionary=(Dictionary *)(Entity *)
                     (*by_num_outputs_dictionary)
                      [the_component->num_outputs()];

               // have to leaf dictionary
               leaf_dictionary->Remove(the_component->component_name());
               leaf_dictionary->putObject();
               if(leaf_dictionary->Cardinality()==0)
                     {
                      by_num_outputs_dictionary->
                       Remove(the_component->num_outputs());
                      by_num_outputs_dictionary->putObject();

                      if(by_num_outputs_dictionary->Cardinality()==0);
                      {
                       by_num_unrecognized_dictionary->
                        Remove(the_component->num_generic_types());
                       by_num_unrecognized_dictionary->putObject();

                       if(by_num_unrecognized_dictionary->Cardinality()==0)
                         {
                               by_num_inputs_dictionary->
                                Remove(the_component->num_inputs());
                               by_num_inputs_dictionary->putObject();
                               by_num_unrecognized_dictionary->deleteObject(TRUE);
                         };
                       by_num_outputs_dictionary->deleteObject(TRUE);
                      };
                      leaf_dictionary->deleteObject(TRUE);
                     };
              };
           };
    };

};


SB_COMPONENT_MATCHES_DICTIONARY *SB_OPERATOR_COMPONENT_LIBRARY::
query(SB_OPERATOR_COMPONENT *query_component)
{
// MODIFIED
```

```
SB_COMPONENT_MATCHES_DICTIONARY *query_result = new
   SB_COMPONENT_MATCHES_DICTIONARY();

// ADDED
 // OC_array corresponds to a particular signature value.
 // OC_integer is type closeness value of that signature.
 Dictionary *INPUT_CLOSENESS_DICTIONARY = new Dictionary(OC_array,
                                                         OC_integer,
                                                         FALSE,
                                                         FALSE);


 Dictionary *leaf_dictionary;
 Dictionary *by_num_inputs_dictionary;
 Dictionary *by_num_outputs_dictionary;

 // ADDED
 Dictionary *BY_INPUT_SIGNATURE_DICTIONARY;
 Dictionary *BY_OUTPUT_SIGNATURE_DICTIONARY;

 // ADDED -- class SIGNATURE comes from Signature.h
 SIGNATURE SIG;
 int QUERY_IN_SIG[ALL_REGIONS];
 int QUERY_OUT_SIG[ALL_REGIONS];
 int SB_IN_SIG[ALL_REGIONS];
 int SB_OUT_SIG[ALL_REGIONS];
 int REGION;
 int MATCH;


  // Get the signatures from the query component.
  SIG.GET_SIGNATURES (QUERY_IN_SIG, QUERY_OUT_SIG);

 // get the correct state_dictionary to start the query

 if(query_component->states()==TRUE)
   {
    by_num_inputs_dictionary=state_dictionary();
   }
 else
   {
    by_num_inputs_dictionary=non_state_dictionary();
   };

 // have correct state dictionary so now find correct
 // input dictionary

 // inputs must match exactly so only get one dictionary

 if((by_num_inputs_dictionary->isIndex(query_component->num_inputs()))==TRUE)
   {
          by_num_outputs_dictionary =
          (Dictionary *)(Entity *)(*by_num_inputs_dictionary)
            [query_component->num_inputs()];
```

```
// Got the corresponding output dictionary so now go through and
// get the INPUT SIGNATURES that are in dictionaries with
// number of output parameters greater than or equal to the
// query component.


DictionaryIterator NEXT_INPUT_SIGNATURE_DICTIONARY =
   DictionaryIterator(by_num_outputs_dictionary,
                           FALSE,
                           query_component->
                           num_outputs());


while(NEXT_INPUT_SIGNATURE_DICTIONARY.moreData())
 {
BY_INPUT_SIGNATURE_DICTIONARY = (Dictionary *)(Entity *)
            NEXT_INPUT_SIGNATURE_DICTIONARY();

     // Got an Input Signature dictionary.


// got an INPUT SIGNATURE dictionary so iterate over it for the
// OUTPUT SIGNATURE DICTIONARIES. The dictionary indices which are
// Array objects corresponding to Input Signatures are returned.
// Only those OUTPUT SIGNATURE DICTIONARIES will be examined where
// the INPUT SIGNATURE of the stored component matches the INPUT
// SIGNATURE of the query component.
DictionaryIterator NEXT_INPUT_SIGNATURE =
  DictionaryIterator(BY_INPUT_SIGNATURE_DICTIONARY, TRUE);

//  Create a temporary set to store all matched signatures in
//  for that particular INPUT SIGNATURE dictionary.
//  Loop through all of the INPUT SIGNATURES for that dictionary.
Set *MATCHED_INPUT_SIGNATURES;
MATCHED_INPUT_SIGNATURES = new Set(OC_array);

while(NEXT_INPUT_SIGNATURE.moreData())
  {
   Array *INPUT_SIGNATURE = (Array *)(Entity *)
                                        NEXT_INPUT_SIGNATURE();

   // Load the Array object into a C++ array.
   for (REGION=1; REGION <=ALL_REGIONS; ++REGION)
    {
    // Note:  C++ array goes from 0 .. ALL_REGIONS -1
    SB_IN_SIG[REGION - 1] =
             *( (Integer *) (Entity *) (*INPUT_SIGNATURE) [REGION]);
    }

   // Check to see if we have matching input signatures.
   MATCH = SIG.MATCH_INPUT_SIGNATURES(QUERY_IN_SIG, SB_IN_SIG);
   if (MATCH == 1)
     {

          // Check for False Match.
          // If MATCH >= 0 then we have a valid match and the
```

```
            // value of MATCH is the type closeness degree.
            MATCH = SIG.CHECK_FALSE_MATCH(QUERY_IN_SIG, SB_IN_SIG);
            if (MATCH >= 0)
              {

                // Add the software base component signature to the
                // set of matched signatures.
                MATCHED_INPUT_SIGNATURES->Insert(INPUT_SIGNATURE);


                // Add type closeness information
                INPUT_CLOSENESS_DICTIONARY->Insert(INPUT_SIGNATURE,
                                                   MATCH);
              };
    };
  }; // end iterate through Input Signatures


// Now, go through the set of matched signatures, and check
// the corresponding OUTPUT SIGNATURE dictionaries.
SetIterator NEXT_OUTPUT_SIGNATURE_DICTIONARY =
 SetIterator(MATCHED_INPUT_SIGNATURES);

while (NEXT_OUTPUT_SIGNATURE_DICTIONARY.moreData())
 {
 Array *SIGNATURE_DICTIONARY_INDEX = (Array *) (Entity *)
                              NEXT_OUTPUT_SIGNATURE_DICTIONARY();

 BY_OUTPUT_SIGNATURE_DICTIONARY =  (Dictionary *) (Entity *)
   (*BY_INPUT_SIGNATURE_DICTIONARY) [SIGNATURE_DICTIONARY_INDEX];


    // Got an Output Signature dictionary.

    //  Create a temporary set to store all matched signatures in for
    //  the specific OUTPUT SIGNATURE dictionary.
    //  Loop through all of the OUTPUT SIGNATURES in that dictionary.
    Set *MATCHED_OUTPUT_SIGNATURES;
    MATCHED_OUTPUT_SIGNATURES = new Set(OC_array);

    // Only those output dictionaries will be examined where the
    // OUTPUT SIGNATURE of the stored component matches the
    // OUTPUT SIGNATURE of the query component.
    DictionaryIterator NEXT_OUTPUT_SIGNATURE =
            DictionaryIterator(BY_OUTPUT_SIGNATURE_DICTIONARY, TRUE);

    // Temporary dictionary to store closeness information in.
    // OC_array corresponds to a particular signature value.
    // OC_integer is a combination of the type closeness value of
    // that signature and the number of output parameters that
    // software base component has in excess of the query component.
    Dictionary *OUTPUT_CLOSENESS_DICTIONARY =
                                              new Dictionary(OC_array,
                                                             OC_integer,
                                                             FALSE,
                                                             FALSE);
```

```
while(NEXT_OUTPUT_SIGNATURE.moreData())
    {
     Array *OUTPUT_SIGNATURE = (Array *)(Entity *)
                                        NEXT_OUTPUT_SIGNATURE();

