CORE
Metadata, citation and similar papers at core.ac.uk
Provided by University of Lincoln Institutional Repository

Using Automated Source Code Analysis For Software Evolution

Liz Burd and Stephen Rank,
Department of Computer Science,
University of Durham
{*Liz.Burd,Stephen.Rank*}@*durham.ac.uk*

Abstract

Software maintenance is one of the most expensive and time-consuming phases in the software life-cycle. The size and complexity of commercial applications probably present the greatest difficulty that maintainers face when making changes to their applications. As a result of the corresponding loss of understanding, business knowledge encapsulated within the system becomes fragmented, and any changes made as a result of new business initiatives become difficult to implement and hence may mean a loss of business opportunities.

This paper outlines an approach to regaining understanding of software which has been used in the Release project at Durham University. This approach involves determining the calling structure of a program in terms of a call-graph, and from this call-graph extracting a dominance tree. Various problems which have been encountered during the construction of tools to perform this task are described.

1. Introduction

Software maintenance is one of the most expensive and time consuming phases in the software life-cycle. However, despite its obvious importance, it is invariably given little emphasis by those working on software applications. Software is perceived as a flexible medium of which change is both feasible and simple. In reality this could be, but is not true. The prolonged and expensive maintenance phase is recognised as being detrimental to the overall maintainability of applications and so the flexibility and adaptability that software may have is soon lost. During this process applications are said to be gaining legacy properties. As a consequence of this rift between the desired and actual properties of our software applications, ways are sought of ensuring that the continued change of applications, their evolution, progresses in a manner which ensure their continued flexibility.

The costs of performing maintenance has been estimated to be 70 to 90 percent of the total life-cycle cost [10]. Furthermore, the costs of performing program comprehension have been widely cited as being between 50 and 90 percent of the overall cost of performing maintenance [10]. Assuming that these figures are accurate, we can estimate that the costs of performing comprehension for maintenance over the life-time of software could account for approximately 35 to 80 percent of the software life-cycle costs (*i.e.*, development and maintenance costs). Clearly, this is an important activity so any approach towards assisting the comprehension process can considerably reduce software costs.

Legacy systems have, embedded within them, a large investment made by the systems developers and/or owners. This investment ranges from low-level code items or objects through to higher level business objects. However, the structure imposed by designers and developers of software systems are, in general, not suitable for continued change and evolution. Part of the reason for this is that during development it is difficult, or even impossible, to predict how the system is going to evolve. The fact that the software must continually change over time, or become increasingly less useful, was emphasised by Lehman [1]. He also pointed out that the structure of evolving software will degrade unless remedial action is taken. This has also been confirmed and shown visually by Burd and Munro [4]. An important aspect of this loss of structure is that business knowledge encapsulated within the system becomes fragmented, and any changes made as a result of new business initiatives become difficult to implement and hence may mean a loss of business opportunities.

The size and complexity of commercial applications probably present the greatest difficulty that maintainers face when making changes to their applications. To assist the comprehension process, maintainers need timely and selective information with regard to their programs. Biggerstaff [2, 3] and Rich and Wills [9] carried out research on design recovery, where the software architecture was represented using techniques such as data-flow graphs, control-flow graphs, call graphs, structure diagrams and cross ref-

erence tables. Although these representations are useful for small programs, little attempt has been made towards achieving abstraction in order to simplify the representation and allow generalisation. A deeper analysis—for example using dominance relations on call graphs [7]—of some of these representations can lead to a greater understanding of the code [5].

The study of software evolution has been identified as a critical aspect of ensuring that the maintainability of software applications is retained or even enhanced. Not only does it provide an abstraction of the information required for the comprehension process, but it also shows the evolutionary path of a software application thus enabling a maintainer to evaluate changes based on more evidence. For instance, errors identified can be first investigated with regard to those areas in the software which have previously been shown to be at fault. Furthermore, the studies of evolution can identify specific locations in the software where considerable maintenance has been performed and therefore can be used in the targeting of preventative maintenance.

The Release project has identified that the calling structure of the code is one of the most frequently used structures within source code as a mechanism for gaining an understanding of that code. This research has also highlighted that this calling structure is also the most likely starting point which maintainers will use when initiating the comprehension process of code that is unfamiliar to them. Thus as a means of making the comprehension process of evolution accessible to maintainers these familiar constructs are adopted as a means of visually representing the code.

