Secrets from the Monster: Extracting Mozilla's Software Architecture

Michael W. Godfrey and Eric H. S. Lee
Software Architecture Group (SWAG)
Department of Computer Science, University of Waterloo
Waterloo, Ontario, N2L 3G1, CANADA
email: {`migod,ehslee`}`@plg.uwaterloo.ca`

ABSTRACT
As large systems evolve, their architectural integrity tends to decay. Reverse engineering tools, such as PBS [7, 19], Rigi [15], and Acacia [5], can be used to acquire an understanding of a system's "as-built" architecture and in so doing regain control over the system. A problem that has impeded the widespread adoption of reverse engineering tools is the tight coupling of their subtools, including source code "fact" extractors, visualization engines, and querying mechanisms; this coupling has made it difficult, for example, for users to employ alternative extractors that might have different strengths or understand different source languages.

The TAXFORM project has sought to investigate how different reverse engineering tools can be integrated into a single framework by providing mappings to and from common data schemas for program "facts" [2]. In this paper, we describe how we successfully integrated the Acacia C and C++ fact extractors into the PBS system, and how we were then able to create software architecture models for two large software systems: the Mozilla web browser (over two million lines of C++ and C) and the VIM text editor (over 160,000 lines of C).

Keywords
Interchange formats, reverse engineering, software architecture.

1 INTRODUCTION

Large software systems must evolve or they risk losing market share to competitors [11]. However, the architectural integrity of such systems often decays over time as new features are added, defects are fixed, performance is tuned, and support for new platforms is added [18, 22]. Reverse engineering tools such as PBS [7, 19], Rigi [15], and Acacia [5], can be used by developers to regain an understanding of the "as-built" software architecture of a system, and to reconcile it with the "conceptual" or intended software architecture [9]. However, most such tools are comprised of tightly coupled subcomponents, such as source code "fact" extractors and visualization engines. This tight coupling has impeded the widespread adoption of such tools, as it is difficult for users to substitute alternative subtools that might have different strengths or model different source code languages.

The TAXFORM (Tuple Attribute eXhange FORMat) project has sought to investigate how different subtools can integrated into a single framework by providing mappings to and from common data schemas for program "facts". Previous work has included the design of generic schemas for procedural and object-oriented programming languages, an exploration of problematic issues in representing and translating facts about programs, and some preliminary experiments in using the Acacia and Rigi extractors as "front-ends" to the PBS system [2].

Our primary motivation for the work described in this paper was the desire to create software architecture models of the Mozilla web browser [14]. Mozilla is written using a combination of C++ and C; however, the extractor for the PBS system, `cfx`, does not support the C++ language, and furthermore we found that it was unable to process much of the portion of Mozilla that is written in C. In this paper, we describe how we created a systematic translation mechanism to allow the integration of the Acacia fact extractors for C and C++ into the PBS system, and how we subsequently used the translators to create software architecture models for two large software systems: Mozilla (over two million lines of C++ and C code) and the VIM text editor [23] (over 160,000 lines of C code).

2 THE PBS AND ACACIA SYSTEMS

The work we describe here involves the PBS [7, 19] and Acacia [5] reverse engineering systems. The Acacia system provides facilities for extracting and visualizing low-level facts about systems written in C and C++, but it provides little automated support for creating high-level views of a system's software architecture. Acacia includes two fact extractors: `CCia` which can be configured to process C++ or C code, and the older extractor `cia` which extracts less information and works only with the C language but which we found to be more robust when applied to some C systems. The results of the extractions are stored in textual databases which can be queried at the command line or by using the CIAO visualization tool.

We have chosen to base our work around the PBS system as we have extensive experience with it, and because it provides rich support for the creation and querying of high-level views of software systems. PBS includes a special "relational calculator" language called `grok` that allows users to create customized views quickly and easily [19]. Extracted "facts" about a system are stored using a generic schema language called TA (Tuple Attribute); a user may define desired abstract relations on these facts, which the `grok` interpreter then processes, by performing the appropriate relational calculations, to create high level architectural views of the system. In this way, a user can create structured and multi-layered views of the system's software architecture which can be navigated and queried by the PBS visualization tool.

