INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Supporting Library Interface Changes in Open System Software Evolution

by

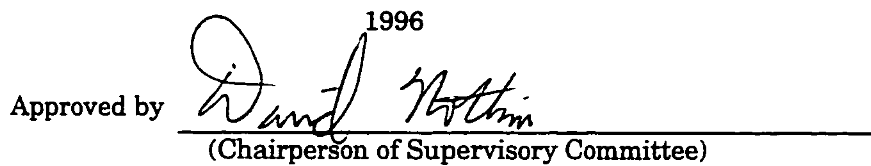
Kingsum Chow

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1996

Approved by 
(Chairperson of Supervisory Committee)

____

____

____

____

Program Authorized
to Offer Degree . Computer Science and Engineering

Date August 13, 1996

UMI Number: 9704478

Copyright 1996 by
Chow, Kingsum

All rights reserved.

UMI Microform 9704478
Copyright 1996, by UMI Company. All rights reserved.

This microform edition is protected against unauthorized
copying under Title 17, United States Code.

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

© Copyright 1996

Kingsum Chow

Doctoral Dissertation

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature ________

Date 8/9/96

University of Washington

Abstract

Supporting Library Interface Changes in Open System Software Evolution

by Kingsum Chow

Chairperson of the Supervisory Committee: Professor David Notkin
Department of Computer Science and Engineering

Software libraries provide leverage in large part because they are used by many applications. As Parnas, Lampson and others have noted, stable interfaces to libraries isolate the application from changes in the libraries. That is, as long as there is no change in a library's syntax or semantics, applications can use updated libraries simply by importing and linking the new version. However, libraries are indeed changed from time to time and the tedious work of adapting the application source to the library interface changes becomes a burden to multitudes of programmers. This dissertation introduces an approach and a tool set based on concrete syntax tree pattern matching intended to reduce these costs. Specifically, in our approach, a library maintainer annotates changed functions with rules that are used to generate tools that will update the applications that use the updated libraries. Thus, in exchange for a small added amount of work by the library maintainers, costs to each application maintainer can be reduced. We present the basic approach, describe the tools that support the approach, and discuss the strengths and limitations of the approach.

Table of Contents

List of Figures . . . . **vii**

List of Tables . . . . **ix**

Chapter 1: Introduction **1**

1.1 The problem 2

1.1.1 Difficulty in designing stable interface 3

1.1.2 The tension against changing interface 4

1.1.3 Closed systems and synchronous changes 5

1.1.4 Open systems and asynchronous changes 5

1.1.5 Open system software evolution 6

1.1.6 Current practices to handle interface changes 7

1.1.7 Problems with current practices 7

1.1.8 Integrity of application source 8

1.2 An interface change example 9

1.3 Cost of library interface change 12

1.4 Characteristics of acceptable solutions 12

1.5 Our approach 13

1.6 Chapter summary 15

1.7 Overview 16

Chapter 2: Patterns of Library Interface Changes **18**

2.1 Terminology 19

2.2 Case study: Borland C++ 20
2.2.1 General changes 21
2.2.2 Class library changes 24
2.3 Case study: libg++ 28
2.4 Case study: NIH class library 32
2.5 Case study: rdist 35
2.6 Case study: gzip 36
2.7 Case study: make 37
2.8 Case study: diff 39
2.9 Common patterns of library interface changes 44
2.9.1 Changing function prototypes 46
2.9.2 Changing function semantics 48
2.9.3 Rearranging functions 50
2.9.4 Changing data types 51
2.9.5 Changing header files 54
2.9.6 Changing global variables 55
2.9.7 Changing protocols 56
2.9.8 Changing syntax 57
2.9.9 Some language specific changes 57
2.10 Discussion 57
2.11 Chapter Summary 58
Chapter 3: The Change Process 59

3.1 Overview of the change process 60
3.1.1 First phase: change specification 60
3.1.2 Second phase: change adaptation 63
3.1.3 Program Representation 64
3.2 Requirements of the change process 65
3.2.1 The programming paradigm 65
3.2.2 Tools 66
3.3 Chapter summary 68
Chapter 4: The Change Specification Language 69
4.1 The criteria for a good change specification language 69
4.2 The design of the change specification language 70
4.2.1 Change specification bracketing keywords 72
4.2.2 Function change specifications 72
4.2.3 Data type change specifications 78
4.2.4 Global variable change specifications 79
4.2.5 Header file change specifications 79
4.3 Discussion 80
4.4 Chapter summary 81
Chapter 5: The Design of Our System 82
5.1 Overview of our design 84
5.2 Detailed design 87
5.2.1 File dependency extractor 88

5.2.2 Change specification extractor 88
5.2.3 Grammar modifier 89
5.2.4 Transformer modifier 90
5.2.5 Syntax tree constructor 91
5.2.6 Syntax tree transformer 91
5.2.7 Program representation details 91
5.2.8 Program source regenerator 93
5.2.9 Upgrade program 94
5.2.10 Type system 94
5.2.11 Symbol Tables 94
5.3 Further remarks on our design 95
5.3.1 Engineering decisions 95
5.3.2 On keeping the look and feel of the application code after upgrade 95
5.4 Chapter summary 97
Chapter 6: Validation and Evaluation **98**
6.1 Validation 98
6.1.1 Design of the validation suites 99
6.1.2 Validation suites based on examples 100
6.1.3 Validation using changes from diff-2.2 108
6.2 Status 114
6.3 Evaluation 119
6.3.1 Limitations and extensions 119

6.3.2 Other comments 122
6.4 Chapter summary 127

Chapter 7: Related Work **128**

7.1 Software maintenance 128
7.2 Program restructuring 129
7.3 Program representations 130
7.4 Interface changes 131
7.5 Change specifications 133
7.6 Matching code fragments 133
7.7 Program transformations for interface changes 135
7.8 Source reconstruction 136
7.9 Chapter summary 136

Chapter 8: Conclusion **137**

8.1 Contributions 137
8.2 Future work 138
8.3 Concluding remarks 139

Bibliography 140

Appendix A: Implementation Details **150**

A.1 A C-like grammar using PCCTS 150
A.2 A C-like syntax tree transformer using Sorcerer 170

A.3 Upgrade program 175

Appendix B: Purdue Compiler Construction Tool Set 187

B.1 ANTLR and DLG 187

B.2 Sorcerer 188

List of Figures

Figure 1-1: The original assign function 9
Figure 1-2: The new assign function 9
Figure 1-3: An illustration of the new function 10
Figure 1-4: The first scenario of the use of the assign function 10
Figure 1-5: The second scenario of the use of the assign function 11
Figure 1-6: The third scenario of the use of the assign function 11
Figure 1-7: Overview of our approach 14
Figure 1-8: Overview of the change process 15
Figure 2-1: Illustration of the change in Borland's MAKE - Makefile 22
Figure 3-1: Two phases of the change process 61
Figure 4-1: A change specification to swap two arguments (a and b) of a function call. . 70
Figure 4-2: The canonical form of a change specification. 72
Figure 4-3: A tree pattern in a pictorial form 75
Figure 4-4: The result of the simple swap specification. 76
Figure 4-5: A change specification to swap two arguments with type checking. 77
Figure 4-6: Adding a third argument to a function call.. 78
Figure 4-7: A data type change specification example. 79
Figure 4-8: A global variable change specification 80
Figure 4-9: A header file change specification. 80
Figure 5-1: Two phases of the change process. 83
Figure 5-2: A Makefile example showing the new update target. 84
Figure 5-3: The data flow diagram for our design 85
Figure 5-4: A Sorcerer syntax tree pattern example. 90
Figure 5-5: A generic function call subtree.. 92
Figure 5-6: A tagged function call subtree. 93
Figure 6-1: A Makefile template for our validation suites. 101
Figure 6-2: Before reordering arguments based on parameter types. 102
Figure 6-3: After reordering arguments based on parameter types 102

Figure 6-4: Before inserting an argument. 103
Figure 6-5: After inserting an argument. 103
Figure 6-6: Before applying data type change. 104
Figure 6-7: After applying data type change.. 105
Figure 6-8: Before applying a global variable change. 106
Figure 6-9: After applying a global variable change. 106
Figure 6-10: A specification for include directive changes. 107
Figure 6-11: Before include directive changes were applied. 108
Figure 6-12: After include directive changes were applied. 108
Figure 6-13: Interface change specifications in system.h. 109
Figure 6-14: Interface change specifications in diff.h.. 110
Figure 6-15: Before interface changes were applied to dir.C. 111
Figure 6-16: After interface changes were applied to dir.C. 111
Figure 6-17: Before interface changes were applied to sdiff.C.. 111
Figure 6-18: After interface changes were applied to sdiff.C. 112
Figure 6-19: Before interface changes were applied to diff3.C. 113
Figure 6-20: After interface changes were applied to diff3.C.. 113
Figure 6-21: Before changes were applied to util.C. 114
Figure 6-22: After changes were applied to util.C. 114
Figure 6-23: Our tools maintained the integrity of sdiff.C. 115

List of Tables

Table 2-1: The list of case studies 19
Table 2-2: Renamed functions in Borland C++ 24
Table 2-3: Renamed global variables in Borland C++ 24
Table 2-4: Renamed functions in Borland C++ class library 25
Table 2-5: Functions added to Borland C++ class library 25
Table 2-6: Function prototype changes in Borland C++ class library 26
Table 2-7: Functions with semantic changes in Borland C++ class library 28
Table 2-8: Functions with parameter type changes in libg++ 29
Table 2-9: Functions with default parameters changed to non-default in libg++ 29
Table 2-10: Functions added to libg++ 30
Table 2-11: Functions made public in libg++ 30
Table 2-12: Functions with changed return types in libg++ 31
Table 2-13: Functions with changed return values in libg++ 31
Table 2-14: Renamed types in libg++ 31
Table 2-15: New header files in libg++ 32
Table 2-16: Splitting header files in libg++ 32
Table 2-17: Eliminated macros in NIHCL 33
Table 2-18: Renamed types in NIHCL 34
Table 2-19: Changed types in NIHCL 34
Table 2-20: Removed variables in rdist 35
Table 2-21: Renaming header files in rdist 35
Table 2-22: New header files in rdist 36
Table 2-23: Moving variables in rdist 36
Table 2-24: Adding global variables in gzip 36
Table 2-25: A data type with one member type change in make 37
Table 2-26: A data type with one member removed in make 37
Table 2-27: A data type with a member name change in make 38
Table 2-28: Data types with new members in make 38
Table 2-29: Global variables added in rdist 38

Table 2-30: Global variables removed in rdist 39
Table 2-31: Functions added in rdist 39
Table 2-32: Functions renamed in diff 39
Table 2-33: Changing const function parameters to non const in diff 40
Table 2-34: Functions and macros introduced in diff 41
Table 2-35: Functions or macros removed from diff. 41
Table 2-36: Header files removed from diff. 41
Table 2-37: Data types renamed in diff 42
Table 2-38: New data types in diff 42
Table 2-39: Renaming global variables in diff 42
Table 2-40: New variables introduced in diff 43
Table 2-41: Global variables removed in diff 43
Table 2-42: Global variables with minor type changes in diff 43
Table 2-43: Replaced global variables in diff 44
Table 2-44: Patterns of interface changes 45
Table 4-1: bracketing keywords for change specifications 73
Table 4-2: Keywords for function change specifications 73
Table 4-3: Keywords for data type change specifications 78
Table 4-4: Keywords for global variable change specifications 79
Table 4-5: Keywords for header file change specifications 80
Table 6-1: Status of our implementation 116

Acknowledgments

Many people helped me to reach this point. I like to thank my wonderful wife, Hoon Goh, who has always believed in me in my study and research and delivered our first daughter, Ida, during our tough months. I also like to thank my advisor, David Notkin, to allow me to pursue freely into a research area of my interest and to have confidence on finishing my work. I thank all my reading committee members, David Notkin, Alan Shaw and Nancy Leveson, for their careful reading of my dissertation. I also like to thank other members of my supervisory committee, Craig Chambers and Ihsin Tsaiyun Phillips.

I thank Michael VanHilst for teaching me how to write better English, and his wife, Angella, for allowing him to help me even a couple of weeks before she delivered their first son, Marco. I thank Gail Murphy for polishing some of my research ideas and always coming up with suggestions to further improve my presentations. I thank Erica Lan for convincing me to keep trying to use the Purdue Compiler Construction Tool Set (PCCTS), which simplifies some of the design of my approach. I thank Yoshi Yamane for sharing his invaluable experience in thesis writing with me. I also thank Kristan Giessel for proof reading my dissertation.

I thank Jim Quinlan, Scott Robinson and Mike Tripp, who believe more in me than myself. I was convinced that interface changes were a big problem during one summer when I worked with them. I also thank Terence Parr and other developers of PCCTS, who answered many of my stupid questions. I thank Andy Glew, Gil Nieger, John Wilkes and Ted Biggerstaff for many useful technical discussions. I thank Chi Keung Tam and Mark Huss for sharing their personal experience with library upgrade problems. I also thank Wen-Hann Wang, Yi-Jing Lin, Derrick Weathersby, Ton A. Ngo and Shun-tak Leung for being there when I needed support.

Dedication

To my wife, Hoon and my daughter, Ida.

To my mother and father.

Chapter 1

Introduction

> "[Design] intermodule interfaces that are insensitive to the anticipated changes." - David Parnas

Libraries provide leverage in large part because they are used by many applications. As Parnas [Parnas 1972], Lampson [Lampson 1984], and others have noted, stable interfaces to libraries isolate the application from changes in the libraries. That is, as long as there is no change in a library interface's syntax or semantics, applications can use updated libraries simply by importing and linking the new version.

This is a great idea, but not completely practical. Pancake, citing Ungar, recently summarized the practical problems of this approach:

> Ungar cautioned that "the notion of interface is much more of an illusion than people give it credit for. The problem is that nobody really knows how to formally write down the definition of an interface in a way that can be checked and will guarantee the thing will really work when used by different clients... You try to get all your design decisions to hide behind interfaces so that changes percolate through as few levels as possible. But every honest programmer will admit there are times when you have to change your mind, and the interface changes." The problem of how to manage such changes has not been solved, particularly if components are in use across a number of organizations or if the components have been modified or built upon in any significant way [Pancake 1995, p. 35].

How to carefully write down the definition of an interface is clearly a problem, but the bigger problem, as Pancake pointed out, is how to manage interface changes, particularly if components are in use across a number of organizations.

One of the main goals of this thesis is to make progress towards solving this particular problem. In addition, it also provides a practical programming model and an effective solution to at least one central aspect of this problem. In particular, this thesis provides a solution for both the component providers and component users with respect to the many tedious adaptations to interface changes of a reusable component after its initial release. This thesis addresses an approach to reduce some of the costs of updating applications when a library, upon which the applications depend, changes in unpredicted ways, especially when the syntax and/or the semantics of the library change.

1.1 The problem

To decrease costs and increase reliability, modern software products are often created or assembled by reusing software components. A software development team often uses some components that are provided by other teams within the same organization or from other organizations, e.g. commercial software libraries. If the semantics of a reused component are changed, other components that depend on the stable semantics of this component may be affected. Although it is not desirable, it may be necessary to change the semantics of a reused component due to unanticipated changes. In such case, the problem is to identify the numerous other software components or applications both within the same organization or outside that use this changed component [Moriconi & Winkler 1990], and to propagate appropriate changes to those other components or applications to make them compatible with the new version of the software component. In other words, the problem is both **when** and **how** to propagate these changes to update other software components that are not even visible to the maintainer of that reused component. We call this process **asynchronous software evolution** [Chow & Notkin 1996a], due to the asyn-

chronous nature of the times when changes are applied to a reused component and then separately to the numerous other components that use it.

The problem of asynchronous software evolution is becoming ever more important due to the huge growth in the number of vendor libraries that support an even larger growth in the number of applications. Although it is hard to estimate precisely, it is easy to believe that almost every application imports from a number of libraries. Some of these libraries may even reuse routines from other libraries. This problem manifests itself in two dimensions: the length of the chain of the use relations among the libraries and the multiplicity of the library users, which can be applications or other libraries.

1.1.1 Difficulty in designing stable interface

If a library interface is stable, i.e., the changes in the future releases of the library do not need the application maintainers to change their code, then we have an ideal library interface that satisfies Parnas' criteria of "designing intermodule interfaces that are insensitive to the anticipated changes" [Parnas 1977, p.313]. But in practice, the future is hardly predictable and having a stable interface that can handle future changes is hard, if not impossible. Anticipating the need to change is still a good idea, of course. One reason is that syntax – and semantics – maintaining changes are still easiest to handle. Another reason is that, as we will see in Chapter 2, some interface changes are easier to handle than others, and we may get simpler interface changes through better anticipating changes. Nevertheless interface changes are sometimes simply inevitable.

No one should write or need to write all the code from scratch. Most of us would agree that using well debugged and well maintained libraries yields a significant saving in terms of software development cost. However, some of the libraries that we use may not have been perfectly debugged or had their interface adequately polished. For reasons of business expediency, such libraries are released. And again, for the same reasons, we use those

libraries. The release and adaptation of program libraries may occur within a software development group, across several groups within an organization, or even across different organizations. The web of library use relationships becomes more complicated when versions of library releases are also involved. The users of a program library may also be the developers of another program library.

The maintainers of a library may decide to change its interface for a reason. An example of such a requirement is the standardization of the syntax of some related interfaces across the same library or, more often, to make them look more like another library which has an interface becoming popular in the programming world. Other examples include providing extensions to the current services and providing semantic compatibility with new devices or other libraries.

1.1.2 The tension against changing interface

Library maintainers may be reluctant to change the interface of a library even with valid reasons for doing so. They fear that users of the current version of the library may reject the new version because of the additional requirement for them to change their code. Some of the library users may not be able to change their code properly. Others may forget to change something that needs to be changed. The rest may find it too annoying, distracting or time consuming to deal with something they simply hate to do.

Our premise is that library maintainers may be less reluctant to change the interface if they can somehow help the library users adapt their code to the new interface. When changing an interface, a library maintainer probably has a good idea of the effect it will have on the users of that interface. In fact, as we will see later, the library maintainer may know what changes are required in the users' code because, since they are the people who designed and implemented the interface, that they should know a lot about it.

In summary, when a library maintainer revises a library, there is a tension between improving the interface by changing it and leaving the interface the same as before. The library maintainer may decide against changing the interface if the cost to the application maintainers in terms of effort to upgrade outweighs the benefits of the improvement. Thus, reducing the cost to upgrade the applications would give library maintainers more leverage in improving a library interface.

1.1.3 Closed systems and synchronous changes

Many researchers [Griswold & Notkin 1993] [Johnson & Opdyke 1993] [Casais 1992] [Bergstein 1991] [Lieberherr & Xiao 1993] have been working on the restructuring of programs. Their work focuses on the meaning preserving transformations of program structures within a program. They assume that all source code is accessible and modifiable. We call such a system a closed system because the use of a program structure is closed within an entity. When an application programmer and a library maintainer are the same person, and the library is only used by the application, then we have the simplest case of a closed system. When all source code is available, as in a closed system, the effect of a structural change to one part of a program on the rest of the program can be determined at the time of the introduction of the structural change. Since the adaptation of changes can occur right after the introduction of changes, we call these synchronous changes.

1.1.4 Open systems and asynchronous changes

Library maintainers do not enjoy the benefit of a closed system because one of the purposes of a library is to allow it to be reused by many others. Under these circumstances, users' code is not generally available. When some source code is neither accessible nor modifiable at the introduction of a structural change to a program, we call such a system an open system because the use of some program structures is open to many others outside the closed system. Even in an open system, if library maintainers can access and change

the users' code when necessary, then application maintainers who trust the library maintainers, can then do less work.

In most cases, however, library maintainers do not know who their users are, much less be able to access and help them adapt their code. The library maintainers should still be able to specify the required adaptation changes in some way and let those changes propagate into the library users' code, at a different time and place from the library maintainers. Ideally, the integration of library interface changes and the application code changes in response to such interface changes should require little intervention from an application maintainer. However, the application changes cannot occur right after the introduction of library changes here. We call these asynchronous changes.

While synchronous changes are possible in closed systems, asynchronous changes are required in open systems for the reasons discussed earlier. Asynchronous changes can be used with closed systems as well. Thus asynchronous changes provide a more general approach to maintenance of software systems.

1.1.5 Open system software evolution

The problem of how to manage library interface changes across organizations continues to grow. There are more and more software libraries as well as software applications. When a library interface is changed, the application maintainers that depend on this library may need to be change their code if they want to upgrade their code to the new version. One may wonder why they would upgrade and deal with the code changes. The usual reason is that while the new library release requires the application to adapt in some inconvenient ways, it also brings many new features to the application that may be valued more than the cost of upgrading the application code. It is also easier for future updates if an application is regularly updated and maintained with each library release.

1.1.6 Current practices to handle interface changes

A library maintainer is only responsible for the library's code while application maintainers are responsible for the application code, including adapting the code to use a new library release. Nevertheless, when a library interface is changed, the library users need to be notified of those changes. The changes and suggested way to adapt code to those changes are usually written in a human language as part of a printed document, the README file associated with the new library or e-mail announcements. While there are also other methods (e.g. printed documentation, list of frequently asked questions) of library change notification, by far, the most widely used method is to include a README file in a software distribution.

README files are electronic documents that are distributed along with a software distribution. They are often used to provide last minute information and usually changes to the documentation. Because it is much easier to add a README file to a software release than to reprint documents, it is also used by some software library providers to specify changes, including interface changes, to a release.

1.1.7 Problems with current practices

There are many weaknesses of the README file approach. The first problem is that it is not clear whether this kind of information will actually reach clients, e.g., the intended audience may not be aware of the existence of the README file. Furthermore, it is not clear that clients will be sufficiently motivated to read the README file. Even if they do read the README file, the updating activity may still be difficult to understand and to carry out. The README file may not provide complete or even sufficient information for all maintenance issues; there is no standard method or programming tool to verify that the README file is consistent and up to date.

The distribution of responsibility in the programming paradigm is also at fault. When changing the library interface, the maintainer has information about potential effects of each change that is important for the application maintainer to know. The library maintainer may attempt to write the pertinent information down informally (e.g. README files) at the end of the project, prior to release. But this is far from reliable. The library maintainer should be required to think more carefully about the interface change since it may affect many applications and a formal or at least semi-formal mechanism should be available for the library maintainer to pass important change information to the application maintainers.

1.1.8 Integrity of application source

Software structure tends to degrade over time [Belady & Lehman 1985] due to changes and fixes that are made to it over its lifecycle. Of special concern here is that the integrity that is the look and feel of application source code may suffer due to changes that are made to it to adapt to library releases. Application maintainers are rightfully concerned with the effect of changes on the integrity of the application source code. If the change adaptation is done by application maintainers, they can do what they see fit and they can also apply restructuring techniques [Griswold & Notkin 1993] [Johnson & Opdyke 1993] to restructure the program after making changes. If the change adaptation is done by another program, then application maintainers may be concerned with a particular kind of integrity of the application source - the look and feel of the code. It may be important to keep the spacing, indentation and commenting of the program as much as possible to maintain the look and feel and thus the integrity of the application code. This is a valid concern because it is possible to have an upgrade solution that adapts the code and thus making it compile and run without maintaining its original programming style.

The Law of Least Astonishment [James, 4.1] dictates that people do not like unpleasant surprises in their code, although style is not critical in making the code compile and run.

1.2 An interface change example

To clarify why handling changes to libraries is hard for applications, we'll use a simple example taken from a commercial library [Borland 1993a].[1] The example illustrates an interface change involving overloaded functions and default arguments.

The example addresses the assign function, which is used to copy values from one string to another. The original library function is given in Figure 1-1.

```
assign(char*, char*, int = NPOS );
```

Figure 1-1: The original assign function

The first parameter is the target string, the second is the source string, and the third parameter gives the number of characters to copy. The default of NPOS for the third parameter means that the full source string will be copied[2]. Later, the library maintainers decided to change the function to insert a new third argument as given in Figure 1-2.

```
assign(char*, char*, int = 0, int = NPOS );
```

Figure 1-2: The new assign function

In the new version, the first two parameters remain unchanged. However, the old third parameter (the number of characters to copy) becomes the fourth parameter, and a new third parameter is introduced. This new parameter indicates the position in the source string from which copying should start.

1. The example is modified slightly to use the C language syntax to simplify the presentation.
2. Default arguments are supplied if not explicitly specified in a function call [Stroustrup 1991].

An illustration of the use of the new assign version of the library is given in Figure 1-3.

```
char* s1 = "abcdef";
char s2[7];
...
assign(s2, s1, 2, 3 );
```

Figure 1-3: An illustration of the new function

After executing this code, s2 should contain "cde". (In C strings are arrays of characters and indexing starts with zero.)

We are going to look at a few simple scenarios of what can confront an application maintainer whose application is already using the old library version of the assign function.

In the first scenario (Figure 1-4), throughout the application all calls to the original assign function use only the first two parameters.

```
char* s1 = ...
char* s2 = ...
...
assign(s2, s1);
```

Figure 1-4: The first scenario of the use of the assign function

In this case, the code in the application does not need to be changed in response to the changes to the library.

In the second scenario (Figure 1-5), the application contains calls to the assign function that use three parameters. If the application maintainer imports the new library without making any changes to the application, the code will still compile:.

```
char* s1 = ...
char* s2 = ...
int i = ...
...
assign(s2, s1, i);
```

Figure 1-5: The second scenario of the use of the assign function

However, the interpretation of the variable i has changed from the number of characters to copy to the starting position to copy. The behavior of this piece of code is changed due to a change in the interface of the library's assign function. This new behavior is unlikely to be what the application maintainer wants.

In the third scenario (Figure 1-6), the application has the same kinds of calls to assign as in the second scenario, but the application maintainer is aware of the library change. In this case, the maintainer inserts a third argument of 0 in each call. This will make the application code behave the same as before.

```
char* s1 = ...
char* s2 = ...
int i = ...
...
assign(s2, s1, 0, i);
```

Figure 1-6: The third scenario of the use of the assign function

These scenarios identify several problems. First, if the application maintainer is unaware of the changes to the library (perhaps because of not reading the accompanying documentation carefully), their application will in many cases stop working properly. Second, every application maintainer has to decide whether the changes will affect the application and, if so, update the potentially large number of call sites by hand. This example only considers one function in what might be a large library. Increases in the number of functions in the library that change may significantly increase the number of call sites that the

application maintainer has to update. Third, many applications will have some call sites that require changes and others that do not; handling selective changes like this may increase the difficulty of the application maintainers' job in handling library updates.

This simple example is characteristic of many kinds of changes that take place when libraries are modified.

1.3 Cost of library interface change

The cost of a library interface change is the combined effort of introducing the interface change to a library release and of upgrading all applications that may be affected due to this change. This is a generalization from the example shown earlier. While some application code may not need any changes, it still requires effort to verify that is indeed so.

1.4 Characteristics of acceptable solutions

In addition to the high total cost in effort, there is the high risk of errors in the standard asynchronous upgrade approach, both in assessing whether to make changes and also in making those changes manually. Thus, effective solutions to this problem should have three characteristics.

1. They should relieve the application maintainers, as much as possible, from having to identify whether or not the changes to the library require changes to the application.
2. They should relieve the application maintainers, as much as possible, from having to identify and manually update each of the locations in the application that must change in response to library changes. Reducing the number of locations that must

be updated manually will necessarily reduce the number of additional errors introduced during maintenance [Collofello & Buck 1987].

3. They should maintain the integrity (look and feel) of the application source around the changes need for the library upgrade [James, 4.1].

The next section sketches the basic approach we take, which is intended to satisfy these properties.

1.5 Our approach

There are many application maintainers since there are many applications – and (logically) only one library maintainer – since there is only one library. We require the library maintainer to annotate any changes that are made to indicate what updates must be made to the applications to accommodate the updated library. We then provide a set of tools that uses these annotations to semi-automatically update any selected applications that use the updated library. The idea is that the small amount of additional work done by the one library maintainer can provide leverage for the many application maintainers. This is illustrated in Figure 1-7.

In contrast to synchronous software evolution, our model of asynchronous software evolution handles the specification of changes, the propagation of changes and the necessary program transformations asynchronously, across time (between when a library is updated and when each application is updated) and space (between where they are changed).

Our model requires the library maintainers to specify interface changes and also how existing users' code can be transformed to adapt to those changes. The specification of the library interface changes, as well as the specification for the transformations, is then extracted by a tool on the application side. Another tool on the application side constructs

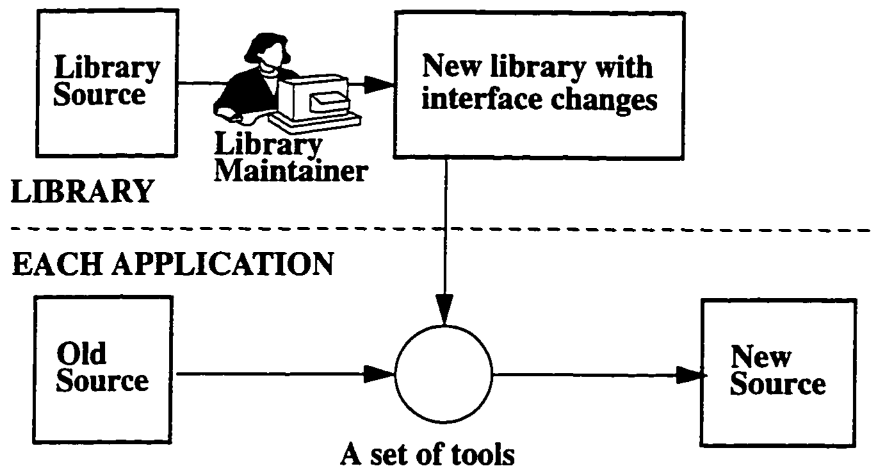


Figure 1-7: Overview of our approach

an internal representation of the library client program, checks for dependencies on the interface changes, and performs code transformation as stated in the transformation specification. An updated application, which has been adapted to the new library, is produced at the end.

The refinement of the set of tools is illustrated in Figure 1-8. A library maintainer revises a library with interface changes. She records the changes in the interface as well as the transformation required for the users' code to adapt to the interface changes. When a client application sees these interface changes, its original source is extracted by a tool that constructs an internal representation of the program, a syntax tree (ST) in our implementation. The internal representation is then modified by a change propagation tool into another internal representation, from which a new source code is generated.

In our research, we use a model of asynchronous software evolution [Chow & Notkin 1996b] that can be applied to common programming practices such as using makefiles and programming in ANSI C. By using a model like this, our research in software engineering

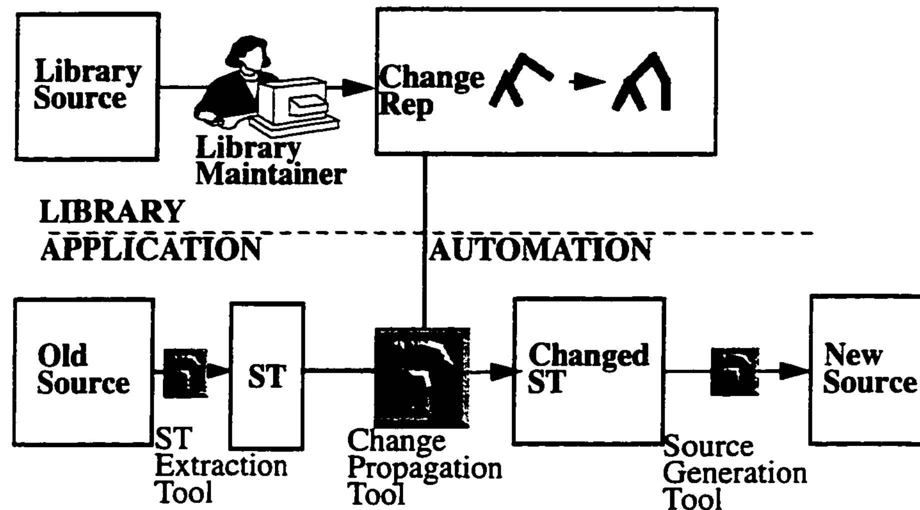


Figure 1-8: Overview of the change process

can be easily realized. In fact, we will show that our approach can handle a significant number of library interface changes semi-automatically for C programs. We focus on library interface changes as the main problem of asynchronous software evolution because the interface is the link between software components in many imperative programming languages, e.g. a package specification in Ada or a header file in C. Although our approach – specifically, the tools – are not industrial strength, the prototype acts as a proof of concept for the approach.

