A Quick Tools Strategy for Program Analysis and Software Maintenance

Bret Johnson, Steve Ornburn, and Spencer Rugaber

College of Computing
and
Software Research Center
Georgia Institute of Technology
Atlanta, GA 30332-0280

Abstract

Most software maintenance tasks are driven by specific customer requests for program corrections or enhancements. These often require detailed analyses of specific code segments. Monolithic tools may not be flexible enough to deal with such specific requests. This paper describes a strategy for quickly producing new special-purpose tools. The strategy combines existing tools including simple, off-the-shelf text processing tools; rule-based, language-specific analysis tools; and a commercial CASE tool.

1 Background

1.1 The maintenance context

The greatest part of the software maintenance process is devoted to understanding the system being maintained. Fjeldstad and Hamlen report that 47% and 62% of the time spent on actual enhancement and correction tasks, respectively, are devoted to comprehension activities. These involve reading the documentation, scanning the source code, and understanding the changes to be made[1]. The implication is that if we want to improve maintenance, we should facilitate the process of comprehending existing programs.

The most desirable approach to maintaining a software system is to devote a significant amount of upfront effort to understanding and documenting the overall purpose and behavior of the software. Unfortunately, the realities of a software maintenance shop often require quick responses to unanticipated problems or enhancement requests. The comprehensive study approach must then be discarded in favor of a more responsive one.

The same argument holds for tools. However desirable a comprehensive program analysis tool is, the state of the art is such that monolithic tools are often not flexible enough to deal with spontaneous requests for information. An approach is needed where specific questions can be answered by quickly producing special-purpose analysis tools. And this, in turn, can best be accomplished by combining existing tools.

This paper describes a strategy for quickly producing tools. It has been successfully used in the context of the maintenance and reverse engineering of a large, real-time software system used for telephony. The tools include off-the-shelf text processing tools; rule-based, language-specific tools; and a commercial CASE tool. The approach is illustrated by examples including the generation of cross reference information, calling trees, run-time stack analysis, and code restructuring.

1.2 A large-scale, real-time, embedded software system

Maintenance and reverse engineering activities were carried out on software components within a large, real-time system built and maintained by a major telecommunications company. This system, a digital subscriber carrier that has undergone significant modification and enhancement in the ten years it has been on the market, extends both regular and special telephone services from a switching center to residential and business communities. Its main application is to increase the number of subscribers that can be economically served by a feeder cable. Functionally, it converts between the digital signals carried on the

telephone network and the analog signals required by telephone subscribers. It also does the time-slot management and multiplexing required to complete the interface between telephone network and telephone subscriber.

The application software is coded in the PL/M programming language and consists of approximately 200K lines of code. The operating system on which the application runs is custom-built and consists of an additional 10K lines of assembly language code. Software engineers assigned to the product are responsible for maintaining both the application and the operating system. All of the product software is ROM-based, which eliminates the possibility of field patches, and therefore dictates that high levels of quality be achieved and maintained with each subsequent release of the system software.

Finally, while the code is structured in the sense that there are no explicit "go-tos," procedures are generally not well organized. They have been decomposed into monolithic, deeply-nested conditional statements rather than sequences of simply described functions. Algorithms are often implemented using convoluted idioms. And there is no distinction between the normal execution paths through a procedure and those associated with exception handling.

2 Tool strategy

The quick tools strategy involves three phases. The first phase constructs throw-away tools from generic, off-the-shelf text processing utilities. The second phase produces more robust analysis tools, customized to deal with the specifics of the client's development environment. The final phase uses an existing CASE tool to support the graphical display of the data generated by the first two phases.

2.1 Phase 1: Prototyping with awk and shell scripts

One characteristic of the UNIX operating system is a large collection of utility commands adept at manipulating textual, line-oriented data. Another is a command language that can be used to easily compose existing commands. The command language, called the shell, composes commands by passing the output of one to the input of the next. Facilities also exist to save results in files, abbreviate complicated commands or file paths with short identifiers, and abstract composite commands into command files that can in turn be used to construct higher-level commands.

Among the available UNIX text processing tools are **sed** and **awk**. **Sed** is a stream editor, capable of transforming an input file through a sequence of line-by-line edits. Lines to be edited can be selected via regular expression pattern matching. **Awk** is even more powerful. It is capable of treating an input file as if it consisted of a sequence of fields. Field values can be examined and altered.

