Semi-automatic Update of Applications in Response to Library Changes

Kingsum Chow and David Notkin
Department of Computer Science & Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350, USA

{kingsum,notkin}@cs.washington.edu

Technical Report UW-CSE 96-03-01

Abstract

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 paper introduces an approach and a toolset 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.

Keywords: Software evolution, software maintenance, software libraries, asynchronous software evolution, asynchronous software maintenance, program restructuring.

1 Introduction

Libraries provide leverage in large part because they are used by many applications. As Parnas [Parnas 1972 & 1979], 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's syntax or semantics, applications can use updated libraries simply by importing and linking the new version.

In principle, this is a great idea. However, David 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]

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 they make to indicate what changes must be made to update the applications to accommodate the updated library. We then provide a set of tools that use these annotations to semi-automatically update any applications that use the updated library. The idea is that the small additional work by the (one) library maintainer can provide leverage to the many application maintainers. This is illustrated in Figure 1.

In this paper, we address this approach to reducing some of the costs of updating applications when a library upon which the applications depend changes in unstable ways: that is, the syntax and/or the semantics of the library change. Section 2 describes an example, taken from a programming support library for Borland C++, that illustrates the general problem. Section 3 describes our approach to reducing costs, which consists of two parts:

a set of rules that describe the changes from the earlier version of the library to the later version; and a set of tools that apply these rules to applications to update them to properly use the updated library. The tools either make proper updates or else identify where they cannot decided how to handle the library's changes. Section 4 discusses the status of our approach, including its strengths and weaknesses. Section 5 places the work in the greater context of software maintenance, software restructuring, etc. Section 6 discusses other benefits of our approach. Section 7 summarizes our work and suggests future research areas.

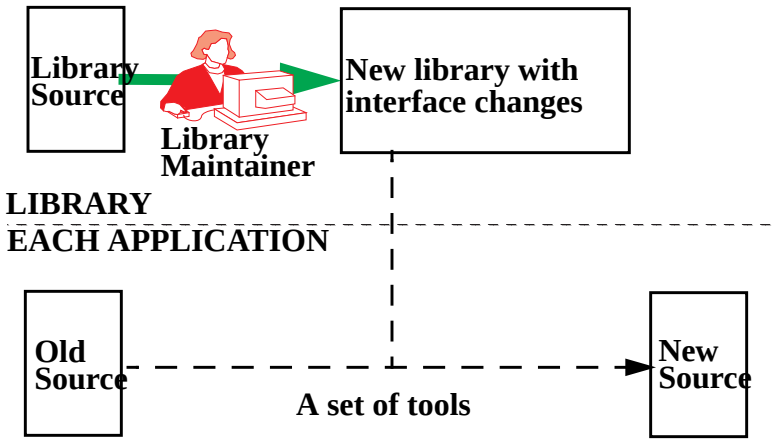

Figure 1. Overview of our approach

2 An example of an interface change

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 some interface changes involving overloaded functions and default arguments. Although the example is simple, Section 4.1 describes a much broader set of changes that can be handled by our approach and tools; limitations on the kinds of changes that can be handled are addressed in Section 4.2

1. The example is modified slightly to use the C language syntax to simplify the presentation.

The example addresses the assign function, which is used to copy values from one string to another. The original library function was:

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

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. Later, the library maintainers decided to change the function to:

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

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.

For example, for the new version of the library:

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

After executing this code, s2 should contain “cde”. (Remember, in C strings are arrays of characters and indexing takes place from the zeroth character.)

