Title

Multi-Language Support in a Program Analysis and Visualization Tool

Author

Moskovics, Stuart

Publication Date

2000-06-20

Peer reviewed

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Multi-Language Support in a Program Analysis and Visualization Tool

A thesis submitted in partial satisfaction of the
requirements for the degree Master of Science
in Computer Science

by

Stuart Phillip Moskovics

Committee in charge:

Professor William G. Griswold, Chairperson
Professor William E. Howden
Professor Keith Marzullo

2000

The thesis of Stuart Phillip Moskovics approved:

Chair

University of California, San Diego

2000

To my parents

TABLE OF CONTENTS

Signature Page . . . iii

Dedication . . . iv

Table of Contents . . . v

List of Figures . . . vii

Acknowledgements . . . ix

Abstract . . . x

I Introduction . . . 1
A. Motivation . . . 1
B. Approaches to Multi-Language Analysis . . . 3
C. Hypothesis . . . 4
D. Results . . . 5
E. Overview of the Thesis . . . 5

II The StarTool . . . 6
A. The Star Diagram . . . 6
B. Star Diagram Operations . . . 9
1. Eliding Uninteresting Nodes . . . 9
2. Planning Program Restructuring . . . 10
C. History of the Star Diagram Structure . . . 10
1. Modification for Retargetability . . . 12
2. Language-Dependent Resources . . . 13
D. Adaptation Module Extensions . . . 14

III The Multi-Language StarTool . . . 15
A. User Interface Modifications . . . 16
B. Multiple adaptation layers . . . 16
1. An adaptation layer Mediator . . . 18
2. Mediation through a Hash Table . . . 20
3. Populating the hash table . . . 22
4. Multi-Language Elision Options . . . 23
C. Cross-Language Issues . . . 26
1. Conversion of SyntaxUnits . . . 28
D. Limitations of the approach . . . 31
E. Adaptation Layer Requirements for Multi-Language Support . . . 32

IV Discussion 34
A. Tool Implementation 34
B. Multi-Language StarTool 34
C. Usability 35
1. Elision Options 35
2. Star Diagram Displays 36
D. Reliance on 3rd-party tools 38
E. Performance 39

V Conclusion 41
A. StarTool Programs 41
B. Contributions of the Research 41
C. Future Work 43

Bibliography 44

LIST OF FIGURES

II.1 A Star diagram built for the variable rooms. 7

II.2 Searching for all references to the variable rooms. Double-clicking on one instance of the variable will bring up the specific section of code containing the variable. 8

II.3 After looking at an instance of the variable rooms, the rooms identifier can be added to the Star Diagram. 8

II.4 The various types of Star Diagrams that can be built. 8

II.5 The trimmed arms window, displaying sections of the Star Diagram that can be annotated and then removed from the view. . . 9

II.6 The StarTool Adaptation Module interface, which contains 18 operations. The identifier sub-tag `al` stands for *adaptation layer*; the tag `su` stands for *syntax unit*. 11

II.7 The adaptation layer relationship with the generic star diagram functionality and the language-specific program representation. . . 13

III.1 Multi-language retarget of StarTool using adapter classes. The *generic star diagram functionality* was not modified; *C-Tcl/Tk-Ada Adapter* is the mediating adapter containing the multi-language functionality. 15

III.2 Dialog box displaying the extensions that can be loaded into the Multi-Language StarTool. 17

III.3 The MultiLanguage StarTool Hash Table interface. 20

III.4 Functions in the adaptation layer that return SyntaxUnits. 23

III.5 Elision options in the C StarTool Polaris. 24

III.6 Elision options in the Tcl StarTool Twinkle. 24

III.7 Elision options in the Ada StarTool Firefly. 24

III.8 Original attempt at providing elision options in the multilanguage StarTool. 25

III.9 Elision options in the Multi-Language StarTool. 26

III.10 Multilanguage Star Diagram with all C identifiers similar to a C variable and all Ada identifiers similar to an Ada variable. The view has been elided to show that the Diagram pulls in nodes from both C and Ada sources. 27

III.11 Process for conversion of SyntaxUnits. This process occurs once per source file loaded into the Star Diagram. 28

III.12 Pseudo-code for conversion of SyntaxUnits to other adaptation layers. 30

IV.1 The desired "customizable stacking" options, similar to the elision window. 37

IV.2 The function that decides whether nodes are stackable for the display. 38

Acknowledgements

I would like to thank my advisor, Bill Griswold, for his enormous guidance and support. He has been extremely patient with me during my research and writing, and it has been a great pleasure to work with him.

I would like to thank Jim Hayes for helping me to understand the StarTool's retargability mechanisms. His fabulous work with redesigning the StarTool laid the groundwork for the multi-language extensions created during my research and implementation. I would also like to thank Jimmy Yuan for being a constant source of support during my undergraduate and graduate career.

I would like to thank my parents for their constant support during my school career. From my early childhood, they have always encouraged me to achieve. I would not have made it this far without them.

This work was supported by NSF Grant CCR-9508745 and UC MICRO Grant 98-054 with Raytheon Systems Corporation.

ABSTRACT OF THE THESIS

Multi-Language Support in a Program Analysis and Visualization Tool

by

Stuart P. Moskovics
Master of Science in Computer Science
University of California, San Diego, 2000
Professor William G. Griswold, Chair

Restructuring and analyzing software is difficult. Tools that allow programmers to view and plan modifications to existing programs can ease the burden of maintenance and change. Modern software engineering projects often use many different programming languages, including the use of multiple languages in a single project.

The StarTool is a program visualization and restructuring tool for software programs. This thesis discusses a method used to improve the Star Diagram's retargetability features by providing support for understanding multi-language software programs. Our research shows a simple and extendible mechanism to use single-language retargetable program analysis tools for multiple-language analysis.

Chapter I

Introduction

I.A Motivation

The computing industry has recently experienced substantial increases in available computer processing power and fast memory, allowing for larger and more complex software. The job of restructuring and enhancing such software is difficult and time-consuming. It is not uncommon for programmers to start work on a software project with minimal or no knowledge of the pre-existing system and code structure. Any method or tool to help the engineer understand program structure can be a valuable time-saver and assist in producing quality changes.

Large software projects are increasingly being written using multiple languages. Tcl/Tk is used to quickly create graphical user interfaces; it is also used because it is portable across platforms. Frequently, the interface portion of a program can be written in a language such as Tcl/Tk while the rest could be in another language. A computation-intensive program might require the efficiency of C, while a highly critical program dealing with a nuclear reactor would need the software safety of Ada. Programs written for Microsoft Windows commonly have their graphical user interface written in Visual Basic while the performance-sensitive code is written in Visual C++. Choice of programming languages can also involve the costs associated with their use. Studies have shown that a line of

Ada code costs about half as much as a line of C code, producing 70 percent fewer internal fixes [Zeigler, 1995]. Some languages also have better compiler and tool support than others, making their use more attractive to the programmer.

Many program analysis tools have been created and studied for program restructuring and understanding. However, there has been a lack of readily available tools that were capable of processing programs written in multiple programming languages. An excellent tool to analyze C would be completely useless for the portions of a project written in Ada. There aren't well-established methods of taking existing program analysis tools and combining them to be used for multiple languages. Generic tools such as UNIX *grep* can be used to search for identifiers in source files of multiple languages, but the results do not indicate a multi-language analsyis. Grep also lacks a graphical interface, minimizing the comprehensibility of its output. One option would be use a separate analysis tool for different languages; for example, to analyze a program written in C and Ada, the C code could be viewed in a C restructuring tool, while the Ada code is loaded in an Ada analysis tool. Unfortunately, this approach provides no means to integrate the separate analyses into one result. For example, attempting to locate identifiers and variables that are used across multiple languages would be very difficult. Multi-language tools are capable of examining cross-language issues that could not be economically explored with multiple single-language tools.

This is the problem faced by users of the StarTool, a program restructuring and analysis tool developed at the UCSD Software Evolution Laboratory [Griswold et al., 1996]. This tool builds Star Diagrams, graphical views of program elements that are customizable to the user. Hayes redesigned the StarTool infrastructure to allow easy retargetability to new programming languages [Hayes, 1998]. The new StarTool hides language-specific representation information in an *adaptation layer* containing 14 functions. A StarTool for a new language can be built by taking existing program representations and adding an interface through the creation of a language-specific adapter. Based on this interface, StarTools were built for C, Tcl/Tk, and Ada.

Raytheon, a defense, engineering, and aviation business with offices in California, has been a long-term user of the UCSD StarTool. The StarTools for both C and Ada have been beta-tested at Raytheon on their software. Raytheon has been one of the major motivators of a multi-language StarTool; since they have software that uses both C and Ada, they have requested a StarTool implementation that can help them to understand and restructure those types of programs.

I.B Approaches to Multi-Language Analysis

Through the use of a common representation approach, retargetable analysis tools are often usable for multi-language analysis. An example is a compiler that is capable of linking together object code that is derived from multiple source languages. By requiring the language-specific code generators to use a common representation in their output, multi-language linkers can understand and combine program representations from different languages.

The Computer Science Department at the Tennessee Technological University has developed a program called Poly CARE, a multi-language program analysis tool. Poly CARE was extended from the original CARE tool used to facilitate the comprehension of C programs. Using a graphical interface, Poly CARE's intended use is the comprehension and re-engineering of multi-language programs. Through user studies, the creators of Poly CARE found that engineers using the tool were 37% more productive when maintaining code than when not using the tool [Linos et al., 1993] [Linos, 1995]. The tool has two main modules, a code analyzer and a display manager. The code analyzer uses *flex* and *bison*, common UNIX tools for lexical analysis and parser generation. The lexer and parser for each language supported by Poly CARE will be implemented using the same tool-set. This reduces code-size and can help aid in efficiency and optimization. Unfortunately, this limits the use of readily available language parsers and program slicers, which could reduce the amount of work to integrate a new language into Poly CARE. A literature search into the mechanisms used by Poly CARE to integrate multiple-language information turned up very little information, so a complete analysis of its multi-language retargetability features was not possible.

