Reverse Engineering with a CASE Tool

Bret Johnson
Research advisors: Spencer Rugaber and Rich LeBlanc

October 6, 1994

Abstract

We examine using a CASE tool, Interactive Development Environment's Software through Pictures (StP), to support reverse engineering. We generate structure charts in StP from the automated analysis of C source code. The advantages of this approach are that one can use the CASE tool's support for drawing, linking, and modifying pictorial notations for program design in order to make it easier to construct a reverse engineering tool. Additionally, one can then use the design representations with the CASE tool to do reengineering for maintenance.

1 Introduction

Reverse engineering, in the context of this paper, is the act of taking the source code for a program and extracting from it information about the design of the program. Reverse engineering is frequently required when an existing program needs to be rewritten or modified. Unfortunately, all too often such programs have been written with little attention to the principles of software engineering such as providing adequate internal and external documentation.

Computer Aided Software Engineering (CASE) tools are software tools that are intended to support forward engineering–the transformation from requirements to design to source code that occurs when writing new software. CASE tools frequently allow the the software engineer to represent the design of his program pictorially, through notations such as structure charts.

This work examines using a CASE tool, Interactive Development Environment's Software through Pictures (StP), intended for forward engineering, for reverse engineering. StP supports many different pictorial notations for representing a program's design at a level more abstract than source code.

These notations include entity relationship diagrams, structure charts, data structure diagrams, and data flow and control flow diagrams. StP is a typical CASE tool, for most other CASE tools also support these notations. One typically thinks of the design information in these notations as coming from the software engineer; he takes ideas in his head about how the program should be designed and from these ideas draws the diagrams that represent the program design. However it is also possible that these diagrams could be generated from source code analysis of an existing program. This source code analysis could be either completely automated or partially automated and partially requiring user interaction. Therefore, given the source code for some program, one could construct a pictorial representation of the design of the program. This pictorial representation could be viewed and modified using StP. And thus StP could be used to support reverse engineering.

In our work we generate structure charts from analysis of C programs. The analysis is mostly automated, though the reverse engineer can control the analysis in a few different ways. Section 2 describes the details of our reverse engineering tool. Section 3 explains how to run the tool here at Georgia Tech. This section tells where the required files are located and exactly how they are used. Section 4 examines possibilities for future research. And Section 5 gives concluding remarks.

2 The Reverse Engineering Tool

Our tool generates StP structure charts by analyzing C source code. The structure charts generated by our tool are basically call trees. They say what functions call what other functions. The structure chart notation supports including other information in the structure chart, such as indicating if a function is called from inside a loop or a conditional statement or what the function's parameters are. Our tool, however, does not currently generate structure charts containing these additional notations.

Our tool works in four phases. It analyzes C source code using NewYacc, extracts the important parts of this analysis using Awk, generates a structure chart using a layout program written in C, and allows the reverse engineer to interact with the structure chart using StP. To better illustrate the actions of the various phases of the tool, we show the of the intermediate text files that are generated by the different phases when analyzing the simple C program in Figure 1.

The first phase uses an enhanced version the parser generator Yacc [1]

```
int a, b;

main()
{
  init();
  process();
}


init()
{
  init_a();
  init_b();
}


init_a()
{
  a = 10;
}


init_b()
{
  b = 20;
}


process()
{
  int   i;


  for (i = 0; i < a; ++i)
    printf("b == %d\n", b);
}
```

Figure 1: A Sample C Program to be Reverse Engineered

```
DCL:example.c:0:<1,5>-<1,5>:a
DCL:example.c:0:<1,8>-<1,8>:b
FDC:example.c:0:<3,1>-<3,4>:main
FRF:example.c:2:<5,3>-<5,6>:init
FRF:example.c:2:<6,3>-<6,9>:process
FDC:example.c:0:<10,1>-<10,4>:init
FRF:example.c:2:<12,3>-<12,8>:init_a
FRF:example.c:2:<13,3>-<13,8>:init_b
FDC:example.c:0:<17,1>-<17,6>:init_a
DEF:example.c:2:<19,3>-<19,3>:a
FDC:example.c:0:<23,1>-<23,6>:init_b
DEF:example.c:2:<25,3>-<25,3>:b
FDC:example.c:0:<29,1>-<29,7>:process
DCL:example.c:2:<31,9>-<31,9>:i
DEF:example.c:2:<34,8>-<34,8>:i
REF:example.c:2:<34,15>-<34,15>:i
REF:example.c:2:<34,19>-<34,19>:a
REF:example.c:2:<34,24>-<34,24>:i
FRF:example.c:2:<35,5>-<35,10>:printf
REF:example.c:2:<35,25>-<35,25>:b
```

Figure 2: Output of NewYacc Phase

called NewYacc [2]. NewYacc contains a number of features which make it easy to write a source code analyzer. These features are detailed in [2]. We augmented the C grammar provided with NewYacc to output information concerning the definition of and references to functions, variables, and labels. Only about 20 lines of the C grammar file needed to be modified and a few auxiliary functions written. The output of this NewYacc analysis on our sample program is shown in Figure 2. In the output "DCL" means a variable declaration, "FDC" means a function definition, "FRF" means a function reference (function call), "REF" means a variable reference, and "DEF" means a variable definition (assignment to a variable). The output also tells the file name and line and column numbers where each identifier appears and the level of nesting of that identifier's scope.