In the first phase, we constructed a procedure cross reference tool using **sed**, **awk**, and other UNIX tools in the following manner. First, a small **awk** script extracted lines containing procedure declarations. (See Figure 1.) An **awk** script consists of a series of statements. Each statement contains a pattern and an action. The pattern determines the lines in the input file to which the action will be applied. Actions are contained within curly braces. Comments begin with '#' and continue to the end of the line. The script prints out records describing procedure declarations and the names of the files in which they occur.

Several things should be noted about this script. Most importantly, it is just an approximation. **Awk** patterns are not powerful enough to completely define the PL/M language. However, these few rules were adequate for this job. A similar script was written to post-process the data produced by the script given above. It removed all of the extraneous (for our purposes) PL/M code leaving only the filename and the procedure name.

The second point is that the script depends not only on the PL/M language but also on the coding conventions used in this particular maintenance shop. Knowledge of the conventions was taken advantage of to more quickly arrive at a working solution.

The third point is that this script is not the first version. It was refined over time by a series of trial and error experiments. For example, procedure declarations may actually be continued over several lines. The first version had to be adapted to take into account this possibility.

The pair of scripts described above were complemented with a similar pair that detected procedure invocations. Both pairs were invoked by commands written in the shell language that gave access to all of the source files. The output data were placed into two text files that were combined using further scripts to produce the procedure calling information that is partially displayed in Figure 2. In the figure each procedure declaration occurs on a line beginning with '**'. It includes the procedure name and the name of the file in which the declaration occurs. Subsequent lines indicate other files that contain references to the pro-

```
BEGIN                                   { C = 0; N = 0; K = 0 }
#
# The pattern 'BEGIN' is matched before any line of the input file has been
#   read.  'C' is as flag indicating whether or not the input line is contained
#   within a PL/M comment.  Likewise, 'N' indicates whether the procedure label
#   has been seen, and 'K' whether the 'PROCEDURE' keyword has been detected.
#
/\/\*/                                  { C = 1 }
#
# A comment has been detected.
#
C==0 && /:/                             { L = $0; N = 1 }
#
#  Look for a line with a ':' but not in a comment.  If you find one, remember
#    the whole line ($0) in 'L', and set the 'N' flag.
#
C==0 && N==1 && /PROCEDURE/             { K = 1 }
#
# If the 'PROCEDURE' keyword is found after the label, then set the 'K' flag.
#
/;/ && C==0 && N==1 && K==1             { print FILENAME " " L }
#
# Once the terminating ';' is found, print the name of the file and the record.
#
/;/                                     { N = 0; K = 0 }
#
# Reset the 'N' and 'K' flags when the declaration is complete.
#
/\*\//                                  { C = 0 }
#
# Reset the 'C' flag when the end of the comment is detected.
```

Figure 1: **Awk** Script for Extracting Procedure Definitions

cedure. The total count of script lines is well under one hundred. Writing the scripts took an afternoon; running them took around twenty minutes.

2.2 Phase 2: Adding precision with New-Yacc

The tools described above were able to rapidly produce useful results. However, there are limitations on the accuracy and completeness of the data produced. The tools rely on regular expression pattern matching. Regular expressions can be used to describe language constructs at the lexical level. For example, it is easy to write a regular expression to detect all occurrences of an identifier in a program source file. However, programming languages are more complicated than this. Many languages, including PL/M, allow the use of nested scopes, in which several identifiers can have the same name. Regular expressions are incapable of differentiating between identifier uses at different levels, but other tools are.

A more powerful pattern matching capability is provided by context free parsing. For example, the UNIX tool **yacc** (Yet Another Compiler Compiler) uses a variant of context free parsing to automatically construct parsers for many programming languages such as C and Pascal. **Yacc** takes as input a grammatical description of the programming language written in a notation called *BNF* (Backus-Naur Form). The grammar is annotated with C language program segments to specify the intended translation of a program. **Yacc** constructs a parser that, when given a program written in the programming language, ap-

```
** ABORT_SYS_CE_VER_CTL_SEQ_P (CPMP106)
   CPMP101   CPMP106   CPMP114

** ACCESS_NVS_DIRECT (CUTP127, RTEP110)
   BPIP119   CAUP102   CAUP103   CAUP105
   CMAP405   CMAP406   CMAP407   CPIP103
   CPIP106   CPIP110   CPIP111   CTEP108
   CUTP145   RTEP110

** ACO_TSK (CALP101, RALP101)
   CALP101   RALP101

** ACTIVATE_CP (BUTP114)
   BINP104   BUTP114

** ACTIVE_RING_GENERATOR (BASP319)
   BASP319   CASP320   CASP323

** ACTIVE_SLEEVE_CHECK (CTAP113)
   CTAP113   CTAP116

** ACTIVE_TMG_FAIL (RDSP207)
   RDSP108   RDSP207
```