I.C Hypothesis

We hypothesize that a single-language program analysis tool designed for retargetability can be extended into a multi-language tool by using a multiple-level adapter approach with a mediator. If the program representation specific to a source language is fully separated from generic display and analysis functions, multi-language capability can be enabled by mediating between the separate language instantiations and deciding which language implementation is involved in tool queries. This approach allows adding support for additional languages to a multi-language tool with minimal effort once the language-dependent portion of the tool has been created. By using a mediator with multiple-level adapters, the multi-language tool can understand issues specific to multi-language programs, specifically the sharing of information across multiple programming languages.

We decided to test our hypothesis on the program analysis tool StarTool. We hypothesized that by using Hayes's adaptation layer interface, a multi-language StarTool could be created without modifying any of the pre-existing code used to create the C, Tcl/Tk, and Ada StarTools. Moreover, we desired this new multi-language tool to be easily extendible; any new StarTool written for a new language could be integrated into our multi-language tool through the addition of a new adaptation layer and minimal modifications to the mediator. Any code to create the multi-language tool would be in *addition* to the pre-existing code, preserving the retargetability interface to allow for adaptations to new languages.

I.D Results

We successfully built two multi-language StarTools: one that supports C and Tcl/Tk, and another that supports C, Tcl/Tk, and Ada. These tools allow a programmer to load, display, and analyze source files from different languages in one tool. We created a mediator that was capable of handling different language representations by using a multi-level adapter approach. The mediators for the two StarTools were created in 100 hours of work and they use less than 2,000 lines of code. The requirement that we could not modify the previous retargetability structure was challenging but eventually proved that Hayes's interface allowed for a multi-language design. One example of the difficulty we encountered is the mechanism Hayes designed to interface with adaptation layers; this mechanism required that a StarTool had only one adapter built into the tool. The multi-language StarTool was built by working around this requirement. We were also able to structure our multi-language StarTool such that additional languages can be easily added to the interface.

The merged version of the single-language tools had no mechanism to recognize whether variables and procedures were used across multiple languages. We extended the identifier-matching mechanism to convert symbols in one language-specific adaptation layer to another language-specific adaptation layer. This extended StarTool is more useful to a user attempting to understand a multi-language program.

I.E Overview of the Thesis

Chapter II explains the Star Diagram structure. Chapter III describes the modifications to the retargetable Star Diagram structure we performed to support multi-language programs. Chapter IV discusses the usefulness and limitations of our approach for multi-language programs. Chapter V summarizes our work and presents opportunities for further research.

Chapter II

The StarTool

II.A The Star Diagram

The Star Diagram is a graphical tool that helps a programmer with program visualization and planning for program restructuring [Bowdidge, 1995]. Star Diagrams are built around specific information that the programmer is looking for in a set of source files. The programmer first loads a set of source files to be analyzed by the StarTool. A variable or identifier from one of the loaded source files is then chosen to be the main context of the Star Diagram, and the StarTool looks for all references to the chosen variable throughout the source files. The results are then displayed in a graphical format.

The Star Diagram content is a tree shown with the root at the left and the tree growing sideways to the right. The chosen variable becomes the root node of the Star Diagram and all references to that variable are its children. Any references to those children are the next level's children, and so on, until the leaf nodes are the source files containing the identifier. The Star Diagram *stacks* all nodes that refer to the same variable or operation. Stacked nodes appear as a single node but the node is drawn with other nodes behind it. This provides a compact but complete view of the sources, allowing the programmer to focus on a chosen aspect of restructuring. A leaf node's parent is the function within the source file containing the reference to the identifier. The Star Diagram is thus a tree that contains all of the direct and indirect uses concerning a specific object while excluding irrelevant source code.

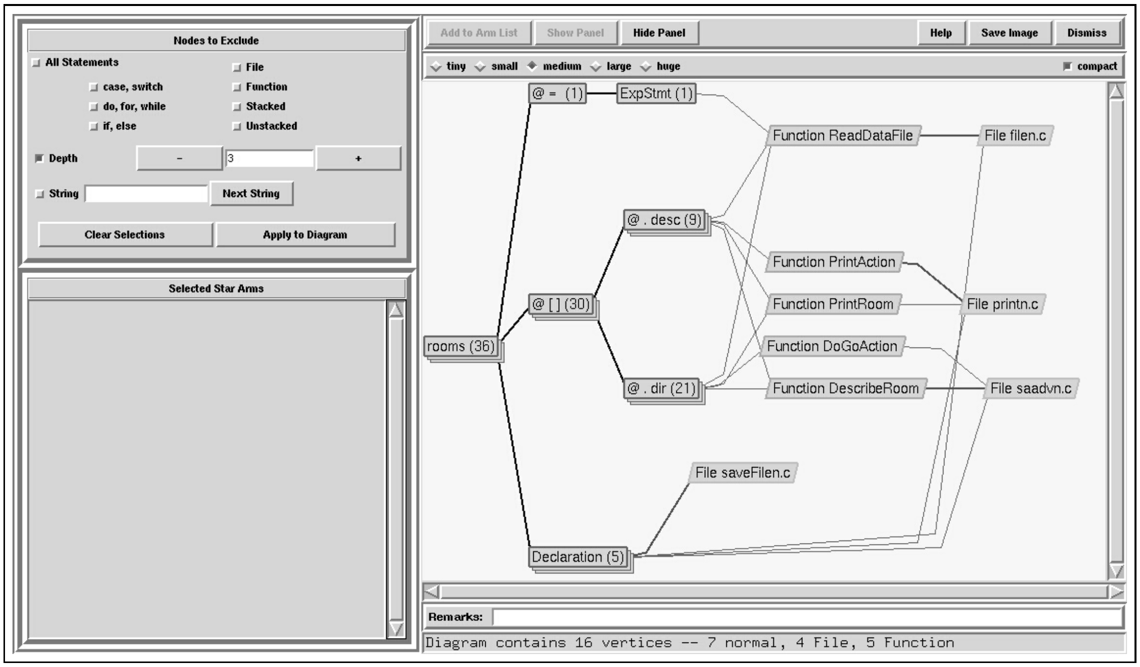

Figure II.1: A Star diagram built for the variable rooms.

The programmer has many choices for customizing what information is included in the Star Diagram. In addition to including all references to the same identifier, the StarTool can also build a Star Diagram including all identifiers with the same name, identifiers with the same type, and identifiers with the same underlying type. The programmer also has the option of including all identifiers that match a certain pattern by searching for matches based upon a regular expression. These options are included since the goal of the tool is not to make assumptions regarding how the programmer will perform their restructuring but to provide the capability to view the data in any way they see fit.

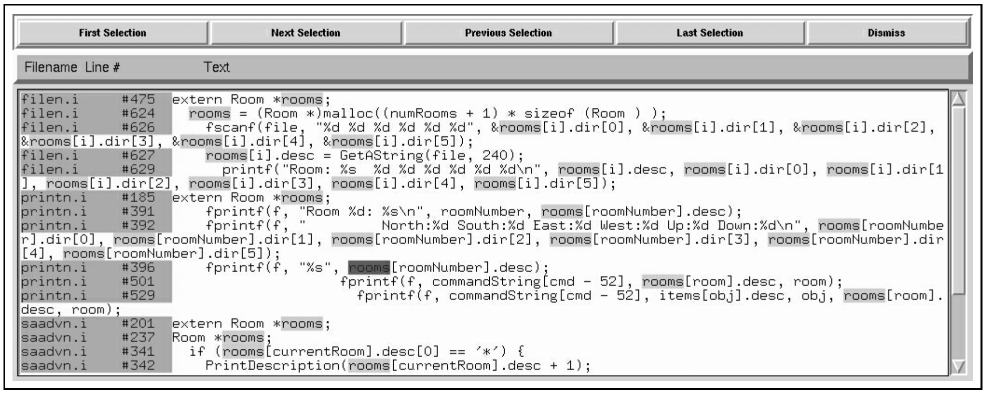

Figure II.2: Searching for all references to the variable rooms. Double-clicking on one instance of the variable will bring up the specific section of code containing the variable.

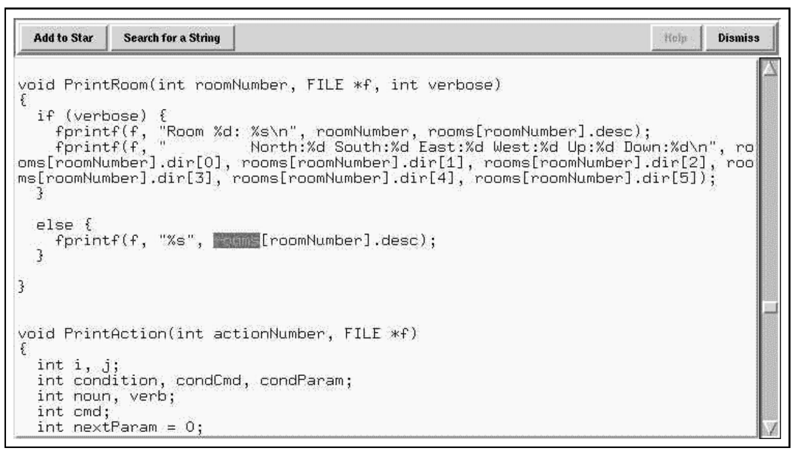

Figure II.3: After looking at an instance of the variable rooms, the rooms identifier can be added to the Star Diagram.

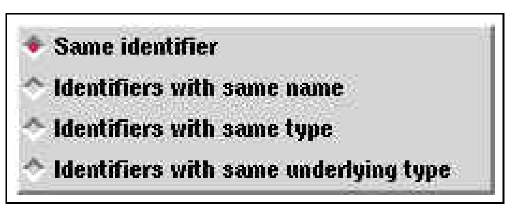

Figure II.4: The various types of Star Diagrams that can be built.