     // Load the Array object into a C++ array.
     for (REGION=1; REGION <=ALL_REGIONS; ++REGION)
      // Note: C++ array goes from 0 .. ALL_REGIONS - 1
      SB_OUT_SIG[REGION - 1] = *( (Integer *) (Entity *)
              (*OUTPUT_SIGNATURE) [REGION] );

     // Check to see if we have matching OUTPUT signatures.
     MATCH = SIG.MATCH_OUTPUT_SIGNATURES(
                                        QUERY_OUT_SIG, SB_OUT_SIG);
     if (MATCH == 1)
       {

        // Measure type closeness degree.
        MATCH = SIG.CALC_OUT_CLOSE_DEGREE
                        (QUERY_OUT_SIG, SB_OUT_SIG);

        // Add the software base component signature to the
        // set of matched signatures.
        MATCHED_OUTPUT_SIGNATURES->Insert(OUTPUT_SIGNATURE);

        // Add closeness information
        OUTPUT_CLOSENESS_DICTIONARY->Insert(OUTPUT_SIGNATURE,
                                                MATCH);

       };
    } // end iterate through OUTPUT Signatures


   // Now, go through the set of matched signatures, and get
   // the corresponding leaf dictionaries.
   SetIterator next_leafs_dict =
     SetIterator(MATCHED_OUTPUT_SIGNATURES);

   while (next_leafs_dict.moreData())
    {
     Array *SIGNATURE_DICT_INDEX = (Array *) (Entity *)
                                                next_leafs_dict();

     leaf_dictionary = (Dictionary *) (Entity *)
      (*BY_OUTPUT_SIGNATURE_DICTIONARY) [SIGNATURE_DICT_INDEX];

     // Got the corresponding leaf dictionary so now go through
     // and get the components that are in the dictionary.

     DictionaryIterator next_component=
        DictionaryIterator(leaf_dictionary);

     while(next_component.moreData())
      {
       SB_OPERATOR_COMPONENT *the_component=
```

```
                    (SB_OPERATOR_COMPONENT *)(Entity *)next_component();
            // currently always true
                    if(query_component->filter(the_component)==TRUE)
                    {
                        // put the components in the
                        // return result dictionary

                        long TOTAL_CLOSENESS = 10000;
                        TOTAL_CLOSENESS = TOTAL_CLOSENESS          +
                        ( (the_component->num_outputs() -
                            query_component->num_outputs()) * 1000)  +
                            INPUT_CLOSENESS_DICTIONARY->
                            getIntegerElement(
                                    SIGNATURE_DICTIONARY_INDEX)   +
                            OUTPUT_CLOSENESS_DICTIONARY->
                            getIntegerElement(
                                    SIGNATURE_DICT_INDEX);


                        query_result->Insert(TOTAL_CLOSENESS,the_component);
                    };

                  };
                 };
          };
        };
  };


  // add code to interface to semantic check routine here

 return query_result;

};



void SB_OPERATOR_COMPONENT_LIBRARY::list(ofstream& outstream)
{
 operator_component_dictionary()->printOn(outstream);
};
```

APPENDIX G - MODIFICATIONS TO ORIGINAL CAPS GRAPHICAL USER INTERFACE TAE SOURCE CODE

The TAE source code listed below contains Ada code for one new panel developed for this thesis (pan_mapping_s.a, pan_mapping_b.a) and all significant modifications to TAE source code originally revised by Dogan Ozdemir [Ozde92].

```
-- *** TAE Plus Code Generator version V5.1
-- *** File      : global_s.a
-- *** Generated  : May 21 16:12:31 1992
-- *** Revised by : Dogan Ozdemir
-- ******************************************************************************
-- *
-- *   Global                    -- Package SPEC
-- *
-- ******************************************************************************

with X_Windows;
with Text_IO;
with TAE;
with SYSTEM;
use TAE,SYSTEM,Text_IO;

-- ADDED
with CREATE_OPERATOR_PARAMETER_FILES_PKG, PARAMETER_MAPPING_PKG,
   PARAMETER_LIST_PKG;
use  CREATE_OPERATOR_PARAMETER_FILES_PKG, PARAMETER_MAPPING_PKG,
   PARAMETER_LIST_PKG;
with PSDL_ID_PKG;
use  PSDL_ID_PKG;

package Global is

--| PURPOSE:
--| This package is automatically "with"ed in to e:ch panel package body.
--| You can insert global variables here.
--|
--| REGENERATED:
--| This file is generated only once.
--|

 package Taefloat_IO is new Text_IO.Float_IO (TAE.Taefloat);

 Default_Display_Id : X_Windows.Display;
```

```
--ADDED

-- MODIFIED/ADDED
 library          : String (1..10):= ('A','d','a',others=>' ');
 COMPONENT_IS_OPERATOR : BOOLEAN;

 lib_to_delete     : String (1..10):= (others=>' ');
 path              : String(1..80):= (others=>' ');
 proto_prefix      : String(1..80):= (others=>' ');
 Query_psdl        : String(1..80):= (others=>' ');
 Directory         : String(1..80):= (others=>' ');
 kwquery_outfile   : String(1..15):="kwquery_outfile";
 query_outfile     : String(1..13):="query_outfile";
 component         : String(1..80):= (others=>' ');
 directory_array   : String (1..27):= (others=>' ');
 directory_file_name: String(1..14) := "directory_file";
 lib_vec           : s_vector(1..20):= (others=> new STRING(1..10));
 file_vec          : s_vector(1..200):= (others=> new STRING(1..80));
 Is_a_directory    : Boolean:=FALSE;
 Upper_directory   : Boolean:=FALSE;
 Component_add     : Boolean:=FALSE;
 Component_update  : Boolean:=FALSE;
 Query             : Boolean:=FALSE;
 current_directory : array(1..20) of String(1..80);
 directory_file    : file_type;
 cur_dir_index     : integer:=1;
 lib_count         : integer:=0;
 num_of_comp       : integer:=1;

-- MODIFIED
 com               : constant String:=
                     "/n/sun51/work/dolgoff/caps/src/software_base/sb ";

 parse             : constant String:="/n/sun54/work/caps92/src/software_base/integrate/";
 gui_directory     : constant String:="/n/sun54/work/caps92/src/user_interface/sb_interface/";


--ADDED
 OPERATOR_FILE_NAME : constant STRING := "operator_psdl_spec.txt";
 TYPE_FILE_NAME     : constant STRING := "type_psdl_spec.txt";
 PROTOTYPE_FILE_NAME : constant STRING := "prototype_psdl_spec.txt";
 com2               : constant String:=
                      "/n/sun51/work/dolgoff/caps/src/software_base/opsig ";
 com3               : constant String:=
                      "/n/sun51/work/dolgoff/caps/src/software_base/adtsig ";
 THE_STATUS             : COMPONENT_STATUS;
 HAS_GENERICS           : BOOLEAN;
 QC_IN_PARAMS,
 QC_OUT_PARAMS,
 SBC_IN_PARAMS,
 SBC_OUT_PARAMS,
 GEN_PARAMS             : PARAMETERS;
 QC_IN_COUNT,
 QC_OUT_COUNT,
 SBC_IN_COUNT,
```

```
SBC_OUT_COUNT,
GEN_COUNT          : INTEGER;
SBC_NAME,
QC_NAME            : PSDL_ID_PKG.PSDL_ID;

-- ..........................................................
-- .
-- .   Application_Done            -- Subprogram SPEC
-- .
-- ..........................................................

function Application_Done
 return Boolean;

--| PURPOSE:
--| This function returns true if a "quit" event handler has called
--| Set_Application_Done, otherwise it returns false.



-- ..........................................................
-- .
-- .   Set_Application_Done         -- Subprogram SPEC
-- .
-- ..........................................................

procedure Set_Application_Done;

--| PURPOSE:
--| This procedure can be used by an event handler, typically a "quit"
--| button, to signal the end of the application.

--ADDED

--..........................................................
-- .
-- .   system_call                  -- Subprogram SPEC
-- .
-- ..........................................................

procedure system_call(command : STRING);

--| PURPOSE:
--| This procedure is used to make unix system calls from within the program.

--..........................................................
-- .
-- .   strlen                       -- Subprogram SPEC
-- .
-- ..........................................................

procedure strlen(s: in String; n: in out Integer);

--| PURPOSE:
--| This procedure is used to get the length of strings.
```

```
--.....
-- .
-- .   list_directory              -- Subprogram SPEC
-- .
-- .....

procedure list_directory(file    :in out file_type;
                                                    file_name:in out string;
                                                    file_vec :in out s_vector;
                                                    I :in out integer);

--| PURPOSE:
--| This procedure is used to obtain the contents of unix directory structures.