We decided to adapt the C and C++ extractors from the Acacia system for use within PBS for several reasons. Our primary motivation was the desire to create software architecture models of systems written in C++ without having to create a customized C++ fact extractor.[1] The Acacia C++ extractor, `CCia`, performs a detailed extraction of entities and relationships of C++ code[2], and it uses a production-quality front end.[3] Second, the fact extractor for PBS, `cfx`, supports only the C language and has been found to be fairly fragile; we hoped to gain an alternative extractor for the C language, and also evaluate the relative quality of each extractor.[4] And finally, we wished to explore the practical problems in translating "facts" extracted by one system for use with a different system.

3 TRANSLATING ACACIA OUTPUT INTO TA

We decomposed the task of creating a translation mechanism from Acacia into TA (PBS's format) into two stages. First, we adapted Acacia's C language extractors for use as drop-in replacements for PBS's C extractor, and then we built on this experience to create a mechanism for translating `CCia` output of C++ code into PBS. This second step also involved the creation of new `grok` scripts for modelling and visualizing object-oriented systems in PBS.

The PBS extractor `cfx` generates an intermediate format that is used by another tool, `fbgen`, to generate textual tuples (in TA format) that describe attributes of the program entities (*e.g.,* files, functions, variables, macros) and their interrelationships (*e.g.,* containment, function calls, variable references, macro uses). For example, the following TA facts are taken from an extraction of the source code for version 3.0 of the `ctags` system:

```
funcdcl read.h fileClose
funcdef read.c fileClose
funcdcl main.h getFileSize
funcdef main.c getFileSize
linkcall fileClose getFileSize
```

These TA facts assert that `fileClose` and `getFileSize` are C functions declared in `read.h` / `main.h` respectively, defined in `read.c` / `main.c` respectively, and that there is a call from `fileClose` to `getFileSize` that must be resolved by the linker. The resolution of which function calls which other function and what these relationships mean at the file and subsystem level is performed subsequent to the extraction by a set of `grok` scripts.

Acacia extraction output is stored in two semi-colon delimited plain-text databases, one for entities and one for relationships. Each entity is assigned a unique identifier (UID) by the extractor.[5] A typical entry in the entity database includes the entity's name, its UID, the UID of the containing file, its visibility, its signature/datatype (if appropriate), and whether the entity is a declaration or a definition (if the entity is a function or variable). Resolution of relationship information (*e.g.,* "which function `f` is being called by function `g`?") is performed by the extractor; a typical relationship database entry lists the details of each entity involved in the relationship (including the UIDs) together with attributes of the relationship (*e.g.,* two functions may be "friends", or one may call the other, or one may be a template instantiation of the other).

While the Acacia and PBS fact extractors perform similar tasks and are used in similar ways, there were a number of semantic discontinuities that had to be addressed. In particular, the idea of what an entity is (*e.g.,* is function declaration a distinct entity from a like-named function definition?) and how entities involved in relationships are resolved (*e.g.,* if `f` calls `g`, does `f` call the declaration or the definition of `g`, and is there also a relationship between their respective containing files?) were incompatible. For example, unlike PBS, Acacia considers declarations and definitions to be distinct entities, and they are given distinct UIDs. Also, the function call relationship described above in TA would be modelled by Acacia as a relationship between the function *definitions* in the "dot-c" files. This is subtly different from the PBS assumption and required "unfolding" some of the relationships extracted by Acacia in the conversion scripts.

[1] Creating a correct and robust parser for C++ is known to be a difficult problem due to the language's inherent complexity. By comparison, a high-quality fact extractor for the Java language was created by a member of our group in only a few days [4].

[2] We also briefly considered using two other C++ extractors: Gen++ [6] and Datrix [10]. Anecdotal evidence suggested that the Gen++ tool was relatively fragile and hard to configure, and we found that while Datrix extracts finely grained entity-level information, it does not resolve relationship references beyond the "name-level" [2].

[3] `CCia` is built around the Edison Design Group (EDG) C++ front end, a commercial product.

[4] Murphy [16] and Armstrong [1] have performed comparative analyses of several extractors.

[5] `cia` uses a simple counter to implement UIDs while `CCia` generates an eight digit hexadecimal UID using an an attribute-based hashing function.

There were two major steps in the conversion process. First, simple textual queries were made of the entity and relationship databases, and processed through `awk` and `perl` scripts to generate TA. Then, a `grok` script was used to change the semantic model of the facts to what the PBS tool was expecting.