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

In the first scenario, throughout the application all calls to the original assign function use only the first two parameters:

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

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, the application contains some calls to assign 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);
```

However, this time, the interpretation of 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 assign function in the library. This is hardly ever what the application maintainer wants.

In the third scenario, the application has the same kinds of calls to assign as in the second scenario, but the application maintainer is now aware of the library change. In this case, the maintainer just needs to insert a third argument of 0 to the caller. This will make the application code behave the same as before.

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

These scenarios identify several problems. First, if the application maintainer is unaware of the changes to the library (perhaps because they didn’t read the provided documentation carefully), their application will almost surely 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 may have 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 handing library updates.

This simple example is characteristic of many other kinds of changes that take place when libraries are modified. (See Section 4.1 for a description of other classes of changes we have identified as common.)

Effective solutions to this problem should have two 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, as much as possible, relieve the application maintainers 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].

The next section sketches the basic approach we take, which is intended to satisfy these properties. It also provides technical details about the languages and tools we use to help automate such changes.

3 The design of our approach

One approach taken to similar problems is, roughly, to change the application automatically when the library is updated [Griswold & Notkin 1993][Opdyke 1992]. This approach requires that the full source code for the library and application be available at one time and on one computer, so that the needed changes can be propagated from the library to the application. But this approach does not transfer to situations in which the library and the applications are not on the same computer, since the changes cannot be propagated directly.

Our approach instead generates tools based on the library modifications, and distributes those tools along with the updated libraries. When an application maintainer decides to import the updated library, the supplied tools can also be run to update the application. To

allow tools to be generated, 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 2 shows a more detailed view of our approach, which we describe below.

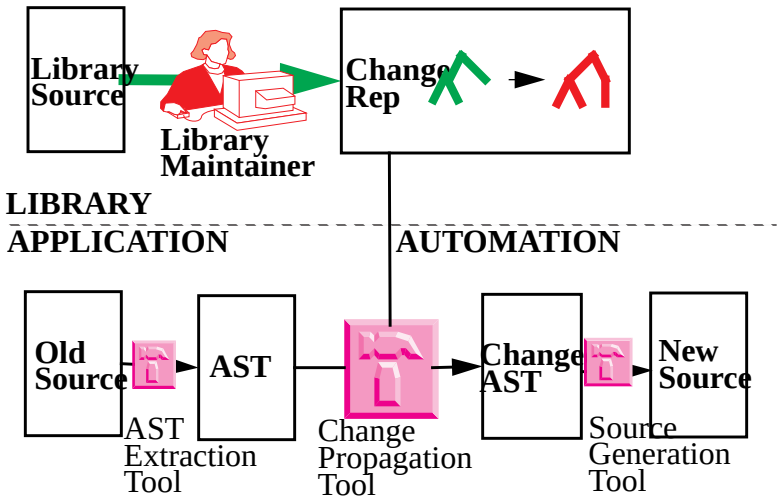

Figure 2. Overview of our design

Our approach requires 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. ".h" files for C/C++ or ".spc" 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. This 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.

These files are then used to build a parser and a syntax tree transformer. The parser constructs a syntax tree for the application program and then performs token level code transformation. 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 emitted at the end. The stages of our approach are summarized in the data

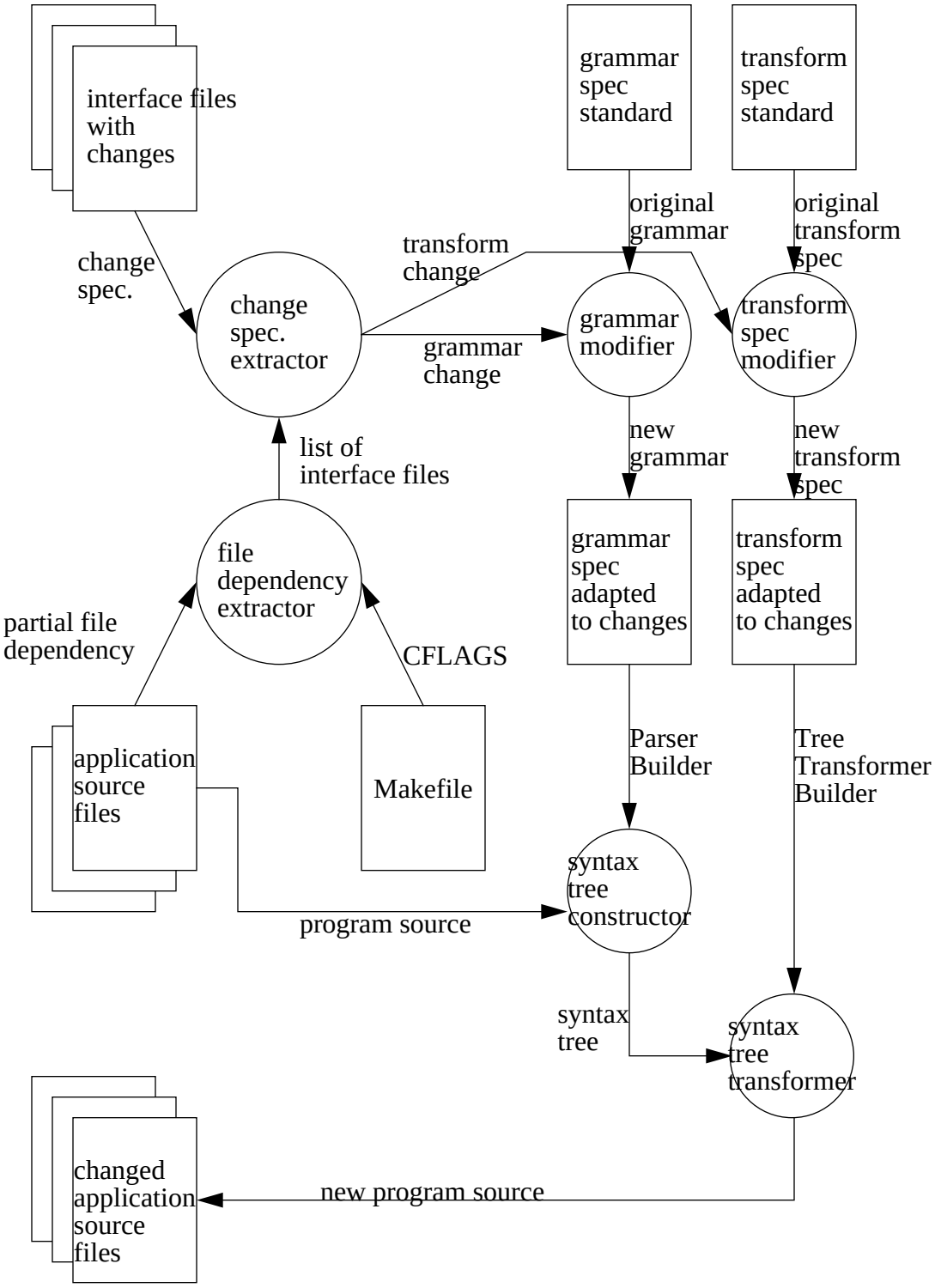

Figure 3. The data flow diagram for our design

flow diagram in Figure 3. (We do not address every part of the figure in this paper, due to space restrictions.)

In Figure 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, e.g. 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.

Although the change specification is released as part of the library code, the application maintainers still have the final decision as 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 4.