--.....
-- .
-- .   list_components              -- Subprogram SPEC
-- .
-- .....

procedure list_components(file    :in out file_type;
                                                    file_name:in out string;
                                                    file_vec :in out s_vector;
                                                    I :in out integer);

--| PURPOSE:
--| This procedure is used to read the component list from a text file and
--| fill them into a s_vector structure to be displayed in a TAE panel.

--.....
-- .
-- .   read_directory              -- Subprogram SPEC
-- .
-- .....

procedure read_directory(file    :in out file_type;
                                                    file_name:in out string;
                                                    dir_name :in out string);

--| PURPOSE:
--| This procedure is used to read the name of the current directory and
--| to get the path from a text file.

--.....
-- .
-- .   errorstring                 -- Subprogram SPEC
-- .
-- .....

procedure errorstring(file    :in out file_type;
                                                    file_name:in out string;
                                                    err_str  :in out string);

--| PURPOSE:
--| This procedure is used to read the error message given by the software
--| base program.
```

```
--.....
-- .
-- .   parse_line                -- Subprogram SPEC
-- .
-- .....

procedure parse_line(s: in String);

--| PURPOSE:
--| This procedure is used to determine if the selected line is a directory
--| or a file and if it is a directory it gets the identity of the directory.

end Global;



*******************************************************************************
*******************************************************************************
*******************************************************************************



-- *** TAE Plus Code Generator version V5.1
-- *** File:      pan_mainmenu_b.a
-- *** Generated:  Sep  1 21:37:24 1993
-- *** Revised by : Dogan Ozdemir
-- ***************************************************************************
-- *
-- *   Panel_mainmenu              -- Package BODY
-- *
-- ***************************************************************************
with TAE; use TAE;
with Text_IO;
with Global;
use Text_IO, Global;

--ADDED
with OPERATOR_MATCH_ROUTINES, CREATE_OPERATOR_PARAMETER_FILES_PKG;
use  OPERATOR_MATCH_ROUTINES, CREATE_OPERATOR_PARAMETER_FILES_PKG;

-- One "with" statement for each connected panel.
with Panel_compsel;
with Panel_keyword;

package body Panel_mainmenu is

--| NOTES:
--| For each parameter that you have defined to be "event-generating" in
--| this panel, there is an event handler procedure below.  Each handler
--| has a name that is a concatenation of the parameter name and "_Event".
--| Add application-dependent logic to each event handler.  (As generated
--| by the WorkBench, each event handler simply logs the occurrence of
--| the event.)
--|
--| You may want to flag any changes you make to this file so that if
```

```
--| you regenerate this file, you can more easily cut and paste your
--| modifications back in.  For example:
--|
--|   generated code ...
--|   -- (+) ADDED yourinitials
--|   your code ...
--|   -- (-) ADDED
--|   more generated code ...
--|
--|
--| REGENERATED:
--| The following WorkBench operations will cause regeneration of this file:
--|     The panel's name is changed (not title)
--| For panel:
--|   mainmenu
--|
--| The following WorkBench operations will also cause regeneration:
--|     An item is deleted
--|     A new item is added to this panel
--|     An item's name is changed (not title)
--|     An item's data type is changed
--|     An item's generates events flag is changed
--|     An item's valids changed (if item is type string and connected)
--|     An item's connection information changed
--| For the panel items:
--|   cancel,        browse,        query,        help,
--|
--| CHANGE LOG:
--|  1-Sep-93   TAE     Generated

 --ADDED
operator_file  : file_type;
type_file      : file_type;
operator_list  : String(1..13):="operator_list";
type_list      : String(1..9):="type_list";
Dummy          : Boolean;



 -- ..............................................................................
 -- .
 -- .    Initialize_Panel                -- Subprogram BODY
 -- .
 -- ..............................................................................

procedure Initialize_Panel
  ( Collection_Read
     : in TAE.Tae_Co.Collection_Ptr ) is

--| NOTES: (none)

begin -- Initialize_Panel

  Info := new TAE.Tae_Wpt.Event_Context;
  Info.Collection := Collection_Read;
  TAE.Tae_Co.Co_Find (Info.Collection, "mainmenu_v", Info.View);
```

```
  TAE.Tae_Co.Co_Find (Info.Collection, "mainmenu_t", Info.Target);

exception

  when TAE.UNINITIALIZED_PTR =>
    Text_IO.Put_Line ("Panel_mainmenu.Initialize_Panel:  "
      & "Collection_Read not initialized.");
    raise;

  when TAE.Tae_Co.NO_SUCH_MEMBER =>
    Text_IO.Put_Line ("Panel_mainmenu.Initialize_Panel:  "
      & "(View or Target) not in Collection.");
    raise;

end Initialize_Panel;
```

```
-- ..........................................................................
-- .
-- .   Create_Panel                  -- Subprogram BODY
-- .
-- ..........................................................................

procedure Create_Panel
 ( Panel_State
    : in TAE.Tae_Wpt.Wpt_Flags
     := TAE.Tae_Wpt.WPT_PREFERRED;

   Relative_Window
    : in X_Windows.Window
     := X_Windows.Null_Window ) is


--| NOTES: (none)

begin -- Create_Panel

 if Info.Panel_Id = Tae.Null_Panel_Id then
  TAE.Tae_Wpt.Wpt_NewPanel
   ( Dummy              => "",
     Data_Vm            => Info.Target,
     View_Vm             => Info.View,
     Relative_Window      => Relative_Window,
     User_Context        => Info,
     Flags              => Panel_State,
     Panel_Id            => Info.Panel_Id );
 else
  Text_IO.Put_Line ("Panel (mainmenu) is already displayed.");
 end if;

exception

 when TAE.UNINITIALIZED_PTR =>
  Text_IO.Put_Line ("Panel_mainmenu.Create_Panel:  "
   & "Panel was not initialized prior to creation.");
  raise;

 when TAE.TAE_FAIL =>
  Text_IO.Put_Line ("Panel_mainmenu.Create_Panel:  "
   & "Panel could not be created.");
  raise;

end Create_Panel;
```

```
-- ..........................................................................
-- .
-- .   Connect_Panel                -- Subprogram BODY
-- .
-- ..........................................................................

procedure Connect_Panel
 ( Panel_State
    : in TAE.Tae_Wpt.Wpt_Flags
      := TAE.Tae_Wpt.WPT_PREFERRED;

   Relative_Window
    : in X_Windows.Window
      := X_Windows.Null_Window ) is

--| NOTES: (none)

begin -- Connect_Panel

 if Info.Panel_Id = Tae.Null_Panel_Id then
  Create_Panel
   ( Relative_Window      => Relative_Window,
     Panel_State          => Panel_State );
 else
  TAE.Tae_Wpt.Wpt_SetPanelState (Info.Panel_Id, Panel_State);
 end if;

exception

 when TAE.Tae_Wpt.BAD_STATE =>
  Text_IO.Put_Line ("Panel_mainmenu.Connect_Panel:  "
   & "Invalid panel state.");
  raise;

end Connect_Panel;
```

```
-- ..........................................................................
-- .
-- .   Destroy_Panel                -- Subprogram BODY
-- .
-- ..........................................................................

procedure Destroy_Panel is

--| NOTES: (none)

begin -- Destroy_Panel

 TAE.Tae_Wpt.Wpt_PanelErase(Info.Panel_Id);

exception

 when TAE.Tae_Wpt.BAD_PANEL_ID =>
  Text_IO.Put_Line ("Panel_mainmenu.Destroy_Panel:  "
   & "Info.Panel_Id is an invalid id.");
  raise;

 when TAE.Tae_Wpt.ERASE_NULL_PANEL =>
  -- This panel has not been created yet, or has already been destroyed.
  -- Trap this exception and do nothing.
  null;

end Destroy_Panel;
```

```
--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--
-- begin EVENT HANDLERS
--
-- ..........................................................
-- .
-- .   cancel_Event                -- Subprogram SPEC & BODY
-- .
-- ..........................................................

procedure cancel_Event
  ( Info : in TAE.Tae_Wpt.Event_Context_Ptr ) is

--| PURPOSE:
--| EVENT HANDLER.  Insert application specific information.
--|
--| NOTES: (none)