We now discuss our experience in using these translation mechanisms on the VIM and Mozilla systems.

4 EXTRACTING VIM'S SOFTWARE ARCHITECTURE

Our first two example systems written in C that we tried out were the VIM text editor (150,000 lines of code) and its companion tool `ctags` (12,000 lines of code). The source code for VIM made `CCia` crash; we discovered that `CCia` was less robust than `cia` when applied to some C systems that used non-ANSI conventions. Consequently, we also added support for the older `cia` extractor, although it extracts less information and with a different output format than `CCia`.

The fact extraction and conversion of `ctags` was straightforward, although it revealed some internal problems with the `CCia` extractor. We found that the `CCia` extractor sometimes created multiple UIDs for the same entity. While this might seem benign, it proved to be troublesome; when a function declaration had multiple UIDs, some relations were resolved incorrectly. Once we discovered this problem, we were able to work around it by discarding the `CCia` UIDs and using our own "name mangling" convention within a grok script to work out entity resolutions correctly. In so doing, we found our results still differed from the `cfx` extraction, we discovered several subtle bugs in how PBS performs "linking" (entity resolution) that have since been fixed.

Results for VIM

We performed a full extraction on version 5.6 of the VIM editor using both `cfx` and `cia`, and then we translated the `cia` facts into TA format using our scripts. The `cia` extraction was faster, but when combined with the translation time, the total was slightly more than that for the `cfx` extraction. The total time for both approaches was slightly faster than a full compile of the system.[6]

The full distribution of version 5.6 of VIM, which includes the companion utility `ctags`, comprises over 163,000 lines of C code (including comments and blank lines). The breakdown of the distribution into header files (`.h` and `.pro` files) and implementation files (`.c` files) is shown below:

[6] On a Sparc running Solaris 2.6 with four 300MHz processors and 1 gigabyte of memory, the `cfx` extraction took 4:27 minutes, the `cia` extraction took 1:52 minutes, the translation of the `cia` output to TA took 3:20 minutes, and a full compile of VIM took 6:29 minutes.

File type	Total # of files	Total KLOC
`.h`	35	8,051
`.pro`	47	1,316
`.c`	67	154,360
TOTAL	149	163,727

Unlike Mozilla, almost all of the source code files are included in a typical compile. We found that the breakdown of the system into source files was primarily based on functionality and features; while VIM can be compiled to run on a variety of platforms, most of the platform-specific code is distributed throughout the various source files.

We found that a `cfx` extraction of VIM (ignoring `ctags`) produced over 43,000 "facts".[7] Performing a analogous extraction using `cia` plus our translation scripts produced over 51,000 facts. Comparing the two extractions in detail, we found several notable differences:

- `cia` (and `CCia`) perform macro expansion to extract more detailed relationship information. For example, if a function `f` calls a macro `m` that in turn expands to a call to a function `g`, then both Acacia extractors will record that `f` uses macro `m` *and* that `f` calls function `g`. `cfx` does not perform this level of analysis. This was the primary source of "extra" facts extracted by `cia`.
- We added some extra detail that `cia` extracts but `cfx` does not model, including references to library variables, such as `__ctype` and `errno`.
- `cia` does a more accurate extraction of function call information than `cfx`. We found that `cfx` missed a number of straightforward function calls that `cia` found.
- A fairly common programming convention in C is to define a macro named `EXTERN` that precedes function and variable declarations in "`.h`" files. This macro expands to the keyword `extern` in all implementation files that use (but do not define) the entity, and expands to the empty string in the implementation file that defines the entity. We found that `cfx` was able to model this convention correctly, but that `CCia` did not.

In summary, we found that we were able to successfully adapt `cia` and `CCia` into high quality C extractors for the PBS system with performance similar to that of the native PBS C extractor. With the exception of the `EXTERN` problem, we were able to adjust for all of the semantic inconsistencies and other problems using `grok` scripts.

Observations about VIM

Figure 1 shows a top level view of the software architecture model for VIM. This model was created using a variety of

[7] A total of 30 kinds of facts were extracted for the C language model, including `funcdef`, `usemacro`, and `include`. Precise details of the schemas for entities and relationships extracted by `cfx` can be found elsewhere [2, 7, 19].