This paper will describe an approach which has been used to study commercial applications. The problems (such as problems writing tools to process this language) that have been identified through this study and the preliminary results that have been obtained are also described.

The language which is being studied is the user language from a database system known as 'Model 204', first produced by the Computer Corporation of America[1] in 1965, and further developed since then. The user language is used to construct database queries and to provide user interaction. It is more than a query language, as it can be used to write general-purpose programs.

Section 2 describes the means by which the calling structure of a program is extracted from the source. In section 3, the process for determining dominance trees is outlined. Results and conclusions are presented in sections 4 and 5 respectively.

2. Call-Graph Extraction

The overall approach adopted within Release is through a stepwise method. The first of these steps involves the generation of a call graph. The objective of this step is to identify the calling structure of the code on a procedural basis. This step consists of two tasks. These are:

Check preconditions for producing a call graph The generation of a true calling structure of the software relies on each procedural unit being logically distinct from other procedures. There should be no use of constructs so as `GO TO`s or 'fall through' to other procedures (as is common in COBOL). If any of these conditions do not hold then the code should be restructured before the call graph is generated.

Generate the 'Perform' graph A perform graph is constructed from the call graph for each procedural unit. For each call in a procedural unit, a link is made between the calling and called procedural unit. These caller/callee relationships are referred to as the call graph pairs. Where more than one call is issued to a particular procedural unit within one procedural unit, then the duplicate call is recorded numerically. A call graph is a call-directed graph (CDG). A call-directed graph of a code module is formally defined as a directed graph $CDG = (N, E)$ where $N = S \cup PP$ is the union of S, the set of the entry procedural unit(s), and PP, the set of all procedural units, and E is the call relation $(S \cup PP) \times PP$.

Call-graphs are represented in the 'CLL' file format; each line in a CLL file is of the form "`parent : child`", where `parent` and `child` are the two nodes connected by a directed edge.

The process of call-graph-extraction is done by parsing the source code (using a parser generated by the Bison parser generator). This section describes the process in more detail.

2.1. Source Files

One of the first problems of analysing software for evolution is finding a suitable parser. This project involves analysing Model 204, a language for which there are no suitable parsers available (the only previously known tool is the interpreter used by the database system itself). Therefore it was decided that the most appropriate solution was to build a parser to extract the appropriate constructs.

The Model 204 source files from each release have been concatenated into a single file, numbered according to release sequence number. Source programs have the format shown in figure 1.

2.2. Call-Graph Extraction Methods

This section describes the means by which call graphs are extracted from source files.

[1] `http://www.cca-int.com/`

Preamble Including global variable declarations, file control commands, *etc.* (optional).

Main Program of the form:
`BEGIN`
Main program statements
`END`

Procedure (Subroutine) Definitions (optional); of the form
`FOO: SUBROUTINE` or
`SUBROUTINE FOO(%BAR IS` *type*`,...)`
`...`
`END SUBROUTINE` $name_{\mathrm{opt}}$

Figure 1. Format of Source Files

Initially, the calling structure of source was determined manually [8]. These pilot studies highlighted the benefits of the analysis process for the study of evolution. A number of important points have been identified from this study:

- The process provided essential information to maintainers that they would not have otherwise been aware of. From interviewing maintainers of these commercial applications, it was clear that in some instances the implementation approach that was adopted by the maintainer would not have been considered if they were able to see the consequence of a specific change.
- The manual approach was costly in terms of time and prone to errors; for commercial use the approach must be automated to prevent these problems and produce timely information.
- Earlier feedback would be more beneficial. The manual approach analysed only completed system changes. To maximise the benefit of the approach what is needed is daily feedback on changes. In this way maintenance approaches can be adjusted depending on the resulting evolutionary trend. Thus it is proposed to use the automated system to analyse each nightly build. However, this intensifies the problems experienced with the manual approach highlighted above.

In order to increase the efficiency of this process, an extraction tool is in the process of being written. This tool is being created using the parser-generator Bison, which generates an LALR(1) parser from a definition script (using the same input language that YACC accepts). There are several issues which have been discovered while writing the tool, most of which have been problems with the source language and its documentation. Some example problems are described in the next section.