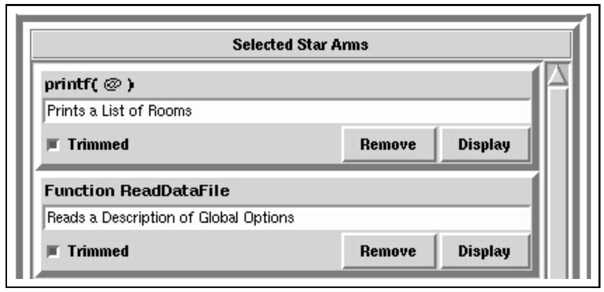

Figure II.5: The trimmed arms window, displaying sections of the Star Diagram that can be annotated and then removed from the view.

II.B Star Diagram Operations

The goal of the Star Diagram is to allow the programmer to view the important uses of program components to aid in restructuring. As you look at the diagram from left to right, you can see higher-level views of the use of a structure, from the actual identifier use to layers of function calls above the use of the identifier. The Star Diagram main window has three main components. The main window, shown on the right-hand side, contains all the nodes in the tree. The left-hand side contains the elision window and the selected Star arm window, shown on the top and bottom, respectively.

II.B.1 Eliding Uninteresting Nodes

Since any restructuring requiring the use of an analysis tool will most likely affect many program modules, a Star Diagram is capable of storing unlimited nodes, bounded only by available memory. This abundance of nodes can clutter the diagram and make it difficult to perform a restructuring. To improve the usability of the StarTool, a single-language Star Diagram allows for the elision of language-specific node types and nodes containing programmer-chosen strings. For example, a StarTool user might choose to ignore all function calls or conditional statements, choosing to focus on other aspects of the source.

II.B.2 Planning Program Restructuring

The ability to elide node types and certain strings from a Star Diagram is useful, but sometimes a programmer needs to remove whole sections of the diagram to focus on constructing a restructuring plan. The bottom-left side of the window contains a set of trimmed arms, or portions of the tree that have been removed from view. These arms would generally be parts of the display that are not related to the restructuring being performed. Each trimmed arm contains a description (the text from the root node that was shown in the diagram) and an optional text box that can be used for annotation. This provides the ability for the programmer to record a note describing the trimmed arm or maybe a potential restructuring on the arm. The trimmed arms also have push-buttons to re-include them in the Star Diagram or to build a new Star Diagram including only the trimmed arm.

II.C History of the Star Diagram Structure

The Star Diagram was created by Bowdidge as a program visualization user interface for tool-assisted software restructuring [Bowdidge, 1995]. Chen created a C Star Diagram Tool in 1996 with 5,000 lines of Tcl/Tk and 800 lines of C++ [Chen, 1996]. This code was written on top of an AST front end already written in C++ [Morganthaler and Griswold, 1995]. Chen added two facilities to the Star Diagram to aid in the use of the Star Diagram: elision and trimming. In 1998, Hayes invented a method for adapting the StarTool to different program representations, creating StarTools for C, Tcl/Tk, and Ada [Hayes, 1998]. The StarTool had always been used to study restructuring of C files. However, it is common for large software to be written in a combination of different programming languages. The StarTool itself includes a major portion of its functionality in Tcl/Tk. Previous authors of StarTool implementations have desired to use the StarTool to analyze a restructuring of the StarTool itself, providing the ultimate test of the StarTool usefulness. Elbereth, a Java-only StarTool that was written

```
int al_elaborate(int &argc, char *argv[]);

char *al_elision_attributes();
char *al_merging_attributes();
char *al_similarity_attributes();

/* Provides iteration of elements appropriately similar to #prototype#
   under/inside the #container#. */
SyntaxUnit first_similar_su(SyntaxUnit container, SyntaxUnit prototype, char *similarity);
SyntaxUnit next_similar_su();

/* Provides iteration of elements with #attribute# under/inside the #container#. */
SyntaxUnit first_su_with_attribute(SyntaxUnit container, char *attribute);
SyntaxUnit next_su_with_attribute();

/* Formerly the ast_parent operation. */
SyntaxUnit su_superunit(SyntaxUnit item);

/* Given a SyntaxUnit #item# and the #subunit# from which it was reached, returns a label
   indicative of #item#, possibly with an indication of which position #subunit# resides. */
char *su_label(SyntaxUnit item, subunit);

int su_skip_test(SyntaxUnit item);

struct FilePosition {
  int line, column;
};

char        *su_file(SyntaxUnit item);
FilePosition su_begins(SyntaxUnit item);
FilePosition su_ends(SyntaxUnit item);
SyntaxUnit   file_to_su(char *pathname);
char        *file_text(SyntaxUnit item);
char        *file_filters();
SyntaxUnit   file_range_to_su(SyntaxUnit container, FilePosition *range_begin,
                              FilePosition *range_end);
```

Figure II.6: The StarTool Adaptation Module interface, which contains 18 operations. The identifier sub-tag `al` stands for *adaptation layer*; the tag `su` stands for *syntax unit*.

in Java, was also created in 1998 but does not use the same retargetable code structure as the tools created by Hayes [Korman and Griswold, 1998].

II.C.1 Modification for Retargetability

Hayes restructured the StarTool with the goal of supporting retargetability to new languages by making the tool representation and language independent. The theory was that the StarTool could be adapted to existing program representations in a short amount of time as a means of retargeting the StarTool to different programming languages. Hayes realized the algorithms to build, elide, and display a Star Diagram could be language-independent if the language-dependent information was accessed via an interface common to all StarTools. In Hayes's implementation, the information required to parse and analyze a specific programming language's source files is kept in what Hayes termed an *Adaptation Module* (see Figure II.6). Using Hayes's structure, a single-language StarTool is built by the generic, language-independent StarTool submitting requests to the language-specific Adaptation Module. The language-dependent Adaptation Module is responsible for processing and storing the AST nodes that are built from source files. Functions included in an adaptation module are file-to-AST and AST-to-file mapping functions, node attribute functions, and AST traversal functions. This approach was successfully used to build three separate StarTools, *polaris*, *twinkle*, and *firefly*, each capable of working with C, Tcl/Tk, and Ada files, respectively.

Separation of the language-dependent implementation from the language-indepdent StarTool was achieved without excessive genericity via a *query* interface. Each StarTool feature was assigned an operation in the adaptation layer that returns a list describing the language-specific implementation. For example, to determine the merging attributes that are used in the Ada StarTool Firefly, the generic StarTool calls the adaptation layer function *al_merging_attributes*, which then returns a concatenated string containing *package*, *subprogram*, and *task*, which are the Firefly merging operations. Through this function call, the StarTool can han-

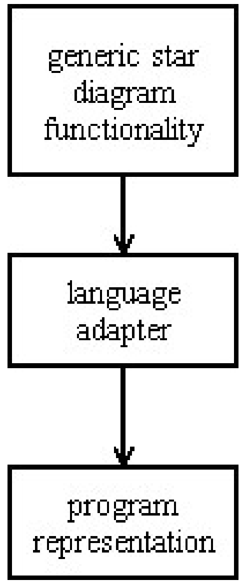

Figure II.7: The adaptation layer relationship with the generic star diagram functionality and the language-specific program representation.

dle any sort of merging parameters without having specific support requirements in the language-indepdent module. Elision, browsing, and similarity attributes are queried through the similar function calls.

II.C.2 Language-Dependent Resources

Hayes used readily-available program representations to prove the usefulness of his retargetability interface. The language-dependent portion of polaris, the C StarTool, uses the Ponder language toolkit [Griswold and Atkinson, 1995]. The Ponder toolkit generates program ASTs from C source files. Hayes didn't have a Tcl/Tk program representation readily available, so he built one himself. The Ada program representation came from the Gnu Ada Compiler Gnat [Dewar, 1994]. Gnat is a public-domain Ada 95 compiler and code-generator that integrates with the Gnu gcc compiler. Gnat's program representation is built with AST nodes containing information about program symbols. The Gnat compiler provides facilities for manipulating an AST representation of Ada sources.

II.D Adaptation Module Extensions

Since each single-language StarTool has a language-dependent and language-independent portion, Hayes created a generic *StarAdapter* C++ class that includes virtual functions with some default implementations that can be overridden in a language's Adaptation Module. To create an adaptation layer for a specific language, a language-dependent class needs to be built on top of the StarAdapter. The pure virtual implementations are replaced with language-specific functions, and the provided default implementations are overridden if needed. The super-classes built upon the StarAdapter are *IcariaStarAdapter* for C, *TclStarAdapter* for Tcl, and *GnatStarAdapter* for Ada.

The generic StarTool engine links with the language-dependent Adaptation Modules for each language's StarTool. However, each Adaptation Module uses a unique data structure to store AST Nodes and the other associated program representation information, such as type, scope, and line number. To allow all adaptation modules to share the same interface, information is passed between the generic StarTool and the Adaptation Modules via a *SyntaxUnit.* The SyntaxUnit is actually a *void ** in C, a generic data store that points to an area of memory. Using this approach, the representation- and language-independent StarTool interface has no concern as to the language and representation being used in the Star Adapters.

Chapter III

The Multi-Language StarTool

Our goal was to leverage the StarTool's retargetability interface to create a single StarTool capable of analyzing programs written in multiple languages. In addition, we prohibited ourselves from modifying Hayes's interface to create our new tool. The architecture we designed to support a multi-language tool can be found in Figure III.1.

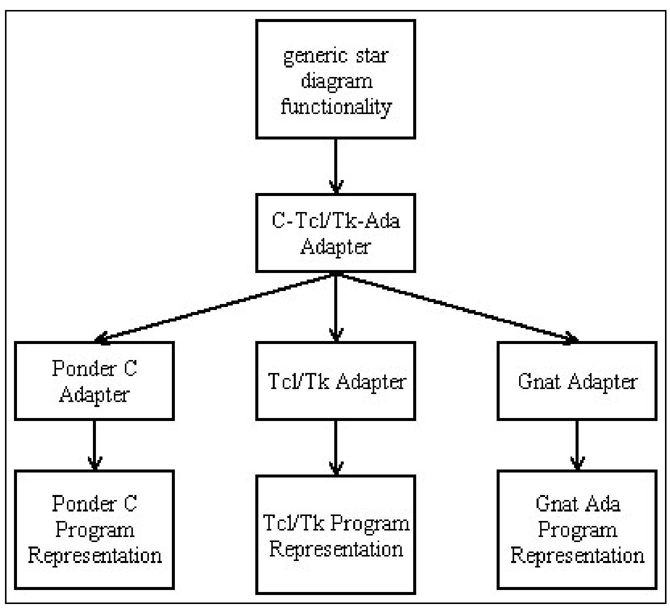

Figure III.1: Multi-language retarget of StarTool using adapter classes. The *generic star diagram functionality* was not modified; *C-Tcl/Tk-Ada Adapter* is the mediating adapter containing the multi-language functionality.