The second phase uses Awk [3], a programming language especially good for parsing simple text file formats and scanning for patterns. Our Awk

```
D main
R init
R process
D init
R init_a
R init_b
D init_a
D init_b
D process
R printf
```

Figure 3: Output of Awk Phase

program extracts from the output of the NewYacc phase just the information needed to generate structure charts. Our Awk program is only only three lines long. It simply picks out the lines of the NewYacc output representing function definitions and function references (function calls) and outputs the name of the function together with an "R" or a "D" indicating if this is a definition or reference. The output of this phase for our continuing example is shown in Figure 3. Note that all of the "R" functions appearing after a "D" function are functions called by that "D" function.

The third phase is the most involved. It is a C program that reads in the output of the Awk phase and converts it to a structure chart diagram in the format required by StP. The main task of the program is to lay out the structure chart in an aesthetically pleasing way, and thus this program is named **layout**. The program reads in the output of the Awk phase and builds a data structure describing the program being analyzed. This data structure is basically a list of subprogram objects, each subprogram object describing one subprogram (the layout program was designed in an object-oriented fashion). Each of these subprogram objects contains, as one of its components, a list of the subprograms that this subprogram calls. This data structure can be viewed as a call tree. The subprograms are the nodes in the tree and the subprograms they call are the children of these nodes. It is the user's responsibility to specify which subprogram should serve as the root of the tree. Typically, the user will specify "main" as the root for a C program. Currently, the Smalltalk prototype for the layout program allows the user to specify any subprogram name he wants for the root of the tree,

while the C version of the layout program always assumes that "main" is at the root.

After this data structure is built, the program proceeds to create the StP structure chart diagram file. This operation proceeds in two steps. First, a postorder traversal of the call tree is made. The root of each subtree of the call tree is assigned a number giving the number of units of width that that subtree is allotted in the StP structure chart diagram. Because this is a postorder traversal of the tree, the assignments are made bottom up. Leaves are assigned enough space for a single box in the diagram plus a little extra space so that boxes don't touch each other. A parent node's width assignment is equal to the sum of the width assignments of its children. Next the program does a preorder traversal of the call tree, printing out in the StP structure chart file format each node in the tree and the arc between that node and its parent. This output for our continuing example is in Figure 4. A description of the StP structure chart file format can be found on pages 9-39 – 9-41 of [4].

The reverse engineer has some control over the generation of the structure chart: he can specify predefined subprograms and subprograms to be ignored. Note that these two features are present in the prototype of the layout program, written in Smalltalk, but they have not yet been implemented in the final C version of the layout program.

The reverse engineer can specify that certain subprograms are predefined. Predefined subprograms are subprograms whose implementations are not part of the current program being analyzed. For example, the standard library functions in C should probably be specified by the reverse engineer as being predefined. Structure charts represent predefined subprograms in a different way, with two lines on the sides of their boxes instead of one.

The reverse engineer can also specify that certain subprograms are to be ignored. He might choose to ignore subprograms that do an insignificant part of the computation so that these subprograms do not appear in the structure chart and are thus not distracting. The program will simply skip over ignored subprogram nodes and all of their children when traversing the call tree.

The final phase is to read the structure chart generated by the layout program into StP. Once in StP the reverse engineer can pan over and zoom in on the diagram. He can also make modifications to the diagram, perhaps to increase its aesthetics. The structure chart for our example, as printed by StP, is shown in Figure 5.

```
scefile6
6  5 500
0
1
101  14
1000  200
1
main
1.00 1.00
0
0
1
102  14
904  328
1
init
1.00 1.00
0
0
1
103  14
808  456
1
init_a
1.00 1.00
0
0
2
102 103
2
1
104  14
1000  456
    .
    .
    .
```