Figure 2: Calling Tree Records Produced by **Awk**

plies the translations. The translations typically specify how to construct object code for a specific machine architecture.

NewYacc is a preprocessor to **yacc** developed by Purtilo and Callahan[5]. It can be used to analyze and transform programs at the source level rather than at the level of compiled object code. And because the source language is expressed in BNF and context free parsing is used, tools built using **NewYacc** much more precisely reflect the original source language constructs than do tools using regular expressions.

In order to apply **NewYacc** to program analysis tasks, two steps are required. The first step is to construct a grammatical description of the source language being used, in our case PL/M. This step takes the descriptive grammar for the language found in the language reference manual and adapts it to the format required by **yacc**. This task is a somewhat involved technical problem that requires knowledge of the details of context free parsing, but it needs to be performed only once per language and not once per tool. In our case, the effort took about a week. However **yacc** grammars for many programming languages are available commercially or in the public domain.

The second step is to add **NewYacc** rules to the grammar to describe the particular kind of analysis that a tool requires. The rules indicate what **NewYacc** should do when it encounters a specific language construct while parsing a program. One example that we implemented involves a tool that constructs a calling tree for a program. A calling tree is a diagram that displays a tree structure. Nodes in the tree correspond to functions or procedures in the program. The children of any given node denote all of the subprograms directly called from the function or procedure denoted by that node.

The **NewYacc** rules for gathering calling information from a PL/M program are surprisingly simple, consisting of a few lines added to the PL/M grammar. Figure 3 shows PL/M grammar rule for procedure definitions with a **NewYacc** annotation added.

```
proc_def : proc_stmt block_body
           [ (CALL_TREE) #1 unset_proc_name() ]
         ;
```

Figure 3: **NewYacc** Rule for Procedure Definitions

The first line in Figure 3 is the **yacc** rule that specifies how to parse procedure definitions. A procedure definition consists of a procedure statement followed by the body of a block. Procedure statements and block bodies are defined elsewhere in the grammar description. The second line, delineated by square brackets, is a **NewYacc** rule. It indicates that when a calling tree is being constructed (CALL_TREE), two things should be done. First, the components of the procedure definition should be asked whether they have anything to contribute. "#1" corresponds to the result of analyzing the procedure statement, illustrated in Figure 4. The second thing that **NewYacc** does is to call a small C language function called *unset_proc_name*. This function is one of several responsible for remembering the name of the current procedure.

The second **NewYacc** rule for constructing calling trees is given in Figure 4. The first two lines are **yacc** grammar rules describing what a procedure statement looks like. The lines in square brackets indicate that **NewYacc** should output a record labeled with the words "PROCEDURE_DEFINITION". The remainder of the record is produced by a utility rule called *CTPN* that prints the current line and column numbers of the source file and then the name of the procedure being defined. *Set_proc_name* is another C function that helps remember the name of the cur-

```
proc_stmt : label PROCEDURE opt_formal_param_list
            opt_proc_type ol_proc_attribute ';'
            [ (CALL_TREE) "PROCEDURE_DEFINITION"
              #1(CTPN) "\n"
            set_proc_name( &1(PROCEDURE_NAME) ) ]
          ;
```

Figure 4: **NewYacc** Rule for Procedure Statements

rent procedure. Two similar rules exist for handling procedure call statements. The result of running the **NewYacc** calling tree tool is a series of records, each of which indicates that one function or procedure calls another.

Once the grammar for the PL/M language had been specified for **yacc**, adding the **NewYacc** rules took only a matter of minutes. One of the shell command files developed in the previous phase was used to apply the **NewYacc** rules to all of the source files. Other examples of the use of **NewYacc** rules are described in Section 2.4.