III.A User Interface Modifications

The user interface for the three single-language implementations of the StarTool all use a common interface implemented in Tcl/Tk. The Tcl/Tk source-code is completely representation- and language-independent. Since this Tcl/Tk code was already structured to handle ASTs from various language implementations, there were no changes required to the user interface portion of the Tcl/Tk code to build a multi-language tool. Any language-specific information that was required for display on the interface (such as the programming language supported by the specific tool or the file extensions to be loaded) was retrieved through the Adaptation Module via a query interface containing 14 functions. Therefore, the user interface created by Hayes to support retargetable StarTool implementations was readily adaptable to multi-language StarTools. The only modifications needed for multi-language support were the 14 query functions in the adaptation layers.

III.B Multiple adaptation layers

We modified one function in the adaptation module to support loading files from multiple languages, *file_filters*. The *file_filters* function returns the file mask used for displaying the default files to be loaded into the StarTool. The programmer has the option of loading files into the StarTool by specifying files or a file-mask on the command-line, or they can choose the *Load Files* option which brings up a dialog for choosing files. The *file_filters* function in the C StarTool used **.c*.i* (*.i refers to .c files that have already been run through a pre-processor), the Tcl StarTool used **.tcl*, and the Ada StarTool used **.adb*.ads*. For the multi-language tool, the *file_filters* function combines these file-masks to return all of the file filters as a combined string. Thus, the Tcl/Tk file load window for the multi-language tool allows for the loading of C, Tcl/Tk, and Ada source files, as can be seen in Figure III.2.

On the surface, it seemed possible to simply take all of the separate

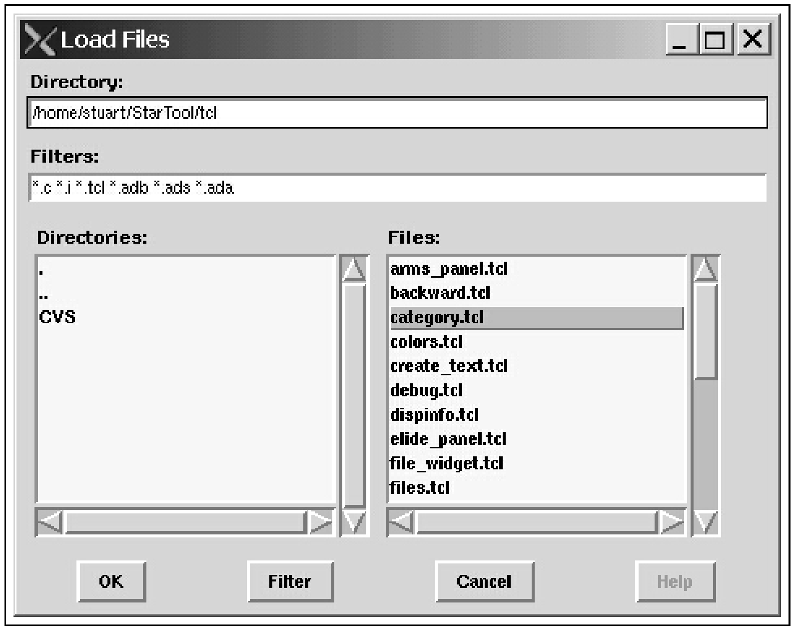

Figure III.2: Dialog box displaying the extensions that can be loaded into the Multi-Language StarTool.

adaptation layer implementations and link them together. However, one aspect common to the single-language tools is that each tool instantiation contains only one adaptation layer. Moreover, each of the adaptation layers uses the same exact function names to help with project management. If the IcariaStarAdapter (the C StarTool adaptation layer) is processing a SyntaxUnit, it assumes that the SyntaxUnit is always an Icaria AST Node cast to a *void **. Under no circumstance is the IcariaStarAdapter prepared to receive a SyntaxUnit that is actually a Tcl/Tk AST Node. It was obvious that while Hayes created a completely retargetable interface, this structure was not originally intended to be included in a multi-language tool.

The adaptation layers and the code that handles the calls to the adaptation layers are found in two modules in each implementation. The C implementation uses polaris.cxx and IcariaStarAdapterClass.cxx, the Tcl implementation uses twinkle.cxx and TclStarAdapterClass.cxx, and the Ada implementation uses firefly.cxx and GnatStarAdapterClass.cxx. The polaris.cxx, twinkle.cxx, and firefly.cxx files all contain same-named function calls that are one layer above respective calls in the adapter modules; the upper layer functions are wrappers for the actual adaptation modules. However, since these upper layers use the same name and prototypes, they are not available for inclusion in a multi-language tool.

III.B.1 An adaptation layer Mediator

We proceeded to integrate the multiple adaptation layers into a single codebase, allowing a single StarTool to process multiple languages. Since the functions one-layer above each of the adaptation layers had the same function name, we created a merged upper-layer that would serve as a *mediator*. The mediator receives requests intended for one of the adaptation layers and chooses which adaptation layer receives the information; the mediator also processes language-specific information returned by the mediators. The modules polaris.cxx, twinkle.cxx, and firefly.cxx were combined into one single module, twinklepolaris.cxx for the C/Tcl-Tk StarTool and twinklepolarisfirefly.cxx for the C/Tcl-Tk/Ada StarTool. This mediator is responsible for all functionality found in the upper layer of the single language tools.

All information is passed between the StarTool user interface and the adaptation layers as generic SyntaxUnits; these memory locations provide no information regarding identifier context or the information stored at the SyntaxUnit's memory address. The common interface used to process information in the adaptation layers made the merging of the adaptation layers easy. However, this generality created difficulty in merging the implementations. Our main goal was to provide multi-language capability using Hayes's retargetable adaptation layer without modifying his structure. In the single language tools, the language independent code never required a decision regarding which Adaptation Module should receive a SyntaxUnit. In a multi-language tool, SyntaxUnits can be processed by the IcariaStarAdapter, TclStarAdapter, or GnatStarAdapter. The general *void ** associated with each SyntaxUnit provides no means to indicate to which language (and to which language implementation) a SyntaxUnit is associated.

One obvious solution would have been to change the structure of the SyntaxUnit, adding a specialized data store that included object type and language information. This would have required changing the rest of the StarTool implementation, including the single-language adaptation layers which are out of our control, since they are developed by others. Another possibility would have been to combine the multiple adaptation layers into one large adaptation layer. This choice was avoided since the addition of another language to our multi-language tool or modification of a pre-existing language would be difficult since language information previously stored in a single-language module would be exposed to other language implementations. It was important that the effort to add a language to the multi-language StarTool be incremental and non-redundant. We desired to add onto the representation-independent structure without sacrificing the ease of adapting another language into the multi-language tool.

Our solution was to create a mediator responsible for associating SyntaxUnits with languages. We analyzed several approaches to handling this task. One possibility was to create an address pool from where the SyntaxUnits would be distributed. For example, any SyntaxUnit with a memory address from 0 through 10,000 would be a C AST Node, while 10,000 through 20,000 would be a Tcl/Tk AST Node. This approach would not be very efficient as it would require allocation of memory that will probably not be used during the operation of the StarTool. It also is not robust as it intrinsically requires hard limits on the number of AST Nodes that could be loaded into the tool from any implementation. It would be possible to allocate extra memory during run-time to extend these pointer allocations, but this approach would require the program to pause for allocation and to modify its table of language-pointer associations, forcing the user to wait for the program to adjust itself. In order to process a very large software package, a full recompile of StarTool would be necessary to change these pointer settings, which is not very desirable. This approach might also require the address pool to have knowledge of the Operating System and architecture, since code working

```
AssociateSyntaxUnitToLanguage(SyntaxUnit, Language)
{
        Language_SyntaxUnit_Map[SyntaxUnit] = Language;
}

GetLanguageFromSyntaxUnit(SyntaxUnit)
{
        return Language_SyntaxUnit_Map[SyntaxUnit];
}
```

Figure III.3: The MultiLanguage StarTool Hash Table interface.

with pointers might not be portable to every platform.

III.B.2 Mediation through a Hash Table

Since the address pool was unworkable, we decided to implement a hash table. The advantages of the hash table are that it is simple, easy to understand, and easy to implement. The disadvantage of this approach is that the hash table requires extra space to store its information, dependent on the hash table's internal data structure. The hash table would take as input a SyntaxUnit and return the language associated with the specified SyntaxUnit. Implementing the hash table required providing two operations, shown in Figure III.3:

We used the STL (Standard Template Library) *map* [ANSI, 1997] as the basis for the hash table. The STL *map(Key, T, Compare)* supports unique keys and provides for fast retrieval of another type T based on a given key. STL map is implemented using red-black trees, so the time to insert a SyntaxUnit into the hash table or to retrieve the language associated with a SyntaxUnit is of the order O(log n) [Cormen, et al., 1997]. However, the simplicity of the hash table did not come without added costs. Memory space is required to store the hash table entries. Each hash entry contains a *void ** pointer and an associated integer indicating the language (and adaptation layer implementation) that a SyntaxUnit was generated from. On a 32-bit machine, each hash table entry requires 8 bytes of memory, in addition to the STL data structure overhead. We felt that this was a reasonable requirement to support multi-language Star Diagrams without modifying the adaptation layer structure.