Figure 4: Partial Output of Layout Phase

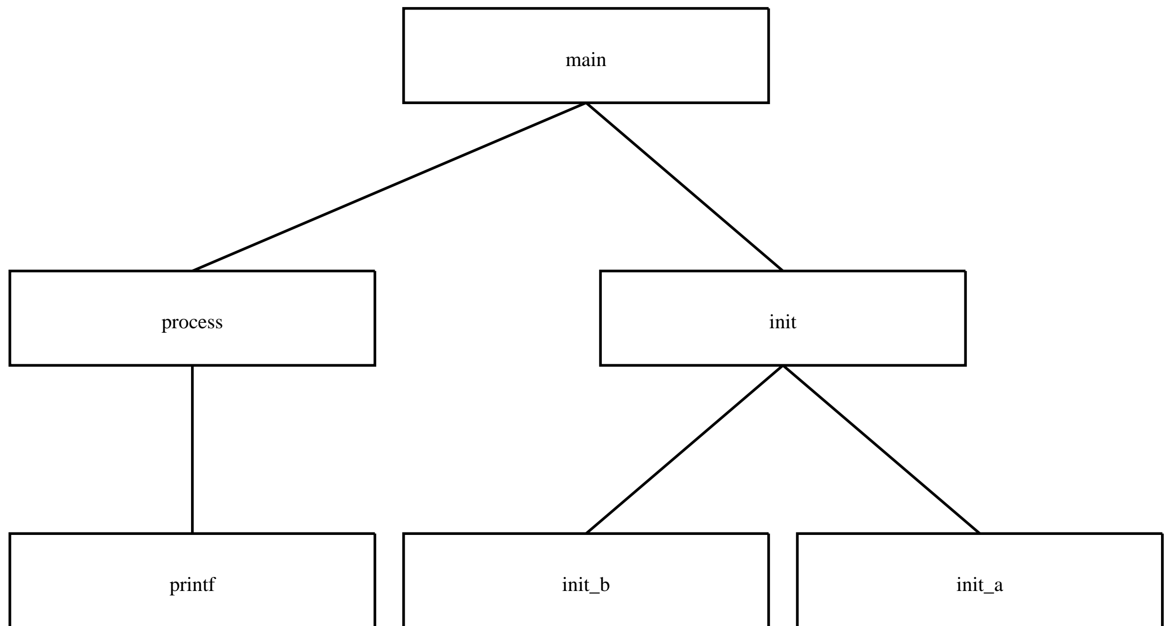

Figure 5: Final Structure Chart

3 Using Our Tool at Georgia Tech

For those readers at Georgia Tech who wish to run our tool, we now describe exactly what steps you must go through.

All of the tools are compiled for Sparc machines (though they can be compiled for other machines), so log into a Sparc machine. Change to the **CToStructureChart** directory, in whoever's account is keeping these files. For simplicity, we will work entirely out of this directory. Copy all of the C source files that you wish to reverse engineer to this directory. Run the C preprocessor (**/lib/cpp**) on these source files and save the resulting files. Next type **makeSC** followed by the names of the files preprocessed above. **makeSC** is a four-line shell script taking as many arguments as you choose to give it, each a file name. **makeSC** concatenates all of the argument files (concatenating several C source files gives a valid C source file), and runs the resulting large file through **canal** (the NewYacc source analyzer), **filter.awk** (the Awk program), and **layout** (the C layout program). The final result, an StP structure chart file, is written to standard output, so you should redirect it to a file. You should give the structure chart file a name

ending in **.sce**, StP's standard naming convention for structure chart files.

Now all that remains is to read the structure chart diagram into StP. Start the StP Main Menu, select the SCE icon, fill in the name of the structure chart file in the "Diagram:" field, and press the "Execute" button. Now you are free to manipulate the structure chart as you please.

4 Possibilities for Future Work

In the future we wish to investigate how StP can be used interactively as a reverse engineering tool. In particular, the structure charts generated by our tool are often too big. When printed on a single sheet of paper, these structure charts are too small to read. Big structure charts need to be broken down into a group of linked smaller charts. We envision a tool where the reverse engineer can interactively break a large structure chart into linked smaller charts using the StP structure chart editor. Unfortunately, the standard StP structure chart editor does not have facilities built in for breaking apart structure charts. We hope to find an effective way to add such functionality to the structure chart editor. There are a few possible methods of accomplishing this task that we are considering. Perhaps one of these methods will be effective and appear in a future paper.

5 Conclusion

We have successfully demonstrated that a CASE tool can be used to support reverse engineering. Our structure chart generator appears to be a genuinely useful tool for a reverse engineer. It was much easier to write this tool to work in conjunction with StP than it would have been to write a stand alone, interactive structure chart generator and viewer. NewYacc has also proven to be a handy tool, making it relatively simple to write source code analyzers.

We verified that StP implements the open architecture its makers advertise. Without its simple ASCII text file format for structure chart diagram files, a file format described thoroughly in the *StP User's Manual*, writing our tool would have been much more difficult.

This principle disadvantage of building a reverse engineering tool on top of a CASE tool is that the CASE tool's architecture is not totally open. For example, in StP the user cannot completely redefine the user interface for the structure chart editor. If one is interested in developing a high

quality, commercial reverse engineering product, this lack of a complete flexibility in being able to change the functionality of the CASE tool could be an argument for writing the entire reverse engineering tool from scratch. However, CASE tools, especially StP, can be customized in a fairly large number of ways. Thus for developing inhouse reverse engineering tools, using a CASE tool appears very promising.

References

[1] S. Johnson, "Yacc: Yet Another Compiler-Compiler," Bell Laboratories, 1979.

[2] E. White, J. Callahan, and J. Purtilo, "The NewYacc User's Manual," University of Maryland.

[3] A. Aho, B. Kernighan, and P. Weinberger, "Awk – A Pattern Scanning and Processing Language," Bell Laboratories.

[4] Interactive Development Environments, *Software through Pictures User Manual*, 1988.