```
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 4. A Makefile example showing the new update target.

As we will describe in Section 5, this is similar to work in program restructuring, except that the temporal and geographical distance between the library and application maintainers introduces significant technical problems, e.g. change specification, representation and propagation.

3.1 The language for describing library changes

The library maintainer needs to specify the adaptation code for the interface change. Something like the following would be sufficient for the example presented earlier:

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 to the caller.

In our current prototype, the actual change specification in the ".h" file would be specified in the following way (Figure 5).

```
/*
 * _ASE_BEGIN_
 * FNAME = assign
 * PATTERN = #(fc:FCALL_assign id:ID lp:L_PAREN arg1:gen_expr
co1:COMMA arg2:gen_expr co2:COMMA arg3:gen_expr rp:R_PAREN)
 * ACTION = #gen_logical_expr = #(#fc, #id, #lp, #arg1, #co,
#arg2, #co, 0, #co2, #arg3, #rp);
 * _ASE_END_
 */
```

Figure 5. A change specification example showing the addition of a third argument.

The details of the change specification language are not important here,[1] but the basics are clear. A function name is defined (FNAME, in this case), which describes the function that the library maintainer has modified. The PATTERN describes a syntactic structure to be matched against any function invocation that matches the FNAME. In this case, it is basically matching those invocations of assign that have exactly three arguments (as determined by matching two commas). If and only if that PATTERN matches, the tool applies the ACTION code. In this case, the action inserts the added 0, as described above.

In the case of a typed language, it may also be important to check the types of the arguments in additional to the number of arguments. In this case, a stricter condition can be added:

if the first argument is "char" and the second is "int".*

An example is shown in Figure 6. In this example, a guard command specifies the semantic condition that must be true before a transformation takes place.

1. As we show in the next section, the specific syntax is motivated by the language processing system that we used to implement our toolset. Details of the specification are found elsewhere [Chow 1996].

```
/*
 * _ASE_BEGIN_
 * FNAME = foo
 * PATTERN = #(fc:FCALL_foo id:ID lp:L_PAREN arg1:gen_expr
co1:COMMA arg2:gen_expr rp:R_PAREN)
 * GUARD = (arg1->Same(Ptr(charType)) && arg2->Same(intType))
 * ACTION = #gen_logical_expr = #(#fc, #id, #lp, #arg1, #co,
#arg2, #co, 0, #co2, #arg3, #rp);
 * _ASE_END_
 */
```

Figure 6. A change specification example showing the semantic checking.

Sometimes, the change may even depend on the actual value of an argument. Our matching language is rich enough to capture these and other kinds of patterns. However, since our approach is based on lexical, syntactic and semantic analysis, we can only detect values that can be determined at compile time.

Test programs that use the old library interface can also be used to test the interface change specification that is produced. This allows the library maintainer to gain confidence about the interface changes by simply using existing test programs.

3.2 The implementation of our semi-automatic tool

For our current prototype implementation, we use the readily available language processing tools, 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 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 are very portable and versatile. The change specification is written in patterns that are similar to PCCTS/Sorcerer so that little conversion needs to be made for them to be used by the other tools in our approach. In particular, Sorcerer supports tree-to-tree transformations, 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 nice error messages.

A comparison between PCCTS and lex/yacc is beyond the scope of this paper, however.

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 conveniently done during using PCCTS but changes that depend on a sequence of tokens, i.e., tree patterns, are better done using Sorcerer.

4 The status of our approach

In our approach, we provide a mechanism for the library maintainer to pass interface change specification and adaptation rules to the application maintainers. The mechanism that we employ does not have any restriction on actual kinds of the changes. It instead depends on the grammar of the programming language and the change specification using that grammar. To demonstrate our approach, we wrote a grammar for the C language with two interesting extensions: function overloading and default arguments. We added these two extensions to handle richer, more useful, sets of interface changes. We then wrote interface change specifications based on this grammar.

4.1 Current status of our implementation

Our current prototype system can handle a number of common interface changes such as changing the names of include files, variables and functions and changing the order of

parameters. Table 1 summarizes the various classes of changes that can be handled using our approach. A more detailed study of patterns of interface changes can be found in [Chow 1996].

Table 1 Classes of supported changes

Kind of Change	**Current status**
add and remove include files or change their names.	implemented
add, remove or rename function names.	implemented
move a function from an include file to another.	designed, not implemented
add, remove and reorder function parameters.	implemented
change default arguments to non default.	implemented
change the return type of a function.	implemented
change the type of a function parameter.	implemented for common cases[a]
change the meaning of an in-coming function parameter.	implemented
remove, add, rename and change a struct field.	implemented
remove, add, rename and change a global variable.	implemented

a. Complex cases involving overloading are designed but not yet implemented.