2.3 Phase 3: Integrating with a CASE tool

Reverse engineering the source code of a software system produces a high level description of it. This description may take a variety of forms depending on the intended purpose of the reverse engineering. Similarly, modern Computer Aided Software Engineering (CASE) products typically offer a variety of diagraming tools capable of displaying design artifacts using diverse representations. It is appealing to consider applying the power of CASE tools to the problems of reverse engineering.

We conducted a small exercise to test the feasibility of this approach. **Software Through Pictures** (**STP**) is a CASE tool developed by Interactive Development Environments, Inc. **STP** features a variety of graphical editors including one called the Structure Chart Editor that can be used to draw the style of diagrams used with Structured Design[6]. Structure Charts include information describing the calling hierarchy of a program. The exercise consisted of using the output of the **NewYacc** calling tree tool as input to the diagram editor.

Fortunately, **STP** has a very open architecture. Diagrams are stored in text files, and the format of the files is described in the reference manual. It was necessary only to convert the **NewYacc**-produced records into the format required by **STP**. This was accomplished by a short program that decided how to position the nodes in the the calling tree. The program took about three days to write, but, once again, it can be reused in any situation that requires the graphic placement of symbols in a diagram. The part of the program that understands the structure of the diagram file was encapsulated. By substituting similar modules, we intend to target other diagraming tools, such as EDGE[3] and SDT[2]. An example of a **NewYacc**-generated Structure Chart is given in Figure 5.

2.4 Other examples

While reverse engineering the digital subscriber system, we needed to be able to quickly build tools for a variety of purposes. Our strategy was to combine existing tools while building an infrastructure to support more elaborate activities. Among the tools that we built or intend to build using this approach are the following.

- A straightforward extension of the calling tree tool is to display parameters and to allow software maintenance personnel to build nested diagrams that place details in lower-level diagrams.
- We would like to add to the CASE version of the calling tree tool the ability to view source code by clicking on a node in the tree.
- The PL/M code consists of highly nested conditional and case statements used to implement a decision tree architecture. It is occasionally desirable to see the structure of the nesting without seeing the details. We call this a *code skeleton*, and a few **NewYacc** rules sufficed to produce this tool.
- **STP** provides tools to describe both state machines and decision tables. We would like to target these representations in a manner similar to what we provided for the calling trees.
- A more ambitious task is the automatic detection of semaphores. A semaphore is an integer variable typically used to control access to a critical section of code. The PL/M code makes heavy use of global variables for this purpose. Using **NewYacc** global variables can be detected when they are declared. Their later occurrence within the conditional part of an **if** statement suggests their use as a semaphore.
- **Lex** is a UNIX tool for describing the lexical properties of a programming language. It can be

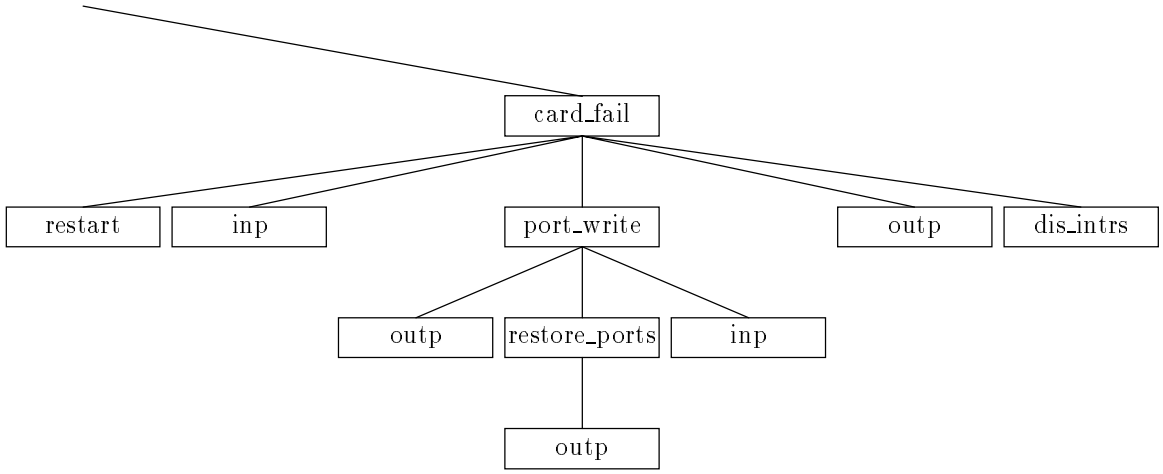

Figure 5: Part of a **STP** Calling Tree Diagram Produced by **NewYacc**

used to automatically generate a lexical analyzer from a regular expression description of the tokens in the language. We constructed a lexer for PL/M programs using this approach. Only 280 lines of code were required.

