Reverse Engineering Interface Protocols for Comprehension of Large C++ Libraries during Code Evolution Tasks

Edward B. Duffy, Jason O. Hallstrom and Brian A. Malloy
Computer Science Department
Clemson University
Clemson, SC 29634, USA

E-mail: {eduffy,jasonoh,malloy}@cs.clemson.edu

Abstract

In this paper, we describe our trace monitoring system and a methodology for reverse engineering interface protocols to capture the sequence of method invocations for large C++ applications. To evaluate our system, we present a case study using the networking library in the Mozilla Internet Application Suite, and three Mozilla applications: Firefox, Thunderbird and Sunbird. We use trace monitoring of the library to capture the interface protocols for the classes in the library and our preliminary results support our assumption that interface protocols follow a specific pattern and that these patterns can facilitate comprehension of the underlying interactions among the classes in the system.

1. Introduction

The process of software maintenance, including modification, refactoring, and usage of complex object oriented systems, requires knowledge about the system under study and, in particular, about the interactions among the classes and components of the system. However, software artifacts that describe these interactions are frequently unavailable and for large, open-source C++ applications, they are virtually nonexistent. Thus, much of the research in software engineering has focused on the development of tools to automatically generate information to improve comprehension of the application under study, and thereby facilitate the maintenance effort.

In this paper, we present Hylian, our system for code regeneration and trace monitoring in large C++ applications. Hylian uses an augmented form of the GNU *gcc* parser to output parse trees in XML format, permit modification of the parse trees, and to regenerate a modified version of the source code. We used an earlier version of Hylian to reverse engineer a grammar for the *gcc* C++ parser, version 4.0.0 [5]; the current version permits modification of the parse tree to extract trace information of a C++ application under study.

To demonstrate the utility of Hylian, we generate trace information, extract the sequence of method invocations, and generate regular expression representations of the interface protocols for *Necko*, a large networking library written in C++ [9]. To exercise the *Necko* library, we use three large applications in the Mozilla Internet Application Suite: Firefox, Thunderbird and Sunbird, a browser, mailer and calendar application respectively [8]. We then choose classes in the *Necko* library that are used by the three applications and examine the regular expression representation of the interface protocols for the *Necko* library.

Our preliminary results support our assumption that the interface protocols for these classes follow a specific pattern and that these patterns can be used to facilitate comprehension of the class and to guide usage of the class by developers unfamiliar with the *Necko* library. Moreover, the regular expression representations of the interface protocol for a class can serve as examples, or templates, of correct usage of a class for a large library. We conjecture that these examples of library usage exemplify the comprehension model needed in the maintenance of large libraries and, together with other comprehension tools, can facilitate the maintenance effort.

In the next section, we review the terminology and concepts that we use in our work. In Section 3 we describe our trace monitoring methodology and its use in reverse engineering interface protocols. In Section 4 we present the case study described above and in Section 5 we review related research. In Section 6 we draw conclusions.

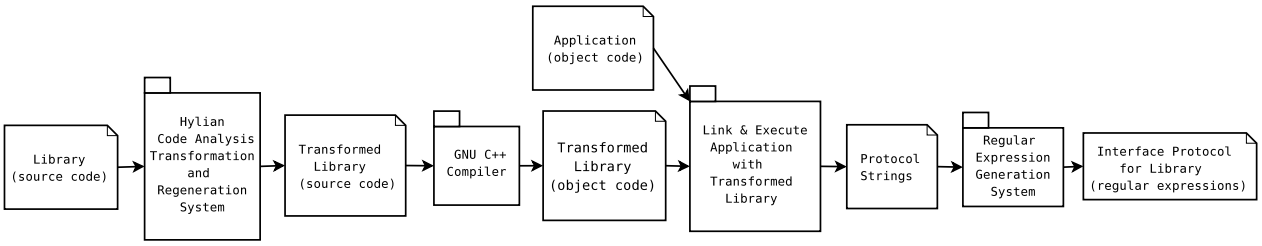

Figure 1. The Hylian **System.** *This figure illustrates* Hylian*, our code analysis and trace monitoring system.*

2. Terminology and Concepts

In this section, we review the terminology and concepts that we use in our work. In Section 2.1 we review grammars and parse trees, and in Section 2.2 we review the concept of *trace monitoring*.