  Value : array (1..1) of String (1..TAE.Tae_Taeconf.STRINGSIZE);
  Count : TAE.Taeint;

begin -- cancel_Event

  -- Begin default generated code
  --
  TAE.Tae_Vm.Vm_Extract_Count (Info.Parm_Ptr, Count);
  Text_IO.Put ("Panel mainmenu, parm cancel: value = ");
  if Count > 0 then
    TAE.Tae_Vm.Vm_Extract_SVAL (Info.Parm_Ptr, 1, Value(1));
    Text_IO.Put_Line (Value(1));
  else
    Text_IO.Put_Line ("none");
  end if;
  --
  -- End default generated code
-- ADDED
  Global.Set_Application_Done;

  -- Begin generated code for Connection
  --
  Destroy_Panel;
  --*  TCL: Quit
  --
  -- End generated code for Connection

end cancel_Event;
```

```
  -- ..........................................................................
  -- .
  -- .   browse_Event                -- Subprogram SPEC & BODY
  -- .
  -- ..........................................................................

  procedure browse_Event
    ( Info : in TAE.Tae_Wpt.Event_Context_Ptr ) is

  --| PURPOSE:
  --| EVENT HANDLER.  Insert application specific information.
  --|
  --| NOTES: (none)

    Value : array (1..1) of String (1..TAE.Tae_Taeconf.STRINGSIZE);
    Count : TAE.Taeint;
--ADDED
   N :integer:=1;

  begin -- browse_Event

    -- Begin default generated code
    --
    TAE.Tae_Vm.Vm_Extract_Count (Info.Parm_Ptr, Count);
    Text_IO.Put ("Panel mainmenu, parm browse: value = ");
    if Count > 0 then
      TAE.Tae_Vm.Vm_Extract_SVAL (Info.Parm_Ptr, 1, Value(1));
      Text_IO.Put_Line (Value(1));
    else
      Text_IO.Put_Line ("none");
    end if;
    --
    -- End default generated code

    -- Begin generated code for Connection
    --
--ADDED
    strlen(library,N);

    if TAE.Tae_Misc.s_equal (Value(1), "Types") then null;
--ADDED
       system_call(com&"tl "&library(1..N)&" "&"type_list");
       list_components(type_file,type_list,file_vec,num_of_comp);
       TAE.Tae_Wpt.Wpt_SetStringConstraints(
        Panel_compsel.Info.Panel_Id,"compsel",taeint(num_of_comp),file_vec);
          Dummy:=TAE.Tae_Wpt.Wpt_Pending;
          system_call("rm type_list");

      Connect_Panel (TAE.Tae_Wpt.WPT_INVISIBLE);
      Panel_compsel.Connect_Panel (TAE.Tae_Wpt.WPT_VISIBLE);

    elsif TAE.Tae_Misc.s_equal (Value(1), "Operators") then null;
--ADDED
       system_call(com&"ol "&library(1..N)&" "&"operator_list");
       list_components(operator_file,operator_list,file_vec,num_of_comp);
```

```
    TAE.Tae_Wpt.Wpt_SetStringConstraints(
     Panel_compsel.Info.Panel_Id,"compsel",taeint(num_of_comp),file_vec);
       Dummy:=TAE.Tae_Wpt.Wpt_Pending;
       system_call("rm operator_list");

    Connect_Panel (TAE.Tae_Wpt.WPT_INVISIBLE);
    Panel_compsel.Connect_Panel (TAE.Tae_Wpt.WPT_VISIBLE);

  end if;
  --
  -- End generated code for Connection

end browse_Event;
```

```
-- .....
-- .
-- .   query_Event                -- Subprogram SPEC & BODY
-- .
-- .....

procedure query_Event
  ( Info : in TAE.Tae_Wpt.Event_Context_Ptr ) is

--| PURPOSE:
--| EVENT HANDLER.  Insert application specific information.
--|
--| NOTES: (none)

   Value : array (1..1) of String (1..TAE.Tae_Taeconf.STRINGSIZE);
   Count : TAE.Taeint;
--ADDED
    FINAL_MATCHES,
    OPERATOR_SIGNATURE_MATCHES : FILE_TYPE;
    NUM,
    LEN                 : INTEGER;
    NO_MATCH            : BOOLEAN := TRUE;
    MATCH_FILE_NAME     : STRING(1..11) := "final_match";

 begin -- query_Event

   -- Begin default generated code
   --
   TAE.Tae_Vm.Vm_Extract_Count (Info.Parm_Ptr, Count);
   Text_IO.Put ("Panel mainmenu, parm query: value = ");
   if Count > 0 then
     TAE.Tae_Vm.Vm_Extract_SVAL (Info.Parm_Ptr, 1, Value(1));
     Text_IO.Put_Line (Value(1));
   else
     Text_IO.Put_Line ("none");
   end if;
   --
   -- End default generated code

   -- Begin generated code for Connection
   --
   if TAE.Tae_Misc.s_equal (Value(1), "Keyword") then null;
     Connect_Panel (TAE.Tae_Wpt.WPT_INVISIBLE);
     Panel_keyword.Connect_Panel (TAE.Tae_Wpt.WPT_VISIBLE);

   elsif TAE.Tae_Misc.s_equal (Value(1), "PSDL") then null;
     Connect_Panel (TAE.Tae_Wpt.WPT_INVISIBLE);
     Panel_compsel.Connect_Panel (TAE.Tae_Wpt.WPT_VISIBLE);
-- ADDED
   if COMPONENT_IS_OPERATOR then

     -- get signature of query operator component
     system_call(com2);

     -- get all software base components that match query
```

```
-- component signature
system_call(com&"cq Ada "&OPERATOR_FILE_NAME&" query_sig_match");

-- load query operator component parameter mapping data structures
THE_STATUS := QUERY_COMPONENT;
CREATE_OPERATOR_PARAMETER_FILES(THE_STATUS, QC_IN_PARAMS, QC_IN_COUNT,
    QC_OUT_PARAMS, QC_OUT_COUNT, GEN_PARAMS, GEN_COUNT, HAS_GENERICS,
    QC_NAME);

-- save query operator component psdl files
system_call("cp "&Global.OPERATOR_FILE_NAME&" svopspec.txt");
system_call("cp "&Global.PROTOTYPE_FILE_NAME&" svprotspec.txt");

OPEN(OPERATOR_SIGNATURE_MATCHES, MODE => IN_FILE,
                    NAME => "query_sig_match");

CREATE(FINAL_MATCHES, MODE => OUT_FILE,
                NAME => MATCH_FILE_NAME);

-- continue to filter the matched components
while not END_OF_FILE(OPERATOR_SIGNATURE_MATCHES) loop

 -- get a candidate component
 GET_LINE(OPERATOR_SIGNATURE_MATCHES, COMPONENT, LEN);

 -- get the component's name
 strlen(COMPONENT, LEN);

 -- get the psdl file associated with that name
 system_call(com&"cv Ada "&COMPONENT(1..LEN)&" outpsdl outspec outbody");

 -- copy the software base component psdl file into the
 -- appropriate files
 system_call("cp outpsdl "&OPERATOR_FILE_NAME);
 system_call("cp outpsdl "&PROTOTYPE_FILE_NAME);

 -- load software base operator component parameter mapping
 -- data structures
 THE_STATUS := SOFTWARE_BASE_COMPONENT;
 CREATE_OPERATOR_PARAMETER_FILES(
   THE_STATUS, SBC_IN_PARAMS, SBC_IN_COUNT,
   SBC_OUT_PARAMS, SBC_OUT_COUNT, GEN_PARAMS, GEN_COUNT,
   HAS_GENERICS,SBC_NAME);

 if HAS_GENERICS then

   -- see if an instantiation of the software component by the
   -- query component is possible
   FIND_INSTANTIATION(QC_IN_PARAMS, QC_OUT_PARAMS, SBC_IN_PARAMS,
              SBC_OUT_PARAMS, GEN_PARAMS, NO_MATCH);

 else

   -- no generics in software base component so check to ensure
   -- both components match at the Array component level (i.e.
   -- array element and index) if the components have parameters
```