With the capability to associate SyntaxUnits and languages in place, the change to the adaptation layers proved straightforward. Most of the functions that work with Syntax Units have one of these characteristics: 1) The function receives an identifier (a filename or an enumerated language type) indicating the language being worked with, or 2) The function is passed in a SyntaxUnit that provides context information since it has already been mapped to a source language. In these functions, we call *GetLanguageFromSyntaxUnit* to determine the Syntax Unit's source language and which adaptation layer implementation should be called. This builds upon Hayes's approach so that the representation-independent module does not have knowledge of the separate language implementations. Since the data processing by the central adapter is completely transparent to the adaptation layers, each implementation does not need to know that their AST data is passed through a central adapter before their own adapter.

The exceptions to these rules are the function pairs {*first_similar_su* and *next_similar_su*} and {*first_su_with_attribute* and *next_su_with_attribute*}. The *similar_su* functions are used to look for a SyntaxUnit that is similar in a certain way to another SyntaxUnit, while the *su_with_attribute* functions locate a SyntaxUnit containing a certain attribute. The *first* function is always called to start the search process; if a matching SyntaxUnit is found, more SyntaxUnits can be found through successive calls to the *next* functions. The *next* functions do not receive a SyntaxUnit as a parameter, which poses a problem for the mediator since no language context can be found.

The lack of a language indicator as an input to the *next* functions was not problematic in the single-language StarTools since there was only one adaptation layer that could receive a *next* call, obviating the need for a language lookup. For the multi-language tool, we cache the language that is used in a *first* call; each time a *next* function is called, the cached language value is used to determine which adaptation layer to be called. It is not legal that *next_similar_su* could be called after *first_su_with_attribute* sets the global value, or vice versa, since the StarTool requires the appropriate *first* call before a subsequent *next* call can go through. Setting and checking this global data value does require extra overhead, including a data assignment for each *first* call and a data comparison for each *next* call. However, these operations require a small amount of time and are reasonable, considering that it allows us to use Hayes's retargetable adaptation layer for multi-language processing.

III.B.3 Populating the hash table

We considered associating all of the nodes within a source file with its source language by starting at the root node for a file and iterating through all of the nodes, assigning each node individually. Since the adaptation layer doesn't have a mechanism to iterate through all of its nodes, we would have been forced to modify the language-dependent adaptation layers, violating one of the goals of our work. An all-node iteration might also cause large delays during the initial processing of loaded source files. Instead, we located all of the adaptation layer functions that return SyntaxUnits and captured the return values in the merged upper layer. A total of 8 functions within the upper merged layer, located in Figure III.4, return SyntaxUnits. When one of these functions returns a SyntaxUnit, *AssociateSyntaxUnitToLanguage* is called with the returned SyntaxUnit and its associated language. Two other functions, *file_to_su* and *file_range_to_su*, process files and return a SyntaxUnit representing the file. They are able to use the file-name extension (*.c*.i for C, *.tcl for Tcl/Tk, *.adb*.ads for Ada) to associate the newly created SyntaxUnit with a language. The rest of the functions either set the cached last-language value or retrieve its contents for language context. By isolating the language association operations to the functions that return Syntax-

```
SyntaxUnit first_similar_su(SyntaxUnit originalSyntaxUnit)
SyntaxUnit next_similar_su()
SyntaxUnit first_su_with_attribute(SyntaxUnit originalSyntaxUnit)
SyntaxUnit next_su_with_attribute()
SyntaxUnit su_superunit(SyntaxUnit originalSyntaxUnit)
SyntaxUnit su_subunit(SyntaxUnit originalSyntaxUnit)
SyntaxUnit file_to_su(char *filename)
SyntaxUnit file_range_to_su(FileRange theFileRange)
```

Figure III.4: Functions in the adaptation layer that return SyntaxUnits.

Units, we were able to create a process that can be extended to more languages with minimal effort. Modifications required for adding support for a new language association involve only 8 functions and less than 100 lines of code.

III.B.4 Multi-Language Elision Options

Elision options are passed through the adaptation layer via three functions, *al_browsing_attributes*, *al_elision_attributes*, and *al_merging_attributes*. As an example, the merging attributes returned by the Tcl StarTool are *file* and *proc*, while the Ada StarTool returns *Program*, *SubProgram*, and *Task*. Since programming languages do not have constructs that always map to each other, we encountered a difficult issue regarding how to display elision options to the user.

For our original multi-language tool, we originally proposed to take the union of all of the attributes and present them to the user. This provides complete flexibility to the programmer, allowing the elision of certain types of nodes from one language implementation, while keeping them in another language implementation. A view of the elision window using this methodology can be seen in Figure III.8.

This interface was too cluttered to actually be useful. Previous user studies with the Star Diagram have shown that a poorly designed interface can frustrate the StarTool user, reducing the usefulness of the tool [Cabaniss, 1997].

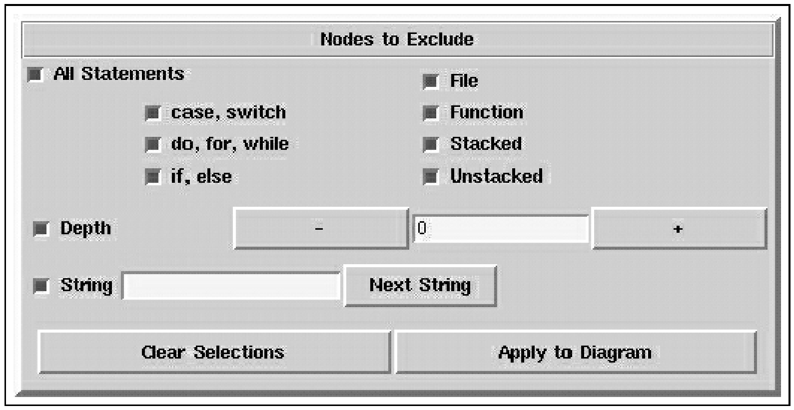

Figure III.5: Elision options in the C StarTool Polaris.

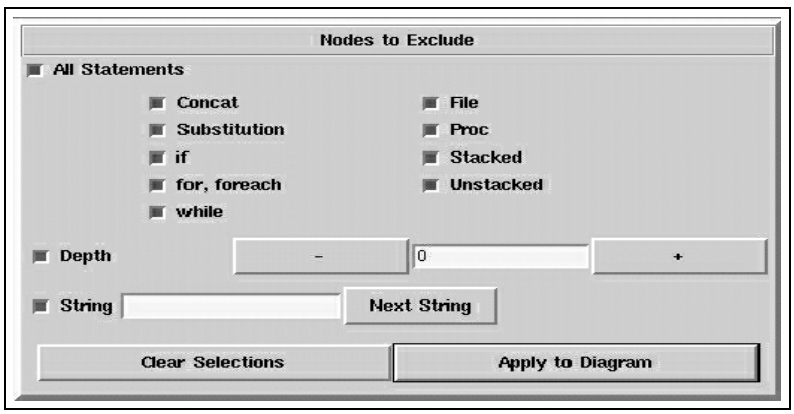

Figure III.6: Elision options in the Tcl StarTool Twinkle.

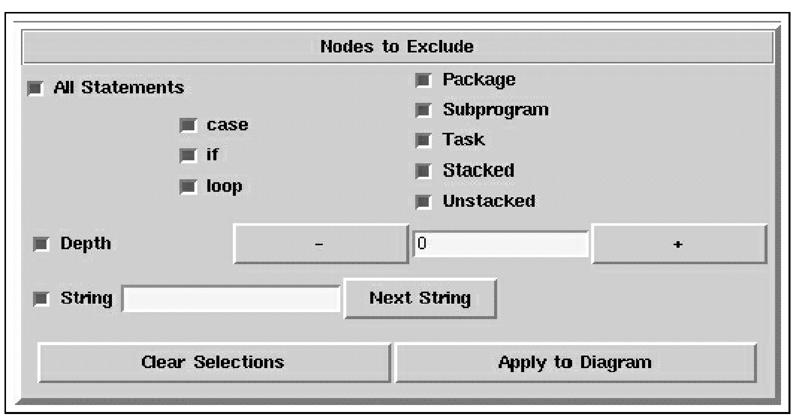

Figure III.7: Elision options in the Ada StarTool Firefly.

Figure III.8: Original attempt at providing elision options in the multilanguage StarTool.

We decided that the programmer wouldn't actually want to think about low-level language constructs as part of an overall multi-language program restructuring. Rather, they would be focusing on whole-program analysis and would prefer to operate at a higher level. To provide this interface, we merged the elision/merging attributes into more generic *groups* of attributes for compact presentation to the programmer. The attributes that are available for elision in the multi-language tool are *Conditional Statements*, *Loop Statements*, *Case Statements*, *Compilation Units*, *Functions*, and *Tasks*. The new elision panel can be seen in Figure III.9.

The mediator used for the language and SyntaxUnit association is also used for language and attribute associations. When the mediator receives one of the high-level attribute groups, it determines the language that will be receiving the language-specific attribute and *converts* the generic group into the language's appropriate attributes. For example, if the mediator receives *Conditional State-*

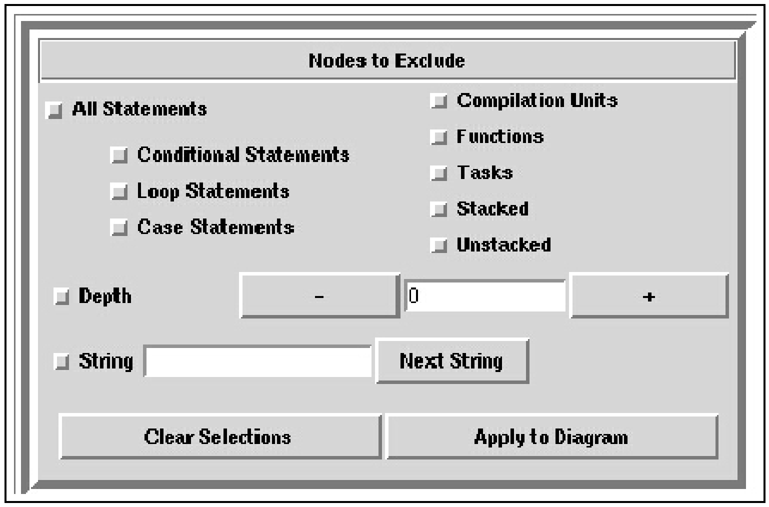

Figure III.9: Elision options in the Multi-Language StarTool.

ments, the C StarTool receives *if*, the Tcl StarTool receives *if* and *else*, and the Ada StarTool receives *if*. As with the SyntaxUnit mediation, this conversion is transparent to the adaptation layers.