2.1 Grammars, Parse Trees and ASGs

A grammar defines a language by specifying valid sequences of derivation steps that produce sequences of terminals, known as the *sentences* of the language. One procedure for using a grammar to derive a sentence in its language is to begin with the start symbol S and apply the production rules in some sequence until only non-terminals remain. This process defines a tree whose root is the start symbol, whose nodes are non-terminals and whose leaves are terminals. This tree is known as a *parse tree*; the process by which it is produced is known as *parsing*. Our system, Hylian, generates parse trees that we augment to monitor the execution of the libarary unders study.

2.2 Trace Monitoring

A *trace monitor* is a software artifact that observes the actions in a software system and, when certain activities are detected, the monitor executes some code of its own [2]. Trace monitors are especially useful for the detection or verification of runtime behavior. In our work, we use trace monitoring to detect class method invocations and to record a history of these invocations.

3. Protocol Extraction Methodology

In this section, we describe our system for monitoring the execution of an application and its corresponding library, and for reverse engineering interface protocols for C++ classes in the library. In the next section we present the *Hylian* system that we utilize and in Section 3.2 we describe our approach to regular expression generation.

3.1 Overview of the Hylian System

Figure 1 summarizes the flow of information through the system that we use. The source code for a C++ library is shown as input to Hylian, shown as a tabbed box to the left side of the figure. Hylian uses an augmented version of the GNU *gcc* parser, version 4.0.0, to generate a parse tree representation of the library code in XML format [5]. We produce transformed code to monitor the library by augmenting the parse trees with parse subtrees that contain code to trace the method invocations in the library; this phase is illustrated in Figure 1 as a tabbed box labeled Transformed Library. The Transformed Library is compiled into object code by the GNU C++ compiler, which is linked with the object code for the application that will utilize the library. The resulting executable, together with the input to the application, produces the Protocol Strings, which are then transformed into the Interface Protocol for the library, expressed as regular expressions.

3.2 Construction of Regular Expressions

We use an iterative algorithm to convert each protocol string into a regular expression and, for a protocol string of length n, our algorithm runs in $O(n^3)$ time. We first search the protocol string for recurring patterns of size 1, then recurring patterns of size 2, and continue the search, looking for recurring patterns of size $n/2$. For example, in searching for patterns of size 2, the string "abab" will be converted to $(ab)^+$. When the protocol string for each object is converted to a regular expression, we then use a *perl* package, Regexp::Assemble, to construct a single regular expression from the set of protocol strings generated by each instantiation of the class under consideration.

There is an abundance of research describing techniques to recover interface protocols using a finite state machine or regular expression representation [4, 7, 10, 11]. Our future work includes an investigation into these techniques to improve our protocol recovery process.

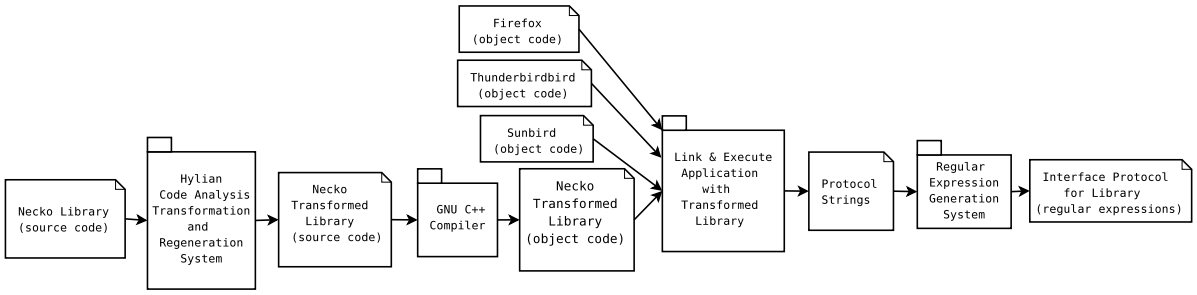

Figure 2. Study Summary. *This figure illustrates our study to evaluate our methodology for generating interface protocols for the Necko Networking Library utilized by the Mozilla Internet Application Suite.*