```
    -- of type Array
    MATCH_ARRAYS(QC_IN_PARAMS, QC_OUT_PARAMS, SBC_IN_PARAMS,
            SBC_OUT_PARAMS, NO_MATCH);
   end if;


   if not NO_MATCH then

    -- software base component successfully passed this filter
    -- so add it to "final_match" file
    PUT_LINE(FINAL_MATCHES, COMPONENT);

   end if;

  end loop;

  CLOSE(OPERATOR_SIGNATURE_MATCHES);
  CLOSE(FINAL_MATCHES);

  -- restore query operator component psdl files
  system_call("cp svopspec.txt "&OPERATOR_FILE_NAME);
  system_call("cp svprotspec.txt "&PROTOTYPE_FILE_NAME);

  -- remove files that are no longer needed
  system_call("rm svopspec.txt");
  system_call("rm svprotspec.txt");
  system_call("rm query_sig_match");
  system_call("rm outpsdl");
  system_call("rm outspec");
  system_call("rm outbody");

 else

  -- get signature of query type component
  system_call(com3);

  -- get all software base components that match query
  -- component signature
  system_call(com&"cq Ada "&TYPE_FILE_NAME&" "&MATCH_FILE_NAME);


 end if;


 -- load the component selection TAE display item with
 -- the components listed in "final_match"
 list_components(FINAL_MATCHES, MATCH_FILE_NAME, FILE_VEC, NUM);
 TAE.Tae_Wpt.Wpt_SetStringConstraints(
    Panel_compsel.Info.Panel_Id,"compsel",taeint(NUM),FILE_VEC);
 Dummy:=TAE.Tae_Wpt.Wpt_Pending;


 -- remove files that are no longer needed
 system_call("rm "&MATCH_FILE_NAME);

 end if;
```

```
  --
  -- End generated code for Connection

end query_Event;
```

```
-- .....
-- .
-- .   help_Event                    -- Subprogram SPEC & BODY
-- .
-- .....

procedure help_Event
 ( Info : in TAE.Tae_Wpt.Event_Context_Ptr ) is

--| PURPOSE:
--| EVENT HANDLER.  Insert application specific information.
--|
--| NOTES: (none)

 Value : array (1..1) of String (1..TAE.Tae_Taeconf.STRINGSIZE);
 Count : TAE.Taeint;

begin -- help_Event

 -- Begin default generated code
 --
 TAE.Tae_Vm.Vm_Extract_Count (Info.Parm_Ptr, Count);
 Text_IO.Put ("Panel mainmenu, parm help: value = ");
 if Count > 0 then
  TAE.Tae_Vm.Vm_Extract_SVAL (Info.Parm_Ptr, 1, Value(1));
  Text_IO.Put_Line (Value(1));
 else
  Text_IO.Put_Line ("none");
 end if;
 --
 -- End default generated code

end help_Event;
```

```
  --
  -- end EVENT HANDLERS
  --
  ------------------------------------------------------------------------



  -- ..........................................................
  -- .
  -- .   Dispatch_Item                -- Subprogram BODY
  -- .
  -- ..........................................................

  procedure Dispatch_Item
    ( User_Context_Ptr : in TAE.Tae_Wpt.Event_Context_Ptr ) is

  --| NOTES: (none)

  begin -- Dispatch_Item

    if TAE.Tae_Misc.s_equal ("cancel", User_Context_Ptr.Parm_Name) then
      cancel_Event (User_Context_Ptr);
    elsif TAE.Tae_Misc.s_equal ("browse", User_Context_Ptr.Parm_Name) then
      browse_Event (User_Context_Ptr);
    elsif TAE.Tae_Misc.s_equal ("query", User_Context_Ptr.Parm_Name) then
      query_Event (User_Context_Ptr);
    elsif TAE.Tae_Misc.s_equal ("help", User_Context_Ptr.Parm_Name) then
      help_Event (User_Context_Ptr);
    end if;

  end Dispatch_Item;

end Panel_mainmenu;



*******************************************************************************
*******************************************************************************
*******************************************************************************



-- *** TAE Plus Code Generator version V5.1
-- *** File:      pan_mapping_s.a
-- *** Generated:  Sep  1 21:37:24 1993
-- *** Revised by Scott Dolgoff
--  ***************************************************************************
--  *
--  *   Panel_mapping                -- Package SPEC
--  *
--  ***************************************************************************

with TAE;
with X_Windows;
```

```
package Panel_mapping is

--| PURPOSE:
--| This package encapsulates the TAE Plus panel:  mapping
--| These subprograms enable panel initialization, creation, destruction,
--| and event dispatching.  For more advanced manipulation of the panel
--| using the TAE package, the panel's Event_Context (Info) is provided.
--| It includes the Target and View (available after initialization)
--| and the Panel_Id (available after creation).
--|
--| INITIALIZATION EXCEPTIONS: (none)
--|
--| NOTES: (none)
--|
--| REGENERATED:
--| The following Workbench operations will cause regeneration of this file:
--|    The panel's name is changed (not title)
--| For panel:
--|  mapping
--|
--| CHANGE LOG:
--|  1-Sep-93   TAE     Generated


 Info : TAE.Tae_Wpt.Event_Context_Ptr;   -- panel information
```

```
-- ..........................................................................
-- .
-- .   Initialize_Panel                -- Subprogram SPEC
-- .
-- ..........................................................................

procedure Initialize_Panel
  ( Collection_Read                     -- TAE Collection read from
     : in TAE.Tae_Co.Collection_Ptr );   -- resource file

--| PURPOSE:
--| This procedure initializes the Info.Target and Info.View for this panel
--|
--| EXCEPTIONS:
--| TAE.UNINITIALIZED_PTR is raised if Collection_Read not initialized
--| TAE.Tae_Co.NO_SUCH_MEMBER is raised if the panel is not in
--|   Collection_Read
--|
--| NOTES: (none)




-- ..........................................................................
-- .
-- .   Create_Panel                   -- Subprogram SPEC
-- .
-- ..........................................................................

procedure Create_Panel
  ( Panel_State                         -- Flags sent to Wpt_NewPanel.
     : in TAE.Tae_Wpt.Wpt_Flags
       := TAE.Tae_Wpt.WPT_PREFERRED;

    Relative_Window                     -- Panel origin is offset from
     : in X_Windows.Window              -- this X Window.  Null_Window
       := X_Windows.Null_Window );      -- uses the root window.

--| PURPOSE:
--| This procedure creates this panel object in the specified Panel_State
--| and stores the panel Id in Info.Panel_Id.
--|
--| EXCEPTIONS:
--| TAE.UNINITIALIZED_PTR is raised if the panel is not initialized
--| TAE.TAE_FAIL is raised if the panel could not be created
--|
--| NOTES: (none)
```

```
-- ..................................................................
-- .
-- .   Connect_Panel                -- Subprogram SPEC
-- .
-- ..................................................................

procedure Connect_Panel
  ( Panel_State
     : in TAE.Tae_Wpt.Wpt_Flags
       := TAE.Tae_Wpt.WPT_PREFERRED;

    Relative_Window                  -- Panel origin is offset from
     : in X_Windows.Window             -- this X Window.  Null_Window
       := X_Windows.Null_Window );      -- uses the root window.

--| PURPOSE:
--| If this panel doesn't exist, this procedure creates this panel object
--|   in the specifiec Panel_State and stores the panel Id in
--|   Info.Panel_Id.
--| If this panel does exist, it is set to the specified Panel_State.
--|   In this case, Relative_Window is ignored.
--|
--| EXCEPTIONS:
--| TAE.UNINITIALIZED_PTR is raised from Create_Panel if the panel is
--|   not initialized
--| TAE.TAE_FAIL is raised from Create_Panel if the panel could not be
--|   created
--| TAE.Tae_Wpt.BAD_STATE is raised if the panel exists and the
--|   Panel_State is an invalid state
--|
--| NOTES: (none)
```

```
  -- ..........................................................................
  -- .
  -- .   Destroy_Panel                -- Subprogram SPEC
  -- .
  -- ..........................................................................

  procedure Destroy_Panel;

  --| PURPOSE:
  --| This procedure erases a panel from the screen and de-allocates the
  --| associated panel object (not the target and view).
  --|
  --| EXCEPTIONS:
  --| TAE.Tae_Wpt.BAD_PANEL_ID is raised if Info.Panel_Id is an invalid id.
  --|
  --| NOTES:
  --| Info.Panel_Id is set to TAE.NULL_PANEL_ID, and should not referenced
  --| in any Wpt call until it is created again.