1.6 Chapter summary

We regard adapting source code to library interface changes as one activity in software evolution [Chow 1995]. In this chapter, we discussed the myth of stable library interfaces. Some reasons for the inadequate support to handle library interface changes were suggested. Both research and industrial work that may be of use to handle this problem were described. A model of handling asynchronous software evolution was proposed and was shown to work with a small example.

The principle of the asynchronous software evolution model is to enable the library maintainers to propagate necessary interface changes to the users' code. A small amount of extra work for the library maintainer will reduce the amount of adaptation work for a large number of application maintainers.

In particular, in our approach, the library maintainer needs to pay the one time cost of providing the grammar and the transformation specification. Then, for each interface change, the change specification must be provided. In this way, the major cost of upgrading source code for library interface changes is shifted from the many application maintainers to the library maintainer. This is the main value of our approach and implementation, making the thousands of application maintainers save costs and reduce errors when the library interface is indeed changed. Another benefit of the approach is that it encourages library maintainers to think more deeply through the ramifications before making interface changes.

As we will see later, our implementation assumes the existence of interface files that contain the interface specification. This realistic assumption allows us to build a prototype tool set to demonstrate our idea nicely and practically. Also, by using comments, rather than new language features, to introduce library change specifications, we can more easily adapt our approach elsewhere.

In this thesis, we tackle the real life problem of unstable library interfaces directly. We identify the requirements to reduce the cost of updating source code that is affected by interface changes. We propose the model of our approach and give a reasonable implementation that works with real life examples.

1.7 Overview

The rest of this dissertation is organized as follows: Chapter 2 describes the library interface change patterns; Chapter 3 provides a more detailed overview of our approach; Chap-

ter 4 describes our change specification language; Chapter 5 describes our design and implementation; Chapter 6 evaluates our approach; Chapter 7 discusses related work; and Chapter 8 summarizes this thesis with concluding remarks.

Chapter 2

Patterns of Library Interface Changes

Our approach to supporting interface changes in open system software evolution is based on 7 case studies (Sections 2.2 - 2.8) of interface changes and the compilation of the results into a common pattern list (Section 2.9). In these studies, we identified a number of common changes. So, although the various kinds of interface changes are theoretically unbounded, there are actually common patterns of interface changes that show up again and again. The objective of our approach is to provide mechanisms so that the majority of the change patterns can be handled efficiently, both in terms of their specification and propagation. This chapter describes the common patterns of interface changes and the case study examples from which they are derived. We also discuss briefly the effects of those interface changes on application source. The case study examples were chosen from systems where there was interface change information as well as source code. The range of systems that are covered represents systems in various stages of development. Although the examples (Table 2-1) are taken from C and C++ programming languages, they can be generalized to other imperative programming languages that explicitly represent interface information. We do not imply that the patterns of the interface changes will be the same for other languages however. Language features certainly play an important role in what interface changes are feasible. But, as we will see shortly, most of the changes we look at are possible with other imperative languages, too.

We studied a wide variety of interface changes here. Our selection criteria are the availability of interface changes and the availability of source code. The interface changes in our studies are available in change log files (most of which are taken from a set of GNU programs), README files and release documents. Sometimes, we determined the interface changes by getting two versions of a program and finding the difference manually.

Table 2-1: The list of case studies

System	Description
Borland C++	Borland C++ system and class library
libg++	The GNU C++ class library
nihcl	National Institute of Health Class Library
rdist	Remote file distribution program
gzip	Compress or expand files
make	Maintain, update, and regenerate groups of programs
diff	Differential file comparator

One should note that some of the case studies (e.g. rdist, gzip, make and diff) are not taken from stand alone libraries but rather stand alone applications. However, the interface changes discovered from them still give invaluable insights to the problem of interface changes.

In this chapter, we first present the case studies of interface changes and then we present the summary of the interface changes and their consequences. As we will see shortly, from our case studies, common interface changes are dominated by function changes, data type changes, global variable changes and include directive changes.

2.1 Terminology

To facilitate the discussion of the patterns of library interface changes, we define the following terms that are used in the rest of this chapter:

- Interface - An interface includes both the syntactic constructs and the operational semantics upon which the client depends but is informally defined.[1]

1. We ignore other issues of interface such as timing.

- Header file - An include file contains interface information that is exported to clients. This terminology is not restricted to C programs. For example, it could mean the specifications files in Ada [USDoD 1980]. We use header files or interface files interchangeably in this thesis.

- Source file - A source file is a synonym for ".c" (C), ".cc" (C++), ".cpp" (C++), ".imp" (Ada) or ".pas" (Pascal) files.

- Function - A function is a generalized term for functions, procedures, subprograms and subroutines.

2.2 Case study: Borland C++

We first studied the interface changes in Borland C++ version 4.0. The interface change information was taken mostly from the README file [Borland 1993a] and chapter 1 of the library reference [Borland 1993c]. Some of the changes and how to handle them can also be found in the programmer's guide [Borland 1993b].

We picked this study because Borland C++ was, and is still, a very popular development environment among PC programmers. Furthermore, Borland has a history of delivering excellent products, such as Turbo Pascal. Despite the strong technical abilities of the Borland developers, sometimes they do need to make interface changes for many reasons.

In this study, the relevant documents [Borland 1993a] [Borland 1993b] [Borland 1993c] were read and the interface changes were determined from the documents. The documents actually described the interface changes very well but to get some hands-on experience, we also experimented with both Borland C++ versions 4.0 and 3.1.

We hoped to get interface changes from the C++ library only but we got more than we hoped for. An example of a language change in their "Make" program was also observed. This clearly shows that changes can come in many unexpected places!

We focused on the changes in common language constructs in our studies. Thus, we focused on changes that are not tied to special features of C++, e.g., class inheritance. Nevertheless, we will discuss the impact of class inheritance changes on the general problem of responding to interface changes.

There are many reasons why the interfaces were changed. One was the C++ language was being standardized. Another one was the need to extend some function prototypes to handle newer requirements. Yet another one was to standardize some function arguments. Perhaps one more reason was to make some function prototypes more compatible with other libraries. Most of the times we could only speculate the reasons for the changes because the real reasons were not documented.

In general, there are two kinds of changes in this release. The first kind, which is more general, applies to whoever uses the new version. The second kind applies to only those who use the class library. We will present them in order.

2.2.1 General changes

(1) A change in Borland's MAKE syntax

The first change we noticed from this study is not an interface change in the C++ language or the library but a change in the Borland's MAKE language [Borland 1993a]. Unlike the pervious versions (i.e., version 3.1 or earlier) of MAKE, the new MAKE will not generate some space between two directly adjacent macros, i.e., two macros without any space in between them.

```
Makefile:

SRC1 = main.cpp
SRC2 = support.cpp

ALLSRC = $(SRC1)$(SRC2)
...
```

Figure 2-1: Illustration of the change in Borland's MAKE - Makefile

An example Makefile is shown in Figure 2-1. When ALLSRC is expanded it would become

```
main.cppsupport.cpp
```

instead of

```
main.cpp support.cpp
```

This change in MAKE can be considered as one kind of a library interface change because it affects the application's MAKE programs directly in a way that it requires corresponding responses in the application's MAKE programs to adapt to this change.

(2) Special protocol requirement

Sometimes, a certain protocol of function calls is required even though other functions are available. For example, when creating secondary threads in version 4.0, the _beginthread() and _endthread() functions must be used to ensure the proper initialization and cleanup. Although it has not been fully elaborated in the documentation, it appears that the function calls CreateThread() and ExitThread() from version 3.1 will need to be replaced by those two functions respectively.

(3) Obsolete functions

Some functions were obsolete in the new version. For example, DOS3Call is obsolete under Win32 in version 4.0.

(4) Adding functions

Some functions were added to the new version. For example, there were two new functions that provide access to 32-bit operating system file handles:

```
_open_osfhandle()
_get_osfhandle()
```

(5) Removing functions

Some functions were removed from the old version. For example, the following function was removed:

```
Type_info::fname()
```

(6) Renaming functions

Some functions were renamed to comply with ANSI naming requirements [Borland 1993c, Chapter 1]. For example, 7 global functions were renamed in 2 header files (see Table 2-2).

(7) Renaming global variables

Some global variables were renamed to comply with ANSI naming requirements [Borland 1993c, Chapter 1]. For example, 7 global variables were renamed in 5 header files (see Table 2-3).

Table 2-2: Renamed functions in Borland C++

Old name	New name	Header file
_chmod	_rtl_chmod	io.h
_close	_rtl_close	io.h
_creat	_rtl_creat	io.h
_heapwalk	_rtl_heapwalk	malloc.h
_open	_rtl_open	io.h
_read	_rtl_read	io.h
_write	_rtl_write	io.h

Table 2-3: Renamed global variables in Borland C++

Old name	New name	Header file
daylight	_daylight	time.h
directvideo	_directvideo	conio.h
environ	_environ	stdlib.h
sys_errlist	_sys_errlist	errno.h
sys_nerr	_sys_nerr	errno.h
timezone	_timezone	time.h
tzname	_tzname	time.h

2.2.2 Class library changes

(8) Removing function parameters

Some function parameters are removed in this release. For example, the following member functions for direct containers (except dictionaries) no longer take a delete parameter.

```
Flush
Delete
```

Thus, old function calls of these functions using an argument will not compile but syntax errors will be noted by the compiler.

(9) Renaming functions

Some functions were renamed but, in some cases, the reasons were not given. For example, bag and set container member function FindMember was renamed to Find but took the same parameters (see Table 2-4).

Table 2-4: Renamed functions in Borland C++ class library

Old name	New name	Class
FindMember	Find	bag container
FindMember	Find	set container

(10) Adding functions

Some functions were added. For example, containers now have a member function called DeleteElements with this prototype:

```
void DeleteElements()
```

Also, functions might be added to make many classes to look consistent in their interfaces. For example, all list and double-list containers were given the DetachAtHead member function.

Table 2-5: Functions added to Borland C++ class library

New function	Classes
DeleteElements	containers
DetachAtHead	list and double-list containers

(11) Changing function prototypes

Sometimes, the function prototype changes were explained. For example, the Detach and Flush member functions were changed to make them similar to other classes. Sometimes,

the reasons for the changes are obvious, for example, the assign member function of the string class was changed to include one more default argument.

The old prototype for this function was:

```
assign( const string&, size_t = NPOS );
```

It was changed to:

```
assign( const string&, size_t = 0, size_t = NPOS );
```

The size_t parameter in the old version was the number of characters to copy. In the new version that is the second size_t parameter; the first one is the position in the passed string to start copying.

The same change was made in 11 other functions (see Table 2-6).

Table 2-6: Function prototype changes in Borland C++ class library

Function
`string( const string &, size_t, size_t );`
`string( const char *, size_t, size_t );`
`string( const char *, size_t, size_t );`
`assign( const string &, size_t, size_t );`
`append( const string &, size_t, size_t );`
`append( const char *, size_t, size_t );`
`prepend( const string &, size_t, size_t );`
`prepend( const char *, size_t, size_t );`
`compare( const string &, size_t, size_t );`
`insert( size_t, const string &, size_t, size_t );`
`replace( size_t, size_t, const string &, size_t, size_t );`

(12) Protocol changes

There is a rather complicated change when a particular group of macros was used. The old use required the use of both "declare" and "define" macros in the same file, for example:

```
DIAG_DECLARE_GROUP( Sample );
DIAG_DEFINE_GROUP( Sample, 1, 0 );
```

There was also a "create" macro that replaced the two macros above, for example:

```
DIAG_CREATE_GROUP( Sample, 1, 0 );
```

The new version still had both the "declare" and "define" macros. But the "define" macro now included the function of the "declare" macro too, i.e., making it similar to the "create" macro. Furthermore, the "declare" macro could not be used together with a "define" macro. This has two consequences:

1. The "create" macro is removed.
2. The "declare" macro is valid in files where the "define" macro is not used.

Thus, code that uses both the "declare" and "define" macros needs to remove the "declare" macro. Code that uses the "create" macro should be changed to use the "define" macro.

(13) Semantic changes in functions

The meanings of these "find" functions (see Table 2-7) were all revised in a similar manner. The interpretation of the return value of this set of functions was changed. In the new version, a successful find returns the character location and a failed find returns NPOS. Since it is the interpretation of the integer values that changes, we classify this as a semantic change. The interpretation of the old values was not clear but most people would think it should return 0 for a failed search.

However, even with this newly revised set of find functions, there was some inconsistency. For example, from the set of "find_first_of", "find_first_not_of", "find_last_of" and "find_last_not_of" functions (two versions of each, one taking one argument and the other taking two arguments), all functions return NPOS when they fail except the "find_last_of" function that takes one argument. Most of us would strongly suspect that there was a misprint for that particular "find_last_of" function and indeed it should probably fail with a return value of NPOS. Here, we noticed not only the problem of semantic changes, but also the difficulty in checking the correctness or consistency of README files by the library maintainers, or the comments in the code, for that matter.

Table 2-7: Functions with semantic changes in Borland C++ class library

Function
`size_t find_first_of( const string _FAR &s ) const`
`size_t find_first_of( const string _FAR &s, size_t pos ) const`
`size_t find_first_not_of( const string _FAR &s) const`
`size_t find_first_not_of( const string _FAR &s, size_t pos ) const`
`/*!*/ size_t find_last_of( const string _FAR &s ) const`
`size_t find_last_of( const string _FAR &s, size_t pos ) const`
`size_t find_last_not_of( const string _FAR &s ) const`
`size_t find_last_not_of( const string _FAR &s, size_t pos ) const`

In Table 2-7, the function (marked "/*!*/") actually did return NPOS for a failed search in our experiment. Thus, the README document is incorrect.

2.3 Case study: libg++

A public domain library is a natural choice for the study of library interface changes. Here we studied the GNU C++ library (commonly known as libg++) [Lea 1988]. We extracted the changes from the change log file for libg++ version 2.6.2. There had been a lot of changes for the past revisions of this library. While we did not attempt to verify that the change log is complete, we believe nevertheless it contains a good sample of the interface

changes. We observed the following interface changes in the GNU C++ library. Most of the changes are annotated with descriptions to provide more concrete examples instead of the generalization.

(1) Changing parameter types

Some function parameter types were changed. An example is given in Table 2-8.

Table 2-8: Functions with parameter type changes in libg++

Function	New parameter type	Old parameter type
srand	unsigned	signed

(2) Changing default parameters to non-default

Some default parameters were changed to non-default. An example is given in Table 2-9. In this example, the fourth parameter, sk, had a default value but was changed to have no default value. The reason that was given is to prevent ambiguous matching.

Table 2-9: Functions with default parameters changed to non-default in libg++

Old function	New function	Header file
istream(int filedesc, char* buf, int buflen, int sk = SOME_CONST, ostream* t = 0)	istream(int filedesc, char* buf, int buflen, int sk, ostream* t = 0)	istream.h

(3) Adding functions

Some functions were added. Some of these newly added functions were overloaded versions of an existing one.

Table 2-10: Functions added to libg++

New function	Header file
ostream << (const void * p)	ostream.h
re_comp()	std.h
re_exec()	std.h
more (overloaded) versions of abs()	builtin.h
more (overloaded) versions of even() and odd()	builtin.h
check_state() in File class	stream.h
rewind()	std.h
bsearch()	std.h
char* chr(ch)	builtin.h
virtual destructors	many classes

(4) Making member functions public

Some member functions were made public. In fact, making functions public is like adding functions, from the view point of a user using the function. A few examples are shown in Table 2-11.

Table 2-11: Functions made public in libg++

Function	Header file
opterr (was private)	Getopt.h
(unspecified)	Poisson.h
(unspecified)	Lognormal.h

(5) Changing function return types

Some function return types were changed. Two examples are shown in Table 2-12.

Table 2-12: Functions with changed return types in libg++

Function	New return type	Header file
puts	int	stdio.h
qsort	void	std.h

(6) Changing meaning of function return values

The meaning of some function return values were changed. Table 2-13 shows two such examples.

Table 2-13: Functions with changed return values in libg++

Function	Header file	New meaning
low()	MPlex.hP	returns lowest valid index
contains(Regex)	String.h	unspecified

(7) Renaming a type

Some data types were renamed. Table 2-14 shows one such example.

Table 2-14: Renamed types in libg++

Old type	New type	Header file
libm_exception	struct exception	math.h

(8) Renaming header files

All header file names were shortened to make the system run on Unix System V.

(9) Adding header files

Some header files were added (see Table 2-15).

Table 2-15: New header files in libg++

New header file
new.h

(10) Spliting header files

A header file was split into two. Table 2-16 shows an example in which a new header file was used.

Table 2-16: Splitting header files in libg++

New header file	Originally in
Regex.h	String.h

(11) Moving type declarations from a header file to another

Some type declarations were moved from one file to another. For example, bool enum was moved to bool.h from some unspecified source.

2.4 Case study: NIH class library

The National Institute of Health Class Library [Gorlen et al 1990] is available from the public domain. It was one of the first freely available class libraries and had gone through a few revisions. Unfortunately, due to the absence of a change log file, it is harder to determine interface changes in this library. We inferred the interface changes from the README file and source code, when ambiguity arose.

We obtained versions 3.0 and 2.2 of this library. Following are the changes in the usual summary format.

(1) Library name change

The library name was changed from OOPS to its now popular name, NIHCL.

(2) Removing global variables

A number of global variables were moved to become static members of a new class. We did not present an example here.

(3) Removing macros

Some macros were removed. Two of them are given in Table 2-17.

Table 2-17: Eliminated macros in NIHCL

Eliminated macros
READ_OBJECT_AS_BINARY
STORE_OBJECT_AS_BINARY

(4) Adding functions

Some functions were added. An example is: dumpOn().

(5) Renaming types

Several types were renamed. They are summarized in Table 2-18.

Table 2-18: Renamed types in NIHCL

Old name	New name
Arrayobid	ArrayOb
Linkobid	LinkOb

(6) Changing types

Some types were changed. Table 2-19 shows one such example.

Table 2-19: Changed types in NIHCL

Type name	Old type	New type
bool	char	int

(7) Adding default parameters to functions

An extra parameter was added to one function. The extra parameter contains a default value. This function was:

```
String(char c, unsigned l=1);
```

It was later changed to:

```
String(char c, unsigned l=1, unsigned extra=DEFAULT_STRING_EXTRA);
```

(8) Adding functions

A function was added. For example,

```
void Process::select(FDSet& rdmask, FDSet& wrmask, FDSet& exmask);
```

2.5 Case study: rdist

Here we present our case study of the rdist [Cooper 1994] program. The program we studied here is not really a library distribution. However, the change log file provided in the distribution contained interface changes. This is a good example of a list of interface changes that were introduced during some later phase (as identified by their big version number) of a project development. We observed the following interface changes from the change log file of the rdist program (version 6.1.0).

(1) Removing variables

A variable was removed. It is given in Table 2-20.

Table 2-20: Removed variables in rdist

Removed variable	Header file
_PATH_OLDRDIST	pathnames.h

(2) Renaming header files

Some header files were renamed. Table 2-21 illustrates two such examples.

Table 2-21: Renaming header files in rdist

Old header name	New header name
configdata.h	config-data.h
patchlevel.h	version.h

(3) Adding header files

Some header files were added. Table 2-22 illustrates one such example.

Table 2-22: New header files in rdist

New header file
config-os.h

(4) Moving variables between files

Some variables were moved from a file to another. Table 2-23 illustrates one such example.

Table 2-23: Moving variables in rdist

Variable	Old header file	New header file
VERSION	config.h	version.h

2.6 Case study: gzip

We studied the interface changes in the gzip ("ftp.prep.ai.mit.edu:/pub/gnu") program also because of the availability of a good change log file. The following interface change was observed in the change log file of the gzip program (version 1.2.2).

(1) Adding global variables

Some global variables were added to a header file. Table 2-24 illustrates an example of such a change.

Table 2-24: Adding global variables in gzip

New variable	Header file
OF	lzw.h

2.7 Case study: make

The make program is quite an interesting case study. The version number 3.70 tends to indicate it is nearing its maturity but still a number of interesting interface changes were observed. They were extracted from the change log file of the make program (version 3.70). Again, annotations with typical examples are provided to clarify the kinds of changes.

(1) Changing data types (struct) by changing member types

Some data types were changed by changing member types. Table 2-25 illustrates one such example.

Table 2-25: A data type with one member type change in make

Data Type	Member	Old type	New type	Header file
struct child	pid	int	pid_t	job.h

(2) Changing data types (struct) by removing members

Some data types were changed by removing members. Table 2-26 illustrates one such example.

Table 2-26: A data type with one member removed in make

Data Type	Removed member	Header file
struct rule	subdir	rule.h

(3) Changing data types (struct) by changing member names

Some data types were changed by changing member names. Table 2-27 illustrates one such example.

Table 2-27: A data type with a member name change in make

Data Type	Member	New name	Header file
struct commands	lines_recurse	lines_flags	commands.h

(4) Changing data types (struct) by adding members

Some data types were changed by additional members. Table 2-28 illustrates one such changed data type.

Table 2-28: Data types with new members in make

Data Type	Member	Header file
struct commands	command_lines	commands.h
struct commands	lines_recurse	commands.h

(5) Adding global variables

Some global variables were added. Table 2-29 illustrates two such examples.

Table 2-29: Global variables added in rdist

New global variable	Header file
starting_directory	make.h
unblock_sigs	job.h

(6) Renaming global variables

Some global variables were removed. Table 2-30 illustrates one such example.

Table 2-30: Global variables removed in rdist

Variable name	New name	Header file
files_remade	commands_started	make.h

(7) Adding functions

Some functions were added. Table 2-31 illustrates one such example.

Table 2-31: Functions added in rdist

New function	Header file
start_waiting_jobs	job.h

2.8 Case study: diff

We were fortunate to obtain two versions of the diff program and two change log files. The following interface changes were observed in the change log files of the diff program (version 2.7 and version 2.2).

(1) Renaming functions

Some functions were renamed. Table 2-32 illustrates a few of these examples.

Table 2-32: Functions renamed in diff

Old function name	New function name	Header file
bcmp	memcmp	system.h
bcpy	memcpy	system.h

Table 2-32: Functions renamed in diff

Old function name	**New function name**	**Header file**
index	strchr	system.h
rindex	strrchr	system.h

(2) Changing const function parameters to non-const

Some function parameters were changed from const to non-const. Table 2-33 illustrates one such example.

Table 2-33: Changing const function parameters to non const in diff

Function	**Change**
format_ifdef	first argument is no longer const pointer

(3) Adding functions or macros

Many functions and macros were added. Table 2-34 shows a list of those.

(4) Removing functions or macros

Some functions and macros were removed. Table 2-35 illustrates a few of these.

(5) Removing a header file

Some header files were removed. Table 2-36 illustrates one.

(6) Renaming data types

Some data types were renamed. Table 2-37 illustrates one example.

Table 2-34: Functions and macros introduced in diff

New function or macro	Header file
filename_cmp	system.h
filename_lastdirchar	system.h
HAVE_FORK	system.h
HAVE_SETMODE	system.h
initialize_main	system.h
same_file	system.h
S_IXOTH	system.h
S_IXGRP	system.h
S_IXUSR	system.h
SEEK_SET	system.h
SEEK_CUR	system.h
STDIN_FILENO	system.h
STDOUT_FILENO	system.h
STDERR_FILENO	system.h
PARAMS	system.h
index	diff.h
rindex	diff.h
line_cmp	diff.h
version_string	diff.h
change_letter	diff.h
print_number_range	diff.h
find_change	diff.h

Table 2-35: Functions or macros removed from diff.

Removed function or macro	Header file
S_IXGRP	system.h
S_IXOTH	system.h
S_IXUSR	system.h

Table 2-36: Header files removed from diff.

Header file
limits.h

(7) Adding new data types

Table 2-37: Data types renamed in diff

Old function name	New function name	Header file
struct direct	struct dirent	system.h

Some data types were added. Table 2-38 illustrates one example.

Table 2-38: New data types in diff

New type	Header file
enum line_class	diff.h

(8) Renaming global variables

Some global variables were renamed. Table 2-39 illustrates one example.

Table 2-39: Renaming global variables in diff

Old variable name	New variable name	Header file
line_prefix	line_format	diff.h

(9) Adding variables

Some variables were added. Table 2-40 illustrates a list of the examples.

(10) Removing global variables

Some global variables were removed. Table 2-41 illustrates some examples.

Table 2-40: New variables introduced in diff

New variable	Header file
ignore_some_changes	diff.h
horizon_lines	diff.h
default_line_format	diff.h
common_format	diff.h
line_prefix	diff.h
ifndef_format	diff.h
ifdef_format	diff.h
ifnelse_format	diff.h
unidirectional_new_file_flag	diff.h
file_label	diff.h
group_format	diff.h

Table 2-41: Global variables removed in diff

Removed variable	Header file
PR_FILE_NAME	diff.h
Is_space	diff.h
textchar	diff.h

(11) Changing const global variables to non-const

Some const global variables were changed to non-const. This change can be considered as a minor type change. This change probably has no effects on clients, however. Table 2-42 illustrates some examples.

Table 2-42: Global variables with minor type changes in diff

Variable	Header file
group_format	diff.h
line_format	diff.h

(12) Replacing global variables

Some global variables were replaced by others. Table 2-43 illustrates some examples.

Table 2-43: Replaced global variables in diff

Old variable name	Old type	New variable name	New type	Header
ignore_regexp	char *	ignore_regexp_list	struct regexp_list *	diff.h
function_regexp	char *	function_regexp_list	struct regexp_list *	diff.h

2.9 Common patterns of library interface changes

In previous sections, the case studies showed many examples of interface changes. Here, we classify them using a table. Table 2-44 contains a union of all observed interface changes from the case studies. Each "Yes" entry corresponds to a kind of interface change for a software system that was studied.

Our general classification of interface changes is shown in the first column of Table 2-44. The classification contains the following major categories: changes in function prototypes, changes in functions, changes in function semantics, changes in data types, changes in global variables and changes in header files. Some non obvious changes are also shown in Table 2-44. They are: protocol changes and language syntax changes.

Our classification of interface changes enables us to study interface changes in some systematic manner. Thus, we introduce possible interface changes that were not observed in our case studies. So, some changes do not have associated check marks across the table.

Our classification of interface changes is only based on our empirical study of interface changes. It may not be complete because the case studies may not be complete. Further-

Table 2-44: Patterns of interface changes

Change	Borl[a]	libg++	nihcl	rdist	gzip	make	diff
Function prototypes							
Renaming functions	Yes						Yes
Adding parameters	Yes		Yes				
Removing parameters	Yes						
Reordering parameters	Yes						
Changing defaults to non default		Yes					
Changing non defaults to default							
Function semantics							
Changing parameter types		Yes	Yes				
Changing return types		Yes					
Changing return value meaning	Yes	Yes					
Changing parameter meaning							
Function replacements							
Adding functions	Yes	Yes	Yes			Yes	Yes
Repacing functions	Yes		Yes	Yes			Yes
Data types							
Renaming types		Yes	Yes				
Changing data type semantics							
Renaming fields in data types						Yes	
Adding fields to data types						Yes	
Replacing fields from data types						Yes	
Adding types							Yes
Replacing types				Yes			
Global variables							
Renaming global variables	Yes					Yes	Yes
Adding global variables					Yes	Yes	Yes
Replacing global variables			Yes	Yes			
Header files							
Renaming header files		Yes	Yes	Yes			
Adding header files		Yes		Yes			Yes
Repacing header files							Yes
Transfer between header files		Yes		Yes			Yes
Protocol change (e.g. Section 2.2.1)	Yes						
Language syntax (e.g. Section 2.2.1)	Yes						

a. Case study: Borland C++ (see Section 2.2)

more, the study of interface changes was based on documents which themselves are not necessarily accurate (see Table 2-7 and the surrounding discussion).

The quality of individual software systems should not be inferred from Table 2-44. Some people may conclude that a software system that has fewer interface changes is better software. But Table 2-44 does not serve that purpose. The number of interface changes may be related to the length of history or revisions of a system. Also, a software system that discloses interface changes in an explicit manner may also have more interface changes. Further, the size of the system is clearly a factor.

The interface changes are classified based not only on their syntactic constructs but also their semantics. The consequences of these changes on the application maintainer are discussed below. In general, every application maintainer needs to spend a lot of effort to upgrade their source code for these changes. However, most of the upgrade process can be automated to reduce the effort required by each application maintainer.

2.9.1 Changing function prototypes

The most common interface changes probably fall within the category of function prototype changes. Some researchers interested in program restructuring [Griswold & Notkin 1993] have also identified some need for function prototype changes. In the rest of this section, we present the kinds of changes within this category.

(1) Renaming functions

One common function prototype change is renaming functions. Renaming a function means that only the name of the function is changed but all else remains the same. When a function is renamed, the application maintainer needs to determine if there is a function call using the old version and needs to rename that to a new name. A scan of all the func-

tion calls by the old name yields all possible sites of such calls. But some of these calls may not call the changed function. For example, if an application source does not include the header file containing the change, i.e., the application source does not use the changed function, it should not rename the function.

(2) Adding function parameters

Another common interface change is adding parameters to a function. In some programming languages, a default value can be used for a parameter. If that is the case, the library maintainer may specify that default value and there should be no upgrade problem because no application code needs to be changed. If the language does not support default parameters, or if a default value is not used, then the application maintainer needs to search for the function calls which use the old version and change them to use the new version by inserting an appropriate argument. The library maintainer should be able to pick an appropriate argument without no trouble.

There is actually a good reason why library maintainers are sometimes unable to use default parameters even when the language has that feature. Most programing languages (e.g. C++, Ada) only support default parameters as the right most parameters. If a parameter needs to be inserted at the front, no default values can be used.

(3) Removing function parameters

In contrast to adding function parameters, removing a function parameter does not appear to be a common interface change. But it may happen, for example, when an argument can be ignored in the recent release. Even in this case, the application maintainer still needs to adjust the number of parameters in the function call to adapt to this new prototype.

(4) Reordering function parameters

Another function prototype change, reordering function parameters, may seem silly. But consistency among all function prototypes in a set of libraries may dictate this need. For example, the reverse order of the arguments in bcopy() and memcpy() is an obstacle for people learning the use of the C runtime library. In general, application programmers need to check for function calls that use the old interface and reorder the arguments accordingly.

(5) Changing default parameters to non-default ones

One function prototype change that is similar to adding parameters is changing default parameters to non-default ones. However, it is easier because the default is known. Because of the known default value, application maintainers need to insert the default value only when that argument is not supplied. When that argument is supplied, the function call should not be changed.

(6) Changing non-default arguments to default ones

The last function prototype change we studied is changing non-default argument to default ones. Since all existing calls should still work, this kind of interface change poses little problem for application maintainers.

2.9.2 Changing function semantics

Another category of interface change related to functions is changing function semantics. Function semantics are changed in two ways: changing types associated with a function

(including return types and parameter types) and changing the interpretation of values (including return values and parameter values).

(1) Changing function return types

The first kind of function semantic change we studied is changing function return types. The return type of a function may be changed to a more general type, a more limited type, or even a different type.

When a function return type is changed and a function call to that function has been determined, the application maintainer may not need to upgrade the code if the return value is ignored by the application source. If the return value is used, the application maintainer may use a suitable type conversion function to convert the new value to the old type of the receiving variable, or change the type of the receiving variable to match the new return type. The former method requires the old return type is still available while the latter method may require multiple changes because the variable may be declared in another place and the variable may be used in other places, too.

(2) Changing parameter types

Another function semantic change is changing parameter types. There are many possible reasons for this change. For example, the type of an argument may be changed due to other data type changes (Section 2.9.4). When a parameter type is changed, the application maintainer may need to apply a suitable type conversion function to the old function argument to convert it to conform to the new type, if implicit conversion is not available.

(3) Changing the meaning of parameters

Function semantic changes include not only type changes but also interpretation of values even if there is no type change. The meaning of the value of a parameter may be changed for many reasons. For example, the meaning of some special integer may have a different meaning because of hardware changes. When the meaning of a function parameter is changed and a function call using that function is detected, the application maintainer needs to replace the old argument with an appropriate new expression.