In our prototype system, the library maintainer needs to provide a standard release of the language grammar and transformation grammar. Of course, the interface specification changes must also be provided in the include files.

The application maintainers only need to add a new target, say update, in the Makefile (see Figure 4). The new target simply runs our tool to preprocess the source files. This way, the library clients just need to type

```
make update
```

to apply our tool to all the source files. After that, the library maintainers may examine the changes made or proposed by the tool and proceed to making their projects if no change in their source code is required or if they are satisfied with the changes that are made to their source.

Our prototype system will not alter any white space or indentation unless a piece of code is changed. This will make the look and feel of the program the same as before and thus

the application maintainers only need to focus on changes, if any, that adapt to the new interface. The action description determines how modified code appears in the updated application.

4.2 Limitations

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 not handle changes that are dependent on two places of the source code. For example, if foo() does something that bar() doesn't need to be called, then when we see bar() in the source code, we cannot tell if foo() is already in used. A change dependency on multiple locations of the source code is not easy to handle but fortunately, this does not happen too often at all. 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 an output parameter of a function. In this case, the declaration of the variable into which the output parameter will be written generally has to be changed, too. But we cannot yet handle these two coordinated but physically separate changes. This specific case is more promising than the general case, since we could apply knowledge about type-casting or the library maintainer could provide additional type conversion routines to aid in the application update.

In addition, there are a number of language specific features that our implementation has not been able to handled. The common problem among these language features is aliasing. For example, the alias of a variable to another variable when change is based on or a function pointer that may point to any real functions and some of which may have some change specification. Also, the introduction of inheritance class hierarchy and virtual member functions make some C++ changes hard to handle. These are interesting issues in

future work. However, even in these cases, the application maintainers are still no worse than, if not better than before.

Because of the employment of various tool for change propagation and transformation, a small amount of disk space (~10 MB) may need to be reserved by each library client for upgrading their source. However, the same tool can be used for source code upgrade of other software applications using other languages and interfaces, with the other grammar and transformation specifications.

5 Related work

One of the activities that is often done before making semantic changes to a program is to restructure it first. Restructuring is often necessary due to the natural degradation of software structure [Lehman 80] after a series of modification. The restructuring approach [Griswold & Notkin 93] focuses 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 this approach and our work however. First, the entire program is presumably visible and changeable by the restructuring tool while parts of the program are either not visible or not changeable in my proposal. Second, some structural changes in the restructuring approach need not be dealt with by our change adaptation tool because they do not result in interface changes.

In fact, there are a number of researchers [Johnson & Opdyke 1993], [Casais 1992], [Bergstein 1991] and [Lieberherr & Xiao 1993] working along the same line as [Griswold & Notkin 1993] i.e., maintaining changes in various different ways, not merely interface changes, within a program. However, in this kind of closed system, it is indeed required that all source code is accessible and modifiable. Typically, the programmer and the maintainer are the same person in a closed system.

In the industry, however, some sort of a 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 also make more assumptions about the source and thus make themselves very restrictive and will work only in a narrow domain. In fact, these transformation scripts can be generated by a more general mechanism such as our implementation.

6 Discussion

6.1 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 mark certain tree nodes (derived from tokens) to make the identification of a tree pattern easier in the later stage. However, in this case, there will be an added overhead of the regeneration of the parser in each transformation. A balance between these two concerns will be studied in detail in the future.

6.2 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 and suggestions or corrections will be made at that time. Thus, naturally, 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.

6.3 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, which may be discouraged, is the size of the ".h" files which at a minimal is definitely distraction to new users, and it also subjects them to misuse of the old interface. Keeping the old interface also means keeping bigger 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.