III.C Cross-Language Issues

After the mediators were added to handle language and attribute mapping, we were successfully able to load source files from multiple languages into one Star Diagram using a single StarTool executable. Files containing C, Tcl, and Ada extensions were easily loaded into the StarTool for processing. For example, the user could build a diagram containing all of the C nodes similar to a C variable and all of the Ada nodes similar to an Ada variable, as seen in Figure III.10. This would require a two-step process, first adding the C identifier, then adding the Ada identifier. The programmer could also *search* for all instances of a text pattern across multiple-language sources and then add the results to a Star Diagram. Although these Star Diagrams are interesting to look at and quite useful to a programmer, we realized that a multi-language tool needs to do more than process

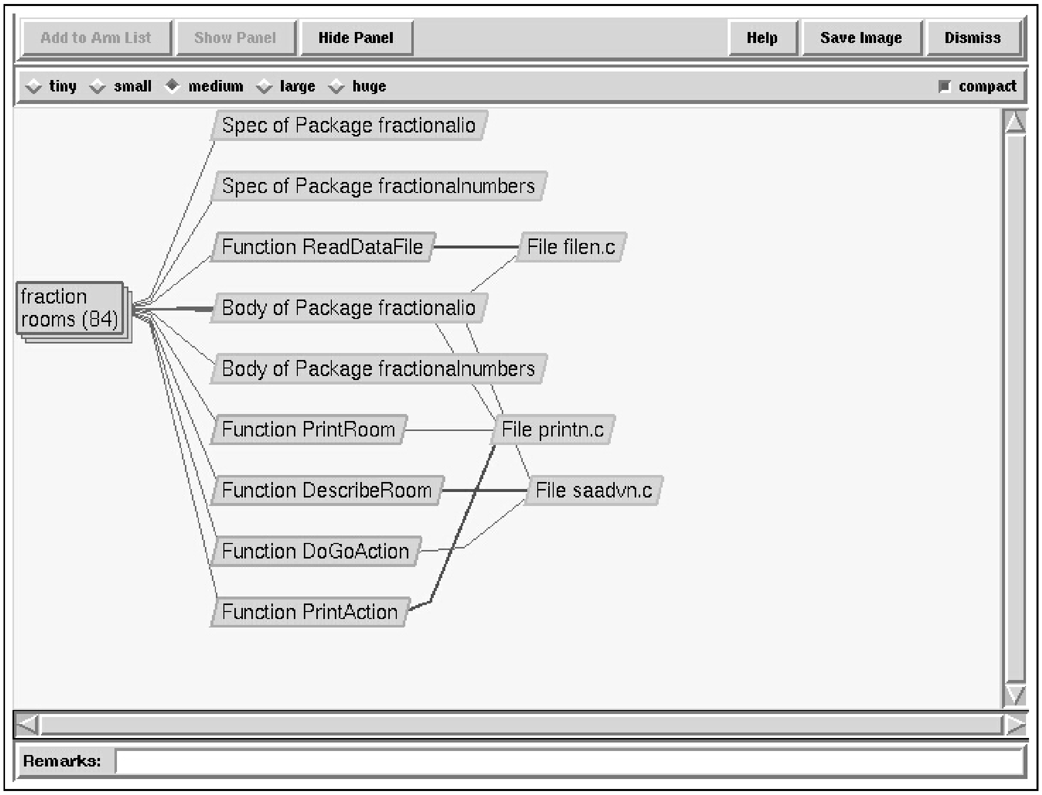

Figure III.10: Multilanguage Star Diagram with all C identifiers similar to a C variable and all Ada identifiers similar to an Ada variable. The view has been elided to show that the Diagram pulls in nodes from both C and Ada sources.

sources from multiple languages. To be fully useful, the tool needed to understand cross-language issues that do not exist in single-language programs. The nature of a true multi-language program is that some of the variables or functions are shared across multiple languages. A C function might call a Tcl function, or an Ada function might change or access a variable that is declared in a C file. We desired a Star Diagram built on a cross-language identifier to automatically include any instance of the identifier in every language source loaded into the StarTool. This sort of multi-language view ensures that the programmer will see any occurrence of the use of an identifier across all languages, helping to reduce the possibility of software errors.

III.C.1 Conversion of SyntaxUnits

Since each adaptation layer may use unique methods and data structures to store language representation, SyntaxUnits created by one adaptation layer are not correctly processed by other adaptation layers. To use Hayes's adaptation layers without modification, we created a temporary *dummy* SyntaxUnit that can be correctly parsed by other implementations. The dummy SyntaxUnit contains the information represented by the old SyntaxUnit but in the correct data structure format for another implementation.

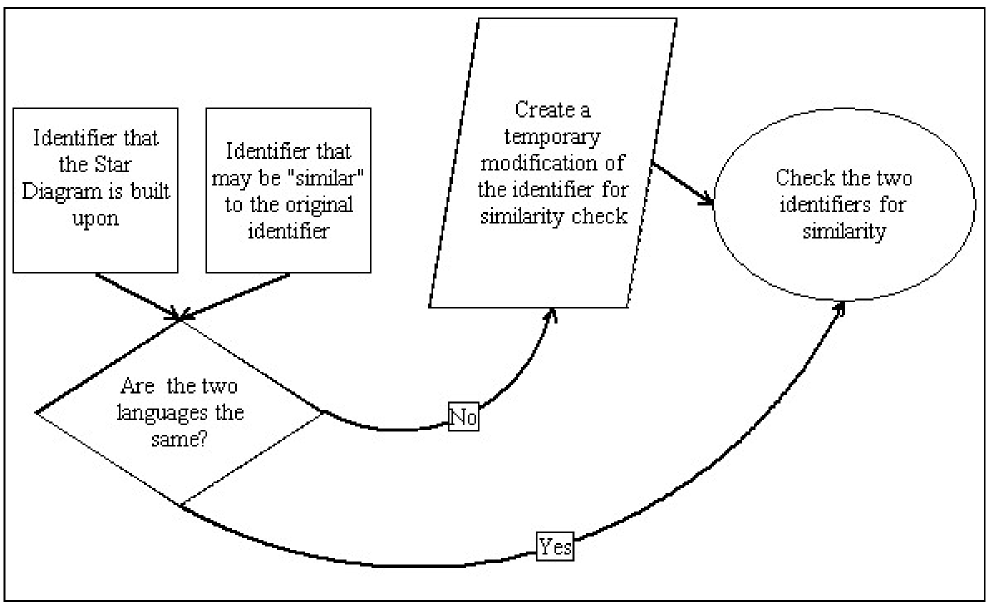

Figure III.11: Process for conversion of SyntaxUnits. This process occurs once per source file loaded into the Star Diagram.

When the StarTool builds a Star Diagram, it searches for identifiers that are *similar* in some chosen way to the specified identifiers. For example, the StarTool might be asked to search for identifiers that have the same name, type, or the same underlying type as chosen identifiers. The language-independent front-end has a function called *insert_similar_nodes* that is responsible for filling a Star Diagram with objects that are similar to a chosen object. This is performed through a call to *first_similar_su* and successive calls to *next_similar_su*. When searching for similar identifiers, *first_similar_su* in the multi-language tool checks the language of the source identifier and the language of the identifier to be compared. If they are the same, the tool lets the similarity check proceed as it did in the single-language tools. Otherwise, a *dummy* temporary SyntaxUnit is created to encapsulate the original SyntaxUnit information for the specific implementation. The function *ConvertSU* receives the node to be converted along with the destination language, which then calls either *ConvertSU_C*, *ConvertSU_Tcl*, or *ConvertSU_Ada*. This process can be seen in Figure III.11.

The conversion of SyntaxUnits from one implementation to another requires filling in an identifier's *label* (or name), its *kind*, and its *scope*; this information is required to search for an identifier that is *similar* to an identifier from a different language. The first step for creating a dummy SyntaxUnit for language conversion is to allocate a new AST Node for the target representation. The C adaptation layer uses an AST Node type defined in the Icaria library, the Tcl type is a custom AST representation built by Hayes, and the Ada adaptation layer uses a combination of Gnat program representation information and other StarTool-specific AST information. This process requires a memory allocation for the new AST Node.

The new SyntaxUnit then needs to take on the *label* contained by the old SyntaxUnit, which is retrievable through the *su_label* function. This step may require some conversion of the label, since Hayes's retargetable interface does not guarantee that a label from one adaptation layer will match a label from another adaptation layer. Also, some parsers might perform name mangling on an identifier that would need to be processed.

The conversion routines then fill in *kind* information. Objects in Star Diagrams can hold many different attributes including AST Identifier, Variable, Declaration, etc. Reconciling the different types among the adaptation layers proved to be one of the difficult steps in creating a multi-language tool. The Icaria Libary

```
ConvertSyntaxUnit(SyntaxUnit oldSyntaxUnit, SyntaxUnit newSyntaxUnit,
                  Language newLanguage)
{
        newSyntaxUnit.allocateMemory();
        newSyntaxUnit.label =
                CreateNewLabel(oldSyntaxUnit.getLabel());
        newSyntaxUnit.Child = NULL;
        newSyntaxUnit.Parent = NULL;
        newSyntaxUnit.Siblings = NULL;
        newSyntaxUnit.IdentifierType =
                NewIdentifier(oldSyntaxUnit.getIdentifierType());
        AssociateSyntaxUnitToLanguage(newSyntaxUnit,newLanguage);
}
```

Figure III.12: Pseudo-code for conversion of SyntaxUnits to other adaptation layers.

in the C StarTool has 14 types, the Tcl tool supports 3 major types, and the Ada tool has 211 major types. The Ada interface includes much more types since it was built directly on an Ada language parser, while the Tcl interface was hand-built and the C interface used a C program slicer. The issues created by these large differences in type information are addressed in Chapter IV.