(4) Changing the meaning of function return values

Yet another function semantic change is changing the meaning of function return values. In fact, this appears to be more popular in the case studies. For example, in the study of Borland C++ class library, many search functions return NPOS (i.e., -1) instead of 0 when the search fails. Perhaps the most typical problem is the interpretation of the error status of a function call, which is commonly returned as an integer value. In some systems, the return value is 0 when there are no errors. But in other systems, the return value is 1. Thus, changing the meaning of the return value is sometimes made to achieve consistency in the interpretation not only within a library but also across libraries. When the meaning of a function return value is changed and a function call using that function is detected, the application maintainer may apply a suitable value conversion function to the return value, if the return value is used.

2.9.3 Rearranging functions

A third category of interface changes is the rearrangement of functions, which include deleting, adding and moving functions.

(1) Deleting functions

The library maintainer may choose to delete functions for many reasons. Usually, another function will be provided to supplement the removed function. In this case, the application maintainer should locate the old function calls and replace them with appropriate calls.

(2) Adding functions

A more common kind of rearranging functions is adding more functions. It directly supports new features added to a library. Unless some other functions are deleted, there is no reason to believe that the application code needs to be changed immediately in response to this kind of change.

(3) Moving functions

Moving functions is, perhaps, a result of restructuring a library. For example, a group of functions may be split from one file to several files or joined from several files to one. Both can be carried out for reasons of maintaining cohesion within a file. When a function is moved from one header file to another and the usage is detected, the application maintainer should include the new header file, if it is not already included.

2.9.4 Changing data types

Another category of interface changes is changing data types. Data types are perhaps less susceptible to changes if information hiding is used. However, some language features (e.g. C struct) does not support this. In fact, more data type changes were observed in the C systems than in the C++ systems.

(1) Renaming data types

A common data type change is renaming data types. When a data type is renamed, the application maintainer just needs to rename old usage of the old name with new usage of the new name.

(2) Changing data type semantics

Another data type change is changing data type semantics. While this change was not observed in the case studies, it is conceivable that the interpretation of the values of data members may change in the course of time. For example, for the same data type,

```
struct T_pair { int x; int y; };
```

The old interpretation could be that (0, 0) is the center of the screen while the new interpretation could be that (0, 0) is the upper left corner.

When the data type has a new semantic meaning, the library maintainer does not have to fix the place of the declaration of the type but has to replace the usage of the members of the type in many other locations with appropriate expressions.

(3) Renaming data type members

Another data type change is renaming data type members. When a member of a data type is changed, the application maintainer needs to replace usage of that member with a new name.

(4) Adding data type members

A easier data type change is perhaps adding data type members. This change usually does not require any upgrade effort by the application maintainer because this should have no effect on existing code at all.

(5) Replacing data type members

A less easy data type change is replacing data type members. This change requires upgrade effort by the application maintainer if the removed member is used. There is perhaps no general solution to this problem. Fortunately, it is not expected that a member would be removed without adjustment by other suitable mechanisms.

(6) Adding data types

Along with adding functions, adding data types provides a change that enhances the features of a system. In general, like adding functions, this change should require no extra effort from the application maintainer.

(7) Replacing data types

Replacing data types, like replacing functions, though undesirable, forces application maintainers to remove some old usage of obsolete data types. When a data type is removed, the application maintainer should study other new features that are presumably introduced to a library and find a suitable replacement. This manual process can be potentially much improved by using a tool that is aware of this change.

(8) Moving data types

The last kind of data type changes we observe here is moving data types from one header file to another. When a data type is moved from one header file to another, the library maintainer needs to include the other header file for the application source to continue to work. This manual process can be potentially much improved by using a tool that is aware of this change.

2.9.5 Changing header files

A category of interface changes that distances open system software evolution from closed system software evolution is changing header files, which may not be needed to directly consider in the closed model.

(1) Renaming header files

One common header file change is renaming header files. Renaming header files is sometimes used to provide a uniform naming scheme for a library. Other times, it is used to port header files to another file system that have some file name restrictions. When a header file is renamed, the library maintainer needs to replace the old header file name with the new one. It is important to change only the #include or similar directives in some other systems that includes the file (not so straightforward because #include directives may depend on the user's prevailing search path which is defined by the user in many ways, e.g. specified in a Makefile or dependent on environmental variables).

(2) Adding header files

Another change is adding header files. Adding a header file may provide additional features that can be directly used by the application. However, since the application code did not use the header file already, there should be no effect.

(3) Replacing header files

Yet another change is removing header files. Removing a header file breaks all code that depends on the interface provided by that header file. The library maintainer probably provides other header files to provide similar kinds of services. The application maintainer just needs to find out what services are now available and apply them to the application code. This manual process can be improved by using a tool that recognizes the interface change.

2.9.6 Changing global variables

Another category of interface changes is changing global variables. It is commonly believed that global variables are not good interface choices. However, in some programming practices, they are still being used.

(1) Renaming global variables

The first kind of interface change within the category of changing global variables is, again, renaming them. When a global variable is renamed, the application maintainer just needs to replace each use of the old name with a new name. The application maintainer cannot rely on a textual find and replace of the source because scoping, common in pro-

gramming languages, allows a variable to hide another of the same name from an outer scope.

(2) Adding global variables

Another interface change is adding global variables. Adding global variables to a header file does not introduce upgrade problem to an application, however.

(3) Replacing global variables

In contrast, another interface change, replacing global variables, introduces upgrade problem to an application. When a global variable is removed from a header file that is used by an application, the application maintainer needs to search for all usage of the removed global variable and replace each usage with a suitable one, which may not be easy to determine.

2.9.7 Changing protocols

A rather complicated interface change is the change of protocols. A protocol change refers to the change of the order of invoking a group of functions. We observed one such example in the case study of Borland system (see Section 2.2.1). It is hard to believe there is a general way to deal with this kind of changes, however. A tool may perform well on some instances of this problem. Even in very complex cases, the least a tool can do is to flag all possible problems and alert the application maintainer.

2.9.8 Changing syntax

A category of interface changes that is not common in well established languages is changing the syntax of a language. In fact, although we observed this in our case studies (see Section 2.2.1), this change does not come from the principal programming language, but instead comes from a supporting development language, Borland's MAKE. When the change is trivial to handle, as in the case of Borland's MAKE, a tool can automate the process beautifully.

2.9.9 Some language specific changes

The changes described earlier are quite language independent. For C++ libraries, there are special changes that are specific to the C++ language, e.g., adding, renaming and removing a class member. These are similar to the changes that can be made to a C struct, which was covered earlier. Although a class member can be a function in addition to a data member, this poses no new problem. However, the access mode of a member needs to be dealt with in C++. Also, a member function can be const or non const. We would need to introduce the access mode and the constness of an expression of an object to deal with these C++ specific problems. Inheritance of classes also poses new problems. In this dissertation, we do not address these problems directly.

2.10 Discussion

In real life, as we observed in the case studies, interface changes do not come individually. The library maintainer introduces multiple but related interface changes in each revision. The combination of interface changes may have a different meaning from the sum of individual changes. For example, the result of removing a function and adding another function may be a replacement of a function.

The library maintainer should know the net result of a group of changes that he introduced to a library release. As a result, requiring the library maintainer to specify changes in a library release has an added advantage of resolving a group of interface changes into a few more meaningful ones.

It is difficult to quantify what kinds of interface changes are more common. It would be nice to quantify this with respect to either the kinds of the programs or the development stage of the programs. Sometimes, a batch of interface changes arose because of standardization of languages or programming style within an organization. Getting a good observation perhaps requires us to look at many more case studies in detail and characterize them according to some criteria listed earlier. That would indeed be a useful study.

Yet another thing we would really like to discover is what kinds of interface changes are more costly. This cannot be determined with just the case studies on the libraries alone. For it to be determined, the cost to all the application developers who depend on a particular library change needs to be known. Again, even though this is an unknown number, it is hard to believe that this is a small factor.

2.11 Chapter Summary

In this chapter, we presented 7 case studies and summarized the interface changes that were observed from those studies. The case studies presented in this chapter are empirical and they strongly support our taxonomy of interface changes. The patterns of changes are dependent on many factors (e.g. types of applications intended for, size of the users, the personality of the library maintainer and the culture of the organization). Nevertheless, we believe a general mechanism to handle interface changes across organizations can provide an effective solution to the task of upgrading application code in response to interface changes. The overview of our change process is presented in Chapter 3.

Chapter 3

The Change Process

"You must be the change you wish to see in the world." - Gandhi

In the previous chapters, we identified the problem of supporting changes in open system software evolution. We classified library interface changes as an important example of the problem. This chapter outlines the change process. Included in the outline is an explanation of how the process can be used to make maintenance for open system software evolution a less costly process for the library maintainer and, especially, for the application maintainer. Further details are provided in the next two chapters. Chapter 4 describes the change specification language used in our approach. Chapter 5 describes the design of the supporting tools.

Our approach is driven by the characteristics of acceptable solutions (see Section 1.4). It supports flexibility in propagating changes from one source to another and specifying the needed changes. A general change specification mechanism is provided. Automating change propagation will greatly reduce the tedious work for each application maintainer to upgrade their code.

We provide a change specification mechanism based on the lexical and syntactic structures of common programming languages. This allows the library maintainer to easily specify most common interface changes (described in Chapter 2) and the necessary responses to those changes. A tool set is also provided to automatically handle the propagation of changes based on the change specifications. Further, a mechanism is provided to apply required changes to the application code. Finally, our approach regenerates the

application code with the required changes incorporated without changing unaffected portions of the code.

3.1 Overview of the change process

The change process in open system software evolution has two major phases. The first phase is the specification of the interface changes by the library maintainer. This phase only happens once – when a revision of the library is released. The second phase is the adaptation of the application code in response to the interface changes. The second phase happens when each application upgrades at a later time when the newly released library is used. Clearly the first phase must be completed before the second phase can proceed.

Figure 3-1 illustrates the change process (similar to Figure 1-8). In this figure, the upper half of the figure shows the first phase of the change process during which the library maintainer specifies the interface changes. The lower half of the figure shows the second phase of the change process during which the application code is adapted in response to the library interface changes.

3.1.1 First phase: change specification

The first phase of the change process is change specification. The change specification includes interface changes and the responses required for application code to adapt to interface changes. The design of the change specification language appears in Chapter 4.

The change specification is manually written by the library maintainer. An alternative might be to automatically deduce a specification based on changes between two versions of a header file. Such an approach might be preferred by the library maintainer. However, that technique has a number of limitations. First, common programs (e.g. diff) that deduce

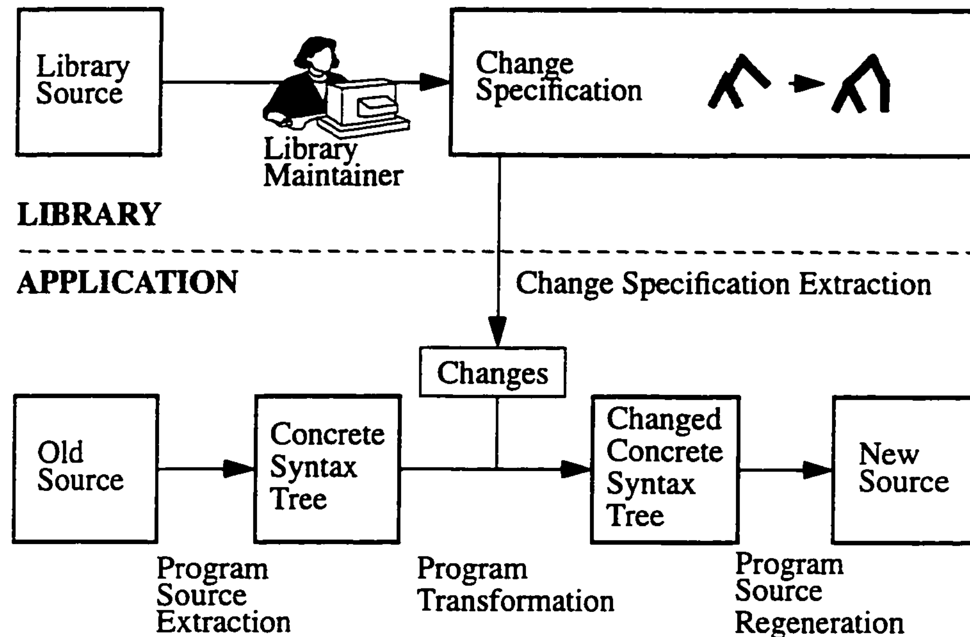


Figure 3-1: Two phases of the change process

difference between two files record only the textual difference,[1] which is not necessarily the same as interface changes. A change in a comment is not an interface change. Having no textual changes, on the other hand, does not mean that there is no interface change. For example, swapping two arguments of a function may not yield any textual change in the header file. Second, the difference between two releases of a library is not the sum of the differences between individual files. For example, the transfer of some entities from a header file to another is not easily determined simply by comparing each pair of versions of header files. Third, the intent of the changes and its consequence may be difficult to determine from the difference in the versions. For example, when a function prototype is changed from having one parameter to two parameters of the same type, it is not clear whether an extra parameter is added to the first or second position.

1. Some program can detect semantic difference [Jackson & Ladd 1994] but with some limitation (see Section 7.5).

Writing a change specification not only makes the interface changes explicit, but also makes possible automation in the later processes in the second phase of the change process. In fact, this first step is the only major manual process. By using a suitable change specification language, the library maintainer may not find it too difficult to write change specifications.

The necessity to write change specifications manually may also discourage the library maintainer from making frequent changes to the library interface. Nevertheless, change specification makes it possible for the library maintainer to change interfaces when it is necessary to do so.

The change specification is written in a change specification language and included as comments in header files. A language to specify changes has many advantages. First, it creates the possibility for testing change specifications with tools. Second, it forces the library maintainer to write changes in a standard way and make both the interface changes and their consequences more clearly visible. As part of the header files, specifications can be released along with the header files and would never be out of sync with the library release. The change specification is included as comments in header files so that no new language feature needs to be introduced thus making this technique generally applicable to systems written in common programming languages.

Change specifications can be verified by extensive testing on the library side of open system software evolution (Section 1.1.4). Although testing alone cannot tell the absence of bugs, the library maintainer can use existing test suites to gain confidence not only in the new library release but also the interface change specifications from the previous release. In other words, the library maintainer can apply the change adaptation to their test suites just like the application maintainer. Thus, an existing test suite for a library can be reused to raise a maintainer's confidence about the accuracy of the change specifications he has written.

The change specification language needs to support change specifications in two ways: specifying interface changes and specifying necessary actions required on the application code. The interface change specifications are necessary to signal the possibility of outdated and incorrect usage of the interface. The adaptation actions suggest ways that can be applied to the application code to remedy the situation.

3.1.2 Second phase: change adaptation

The second phase of the change process is change adaptation. It is illustrated in the lower half of Figure 3-1. The tools automatically process applications to apply appropriate changes. Specifically, the tools extract the interface change specifications written in the change specification language and apply necessary changes to the application code.

The change adaptation process automates these major tasks (arrows in Figure 3-1) when applied to each application source:

1. extracting change specification from header files ("Change Specification Extraction"),
2. extracting the program's representation ("Program Source Extraction"),
3. modifying the program representation based on the change specification ("Program Transformation"), and
4. regenerating program source from the modified representation ("Program Source Regeneration").

The extraction of change specifications from the header files is needed to obtain the change specifications, which may be applied later. The extraction of the program's representation provides a convenient data structure to modify the program. The modification

combines the results of the previous two tasks. It results in a modified program representation. A new program source is generated from the modified program representation.

3.1.3 Program Representation

In our approach, concrete syntax trees [Aho et al 1986, p.49] are used to represent programs. Syntax trees are well studied, making them easy to use. An easy to use representation is important because the program's representation also influences the design of the change specification language by constraining how changes can be written.

A program representation also needs to support program transformations conveniently. Syntax trees are primary representations of programs [Morgenthaler & Griswold 1995] and thus make them suitable for program restructuring in common programming languages.

Other representations – the text itself, control flow graphs and program dependency graphs – were also considered but rejected. Program text supports change specifications based on regular expressions. But the natural limitation of regular expressions is that it is not as rich as context free grammars. For example, balanced parenthesis, an important language feature, cannot be specified in regular expressions alone. However, a program text representation requires less space than a syntax tree representation. While constructing and maintaining a syntax tree for an entire program over all source files is expensive, constructing and maintaining a syntax tree for individual program source files is much cheaper. As we will describe later in Chapter 5, we design our approach to use only a subset of the complete syntax tree for a file, thus further reducing the cost. Using syntax trees also makes it easier to use type information, scope information and symbol tables because of the extra help from syntactic analysis. Also, grammars are already available for common programming languages (e.g. C) in many compiler construction systems (e.g., yacc [Johnson 1975], the Purdue compiler construction tool set [Parr 1995]).

We found it necessary to augment concrete syntax trees with extra information in order to allow the regeneration of program text with the same look and feel (see Section 1.4). Typical extra information examples are spacing, indentation and comments in the source. The design of our concrete syntax trees is detailed in Section 5.2.7.

3.2 Requirements of the change process

Earlier in this chapter we mentioned the tasks requirements for the change process. In this section, we describe the programming paradigm requirement and the requirements for the tools to carry out the tasks.

3.2.1 The programming paradigm

In our model, all of the information needed for interface changes is provided in the header files. The rest of the library, namely implementation of the interface, must eventually be linked but otherwise is not involved during program transformation in response to interface changes. Header files are a logical place to put interfaces. For example, interfaces are often written in the header files in C and C++ languages. In Ada, interfaces are written in specification files, which are similar to header files. Many other languages (e.g. Turbo Pascal) also support this feature.

This requirement has a further advantage in that it reduces the program text to analyze by ignoring everything else from the library (e.g. README files, implementations of library interface) except the header files. Our approach also encapsulates change specifications within the header files and simplifies the design of the change specification language to some extent.

The use of header files means that the programs that are considered for open system software evolution are file based, i.e., each segment of a program resides in a file. A non file based language, e.g. Smalltalk, can use our approach with some adjustments through the user interface of the programming environment. Still, it is reasonable to believe that far more common programs (e.g., C, C++, Fortran, Ada) are still written in file based systems.

Constructs for comments are also required in the programming languages. This requirement provides a place to write change specifications without the need to add features to the programming languages. Again, this feature is available in almost all common programming languages.

3.2.2 Tools

The change process requires a number of tools to aid in its automation. The requirements of these tools are described here. Tools are implementations of the edges shown in Figure 3-1.

(1) Change specification extractor

A change specification extractor extracts change specifications from the library header files. While change specifications are still quite readable and understandable (see Chapter 4), not requiring the application maintainers to read them allows maximal automation possibility. A complication with change specifications in header files is that not all change specifications require an application program source to update. For example, if a changed function was not used by an application, there may be no need to upgrade the application in response to that change. An application program source should only need to respond to an interface change provided that interface was already used by the application. Thus, the change specification extractor needs to detect (1) the use of the new library release from the application program source, and (2) the use of the header files that contain interface

changes, before it can extract the change specifications from those header files. In our approach, the change does not propagate when it is made to the library. Instead the change is propagated when the new library is used by an application. If the user does not upgrade his/her software using the new library, then the change will not propagate there. When the changed library is used, i.e., being parsed as a header file included in a user source file, the change specification is noted and the user code is then changed accordingly.

The change specification extractor scans a header file for change specifications which are written in the change specification language (see Chapter 4). Section 5.2.2 will detail our design of the change specification extractor.

(2) Program source extractor

A program source extractor extracts the program source into the intermediate representation used for manipulation and change when necessary. Thus, a program source extractor is similar to the front end of a compiler [Aho et al 1986]. However, as mentioned earlier, the difference between our program source extractor and a compiler front end is the need to keep richer program information, including information such as spaces and comments that plays no role in compiler code generation, in the program representation. This is to help regenerate the program source later. Section 5.2.5 will detail our design of the program source extractor.

(3) Program transformer

A program transformer upgrades a program to conform to new interface changes and their requirements. The program transformer manipulates the syntax tree program representation. The transformation can be based on pattern matching of program constructs (see Chapter 5) and changing those constructs as required by the library change. The change specifications from the header files should be used here to apply the changes to the appro-

priate locations of the application program. A typical example of a program transformation matches a subtree of the program representation with a change specification, and then transforms the subtree according to the action rule specified for this change. A typical action rule reorganizes a tree structure. The tree patterns and the action rules are parts of the change specification language which is described next in Chapter 4.

(4) Program source regenerator

A program source regenerator regenerates the program source while maintaining the integrity of the source before the change. The program source regenerator must maintain the integrity of the source to achieve an acceptable solution (see Section 1.4). The source code can be regenerated when the changes have been propagated and the syntax trees have been modified accordingly.

3.3 Chapter summary

In this chapter, the rationale behind the development of the change process was given. Further requirements of my design of the change process were listed. The requirements include a simple and common programming paradigm and four tools to perform the four major automation tasks: extracting change specifications, extracting program sources, transforming programs and regenerating program sources. All four tools need to work with a single program representation. The selection of concrete syntax trees as our program representation was also explained.

Chapter 4

The Change Specification Language

The need for a change specification language was established earlier in the description of the change process (see Chapter 3). This chapter describes the design and definition of a language that meets that need. The design of the change specification language also influences the design of the tool set (see Chapter 5) that propagates the changes to application code.

4.1 The criteria for a good change specification language

There are many aspects to the criteria for a good change specification language. First, the change specification language should be expressive. That is, it should allow the library maintainer to express common interface changes described in Chapter 2. Second, the language should be easy to use by the library maintainer when writing change specifications. Third, the language should be readable by both the library maintainer and the application maintainer. Although the specifications are processed by tools, the readability of the language is still important for the application maintainer to see what changes are applied. Fourth, it should be easy to build parsers for the language. The performance of extracting and application of changes can thus be increased.

Expressiveness of a change specification language is the most important criteria. It must express the interface changes that are needed. Among the changes that it can support, a change specification language should be expressive enough to describe changes that were covered earlier in the case studies (see Chapter 2). The interface changes that need to be handled are function changes, data type changes and global variable changes. The lan-

guage needs to express patterns that can be matched against function calls, use of data types (e.g. variable declarations) and use of global variables. These syntactic constructs are important for recognizing code that needs to be updated.

4.2 The design of the change specification language

Our design of the change specification language focused on the expressiveness of the language. Each change specification in our language consists of a variable number of keywords for that change and an expression for each keyword. Figure 4-1 shows an example with a set of keywords for a function prototype change and their corresponding expressions. We match tree patterns and then describe the transformations on those matched subtrees. The expressions and the keywords will be described shortly.

```
int foo(int a, int b);
/*
 * _ASE_BEGIN_
 * FNAME = foo
 * ARGS = 2
 * PATTERN = #(fc:FCALL_foo_2 id:ID lp:L_PAREN b:gen_expr co:COMMA
c:gen_expr rp:R_PAREN)
 * ACTION = #gen_logical_expr = (SORASTBase*) #(fc, id, lp, c, co,
b, rp);
 * _ASE_END_
 */
```

Figure 4-1: A change specification to swap two arguments (a and b) of a function call.

The expressiveness of our language is achieved by allowing our expressions to match syntax tree patterns. Syntax trees are good program representations. They shield library maintainers from the distraction of non program elements such as comments and spacings, as well as some program elements such as expression arguments (when the library maintainer does not care whether the actual argument is a complex expression). In using syntax

tree patterns, library maintainers only need to focus on the program structure that is affected, not the actual program source.

The usability of our language is enhanced by the ability to insert change specifications as comments in a library distribution. Thus, it does not require changing a programming language. Our language is also quite easy to read for two other reasons. First, it is written in a structured format as shown in Figure 4-1. Second, a syntax tree notation [Parr 1994] is used to express tree patterns. The implementation to parse this language is made easier by having a structured format so that it can be parsed mostly by matching regular expressions.

Our design of the change specification language also keeps two specific goals in mind. One is to support the general idea of supporting interface changes in open system software evolution (language independent). The other is to support a specific programming language (language dependent). These two features make it easy to adapt our approach to different languages without changing the structure of our change specification language. A simple specification structure of the following form achieves both goals.

```
IDENTIFIER = EXPRESSION
```

IDENTIFIER is a language independent keyword for supporting interface changes while EXPRESSION is the language dependent expression for IDENTIFIER. Here is a specific example taken from Figure 4-1.

```
FNAME = assign
```

In this example, FNAME specifies a function name. The function name specified is "assign".

A change specification consists of a number of simple specifications as described earlier but bracketed in a pair of keywords. The use of change bracketing keywords allows effi-

cient parsing and analysis of change specifications. The canonical form of a typical change thus looks like Figure 4-2.

```
Begin change bracketing keyword
Identifier(1) = Expression(1)
...
Identifier(n) = Expression(n)
End change bracketing keyword
```

Figure 4-2: The canonical form of a change specification.

For example, in Figure 4-1, _ASE_BEGIN_ and _ASE_END_ are bracketing keywords. The details of the function interface change is captured between these two keywords. In this example, 4 identifiers (FNAME, ARGS, PATTERN, ACTION) are used. Each of them is assigned an expression. The type of an expression depends on the keyword. For example, the type of the expression for FNAME is a name. The type of an expression for ARGS is an integer. The type of an expression for PATTERN is a concrete syntax tree pattern. The type of an expression for ACTION is Sorcerer [Parr 1994] or C++ code.

Tables 4-1 - 4-5 show the list of the keywords from our change specification language.

4.2.1 Change specification bracketing keywords

To help ease the extraction of the changes from header files, we introduce four pairs of keywords to bracket each change. Table 4-1 shows these bracketing keywords.

4.2.2 Function change specifications

Table 4-2 shows the list of the keywords for function change specifications. Some of these keywords were already used in Figure 4-1 but they are described in more detail here.

Table 4-1: bracketing keywords for change specifications

Bracketing Keywords	Meaning
_ASE_BEGIN_	The begin of a function change specification
_ASE_END_	The end of a function change specification
_ASE_DT_BEGIN_	The begin of a data type change specification
_ASE_DT_END_	The end of a data type change specification
_ASE_GV_BEGIN_	The begin of a global variable change specification
_ASE_GV_END_	The end of a global variable change specification
_ASE_IH_BEGIN_	The begin of an include header change specification
_ASE_IH_END_	The end of an include header change specification

Table 4-2: Keywords for function change specifications

Keyword	Meaning
FNAME	a function call of this name
ARGS	the number of arguments used in the function call
PATTERN	the concrete syntax tree pattern of the function call
GUARD	further semantic condition for the match of the function call
ACTION	what should be done if both syntactic and semantic match is found
FAIL_ACTION	what should be done if syntactic match but not semantic match

In our design of our change specification language, function change specifications use six keywords: FNAME, ARGS, PATTERN, GUARD, ACTION and FAIL_ACTION. These keywords define the function calls that changes are required to make to them to upgrade them to use the new interface. FNAME specifies the function name that an application uses to call the function. ARGS specifies the number of arguments that are passed to that call. For a language that does not support function overloading, ARGS may be unnecessary. ARGS alone is not sufficient, however. GUARD further constrains the matching by adding type checking. The pattern to be matched is a concrete syntax tree pattern specified by PATTERN. ACTION specifies how the function call should be changed if the pattern is matched syntactically and semantically (if GUARD is used). FAIL_ACTION specifies how the function call should be changed if a syntactical concrete syntax tree pattern is matched but the GUARD condition is not matched. Thus, FAIL_ACTION is used with GUARD.

In our change specification language, the keywords FNAME, ARGS, PATTERN and ACTION are always required. If GUARD is used, FAIL_ACTION should also be used.

Figure 4-1 shows a simple example of a function change specification as it might appear in a C header file. In this change specification, four keywords are used. The function name to be looked for is "foo" (FNAME). The number of arguments to match for the function call is 2 (ARGS). The other two expressions will take a little longer to explain.

We found the tree patterns used by Sorcerer a convenient way to manipulate syntax trees. The concrete syntax tree pattern is specified using tree syntax used by Sorcerer [Parr 1994]. Tree patterns are specified in a LISP-like notation of the form:

```
#(root-item item ... item)
```

Root-item is a token defined in the grammar specification (Section 5.1) of the programming language used by the library maintainer. An item can be either a token, a grammar rule or another tree. Rules are also defined in the grammar specification of the language. An example of a grammar using Purdue Compiler Construction Tool Set [Parr 1995] can be found in Appendix A.1. The list of items are specified from the left most child of this tree to the right most child.

In our change specification language, an item consists of two parts of this form:

```
VarId : Token
```

Tokens are specified in the grammar for the programming language. "VarId"s are local handles to the matched tokens. They can be viewed as local variables within this change specification. In Figure 4-1, the symbols fc, id, lp, b, co, c, rp are handles that match the tokens/rules FCALL_foo_2, ID, L_PAREN, gen_expr, COMMA, gen_expr, R_PAREN. Figure 4-3 describes the tree pattern in a diagrammatic form.

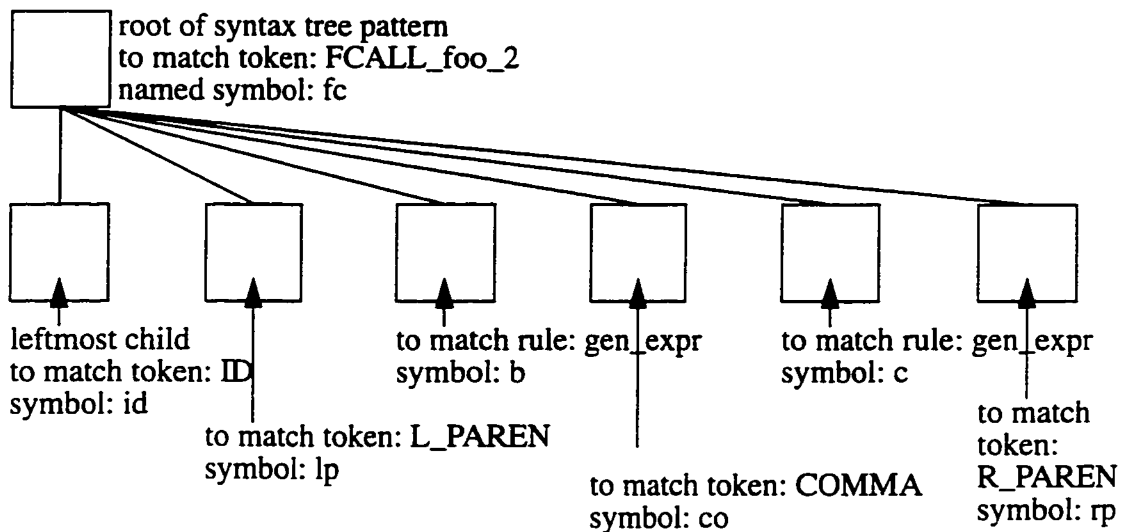


Figure 4-3: A tree pattern in a pictorial form

The root tokens are generated in our approach to ease the implementation. During the tree construction phase, a "foo" function call will be constructed into a tree similar to the one shown in Figure 4-3. The use of artificial root tokens, e.g. FCALL_foo_2, eases the next phase for the search and replace of function calls.

In a function change specification, ACTION has this form:

```
ACTION = #gen_logical_expr = (SORASTBase*) [tree pattern]
```

where #gen_logical_expr is an artifact of our transformation specification. Our complete transformation specification is given in Figure A.2. Here, both the use of #gen_logical_expr and (SORASTBase*) should be treated as fixed. The interesting part is the tree pattern specifications in the ACTION rule. Notice that in Figure 4-1 "b" and "c" are swapped from the PATTERN rule to the ACTION rule. Figure 4-4 graphically represents the action of this change.