- In a similar manner, a parser was built for PL/M using **yacc**. The parser is capable of detecting syntax errors without requiring the overhead of a compilation, which in our case involved downloading files to a mainframe. The parser required 580 lines of code, including the 280 lines from the lexical tool.

- Another tool that we took advantage of was **plm2c**. This tool is freely available on the Internet and provides conversion capabilities for translating programs from PL/M to C. We used it to feed another tool called **DATRIX**. **DATRIX** is a program metrics tool. It takes C programs as input and produces a large collection of program statistics related to maintainability and testability of the input programs. By combining these two tools, we were able to obtain program metrics for PL/M programs without writing any additional code ourselves.

- Our most ambitious plans involve the concept of *partial evaluation*. Partial evaluation divides the inputs to a program into two categories, those which are relatively constant and those that are likely to change. In our case, we are interested in partially evaluating the PL/M programs themselves. In particular, in diagnosing a program fault, information may be available about the state of the program at the time of the fault. This information takes the form of values for certain of the program's global variables. We would like to see a simplified version of the program where those values have been "fixed". For example, if a global Boolean flag is known to be **FALSE**, then any statement dependent on its being **TRUE** should be removed from the program. Likewise, **if** statements that test the variable should be removed and statements in the **else** branch of the statement should be "promoted". The result will be a simplified version of the program that is easier to comprehend. **NewYacc**'s program transformation capabilities are directly applicable to this problem.

3 Applications of the strategy

3.1 Redesigning a component: slow line test restructuring

One of the tasks performed by the telephony software is Automatic System Test (AST). AST is responsible for testing transmission paths and line cards. Reverse engineering techniques were applied to the analysis of the code responsible for testing individual line cards, a set of procedures known to the product's software engineers as being particularly difficult to understand. This reverse engineering exercise was exploratory and did not begin with any well-developed conjectures about how line cards were tested. As a

consequence, the work concentrated initially on extracting data from the source code, and only later did a conceptual model describing the process of line card testing emerge. The conceptual model differed considerably in structure from the original code. The line card test was reimplemented around a new design based directly on the conceptual model.

In this exercise, we proceeded as follows:

- first, we collected and organized data extracted from the original source code;
- second, we allowed patterns in the data to remind us of more general designs and design problems with which we were familiar; and
- third, we collected and analyzed additional data to confirm the perceived pattern and fill in missing details.

Of course, we did not pick the right patterns on the first try and had to repeat the second and third steps until we had a consistent interpretation of the existing code.

There were several hypotheses associated with this analysis that proved troublesome and required additional investigation before they were confirmed. For example, the order in which two operations were performed was not the same in all paths, i.e., one path showed operation A followed by B and the other B followed by A. We were concerned that the difference in order was significant. By tracing calling chains and collecting global variable references, including accesses to physical devices, we were able to determine that the order did not matter. Since the operations commuted, the distinction between the two cases was eliminated, and the code shortened.

Exception handling proved difficult to untangle until we observed that many of the resources required for a test were shared among several tasks. This observation led us to ask how deadlock was prevented. Of the alternative strategies, it appeared that the system employed a version of the deny-hold-and-wait strategy[4], i.e., if a resource could not be obtained, all resources acquired thus far were released and the test was abandoned.

This analysis was not always straightforward, and two resources in particular did not appear at first to fit the hypothesized pattern. Specifically, we did not have any evidence that they were being released when exceptions occurred. By constructing and tracing the calling chains, we were able to determine that in the case of one important resource a paranoid strategy was employed: as soon as a low level procedure determined the test could not be completed, the resource was released, even before the exception flag was set or control returned to the main testing procedure. In the second case, we observed that along some paths a message releasing a resource was never being sent. While this failure to send a message appears to have been a bug in the original program, by examining various references to the resource, we found evidence that the bug had been fixed, though in a convoluted way. Rather than figuring out who was failing to release the resource, the maintenance programmer modified other components so that they would forcibly reclaim the resource without ever identifying and notifying the task currently holding it. The underlying assumption, correct but undocumented, was that the unidentified task was by that time no longer using the resource and had merely neglected to release it.