Application	Version	Units	Parse Tree
Necko	2.0a1pre	101	1,542,653
Firefox	3.0a8	1,262	27,857,127
Sunbird	0.6a1	1,586	33,023,605
Thunderbird	3.0a1pre	1,826	35,190,012

Table 1. Testsuite Statistics.

4. Case Study: The Mozilla Internet Suite

In previous sections of this paper, we described our approach for exploiting dynamic trace monitoring to capture the interface protocols of classes in an object-oriented system. In this section we describe a study that we conducted to evaluate our methodology and to show the utility of our approach. Our study involves an investigation into three applications included in the Mozilla Internet Application Suite: *Firefox*, a commonly used browser; *Thunderbird*, a mailer; and *Sunbird*, a calendar management application [8]. We use these applications to investigate usage of the *Necko* networking library included in the Mozilla Internet Application Suite.

Figure 2 illustrates our use of the Hylian analysis system, presented in Section 3, for transforming library code to provide dynamic trace monitoring and generate interface protocols for the classes in a library. In our study, we use the *Necko* library, listed on the left side of the figure, as input to our system, transform and regenerate the code, and use the GNU C++ compiler to generate object code for the transformed library, *Necko* Transformed Library, shown in the middle part of the figure. We then link the object code representation of the *Necko* Transformed Library, first with an object code representation of the *Firefox* application, and then with an object code representation of the *Thunderbird* application, and finally with an object code representation of the *Sunbird* application to produce protocol strings for each application. We then find the union of the protocol strings generated for *Necko* usage by each application to produce protocol strings for *Necko* classes, Protocol Strings, shown as a folded edge box (folded box) at the middle right of the figure. We then generate an Interface Protocol for Library, as regular expressions.

4.1 The Mozilla Application Suite

The statistics in Table 1 provide information about the version and size of the four applications that we study in this section. The first column lists the application, Application, the second column lists the version number, Version, the third column lists the number of compilation units, Units, and the fourth column lists the number of lines in the parse tree files, Parse Trees, for each application. The applications are ordered in the table by their parse tree size. For example *Necko*, the Mozilla networking library, is listed in the first row of the table and consists of version 2.0a1pre, 101 compilation units and 1,542,653 lines in the parse tree file. The largest application, *Thunderbird*, is listed on the last line of Table 1 and consists of version 3.0a1pre, 1,826 compilation units and 35,190,012 lines in the parse tree file.

4.2 Usage of the *Necko* Networking Library

Table 2 provides detailed information about the usage of *Necko* by the *Firefox*, *Thunderbird*, and *Sunbird* applications, including some information about the generated protocol strings for *Necko* classes. To exercise the *Firefox* browser, we visited medium-sized websites containing a number of images: *nytimes.com*, *washingtonpost.com*, *slashdot.org*, and *digg.com*. To exercise the *Thunderbird* application, we sent two emails consisting of plaintext and HTML markup, and we received an email message with a 1MB file attachment. To exercise the *Sunbird* calendar application, we synchronized the calendar component with three ICL formatted online calendars.

Application	Classes	Instantiations	Protocol Strings		
			Number	Longest String	Average Length
Firefox	76	14,055	14,055	59,070	31
Sunbird	55	1,297	1,297	17,578	68
Thunderbird	54	1,591	1,591	47,649	88

Table 2. Usage of *Necko* by the three Mozilla applications.