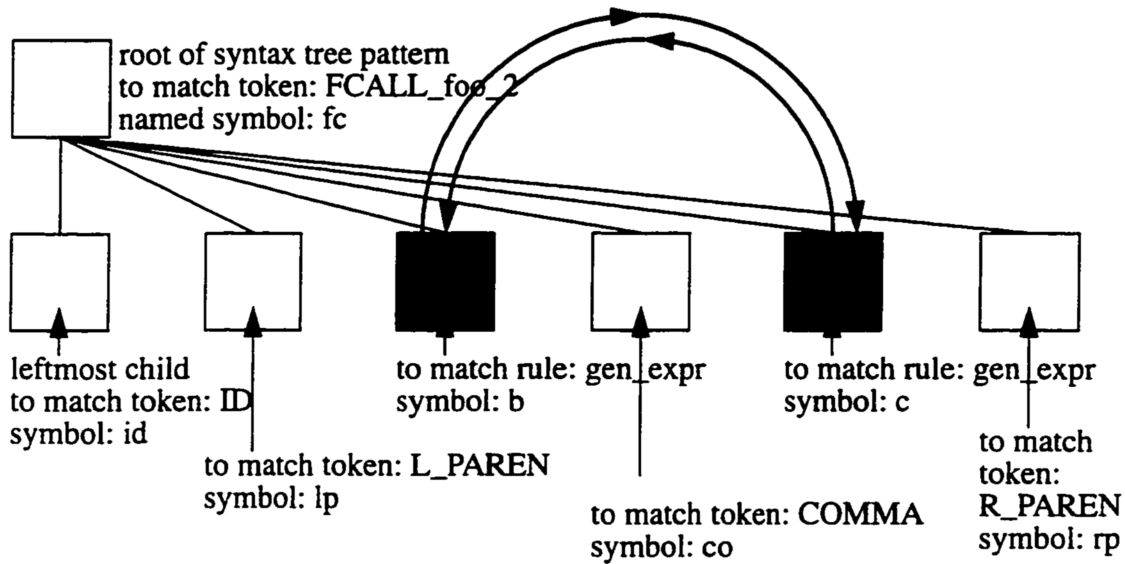


Figure 4-4: The result of the simple swap specification.

The previous example only matches a "foo" function call when two arguments are passed. In a language that supports overloading, this is not sufficient. Our approach allows a library maintainer to further specify semantic requirements for a match. A semantic requirement is usually a type constraint. Figure 4-5 shows an example that uses GUARD and FAIL_ACTION to explore this option.

In Figure 4-5, the GUARD condition constrains the pattern to match only those calls where the type for "b" is "integer" and the type for "c" is "char". The specification of the GUARD condition is based on a few supporting programs for the programming language (see Section 5.2.10). In our case, we chose to support "GetType()" function, which returns the type of a tree, and "Same()" which compares two types. Our approach also supports other functions such as the examination of values. However, the fact that values can only be determined at compile time limits the extensive benefit of those functions. The FAIL_ACTION rule essentially says that the tree should remain as it is if the GUARD condition is not met.

```
// This example shows the swap of 2 arguments w/ the need to check
// parameter types.
//
int swap(int x, int y);
/* old: int swap(int x, char c); */
int swap(char c, int x);

/*
 * _ASE_BEGIN_
 * FNAME = swap
 * ARGS = 2
 * PATTERN = #(fc:FCALL_swap_2 id:ID lp:L_PAREN b:gen_expr
co:COMMA c:gen_expr rp:R_PAREN)
 * GUARD = b->GetType()->Same(intType) &&
   c->GetType()->Same(charType)
 * ACTION = #gen_logical_expr = (SORASTBase*) #(fc, id, lp, c, co,
b, rp);
 * FAIL_ACTION = #gen_logical_expr = (SORASTBase*) #(fc, id, lp, b,
co, c, rp);
 * _ASE_END_
 */
```

Figure 4-5: A change specification to swap two arguments with type checking.

The next function change specification example (Figure 4-6) shows a solution to the "assign" problem outlined in Section 1.2. The example has a slightly complicated action rule due to the need to create nodes in a tree. It uses data structures defined in the supporting programs for the transformations to introduce two new tokens, INT and COMMA, and puts them in a syntax tree. Notice the introduction of symbols my_1 and my_2 in the last part of ACTION rule.

Essentially, the example in Figure 4-6 says the following:

```
If there is any code that uses the function assign(), check the
number of parameters.If it is 2, don't do anything. If it is 3,
insert a third argument of 0 in the caller.
```

```
// This change inserts a third argument when 3 arguments
// are used.
//
// was: int assign(char* dst, char* src, int len = 0);
// now:
int assign(char* dst, char* src, int start = 0, int len = 0);

/*
 * _ASE_BEGIN_
 * FNAME = assign
 * ARGS = 3
 * PATTERN = #(fc:FCALL_assign_3 id:ID lp:L_PAREN arg1:gen_expr
col:COMMA arg2:gen_expr co2:COMMA arg3:gen_expr rp:R_PAREN)
 * ACTION = AST* my_1 = new AST; AST* my_2 = new AST;
   my_1->setToken(INT); strcpy(my_1->getText(), "0");
   my_2->setToken(COMMA);strcpy(my_2->getText(), ",");
   #gen_logical_expr = (SORASTBase*) #(fc, id, lp, arg1, col, arg2,
   co2, my_1, my_2, arg3, rp);
 * _ASE_END_
 */
```

Figure 4-6: Adding a third argument to a function call.

4.2.3 Data type change specifications

In the design of our change specification language, three keywords can be used to help specify data type changes. Table 4-3 shows the list of keywords for data type change specifications and what they mean..

Table 4-3: Keywords for data type change specifications

Keyword	**Meaning**
DT_OLD_STRUCT	the struct identifier to be matched
DT_OLD_MEMBER	the member of the struct to be replaced
DT_NEW_MEMBER	the new member name

Figure 4-7 shows an example of a data type change. An old member of the data type “old_struct” is to be replaced with the name, “new_member”.

```
// struct old_struct {
//     int old_member;
// };
struct old_struct {
    int new_member;
};
// _ASE_DT_BEGIN_
// DT_OLD_STRUCT = old_struct
// DT_OLD_MEMBER = old_member
// DT_NEW_MEMBER = new_member
// _ASE_DT_END_
//
```

Figure 4-7: A data type change specification example.

4.2.4 Global variable change specifications

In the design of our change specification language, two keywords are used to help specify renaming global variables. Table 4-5 shows the list of keywords for global variable change specifications and what they mean.

Table 4-4: Keywords for global variable change specifications

Keyword	Meaning
GV_OLDNAME	Old name for the global variable.
GV_NEWNAME	New name for the global variable, to replace the old name.

Figure 4-9 shows an example of a global variable change. An old name “gv_old” is to be replaced with a new name “gv_new”.

4.2.5 Header file change specifications

In the design of our change specification language, two keywords are used to help specify renaming and adding header files. Table 4-5 shows the list of keywords for include header change specifications and their meanings.

```
// global var
// int gv_old;
int gv_new;
//
// _ASE_GV_BEGIN_
// GV_OLDNAME = gv_old
// GV_NEWNAME = gv_new
// _ASE_GV_END_
//
```

Figure 4-8: A global variable change specification

Table 4-5: Keywords for header file change specifications

Keyword	Meaning
IH_OLDNAME	Old name for the header file.
IH_NEWNAME	New name for the header file, to replace the old name.

Figure 4-9 shows an example of an include header file change. An old name "lib_rename.h" is to be replaced with a new name "lib_newname.h".

```
//
// _ASE_IH_BEGIN_
// IH_OLDNAME = \"lib_rename.h\"
// IH_NEWNAME = \"lib_newname.h\"
// _ASE_IH_END_
//
```

Figure 4-9: A header file change specification

4.3 Discussion

Although it is probably easier to use an existing change specification language, a library maintainer is not restricted to use the language we chose. We establish the approach to design a library interface change specification language. A library maintainer may instead use a more appropriate language for the programming language or specific changes in which he is interested. Nevertheless, we have laid down the foundation and shown an

example for a change specification language based on tree pattern matching and manipulation.

A library maintainer may specify the grammar and transformation specifications for the programming language plus a set of supporting functions for type checking and scope analysis to implement a different change specification language. In fact, a library maintainer may release both grammar and transformation specifications with a new library release to handle changes that have not been designed for it in an earlier change specification language.

In a library development environment, test suites are sometimes available to test a new release of the library. Our approach to interface changes complements this testing effort because a library maintainer can use the same test suites to test the correctness of his change specifications.

4.4 Chapter summary

This chapter introduced our design of the change specification language. The change specifications are based on tree pattern matching and modification. The tree pattern specifications are based on PCCTS and Sorcerer. Examples were given to show how to use the change specification language.

Chapter 5

The Design of Our System

> "People are smart but careless. Keep them smart. Help them to avoid careless mistakes." - anonymous

This chapter describes the our design to handle change specification (see Section 3.1.1) and change propagation (see Section 3.1.2). We use a set of tools (see Section 3.2.2) to handle interface changes written in a change specification language (see Section 4.2).

One approach generates tools based on the library modifications, and distributes those tools along with the updated libraries. When an application maintainer decides to import or use the updated library, the supplied tools can be run to update the application. To generate the tools, the library maintainer is required to provide, in addition to the actual library updates, a description of how affected sites in the application must be updated to properly use the updated library. Figure 3-1 (repeated here as Figure 5-1 for convenience) shows a more detailed view, which we describe below.

We require a library maintainer to specify interface changes and how existing application code can be transformed to adapt to those changes. These are specified in the interface files (e.g. header files for C/C++ or specification files for Ada [USDoD 1980]) that are distributed as part of a library. The specification of the library interface changes as well as the specification for the transformation are then extracted by a process on the application maintainers' side. The process first checks for file dependencies on the interface changes and then adds the library change specification to a standard grammar specification file and a standard transformation specification file for the implementation language.

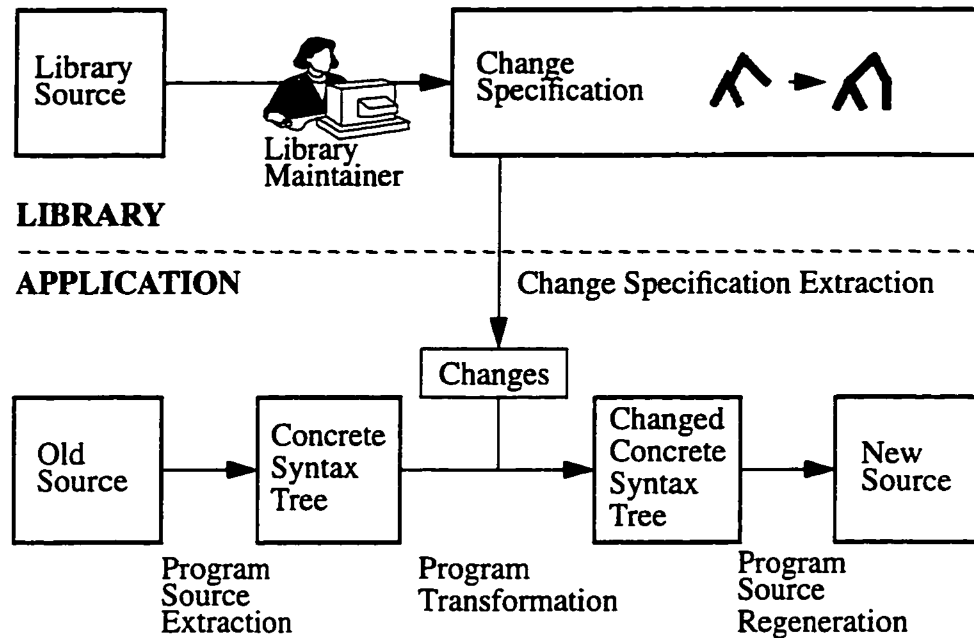


Figure 5-1: Two phases of the change process

The grammar and transformation specification files are then used to build a parser and a syntax tree transformer. The parser constructs a syntax tree from the application program and performs token level code transformations. The syntax tree transformer performs syntax tree transformation as stated in the transformation specification. A new application program, which has been adapted to the new library, is produced at the end.

Although the change specification is released as part of the library code, the application maintainers make the final decision of whether to update their source code or not. This is usually done by adding an extra target in the Makefile. An example of this target is given in Figure 5-2.

In Figure 5-2, an application maintainer uses the upgrade_source program to update his program sources specified by the definition of $(SRC). The update process also depends

```
ASE_BIN = /homes/gws/kingsum/thesis/ase/bin
GRAMMAR_FILE = $(ASE_PATH)/grammar.g
TRANSFORM_FILE = $(ASE_PATH)/transformer.sor

update: $(SRC)
$(ASE_BIN)/upgrade_source $(GRAMMAR_FILE) \
$(TRANSFORM_FILE) CFLAGS= $(CFLAGS) SRC= $(SRC)
```

Figure 5-2: A Makefile example showing the new update target.

on a standard grammar file for this programming language and a standard transformation file to help manipulate the syntax trees.

Our work is similar to work in program restructuring, except that it allows temporal and geographical distance between the library and application maintainers. This also introduces significant technical problems, e.g. change specification, representation and propagation, which we must address.

5.1 Overview of our design

The overview of our design is captured in the data flow diagram in Figure 5-3.

In our model, the header files from a library release are the only files that are of concern to the application maintainers during compilation while the rest of the library release may be used for linking. Since the library maintainer specifies the changes in the header files (see Chapter 3), the first step of our design is to provide a program called the change specification extractor, to extract the change specification from the header files. But the change specification extractor needs information from the application code to determine if the change should really be applied, thus, it extracts the file dependency information from the application code. The file dependency is required to check if a particular header file is actually included and thus whether the change within that file should be applied to the application code. To help with this process, another program called the file dependency

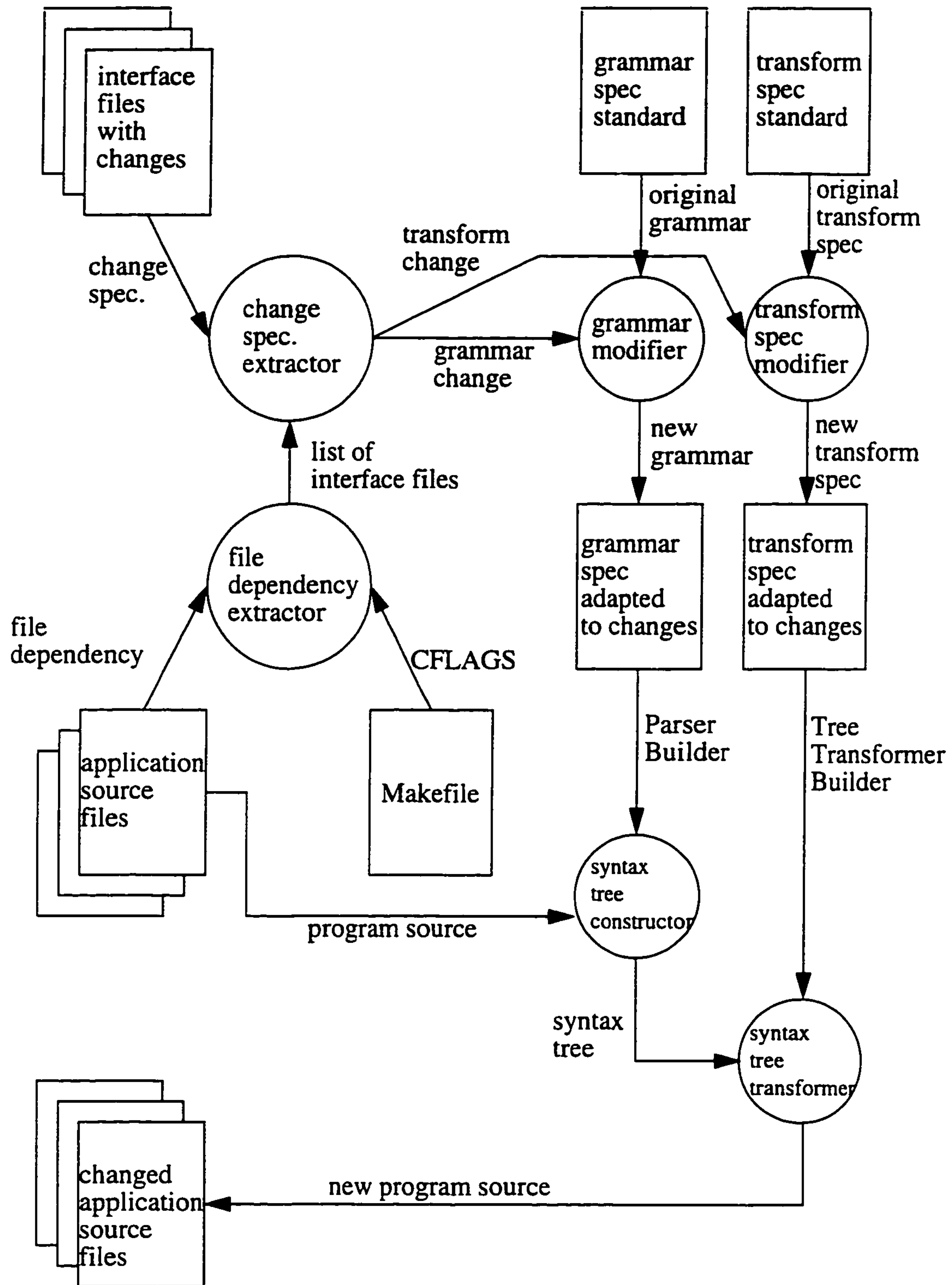


Figure 5-3: The data flow diagram for our design

extractor is provided. Together the change specification extractor and the file dependency extractor perform the change specification extraction in Figure 5-1.

There are two outputs from the change specification extractor. They are the grammar change specification and the syntax tree transformation change specification. We chose to design our extractor in this way to maintain the feasibility in handling changes. The grammar change specification can be used by a grammar modifier to make token level changes while the syntax tree transformation change specification can be used by a syntax tree transformation specification modifier to make syntax tree transformations. The outputs of these two modifiers are a new program called the syntax tree constructor and another program called the syntax tree transformer.

The syntax tree constructor performs the program source extraction in Figure 5-1. It constructs the syntax tree from the application code by using the grammar change specification and provides the syntax tree to the syntax tree transformer. The syntax tree transformer performs the program transformation in Figure 5-1. It transforms the syntax tree by applying change specifications to it. The new syntax tree is then unparsed and a new source code is regenerated while maintaining the look and feel of the application program.

Here is the description of our process from the viewpoints of the library maintainer and the application maintainer. In Figure 5-3, a library maintainer revises a library with some interface changes. In addition to recording the change in the interface, the library maintainer writes a transformation rule that describes how to adapt the application code to work properly in the face of the interface change. When the application maintainer decides to import the updated library, its original source is extracted by a tool to construct some internal representation, i.e., a syntax tree. The internal representation is then modified by a change propagation tool into another internal representation, from which a new source code is generated.

5.2 Detailed design

For our prototype implementation, we use readily available language processing tools, such as the Purdue Compiler Construction Tool Set (PCCTS) [Parr 1995] and Sorcerer [Parr 1994], text processing tools, such as Perl [Wall & Schwartz 1990], and a collection of our own C++ [Stroustrup 1991] programs that work with PCCTS and Sorcerer.

The file dependency extractor, the change specification extractor, the grammar modifier, and the syntax tree transformation specification modifier are all implemented in Perl. Perl scripts are very convenient here because they are very portable and versatile. The change specification is intentionally written in terms of patterns that are similar to PCCTS/Sorcerer. Little conversion is needed to make them usable by the other tools in our approach. In particular, Sorcerer supports tree-to-tree transformations (see Section 4.2), and the language we use to describe needed application updates is taken almost directly from Sorcerer.

The source syntax tree constructor is implemented using PCCTS and a set of C++ programs that coordinate with the base PCCTS system to perform semantic checking such as scope analysis and type checking. PCCTS is similar to a more popular tool set, Lex [Lesk & Schmidt 1975] and YACC [Johnson 1975], but we are using PCCTS for the following reasons:

1. PCCTS generates C++ code,
2. PCCTS coordinates very well with Sorcerer, a tree parser transformer, and
3. PCCTS provides good error messages.

The syntax tree transformer is implemented with the help of Sorcerer and some added C++ programs to help with activities such as unparsing.

Changes at the token level are done while running the PCCTS generated code, but changes that depend on a sequence of tokens, i.e., tree patterns, are better carried out using Sorcerer.

Next, we describe the tools in detail.

5.2.1 File dependency extractor

The file dependency extractor extracts the header files that are actually used in a source file. The determination of the included files can be quite complicated. For example, it may be complicated by the use of environmental variables, compilation options and file systems that may have soft links. But, a compiler preprocessor will pick up the included files which are then used by the file dependency extractor. For example, in the case of handling C/C++ programs using the GNU C++ compiler, we can construct a file dependency extractor which uses a g++ command line option, "-E", to preprocess a C/C++ source. The extractor can determine the included files based on regular expression pattern matching.

5.2.2 Change specification extractor

The change specification extractor extracts the specification from the list of included files as deduced from the file dependency extractor. Again, we construct the change specification extractor using Perl. This extractor extracts change specifications from the included files. The actual extraction process is carried out by a Perl routine that examines all comments and, based on its discovery, other routines that extract changes according to the classification of function, data type, global variable or header file changes.

The Perl script that looks for change specifications basically searches for the change encapsulation keywords and delegates the actual change extractions to other routines. The routine that extracts function changes searches for keywords for function changes and

stores the change specifications in some arrays for later use. Similarly, the routine that extracts data changes searches for keywords for data type changes and stores the change specifications. Similarly, the routine that extracts global variable changes searches for keywords for global variable changes and stores the change specifications.

5.2.3 Grammar modifier

The grammar modifier uses the change specifications that are obtained from the change specification extractor and applies them to a standard grammar to produce a new grammar with additional changes to adapt to the new interface.

The modified standard grammar used in our approach differs from a grammar for compilation in a few areas. First, our grammar must keep the syntax for preprocessing directives, which are often preprocessed away before a compiler sees them. Second, our grammar cannot throw away comments or spacings. Third, our grammar has a number of locations where change specification code can be inserted. An entire grammar is provided in Appendix A.1.

The grammar modifier processes the change specifications and inserts them into various places in the grammar file. In our design, all insertion locations are marked with "_ASE_". Our grammar modifier looks for insertion points in the standard grammar file and inserts change specifications in those places. It locates the insertion points by matching keywords, e.g., _ASE_NAMES_, _ASE_TOKENS, etc. The code that is inserted depends on the location. The insertion at "_ASE_NAMES" only requires function names. The insertion at "_ASE_TOKENS_" requires token names which is the concatenation of a tag (FCALL), a function name and number of arguments for the function call. The insertion at "_ASE_FCALL_" requires the matching pattern for a function call. The insertion at "_ASE_GV_BEGIN_" requires the matching pattern and the transformation for global

variable changes while the insertion at "_ASE_DT_BEGIN_" requires the matching pattern and the transformation for data type changes.

5.2.4 Transformer modifier

The transformer modifier works similarly to the grammar modifier. It takes a standard tree transformation specification and applies change specifications to it.

In our design, a tree transformation specification is written in Sorcerer. Such a specification is still quite similar to a grammar specification but the matching is based on syntax tree patterns instead of lexical structures. A simple tree pattern is given in Figure 5-4. Here, a conditional expression is matched with a tree pattern that has a root of COND_EXPR token and children of some expression, a QUESTION token, another expression, a COLON token and yet another expression.

```
1:   gen_cond_expr:
2:   #(COND_EXPR gen_logical_expr QUESTION gen_logical_expr COLON
gen_cond_expr)
3:   | gen_logical_expr
4:   ;
```

Figure 5-4: A Sorcerer syntax tree pattern example.

A complete standard tree transformation specification is given in Appendix A.2.

The transformer modifier searches for some predefined keyword, e.g., "_ASE_ACTION_" and inserts code for change specifications there.

5.2.5 Syntax tree constructor

The syntax tree constructor is generated from the grammar modified with change specifications using PCCTS. The grammar specifies how concrete syntax trees are to be constructed. The modifications introduced by the grammar modifier allows modifications to nodes of the tree during syntax tree construction.

Further details on PCCTS are given in Appendix B.1.

5.2.6 Syntax tree transformer

The syntax tree transformer is generated from the tree transformation specification modified with change specifications using Sorcerer. The tree transformation specification specifies how syntax trees should be regenerated. Most of the time, no modification is required to act on a syntax tree. In fact, without any change specifications introduced to the tree transformation specification, no change would occur. A syntax tree change specification introduces a tree pattern to match and some manipulation code to apply to the matched pattern (see Section 4.2).

Further details on Sorcerer are given in Appendix B.2.

5.2.7 Program representation details

We choose a concrete syntax tree program representation for reasons that were given earlier in Section 3.1.3. Further, we choose a particular concrete syntax tree structure such that a subtree of interest will have a dummy root node. This node will be marked with a specialized token if the subtree needs to be changed. Figure 5-5 shows an example of a

function call that has this generic structure in our system. Multi-level trees are handled by our system but to simplify the discussion, they are not shown in Figure 5-5.

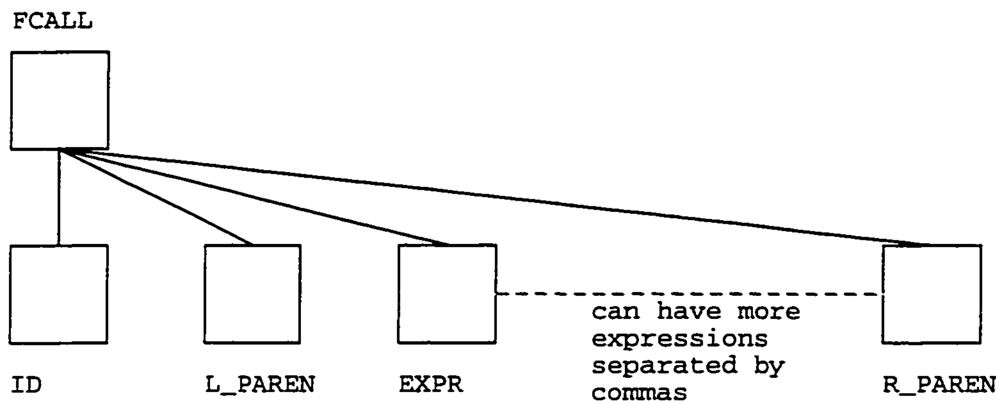


Figure 5-5: A generic function call subtree.

One limitation of Sorcerer is the rules must be written to be deterministic based on the root node of a subtree. This problem is solved by generating specialized tagged nodes as roots of subtrees.

If a function call is to be transformed, it would be tagged with a specialized token instead during the syntax tree construction phase (PCCTS code). This makes it easier for the tree transformer to make transformations because locating a tag is relatively easier. Figure 5-6 shows an example of a tagged function call for transformation. In our design, the tag not only indicates the function name but also the number of arguments. Thus, in our change specification naming system, a function call subtree that is tagged with FCALL_swap_2 should have only 2 arguments in the function call.

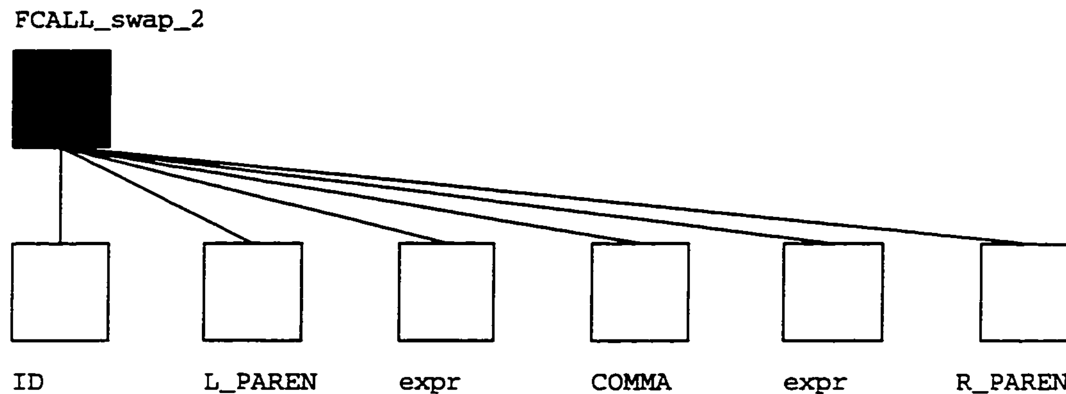


Figure 5-6: A tagged function call subtree.

5.2.8 Program source regenerator

To maintain the integrity of the original program source, a first attempt could be to have each comment or white space as a separation token. This is not a good idea because of the resulting larger number of comment or white space tokens. A slight improvement can be made by joining these two kinds of tokens between each pair of other tokens. This is acceptable in terms of syntax tree construction and unparsing of the tree to regenerate the source code. However, the specification of syntax tree changes would be awkward here because they will be longer to contain the comment and white space tokens, which are sources of distraction. Our last implementation is to have each normal token contain a string of text for the comment and white space. This compromise allows us to have the same simple rules of changing the syntax trees while keeping the comment and white space for unparsing to regenerate the source code with the same look and feel. More importantly, this implementation is still simple.

The regeneration of the program source is done by preorder traversal of the concrete syntax tree, i.e., syntax-directed editor unparsing, with two refinements. First, the text string for comment and white space for a node is printed first. Second, an infix operator is printed with inorder transversal. The first refinement allows non-lexical program elements to be regenerated the same as before. The second refinement achieves the same goal but is required because infix operators are kept as roots of expression subtrees in our syntax tree representations. The choice - to select infix operators as roots of expression subtrees - is quite common because expression evaluations and type checking are simpler to perform.

5.2.9 Upgrade program

The file dependency extractor, the change specification extractor, the grammar modifier and the transformer modifier are actually written as one large Perl script. The script of this upgrade program is provided in Appendix A.3.

5.2.10 Type system

Our type system is implemented in C++ using 13 classes. The type system defines simple types, e.g. int's, float's and char's. It supports complex types such as arrays and pointers. Comparison functions for equality between types are also provided.

5.2.11 Symbol Tables

Our symbol table system is implemented in C++ using 7 classes. The symbol table system provides an important piece of information, scoping, to our system. Each variable is marked with a scope in our symbol table. It is important for global variables where changes should only be applied when a variable has global scope.

5.3 Further remarks on our design

5.3.1 Engineering decisions

In our actual implementation, both the syntax tree constructor and the syntax tree transformer can be used to change the program. This flexibility allows us to experiment with various ways to do program transformation. For example, we can use the syntax tree constructor to tag certain tree nodes (derived from tokens) to make the identification of a tree pattern easier in the later stage. In our experience, node tagging makes tree pattern matching easier to do. It is often used to handle changes related to function calls. However, in this case, there will be an added overhead to the regeneration of the parser in each transformation. A balance between these two concerns is left for future work.

5.3.2 On keeping the look and feel of the application code after upgrade

Not only should the updated application code compile, it should also look exactly the same as before except for those code fragments that are changed. The latter requirement is important for many programmers we talk to because it retains the individual style of the code.

The difference between the look and feel of the code and the program that can compile is the difference between the human artifacts that are introduced to the code and the syntax tree of the code. The human artifacts that are traditionally introduced are to help us understand our code better. In general, there are two kinds of such artifacts that are conventionally dropped by the compiler:

1. the spacing between the code fragments, i.e., a sequence of white space characters,
2. the comments between the code fragments

Another human artifact is the choice of names (including variable names, function names, type names, etc.). But names are conventionally kept by the parser and thus it is not an extra problem here. We have thus reduced the problem to keeping the comments and spacing during the transformation of the application code.

One approach is to introduce comments as separate tokens. Then, programs can be regenerated by these tokens just like other tokens already existed in the grammar. However, this would make both grammar and transformation rules too complicated because the possibility of tree pattern matching needs to include a variable number of such tokens.

Another approach is to introduce comments or spacings before (or after) each token. The problem with this approach is deciding whether the comments or spacings should follow or precede a token. For example, the first character of the source code may be part of a comment or a spacing. Thus, making the comment follow a token requires us to introduce a "begin" token which precedes all other tokens. Although that can be done, it is awkward and not consistent with the grammar of the languages, which normally don't have such a notion. However, the "end of file" token is a generally recognizable token in programming languages because we do need to know when the program ends. Thus, it makes the use of a comment preceding a token more feasible without changing the grammar. The latter approach is taken in our design.

The attachment of comments to tokens can be facilitated if a particular programming scheme is known to be used. For example, if there is always a header comment before each function and smaller comment fragments following blocks of statements, not starting on a new line, we can attach a new line comment to the next token while attaching a non new line comment to the previous token. However, this strategy relies on a particular programming scheme. Further, this strategy seems to be more important for code restructuring, such as meaning preserving restructuring [Griswold 1991], than code adaptation to interface changes. The latter does not restructure the code as extensively as other restruc-

turing tools. Nevertheless, the benefit of this kind of attachment of comments is still not convincingly effective. So, this is not used.