This example illustrates the importance of analyzing data extracted from the program text, e.g., constructing models of control flow, calling chains, and variable references.

3.2 Diagnosing and fixing a design error: the sleepy system problem

The software engineers responsible for the product, at the time they approached us, had a plausible hypothesis that a rare system failure was the coincidence of a fault in the software control of a watchdog timer and a stack overflow. Reverse engineering techniques were used to analyze low-level data to identify circumstances under which stack overflow was possible. Reverse engineering techniques were also used to abstract the actual code and thereby to construct a model demonstrating the interaction between stack overflow and the design fault in the timer.

This model was then used to test the efficacy of the repair proposed by the software engineers. The tests had to be run against the model because the combination of events leading to the system failure involved interactions among internal components and could not be recreated by simply manipulating the system's environment.

It is always difficult to estimate the stack requirements in large multitasking systems. Because of limited physical memory and the lack of sophisticated memory management, stacks should not be oversized. On the other hand, stack overflow is not acceptable. Hence, each process's stack must be accurately sized to accommodate the longest calling chain plus space for interrupt service routines.

Queries run against our cross reference data proved to be of particular value in this problem because analysis of calling chains is an important part of estimating stack requirements. When the code for each procedure and function was analyzed, an estimate was computed for the amount of stack space required. By constructing calling chains from cross reference data and then joining them with data on the size of the activation record for each call, estimates of stack usage were constructed. A similar procedure was used to calculate the stack requirements for interrupt service routines.

Software maintenance had over the years increased stack requirements, but the stacks had not been routinely resized because of the tedious nature of the calculation. After several years of maintenance, worst case combinations of interrupts on nearly full stacks, while a rare event, could cause stack overflow and system failure. With the new tools, derived from our reverse engineering work, it is now possible to periodically reestimate stack requirements and to confirm that maintenance has not reintroduced the possibility of stack overflow.

4 Conclusions

The strategy that we employed was the following. After understanding the information that needed to be extracted, we built a series of scripts using available text processing utilities. The scripts were tested and refined until they were successfully working on a sample of the system's source files. Then, using a shell command file that we built, the scripts were applied to all of the system sources. Occasionally, post-processing scripts were applied to further refine the data.

A grammar for PL/M was constructed in a form suitable for use by **yacc**. If the specific problem to be solved required more precise program analysis than that available from the text processing utilities, **NewYacc** rules were constructed and applied. When graphical display was desired, the data was transformed into a format understandable by the **STP** CASE tool.

The quick tools approach has an additional benefit besides its flexibility and responsiveness. Reverse engineering and program analysis are new areas of research, and a maintenance shop may be reluctant to undergo a large scale investment in unproven tool technology. Ad hoc tools, such as those described here, can quickly demonstrate tangible results. These results increase management confidence in the value of reverse engineering efforts.

Acknowledgement

We gratefully acknowledge the contributions made by Richard LeBlanc to the **STP** exercise.

References

[1] R. K. Fjeldstad and W. T. Hamlen. "Application Program Maintenance Study: Report to Our Respondents". *Proceedings GUIDE 48*, Philadelphia, PA, 1979.

[2] V. M. Markowitz and W. Fang. SDT: A Database Schema Design and Translation Tool Reference Manual. LBL-27843, Lawrence Berkeley Laboratory, May 1991.

[3] Frances Newbery Paulish and Walter Tichy. "EDGE: An Extendible Graph Editor". *Software-Practice and Experience*, Vol. 20, No. S1, pp. 63-88, June 1990.

[4] James L. Peterson and Abraham Silberschatz. *Operating System Concepts, Second Edition.* Addison-Wesley, 1985.

[5] James J. Purtilo and John R. Callahan. "Parse Tree Annotations". *Communications of the ACM.* Vol. 32, No. 12, pp. 1467-1477, December 1989.

[6] W. P. Stevens, G. J. Myers, and L. L. Constantine. "Structured Design". *IBM Systems Journal*, Vol. 13, No. 2, pp. 115-139, 1974.