  -- ..........................................................................
  -- .
  -- .   Dispatch_Item                -- Subprogram SPEC
  -- .
  -- ..........................................................................

  procedure Dispatch_Item
    ( User_Context_Ptr                -- Wpt Event Context for a PARM
      : in TAE.Tae_Wpt.Event_Context_Ptr ); -- event.

  --| PURPOSE:
  --| This procedure calls the Event Handler specified by User_Context_Ptr
  --|
  --| EXCEPTIONS:
  --| Application-specific
  --|
  --| NOTES: (none)

end Panel_mapping;


*******************************************************************************
*******************************************************************************
*******************************************************************************



-- *** TAE Plus Code Generator version V5.1
-- *** File:      pan_mainmenu_b.a
-- *** Generated:   Sep  1 21:37:24 1993
-- *** Revised by : Dogan Ozdemir
```

```
-- ******************************************************************************
-- *
-- *   Panel_mainmenu                -- Package BODY
-- *
-- ******************************************************************************
with TAE; use TAE;
with Text_IO;
with Global;
use Text_IO, Global;

--ADDED
with OPERATOR_MATCH_ROUTINES, CREATE_OPERATOR_PARAMETER_FILES_PKG;
use  OPERATOR_MATCH_ROUTINES, CREATE_OPERATOR_PARAMETER_FILES_PKG;

-- One "with" statement for each connected panel.
with Panel_compsel;
with Panel_keyword;

package body Panel_mainmenu is

--| NOTES:
--| For each parameter that you have defined to be "event-generating" in
--| this panel, there is an event handler procedure below.  Each handler
--| has a name that is a concatenation of the parameter name and "_Event".
--| Add application-dependent logic to each event handler.  (As generated
--| by the WorkBench, each event handler simply logs the occurrence of
--| the event.)
--|
--| You may want to flag any changes you make to this file so that if
--| you regenerate this file, you can more easily cut and paste your
--| modifications back in.  For example:
--|
--|   generated code ...
--|   -- (+) ADDED yourinitials
--|   your code ...
--|   -- (-) ADDED
--|   more generated code ...
--|
--|
--| REGENERATED:
--| The following WorkBench operations will cause regeneration of this file:
--|    The panel's name is changed (not title)
--| For panel:
--|   mainmenu
--|
--| The following WorkBench operations will also cause regeneration:
--|    An item is deleted
--|    A new item is added to this panel
--|    An item's name is changed (not title)
--|    An item's data type is changed
--|    An item's generates events flag is changed
--|    An item's valids changed (if item is type string and connected)
--|    An item's connection information changed
--| For the panel items:
--|   cancel,        browse,        query,        help,
--|
```

```
--| CHANGE LOG:
--|  1-Sep-93   TAE     Generated

 --ADDED
operator_file  : file_type;
type_file      : file_type;
operator_list  : String(1..13):="operator_list";
type_list      : String(1..9):="type_list";
Dummy          : Boolean;



 -- ..............................................................
 -- .
 -- .   Initialize_Panel                -- Subprogram BODY
 -- .
 -- ..............................................................

 procedure Initialize_Panel
  ( Collection_Read
     : in TAE.Tae_Co.Collection_Ptr ) is

 --| NOTES: (none)

 begin -- Initialize_Panel

  Info := new TAE.Tae_Wpt.Event_Context;
  Info.Collection := Collection_Read;
  TAE.Tae_Co.Co_Find (Info.Collection, "mainmenu_v", Info.View);
  TAE.Tae_Co.Co_Find (Info.Collection, "mainmenu_t", Info.Target);

 exception

  when TAE.UNINITIALIZED_PTR =>
    Text_IO.Put_Line ("Panel_mainmenu.Initialize_Panel:  "
      & "Collection_Read not initialized.");
    raise;

  when TAE.Tae_Co.NO_SUCH_MEMBER =>
    Text_IO.Put_Line ("Panel_mainmenu.Initialize_Panel:  "
      & "(View or Target) not in Collection.");
    raise;

 end Initialize_Panel;
```

```
-- ..................................................................
-- .
-- .   Create_Panel                  -- Subprogram BODY
-- .
-- ..................................................................

procedure Create_Panel
 ( Panel_State
    : in TAE.Tae_Wpt.Wpt_Flags
      := TAE.Tae_Wpt.WPT_PREFERRED;

   Relative_Window
    : in X_Windows.Window
      := X_Windows.Null_Window ) is


--| NOTES: (none)

begin -- Create_Panel

 if Info.Panel_Id = Tae.Null_Panel_Id then
  TAE.Tae_Wpt.Wpt_NewPanel
   ( Dummy              => "",
     Data_Vm            => Info.Target,
     View_Vm             => Info.View,
     Relative_Window      => Relative_Window,
     User_Context        => Info,
     Flags             => Panel_State,
     Panel_Id           => Info.Panel_Id );
 else
  Text_IO.Put_Line ("Panel (mainmenu) is already displayed.");
 end if;

exception

 when TAE.UNINITIALIZED_PTR =>
  Text_IO.Put_Line ("Panel_mainmenu.Create_Panel:  "
   & "Panel was not initialized prior to creation.");
  raise;

 when TAE.TAE_FAIL =>
  Text_IO.Put_Line ("Panel_mainmenu.Create_Panel:  "
   & "Panel could not be created.");
  raise;

end Create_Panel;
```

```
-- ..........................................................................
-- .
-- .   Connect_Panel                  -- Subprogram BODY
-- .
-- ..........................................................................

procedure Connect_Panel
 ( Panel_State
    : in TAE.Tae_Wpt.Wpt_Flags
      := TAE.Tae_Wpt.WPT_PREFERRED;

   Relative_Window
    : in X_Windows.Window
      := X_Windows.Null_Window ) is

--| NOTES: (none)

begin -- Connect_Panel

 if Info.Panel_Id = Tae.Null_Panel_Id then
  Create_Panel
   ( Relative_Window      => Relative_Window,
     Panel_State          => Panel_State );
 else
  TAE.Tae_Wpt.Wpt_SetPanelState (Info.Panel_Id, Panel_State);
 end if;

exception

 when TAE.Tae_Wpt.BAD_STATE =>
  Text_IO.Put_Line ("Panel_mainmenu.Connect_Panel:  "
   & "Invalid panel state.");
  raise;

end Connect_Panel;
```

```
-- ..........................................................................
-- .
-- .   Destroy_Panel                -- Subprogram BODY
-- .
-- ..........................................................................

procedure Destroy_Panel is

--| NOTES: (none)

begin -- Destroy_Panel

 TAE.Tae_Wpt.Wpt_PanelErase(Info.Panel_Id);

exception

 when TAE.Tae_Wpt.BAD_PANEL_ID =>
  Text_IO.Put_Line ("Panel_mainmenu.Destroy_Panel:  "
   & "Info.Panel_Id is an invalid id.");
  raise;

 when TAE.Tae_Wpt.ERASE_NULL_PANEL =>
  -- This panel has not been created yet, or has already been destroyed.
  -- Trap this exception and do nothing.
  null;

end Destroy_Panel;
```

```
--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--
-- begin EVENT HANDLERS
--
-- ..................................................................
-- .
-- .   cancel_Event                -- Subprogram SPEC & BODY
-- .
-- ..................................................................

procedure cancel_Event
  ( Info : in TAE.Tae_Wpt.Event_Context_Ptr ) is

--| PURPOSE:
--| EVENT HANDLER.  Insert application specific information.
--|
--| NOTES: (none)

  Value : array (1..1) of String (1..TAE.Tae_Taeconf.STRINGSIZE);
  Count : TAE.Taeint;

begin -- cancel_Event

  -- Begin default generated code
  --
  TAE.Tae_Vm.Vm_Extract_Count (Info.Parm_Ptr, Count);
  Text_IO.Put ("Panel mainmenu, parm cancel: value = ");
  if Count > 0 then
    TAE.Tae_Vm.Vm_Extract_SVAL (Info.Parm_Ptr, 1, Value(1));
    Text_IO.Put_Line (Value(1));
  else
    Text_IO.Put_Line ("none");
  end if;
  --
  -- End default generated code
-- ADDED
  Global.Set_Application_Done;