A language transformer, translating a program from one language to another, needs to deal with this translation process more thoroughly. Because language structure can differ significantly between languages, good heuristics of comment placement in the destination language may be necessary.

Limitations in using abstract syntax trees to unparse programs are also reported earlier [van den Brand & Visser 1996]. Recently, there has been movement towards using concrete syntax trees [van den Brand 1996].

5.4 Chapter summary

In this chapter, we presented the design of our approach. We also discussed some design decisions such as two complementary mechanisms for code transformation and mechanisms to regenerate program source.

Chapter 6

Validation and Evaluation

> "If a new method or tool has an *obvious* positive effect on quality and/or productivity, adopt it." - Alan Davis

A first step in validating our thesis was developing a prototype system that supports common interface changes in open system software evolution. A second step was to test the basic concepts by applying the prototype system to examples derived from our case studies, as well as to some of our own examples. Overall, we hope to demonstrate the proof of concept of a working approach in a new research area in software evolution.

We have not produced a production quality tool set, however. One reason is expense. Another reason is the advantages of developing successive prototypes. In the absence of a production quality tool set, it is at best difficult to validate our tools with use real library maintainers and real application maintainers.

We also evaluate our thesis and the effectiveness of our approach in this chapter.

6.1 Validation

Using real library maintainers and application maintainers is an ideal way to validate our approach to supporting interface changes in open system software evolution. One of the advantages of utilizing real users would be to test our approach in real working environments with real interface changes. Instead, however, we use real interface change examples that are derived from existing software systems to simulate real software evolution

without using real users. Thus, although our validation method is not ideal, it nevertheless supports the proof of a new concept in software evolution.

Our approach was validated by running our tool set on the library interface change examples from Chapter 2. The validation process has these goals in mind:

(1) To validate that transformations are accurate.

If the transformations are accurate, then the upgraded applications should produce the same results as before.

(2) To validate that our approach scales well.

The transformations and the propagation of changes should scale well with respect to the size of the library in terms of the number of header files and the dependencies within the header files and the size of the application, which may contain another set of header files in addition to a set of source files.

6.1.1 Design of the validation suites

With those goals in mind, the change examples were designed to run through our tool set with but some minor adjustments. We targeted our tools to a common programming language – C. We added two extensions – function overloading and default arguments – to our targeted language to give some flavors of other languages, e.g. Ada and C++. We did not target our tools to C++ but we outline an approach to extend our tool set to handle C++ in Section 6.3.

There was a lack of suitable C grammars for PCCTS. An old version C grammar in PCCTS was available for some time but it did not have an interface with C++. C++ is the

language with which we built our other parts of the tool. So, not having a C++ interface made the integration of the tools difficult. We could have modified this grammar to make our C grammar as complete as possible. But, first, that would have been a very tedious task. Second, the building of a complete grammar would not provide further support to our thesis. But PCCTS was picked for our implementation for reasons described in Section 5.2. So PCCTS was used and an adequate C grammar was constructed from scratch. To demonstrate our approach, we wrote a grammar for the C language with two major extensions: function overloading and default arguments. We then wrote interface change specifications based on this grammar. The C grammar was considered adequate because it contained good structures for issues of interface definition and usages, and issues regarding the integrity of source code. It also removed unnecessary details and prevented them from distracting from the purpose of this thesis. Recall that the C grammar is different from a C grammar for a compiler in a few different ways – to handle preprocessing directives and comments and spacings. The C grammar is given in Appendix A.1.

Due to the construction of that C grammar, the interface change application code examples were slightly modified to adapt to this grammar, but without the loss of the flavor of C language. The modified C programs can still compile using a common C compiler.

6.1.2 Validation suites based on examples

We validated our approach using samples derived from interface changes categories. To run the update process, we assumed a Makefile existed and all source files for an application were specified by $(SRC) in that Makefile. Further, we assumed that a new target, update, is created to call our tool set to update the application source code. Such a Makefile is easy to write and a skeleton of such a Makefile is provided in Figure 6-1.

In the Makefile shown in Figure 6-1, a new include path for the new library release is defined in line 1 (LIB_INC_PATH). A local directory for working with the upgrade is

```
1:  LIB_INC_PATH = ../include-new
2:
3:  ASE_PATH = ASE
4:  CFLAGS = -I$(LIB_INC_PATH)
5:  ASE_BIN = /homes/gws/kingsum/thesis/ase/bin
6:
7:  GRAMMAR_FILE = $(ASE_PATH)/grammar.g
8:  TRANSFORM_FILE = $(ASE_PATH)/transformer.sor
9:
10: SRC_DIFF = dir.C sdiff.C analyze.C diff3.C util.C
11:
12: update_diff:
13:     cp $(GRAMMAR_FILE).std $(GRAMMAR_FILE)
14:     cp $(TRANSFORM_FILE).std $(TRANSFORM_FILE)
15:     $(ASE_BIN)/upgrade_source $(GRAMMAR_FILE)
             $(TRANSFORM_FILE)CFLAGS= $(CFLAGS) SRC= $(SRC_ALL)
```

Figure 6-1: A Makefile template for our validation suites.

defined in line 3 (ASE). The macro, CFLAGS (line 4), should be used for both the upgrade and the actual compilation of the programs. The tool directory is also defined in line 5 (ASE_BIN). The grammar specification file and the tree transformation specification file are defined in lines 7 and 8. We also need to know all the source files, which are conveniently defined in line 10 (SRC_DIFF). The update target (line 12) is processed by copying a standard grammar file and a standard tree transformation file (lines 13 and 14) and finally, our tool, upgrade_source, process the rest (line 15).

Based on our experience and our case studies of interface changes (see Chapter 2), we created the following interface change examples to validate our approach in response to common interface changes by applying our tools to each interface change example.

(1) Function changes

We first validated our support for function changes. We demonstrated swapping arguments without requiring type checking on the parameters (see Figure 4-1) We do not show the

examples here because we show our support for a bigger problem next. We validated that our approach can support reordering arguments based on parameter type checking. We used our tools to apply the change specification (see Figure 4-5) to a program (see Figure 6-2) and examined the results (see Figure 6-3). Only the swap functions that carried both "int" and "char" arguments were reordered (lines 8-9), as specified in the interface change specification. The other swap functions (lines 4-5) were not changed because they did not meet the parameter type requirement for them to be changed.

```
1:   #include "lib2.h"
2:   int main()
3:   {
4:       int x, y;
5:       swap(x,y);
6:       swap(900,200);
7:       char c;
8:       swap(x,c);
9:       swap(x,'c');
10:  }
```

Figure 6-2: Before reordering arguments based on parameter types.

```
1:   #include "lib2.h"
2:   int main()
3:   {
4:       int x, y;
5:       swap(x,y);
6:       swap(900,200);
7:       char c;
8:       swap(c,x);
9:       swap('c',x);
10:  }
```

Figure 6-3: After reordering arguments based on parameter types

We also demonstrated how to apply our approach to specify an interface change for the example described in Section 1.2. In that example, we needed to insert a third argument to a function call. We wrote such a specification (see Figure 4-6) and applied it to a program

(see Figure 6-4). The program (see Figure 6-5) was changed correctly after the interface change was applied. Notice that only function calls that used 3 arguments were affected (lines 6 and 8). The other function (line 5) which used 2 arguments was not affected.

```
1:  int test3()
2:  {
3:      char* p;
4:      char* q;
5:      assign(p,q);
6:      assign(p,q,3);
7:      int x;
8:      assign(p,q,x);
9:  }
```

Figure 6-4: Before inserting an argument.

```
1:  int test3()
2:  {
3:      char* p;
4:      char* q;
5:      assign(p,q);
6:      assign(p,q,0,3);
7:      int x;
8:      assign(p,q,0,x);
9:  }
```

Figure 6-5: After inserting an argument.

(2) Data type changes

We applied an example of a data type change specification (see Figure 4-7) to a program (see Figure 6-6) and observed that correct changes were applied (see Figure 6-7).

Here, the data type change is to rename a struct "old_struct" field from "old_member" to "new_member". By comparing Figure 6-6 and Figure 6-7, appropriate changes were applied by our tools. Line 5 is changed because "v1" has the type struct "old_struct" while

```
1:  int
2:  use_dt_1()
3:  {
4:      struct old_struct v1;
5:      v1.old_member = 1;
6:      struct old_struct2 v2;
7:      v2.old_member = 2;
8:  }
9:
10: int
11: use_dt_2()
12: {
13:     struct old_struct v3;
14:     v3.old_member = 3;
15: }
16:
17: struct my_struct {
18:     struct old_struct old_member;
19:     int a;
20: };
21:
22: int
23: use_dt_3()
24: {
25:     struct my_struct data;
26:     data.old_member.old_member = 3;
27: }
28:
29: int
30: use_dt_4()
31: {
32:     struct old_struct * pos;
33:     pos->old_member = 3;
34: }
```

Figure 6-6: Before applying data type change.

line 7 is not changed because "v2" has a different type. Line 14 is changed because "v3" also has the "old_struct" type. Lines 17-20 show an interesting case - a new definition of a struct in the application code which uses the old_struct. Line 26 in Figure 6-6 presents an interesting test case and our tools correctly changed the second "old_member" to "new_member" (see Figure 6-7, line 26). Lines 32 to 33 shows that our tool set works well with pointers to data types too.

```
1:   int
2:   use_dt_1()
3:   {
4:        struct old_struct v1;
5:        v1.new_member = 1;
6:        struct old_struct2 v2;
7:        v2.old_member = 2;
8:   }
9:
10:  int
11:  use_dt_2()
12:  {
13:       struct old_struct v3;
14:       v3.new_member = 3;
15:  }
16:
17:  struct my_struct {
18:       struct old_struct old_member;
19:       int a;
20:  };
21:
22:  int
23:  use_dt_3()
24:  {
25:       struct my_struct data;
26:       data.old_member.new_member = 3;
27:  }
28:
29:  int
30:  use_dt_4()
31:  {
32:       struct old_struct * pos;
33:       pos->new_member = 3;
34:  }
```

Figure 6-7: After applying data type change.

(3) Global variable changes

We applied an example of a global variable change specification (see Figure 4-9) to a program (see Figure 6-8) and observed that correct changes were applied (see Figure 6-9).

```
1:  int
2:  use_gv_1()
3:  {
4:    {
5:     int gv_old;
6:     gv_old = 10;
7:    }
8:    gv_old = 20;
9:    {
10:    int gv_old;
11:    gv_old = 40;
12:   }
13: }
14:
15: int
16: use_gv_2()
17: {
18:      int gv_old = 1;
19:      gv_old = 30;
20: }
```

Figure 6-8: Before applying a global variable change.

```
1:  int
2:  use_gv_1()
3:  {
4:    {
5:      int gv_old;
6:      gv_old = 10;
7:    }
8:    gv_new = 20;
9:    {
10:     int gv_old;
11:     gv_old = 40;
12:   }
13: }
14:
15: int
16: use_gv_2()
17: {
18:      int gv_old = 1;
19:      gv_old = 30;
20: }
```

Figure 6-9: After applying a global variable change.

Here, the global variable change is to rename a global variable, "gv_old", to a new name, "gv_new". In Figure 6-8, although the variable, "gv_old", shows up many times. The only real global variable is in line 8. Our tools correctly changed only this variable (see Figure 6-9, line 8).

(4) Header file changes

We also constructed some specifications for header file changes (see Figure 6-10) and applied them to a program (see Figure 6-11) and observed that the changes were applied correctly (see Figure 6-12).

```
1:   from lib_add.h
2:   //
3:   // _ASE_IH_BEGIN_
4:   // IH_OLDNAME = \"lib_add.h\"
5:   // IH_NEWNAME = \"lib_add.h\"\n#include \"lib_add2.h\"
6:   // _ASE_IH_END_
7:   //
8:
9:   from: lib_rename.h
10:  //
11:  // _ASE_IH_BEGIN_
12:  // IH_OLDNAME = \"lib_rename.h\"
13:  // IH_NEWNAME = \"lib_newname.h\"
14:  // _ASE_IH_END_
15:  //
```

Figure 6-10: A specification for include directive changes.

Two include directive changes were specified in two files. In Figure 6-10, lines 3-6 specify the need to insert an additional include file ("lib_add2.h") when "lib_add.h" is used, while lines 11-14 specify the need to rename an include file when "lib_rename.h" is used. It should be renamed to "lib_newname.h".

The change specifications were applied to a simple source (see Figure 6-11) to demonstrate that these include directive changes can be handled by our tool set (see Figure 6-12).

```
1:   #include "lib_add.h"
2:   #include "lib_oldname.h"
```

Figure 6-11: Before include directive changes were applied.

```
1:   #include "lib_add.h"
2:   #include "lib_add2.h"
3:   #include "lib_newname.h"
```

Figure 6-12: After include directive changes were applied.

6.1.3 Validation using changes from diff-2.2

We also validated our approach using a case study for the classification of interface changes. We picked the diff case study (Section 2.8) because it had interface changes which showed up in multiple header files and affected multiple source files. Furthermore, a header file often included another header file in this study. Thus, a chain of include directives across multiple header files needs to be resolved to obtain the correct change specifications. Those features from this case study would be a good challenge to our tools. Also, source code for two versions of this system were also available. So we could examine the interface changes to the code level to apply our tools to the interface changes.

We examined the changes that were made between versions 2.2 and 2.7. We looked at the following header files: diff.h, getopt.h, regex.h and system.h. Among these header files, we examined four interface changes in system.h (see Figure 6-13). One of the interface changes was to rename a data type from "direct" to "dirent" (lines 2-5). Another interface change was to replace "bcopy" with "memcpy" and reordering the first two arguments

```
 1:  //
 2:  // _ASE_SN_BEGIN_
 3:  // SN_OLDNAME = direct
 4:  // SN_NEWNAME = dirent
 5:  // _ASE_SN_END_
 6:
 7:  // _ASE_FCALL_BEGIN_
 8:  // FNAME = bcopy
 9:  // ARGS = 3
10:  // PATTERN = #(fc:FCALL_bcopy_3 id:ID lp:L_PAREN ar1:gen_expr
            co1:COMMA ar2:gen_expr co2:COMMA ar3:gen_expr
            rp:R_PAREN)
11:  // ACTION = id->SetText("memcpy"); #gen_logical_expr =
            (SORASTBase*) #(fc, id, lp, ar2, co1, ar1, co2, ar3,
            rp);
12:  // _ASE_FCALL_END_
13:
14:  // _ASE_FCALL_BEGIN_
15:  // FNAME = bcmp
16:  // ARGS = 3
17:  // PATTERN = #(fc:FCALL_bcmp_3 id:ID lp:L_PAREN ar1:gen_expr
            co1:COMMA ar2:gen_expr co2:COMMA ar3:gen_expr
            rp:R_PAREN)
18:  // ACTION = id->SetText("memcmp"); #gen_logical_expr =
            (SORASTBase*) #(fc, id, lp, ar1, co1, ar2, co2, ar3,
            rp);
19:  // _ASE_FCALL_END_
20:
21:  // _ASE_FCALL_BEGIN_
22:  // FNAME = index
23:  // ARGS = 2
24:  // PATTERN = #(fc:FCALL_index_2 id:ID lp:L_PAREN ar1:gen_expr
            co1:COMMA ar2:gen_expr rp:R_PAREN)
25:  // ACTION = id->SetText("strstr"); #gen_logical_expr =
            (SORASTBase*) #(fc, id, lp, ar1, co1, ar2, rp);
26:  // _ASE_FCALL_END_
27:
28:  // _ASE_FCALL_BEGIN_
29:  // FNAME = rindex
30:  // ARGS = 2
31:  // PATTERN = #(fc:FCALL_rindex_2 id:ID lp:L_PAREN ar1:gen_expr
            co1:COMMA ar2:gen_expr rp:R_PAREN)
32:  // ACTION = id->SetText("strrchr"); #gen_logical_expr =
            (SORASTBase*) #(fc, id, lp, ar1, co1, ar2, rp);
33:  // _ASE_FCALL_END_
34:  //
```

Figure 6-13: Interface change specifications in system.h.

```
1:  //
2:  // _ASE_FCALL_BEGIN_
3:  // FNAME = Is_space
4:  // ARGS = 1
5:  // PATTERN = #(fc:FCALL_Is_space_1 id:ID lp:L_PAREN
          ar1:gen_expr rp:R_PAREN)
6:  // ACTION = id->SetText("ISSPACE"); #gen_logical_expr =
          (SORASTBase*) #(fc, id, lp, ar1, rp);
7:  // _ASE_FCALL_END_
8:  //
```

Figure 6-14: Interface change specifications in diff.h.

(lines 7-12). The other two interface changes were to rename functions, from "index" to "strchr" (lines 21-26) and from "rindex" to "strrchr" (lines 28-33). We also examined one interface change in diff.h (see Figure 6-14), which was to rename a function from "Is_space" to "ISSPACE" (lines 2-7).

We examined 5 source files: dir.C, sdiff.C, analyze.C, diff3.C and util.C, which could be viewed as application sources.

The source file, dir.C, was determined by our tool to depend on diff.h, regex.h and system.h. Thus, our tool extracted the interface changes from both header files that contained interface changes and applied all 5 interface changes to dir.C. Only 2 uses from dir.C were found by our tool that needed to be changed: "struct direct" was replaced by "struct dirent" and "bcopy" was replaced by "memcpy" with reordering of the first 2 parameters. The original and the final dir.C are shown in Figure 6-15 and Figure 6-16 respectively.

The source file, sdiff.C, was determined by our tool to depend on getopt.h and system.h. Since there were no interface changes in getopt.h, only interface changes from system.h might need to be applied to this source file. The key fragments of code before the changes were applied and after the changes were applied are shown in Figure 6-17 and Figure 6-18 respectively.

```
1:   int
2:   dir_sort (struct file_data *dir,struct dirdata *dirdata)
3:   {
4:     struct dircnt *next;
5:     ...
6:        bcopy (d_name,data + data_used,d_size);
7:     ...
8:   }
```

Figure 6-15: Before interface changes were applied to dir.C.

```
1:   int
2:   dir_sort (struct file_data *dir,struct dirdata *dirdata)
3:   {
4:     struct dirent *next;
5:     ...
6:        memcpy (data + data_used,d_name,d_size);
7:     ...
8:   }
```

Figure 6-16: After interface changes were applied to dir.C.

```
1:   char *
2:   expand_name (char *name,int isdir,char *other_name)
3:   {
4:     ...
5:     char *p = rindex (other_name, '/');
6:     ...
7:     bcopy (name, r, namelen);
8:     ...
9:     bcopy (base, r + namelen + 1, baselen + 1);
10:  }
11:
12:  int
13:  interact (struct line_filter *diff,struct line_filter *left,
14:       struct line_filter *right,struct FILE *outfile)
15:  {
16:    ...
17:    char *p = index (diff_help, ',');
18:    ...
19:    char *p = index (diff_help, ',');
20:  }
```

Figure 6-17: Before interface changes were applied to sdiff.C.

```
1:   char *
2:   expand_name (char *name,int isdir,char *other_name)
3:   {
4:     ...
5:     char *p = strrchr (other_name, '/');
6:     ...
7:     memcpy ( r,name, namelen);
8:     ...
9:     memcpy ( r + namelen + 1,base, baselen + 1);
10:  }
11:
12:  int
13:  interact (struct line_filter *diff,struct line_filter *left,
14:          struct line_filter *right,struct FILE *outfile)
15:  {
16:    ...
17:    char *p = strstr (diff_help, ',');
18:    ...
19:    char *p = strstr (diff_help, ',');
20:  }
```

Figure 6-18: After interface changes were applied to sdiff.C.

The source file, analyze.C, was determined by our tool to depend on diff.h, regex.h and system.h. However, it did not use anything that was affected by the interface changes. Thus, although interface changes were extracted from diff.h and system.h, no interface changes could be applied to analyze.C because no interface change pattern match was found from the application source.

The source file, diff3.C, was determined by our tool to depend on getopt.h and system.h. Two incidents of dependence on interface changes were detected and applied to this file. The code fragments that were affected are shown in Figure 6-19 (before interface changes were applied) and Figure 6-20 (after interface changes were applied) respectively.

The source file, util.C, was determined by our tool to depend on diff.h, regex.h and system.h. Three usages that were affected by the interface changes were detected and changed by our tool. The code fragments that were affected are shown in Figure 6-21

```
1:   int
2:  copy_stringlist (char **fromptrs, char **toptrs,
3:       int *fromlengths, int *tolengths, int copynum)
4:  {
5:    ...
6:    { if (*fl != *tl || bcmp (*f, *t, *fl)) return 0; }
7:    ...
8:  }
9:
10:  int
11: compare_line_list (char **list1, char **list2,
12:      int *lengths1, int *lengths2, int nl)
13: {
14:   ...
15:     || bcmp (*l1++, *l2++, *lgths1++))
16:   ...
17: }
```

Figure 6-19: Before interface changes were applied to diff3.C.

```
1:   int
2:  copy_stringlist (char **fromptrs, char **toptrs,
3:       int *fromlengths, int *tolengths, int copynum)
4:  {
5:    ...
6:    { if (*fl != *tl || memcmp (*f, *t, *fl)) return 0; }
7:    ...
8:  }
9:
10:  int
11: compare_line_list (char **list1, char **list2,
12:      int *lengths1, int *lengths2, int nl)
13: {
14:   ...
15:     || memcmp (*l1++, *l2++, *lgths1++))
16:   ...
17: }
```

Figure 6-20: After interface changes were applied to diff3.C.

(before interface changes were applied) and Figure 6-22 (after interface changes were applied) respectively.

```
1:   int
2:   line_cmp (char *s1, char *s2,int len1, int len2)
3:   {
4:     ...
5:       if (len1 == len2 && bcmp (s1, s2, len1) == 0) {
6:     ...
7:     while (Is_space (c1)) { c1 = *t1; }
8:     while (Is_space (c2)) { c2 = *t2; }
9:     ...
10:  }
```

Figure 6-21: Before changes were applied to util.C.

```
1:   int
2:   line_cmp (char *s1, char *s2,int len1, int len2)
3:   {
4:     ...
5:       if (len1 == len2 && memcmp (s1, s2, len1) == 0) {
6:     ...
7:     while (ISSPACE (c1)) { c1 = *t1; }
8:     while (ISSPACE (c2)) { c2 = *t2; }
9:     ...
10:  }
```

Figure 6-22: After changes were applied to util.C.

Our tools also maintained the integrity - the same look and feel of comments and spacings - of the unchanged source code. Figure 6-23 shows a portion of sdiff.C which had many comments and spacings that were not altered in any way during by our program transformation tools.

6.2 Status

We provided a mechanism for library maintainers to pass interface change specifications to application maintainers. Our current system can handle a great number of common interface changes such as function changes, data type changes, global variable changes and header file changes. Table 6-1 summarizes the various classes of changes that can be

```
/* SDIFF -- interactive merge front end to diff
   Copyright (C) 1992 Free Software Foundation, Inc.

You should have received a copy of the GNU General Public License
along with GNU DIFF; see the file COPYING.  If not, write to
the Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139,
USA.  */

/* GNU SDIFF was written by Thomas Lord. */

#include "stdio.h"
#include "ctype.h"
#include "system.h"
#include "signal.h"
#include "getopt.h"

#define SEEK_SET 0

/* Size of chunks read from files which must be parsed into lines.
*/
#define SDIFF_BUFSIZE 65536

/* Default name of the diff program */
#define DIFF_PROGRAM "/usr/bin/diff"

/* Users' editor of nonchoice */
#define DEFAULT_EDITOR "ed"

...
```

Figure 6-23: Our tools maintained the integrity of sdiff.C.

handled using our design and tool set. Table 6-1 follows the classifications of interface changes as shown in Table 2-44.

In Table 6-1, each row consists of 3 items: the interface change, the cost of writing the interface change specification and whether that interface change can be specified and handled in our system. The cost of writing an interface change specification is based on how difficult it is to write the specification from our experience. The cost is considered low if it was straightforward. The cost is high if it requires some use of additional functions from a library. The cost is none if it does not require any work at all and, in this case, no work is

Table 6-1: Status of our implementation

Change	Cost[a]	Handled?
Function prototypes		
Renaming functions	Low	Yes
Adding parameters	Low	Yes
Removing parameters	Low	Yes
Reordering parameters	Low	Yes
Changing default arguments to non default	Low	Yes
Changing non default arguments to default	None	Not req'd
Function semantics		
Changing parameter types	High	Yes
Changing return types	High	Yes
Changing return value meaning	High	Yes
Changing parameter meaning	High	Yes
Function replacements		
Adding functions	None	Not req'd
Replacing functions	Low	Yes
Data types		
Renaming types	Low	Yes
Changing data type semantics	Not impl.	No
Renaming fields in data types	Low	Yes
Adding fields to data types	None	Not req'd
Replacing fields from data types	Low	Yes
Adding types	None	Not req'd
Replacing types	Low	Yes
Global variables		
Renaming global variables	Low	Yes
Adding global variables	None	Not req'd
Replacing global variables	Low	Yes
Header files		
Renaming header files	Low	Yes
Adding header files	None	Not req'd
Replacing header files	(2-step)	Yes
Transfer between header files	Not impl.	No
Protocol change (e.g. Section 2.9.7)	Varies	Varies
Language syntax (e.g. Section 2.9.8)	Varies	Varies

a. The amount of work required to write the change specification.

required by the tools, either. The difference between a low cost and a high cost change specification is perhaps very small since there may be little cost to writing additional functions.

Function prototype changes are handled well by our system. One particular function prototype change - changing non default arguments to default - does not require our system to handle it at all because existing function calls can be used without any adjustment for that change. For the rest of the function prototype changes, it is quite straight forward to write the interface change specification (see examples in Figure 4-1, Figure 4-5 and Figure 4-6).

In contrast, function semantic changes are more difficult to handle. This should be regarded as a property of the nature of semantic changes, however. When an argument type is changed, a type conversion function may be provided by a library to convert an existing argument type to the required type. Similarly, when an argument value is changed, a value conversion function may be provided by a library to provide the necessary conversions. These conversion functions need to be applied to existing function calls or arguments of existing calls. Thus, the specification of interface changes is not so straightforward but still can be written in our system.

In addition to parts of a function, an entire function can also be replaced by another one. This is usually accompanied by adding new functions to a library. We did not find it difficult to write this specification change. In fact, such an example has already been shown earlier in our validation of one of the case studies (see Figure 6-13).

Like function changes, some data type changes (e.g. adding a filed to a data type or adding a new data type) do not require any change specifications at all. Changes that require adaptation of application code are renaming data types and fields in data types or replacing them. Common data type changes can be specified based on syntactical structures (see Figure 4-7 for an example) and handled by our system that way.

Global variables change specifications (see Figure 4-9 for an example) do not appear to be difficult to write and are adequately handled by our system. Our system performs scoping analysis to determine whether a variable is global or not.

Specifying and handling header file changes is similar to global variable changes in general. However, because our system is based on extracting interface changes through include directives, removing a header file causes a fundamental problem. This problem can be easily solved by a two step process. In the first step, the header file to be removed should be distributed along with another header file to be used. In this step, the change specification specifies that the other header file should be used. In the second step of the distribution, a change specification that removes the include directive for itself will be sufficient.

We did not specifically test our system for protocol and language changes. In the case of a protocol change, if the change can be specified based on the identification of a particular construct in the program, then the change can be handled as other changes. If the change is more complicated, e.g., a change that depends on multiple code fragments, then our approach cannot handle it. We will elaborate more on the limitations of our approach in the next section. In the case of a language change, we have a similar problem. However, the MAKE language change in Section 2.2.1 is simple to solve by our system or a lexical based system.

Our system adequately handled many common interface changes. However, we do not claim to solve all conceptually possible interface change problems. For example, some interface changes, such as changing data type semantics, are not completely handled yet. One of the problems, for example, in assigning a value to a changed data type attribute, is the need to pass additional information from the left hand side of an assignment syntax tree node to the right hand side. We did not do that because we did not observe this kind of possible change in the case studies (see Table 2-44).

6.3 Evaluation

6.3.1 Limitations and extensions

Our approach is based on interface changes that are stored in the interface specification files. There are, however, some classes of library updates that our current implementation cannot handle.

For example, we cannot handle changes that are dependent on simultaneously changing two non-local places in the source code. Such an example was given earlier in Section 2.2.1 (special protocol requirement). In that example, a "declare" macro needs to be removed if a "define" macro is used later. A change dependency on multiple locations of the source code is not easy to handle but fortunately, this does not happen often. When this happens, a conservative approach is to alert the application maintainers whenever one of the two functions is used and let the application maintainers decide what to do.

A specific example of this problem occurs when a change is made to the type of an output parameter of a function. In this case, the declaration of the variable to which the output parameter will be assigned generally has to be changed. But we cannot yet handle a change which is in a different location than the affected use.

In addition, there are a number of language specific features that our implementation has not been able to handle. A common problem among these language features is aliasing. For example, the alias of a variable to another variable, or when change is based on a function pointer that may point to any function in the program, only some of which may in fact be affected by a semantic change specification.

We also assume that results of function calls are independent of the order of execution of their parameters. In other words, no undesirable side effects will hinder reordering param-

eters of function calls which may yield a different result. This assumption falls in line with the common programming practice that a programmer should not write the code in such a way that the result of a function call depends on the actual order of parameter evaluation. Further, common programming languages (e.g., C, Ada, C++) do not specify the order of parameter evaluations, either.

In our implementation of the system, the grammar is written for a ANSI C-like grammar that requires parameter types for function prototypes. In contrast to K&R C, this grammar provides sufficient information about function prototypes so that our approach can pick a function call that matches one of a set of overloaded functions.

In object-oriented programming languages, there is a new dimension for the effect of interface changes. The propagation of interface changes may move along a class inheritance hierarchy too. For example, in C++ [Stroustrup 1991], changing a data member of a class may have effect on both the clients of the class as well as subclasses of the class. The effect of a protected data member change is like changing a global variable, only this time it can affect not only itself but also its subclasses. Similarly, a member function change may see its effect propagate to many other areas. In order to adapt our approach to deal with interface changes along a class hierarchy, one probable solution is to extend our system to use an appropriate grammar for that language and a tree transformation specification. A new typing system is also required to deduce the type of an object and thus the effects of member function or data member changes. However, like other pointer problems, polymorphic functions that are resolved at run time cause additional problems here. These are interesting issues for future work. However, even in these cases, the application maintainers are still no worse off than, if not better than, before.

There is another context of interface changes. Since libraries often have multiple releases, the true dependency on a header file is not only the use of a header file from a library, but also the header file of a particular release. Our approach can easily be adapted to libraries

with multiple releases. In this case, on the library side, each header file of a library release will be identified by a unique version number. It is also identified by the older version number from which it is changed. The requirement for a version number in a header file can be easily met by using some revision configuration system, e.g. RCS [Tichy 1985]. On the application side, for each header file included by a program source, we can record the version number of the header file in the beginning of the source. When a header file is included, and when the previous version number of a header file is matched with the version number that is being used by a source, a change specification can then be propagated to the source, and the version number that is currently used can be changed to the newer version number.

This simple solution works well when an application upgrades with each library release, which is, perhaps, uncommon. In order to allow an application to upgrade from some older versions of a header file, we can introduce an additional keyword to the change specification language. On the library side, a new keyword, appropriately named VERSION, can be used to associate a change specification with a particular version of a header file from a library release. On the application side, only change specifications with versions that match what is currently being used by an application source will be propagated to it. One drawback of this approach is that library maintainers may need to write multiple change specifications for multiple previous versions. We may relax the matching criteria from matching a unique version to a set of versions. Still, deducing the effect of a chain of change specifications across multiple versions lies in the hand of the library maintainer. The need to develop a tool to combine the effects of chaining multiple change specifications for different versions of the same change into one may be justified when that is found to be common.

6.3.2 Other comments

We also evaluate our approach according to the criteria (see Section 1.4) for an acceptable solution to the problem of handling interface changes in open system software evolution.

Our approach clearly relieves application maintainers from having to identify whether or not the changes to the library require changes to the application. Our tool set is responsible for extracting and propagation changes, if there are any, and if the changes affect the application code, while application maintainers only need to add a few rules in the Makefile.

Our approach also relieves application maintainers from having to identify and manually update each location in the application that must change in response to library changes. When a change may be required, our tool set will determine if a match of a code fragment is found by syntactic tree pattern matching as well as semantic condition matching. When a match is found, it proceeds to change the code fragment as instructed by the change specification.