In some sense our model is similar to Eiffel [Meyer 1992]. The Eiffel keyword “obsolete” can be used to indicate something is obsolete. However, the Eiffel language itself does not support transformation of client code based on this keyword.

7 Summary

In our prototype system, the library maintainer needs to pay the one time cost of providing the grammar and the transformation specification. And 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 are shifted from the burden of thousands if not millions of application maintainers to the library maintainers. In fact, this is the biggest strength of our approach and implementation, to make the thousands of application maintainers happier when the library interface is indeed changed. Another added advantage of this requirement for the library maintainers to specify the changes is to force them to think seriously before making interface changes.

Our implementation assumes the existence of interface files that contain the interface specification. This realistic assumption allows us to build a prototype toolset to demonstration 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.

When a library interface is indeed changed, the library users need to be notified of those changes. The changes and suggested way to adapt the source 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. However, it is not clear whether this kind of information can actually reach the clients. Furthermore, it is not clear if the clients are always very keen to read README files, not to mention that some README

files are not easily understandable by most library users. Even if they do, the updating activity is still manual in addition to the requirement that the clients need to understand the sometimes highly cryptic suggestions in some README files. In addition, README files sometimes do not tell the whole truth. Indeed, the README file itself can be out of date, because there is no other formal process to verify the content of the README files.

The principle of the semi-automatic update of application in response to library changes is to enable the library maintainers to propagate necessary interface changes to the users' code. The small extra work done by the library maintainers is a very small price to pay for the thousands or millions of library users.

In this paper, 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.

Within the area of library interface changes, we have done a study on the pattern of library changes [Chow 1996] to gain confidence in the usefulness of our tool. Future work will be required to evaluate the effectiveness of our model and implementation. For example, some of the process may not require a complete syntax parsing of the program. Light weight lexical approaches [Murphy & Notkin 1995] can be applied when appropriate, for example. Also, the syntax of the change specification may need to be more friendly and versatile. As discussed earlier, future work should also include handling certain language features that make automatic change propagation difficult. A classification of these problems and general guidelines to library maintainers in terms of writing change specification would be useful.

In summary, we regard adapting source code to library interface changes as one of the biggest activity in asynchronous software evolution [Chow 1995]. In future, we will look at other areas of asynchronous software evolution [Chow & Notkin 1996].

References

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

[Borland 1993a] Borland’s README file for release 4.0.

[Borland 1993b] Borland ObjectWindows for C++ Programmer’s Guide 2.0. 1993.

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

[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. Also published as Technical Report Number CS95-418, Computer Science and Engineering, University of California, San Diego.

[Chow 1996] Kingsum Chow, Ph.D. dissertation in preparation, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, USA (1996).

[Chow & Notkin 1986] Kingsum Chow and David Notkin, “Asynchronous Software Evolution” Asia-Pacific Workshop on Software Engineering Research, March 21, 1996, Hong Kong.

[Collofello & Buck 1987] Collofello, J. S., and Buck, J. J, “Software quality assurance for maintenance” IEEE Computer. (Sept. 1987), 46-51.

[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).

[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).

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

[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).

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

[Meyer 1992] Meyer, Bertrand. Eiffel: the language. Prentice Hall (1992).

[Microsoft 1994] MFC Migration Guide. 1994.

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

[Murphy & Notkin 1995] Gail C. Murphy and David Notkin, “Lightweight Source Model Extraction” In Proceedings of the Third ACM Symposium on the Foundations of Software Engineering (FSE ‘95).

[Opdyke 1992] William F. Opdyke, “Refactoring Object-Oriented Frameworks” Ph.D. Thesis. 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] D. 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).

[Parr 1994] Terence J. Parr, “An Overview of SORCERER: A Simple Tree-Parser Generator” International Conference on Compiler Construction (April 1994).

[Parr 1995] Terence J. Parr, Language Translation Using PCCTS and C++ (A Reference Guide).

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

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

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