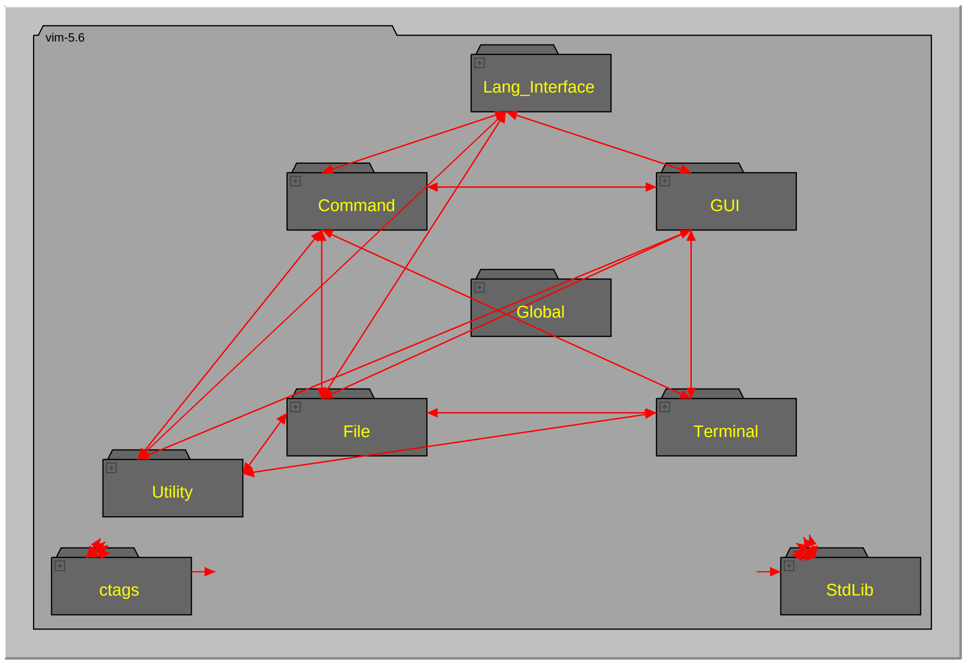

Figure 1: Top level view of the extracted architecture of VIM as shown by the PBS viewer. Folder icons denote subsystems, and arrows denote function calls between subsystem members (some calls are elided and are shown as arrow stubs). The subsystems are described briefly in Fig. 2.

Subsystem name	**# of contained source files**	**Total KLOC**	**Description**
Command	29	55	User command processing
File	16	20	File I/O and buffer manipulation
Lang_Interface	9	6	Interface to prog. langs. (*e.g.*, Perl, Python, tcl)
Global	15	5	Contains global variables, data structures defs, *etc.*
GUI	21	33	User interface code
Terminal	4	5	Mappings for kbd/mouse
Utility	10	14	Implements regexps, message routines, *etc.*
Ctags	36	18	VIM's companion tool
Stdlib	303	72	System `include` files (*i.e.*, not part of distribution)
TOTAL *(all)*	443	228	
(ignoring Stdlib)	140	156	

Figure 2: The major subsystems in our architectural model of VIM version 5.6, as shown in Fig. 1. This model includes only the code that was used during a typical compile of VIM for the Linux operating system running on an Intel 686 processor.

knowledge sources including the system documentation, domain knowledge about text editors, a detailed examination of source code, and the authors' extensive experience in using VIM.

It is not our intention to discuss VIM's software architecture in detail in this paper, as we do so elsewhere [22]; however, we do note some general observations. First, we discovered that VIM has been implemented using a repository-style software architecture [20]. The data structures that implement the buffer being edited are globally accessible variables defined within the *Globals* subsystem; this explains why there are no function call arrows going into or out of the *Globals* subsystem in Fig. 1.

Another result that we found to be surprising was that the *Utility* subsystem had functional dependencies on other subsystems. Upon closer examination, we found that most of these unexpected dependencies were contributed by two large files `misc1.c` and `misc2.c` comprising over 5700 LOC and 2400 LOC respectively. As their names suggest, they contain a variety of unrelated functions; we found comments within the code such as *"Various functions"* and *"functions that didn't seem to fit elsewhere"* that confirmed our hypothesis. Our subsequent "repair" of VIM's architecture resulted in moving many of these functions to other files in other subsystems [22].