Furthermore, our approach maintains the look and feel of the application source around the changes needed for the library upgrade. It uses concrete syntax trees with annotations representing comments and spacings to regenerate the source code except for code fragments that changes have been made.

We also evaluate it from other perspectives.

(1) Design of our change specification language

The change specification language is perhaps the most noticeable user interface. Library maintainers write and read change specifications while application maintainers read them

when they want to examine the interface changes from a library release. Thus, the major concerns regarding the design of the change specification language are (1) the expressiveness of the language, (2) how easy it is to write the change specifications and (3) how easy it is to read them.

The change specification language is expressive enough to cover common interface changes. Most major interface changes such as function changes, data type changes, global variable changes and header file changes can be specified (see Section 6.2). Thus, the change specification language is adequate for practical purposes of handling interface changes.

A language is easy to use if it is logical and consistent. Although the syntax of our change specification language is perhaps not beautiful, it is simple and logical. Our change specification language is simple because it is merely a collection of language independent keywords and language dependent expressions. It is logically bound to the search for a particular program pattern, additional semantic condition and how to replace a code fragment that needs to be changed. In fact, we found it fairly easy to modify a change specification example to suit our needs. People who have basic compiler courses should find the underlying principles simple and straightforward. For similar reasons, the change specifications are not difficult to read, either.

Like other languages, it is possible to make mistakes while using it. Fortunately, library maintainers can use existing test suites which can be used to test libraries and their change specifications as well.

While some approaches may require the introduction of certain specific constructs to a language, our approach does not alter any language syntax or features. Instead, our approach only provides a means of handling the languages with the need to handle asyn-

chronous changes. Thus, our change specification language may be adapted to other programming languages in a shorter period of time.

Nevertheless, our choice of the design of the change specification language may not be optimal. However, we establish a working basis for supporting interface changes in open system software evolution.

(2) Uncertainty

As mentioned in Section 6.3, there are some interface changes that cannot be handled by our approach. This problem is a change that depends on two separate locations of the application source. A typical example is a protocol change such as the one described in Section 2.2.2. Following this example, a "create" macro should be replaced with a "define" macro, which can be handled by our approach. However, a "declare" macro should be removed if a "define" macro is not used.That is not yet specifiable. However, we do allow library maintainers to insert text of any length to a token. Thus, a library maintainer may specify that all "declare" macros be replaced with the same thing plus a warning text message in the source. If the warning message is in the form of a comment, the resulting application source may compile even if it may not work properly. A more conservative library maintainer may introduce some text, though not as comment, simply to break the compilation process. We support both approaches taken by library maintainers using one mechanism – allowing any text to be inserted to a token.

(3) Change conflicts

Possible change conflicts may occur when an application uses two libraries which are changed in some conflicting manner. For example, both libraries may introduce the same function prototype to replace some existing ones. This is a subset of the problem of library interface conflicts (including change conflicts) and should be addressed separately. If there

are no library interface conflicts, there should be no change conflicts because each change is based on a pattern that uses a feature that comes from one unique header file.

(4) Other potential uses

Our approach is not only useful for library interface changes, it can also be used whenever the detection of usage needs to be done at the client site; suggestions or corrections will be made at that time. Thus it can be used to adapt a program to use a competitor's library interface to make the conversion of programs easier to use a newer and, perhaps, more efficient library.

(5) On keeping old interfaces

Some organizations advocate the maintenance of all interfaces. They may introduce new interfaces that are more general but they will never throw away old interfaces. However, the cost of keeping the old interface is discouraging. The outdated interface kept in the header files is a distraction. It also subjects the users to misuse of the old interface. Keeping the old interface also means keeping larger object code for the old interface and new interface together. This results in longer link time. Also, it is more likely to have link time conflicts with other libraries.

Even in the case that old interfaces are kept, and more generic interfaces are introduced, our prototype system can still be used to give warnings to the client users when the use of the old interface is detected.

(6) Implicit type conversions

Some interface changes that involve semantics can be handled by a language that supports implicit type conversions. C++ [Stroustrup 1991] is a language that supports implicit type conversions. For example, for a function prototype that is changed from foo(A a) to foo(B b), if there is an implicit conversion from type A to type B, then existing calls of foo(a) still work. Thus it appears that explicit type conversions are not necessary for such a language. However, later, if the function prototype is changed again from foo(B b) to foo(C c), and even if implicit conversion is possible from type B to type C, the function call foo(a) will not work for languages (such as C++) that do not extend the search for multiple implicit conversions. Thus, explicit type conversions are important even for languages that support implicit type conversions. Furthermore, explicit type conversions are safer because it makes conversions explicit. Our approach supports the use of explicit type conversions. Explicit conversion function calls can be inserted as token texts in our approach.

(7) Interface changes within an organization

Although our discussion has focused on interface changes across organizations, our approach can also be useful within the same organization. In some programming teams, even though all source code is potentially visible to a library maintainer, a library maintainer may not be able to write to it. On the other hand, even if a library maintainer can write to some application sources, he or she may not be able to find out if those files are actually used at all. Thus, our approach is applicable even to a small programming team where the actual update should be applied only by the application maintainer.

6.4 Chapter summary

In this chapter, we validated our approach by applying interface changes that were summarized in Chapter 2. We examined our status. We also compared the strengths and weaknesses.

Chapter 7

Related Work

Our approach directly tackles a newly identified branch of maintenance activity, i.e., supporting interface changes in open system software evolution. Within the area of software maintenance, our approach has some similarities to program restructuring, program representations, interface changes, change specifications and program transformations. In this chapter, we compare our approach with related work in these contexts.

7.1 Software maintenance

Software maintenance refers to the modifications that are made to a software system after its initial release [Ghezzi et al 1991]. Three common activities of software maintenance are corrective, adaptive and perfective maintenance. Corrective maintenance is the activity that removes defects from a software system. Adaptive maintenance is the activity that helps adapt a software system to a different environment. Perfective maintenance is the activity that enhances a software system with more features. Maintenance of programs in response to interface changes generally falls within the categories of adaptive and perfective maintenance.

A fundamental requirement behind software maintenance activities is software understanding [Chikofsky & Cross 1990]. In our approach, we assume library maintainers understand the library system. However, techniques such as design recovery [Biggerstaff 1989] [Biggerstaff et al 1993] or source model extraction [Murphy & Notkin 1995] may be helpful in the absence of an understanding of a program. Our approach does not attempt to help library maintainers understand the library systems.

7.2 Program restructuring

One of the activities that is often done before making semantic changes to a program is to restructure it. Restructuring is often necessary due to the natural degradation of software structure after a series of modifications [Lehman 1980].

The meaning preserving program restructuring [Griswold & Notkin 1993] [Griswold 1991] and the refactoring [Johnson & Opdyke 1993] [Opdyke 1992] approaches focus on the tedious task of maintaining the meaning of a program while its structure is being altered by the maintainer. There are two major differences between them and our work. First, the entire program is presumed to be visible and changeable by the restructuring tools while parts of the program are either not visible or not changeable in our proposal. Second, some structural changes handled by restructuring or refactoring need not be dealt with by our change adaptation tool because they do not pertain to interface changes. Renaming local variables is one such example.

Both restructuring and refactoring approaches establish a set of primitive transformations upon which complex transformations can be built. Orion [Banerjee et at 1987] is an earlier approach that works along the same concept. The Orion data model [Kim 1990] introduced invariants of transformations in addition to primitive transformations. An object-preserving approach [Bergstein 1991] which is based on the Demeter data model [Lieberherr et al 1991] is similar to the Orion approach. In their work, after a reorganization of a program, the meaning of a program is preserved by object-preserving transformations. We do not use a set of primitive transformations and do not attempt to prove program correctness in terms of maintaining some program invariants. We do not limit the choices of transformations but it relies on adequate testing of library change specifications by library maintainers.

There is another restructuring approach [Casais 1992] that also does not require the programmer to select from a set of transformations. In that approach, the goal of a set of transformations is predetermined and a sequence of transformations is automatically selected by that approach to achieve the goal. In our approach, we do not have a fixed goal for which a sequence of operations may be attempted. Instead, we require library maintainers to specify what changes to apply.

Most of these restructuring approaches require the accessibility of all source code. When a transformation is applied, all source code can be changed if necessary. Using our terminology, the library maintainer and the application maintainer behave (logically) as the same person in other approaches. In contrast, our approach attempts to help multi-person maintenance activities.

7.3 Program representations

Abstract syntax trees [Aho et al 1986, p.287] are a common program representation used by compilers. An abstract syntax tree, directly corresponding to the structure of a program, is often constructed when the program is parsed. In our approach, we do not use abstract syntax trees because the type of abstract syntax trees used in compilers do not contain all the syntactic constructs that we need. In our approach, we need to regenerate the program text without altering its integrity. Thus we need to carry additional textual information that is generally ignored by compilers. We use concrete syntax trees to carry more textual information such as semicolons and parentheses.

Program dependency graphs are good program representations for slicing, differencing and integrating programs [Horwitz & Reps 1992]. However, they are expensive program representations [Morgenthaler & Griswold 1995]. They are important for complete program analysis and program restructuring [Griswold & Notkin 1995]. They could be applied to our approach to add general program restructuring mechanisms. However, our

interface change dependencies use syntax tree pattern matching of interface changes. Thus, program dependency graphs do not offer additional benefits to our approach.

7.4 Interface changes

The Eiffel programming language supports some interface changes directly through the notion of obsolete functions [Meyer 1992]. As Meyer put it:

> "By declaring a feature (function) as obsolete, you (library maintainers) keep it usable exactly as it was, while alerting its users (application maintainers) to the existence of a better version. This provides a graceful way to phase out a routine while remaining friends with the developers of client systems." [Meyer 1992, p.73]

The "obsolete" feature is part of the Eiffel language. It would be hard to adapt this feature to another programming language without changing the other language. Our approach can be applied to many languages. In Eiffel, only functions may be declared "obsolete". In our approach, we looked at the interface changes as not only function changes, but also data type changes and global variable changes. The "obsolete" feature, when used for a function, only generates a warning message. In our approach, we support the actual adaptation of the code as well. Also, our approach supports the replacement of a function rather than simply making the old function "obsolete".

A programming practice that does not require altering a programming language has been employed by some library maintainers [Glew 1996]. In this practice, when the prototype of a function is changed, a redefinition of the old function prototype is released along with the new function prototype in a header file. Again, using the interface change example from Section 1.2, the old function prototype would be:

```
int assign(char* dst, char* src, int len);
```

The new function prototype would be:

```
int assign2(char* dst, char* src, int start, int len);
```

Using this practice, we can help enforce a commitment by application maintainers by releasing this redefinition along with the new function prototype:

```
int assign(char* dst, char* src, int len)
{
    assign2(dst, src, 0, len);
}
```

Note that in this example, the new function is carefully named assign2 instead of assign to avoid name clashing. Similar to the Eiffel "obsolete" keyword, this approach gives application maintainers a number of chances to change their own code to use the new function prototype before the old version is no longer supported. In the first phase, the application maintainer is encouraged, but not required, to adapt the application source for this prototype change. In a later release, the library maintainer may replace the definition of "assign" function with something that causes a compilation error messages or prints a run time error message, to force the application maintainer to change the source code to use the newer prototype. This can be done by, for example, a macro redefinition of "assign" with some incorrect syntax. In the third phase, the old prototype is simply removed.

This approach requires little additional tools to help solve some of the library interface change problems. However, in common programming languages, not every interface elements can be redefined using this mechanism. For example, this mechanism cannot deal with preprocessor directives or data type changes (e.g. a C struct field). Further, this mechanism does not offer any help on the application side to adapt the source to a new interface. It does, however, hide the interface change temporarily during the first phase of the release of the new interface.

A derivative of program dependency graphs, called attributed program dependency graphs [Al_Zoubi & Prakash 1991], have been used to analyze the effects of software changes.

The attributes of these program dependency graphs are procedures, functions, types and variables. While this approach has shown some success with Pascal programs, it requires that entire programs are accessible. It is not clear, however, how well the approach can be used to analyze interface changes when application sources are not available.

7.5 Change specifications

Our approach requires the library maintainer to write the change specifications for a library release. Lexical difference between two versions of a library is not sufficient. Recently, Jackson and Ladd have described a new tool that shows some promise in determining the semantic difference between two functions [Jackson & Ladd 1994]. The tool takes two versions of a function and then summarizes the differences between them in terms of the observable input-output behavior of the function by examining the dependencies of the values of the input and output parameters. Although the tool is promising, some problems such as false positives and false negatives still need to be addressed. Nevertheless, this kind of tool can complement our approach to test for absence of semantic differences. In the case of the presence of semantic differences, however, this tool does not offer immediate insights on what the actual change specification should be. Furthermore, although this approach compares two versions of a function of the same name adequately, it is rather limited in what it can do to discover an interface change that requires replacing a function with a different interface.

7.6 Matching code fragments

Our approach is based on matching application code fragments as syntax trees with the changes specified by the library maintainer. Code fragments can be matched lexically or syntactically. Many tools are available to match code fragments lexically. Grep is often used by programmers to search files for regular expressions in source lines [Aho 1980].

Awk [Aho et al 1979] introduces programmer-defined actions that can be applied to matched expressions. Mawk [Brenan 1992] extends Awk by supporting patterns as data. Perl [Wall & Schwartz 1990] essentially combines the benefits of Awk, C and Sed into a single language.

Matching code fragments is an important task in program understanding. Lightweight Source Model Extraction (LSME) [Murphy & Notkin 1995] builds high-level models from large software systems and supports high-level decision making. LSME constructs reasonably good approximations of the source models by matching regular expressions. Tawk [Griswold et al 1996] combines lightweight lexical processing with partial syntax tree matching. Both techniques sacrifice precision for speed.

Complete syntax trees are more precise representations of programs. When accuracy is more important than speed, syntax trees are a good candidate for program representation. In the problem of handling interface changes, speed is not a major concern because the performance of the upgrade process will still be much faster than the manual and laborious procedure carried out by the application maintainer.

The construction of syntax trees is commonly based on context free grammars. Yet another compiler-compiler (Yacc) [Johnson 1975] is a tool that supports parsing of a program based on a user-defined grammar. Purdue Compiler Construction Tool Set (PCCTS) [Parr 1995] combines lexical definitions, grammar specifications and syntax tree transformations (Sorcerer [Parr 1994]) in one tool. Both Perl and Awk divide an input line into fields using a user-definable field separator; each field can be addressed by its position (an integer). PCCTS adds the improvement that the matched patterns are named by the user instead of depending on the positions. In our experience, this improvement greatly reduces the work of maintaining PCCTS grammars.

PCCTS is used in our approach for language specifications and syntax tree transformations. The syntax tree patterns of our specification language mimics the syntax of PCCTS.

Another system that supports both constructing and modification of parse trees is A* [Ladd & Ramming 1995]. In this respect, A* is almost the same as PCCTS. However, A* is perhaps slightly more difficult to use due to the absence of variable naming and rule naming.

7.7 Program transformations for interface changes

In the industry, some sort of script is sometimes provided for people that deal with interface changes, e.g. Borland's ObjectWindows Library converter [Borland 1993b] and Microsoft's migrate [Microsoft 1994]. In general, they are based on lexical analysis and they make more assumptions about the source. As a result, they make themselves somewhat restrictive and may work in a predetermined domain. While we have not examined every last detail of these systems, it seems that these transformation scripts could, in fact, be generated by a more general mechanism such as ours.

The goal of Borland's ObjectWindows Converter is to help their users upgrade their code to adapt from version 1.0 of ObjectWindows to version 2.0. Its focus is on Microsoft Windows programming. It provides a best effort to change some of the users's code. Besides making changes, it also inserts comments in application code when it encounters a questionable construct. To use this tool, you need to pick a source file and run the tool with the same compiler options. In contrast, our approach analyzes the update requirements based on a normal build process, without specifying each individual file for updating.

Similar to Borland's ObjectWindows Converter, Microsoft's Migrate also focuses on Microsoft Windows programming. It is also based on textual substitution. It makes

implicit assumptions about some programming style in Visual C++, only for Windows programming. For example, some Windows header files may be assumed to be included.

7.8 Source reconstruction

Our application source code regeneration method is similar to the DIANA (Descriptive Intermediate Attributed Notations for Ada) approach [Goos et al 1983]. However, while our approach aims to reconstruct a program source exactly the same as before unless there are changes, DIANA only does that to a good approximation. As Goos et al put it,

> "DIANA's design deliberately includes certain normalizations of source programs. These are omissions from DIANA of enough information to reconstruct the original program exactly." [Goos et al 1983, p. 165]

The differences between our approach and the DIANA approach can be traced to a few design decisions. Our approach uses concrete syntax trees while the DIANA approach uses abstract syntax trees. Our approach preserves extra spaces and indentations between lexical tokens but the DIANA approach does not require that. Our approach preserves comments but the DIANA approach does not require that either.

7.9 Chapter summary

In this chapter, we put our research in a broader context by comparing our approach with other related research areas.

Chapter 8

Conclusion

In this chapter, we summarize key contributions of this thesis and suggest areas for future work.

8.1 Contributions

The primary contributions of this thesis are:

- The identification of the problem of handling interface changes as a common asynchronous software evolution problem across organizations.
- The analysis and classification of common interface changes based on real systems.
- The establishment of a model, based on concrete syntax tree pattern matching, for a tool-based solution – a library maintainer bears a one time cost of change specifications which provide savings across applications when upgraded.
- A prototype tool that demonstrates the effectiveness of the tool-based model approach.

Other contributions of this thesis are:

- A layout for a change specification language that serves as the link between the library maintainer and the application maintainer.

- A change propagation mechanism using PCCTS.
- A regeneration of source code with minimal integrity changes.

In summary, this thesis demonstrates that by shifting some of the responsibilities from application maintainers to library maintainers, a simple, novel and effective solution can help solve the problem of handling interface changes in open system software evolution.

8.2 Future work

In our approach, we used PCCTS and Sorcerer to implement change propagation. While it is a good way to produce a prototype system in a reasonable amount of time, other alternatives should also be explored to implement our system. For example, change propagations can be table driven using internal code instead of relying on other tools. However, more studies need to be done to evaluate the alternatives. In additional, our change specification language could also be made more user friendly by field testing.

In addition, some of the processes may not require a complete syntax parsing of the program, e.g., obtaining the "#include" directives in C programs. Separating a program source into smaller chunks of syntax trees may be useful. More work in this area may be fruitful.

Although the primary goal of our approach is to handle program interface changes, its use is not limited to that domain. We believe requirement specification and design changes may also be propagated to the program. However, more understanding in the structures of requirement specification and design may be needed.

Our program source based approach may be extended to handle binary code in the future. That is important, for example, for a linker to detect changes at link time - which may be important for systems using dynamically linked libraries.

8.3 Concluding remarks

The major contributions of this work are the identification of the problem of handling library interface changes in open system software evolution and the establishment of a working model as a general solution to this problem. The working model shows promise to lay a foundation for production quality tools to reduce the cost of upgrading libraries in response to interface changes in software evolution.

Bibliography

[Aho 1980] A. V. Aho, "Pattern matching in strings" in R. V. Book, ed., "Formal Language Theory: Perspectives and Open Problems", pp. 325-347, Academic Press, New York, 1980.

[Aho et al 1979] A. V. Aho, B. W. Kernighan and P. J. Weinberger, "Awk - a pattern scanning and processing language", Software - Practice and Experience, 9(4), pp. 267-280 (April 1979).

[Aho et al 1986] Alfred Aho, Ravi Sethi and Jeffrey D. Ullman, "Compilers Principles, Techniques, and Tools", Addison-Wesley (1986).

[Al_Zoubi & Prakash 1991] R. Al_Zoubi and A. Prakash, "Software Change Analysis Via Attributed Graph-Based Representations", Technical Report CSE-TR-95-91, Software Systems Research Laboratory, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, Michigan.

[Angus 1994] Ian Angus, private communication.

[Banerjee et al 1987] Jay Banerjee, Won Kim, Hyoung-Joo Kim and Henry F. Korth, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases", ACM SIGMOD, pp. 311-322 (December 1987).

[Belady & Lehman 1985] L. A. Belady and M. M. Lehman, "Programming system dynamics or the metadynamics of systems in maintenance and growth", Res. Rep. RC3546, IBM, 1971. Page citations from reprint in M. M. Lehman, L. A. Belady, Editors, *Program Evolution: Processes of Software Change*, Ch. 5, APIC Studies in Data Processing, No. 27, Academic Press, London, 1985.

[Bennett 1991] K. H. Bennett, "Automated support of software maintenance", Information and Software Technology, 33(1), pp. 74-85 (Jan/Feb 1991).

[Bergstein 1991] Paul L. Bergstein, "Object-preserving Class Transformations", Proceedings of OOPSLA '91, pp. 299-313 (1991).

[Bergstein & Hürsch 1993] Paul L. Bergstein and Walter L. Hürsch, "Maintaining Behavioral Consistency during Schema Evolution", Proceedings of the First JSSST International Symposium on Object Technologies for Advanced Software, S. Nishio and A. Yonezawa (editors), pp. 176-193 (November 1993).

[Berliner 1990] Brian Berliner, "CVS II: Parallelizing Software Development", Proceedings of the Winter 1990 USENIX conference, Washington, DC, USA, pp. 341-352 (Jan 22-26, 1990).

[Biggerstaff 1989] Ted J. Biggerstaff, "Design Recovery for Maintenance and Reuse", IEEE Computer, pp. 36-49 (July 1989).

[Biggerstaff et al 1993] Ted J. Biggerstaff, Bharat G. Mitbander and Dallas Webster, Proceedings of the 15th International Conference on Software Engineering (ICSE), pp. 482-498 (1993).

[Boehm 1976] Barry W. Boehm, "Software Engineering", IEEE Transactions on Computers 25(12), pp. 1226-1241 (December 1976).

[Boehm 1981] Barry W. Boehm, "Software Engineering Economics", Prentice-Hall (1981).

[Borland 1993a] Borland's README file for C++ version 4.0.

[Borland 1993b] "ObjectWindows for C++ Programmer's Guide", version 2.0, Borland, (1993).

[Borland 1993c] "Borland C++ for Windows Library Reference", version 4.0, Borland, (1993).

[Brenan 1992] M. D. Brenan, "mawk - pattern scanning and processing language", Unix Manual Page, (January 1992). Available by ftp from oxy.edu (134.69.1.2), public/mawk.

[Casais 1992] Eduardo Casais, "An Incremental Class Reorganization Approach", Proceedings of ECOOP '92, pp. 114-132 (June 1992).

[Chikofsky & Cross 1990] Elliot J. Chikofsky and James H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy", IEEE Software, pp. 13-17 (January 1990).

[Chow 1995] Kingsum Chow, "Program Transformation for Asynchronous Software Maintenance", Proceedings of ICSE-17 Workshop on Program Transformation for Software Evolution, William Griswold, editor, The 17th International Conference on Software Engineering, April 24-28, 1995, Seattle, Washington, USA. The proceedings are also published as Technical Report Number CS95-418, Computer Science and Engineering, University of California, San Diego, pp. 29-34.

[Chow 1996a] Kingsum Chow, research presentation at Intel and discussion with Andy Glew and Gil Nieger, Feb 1996.

[Chow 1996b] Kingsum Chow, "Software Quality Management by Responsibility Driven Software Evolution", Fourteenth Annual Pacific Northwest Software Quality Conference, Portland, Oregon, Oct. 29-30, 1996.

[Chow & Notkin 1996a] Kingsum Chow and David Notkin, "Asynchronous Software Evolution", Asia-Pacific Workshop on Software Engineering Research, March 21, 1996, Hong Kong.

[Chow & Notkin 1996b] Kingsum Chow and David Notkin, "Semi-automatic Update of Applications in Response to Library Changes", Technical Report UW-CSE 96-03-01, Department of Computer Science & Engineering, University of Washington, Seattle, Washington, USA (March 1996).

[Chow & Notkin 1996c] Kingsum Chow and David Notkin, "Semi-automatic Update of Applications in Response to Library Changes", International Conference on Software Maintenance, Monterey, California, USA, Nov 4-8, 1996.

[Cooper 1994] Michael Cooper, "Rdist Version 6.1", "ftp://ftp.usc.edu/pub/rdist/", May 1994.

[Davis 1992] Alan M. Davis, "Why Industry Often Says 'No Thanks' to Research", IEEE Software, November 1992, pp. 97-99.

[Davis 1996] Alan M. Davis, "Eras of Software Technology Transfer", IEEE Software, March 1996, pp. 4-7.

[Ellis & Stroustrup 1992] Margaret A. Ellis and Bjarne Stroustrup, "The Annotated C++ Reference Manual", AT&T Bell Telephone Laboratories, Inc. (1990). Reprinted with corrections (May 1992).

[Feldman 1979] Stuart I. Feldman, "Make - a Program for Maintaining Computer Programs", Software - Practice & Experience, 9(4) pp. 255-265 (April 1979).

[Garlan et al 1986] David Garlan, Charles W. Krueger and Barbara J. Staudt, "A Structural Approach to the Maintenance of Structure-Oriented Environments", ACM Transactions on Programming Languages and Systems, 16(3) pp. 727-774 (May 1994).

[Garlan et al 1994] David Garlan, Charles W. Krueger and Barbara Staudt Lerner, "TransformGen: Automating the Maintenance of Structure-Oriented Environments", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (editor: Peter Henderson), Palo Alto, California, USA, Dec 9-11, 1986. Also published as SIGPLAN Notices 22(1) (January 1987).

[Garlan et al 1994] David Garlan, Charles W. Krueger and Barbara Staudt Lerner, "TransformGen: Automating the Maintenance of Structure-Oriented Environments", ACM Transactions on Programming Languages and Systems, 16(3) pp. 727-774 (May 1994).

[Ghezzi et al 1991] Carlo Ghezzi, Mehdi Jazayeri and Dino Mandrioli, "Fundamentals of Software Engineering", Prentice-Hall, Inc. (1991).

[Glew 1996] Andy Glew, Intel Corp., private communications.

[Goos et al 1983] G. Goos, W. A. Wulf, A.Evans, Jr., and K. J. Butler, "DIANA - An Intermediate Language for Ada" (Revised version), Lecture Notes in Computer Science (eds: G. Goos and J. Hartmanis), Springer-Verlag, Berlin, Germany (1983).

[Gorlen et al 1990] Keith E. Gorlen, Sanford M. Orlow and Perry S. Plexico, "NIH Class Library Revision 3.0 Release Notes" (1990). Available from "ftp://alw.nih.gov/pub/nihcl.tar.Z".

[Griswold 1991] William G. Griswold, "Program Restructuring as an Aid to Software Maintenance", Ph.D. Thesis, Department of Computer Science and Engineering,

University of Washington, Seattle, Washington, USA. Technical Report 91-08-04 (August 1991).

[Griswold et al 1996] W. G. Griswold, D. C. Atkinson and C. McCurdy, "Fast, Flexible Syntactic Pattern Matching and Processing", Proceedings of the IEEE 1996 Workshop on Program Comprehension, March 29-31, 1996, Berlin, Germany.

[Griswold & Notkin 1993] William G. Griswold and David Notkin, "Automated Assistance for Program Restructuring", ACM Transactions on Software Engineering and Methodology. 2(3) pp. 228-269 (July 1993).

[Griswold & Notkin 1995] William G. Griswold and David Notkin, "Architectural Tradeoffs for a Meaning-Preserving Program Restructuring Tool", IEEE Transactions on Software Engineering, 21(4) pp. 275-287 (April 1995).

[Hahne & Hiroyuki 1994] Bruce Hahne and Sato Hiroyuki, "Using YACC and Lex with C++", SIGPLAN Notices. vol.29, no.12. pp. 94-103. Dec. 1994.

[Horwitz & Reps 1992] Susan Horwitz and Thomas Reps, "The Use of Program Dependence Graphs in Software Engineering" Proceedings of 14th International Conference on Software Engineering, May 11-15, 1992, Melbourne, Australia, pp. 392-411.

[Jackson & Ladd 1994] Daniel Jackson and David A. Ladd, "Semantic Diff: a tool for summarizing the effects of modifications", IEEE Proceedings, International Conference on Software Maintenance, pp. 243-52. (Victoria, British Columbia, Canada, 19-23 Sept. 1994).

[Jackson 1995] Michael Jackson, "Software Requirements & Specifications: a lexicon of practice, principles and prejudices", ACM Press, Addison-Wesley (1995).

[James] "The Tao Of Programming", translated by Geoffrey James, transcribed by Seth Robertson, HTMLified by Patrick Fitzgerald, http://iquest.com/~fitz/diversions/tao-prog.html.

[Johnson 1975] S. C. Johnson, "Yacc - Yet Another Compiler-Compiler", Computer Science Technical Report 32, AT&T Bell Labs, Murray Hill, NJ, USA (1975).

[Johnson & Opdyke 1993] Ralph E. Johnson and William F. Opdyke, "Refactoring and Aggregation", Proceedings of the First JSSST International Symposium on Object Technologies for Advanced Software, S. Nishio and A. Yonezawa (editors), pp. 264-278 (November 1993).

[Kim 1990] Won Kim, "Introduction to object-oriented databases", MIT Press (1990).

[Ladd & Ramming 1995] David A. Ladd and J. Christopher Ramming, "A*: A Language for Implementing Language Processors", IEEE Transactions on Software Engineering 21(11) pp. 894-901 (November 1995).

[Lammers 1989] Susan Lammers, "Programmers at Work", Microsoft Press. p.33 (1989).

[Lampson 1984] Butler W. Lampson, "Hints for Computer System Design", IEEE Software pp. 11-28 (January 1984).

[Lea 1988] Doug Lea, "The GNU C++ Library" (libg++), USENIX C++ Conference, 1988.

[Lehman 1980] M. M. Lehman, "On understanding laws, evolution and conservation in the large-program life cycle", J. Syst. Softw. 1, 3 (1980).

[Lesk & Schmidt 1975] M. E. Lesk and E. Schmidt, "Lex - A Lexical Analyser Generator", Computer Science Technical Report 39, AT&T Bell Labs, Murray Hill, NJ, USA (1975).

[Levine et al 1992] John R. Levine, Tony Mason and Doug Brown. "lex & yacc", 2/ed. O'Reilly & Associates, Inc. (1992).

[Lieberherr et al 1991] Karl J. Lieberherr, Paul Bergstein and Ignacio Silva-Lepe, "From objects to classes: algorithms for optimal object-oriented design", Software Engineering Journal, pp. 205-228 (July 1991).

[Lieberherr & Holland 1989] Karl J. Lieberherr and I. Holland, "Assuring Good Style for Object-Oriented Programs", IEEE Software, pp. 38-48 (September 1989).

[Lieberherr & Xiao 1993a] Karl J. Lieberherr and Cun Xiao, "Object-oriented Software Evolution", IEEE Transactions on Software Engineering 19, 4 pp. 313-343 (April 1993).

[Lieberherr & Xiao 1993b] Karl J. Lieberherr and Cun Xiao, "Minimizing Dependency on Class Structures with Adaptive Programs", Proceedings of the First JSSST International Symposium on Object Technologies for Advanced Software, S. Nishio and A. Yonezawa (editors), pp. 424-441 (November 1993).

[Litman 1993] Andy Litman, "An Implementation of Precompiled Headers", Software-Practice and Experience, 23(3), 341-350 (March 1993).

[Moriconi & Winkler 1990] Mark Moriconi and Timothy C. Winkler, "Approximate Reasoning About the Semantics Effects of Program Changes", IEEE Transactions on Software Engineering, 16(9), pp. 980-992, (September 1990).

[Meyer 1992] Bertrand Meyer, "Eiffel: The Language", Prentice Hall (1992).

[Meyers 1992] Scott Meyers, "Effective C++", Addison-Wesley 1992.

[Microsoft 1994a] Microsoft MFC Migration Guide. Microsoft Press, 1994.

[Microsoft 1994b] Microsoft Foundation Classes 3.0: C++ Application Framework for Microsoft Windows (Technical White Paper), July 1994.

[Morgenthaler & Griswold 1995] J. David Morgenthaler and William G. Griswold, "Program Analysis for Practical Program Restructuring", Proceedings of ICSE-17 Workshop on Program Transformation for Software Evolution, William Griswold, editor, The 17th International Conference on Software Engineering, April 24-28, 1995, Seattle, Washington, USA. The proceedings are also published as Technical Report Number CS95-418, Computer Science and Engineering, University of California, San Diego, pp. 75-80.