2.2.1 Example Language Problems

Block structure According to the language documentation[2], '`IF`' statements have the traditional (nested) syntax, given here in slightly modified B.N.F. (using C-style comments):

```
if_statement   →  IF condition THEN
                  statements
                  else_part_opt
                  END IF

else_part_opt  →  /* empty */
else_part_opt  →  ELSE statements
else_part_opt  →  ELSEIF condition THEN
                  statements else_part_opt

statements     →  .../* including if_statement */
condition      →  ...
```

However, figure 2, which contains code taken from release 381, line 7839*ff*, edited for brevity (elisions indicated with '`...`'), illustrates a nested IF (marked with `[*]`) which has no corresponding `END IF` of its own (it is instead ended with the `ELSEIF` marked with a '`[#]`').

```
     IF  %SALE:TYPE = 1 THEN
         ...
     ELSEIF %SALE:TYPE = 2 THEN
         ...
[*]      IF  %DEPT NE %SALE:DEPT THEN
             %ABORT = 24
             CALL ABORT
         ELSE
             %QTY = %QTY + %SALE:QTY
             ...
[#]  ELSEIF %SALE:TYPE = 9 THEN
         ...
     ELSE
         ...
     END IF
```

Figure 2. A Code Fragment Illustrating a Single '`END IF`' Statement Taking The Place of Many.

2.2.2 Comments

The language documentation[3] indicates that comment lines take the form of an optional label (of the form '`FOO:`') followed by an asterisk, and then the text of the comment.

[2] "You must end the IF statement with an END IF statement or an END BLOCK statement." [6, page 11-3].
[3] [6, page 2-12].

However, in some of the source files, a comment starts after a statement, in a form not explicitly mentioned in the documentation. For example:

```
END IF   *** end of processing
```

This problem manifested itself when the initial '*' was interpreted as the multiplication sign, producing a parse error. In order to get around this problem, multiple asterisks are interpreted (in the tokeniser) as beginning comments, even if they're not at the start of a line.

2.2.3 Example Documentation Problems

As documentation is not automatically generated, it is frequently not kept up-to-date with changes to the code. For example, the `CHANGE` statement is specified in one part of the documentation as

```
CHANGE fieldname
[= value] TO newvalue
```

but in another as

```
CHANGE fieldname [(subscript)]
 = value1 TO value2
```

This conflict was only resolved by referring to examples in the source, and turns out to be a mixture of the two:

```
CHANGE fieldname [(subscript)]
[= value] TO newvalue
```

3. Dominance Tree Generation

When studying large commercial applications it is necessary to provide a means of information abstraction, such as calling structures, to assist the maintainer in the comprehension process. However, when studying evolution of large applications this problem becomes even more acute. When the changes of a considerable number of an application's versions are being considered then the timely and accurate extraction and abstraction of appropriate information becomes essential.

Research on the Release project has identified that commercial applications sometimes have several hundred nodes and several thousand arcs. Within Release, studies of up to 30 versions of a single software application have been conducted. Thus a means of information abstraction is an essential aspect of the analysis process if the information presented is to be understandable and usable to the maintainer. The dominance tree is used to provide an abstraction of the information required to study evolution. The dominance tree is a way of abstracting the call graph, but in addition represents high-level modularisation of the software applications through its branches. Each branch of the dominance tree represents a concept or high level function of the system.

The problem with call graphs for large commercial applications is again their size and complexity. Research on the Release project has identified that commercial applications sometimes have several hundred nodes and several thousand arcs. The dominance tree provides an abstraction of this information and is thus the second step of the Release method. The objective of this step is to identify the dominance relationships between procedural units.

A dominance tree is created from a call-directed-acyclic-graph (CDAG). Two relationships can be identified between the nodes of the graph. These are strong and direct dominance [7].

This step is composed of three tasks. These are:

Remove cycles from the call graph Cycles can occur within the source code between one or more nodes. Cycles can be removed by collapsing every strongly connected subgraph (those nodes contained within a cycle) into one node.

Identify entry points Sometimes more than one entry point exists within a single source code module. This means that a number of sets of dominance tree relationships can be identified from a single code module for each of the entry points within the code. Procedural units within more than a single dominance tree should be marked as special cases for analysis during reconstruction.

Generate dominance relations The dominance relations are obtained in the following way. In a CDAG, a node px dominates a node py if, and only if, every path from the initial node s of the graph to py includes px. In a CDAG, a node px directly dominates a node py if, and only if, all the nodes that dominate py dominate px. In a CDAG, there is a relation of strong direct dominance between the nodes px and py if, and only if, px directly dominates and it is the only node that calls py.