Lastly, *scope* information needs to be filled in. The C adaptation layer's AST Node has data values for parent, child, left sibling, and right sibling. Hayes's Tcl AST Node has a data field for children; a new Tcl AST Node automatically sets its parent and sibling to NULL. The Ada AST data structure also has values for parent, child, and sibling. To correctly compute scope information when searching for the *same identifier*, all of these data fields need to be filled in.

After the label, kind, and scope are set, the converted AST node is associated with the target language in the language hash table. The advantage of this approach is that only one conversion routine needs to be written for every language that is added to a multi-language StarTool. However, this conversion routine needs to be aware of the other adaptation layer's representations to be sure that it is fully compatible with the other languages supported by the StarTool. This requirement creates some work for the implementer when adding a new language to the multi-language tool. One side benefit is the AST node conversion is optional, or it can be delayed after introduction of a new language. If a cross-language conversion needs to be performed but is skipped, the StarTool will process the incorrect SyntaxUnit and consider it a non-match when looking for similar nodes. Multi-language Star Diagrams could still be built with such a tool, but cross-language identifier searches will not include identifiers that are located in source files containing the newly added language. Cross-language variable searching might work if the programmer chooses an identifier in the new language; since the conversion routine will already have been written for the other languages, the tool will most likely correctly convert the SyntaxUnit from the newly added language. Choosing a variable in an already implemented language would not find the variable in the new language without the new language's conversion routine.

III.D Limitations of the approach

We were limited by the amount of cross-language variable searching we could perform in the multi-language tool. Since the multi-language StarTool does not have access to the full parse trees of the source files that are loaded, we are not able to extract full information concerning AST nodes. Therefore, we had to make some concessions concerning the ability of the multi-language tool to do Cross-Language searching.

Lacking a full parse-tree, the multi-language StarTool needs heuristics when looking for the *same identifier* across multiple languages. Some programming languages have certain commands that are used to register identifiers or variables that are declared in another language. The StarTool uses *Tcl_Create_Command* to register functions in C that are called in Tcl. Without full AST information from the adaptation layer, the multi-language tool does not know which identifiers have been mapped via a cross-language registration function. Moreover, each language has its own method for registering another language's identifiers and this registration process can be dynamic.

We made the assumption that if the user is looking for the *same identifier* to one that is declared in a procedure, any identifier that has a global scope with the same name and type is considered a match. The same goes for performing a search on a global identifier; it will only find identifiers with a local scope, not global ones. This provides a high confidence that the multi-language tool is finding the information that the programmer is looking for. One requirement for this search to succeed is that if an identifier is used across multiple languages, it must be named the same (have the same label) in every implementation. Since using the same name would be good programming practice, we considered this requirement to be reasonable.

III.E Adaptation Layer Requirements for Multi-Language Support

Our multi-language StarTool can be extended to support more languages if an adaptation layer has been created for the new language. For multi-language support, the single-language adaptation layer needs to be widened through the following functionality:

1. A *make_dummy* function. The new tool must support the creation of a temporary fake SyntaxUnit to search for identifiers in the new language that are similar to chosen identifiers from other languages.

2. A *remove_dummy* function. The new tool must support the deallocation of the dummy SyntaxUnit after it is no longer needed.

3. A function to identify the *name*, *scope*, and *kind* associated with an identifier for the new language.

The difficulty associated with creating these functions was directly related to the data structures used for each adaptation layer. Creation of these functions for the C adaptation layer was the most difficult, due to the complex and multi-layered data structures used by the Icaria toolkit. Tcl/Tk was the easiest language to support, since Hayes created a clean and simple adaptation layer for the Tcl/Tk StarTool. The simplicity of the Tcl/Tk language also simplified the creation of these functions.

Chapter IV

Discussion

In this section we discuss the results of our project as well as an evaluation of the design and its limitations.

IV.A Tool Implementation

The multi-language transformation to the retargetable StarTool was implemented entirely in the C++ programming Language. The C/Tcl-Tk/Ada tool and C/Tcl tool has 21,000 and 16,000 lines of code, respectively; 2,000 lines of code in each multi-language tool is for multi-language support. These totals do not include the Icaria and Gnat libraries. The gnu g++ compiler was used for compilation of the C and C++ sources, and the Gnat add-on for gcc was used to compile the Ada sources. The Gnat compiler is also used for processing Gnat sources when they are loaded into the StarTool.

IV.B Multi-Language StarTool

Our work has produced an extendible multi-language StarTool that can be used to analyze and restructure software written using multiple programming languages. Moreover, we were able to implement our design without modifying the retargetable StarTool interface created by Hayes or the generic user interface. Our framework allows for new languages to be integrated into the StarTool with minimal effort once an adaptation layer has been written for that new language. Cross-language variable searching can be functional with the new adaptation layer through the creation of 3 extra functions to support dummy SyntaxUnits. This provides an incentive to the programmer deciding whether to retarget the StarTool to a preferred new language.

IV.C Usability

We encountered many issues when trying to create a usable interface for a multi-language program analysis tool that would be used by programmers with different programming assumptions and styles of work. Having very little previous research in this field to use in our efforts, we had to make some educated guesses concerning the use of the tool.

IV.C.1 Elision Options

The Elision Options are part of what gives the StarTool its uniqueness; they provide the ability to hone-down a Star Diagram view to support the programmer's needs. In a multi-language tool, the programmer can either be thinking in a multi-language or a single-language perspective. However, the StarTool will always display all of the information from each language loaded into the tool. We desired to give the programmer the maximum flexibility in eliding all possible nodes, shown in Figure III.8. Unfortunately, this made the interface seem cluttered and could overwhelm the StarTool user. We also felt that providing the user with *too much* functionality might be a reason to not use the tool. The multi-language StarTool's merged categories, shown in Figure III.9, is our attempt to provide flexibility to the programmer while keeping the interface as language-generic as possible. Another option for the elision window would be to provide a new elision panel containing one language's elision options for each language loaded into the StarTool. Unfortunately, this scenario is not preferred since the StarTool will run out of window space as more languages are added to the StarTool. Requiring the programmer to scroll through panels of elision options to find what they are looking for would be counter-intuitive.

The elision categories provided might also be problematic to the user. We created the generic label *Compilation Units* to represent the C and Tcl *File* and the Ada *package*. However, an Ada user might feel that a *package* does not belong in the same category as a file. The Ada *task* construct also did not seem to fit in with any of the other language's elision options, so we left Task as an elision option by itself, providing more language-specific information in the elision window than we'd prefer.

IV.C.2 Star Diagram Displays

A multi-language StarTool user is able to retrieve the language associated with an on-screen node by viewing the file or package that the node derives from, assuming the user has not elided that information from the view. A possible improvement to the StarTool would be to add color information to indicate the language associated with an AST Node; the use of color might aid in restructuring by helping to locate cross-language dependencies. The user also has the option of double-clicking on a node to bring up the associated source code to discover the AST's language, but this operation may become tedious. Double-clicking on a stacked node will bring up a listing of all the source files, displaying the nodes' language information. Node display could be further differentiated by using a separate color or box demarcation for identifiers that are used across multiple languages. We would also like to give the StarTool user the option of viewing the Star Diagram with *generic labels*. For example, all function calls would be labeled *function*. A Star Diagram with generic labels might help the user to simplify their restructuring process by stacking chosen identifiers with a common generic label.

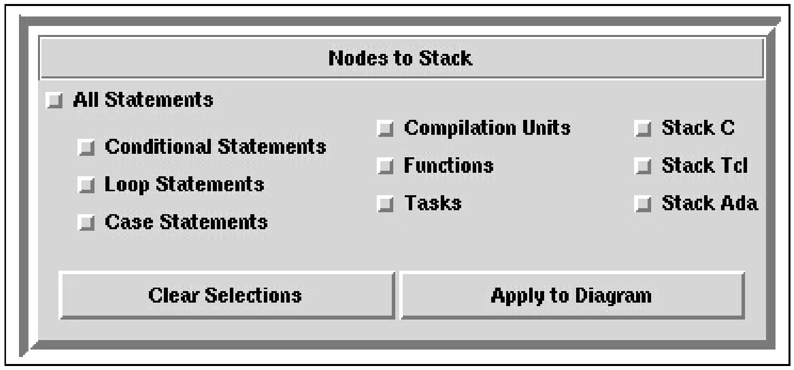

Figure IV.1: The desired "customizable stacking" options, similar to the elision window.

The single-language Star Diagrams stack nodes that are considered *similar*, but extending the *similarity* notion to multiple-languages is difficult because it can be interpreted in numerous ways. For example, consider a C *struct* and an Ada *package*; defining what makes them similar is difficult. One programmer might feel they are similar if the data structures have members with the same name; another programmer might feel they are similar if the structures are the same size. And enumerated types from two different languages might not be syntactically considered the same construct.

We considered providing the StarTool user the option to customize their own stacking, similar to the method used to elide nodes from the view. Our proposed interface can be seen in Figure IV.1. The programmer could selectively choose which kinds of nodes are stacked and which aren't, providing more control over the interface. For example, the user could stack all loop statements and all case statements; they could also stack all of the nodes within a single language, such as every Tcl/Tk node. The code that decides stackability of Star Nodes is shown in Figure IV.2. The algorithm used for determining whether to stack two nodes checks a context label for the identifiers to see if they are the same. To provide the capability to customize stacking, we would have had to modify the Adaptation Layer to provide a path to pass stacking information through.

```
static int
stackable(const SyntaxUnit first,
          const SyntaxUnit first_context,
          const SyntaxUnit second,
          const SyntaxUnit second_context)
{

        return (context_label(first, first_context) ==
                context_label(second, second_context));
}
```

Figure IV.2: The function that decides whether nodes are stackable for the display.