5 EXTRACTING MOZILLA'S SOFTWARE ARCHITECTURE

We next considered how to create a software architecture model of the Mozilla web browser using the Acacia extractor and the PBS system. Mozilla is the "open source" subset of the Netscape browser [14, 17]. It is a huge, multi-function, multiplatform system comprising over two million lines of C++ and C code in the release version we examined (Milestone-9 or "M9").

We rewrote our translation scripts to use an object-oriented language schema; the schemas we created comprised 71 kinds of facts (compared to 24 for the procedural C model) [12]. We created additional infrastructure for the PBS system to be able to create and navigate through software architecture models of object-oriented systems, which consisted mostly of `grok` scripts and data files used by the PBS viewer.

The biggest challenge in creating these scripts was in distinguishing between entities that might have the same name. In C, "name collisions" between globally visible entities are fairly rare, but in C++ they are much more common due to overloading, polymorphism, use of templates *etc*. We used a more complex "name mangling" scheme than we had used with the C scripts; we did not use Acacia's UIDs since, as mentioned above, `CCia` sometimes generated spurious extra UIDs for some entities.

Initial attempts at fact extraction led us to rewrite our translation scripts yet again, as we found the performance to be unacceptable; our approach with VIM has been to use simple minded `awk` scripts to transliterate the Acacia facts into TA using a series of queries, and then perform "intelligent" translation using `grok`. We found we had to read the entire Acacia databases into a large associative array and then generate the "naive" TA facts in one go.[8]

Results for Mozilla
As mentioned above, Mozilla release M9 consists of over two million lines of C++ and C code. The source distribution of C and C++ header and implementation files breaks down as shown below:

File type	Total # of files	Total KLOC
`.h`	4,531	610
`.c`	811	434
`.cpp`	2,079	1,043
TOTAL	7,421	2,087

Total KLOC denotes thousands of lines of source code including comments and blank lines. This count includes all source files for all supported platforms in the source distribution, but does not include header files that are generated automatically during a system build. Using the utility `ctags`, we calculated that there are over 2,500 classes, 33,000 class methods, 18,000 class/struct/union data members, 11,000 global ("`extern`") functions, and 3,500 global ("`extern`") variables in the source code contained in the `tar` file distribution.

Because Mozilla is multiplatform, a large part of its distributed code base consists of parallel sets of platform-specific implementation files [8]. In order to perform an analysis of the relationships within a typical instantiation of Mozilla, it made sense to construct an architectural view of one build of the system. We therefore compiled Mozilla for Linux, and found that it processed 192 of the 811 "`.c`" files and 1319 of the 2079 "`.cpp`" files. We then used a trace of the build process to decide which files to extract facts from.

We used the C++ option of the `CCia` extractor for both the C++ and the C portions of Mozilla. We had considered using the `CCia`'s C extraction option for the C code, but we decided that it would be too awkward to generate two sets of databases with different schemas and different translation mechanisms that then had to be reconciled into a coherent whole. The use of the C++ option required some manual adjusting of the C code to account for the stronger type checking rules of C++; in particular, many C implementation files were edited to add explicit type casting (this approach did not work so easily for macros that take parameters). Additionally, we discovered that the commercial front end used by `CCia` did not recognize the `static const` construct of C++.

[8] Godfrey wrote the original C translation scripts in `awk`; Lee reimplemented them for C++ using `perl`.