  -- Begin generated code for Connection
  --
  Destroy_Panel;
  --*  TCL: Quit
  --
  -- End generated code for Connection

end cancel_Event;
```

```
  -- ..................................................................
  -- .
  -- .   browse_Event                -- Subprogram SPEC & BODY
  -- .
  -- ..................................................................

  procedure browse_Event
    ( Info : in TAE.Tae_Wpt.Event_Context_Ptr ) is

  --| PURPOSE:
  --| EVENT HANDLER.  Insert application specific information.
  --|
  --| NOTES: (none)

    Value : array (1..1) of String (1..TAE.Tae_Taeconf.STRINGSIZE);
    Count : TAE.Taeint;
--ADDED
   N :integer:=1;

  begin -- browse_Event

    -- Begin default generated code
    --
    TAE.Tae_Vm.Vm_Extract_Count (Info.Parm_Ptr, Count);
    Text_IO.Put ("Panel mainmenu, parm browse: value = ");
    if Count > 0 then
      TAE.Tae_Vm.Vm_Extract_SVAL (Info.Parm_Ptr, 1, Value(1));
      Text_IO.Put_Line (Value(1));
    else
      Text_IO.Put_Line ("none");
    end if;
    --
    -- End default generated code

    -- Begin generated code for Connection
    --
--ADDED
    strlen(library,N);

    if TAE.Tae_Misc.s_equal (Value(1), "Types") then null;
--ADDED
       system_call(com&"tl "&library(1..N)&" "&"type_list");
       list_components(type_file,type_list,file_vec,num_of_comp);
       TAE.Tae_Wpt.Wpt_SetStringConstraints(
        Panel_compsel.Info.Panel_Id,"compsel",taeint(num_of_comp),file_vec);
          Dummy:=TAE.Tae_Wpt.Wpt_Pending;
          system_call("rm type_list");

      Connect_Panel (TAE.Tae_Wpt.WPT_INVISIBLE);
      Panel_compsel.Connect_Panel (TAE.Tae_Wpt.WPT_VISIBLE);

    elsif TAE.Tae_Misc.s_equal (Value(1), "Operators") then null;
--ADDED
      system_call(com&"ol "&library(1..N)&" "&"operator_list");
      list_components(operator_file,operator_list,file_vec,num_of_comp);
```

```
   TAE.Tae_Wpt.Wpt_SetStringConstraints(
    Panel_compsel.Info.Panel_Id,"compsel",taeint(num_of_comp),file_vec);
      Dummy:=TAE.Tae_Wpt.Wpt_Pending;
      system_call("rm operator_list");

   Connect_Panel (TAE.Tae_Wpt.WPT_INVISIBLE);
   Panel_compsel.Connect_Panel (TAE.Tae_Wpt.WPT_VISIBLE);

  end if;
  --
  -- End generated code for Connection

end browse_Event;
```

```
-- ..........................................................................
-- .
-- .   query_Event                  -- Subprogram SPEC & BODY
-- .
-- ..........................................................................

procedure query_Event
  ( Info : in TAE.Tae_Wpt.Event_Context_Ptr ) is

--| PURPOSE:
--| EVENT HANDLER.  Insert application specific information.
--|
--| NOTES: (none)

  Value : array (1..1) of String (1..TAE.Tae_Taeconf.STRINGSIZE);
  Count : TAE.Taeint;
--ADDED
   FINAL_MATCHES,
   OPERATOR_SIGNATURE_MATCHES : FILE_TYPE;
   NUM,
   LEN                  : INTEGER;
   NO_MATCH             : BOOLEAN := TRUE;
   MATCH_FILE_NAME      : STRING(1..11) := "final_match";

begin -- query_Event

  -- Begin default generated code
  --
  TAE.Tae_Vm.Vm_Extract_Count (Info.Parm_Ptr, Count);
  Text_IO.Put ("Panel mainmenu, parm query: value = ");
  if Count > 0 then
    TAE.Tae_Vm.Vm_Extract_SVAL (Info.Parm_Ptr, 1, Value(1));
    Text_IO.Put_Line (Value(1));
  else
    Text_IO.Put_Line ("none");
  end if;
  --
  -- End default generated code

  -- Begin generated code for Connection
  --
  if TAE.Tae_Misc.s_equal (Value(1), "Keyword") then null;
    Connect_Panel (TAE.Tae_Wpt.WPT_INVISIBLE);
    Panel_keyword.Connect_Panel (TAE.Tae_Wpt.WPT_VISIBLE);

  elsif TAE.Tae_Misc.s_equal (Value(1), "PSDL") then null;
    Connect_Panel (TAE.Tae_Wpt.WPT_INVISIBLE);
    Panel_compsel.Connect_Panel (TAE.Tae_Wpt.WPT_VISIBLE);
-- ADDED
  if COMPONENT_IS_OPERATOR then

    -- get signature of query operator component
    system_call(com2);

    -- get all software base components that match query
```

```
-- component signature
system_call(com&"cq Ada "&OPERATOR_FILE_NAME&" query_sig_match");

-- load query operator component parameter mapping data structures
THE_STATUS := QUERY_COMPONENT;
CREATE_OPERATOR_PARAMETER_FILES(THE_STATUS, QC_IN_PARAMS, QC_IN_COUNT,
    QC_OUT_PARAMS, QC_OUT_COUNT, GEN_PARAMS, GEN_COUNT, HAS_GENERICS,
    QC_NAME);

-- save query operator component psdl files
system_call("cp "&Global.OPERATOR_FILE_NAME&" svopspec.txt");
system_call("cp "&Global.PROTOTYPE_FILE_NAME&" svprotspec.txt");

OPEN(OPERATOR_SIGNATURE_MATCHES, MODE => IN_FILE,
                    NAME => "query_sig_match");

CREATE(FINAL_MATCHES, MODE => OUT_FILE,
                NAME => MATCH_FILE_NAME);

-- continue to filter the matched components
while not END_OF_FILE(OPERATOR_SIGNATURE_MATCHES) loop

 -- get a candidate component
 GET_LINE(OPERATOR_SIGNATURE_MATCHES, COMPONENT, LEN);

 -- get the component's name
 strlen(COMPONENT, LEN);

 -- get the psdl file associated with that name
 system_call(com&"cv Ada "&COMPONENT(1..LEN)&" outpsdl outspec outbody");

 -- copy the software base component psdl file into the
 -- appropriate files
 system_call("cp outpsdl "&OPERATOR_FILE_NAME);
 system_call("cp outpsdl "&PROTOTYPE_FILE_NAME);

 -- load software base operator component parameter mapping
 -- data structures
 THE_STATUS := SOFTWARE_BASE_COMPONENT;
 CREATE_OPERATOR_PARAMETER_FILES(
   THE_STATUS, SBC_IN_PARAMS, SBC_IN_COUNT,
   SBC_OUT_PARAMS, SBC_OUT_COUNT, GEN_PARAMS, GEN_COUNT,
   HAS_GENERICS,SBC_NAME);

 if HAS_GENERICS then

   -- see if an instantiation of the software component by the
   -- query component is possible
   FIND_INSTANTIATION(QC_IN_PARAMS, QC_OUT_PARAMS, SBC_IN_PARAMS,
              SBC_OUT_PARAMS, GEN_PARAMS, NO_MATCH);

 else

   -- no generics in software base component so check to ensure
   -- both components match at the Array component level (i.e.
   -- array element and index) if the components have parameters
```

```
    -- of type Array
    MATCH_ARRAYS(QC_IN_PARAMS, QC_OUT_PARAMS, SBC_IN_PARAMS,
            SBC_OUT_PARAMS, NO_MATCH);
   end if;


   if not NO_MATCH then

    -- software base component successfully passed this filter
    -- so add it to "final_match" file
    PUT_LINE(FINAL_MATCHES, COMPONENT);

   end if;

  end loop;

  CLOSE(OPERATOR_SIGNATURE_MATCHES);
  CLOSE(FINAL_MATCHES);

  -- restore query operator component psdl files
  system_call("cp svopspec.txt "&OPERATOR_FILE_NAME);
  system_call("cp svprotspec.txt "&PROTOTYPE_FILE_NAME);

  -- remove files that are no longer needed
  system_call("rm svopspec.txt");
  system_call("rm svprotspec.txt");
  system_call("rm query_sig_match");
  system_call("rm outpsdl");
  system_call("rm outspec");
  system_call("rm outbody");

 else

  -- get signature of query type component
  system_call(com3);