[Murphy & Notkin 1995] Gail C. Murphy and David Notkin, "Lightweight Source Model Extraction" Proceedings of the Third ACM Symposium on the Foundations of Software Engineering, pp. 116-127 (FSE '95).

[Notkin et al 1993] David Notkin, David Garlan, William G. Griswold and Kevin Sullivan, "Adding Implicit Invocation to Languages: Three Approaches", Proceedings of the First JSSST International Symposium on Object Technologies for Advanced Software, S. Nishio and A. Yonezawa (editors), pp. 488-510 (November 1993)

[Opdyke 1992] William F. Opdyke, "Refactoring Object-Oriented Frameworks", Ph.D. Thesis. Dept. of Computer Science, University of Illinois at Urbana-Champaign (1992).

[Pancake 1995] Cherri M. Pancake, "The Promise and the Cost of Object Technology: A Five-Year Forecast", Communications of the ACM 38, 10 pp. 33-49 (October 1995).

[Parnas 1972] David L. Parnas. "On the Criteria To Be Used in Decomposing System into Modules", Communications of the ACM pp. 1053-1058 (December 1972).

[Parnas 1979] David L. Parnas, "Designing Software for Ease of Extension and Contraction", IEEE Transactions on Software Engineering. 5(2) pp. 128-138 (March 1979).

[Parnas 1996] David L. Parnas, "Why Software Jewels Are Rare", IEEE Computer, 29(2), pp. 57-60 (February 1996).

[Parr 1994] Terence J. Parr, "An Overview of SORCERER: A Simple Tree-Parser Generator", International Conference on Compiler Construction (April 1994). Available from "ftp://ftp.parr-research.com/pub/pccts/papers/sorcerer.ps.Z".

[Parr 1995] Terence J. Parr. "Language Translation Using PCCTS and C++ (A Reference Guide)". Available from "ftp://ftp.parr-research.com/pub/pccts/Book/".

[Parr et al 1994a] Terence Parr, Russell W. Quong, Will Cohen and Hank Dietz, "PCCTS 1.30 Release Nodes", (Nov 1, 1994). Available from "ftp://ftp.parr-research.com/pub/pccts/1.30/".

[Parr et al 1994b] Terence John Parr, Aaron Sawdey and Gary Funck, "The SORCERER Reference Manual", Version 1.00B13. (Nov 12, 1994). Available from "ftp://ftp.parr-research.com/pub/pccts/sorcerer/".

[Parr & Quong 1996] Terence J. Parr and Russell W. Quong, "LL and LR Translators Need k > 1 Lookahead", ACM SIGPLAN Notices, 31(2), pp. 27-34 (February 1996).

[Reiss 1993] Steven P. Reiss, private conversation.

[Rosenblum 1985] David S. Rosenblum, "A Methodology for the Design of Ada Transformations Tools in a DIANA Environment", IEEE Software, pp. 24-33 (March 1985).

[Rumbaugh et al 1991] James Rumbaugh, Michael Blaha, WIlliam Premerlani, Frederick Eddy and William Lorensen, "Object-Oriented Modeling and Design", Prentice-Hall, Inc. (1991).

[Schwanke & Kaiser 1988] Robert W. Schwanke and Gail E. Kaiser, "Smarter Recompilation", ACM Transactions on Programming Languages and Systems, 10, 4 (October 1988), 627-632.

[Sirkin 1994] Marty J. Sirkin, "A Software System Generator For Data Structures", Ph.D. Thesis, Department of Computer Science and Engineering, University of Washington, Seattle, Washington (1994).

[Stroustrup 1991] Bjarne Stroustrup, "The C++ Programming Language", 2/ed. Addison-Wesley Publishing Company. (1991).

[Stroustrup 1994] Bjarne Stroustrup, "The Design and Evolution of C++", Addison-Wesley (1994).

[Taft 1992] S. Tucker Taft, "Ada 9X: A Technical Summary", Communications of the ACM Nov 1992, p 77.

[Tichy 1985] Walter F. Tichy, "RCS - A System for Version Control" Software - Practice & Experience", 15(7) pp. 637-654 (July 1985).

[Tichy 1986] Walter F. Tichy, "Smart Recompilation", ACM Transactions on Programming Languages and Systems, 8, 3 (July 1986), 273-291.

[USDoD 1980] United States Department of Defense, "Ada Programming Language" MIL-STD-1815.

[Van Den Brand 1996] Mark Van Den Brand, e-mail communication (May 1996).

[Van Den Brand & Visser 1996] Mark Van Den Brand and Eelco Visser, "Generation of Formatters for Context-Free Languages", ACM Transactions on Software Engineering Methodology 5(1) pp. 1-41 (January 1996).

[VanHilst & Notkin 1996] Michael VanHilst and David Notkin, "Using C++ Templates to Implement Role-Based Designs" (ISOTAS '96).

[Wall & Schwartz 1990] Larry Wall and Randal L. Schwartz, "Programming perl", O'Reilly & Associates (1990).

[Watson 1989] Dan Watson, "High-Level Languages and Their Compilers", Addison-Wesley.

[Zimmer 1990] J. A. Zimmer, "Restructuring for Style", Software-Practice and Experience, 20(4), 365-389 (April 1990).

Appendix A

Implementation Details

This appendix describes the implementation details of our approach. Appendix A.1 provides the PCCTS grammar for a C-like language. Appendix A.2 provides the syntax tree transformation specification. Appendix A.3 provides the Perl script for updating source files.

A.1 A C-like grammar using PCCTS

```
// $Id: lang1.g,v 2.74 1996/05/22 00:26:40 kingsum Exp $
<<
#include "mytoken.h"
typedef MyToken ANTLRToken;
#include "myast.h"
#include "type.h"
#include "symtab.h"
>>

// default: for lang1.cpp
<<
const char* DEFAULT_BLOCK_NAME = "anonymous";
// _ASE_NAMES_: Needs introduction of function names
>>

//=======================
// specify for lexer.cpp

#lexaction
<<
char* comments_p = NULL;
>>

//=======================

#token WHITE_SPACE "[\ \t]+"
   <<
   if (comments_p) {
      char* p1 = comments_p;
```

```
      comments_p = new char [strlen(p1) + strlen(lextext()) + 1];
      strcpy(comments_p, p1);
      strcat(comments_p, lextext());
   } else {
      comments_p = new char [strlen(lextext()) + 1];
      strcpy(comments_p, lextext());
   }
   skip();
   >>

#token NEW_LINE"\n"
   <<
   newline();
   if (comments_p) {
      char* p1 = comments_p;
      comments_p = new char [strlen(p1) + strlen(lextext()) + 1];
      strcpy(comments_p, p1);
      strcat(comments_p, lextext());
   } else {
      comments_p = new char [strlen(lextext()) + 1];
      strcpy(comments_p, lextext());
   }
   skip();
   >>

//======================
// The order of the tokens may be important. They will be matched
in the order
// they are declared.

#token INCREMENT "\+\+"
#token DECREMENT "\-\-"
#token LOGICAL_AND "\&\&"
#token LOGICAL_OR "\|\|"
#token EXCLAMATION "\!"
#token TILDE "\~"
#token AMPERSAND "\&"

#tokclass INC_DEC_OP { INCREMENT DECREMENT }
#tokclass LOGICAL_OP { LOGICAL_AND LOGICAL_OR }

#token SHIFT_LEFT "\<\<"
#token SHIFT_RIGHT "\>\>"

// just a few examples of assignment operators
#token ASSIGN"="
#token PLUS_ASSIGN "\+="
#token MINUS_ASSIGN "\-="
#token MULTIPLY_ASSIGN "\*="
#token DIVIDE_ASSIGN "\/="
#token OR_ASSIGN "\|="
```

```
#token AMPERSAND_ASSIGN "\&="
#tokclass ASSIGN_TOKENS { ASSIGN PLUS_ASSIGN MINUS_ASSIGN
MULTIPLY_ASSIGN DIVIDE_ASSIGN
           OR_ASSIGN AMPERSAND_ASSIGN }

// the following would be used as a range in Sorcerer, shouldn't
break them
#token PLUS"\+"
#token STAR"\*"
#token MINUS"\-"
#token DIVIDE"\/"
#token PERCENT"\%"
#token LESS_THAN "<"
#token GREATER_THAN ">"
#token LESS_EQUAL "<="
#token GREATER_EQUAL ">="
#token EQUAL_TO "=="
#token NOT_EQUAL "!="

#tokclass PTR_OP { STAR AMPERSAND }

#tokclass REL_OP { LESS_THAN GREATER_THAN LESS_EQUAL GREATER_EQUAL
EQUAL_TO NOT_EQUAL }

#token L_BRACE "\{"
#token R_BRACE "\}"
#token L_PAREN "\("
#token R_PAREN "\)"
#token L_SQUARE "\["
#token R_SQUARE "\]"
#token COMMA ","
#token COLON":"
#token SEMICOLON ";"
#token QUESTION "\?"

//---------------------------------------------------------------
//
#token C_COMMENT "/\*(~[\*]*(\*~[/])*)*\*/"
   <<
   {
      char* pc = lextext();
      while (pc = strchr(pc, '\n')) {
         newline();
         pc++;
      }
   }
   //
   if (comments_p) {
      char* p1 = comments_p;
      comments_p = new char [strlen(p1) + strlen(lextext()) + 1];
      strcpy(comments_p, p1);
```

```
        strcat(comments_p, lextext());
    } else {
        comments_p = new char [strlen(lextext()) + 1];
        strcpy(comments_p, lextext());
    }
    skip();
    >>

#token CPP_COMMENT "//~[\n\r]*"
    <<
    // fprintf(stderr, "%s\n", lextext());
    // comments_p = new char [strlen(lextext()) + 1];
    // strcpy(comments_p, lextext());
    if (comments_p) {
        char* p1 = comments_p;
        comments_p = new char [strlen(p1) + strlen(lextext()) + 1];
        strcpy(comments_p, p1);
        strcat(comments_p, lextext());
    } else {
        comments_p = new char [strlen(lextext()) + 1];
        strcpy(comments_p, lextext());
    }
    skip();           // should use more or skip? think...
    >>

#token INT_TYPE "int"
#token FLOAT_TYPE "float"
#token CHAR_TYPE "char"
#token BOOL_TYPE "bool"
#token VOID_TYPE "void"

// Preprocessing directives keywords
#token PP_INCLUDE "\#include"
#token PP_DEFINE "\#define"

// Other preprocessing directives to deal with later 12/20/95
#token PP_IFDEF "\#ifdef"
#token PP_IF "\#if"
#token PP_ENDIF "\#endif"

// C/C++ keywords
#token STATIC "static"
#token CONST "const"
#token SIZEOF "sizeof"
#token TYPEDEF "typedef"
#token IF "if"
#token ELSE "else"
#token RETURN "return"
#token WHILE "while"
#token DO "do"
#token FOR "for"
```

```
#token SWITCH "switch"
#token CASE "case"
#token DEFAULT "default"
#token BREAK "break"
// struct
#token STRUCT "struct"
#token DOT "\."
#token ARROW "\-\>"
#tokclass DOT_OR_ARROW { DOT ARROW }
#token ENUM "enum"

// EOF
#token EOF_TOKEN "@"

// Kingsum 6/1/95 need to put them before their use, unlike tokens.
#tokclass ADD_OP { PLUS MINUS }
#tokclass MUL_OP { STAR PERCENT DIVIDE }
#tokclass UNARY_OP { STAR AMPERSAND PLUS MINUS EXCLAMATION TILDE }

// introduce boolean literals because it will be supported in C++
(Working Paper 4/95)
#token BOOL_FALSE"false"
#token BOOL_TRUE"true"
#tokclass BOOL { BOOL_FALSE BOOL_TRUE }

// 12/5/95
// Literals are put at the end because they are usually more
general than
// the previous (more specific) tokens.

#token ID"[_A-Za-z][_A-Za-z0-9]*"
#token FLOAT"[0-9]+\.[0-9]+"
#token INT"[0-9]+"
#token CHAR"\'{\\}[\0x00-\0x7f]\'"// all 128 characters
#token STRING"\"~[\"]*\""

#tokclass LITERAL { FLOAT INT CHAR BOOL STRING }

//======================
// No source derivable tokens should occur after this line.

// The following tokens are artificially introduced for tree-
pattern matching.
// They don't have any printable values.

#token STRUCTDECL
#token ENUMDECL
#token FUNCDECL
#token FUNCDEF
#token PARAMDECL
#token VARDECL
```

```
#token VARARRAYDECL
#token VARSTRUCTDECL
#token COND_EXPR

// unary tokens for sorcerer
#token POINTER// unary *
#token ADDRESS// unary &
#token LOGICAL_NOT// unary !
#token BITWISE_NOT// unary ~
#token UNARY_PLUS// unary +
#token UNARY_MINUS// unary -

// ideas from tmoog
#token FCALL
// _ASE_TOKENS_: // need to add here

//======================
class SimpleParser {

// The type rule returns a pointer to a Type object.
//
type > [Type* pType]:
   ( INT_TYPE
   << $pType = intType; >>
   | FLOAT_TYPE
   << $pType = floatType; >>
   | CHAR_TYPE
   << $pType = charType; >>
   | VOID_TYPE
   << $pType = voidType; >>
   | BOOL_TYPE
   << $pType = boolType; >>

   | struct_type [$pType]

   | ENUM idEnum:ID
   <<
   cerr << "Enum " << idEnum->getText() << " is used.\n";
   $pType = new EnumType(idEnum->getText());
   >>
   )
   ( STAR
   <<
   $pType = new PtrType($pType);
   >>
   )* { AMPERSAND
   <<
   $pType = new RefType($pType);
   >>
   }
   ;
```

```
struct_type! [const Type*& pType]:
   st:STRUCT id:ID
   <<
   cerr << "Struct " << id->getText() << " is used.\n";
   $pType = new StructType(id->getText());
// _ASE_SN_BEGIN_
// _ASE_SN_END_
   {}
   #0 = #(#[st], #[id]);
   >>
   ;

do_all:
   ( include_directive
   | define_directive
   | type_def
   | (struct_decl)?
   | (enum_decl)?
   | (decl_stmt)?
   | (func_decl)?
   | func_def
   )+
   EOF_TOKEN
   <<
   globalScope->Put(cerr, 0);
   >>
   ;

include_directive:
   include_quote_string
   ;

include_quote_string! :
   pp:PP_INCLUDE st:STRING
   <<
// _ASE_IH_BEGIN_
// _ASE_IH_END_
   {}
   #0 = #(#[pp], #[st]);
   >>
   ;

define_directive:
   PP_DEFINE^ ID { ID | LITERAL }
   ;

type_def:
   << const Type* pType = NULL; >>
   TYPEDEF^ ( ID | type > [pType] ) id:ID SEMICOLON
   <<
```

```
    if (pType) {
        cerr << "Type " << id->getText() << " is equivalent to ";
        pType->Put(cerr);
        cerr << endl;
    } else {
        cerr << "typedef for simple ID's\n";
    }
    >>
    ;

struct_decl:
    << Type* pType = NULL; >>
    STRUCT id:ID
    <<
    #0 = #(#[STRUCTDECL], #0);
    pType = new StructType(id->getText());
    cerr << "Struct Decl: " << id->getText() << endl;
    >>
    L_BRACE ( var_def [pType] )+ R_BRACE SEMICOLON
    <<
    currentScope->Enter(new SymTabEntry(id->getText(), pType));
    cerr << "Struct Decl End.\n";
    >>
    ;

enum_decl:
    << Type* pType = NULL; >>
    ENUM id:ID L_BRACE
    <<
    cerr << "\nEnum Decl: " << id->getText() << endl;
    pType = new EnumType(id->getText());
    >>
    id1:ID
    <<
    currentScope->Enter(new SymTabEntry(id1->getText(), pType));
    >>
    ( COMMA id2:ID
    <<
    currentScope->Enter(new SymTabEntry(id2->getText(), pType));
    >>
    )* R_BRACE SEMICOLON
    <<
    #0 = #(#[ENUMDECL], #0);
    cerr << "Enum Decl End.\n";
    >>
    ;

func_decl:
    <<
    const Type* pType = NULL;
    FuncType* fType = NULL;
```

```
    >>
    type > [pType] id:ID
    <<
    fType = new FuncType(id->getText(), pType);
    >>
    param_decls [fType, false] SEMICOLON
    <<
    #0 = #(#[FUNCDECL], #0);
    currentScope->Enter(new SymTabEntry(id->getText(), fType));
    >>
    ;

func_def:
    <<
    const Type* pType = NULL;
    FuncType* fType = NULL;
    >>
    type > [pType] id:ID
    <<
    fType = new FuncType(id->getText(), pType);
    >>
    <<
    SymTabScope* localScope = new SymTabScope(id->getText(),
currentScope);
    currentScope = localScope;
    >>
    param_decls [fType, true] block [NULL]
    <<
    currentScope->Enter(new SymTabEntry(id->getText(), fType));
    #0 = #(#[FUNCDEF], #0);
    >>
    ;

param_decls [FuncType* fType, bool enterSymbols] :
    L_PAREN^ { param_decl [$fType, $enterSymbols]
    (COMMA param_decl [$fType, $enterSymbols])* } R_PAREN
    ;

param_decl [FuncType* fType, bool enterSymbols] :
    <<
    const Type* pType = NULL;
    const Type* pt = NULL;
    >>
    type > [pType] id:ID
    { ASSIGN expr [pt] }
    <<
    fType->AddParam(pType, pt != NULL);
    if ($enterSymbols) {
       currentScope->Enter(new FormalSTE(id->getText(), pType));
    }
    #0 = #(#[PARAMDECL], #0);
```

```
    >>
    ;

block [const char* inName] :
    L_BRACE^
    <<
    if (inName) {
        SymTabScope* localScope = new SymTabScope(inName,
currentScope);
        currentScope = localScope;
    }
    >>
    stmts
    R_BRACE
    <<
    currentScope = currentScope->GetParent();
    >>
    ;

stmts:
    (stmt)*
    ;

var_def [Type* pParentType]:
    ( ( struct_var_def [$pParentType] )?
    | ( simple_var_def [$pParentType] )?
    | array_var_def [$pParentType]
    )
    ;

simple_var_def [Type* pParentType]:
    <<
    const Type* pType = NULL;
    >>
    type > [pType]  id:ID { ASSIGN ( ID | LITERAL ) } SEMICOLON
    <<
    #0 = #(#[VARDECL], #0);
    if ($pParentType) {
        $pParentType->Add(new NameTypePair(id->getText(), pType));
        cerr << "Field: " << id->getText() << " Type: ";
        pType->Put(cerr);
        cerr << endl;
    }
    >>
    ;

struct_var_def [Type* pParentType] :
    <<
    SymTabScope* foundScope = NULL;
    >>
```

```
   STRUCT structId:ID varId:ID SEMICOLON
   <<
   cerr << varId->getText() << " is of type " << structId-
>getText() << endl;
   const SymTabEntry* foundEntry = currentScope->Lookup(structId-
>getText(), foundScope);
   if ($pParentType == NULL) {
      if (foundEntry) {
         currentScope->Enter(new SymTabEntry(varId->getText(),
foundEntry->GetType()));
      } else {
         currentScope->Enter(new SymTabEntry(varId->getText(),
                        new StructType(structId->getText())));
         cerr << "Incomplete type for Struct, id = " << structId-
>getText() << "\n";
      }
   } else {
      if (foundEntry) {
         $pParentType->Add(new NameTypePair(varId->getText(),
foundEntry->GetType()));
      } else {
         $pParentType->Add(new NameTypePair(varId->getText(),
                        new StructType(structId->getText())));
         cerr << "Incomplete type for Struct, id = " << structId-
>getText() << "\n";
      }
   }
   #0 = #(#[VARSTRUCTDECL], #0);
   >>
   ;

array_var_def [Type* pParentType] :
   <<
   const Type* pType = NULL;
   int count = 0;
   >>
   type > [pType] ID indices [count] SEMICOLON
   <<
   #0 = #(#[VARARRAYDECL], #0);
   >>
   ;

stmt:
   decl_stmt
   | expr_stmt// from this one
   | if_stmt
   | return_stmt
   | while_stmt
   | do_stmt
   | for_stmt
   | break_stmt
```

```
    | switch_stmt
    | block [DEFAULT_BLOCK_NAME]
    ;

decl_stmt:// need to use arguments to pass type to item later
    << const Type* pType = NULL; >>
    (struct_var_def [NULL] )?
    | decl_spec > [pType] init_declarator [pType] (COMMA
init_declarator [pType])*
      SEMICOLON
    ;

decl_spec > [const Type* pType]:
    type > [$pType]
    ;

init_declarator [const Type* baseType] :
    << const Type* actualType = NULL; >>
    declarator [$baseType] > [actualType] { initializer
[actualType] }
    ;

declarator [const Type* baseType] > [const Type* actualType] :
    <<
    int count = 0;
    >>
    id:ID { indices [count] }// old form
    <<
    $actualType = $baseType;
    for (int i = 0; i < count; i++) {
       $actualType = new PtrType($actualType);
    }
    currentScope->Enter(new VarSTE(id->getText(), $actualType));
    >>
    ;

arg_decl_list:
    arg_decl ( COMMA arg_decl )*
    ;

arg_decl:
    <<
    const Type* baseType = NULL;
    const Type* actualType = NULL;
    const Type* pt;
    >>
    decl_spec > [baseType] declarator [baseType] > [actualType] {
ASSIGN expr [pt] }
    ;

initializer [const Type* expectedType] :
```

```
    << const Type* pt = NULL; >>
    ASSIGN
    ( assign_expr [pt]
    | L_BRACE initializer_list [$expectedType->GetElemType()] // {
COMMA }
    <<
    >>
    R_BRACE )
    ;

initializer_list [const Type* expectedType] :
    << const Type* pt = NULL; >>
    ( assign_expr [pt]
    | L_BRACE initializer_list [$expectedType->GetElemType()] // {
COMMA }
    <<
    >>
    R_BRACE )
    ( COMMA
    ( assign_expr [pt]
    | L_BRACE initializer_list [$expectedType->GetElemType()] // {
COMMA }
    <<
    >>
    R_BRACE ) )*
    ;

if_stmt:
    << const Type* pt = NULL; >>
    ( IF^ L_PAREN expr [pt] R_PAREN stmt ELSE stmt )?
    | IF^ L_PAREN expr [pt] R_PAREN stmt
    ;

while_stmt:
    << const Type* pt = NULL; >>
    WHILE^ L_PAREN expr [pt] R_PAREN stmt
    ;

do_stmt:
    << const Type* pt = NULL; >>
    DO^ stmt WHILE L_PAREN expr [pt] R_PAREN SEMICOLON
    ;

for_stmt:
    << const Type* pt = NULL; >>
    FOR^ L_PAREN stmt stmt assign_expr [pt] R_PAREN stmt
    ;

break_stmt:
    BREAK^ SEMICOLON
    ;
```

```
switch_stmt:
   << const Type* pt = NULL; >>
   SWITCH^ L_PAREN expr [pt] R_PAREN L_BRACE
      ( case_stmt )*
      { default_stmt }
   R_BRACE
   ;

case_stmt:
   << const Type* pt = NULL; >>
   CASE^ expr [pt] COLON stmts
   ;

default_stmt:
   DEFAULT^ COLON stmts
   ;

return_stmt:
   << const Type* pt = NULL; >>
   RETURN^ { expr [pt] } SEMICOLON
   ;

expr_stmt:
   << const Type* pt = NULL; >>
   expr [pt] SEMICOLON
   ;

assign_stmt:
   << const Type* pt = NULL; >>
   assign_expr [pt] SEMICOLON
   ;

pointers:
   ( STAR )+
   ;

indices [int& dims] :
   <<
   const Type* pt = NULL;
   dims = 0;
   >>
   ( L_SQUARE e1:expr [pt] R_SQUARE
      <<
      if (!pt->IsIntegralType()) {
         cerr << "Integral type expected for: ";
         ((AST*)#e1)->inorder(cerr);
      }
      dims++;
      >>
   )+
```

```
    ;

 func_call! [const Type*& pType] :
    <<
    IntPArray* argVals = new IntPArray;
    TypeArray* paramTypes = new TypeArray;
    int numArgs = 0;
    FuncType* funcType = NULL;
    SymTabScope* foundScope = NULL;
    >>
    fn:ID lp:L_PAREN { fa:fargs [paramTypes, argVals] > [numArgs] }
    <<
    funcType = new FuncType(fn->getText(), anyType, paramTypes);
    const SymTabEntry* foundEntry = currentScope->Lookup(fn-
>getText(), foundScope);
    if (foundEntry && foundEntry->GetType()->Same(funcType)) {
       funcType->Put(cerr);
       cerr << " is matched with ";
       foundEntry->GetType()->Put(cerr);
       cerr << endl;
       $pType = foundEntry->GetType()->GetReturnType();
    } else {
       cerr << "Type Error: No function is suitable for " << fn-
>getText() << " : ";
       funcType->Put(cerr);
       cerr << endl;
       $pType = voidType;
    }
    >>
    rp:R_PAREN
    <<
// _ASE_FCALL_BEGIN_
// _ASE_FCALL_END
    {
       if (#fa) {
           #0 = #(#[FCALL], #[fn], #[lp], #fa, #[rp]);
       } else {
          #0 = #(#[FCALL], #[fn], #[lp], #[rp]);
       }
    }
    >>
    ;

fargs [TypeArray* paramTypes, IntPArray* argVals] > [int numArgs]:
    <<
    const Type* pt = voidType;
    $numArgs = 0;
    >>
    e1:expr [pt]
    <<
    $paramTypes->Add((Type*)pt);
```

```
    if (#e1->type() == INT) {
        $argVals->Add(new int(atoi(#e1->getText())));
    } else {
       $argVals->Add(NULL);
    }
    $numArgs++;
    >>
    ( COMMA e2:expr [pt]
    <<
    $paramTypes->Add((Type*)pt);
    if (#e2->type() == INT) {
        $argVals->Add(new int(atoi(#e2->getText())));
    } else {
       $argVals->Add(NULL);
    }
    $numArgs++;
    >>
    )*
    ;

expr [const Type*& pType] :
   assign_expr [$pType]
   <<
   ((AST*)#0)->SetType(pType);
   >>
   ;

assign_expr [const Type*& pType] :
   cond_expr [$pType] ( ASSIGN_TOKENS cond_expr [$pType] )*
   ;

cond_expr [const Type*& pType] :
   << const Type* pt = NULL; >>
   (logical_expr [pt] QUESTION expr [$pType] COLON cond_expr
[$pType] )?
   <<
   #0 = #(#[COND_EXPR], #0);
   >>
   | logical_expr [$pType]
   ;

logical_expr [const Type*& pType] :
   << const Type* pt = NULL; >>
   rel_expr [$pType]
   ( LOGICAL_OP^ rel_expr [pt]
   <<
   $pType = boolType;
   >>
   )*
   ;
```

```
rel_expr [const Type*& pType] :
   << const Type* pt = NULL; >>
   add_expr [$pType]
   ( REL_OP^ add_expr [pt]
   <<
   $pType = boolType;
   >>
   )*
   ;

add_expr [const Type*& pType] :
   << const Type* pt = NULL; >>
   mul_expr [$pType]
   ( ADD_OP^ mul_expr [pt]
   // need to take  care of this ... assume type coercion based on
first expr for now
   )*
   ;

mul_expr [const Type*& pType] :
   << Type* pt = NULL; >>
   pm_expr [$pType]
   ( MUL_OP^ pm_expr [pt]
   )*
   ;

pm_expr [const Type*& pType] :
   cast_expr [$pType]
   ;

cast_expr [const Type*& pType] :
   unary_expr [$pType]
   ;

unary_expr [const Type*& pType] :
   <<
      const Type* pt = NULL;
   >>
   incop:INCREMENT^ unary_expr [pt]
   <<
   if (!pt) {
      cerr << "Rule Error: pt NULL at line: " << incop->getLine()
<< endl;
   } else if (!pt->IsIntegralType()) {
      cerr << "Type Error: INCREMENT applies to non integral
type.\n";
      cerr << "Type Found: ";
      pt->Put(cerr);
      cerr << endl;
   }
   $pType = pt;
```

```
>>
| decop:DECREMENT^ unary_expr [pt]
<<
if (!pt) {
   cerr << "Rule Error: pt NULL at line: " << decop->getLine()
<< endl;
} else if (!pt->IsIntegralType()) {
   cerr << "Type Error: DECREMENT applies to non integral
type.\n";
   cerr << "Type Found: ";
   pt->Put(cerr);
   cerr << endl;
}
$pType = pt;
>>
| postfix_expr [$pType]
| unary_op_expr [$pType]
| SIZEOF^ L_PAREN type > [$pType] R_PAREN
<<
$pType = intType;
>>
;

unary_op_expr! [const Type*& pType] :
   t11:STAR e11:cast_expr [$pType]
   <<
   t11->setType(POINTER);
   #0 = #(#[t11], #e11);
   $pType = $pType->GetElemType();
   >>
   | t12:AMPERSAND e12:cast_expr [$pType]
   <<
   t12->setType(ADDRESS);
   #0 = #(#[t12], #e12);
   $pType = new PtrType($pType);
   >>
   | t13:EXCLAMATION e13:cast_expr [$pType]
   <<
   t13->setType(LOGICAL_NOT);
   #0 = #(#[t13], #e13);
   if (!$pType->IsIntegralType()) {
      cerr << "Type Error: LOGICAL_NOT.\n";
   }
   >>
   | t14:TILDE e14:cast_expr [$pType]
   <<
   t14->setType(BITWISE_NOT);
   #0 = #(#[t14], #e14);
   if (!$pType->IsIntegralType()) {
      cerr << "Type Error: BITWISE_NOT.\n";
   }
```

```
    >>
    | t1:PLUS e1:cast_expr [$pType]
    <<
    t1->setType(UNARY_PLUS);
    #0 = #(#[t1], #e1);
    if (!$pType->IsNumericType()) {
       cerr << "Type Error: UNARY_PLUS applies to non-numeric
types.\n";
    }
    >>
    | t2:MINUS e2:cast_expr [$pType]
    <<
    t2->setType(UNARY_MINUS);
    #0 = #(#[t2], #e2);
    if (!$pType->IsNumericType()) {
       cerr << "Type Error: UNARY_MINUS applies to non-numeric
types.\n";
    }
    >>
    ;

postfix_expr [const Type*& pType] :
    <<
    SymTabScope* foundScope;
    bool isGlobal = FALSE;
    >>
    ( id:ID^
    <<
    const SymTabEntry* foundEntry = currentScope->Lookup(id-
>getText(), foundScope);
    if (foundEntry) {
       $pType = foundEntry->GetType();
       // cerr << "Var: " << id->getText() << " has type: ";
       // $pType->Put(cerr);
       // cerr << endl;
    } else {
       cerr << "Symbol not found: " << id->getText() << endl;
    }
    // 5/9/96
    #id->SetScope(foundScope);
    isGlobal = #id->IsGlobal();
    // changing global variables 5/9/96 example
// _ASE_GV_BEGIN_
// _ASE_GV_END_
    { }
    >>
    ( left_right_squares [$pType]
    | INC_DEC_OP
    | dot_and_id [$pType]
    | arrow_and_id [$pType]
    )*
```

```
    )?
    | primary_expr [$pType]
    ;

dot_and_id! [const Type*& pType] :
    dot:DOT id2:ID
    <<
    if ($pType->IsStructType()) {
// _ASE_DT_BEGIN_
// _ASE_DT_END_
    } else {
       cerr << "Struct: " /*<< id->getText()*/ << " not found.\n";
    }
    #0 = #(#[dot], #[id2]);
    >>
    ;

arrow_and_id! [const Type*& pType] :
    arrow:ARROW id2:ID
    <<
    if ($pType->IsPtrType()) {
       $pType = $pType->GetElemType();
       const Type* structType = $pType;
       if ($pType->IsStructType()) {
// _ASE_DT_BEGIN_
// _ASE_DT_END_
          {}
          $pType = $pType->GetFieldType(id2->getText());
          if (!$pType) {
             cerr << "In rule - arrow_and_id - ";
             cerr << "Ptr ";
             structType->Put(cerr);
             cerr << " doesn't have field: " << id2->getText() <<
endl;
          }
       } else {
          cerr << "Pointer Struct expected for: " << id2->getText()
<< endl;
       }
    } else {
       cerr << "Pointer Struct expected for: " << id2->getText() <<
endl;
    }
    #0 = #(#[arrow], #[id2]);
    >>
    ;

left_right_squares [const Type*& pType] :
    <<
    const Type* dummyType = NULL;
    >>
```

```
    L_SQUARE expr [dummyType] rs:R_SQUARE
    <<
    if (!$pType) {
       cerr << "pType NULL at line: " << rs->getLine() << endl;
    } else if ($pType->IsPtrType()) {
       $pType = $pType->GetElemType();
    } else {
       cerr << "Error: dereferencing a non pointer type.\n";
    }
    >>
    ;

primary_expr [const Type*& pType] :
    BOOL^
    << $pType = boolType; >>
    | INT^
    << $pType = intType; >>
    | CHAR^
    << $pType = charType; >>
    | FLOAT^
    << $pType = floatType; >>
    | STRING^
    <<
    $pType = new PtrType(charType);
    >>
    | func_call [$pType]
    | L_PAREN^ expr [$pType] R_PAREN
    ;
}
```