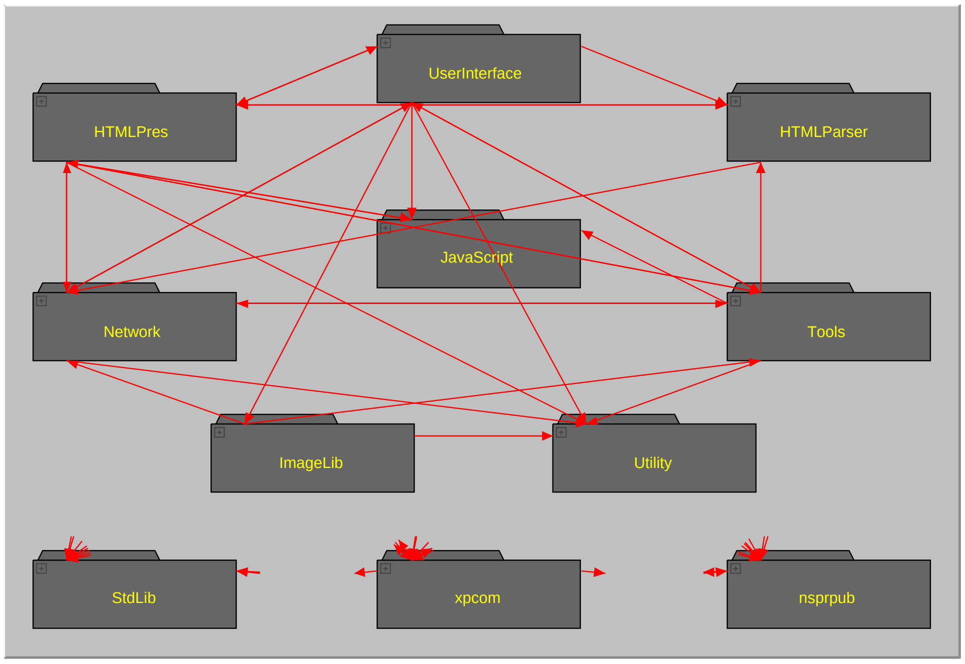

Figure 3: Top level view of the extracted architecture of Mozilla as shown by the PBS viewer. Folder icons denote subsystems, and arrows denote function calls between subsystem members (some calls are elided and are shown as arrow stubs). The subsystems are described briefly in Fig. 4.

Subsystem name	**# of contained subsystems**	**# of contained source files**	**Total KLOC**	**Description**
HTMLPres	47	1,401	484	HTML layout engine
HTMLParser	8	93	42	HTML parser
ImageLib	5	48	15	Image processing library
JavaScript	4	134	47	JavaScript engine
Network	13	142	31	Networking code
StdLib	12	250	45	System `include` files (*i.e.*, "`.h`" files)
Tools	47	791	269	Major subtools (*e.g.*, mail and news readers)
UserInterface	32	378	147	User interface code (widgets, *etc.*)
Utility	4	60	35	Programming utilities (*e.g.*, string libraries)
nsprpub	5	123	51	Platform independent layer
xpcom	23	224	63	Cross platform COM-like interface
TOTAL	200	3,650	1,229	

Figure 4: The major subsystems in our architectural model of Mozilla release M9, as shown in Fig. 3. This model includes only the code that was used during a typical compile of Mozilla for the Linux operating system running on an Intel 686 processor.

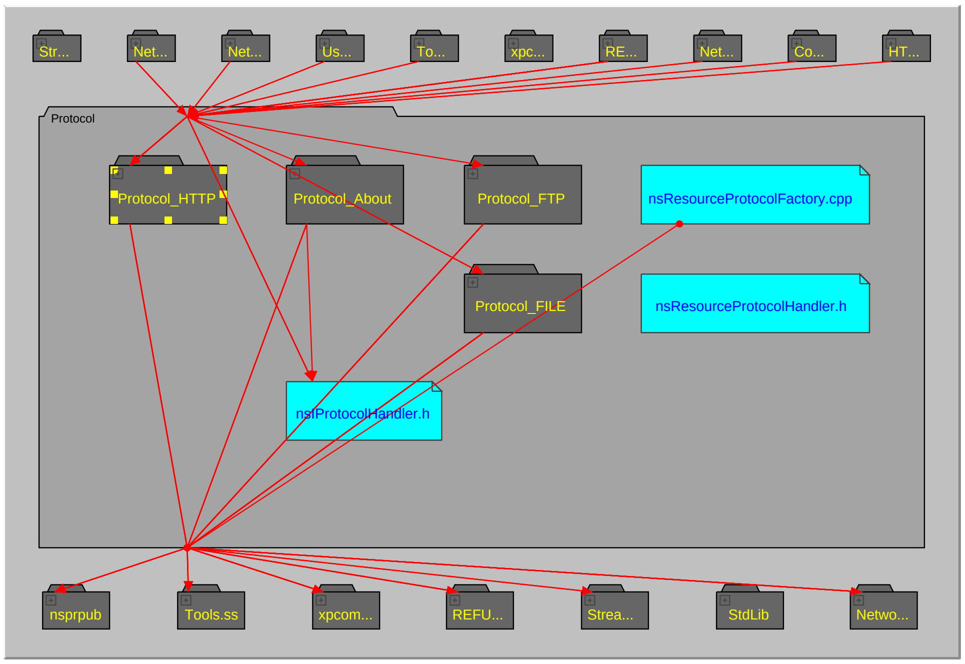

Figure 5: View of the Protocol subsystem of Mozilla (a member of the top-level Network subsystem) as shown by the PBS viewer. Folder icons denote subsystems, document icons denote source files, and arrows denote function calls.