Dominance trees are represented in the CLL format, with the extension that lines are of the form "`parent : child [: {d,s}]`", with `d` or `s` representing direct or strong dominance respectively.

As with call-graphs, dominance trees were initially extracted using manual techniques[8]. Again, manual processes have been automated, leading to an improvement in the time taken to generate dominance trees.

Call-graphs are converted to dominance trees using a tool written in a mixture of Haskell and C[4]. The tool accepts as input a call-graph (in CLL format), and carries out the following steps:

[4] A tool, written in C, for converting paths (described later) into dominance tree was already available.

Cycle Collapsing All nodes involved in a cycle are collapsed into a single node (optionally containing the names of all the nodes). This is because the following steps require acyclic graphs. Cycles are recursively collapsed, until the graph is acyclic. An example is shown in figures 3 and 4.

Path Determination Root nodes in the call graph are identified, and then the list of paths from each root node are determined. This is shown in figure 5.

Dominance Tree Production Path lists are converted to dominance trees by considering the possible paths from a root node to the various other nodes in its tree. The dominance tree for the above-mentioned graphs is shown in figure 6.

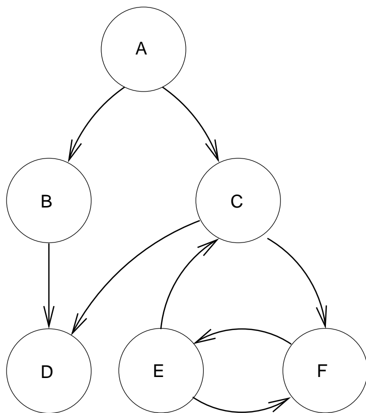

Figure 3. An Example Call-Graph

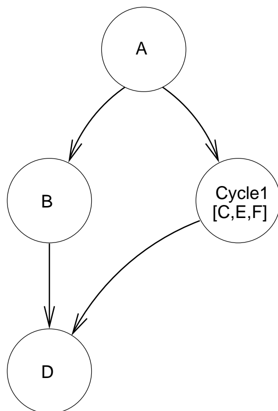

Figure 4. A Call-Graph With a Collapsed Cycle

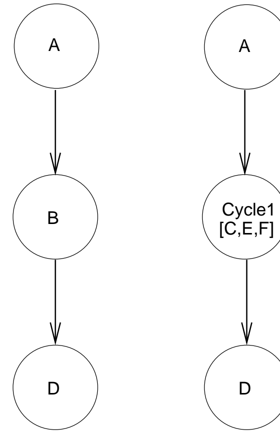

Figure 5. Paths in the Call-Graph

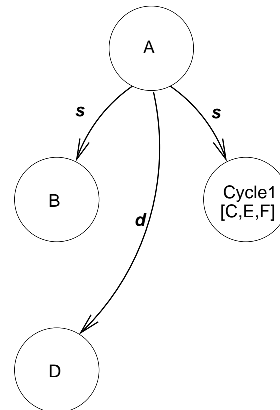

Figure 6. Dominance Tree

4. Results

The results of the pilot trials and tool development have identified some of the difficulties of supporting the evolution process. The development process has highlighted that over the lifetime of the software a number of factors are continually changing. For instance, not only does the software under study evolve but the language itself and company development house styles change. This increases the complexity of the support tool. However, gains obtained for the availability of such a support tool justify the commitment for its development.

In order to evaluate the suitability of the tool three evaluation criteria are considered. These are speed, accuracy, and usability.

As the previous system involved at least a partially manual analysis of the software applications, improvements in the operating of the tool based approach are to be expected. On average the speeds of the semi-manual process per version of code was approximately 30 minutes. The operating speed of the tool running on a standard desktop machine for the same software application has now been reduced to less than half a minute. However, the most significant improvement is the accuracy of the information.

Currently, the tools are invoked *via* a Unix-style command-line interface. Although a more graphical interface could be developed, the tools operate in a batch style (it is not possible, for example, to interact with the parser while it is processing a program), and could be used as part of automatic overnight processing, for which a GUI is not suitable. It would be difficult to justify a claim that, for example, a graphical drag-and-drop interface would be more productive or easier to use than the command-line equivalent. The users at whom the tools is targeted are experts in using the database system for which they develop. This system is primarily accessed through the command line, and so these developers will be familiar with this kind of interface.