The first three columns of Table 2 list the application, Application, the classes used, Classes, and the number of classes instatiated, Instantiations, by the respective application. For example, the *Firefox* application used 76 of the 142 classes in *Necko*, which is 21 more classes than the *Sunbird* application used, and 22 more classes than the *Thunderbird* application used, even though *Firefox* is the smallest of the three applications, as measured by number of compilation units and number of lines of parse tree code (cf. Table 1). The sum of the instantiations in the third column (14,055+1,297+1,591) is 16,943, the total number of class instantiations for all three applications.

The final three columns in Table 2 list information about the protocol strings, Protocol Strings, generated by each of the three applications. The fourth column lists the number, Number, of strings generated and, since each object generates a protocol string, the number of strings is the same as the number of objects listed in the third column of the table. The fifth column lists the length of the longest string, Longest String, generated by each of the respective applications. The *Firefox* application generated the longest protocol string containing $59,070$ method invocations, which means that one of the instantiated classes made $59,070$ invocations of methods in the *Necko* library. The final column of Table 2 lists the average length, Average Length, of the protocol strings generated by the objects of the respective application.

4.3 Class Comprehension in Large Systems: Regular Expressions for Necko

Figure 3 contains two tables that describe information about the use of class nsDiskCacheInputStream in the Necko networking library. The table at the top of the figure contains four columns listing the name of the application, Application, the number of instantiations of nsDiskCacheInputStream, Instantiations, the number of unique sequences of method invocation strings for nsDiskCacheInputStream, Unique Sequences, and the interface protocol expressed as a regular expression, Interface Protocol. The first row of the table at the top of the figure lists information for *Firefox*, which created 166 instantiations of nsDiskCacheInputStream, generated 12 unique method call sequences that are summarized by the regular expression $abc^{+}ded$. The last two rows of the table list information for *Sunbird* and *Thuderbird*, which did not create any instantiations of class nsDiskCacheInputStream and did not use any of the methods.

The table at the bottom of Figure 3 has two columns where the first column, Mapping, specifies the mapping between letters and method names and the second column, Unique Sequences of Method Calls, lists the set of unique sequences of method invocations made by class instantiations of the *Firefox* application on the *Necko* networking library. For example, the first row of the second column of the table lists a sequence of 30 method invocations consisting of calls to ab, followed by a sequence of 25 calls to c, followed by calls to ded. Similarly, the ninth row of the second column of the table summarizes a sequence of 158 method invocations consisting of calls to ab, followed by a sequence of 153 calls to c, followed by calls to ded. Note that we use dots to indicate that some of the 153 calls to c have been elided from the ninth row of the table; however, all of the other sequences are illustrated precisely as they were generated by our test cases.

The regular expression representation of the 12 sequences is listed in the fourth column of the first row of the table at the top of Figure 3, $abc^{+}ded$. In lieu of documentation, UML case tool artifacts or other specification of the usage of a class, the reverse-engineered interface protocol can provide invaluable information about how a class in a large system is used, or may be used. For example, using the mapping, the typical usage of an instantiation of class nsDiskCacheInputStream consists of a call to the constructor, a call to AddRef, followed by one or more calls to Read, then calls to Close, Release and Close.

4.4 Comparison of Class Usage

Figure 4 illustrates a class instantiation history for those classes in the Necko library that were used by the Firefox, Thunderbird and Sunbird applications. Since we are interested in comparing usage by all three applications, the figure only lists those classes that were used more than ten times by each of the applications.

The three bars on the left side of the figure represent usage for the nsFileOutputStream class in the Necko library where the first bar indicates that Firefox created 44 in-

Use of nsDiskCacheInputStream by Three Applications			
Application	Instantiations	Unique Sequences	Interface Protocol
Firefox	166	12	$abc^{+}ded$
Sunbird	0	0	NA
Thunderbird	0	0	NA

Protocol Strings for *Firefox* Application	
Mapping	Unique Sequences of Method Calls
a → nsDiskCacheInputStream b → AddRef c → Read d → Close e → Release	01: abccccccccccccccccccccccded 02: abcccccccccded 03: abccccccccded 04: abccccccccccccded 05: abcccded 06: abccccccccccccccded 07: abcccccded 08: abccccccded 09: abccccccccccccccccccccccc . . . ccccccded 10: abccccded 11: abccded 12: abcccccded