  -- get all software base components that match query
  -- component signature
  system_call(com&"cq Ada "&TYPE_FILE_NAME&" "&MATCH_FILE_NAME);


 end if;


 -- load the component selection TAE display item with
 -- the components listed in "final_match"
 list_components(FINAL_MATCHES, MATCH_FILE_NAME, FILE_VEC, NUM);
 TAE.Tae_Wpt.Wpt_SetStringConstraints(
    Panel_compsel.Info.Panel_Id,"compsel",taeint(NUM),FILE_VEC);
 Dummy:=TAE.Tae_Wpt.Wpt_Pending;


 -- remove files that are no longer needed
 system_call("rm "&MATCH_FILE_NAME);

 end if;
```

```
  --
  -- End generated code for Connection

end query_Event;
```

```
-- ..........................................................................
-- .
-- .   help_Event                    -- Subprogram SPEC & BODY
-- .
-- ..........................................................................

procedure help_Event
  ( Info : in TAE.Tae_Wpt.Event_Context_Ptr ) is

--| PURPOSE:
--| EVENT HANDLER.  Insert application specific information.
--|
--| NOTES: (none)

  Value : array (1..1) of String (1..TAE.Tae_Taeconf.STRINGSIZE);
  Count : TAE.Taeint;

begin -- help_Event

  -- Begin default generated code
  --
  TAE.Tae_Vm.Vm_Extract_Count (Info.Parm_Ptr, Count);
  Text_IO.Put ("Panel mainmenu, parm help: value = ");
  if Count > 0 then
    TAE.Tae_Vm.Vm_Extract_SVAL (Info.Parm_Ptr, 1, Value(1));
    Text_IO.Put_Line (Value(1));
  else
    Text_IO.Put_Line ("none");
  end if;
  --
  -- End default generated code

end help_Event;
```

```
  --
  -- end EVENT HANDLERS
  --
  ------------------------------------------------------------------------



  -- ..........................................................
  -- .
  -- .   Dispatch_Item                -- Subprogram BODY
  -- .
  -- ..........................................................

  procedure Dispatch_Item
    ( User_Context_Ptr : in TAE.Tae_Wpt.Event_Context_Ptr ) is

  --! NOTES: (none)

  begin -- Dispatch_Item

    if TAE.Tae_Misc.s_equal ("cancel", User_Context_Ptr.Parm_Name) then
      cancel_Event (User_Context_Ptr);
    elsif TAE.Tae_Misc.s_equal ("browse", User_Context_Ptr.Parm_Name) then
      browse_Event (User_Context_Ptr);
    elsif TAE.Tae_Misc.s_equal ("query", User_Context_Ptr.Parm_Name) then
      query_Event (User_Context_Ptr);
    elsif TAE.Tae_Misc.s_equal ("help", User_Context_Ptr.Parm_Name) then
      help_Event (User_Context_Ptr);
    end if;

  end Dispatch_Item;

end Panel_mainmenu;
```

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
3.	Center for Naval Analysis 4401 Ford Avenue Alexandria, VA 22302-0268	1
4.	Director of Research Administration Attn: Prof. Howard Code 012 Naval Postgraduate School Monterey, CA 93943	1
5.	Ted Lewis Professor and Chairman Computer Science Department, Code CS/Lt Naval Postgraduate School Monterey, CA 93943-5100	1
6.	Prof. Luqi, Code CSLq Computer Science Department Naval Postgraduate School Monterey, CA 93943	20
7.	CPT Scott Dolgoff 11423 Orchard Green Court Reston, Virginia 22090	3
8.	Chief of Naval Research 800 N. Quincy Street Arlington, Virginia 22217	1

9. Ada Joint Program Office 1
Defense Research & Engineering
Pentagon, 3E118
Attn: John P. Solomond
Washington, D.C. 20301

10. Carnegie Mellon University 1
Software Engineering Institute
Department of Computer Science
Attn: Mario R. Barbacci
Pittsburgh, Pennsylvania 15213-3890

11. Office of Naval Technology 1
800 N. Quincy St.
Code 227 ONT
Attn. Dr. Elizabeth Wald
Arlington, Virginia 22132-5000

12. Advanced Research Projects Agency (ARPA) 1
Integrated Strategic Technology Office (ISTO)
Attn. Kirsti Bellman
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

13. ARPA/ISTO 1
3701 N. Fairfax Dr.
Attn. Edward Thompson
Arlington, Virginia 22203-1714

14. Attn. Mr. William McCoy 1
Code K54, NSWC
Dahlgren, VA 22448

15. Advanced Research Projects Agency (ARPA) 1
Director, Naval Technology Office
1400 Wilson Boulevard
Arlington, Virginia 2209-2308

16. Advanced Research Projects Agency (ARPA) 1
Director, Prototype Projects Office
1400 Wilson Boulevard
Arlington, Virginia 2209-2308

17. Advanced Research Projects Agency (ARPA) 1
Director, Tactical Technology Office
1400 Wilson Boulevard
Arlington, Virginia 2209-2308

18. Dr. Aimram Yehudai 1
Tel Aviv University
School of Mathematical Sciences
Department of Computer Science
Tel Aviv, Israel 69978

19. Dr. Robert M. Balzer 1
USC-Information Sciences Institute
4676 Admiralty Way
Suite 1001
Marina del Ray, California 90292-6695

20. SYSCORP International 1
4821 Spicewood Springs Rd.
Attn. Dr. R. T. Yeh
Austin, Texas 78759

21. Kestrel Institute 1
Attn. Dr. C. Green
1801 Page Mill Road
Palo Alto, California 94304

22. Prof. D. Berry 1
Software Engineering Institute
Carnegie-Mellon University
Pittsburg, PA 15213-3890

23. Massachusetts Institute of Technology 1
Department of Electrical Engineering and Computer Science
Attn. Dr. B. Liskov
545 Tech Square
Cambridge, Massachusetts 02139

24. Massachusetts Institute of Technology 1
Department of Electrical Engineering and Computer Science
Attn. Dr. J. Guttag
545 Tech Square
Cambridge, Massachusetts 02139

25. MCC AI Laboratory 1
Attn. Dr. Michael Gray
3500 West Balcones Center Drive
Austin, Texas 78759

26. National Science Foundation 1
Division of Computer and Computation Research
1800 G St. NW
Attn. Dr. Bruce Barnes
Washington, D.C. 20550

27. NRAD 1
Code 411
Attn. Hans Mumm
271 Catalina Blvd.
San Diego, California 92152-5000

28. NAVSEA, PMS-4123H 1
Attn. William Wilder
Arlington, VA 22202-5101

29. New Jersey Institute of Technology 1
Computer Science Department
Attn. Dr. Peter Ng
Newark, New Jersey 07102

30. Office of Naval Research 1
Computer Science Division, Code 1133
Attn. Dr. Van Tilborg
800 N. Quincy Street
Arlington, Virginia 22217-5000

31. Office of Naval Research 1
Computer Science Division, Code 1133
Attn. Dr. R. Wachter
800 N. Quincy Street
Arlington, Virginia 22217-5000

32. Southern Methodist University 1
Computer Science Department
Attn. Dr. Murat Tanik

Dallas, Texas 75275

33. Ohio State University 1
Department of Computer and Information Science
Attn. Dr. Ming Liu
2036 Neil Ave Mall
Columbus, Ohio 43210-1277

34. University of California at Berkeley 1
Department of Electrical Engineering & Computer Science
Computer Science Division
Attn. Dr. C.V. Ramamoorthy
Berkeley, California 90024

35. University of Maryland 1
College of Business Management
Tydings Hall, Room 0137
Attn. Dr. Alan Hevner
College Park, Maryland 20742

36. University of Pittsburgh 1
Department of Computer Science
Attn. Dr. Alfs Berztiss
Pittsburgh, Pennsylvania 15260

37. Research Administration 1
Code 012
Naval Postgraduate School
Monterey, CA 93940

38. Attn: Dr. David Hislop 1
Laboratory Command
Army Research Office
P. O. Box 12211
Research Triangle Park, NC 27709-2211

39. Monmouth College 1
Computer Science Department
Software Engineering Program
Attn. Jerry Powell
West Long Branch, New Jersey 07764

40. Dr. Joseph A. Goguen 1
Oxford University
Computing Lab
11 Keble Road
Oxford OX1 3QD
England