A.2 A C-like syntax tree transformer using Sorcerer

```
// $Id: lang1sor.sor,v 2.53 1996/05/22 00:27:08 kingsum Exp $
#header <<
#include <iostream.h>
#include "tokens.h"
#include "myast.h"
#include "type.h"
>>

#tokdefs "tokens.h"
class SimpleTreeParser {

gen_do_all:
    ( gen_include_directive
    | gen_define_directive
    | gen_type_def
    | gen_struct_decl
```

```
   | gen_enum_decl
   | gen_decl_stmt
   | gen_func_decl
   | gen_func_def)+ EOF_TOKEN
   ;

gen_include_directive:
   #(PP_INCLUDE STRING)
   ;

gen_define_directive:
   #(PP_DEFINE ID { ID | FLOAT..STRING } )
   ;

gen_type_def:                     // ignore the impact for now
   #(TYPEDEF (ID | gen_type) ID SEMICOLON)
   ;

gen_struct_decl:
   #(STRUCTDECL STRUCT ID L_BRACE (gen_var_def)+ R_BRACE
SEMICOLON)
   ;

gen_enum_decl:
   #(ENUMDECL ENUM ID L_BRACE ID ( COMMA ID )* R_BRACE SEMICOLON)
   ;

gen_func_decl:
   #(FUNCDECL  gen_type  ID  gen_param_decls  SEMICOLON)
   ;

gen_func_def:
   #(FUNCDEF  gen_type  ID  gen_param_decls  gen_block)
   ;

gen_param_decls:
   #(L_PAREN { gen_param_decl (COMMA gen_param_decl)* } R_PAREN)
   ;

gen_param_decl:
   #(PARAMDECL gen_type ID { ASSIGN gen_expr } )
   ;

gen_block:
   #(L_BRACE gen_stmts R_BRACE)
   ;

gen_var_def:
   gen_array_var_def
   | gen_simple_var_def
   | gen_struct_var_def
```

```
    ;

gen_simple_var_def:
    #( VARDECL gen_type ID { ASSIGN ( ID | FLOAT..STRING ) }
SEMICOLON )
    ;

gen_array_var_def:
    #(VARARRAYDECL gen_type ID gen_indices SEMICOLON)
    ;

gen_struct_var_def:
    #(VARSTRUCTDECL STRUCT ID ID SEMICOLON)
    ;

gen_stmts:
    ( gen_stmt )*
    ;

gen_stmt:
    gen_decl_stmt
    | gen_expr_stmt
    | gen_return_stmt
    | gen_if_stmt
    | gen_while_stmt
    | gen_do_stmt
    | gen_for_stmt
    | gen_break_stmt
    | gen_switch_stmt
    | gen_block
    ;

gen_decl_stmt:
    gen_struct_var_def
    | gen_decl_spec gen_init_declarator (COMMA
gen_init_declarator)*  SEMICOLON
    ;

gen_decl_spec:
    gen_type
    ;

gen_init_declarator:
    gen_declarator { gen_initializer }
    ;

gen_declarator:
    ID { gen_indices }
    ;

gen_initializer:
```

```
    ASSIGN ( gen_assign_expr
    | L_BRACE gen_initializer_list /* { COMMA } */ R_BRACE )
    ;

gen_initializer_list:
    ( gen_assign_expr | L_BRACE gen_initializer_list /* { COMMA }
*/ R_BRACE )
    ( COMMA ( gen_assign_expr | L_BRACE gen_initializer_list /* {
COMMA } */ R_BRACE ) )*
    ;

gen_if_stmt:
    #(IF L_PAREN gen_expr R_PAREN gen_stmt { ELSE gen_stmt } )
    ;

gen_while_stmt:
    #(WHILE L_PAREN gen_expr R_PAREN gen_stmt)
    ;

gen_do_stmt:
    #(DO gen_stmt WHILE L_PAREN gen_expr R_PAREN SEMICOLON)
    ;

gen_for_stmt:
    #(FOR L_PAREN gen_stmt gen_stmt gen_assign_expr R_PAREN
gen_stmt)
    ;

gen_break_stmt:
    #(BREAK SEMICOLON)
    ;

gen_switch_stmt:
    #( SWITCH L_PAREN gen_expr R_PAREN L_BRACE
    ( gen_case_stmt )*
    { gen_default_stmt }
    R_BRACE )
    ;

gen_case_stmt:
    #(CASE gen_expr COLON gen_stmts)
    ;

gen_default_stmt:
    #(DEFAULT COLON gen_stmts)
    ;

gen_return_stmt:
    #( RETURN { gen_expr } SEMICOLON)
    ;
```

```
gen_assign_stmt:
   gen_assign_expr SEMICOLON
   ;

gen_expr_stmt:
   gen_expr SEMICOLON
   ;

gen_type:
   ( INT_TYPE
   | FLOAT_TYPE
   | CHAR_TYPE
   | VOID_TYPE
   | BOOL_TYPE
   | gen_struct_type
   | ENUM ID
   ) ( STAR )* { AMPERSAND }
   ;

gen_struct_type:
   #(STRUCT ID);

gen_pointers:
   ( STAR )+
   ;

gen_indices:
   ( L_SQUARE gen_expr R_SQUARE )+
   ;

gen_expr:
   gen_assign_expr // (COMMA gen_assign_expr)*
   ;

gen_assign_expr:
   gen_cond_expr ( ASSIGN..AMPERSAND_ASSIGN gen_cond_expr )*
   ;

gen_cond_expr:
   #(COND_EXPR gen_logical_expr QUESTION gen_logical_expr COLON
gen_cond_expr)
   | gen_logical_expr
   ;

gen_logical_expr:
   #(L_PAREN gen_expr R_PAREN)    // added this rule 5/17
   | #( PLUS..NOT_EQUAL gen_expr gen_expr )
   | #(LOGICAL_AND gen_expr gen_expr)
   | #(LOGICAL_OR gen_expr gen_expr)
   | #(POINTER gen_expr)
   | #(ADDRESS gen_expr)
```

```
    | #(LOGICAL_NOT gen_expr)
    | #(BITWISE_NOT gen_expr)
    | #(UNARY_PLUS gen_expr)
    | #(UNARY_MINUS gen_expr)
    | #(ID ( L_SQUARE gen_expr R_SQUARE | INCREMENT | DECREMENT |
       #(DOT ID) | #(ARROW ID) )*
      )

    | #(INCREMENT gen_logical_expr)
    | #(DECREMENT gen_logical_expr)

    | BOOL_FALSE
    | BOOL_TRUE
    | INT
    | CHAR
    | FLOAT
    | STRING

// _ASE_FCALL_BEGIN_
// _ASE_FCALL_END_
    | #(FCALL ID L_PAREN { gen_expr (COMMA gen_expr)* } R_PAREN)
    | #(SIZEOF L_PAREN gen_type R_PAREN)
    ;

}
```

A.3 Upgrade program

```
#!/uns/bin/perl -w
# using perl 5
# $Id: upgrade_source,v 1.4 1996/05/16 16:36:49 kingsum Exp
kingsum $

local($tmp_root) = '/tmp';

local(@cflags) = ();
local(@src) = ();
local($get_cflags, $get_src) = (0, 0);
local($grammar_file, $transform_file) = ($ARGV[0], $ARGV[1]);

foreach $i (0..$#ARGV) {
   print "arg# $i: $ARGV[$i]\n";

   if ($ARGV[$i] eq 'CFLAGS=') {
      $get_cflags = 1;
      $get_src = 0;
      next;
   } elsif ($ARGV[$i] eq 'SRC=') {
      $get_src = 1;
```

```
            $get_cflags = 0;
            next;
         }

         if ($get_cflags) {
            $cflags[++$#cflags] = $ARGV[$i];
         } elsif ($get_src) {
            $src[++$#src] = $ARGV[$i];
         }
      }

      print "CFLAGS = @cflags\n";
      print "SRC = @src\n";

      foreach $srcfile (@src) {
          update_source($srcfile, @cflags);
      }

      print "upgrade_source finished.\n";
      exit(0);


      sub update_source {
          local($srcfile, @cflags) = @_;

          # fcall changes
          local(@fcall_name);
          local(@fcall_nargs);
          local(@fcall_pattern);
          local(@fcall_action);
          # added 4/30
          local($total_changes) = -1;
          local(@fcall_guard);
          local(@fcall_fail_action);
          # added 5/11
          local($total_gv_changes) = -1;
          local(@gv_oldname);
          local(@gv_newname);
          # added 5/11 for dt_changes
          local($total_dt_changes) = -1;
          local(@dt_old_struct);
          local(@dt_new_struct);
          local(@dt_old_member);
          local(@dt_new_member);
          # added 5/21 for include header changes
          local($total_ih_changes) = -1;
          local(@ih_old_name);
          local(@ih_new_name);
          # added 5/21 for struct name changes (diff-2.2->2.7)
          local($total_sn_changes) = -1;
          local(@sn_old_name);
```

```
local(@sn_new_name);

local($preproc_file) = "$tmp_root/ase.preproc";
local(@includes) = ();

# local variables
local($grammar_temp_file) = "$tmp_root/grammar.temp";
local($transform_temp_file) = "$tmp_root/transform.temp";
# @includes = ();# reset include list
local(%count);

printf("\n\nupgrade_source: $srcfile\n");

# preproc
system("rm -f $preproc_file");
system("g++ -E @cflags $srcfile > $preproc_file");

# extract includes
open(INPUT, "< $preproc_file");
while (<INPUT>) {
  if ( m|^\#\s+\d+\s+"([_a-zA-Z0-9/.-]+\.h)"\s*(\d)?\s*$| ) {
   $includes[++$#includes] = $1;
  }
}
close(INPUT);

# create new grammars and transformers
# remove duplicates, perl book, p.254
for (@includes) {
  $count{$_}++;
}
foreach $include_file (sort keys %count) {
# foreach $include_file (@includes) {
  extract_spec($include_file,
        *total_changes,
        *fcall_name, *fcall_nargs, *fcall_pattern,
*fcall_action,
        *fcall_guard, *fcall_fail_action,
        *total_gv_changes, *gv_oldname, *gv_newname,
        *total_dt_changes,
        *dt_old_struct, *dt_new_struct, *dt_old_member,
*dt_new_member,
        *total_ih_changes, *ih_old_name, *ih_new_name,
        *total_sn_changes, *sn_old_name, *sn_new_name
        );
}

foreach $i (0..$total_changes) {
  printf("\nChange \#$i\n");
  if (defined($fcall_name[$i])) {
      printf("fcall_name = $fcall_name[$i]\n");
```

```
        printf("fcall_nargs = $fcall_nargs[$i]\n");
    }
    printf("fcall_pattern = $fcall_pattern[$i]\n");
    printf("fcall_action = $fcall_action[$i]\n");
    # added 4/30
    if (defined($fcall_guard[$i])) {
        printf("fcall_guard = $fcall_guard[$i]\n");
        printf("fcall_fail_action = $fcall_fail_action[$i]\n");
    }
}

foreach $i (0..$total_gv_changes) {
  printf("\nGV Change \#$i\n");
  printf("gv_oldname = $gv_oldname[$i]\n");
  printf("gv_newname = $gv_newname[$i]\n");
}

foreach $i (0..$total_dt_changes) {
  printf("\nDT Change \#$i\n");
  printf("dt_old_struct = $dt_old_struct[$i]\n");
  printf("dt_old_member = $dt_old_member[$i]\n");
  printf("dt_new_struct = $dt_new_struct[$i]\n");
  printf("dt_new_member = $dt_new_member[$i]\n");
}

foreach $i (0..$total_ih_changes) {
  printf("\nIH Change \#$i\n");
  printf("ih_old_name = $ih_old_name[$i]\n");
  printf("ih_new_name = $ih_new_name[$i]\n");
}

foreach $i (0..$total_sn_changes) {
  printf("\nSN Change \#$i\n");
  printf("sn_old_name = $sn_old_name[$i]\n");
  printf("sn_new_name = $sn_new_name[$i]\n");
}

system("cp $grammar_file.std $grammar_file");
system("cp $transform_file.std $transform_file");
open(G_INFILE, "< $grammar_file");
open(G_OUTFILE, "> $grammar_temp_file");
open(SOR_INFILE, "< $transform_file");
open(SOR_OUTFILE, "> $transform_temp_file");

modify_parser(G_INFILE, G_OUTFILE, $total_changes,
    *fcall_name, *fcall_nargs,
    *fcall_pattern, *fcall_action,
    *fcall_guard, *fcall_fail_action,
    $total_gv_changes, *gv_oldname, *gv_newname,
    $total_dt_changes, *dt_old_struct, *dt_old_member,
    *dt_new_struct, *dt_new_member
```

```
        );
    modify_transformer(SOR_INFILE, SOR_OUTFILE, $total_changes,
             *fcall_name, *fcall_nargs,
             *fcall_pattern, *fcall_action,
             *fcall_guard, *fcall_fail_action
             );

    close(G_INFILE);
    close(G_OUTFILE);
    close(SOR_INFILE);#
    close(SOR_OUTFILE);

    system("mv $grammar_file $grammar_file.old");
    system("mv $transform_file $grammar_file.old");
    system("mv $grammar_temp_file $grammar_file");
    system("mv $transform_temp_file $transform_file");
    # make new program
    system("cd ASE; make ase_upgrade");
    local($tempfile) = "$tmp_root/temp";
    system("ASE/ase_upgrade < $srcfile >! $tempfile");
    system("mv $tmp_root/temp.C $srcfile");
}


sub extract_spec {
    # parameters
    local($include_file,
     *total_changes,
     *fcall_name, *fcall_nargs, *fcall_pattern, *fcall_action,
     *fcall_guard, *fcall_fail_action,
     *total_gv_changes, *gv_oldname, *gv_newname,
     *total_dt_changes,
     *dt_old_struct, *dt_new_struct, *dt_old_member,
*dt_new_member,
     *total_ih_changes, *ih_old_name, *ih_new_name,
     *total_sn_changes, *sn_old_name, *sn_new_name
     ) = @_;

    # local var
    local($i);

    printf("Examining header file: $include_file\n");
    open(H_INFILE, "< $include_file");# open file

    process_file(H_INFILE,
       *total_changes,
       *fcall_name, *fcall_nargs, *fcall_pattern, *fcall_action,
       *fcall_guard, *fcall_fail_action,
       *total_gv_changes, *gv_oldname, *gv_newname,
       *total_dt_changes, *dt_old_struct, *dt_old_member,
       *dt_new_struct, *dt_new_member,
```

```
        *total_ih_changes, *ih_old_name, *ih_new_name,
        *total_sn_changes, *sn_old_name, *sn_new_name
        );# process each file by this subroutine

    close(H_INFILE);#
}


sub process_file {
    local($INFILE,
     *total_changes,
     *fcall_name, *fcall_nargs, *fcall_pattern, *fcall_action,
     *fcall_guard, *fcall_fail_action,
     *total_gv_changes, *gv_oldname, *gv_newname,
     *total_dt_changes, *dt_old_struct, *dt_old_member,
     *dt_new_struct, *dt_new_member,
     *total_ih_changes, *ih_old_name, *ih_new_name,
     *total_sn_changes, *sn_old_name, *sn_new_name
     ) = @_;# subroutine parameter
    local($buf) = "";# local variable

    while ($buf = <$INFILE>) {# scan a line
   if (( $buf =~ m./\*. ) || ($buf =~ m.//.)) {   # begin of C/C++
comment
       process_comment($INFILE,
             *total_changes,
             *fcall_name, *fcall_nargs, *fcall_pattern,
*fcall_action,
             *fcall_guard, *fcall_fail_action,
             *total_gv_changes, *gv_oldname, *gv_newname,
             *total_dt_changes, *dt_old_struct, *dt_old_member,
             *dt_new_struct, *dt_new_member,
             *total_ih_changes, *ih_old_name, *ih_new_name,
             *total_sn_changes, *sn_old_name, *sn_new_name
             );
   }
    }

}


sub process_comment {
    local($INFILE,
     *total_changes,
     *fcall_name, *fcall_nargs, *fcall_pattern, *fcall_action,
     *fcall_guard, *fcall_fail_action,
     *total_gv_changes, *gv_oldname, *gv_newname,
     *total_dt_changes, *dt_old_struct, *dt_old_member,
     *dt_new_struct, *dt_new_member,
     *total_ih_changes, *ih_old_name, *ih_new_name,
     *total_sn_changes, *sn_old_name, *sn_new_name
```

```
      ) = @_;# subroutine parameter
     local($buf) = "";# local variable

     while ($buf = <$INFILE>) {
    if ( $buf =~ m.\*/. ) {
        return;
    } elsif ( $buf =~ /_ASE_FCALL_BEGIN_/ ) {
        $total_changes++;
        process_ase($INFILE, $total_changes,
          *fcall_name, *fcall_nargs, *fcall_pattern, *fcall_action,
          *fcall_guard, *fcall_fail_action
          );
    } elsif ( $buf =~ /_ASE_GV_BEGIN_/ ) {
        $total_gv_changes++;
        process_gv($INFILE, $total_gv_changes, *gv_oldname,
*gv_newname);
    } elsif ( $buf =~ /_ASE_DT_BEGIN_/ ) {
        $total_dt_changes++;
        process_dt($INFILE, $total_dt_changes, *dt_old_struct,
*dt_old_member,
              *dt_new_struct, *dt_new_member);
    } elsif ( $buf =~ /_ASE_IH_BEGIN_/ ) {
        $total_ih_changes++;
        process_ih($INFILE, $total_ih_changes, *ih_old_name,
*ih_new_name);
    } elsif ( $buf =~ /_ASE_SN_BEGIN_/ ) {
        $total_sn_changes++;
        process_sn($INFILE, $total_sn_changes, *sn_old_name,
*sn_new_name);
    }
     }
}


sub process_ase {
    local($INFILE,
     $total_changes,
     *fcall_name, *fcall_nargs, *fcall_pattern, *fcall_action,
     *fcall_guard, *fcall_fail_action
     ) = @_;# subroutine parameter
    local($buf) = "";# local variable

    while ($buf = <$INFILE>) {
   if ($buf =~ /_ASE_FCALL_END_/) {
       return;
   } elsif ( $buf =~ /FNAME\s+=\s+(.*)$/ ) {
       $fcall_name[$total_changes] = $1;
   } elsif ( $buf =~ /ARGS\s+=\s+(.*)$/ ) {
       $fcall_nargs[$total_changes] = $1;
   } elsif ( $buf =~ /PATTERN\s+=\s+(.*)$/ ) {
       $fcall_pattern[$total_changes] = $1;
```

```
    } elsif ( $buf =~ /GUARD\s+=\s+(.*)$/ ) {
        $fcall_guard[$total_changes] = $1;

    # this case must precede the next case, a less general case
    } elsif ( $buf =~ /FAIL_ACTION\s+=\s+(.*)$/ ) {
        $fcall_fail_action[$total_changes] = $1;
    } elsif ( $buf =~ /ACTION\s+=\s+(.*)$/ ) {
        $fcall_action[$total_changes] = $1;
    } else {
        printf("ase ignoring: $buf.\n");
    }
     }
}


sub process_gv {
    local($INFILE,
     $total_gv_changes, *gv_oldname, *gv_newname
     ) = @_;# subroutine parameter
    local($buf) = "";# local variable

    while ($buf = <$INFILE>) {
   if ($buf =~ /_ASE_GV_END_/) {
       defined($gv_oldname[$total_gv_changes]) || die;
       defined($gv_newname[$total_gv_changes]) || die;
       return;
   } elsif ( $buf =~ /GV_OLDNAME\s+=\s+(.*)$/ ) {
       $gv_oldname[$total_gv_changes] = '"' . $1 . '"';
   } elsif ( $buf =~ /GV_NEWNAME\s+=\s+(.*)$/ ) {
       $gv_newname[$total_gv_changes] = '"' . $1 . '"';
   } else {
       printf("process_gv ignored: $buf.\n");
   }
    }
}


sub process_dt {
    local($INFILE,
     $total_dt_changes, *dt_old_struct, *dt_old_member,
*dt_new_struct, *dt_new_member
     ) = @_;# subroutine parameter
    local($buf) = "";# local variable

    while ($buf = <$INFILE>) {
   if ($buf =~ /_ASE_DT_END_/) {
       defined($dt_old_struct[$total_dt_changes]) || die;
       # defined($dt_new_struct[$total_dt_changes]) || die;
       defined($dt_old_member[$total_dt_changes]) || die;
       defined($dt_new_member[$total_dt_changes]) || die;
       return;
```

```
    } elsif ( $buf =~ /DT_OLD_STRUCT\s+=\s+("[_A-Za-z]+").*$/ ) {
        $dt_old_struct[$total_dt_changes] = $1;
    } elsif ( $buf =~ /DT_NEW_STRUCT\s+=\s+("[_A-Za-z]+").*$/ ) {
        $dt_new_struct[$total_dt_changes] = $1;
    } elsif ( $buf =~ /DT_OLD_MEMBER\s+=\s+("[_A-Za-z]+").*$/ ) {
        $dt_old_member[$total_dt_changes] = $1;
    } elsif ( $buf =~ /DT_NEW_MEMBER\s+=\s+("[_A-Za-z]+").*$/ ) {
        $dt_new_member[$total_dt_changes] = $1;
    } else {
        printf("process_dt ignored: $buf.\n");
    }
     }
}


sub process_ih {
     local($INFILE,
      $total_ih_changes, *ih_old_name, *ih_new_name
      ) = @_;# subroutine parameter
     local($buf) = "";# local variable

     while ($buf = <$INFILE>) {
    if ($buf =~ /_ASE_IH_END_/) {
        defined($ih_old_name[$total_ih_changes]) || die;
        defined($ih_new_name[$total_ih_changes]) || die;
        return;
    } elsif ( $buf =~ /IH_OLDNAME\s+=\s+(.*)$/ ) {
        $ih_old_name[$total_ih_changes] = '"' . $1 . '"';
    } elsif ( $buf =~ /IH_NEWNAME\s+=\s+(.*)$/ ) {
        $ih_new_name[$total_ih_changes] = '"' . $1 . '"';
    } else {
        printf("process_ih ignored: $buf.\n");
    }
     }
}


sub process_sn {
     local($INFILE,
      $total_sn_changes, *sn_old_name, *sn_new_name
      ) = @_;# subroutine parameter
     local($buf) = "";# local variable

     while ($buf = <$INFILE>) {
    if ($buf =~ /_ASE_SN_END_/) {
        defined($sn_old_name[$total_sn_changes]) || die;
        defined($sn_new_name[$total_sn_changes]) || die;
        return;
    } elsif ( $buf =~ /SN_OLDNAME\s+=\s+(.*)$/ ) {
        $sn_old_name[$total_sn_changes] = '"' . $1 . '"';
    } elsif ( $buf =~ /SN_NEWNAME\s+=\s+(.*)$/ ) {
```

```
        $sn_new_name[$total_sn_changes] = '"' . $1 . '"';
    } else {
        printf("process_sn ignored: $buf.\n");
    }
     }
}


sub modify_parser {
    local($INFILE, $OUTFILE,
     $total_changes, *fcall_name, *fcall_nargs, *fcall_pattern,
*fcall_action,
     *fcall_guard, *fcall_fail_action,
     $total_gv_changes, *gv_oldname, *gv_newname,
     $total_dt_changes, *dt_old_struct, *dt_old_member,
*dt_new_struct, *dt_new_member,
     $total_ih_changes, *ih_old_name, *ih_new_name,
     $total_sn_changes, *sn_old_name, *sn_new_name
     ) = @_;
    local($buf) = "";
    local($i);
    local(%count);

    while ($buf = <$INFILE>) {
   $buf =~ s/%/%%/g;# necessary for printing %, oh well.
   printf($OUTFILE $buf);
   if ($buf =~ /_ASE_NAMES_/) {
       # remove duplicates, perl book, p.254
       for (@fcall_name) {
      $count{$_}++;
       }
       for (sort keys %count) {
      printf($OUTFILE "const char* FNAME_$_ = \"$_\";\n");
       }
   } elsif ($buf =~ /_ASE_TOKENS_/) {
       foreach $i (0..$#fcall_name) {
      printf($OUTFILE "#token
FCALL_$fcall_name[$i]_$fcall_nargs[$i]\n");
       }
   } elsif ($buf =~ /_ASE_FCALL_BEGIN_/) {
       foreach $i (0..$#fcall_name) {
      printf($OUTFILE
             "if ((strcmp(fn->getText(), FNAME_$fcall_name[$i]) ==
0) && (numArgs == $fcall_nargs[$i])) {\n");
      printf($OUTFILE
              "\t#0 = #(#[FCALL_$fcall_name[$i]_$fcall_nargs[$i]],
#[fn], #[lp], #fa, #[rp]);\n");
             printf($OUTFILE
               "} else ");
       }
   } elsif ($buf =~ /_ASE_GV_BEGIN_/) {
```

```
        foreach $i (0..$#gv_oldname) {
       printf($OUTFILE "if ((#id->IsGlobal()) && (strcmp(#id-
>getText(), $gv_oldname[$i])==0)) {\n");
       printf($OUTFILE "\t#id->SetText($gv_newname[$i]);\n}
else\n");
        }
    } elsif ($buf =~ /_ASE_DT_BEGIN_/) {
        foreach $i (0..$#dt_old_struct) {
       printf($OUTFILE "if ((strcmp(\$pType->GetName(),
$dt_old_struct[$i]) == 0) && (strcmp(id2->getText(),
$dt_old_member[$i]) == 0)) {\n");
       printf($OUTFILE "\tid2->setText($dt_new_member[$i]);\n}
else\n");
        }
    } elsif ($buf =~ /_ASE_IH_BEGIN_/) {
        foreach $i (0..$#ih_old_name) {
       printf($OUTFILE "if (strcmp(st->getText(),
$ih_old_name[$i])==0) {\n");
       printf($OUTFILE "\tst->setText($ih_new_name[$i]);\n}
else\n");
        }
    } elsif ($buf =~ /_ASE_SN_BEGIN_/) {
        foreach $i (0..$#sn_old_name) {
       printf($OUTFILE "if (strcmp(id->getText(),
$sn_old_name[$i])==0) {\n");
       printf($OUTFILE "\tid->setText($sn_new_name[$i]);\n}
else\n");
        }
    }
     }
}


sub modify_transformer {
    local($INFILE, $OUTFILE, $total_changes,
     *fcall_name, *fcall_nargs, *fcall_pattern, *fcall_action,
     *fcall_guard, *fcall_fail_action) = @_;
    local($buf) = "";
    local($i);

    while ($buf = <$INFILE>) {
   printf($OUTFILE $buf);
   if ($buf =~ /_ASE_FCALL_BEGIN_/) {
        foreach $i (0..$total_changes) {
      printf($OUTFILE "| ! $fcall_pattern[$i]\n");
      printf($OUTFILE "<<\n");
      if (!defined($fcall_guard[$i])) { # may need a better sol.
          printf($OUTFILE "$fcall_action[$i]\n");
      } else {
          printf($OUTFILE "if (%s) {\n", $fcall_guard[$i]);
          printf($OUTFILE "$fcall_action[$i]\n");
```

```
            printf($OUTFILE "} else {\n");
            printf($OUTFILE "$fcall_fail_action[$i]\n");
            printf($OUTFILE "}\n");
        }
        printf($OUTFILE ">>\n");
         }
     }
      }
  }
```

Appendix B

Purdue Compiler Construction Tool Set

The Purdue Compiler Construction Tool Set (PCCTS) consists of three parts:

1. a lexer: DFA-based Lexical Analyzer Generator (DLG).
2. a parser: ANother Tool for Language Recognition (ANTLR).
3. a tree transformer: Sorcerer.

Actually, PCCTS contained only a lexer and a parser in the early days. Recently, Sorcerer has been incorporated into PCCTS. In PCCTS, there is one single description file for lexing and parsing, thus it is convenient to describe them together. The details of PCCTS are described in the rest of this appendix.

B.1 ANTLR and DLG

PCCTS is similar to a highly integrated version of YACC and Lex; where ANTLR (ANother Tool for Language Recognition) corresponds to YACC and DLG (DFA-based Lexical analyzer Generator) functions like Lex. PCCTS grammars contain specifications for lexical and syntactic analysis with selective backtracking ("infinite lookahead"), semantic predicates, intermediate-form construction and sophisticated parser exception handling. Rules may employ Extended BNF (EBNF) grammar constructs and may define parameters, return values and local variables. Languages described in PCCTS are recognized via predicated-LL(k) parsers constructed in pure, human-readable, C/C++ code; the C++ programming interface is very good. The documentation is quite complete, but distributed over an original manual plus multiple release notes.

PCCTS can be obtained by ftp (ftp pub/pccts/* from ftp.parr-research.com) and more information can also be obtained from the USENIX newsgroup group - comp.compilers.tools.pccts.

B.2 Sorcerer

Sorcerer is more suitable for the class of translation problems lying between those solved by code-generator generators and by full source-to-source translator generators. Sorcerer generates simple, flexible, top-down, tree parsers that, in contrast to code-generators, may execute actions at any point during a tree walk. Sorcerer accepts extended BNF notation, allows predicates to direct the tree walk with semantic and syntactic context information, and does not rely on any particular intermediate form, parser generator, or other pre-existing application.

Sorcerer can be obtained by ftp (pub/pccts/sorcerer/* from marvin.ecn.purdue.edu) and more information can also be obtained from the USENIX newsgroup - comp.compilers.tools.pccts.

Vita

Kingsum Chow was born in Shanghai, China. Before he learned how to walk, he travelled to Hong Kong with his family. At the end of his secondary school years in Hong Kong, he received all 5 A's in his Advanced Level General Certification Examinations. He was then awarded a scholarship and studied Chemistry at the National University of Singapore for 3 years. Being the best first year student in the Science Faculty among more than 800 students, he received the Sugar Industry of Singapore Book Prize. He graduated with First Class Honors and received the Singapore National Institute of Chemistry Gold Medal for being the best Chemistry student. After working for 2 years as a scientific officer, he continued his education in Chemistry at the University of Washington. He changed his major to Computer Science after receiving a M.S. degree in Chemistry. In 1993, he received a M.S. degree in Computer Science. In 1996, he completed his Ph.D. degree at the University of Washington and joined Intel Corporation.

Chow enjoyed teaching very much and he received excellent evaluations during the times while he was TA for both Chemistry and Computer Science classes. In 1995, he received the Bob Bandes award for his effort as a teaching assistant. In summer 1995, he was an instructor for CSE 143 and received an ovation at the end of his last lecture.

Besides doing research and teaching, Chow enjoyed the game of contract bridge the most. He was a member of the Singapore Youth team and represented Singapore in four Association for South East Asian Nations (ASEAN) Championships. He quickly became a member of the American Contract Bridge League (ACBL) after he moved to the USA. He won a number of regional events and made it to the 6th place of North American Open Pairs (NAOP, Flight C) in Phoenix, Arizona, April 1995. He made Life Master in August, 1995 for his overall achievements in bridge. He stopped playing bridge after that and focused on getting this thesis done.

Chow was married to Hoon Moey Goh and they had their first wonderful daughter, Ida, on February 16, 1996.