IV.D Reliance on 3rd-party tools

The interface used to implement the Ada adaptation layer proved problematic for future use of the tool. The Gnat adaptation layer implementation is actually *based upon* Gnat's Ada AST definitions. When the Gnat StarTool was originally created, the code used version 3.10 of the Gnat source code. During the course of our project, we desired to migrate to Gnat version 3.12 since the new version included Windows DLL capabilities. Unfortunately, the Ada adaptation layer used some identifiers from Gnat's version 3.10 source code that do not exist in Gnat version 3.12. Migrating to the new Gnat version would have required modifying the Ada adaptation layer, so we decided to use the older version of Gnat for the implementation of the multi-language tool. This exemplifies one of the problems associated with directly using a 3rd-party implementation. Had an interface been written on-top of the Gnat AST representation, we might have been able to more easily switch to later versions of the Gnat tool. An interesting task would be to verify that Hayes's Adaptation Layer can still be used with the new Gnat Source Code.

IV.E Performance

The StarTool performance overhead that is incurred by our multi-language extensions occurs during three phases: 1) When a source file is processed, 2) When a Star Diagram is built and the tool does language lookups on SyntaxUnits, and 3) When a multi-language StarTool is being built and *dummy* identifiers are created for cross-language searching. To benchmark the multi-language StarTool performance overhead, we calculated the amount of time required for these operations using the single-language tools and the amount of time to do the same operations in the multi-language tool. Our testing platform was a 200 MHz Sun UltraSparc 2 with 192 megabytes of RAM. The GNU g++ and Gnat Ada compilers were used by the StarTool for compiling the C and Ada sources. We loaded a set of 100 files from C, Tcl/Tk, and Ada sources; each test was run 5 times with the high and low results dropped and the other scores averaged.

1. *Loading source files into the StarTool.* The testing showed that the amount of time to load the source files into the multi-language tool required less than 4.7% more time than the total time of loading the C sources into Polaris, the Tcl/Tk sources into Twinkle, and Ada sources into Firefly. Since our run-time numbers do not include the amount of time to exit the individual tools and re-start the other tools, it is actually faster to use the multi-language tool to load source files from multiple languages.

2. *Building a Star Diagram without cross-language conversions.* To benchmark the slow-down for simply building a Star Diagram, we loaded a series of files from a single language into the multi-language StarTool to prevent it from doing SyntaxUnit conversions. We created Star Diagrams for identifiers with the *same name* as a chosen identifier and calculated the time from selecting *Display* on the main StarTool screen until the Star Diagram appeared on the screen. The multi-language tool required less than 11.5% more time to display the combined Star Diagram than the total time to use the single-

language tools individually. We again did not include the time to exit and re-start the tools. Depending on the size of the project, it may be faster to use the multi-language StarTool to load multi-language sources; in case of an extremely large program, the user might experience at most a 11.5% slowdown in loading sources. We feel that the slight performance decrease is reasonable considering the value of using the multi-language tool. Improvements in the Standard Template Libary map implementation or the substition of a different hash table interface are possible optimizations to improve this performance.

3. *Building a Star Diagram with cross-language conversions.* The last area where the multi-language StarTool affects performance is with cross-language conversion. We again loaded the same sources but this time loaded all of the sources from all of the languages at once. We built star diagrams including the *same name* and the *same identifier* and calculated the time to display the diagrams; the multi-language StarTool required at most 15.1% more time to calculate and display the Star Diagram than using each of the tools individually. We conclude that the extra 3.6% time slowdown is a good result, considering the addition of cross-language searching. Since program restructuring is time consuming, and the extra time to build a multi-language StarTool does not require user interaction, many programmers could conclude that the extra time is outweighted by the multi-language benefits.

Chapter V

Conclusion

V.A StarTool Programs

The StarTool at UCSD now has seven members: the original C-only StarTool, Elbereth for Java, Hayes's retargetable single-language implementations for C, Tcl/Tk, and Ada, and the multi-language C-Tcl/Tk and C-Tcl/Tk-Ada StarTools.

V.B Contributions of the Research

A method for combining retargetable single-language analysis tools into multiple-language analysis tools. We have developed a method for easily and quickly creating multi-language analysis tools from retargetable single-language tools using a multi-level adaptor approach with a mediator. With our new approach, a programmer that creates a StarTool for a new programming language will be able to add its functionality into the multi-language tool with minimal effort. A couple weeks work and less than 1000 lines of source code should suffice to add a new language from a single-language StarTool into our multi-language framework.

The method required to add a new language to the multi-language tool (after an adaptation layer has been created for the new language) is as follows:

1. Modify the hash table definitions to support the association of identifiers with the newly added language.

2. Customize the mediator's elision options to support constructs from the newly added language and the pre-existing supported languages. This might require adding a new elision category to the user interface, combining new categories with existing categories, or the renaming of categories to improve the usability of the interface.

3. Modify the 8 functions within the mediator that return SyntaxUnits to associate SyntaxUnits returned by the new adaptation layer with the newly supported language.

4. Modify the multi-language mediator to pass information to the new adaptation layer upon encountering a SyntaxUnit intended for the new language.

5. Add functionality to create a dummy node for the new language for cross-language searching. This will require implementing the *make_dummy*, *remove_dummy*, and *retrieve_name_scope_kind* function, widening the adaptation layer interface.

Multi-Language StarTool Implementations. We have developed versions of the StarTool for C-Tcl/Tk-Ada and C-Tcl/Tk.

Insights into multi-language program analysis. Through the use of our Multi-Language StarTool, we have discovered several issues with how programmers want to view information that comes from multiple source languages. Information can be displayed in a language-specific form or in a manner that generalizes across multiple languages. Tools capable of performing multi-language analsyis need to use a common interface with a mechanism to retrieve language-specific information hidden behind the interface. We have shown that a mediator combined with an adaptation layer is one effective solution.

V.C Future Work

We would like to test the multi-language interface on a large-scale commercial project. We are in the process of identifying a suitable candidate to help us with using the multi-language tool on a long-term basis. Raytheon is a likely candidate to assist us with using our multi-language extensions to restructure a large, multi-language software program.

The opportunity to provide customizable stacking to the user would be a great addition to the StarTool. More research needs to be done whether this would require modification of the adaptation layer or not. Even if it does, this would still be a worthwhile change. Since the adaptation layer wasn't originally intended to be multi-language ready, this conclusion wouldn't diminish the value of the adaptation layer and Hayes's retargetability approach.

We would like to add support for additional languages to the multi-language StarTool. The languages C, Tcl/Tk, Ada are all imperative programming languages. The similarity among the languages supported by the StarTool might have simplified our multi-language extensions, shadowing some language nuances we might have considered.

We plan to make our work available at the UCSD Software Evolution Laboratory web page, http://www-cse.ucsd.edu/users/wgg/swevolution.html. Binaries for both UNIX and Windows will be available for download.

Bibliography

[ANSI, 1997] Programming languages - C++ (1997). C++ Standard, ISO/IEC 14882:1998.

[Bowdidge, 1995] Bowdidge, R.W. (1995). *Supporting the Restructuring of Data Abstractions through Manipulation of a Program Visualization.* PhD thesis, University of California, San Diego, Department of Computer Science and Engineering. Technical Report CS95-457.

[Bowdidge and Griswold, 1994] Bowdidge, R.W. and Griswold, W.G. (1994). Automated support for encapsulating abstract data types. In *ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 97-9110.

[Cabaniss, 1997] Cabniss, J.L. (1997). *Lessons Learned from Applying HCI Techniques to the Redesign of a User Interface.* Masters Thesis, University of California, San Diego, Department of Computer Science and Engineering. Technical Report CS97-548.

[Chen, 1996] Chen, M.I. (1996) *A Tool for Planning the Restructuring of Data Abstractions in Large Systems.* Masters Thesis, University of California, San Diego, Department of Computer Science and Engineering. Technical Report CS96-472.

[Cormen, et al., 1997] Cormen, T.H., Leiserson, C.E., and Rivest, R.L. (1997) Introduction to Algorithms. The MIT Press, Cambridge, Massachusetts, 1997.

[Dewar, 1994] Dewar, R.B.K. (1994) The GNAT Model of Compilation. In *Proceedings of Tri-Ada '94*, pp. 58-70, November, 1994.

[Griswold et al., 1996] Griswold, W.G., Chen, M.I., Bowdidge, R.W., and Morgenthaler, J.D. (1996). Tool Support for Planning the Restructuring of Data Abstractions in Large Systems. *ACM SIGSOFT '96 Symposium on the Foundations of Software Engineering (FSE-4)*, San Francisco, October, 1996.

[Griswold and Atkinson, 1995] Griswold, W.G., and Atkinson, D.C. (1995). Managing the design trade-offs for a program understanding and transformation tool. *Journal of Systems and Software*, 30(1-2):99-116, July-August 1995.

[Hayes, 1998] Hayes, J.J. (1998). *A Method for Adapting a Program Analysis Tool to Multiple Source Languages.* Masters Thesis, University of California, San Diego, Department of Computer Science and Engineering. Technical Report CS98-600.

[Korman and Griswold, 1998] Korman, W., and Griswold, W.G. (1998) *Elbereth: Tool Support for Refactoring Java Programs.* Technical Report CS98-576, Department of Computer Science and Engineering, University of California, San Diego, April 1998.

[Linos, 1995] Linos, P.K. (1995) PolyCARE: A Tool for Understanding and Reengineering Multi-language Program Integrations. In *First IEEE International Conference on Engineering of Complex Computer Systems*, Nov. 6-10, 1995, pp. 338-341.

[Linos et al., 1993] Linos, P., Aubet, P., Dumas, L., Helleboid, Y., Lejeune, P., and Tulula, P. (1993) Facilitating the Comprehension of C Programs: An Experimental Study. In *Proceedings of the Second IEEE Workshop on Program Comprehension*, Capri, Italy, July 8-9, 1993, pp. 55-63.

[Morganthaler and Griswold, 1995] Morganthaler, J.D., and Griswold, W.G. (1995) Program Analysis for Practical Program Restructuring. In *Proceedings of the ICSE-17 Workshop on Program Transformation for Software Evolution*, Seattle, WA, pp. 75-80, April 1995.

[Zeigler, 1995] Zeigler, S.P. (1995) Comparing Development Costs of C and Ada. Online.