Figure 3. Method Invocation Sequences for Class nsDiskCacheInputStream**.**

stances, Thunderbird created 37 instances and Sunbird create 28 instances of this class. The three bars in the middle of the figure represent usage for the nsBufferedStream class in the necko library where the first bar indicates that Firefox created 66 instances, Thunderbird created 68 instances and Sunbird create 56 instances of this class. The usage of this class is more evenly distributed than the usage of nsDiskCacheInputStream illustrated in Figure 3.

We found that the generated regular expressions for more heavily used classes in Necko can be less readable than the regular expression, $abc^{+}ded$ that we obtained for class nsDiskCacheInputStream. For example, Firefox usage of class nsDiskCacheInputStream generated the expression $abbcdbcb^{*}fbg^{*}ch^{*}(ggh)^{*}c^{*}e^{*}$, which may not be as useful as the one we obtained for class nsDiskCacheInputStream, or may indicate that nsDiskCacheInputStream has more complex usage patterns. Our ongoing work includes an investigation of some of the excellent regular expression generation algorithms in literature [4, 7, 10, 11].

5. Related Work

The generation of interface protocols can be accomplished using either static or dynamic analysis. The static approach has the advantage of finding all possible sequences of method invocations but must address the problems of pointer alias [6] and infeasible paths. Moreover, the static approach may provide interface protocols that are irrelevant to the application under consideration, as we have seen in Section 4.4. the regular expressions can be overly complicated. The dynamic approach has the advantage of providing only those sequences of method invocations that are relevant to the application under consideration and does not suffer the problems of alias or infeasible path resolution. The related work that we review in this section employs the dynamic approach to protocol recovery.

Cornelissen and Moonen describe a technique for addressing the scalability problem in extracting information from execution traces of function calls in Java programs [4]. They observe that certain event sequences are repetitive, where the repitition typically results from the occurrence of method invocations within loops. Their summarization technique entails the use of *similarity matrices* to visualize the repetitive method invocation sequences in the trace. Recurring sequences of method invocations appear as patterns in the matrix. However, the sequences appear in the matrix as diagonal lines and this abstraction results in the loss of information about the identity of the methods in the recurring sequences. The approach that we describe in this paper entails summarizing the recurring sequences of method invocations as regular expressions, which has the advantage of maintaining the identity of the methods involved in the recurring sequences.

Walkinshaw et al. describe the construction of state machines from user supplied scenarios and execution traces of Java programs [10]. They use the scenarios from the user, the execution traces and the QSM state-merging approach to interactively generate a state machine of the system. Our approach differs from that of Walkinshaw et al. in that our technique is fully automated and does not require user supplied scenarios.

Butkevich et al. describe an extension to the Java programming language to facilitate static conformance checking and dynamic debugging of object protocols [3]. *Object protocols* are sequencing constraints on the order in which methods in a Java application may be invoked. In their