The manual adjustment of code was laborious and time consuming. Eventually, we decided that 23 of the 1511 files were too difficult to fix without an enormous effort in restructuring and program understanding. However, we note that we still managed to process more than 98% of the files in the system.

A full source build of Mozilla M9 on a dual processor Pentium-III 450 MHz system with 512 megabytes of RAM running Redhat Linux 6.1 took 35 minutes. The `CCia` extraction took three and a half hours, and the translation into TA using our scripts took another three hours on the same system. The extraction generated over 990,000 facts, taking up over 133 megabytes of disk space (uncompressed). We note that the total extraction time is still much less than the amount of time we spent editing the source code so that the extractor would be able to process it.

Observations About Mozilla

We created the subsystem hierarchy of our software architecture model based on several sources of information, including the source directory structure, examining the extracted facts, the use of an automated subsystem clustering tool [19], reading through the source code and documentation, and browsing the Mozilla website [14]. Our architecture model contains 11 top-level subsystems, as shown in Fig. 3 and Fig. 4; of these, the largest were concerned with HTML layout, the implementation of subtools such as the mail and news readers, and user interface code. Figure 5 shows a typical intra-subsystem view as shown by the PBS viewer/navigator.[9]

As with VIM, we do not discuss Mozilla's software architecture in detail in this paper, as we do so elsewhere [13]; however, we also note some general observations. First, our in-depth examination of Mozilla leads us to conclude that either its architecture has decayed significantly in its relatively short lifetime, or it was not carefully architected in the first place. For example, the top-level view of Mozilla's architecture resembles a near-complete graph in terms of the dependencies between the different subsystems (Fig. 3 shows the function call dependencies); while we might reasonably expect function calls from the user interface subsystem to most other subsystems, we were surprised to see functional dependencies from the image processing library to the network and tools subsystems. Overall, we found the architectural coherence of Mozilla to be significantly worse than that of other large open source systems whose software architectures we have examined in detail (Linux and VIM) [3, 21, 22].

However, we do not consider these results to be surprising, as Netscape was among the first generation of web browsers; it is well known that competition during the "browser wars"

[9] These figures show only function call relations at the file and subsystem level; other information can also be shown by the viewer, such as variable references and class inheritance. Additionally, the architecture views can be navigated hierarchically as well as queried.

has been intense. Netscape and its main competitor, Microsoft's Internet Explorer, have evolved extremely rapidly over the last few years, leading not only to an abundance of new features, but also to a very large number of "bugfix" releases and a notorious reputation for unreliability. Mozilla seems to be a telling example of Lehman's laws of software evolution, which state that a useful software system must undergo continual and timely change or it risks losing market share [11].

6 SUMMARY

In this paper, we have described our experiences in extending the work of the TAXFORM project [2]. We have created automated mechanisms for converting the output of Acacia's C and C++ extractors into generalized textual schemas for procedural and object-oriented languages using the TA notation. We also described our experiences in using these mechanisms in the creation of software architecture models for two large software systems: the Mozilla web browser (over two million lines of C++ and C code) and the VIM text editor (over 160,000 lines of C code).

We have undertaken this work for several reasons: to investigate the practical issues involved in transforming extracted data between abstract schemas; to allow the creation of navigable high-level software architecture models for systems written in C++; and to explore the relative differences between the two reverse engineering systems. We found that we were able to successfully adapt the Acacia extractors for use in the PBS system, and that the conversion of extracted facts is straightforward once a suitable translation mechanism is in place. We note that, as observed by others [1, 16], the robustness of the extractors and quality of the extracted facts varies between tools, and that it is sometimes necessary to "tweak" the source code of the system being examined in order to get the extractor to process it correctly. Finally, we consider that this work represents a significant data point in the quest for seamless data exchange between reverse engineering environments.