The output of the tool has so far been a series of static dominance trees. The dominance trees are then represented using a graphical representation tool which has been developed as part of the Release project. The trees are then loaded either individually for detailed analysis or in series to allow comparison. To assist the comparison, process nodes (representing the procedures) are sorted alphabetically. The nodes are then laid out in a tree formation restricting crossing arcs to a minimum.

Particularly the comparison process has identified a number of problems. The first and most obvious of these problems is that of space. When a large number of versions are represented then it is not possible to represent them with clarity side by side on a screen. The problem is currently overcome by printing the results. The second and more interesting problem surrounds the change of software applications. As indicated above the nodes are laid out alphabetically and to reduce crossing arcs. The laying out of the nodes in alphabetical order ensures that the trees across versions looked the same. However, when nodes are added or deleted through the evolution process then it is harder to see the correspondence between the versions. Furthermore when additional calls are added between existing nodes then it may be necessary to relax the crossing lines weighting on the sort algorithm if the ordering of the nodes between versions is to be maintained. When significant changes are made throughout the lifetime of a software application then the maintenance of position tends to significantly reduce the readability of the resulting visualisations. Overall this is a complex problem that requires considerable further research.

Finally the results of the analysis of the commercial application seems promising based on the results of the pilot trials, however the results of the commercial use of the tool are yet to be completed. In many instances from the trials, it has been found that metrics are more suitable to represent the overall change process and when specific interesting anomalies are found in the metrics data small sets of the dominance trees are then used as a means of further detailed analysis. Thus support for the daily build process can concentrate on displaying the dominance trees from only the previous couple of days' builds.

5. Conclusions

The problems associated with software maintenance are both important and difficult to tackle. In particular, in this paper, some of the problems of source-code understanding have been outlined, and an approach to their solution has been described.

Various problems associated with building a parser for a little-known and aged language have been identified. These include both language and documentation problems. Studying systems which have been changed over many years, as is the case with the systems described here, brings further problems, as the programming language has changed, along with the programs. This leads to further complication in the grammars which are required.

Call graphs (and therefore dominance trees) can be inferred from source code (subject to resolution of the above-mentioned problems). This produces an overview of the system which is useful during program comprehension. It is hoped that further work will enable the use of these tools during development work, as a toolkit for developers and maintainers, in order to decrease costs associated with program understanding.

The next phase of the project will be to extend the tool for the support for data analysis. This will again need to ensure that appropriate abstraction mechanisms are provided

to ensure that the maintainers are able to early comprehend and identify the important information.

More long term research aims concern issues highlighted above such as maintaining placement of nodes. One proposed solution to this problem revolves around the animation of evolution process. Currently the animation of metrics have been attempted and show promising results. However the animation of the dominance trees is a more complex problem but seems to be worthy of pursuit. Finally it is intended to conduct a more detailed analysis of comments, for example to extract meaning from them to support the concept assignment process for branches of the dominance tree. This will therefore lead to the first steps of automated documentation of the evolution process where descriptions can be obtained as to the functional changes of software over time.

References

[1] On understanding law, evolution and conservation in the large program, life cycle. *Journal of Systems and Software*, 1:213–221, 1979.

[2] T. J. Biggerstaff and A. J. Perlis. *Software Reusability; concepts and models*, volume 1. ACM Press, 1989.

[3] T.J. Biggerstaff, B.G. Mitbander, and D. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–83, 1994.

[4] E.L. Burd, M. Munro, and C. Wezeman. Analysing large cobol programs: The extraction of reusable modules. In *Proceedings of the International Conference on Software Maintenance*, November 1996.

[5] A. Cimitile and G. Visaggio. Software salvaging and the call dominance tree. *Journal of Systems Software*, 28(2):117–127, 1995.

[6] Computer Corporation of America, 36-38 Market Street Maidenhead, Berkshire England SL6 8AD. *Model 204 User Language Manual Parts I and II Version 4 Release 2.0*, February 1999.

[7] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland, New York, 1977.

[8] Sonata Pakstiene, Liz Burd, and Malcolm Munro. Black magic recipe for generating dominance trees from COBOL (or any other programming language) source code. Technical report, Department of Computer Science, University of Durham, 2000.

[9] C. Rich and L.M Wills. Recognising a program's design: A graph-parsing approach. 7(1):82–89, 1990.

[10] T. A. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, 10(5), 1984.