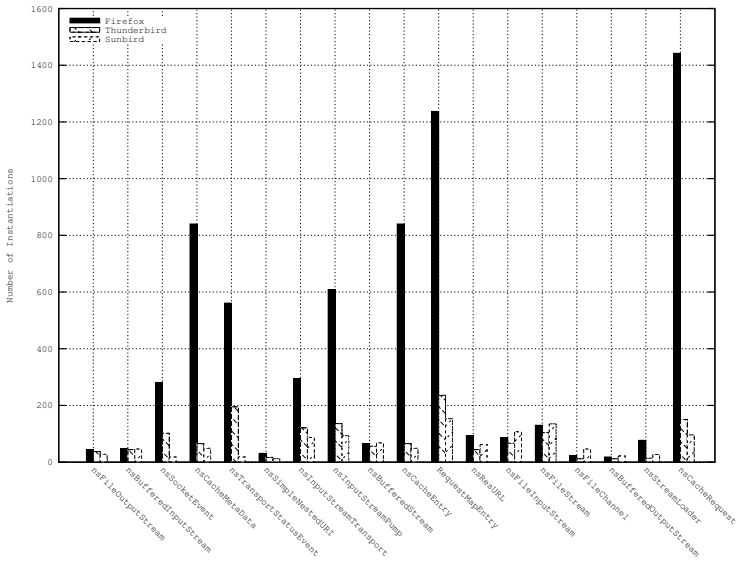

Figure 4. Necko Usage by Firefox, Thunderbird and Sunbird.

work, regular expressions are used to specify the conformance relation between two object protocols. However, the work of Butkevich et al. does not entail the reverse engineering of existing interface protocols, nor do they apply usage of interface protocols to program comprehension.

Archer et al. develop a checkable, executable specification that captures the rules for correctly using an interface in a TinyOS application [1]. They refer to this specification as an *interface contract* and they develop an approach for checking the interface contract using a source-to-source program transformation that adds checks to existing TinyOS applications. However, they do not reverse engineer the contracts and they do not demonstrate the pattern that these contracts typically exemplify.

Quante and Koschke describe a dynamic protocol recovery technique based on *object process graphs* (OPGs) [7]. The advantage of their approach is that OPGs contain information about loops and the context within which a method was called. They introduce a new metric for comparing automata and a case study involving Java and C programs. The focus of Quante and Koschke is on their recovery protocol technique that exploits context to improve the regular expression generation process; their technique improves on our brute force regular expression generation approach.

6. Concluding Remarks

We presented Hylian, our system for code regeneration and trace extraction in large C++ applications. We have demonstrated the utility of Hylian by generating trace information, extracting the sequence of method invocations, and generating regular expression representations of the interface protocols for *Necko*, a large networking library written in C++, utilized by the Mozilla Internet Application Suite [9]. We generated interface protocols for *Necko* using three large applications in the Mozilla Suite that use the *Necko* library: Firefox, Thunderbird and Sunbird [8].

The preliminary results of our case study of *Necko* support our assumption that the interface protocols for these classes follow a specific pattern and that these patterns can be used to facilitate comprehension of a class, to guide usage of the class, or to measure the complexity of usage of the class by developers unfamiliar with the *Necko* library.

References

[1] W. Archer, P. Levis, and J. Regehr. Interface contracts for tinyos. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 158–165, New York, NY, USA, 2007. ACM Press.

[2] P. Avgustinov, J. Tibble, E. Bodden, O. Lhotak, L. Hendren, O. de Moor, N. Ongkingco, and G. Sittampalam. Efficient trace monitoring, March 2006.

[3] S. Butkevich, M. Renedo, G. Baumgartner, and M. Young. Compiler and tool support for debugging object protocols. In *Proceedings of the 8th FSE*, pages 50–59, 2000.

[4] B. Cornelissen and L. Moonen. Visualizing similarities in execution traces. In *Proceedings of the 3rd Workshop PCODA*, pages 6–10. IEEE, 2007.

[5] E. B. Duffy and B. A. Malloy. An automated approach to grammar recovery for a dialect of the C++ language. In *14th Conference on Reverse Engineering, WCRE'07*, Oct 2007.

[6] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, 2000.

[7] J. Quante and R. Koschke. Dynamic protocol recovery. In *Proceedings of the 14th WCRE*, pages 219–228, Washington, DC, USA, 2007. IEEE Computer Society.

[8] The Mozilla Organization. The Mozilla Application Suite. *http://www.mozilla.org*, 2007.

[9] The Mozilla Organization, 2007. Necko. *http://www.mozilla.org/start/1.0/guide/toolkit.html#necko*.

[10] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin. Reverse engineering state machines by interactive grammar inference. In *Proceedings of WCRE '07*, pages 209–218, Washington, DC, 2007.

[11] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of ISSTA*, July 2002.