REFERENCES

[1] Matt Armstrong and Chris Trudeau, "Evaluating Architectural Extractors", *Proc. of the 1998 Working Conference on Reverse Engineering* (WCRE-98), Honolulu HI, October 1998.

[2] Ivan Bowman, Michael W. Godfrey, and Ric Holt, "Connecting Architecture Reconstruction Frameworks", *Proc. of CoSET'99 — Symposium on Constructing Software Engineering Tools*, May 19999. Also published in *Journal of Information and Software Technology*, vol. 42, no. 2, February 2000.

[3] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster, "Linux as a Case Study: Its Extracted Software Architecture", *Proc. of the* 21^{st} *Intl. Conf. on Software Engineering* (ICSE-21), Los Angeles CA, May 1999.

[4] Ivan Bowman, Michael W. Godfrey, and Ric Holt, "Extracting Source Models from Java Programs: Parse, Disassemble, or Profile?", in preparation.

[5] Yih-Farn Chen, Emden R. Gansner, and Eleftherios Koutsofios, "A C++ Data Model Supporting Reachability Analysis and Dead Code Detection", *IEEE Trans. on Software Engineering*, vol. 24, no. 9, September 1998.

[6] P. Devanbu, "A Language and Front-end Independent Source Code Analyzer Generator", *Proc. of the* 14^{th} *Intl. Conf. on Software Engineering* (ICSE-14), 1992.

[7] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong, "The Software Bookshelf," *IBM Systems Journal*, vol. 36, no. 4, November 1997.

[8] Michael W. Godfrey and Qiang Tu, "Evolution of Open Source Software: A Case Study", submitted for publication, available from `http://plg.uwaterloo.ca/~migod/papers/`.

[9] Christine Hofmeister, Robert Nord, and Dilip Soni, *Applied Software Architecture*, Addison-Wesly Longman Inc., Reading MA, 2000.

[10] B. Laguë, C. Leduc, A. Le Bon, E. Merlo, and M. Dagenais, "An Analysis Framework for Understanding Layered Software Architectures", *Proc. of the 1998 Intl. Workshop on Program Comprehension* (IWPC '98), Ischia, Italy, June 1998.

[11] M.M. Lehman, J.F. Ramil, P.D. Wernick, D.E. Perry, and W.M. Turski, "Metrics and Laws of Software Evolution — The Nineties View", *Proc. of the Fourth Intl. Software Metrics Symposium* (Metrics'97), Albuquerque NM, 1997.

[12] Eric H. S. Lee, "The Software Bookshelf for Mozilla", website, `http://swag.uwaterloo.ca/~ehslee/pbs/mozilla/R2/`.

[13] Eric H. S. Lee, "Mozilla: Its Extracted Software Architecture", in preparation.

[14] "The Mozilla Homepage", website, `http://www.mozilla.org/`.

[15] Hausi Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl, "A Reverse Engineering Approach to Subsystem Structure Identification", *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 4, December 1993.

[16] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S-C. Lan, "An Empirical Study of Static Call Graph Extractors", *ACM Transactions on Software Engineering and Methodology* (TOSEM), vol. 7, no. 2, 1998.

[17] “The Open Source Homepage”, website, `http://www.opensource.org/`.

[18] David Parnas, “Software Aging”, *Proc. of the* 16^{th} *Intl. Conf. on Software Engineering* (ICSE-16), Sorrento, Italy, May 1994.

[19] “The PBS Homepage”, website, `http://www-turing.cs.toronto.edu/pbs/`.

[20] Mary Shaw and David Garlan, *Software Architecture: Perspectives of an Emerging Discipline*, Prentice Hall, Englewood Cliffs, New Jersey, 1996.

[21] John B. Tran and R.C. Holt, “Forward and Reverse Repair of Software Architecture”, *Proc. of CASCON 1999*, Toronto, November 1999.

[22] John B. Tran, Michael W. Godfrey, Eric H. S. Lee, and Richard C. Holt, “Architecture Analysis and Repair of Open Source Software”, to appear in *2000 Intl. Workshop on Program Comprehension* (IWPC’00), Limerick, Ireland, June 2000.

[23] “The VIM Homepage”, website, `http://www.vim.org/`.