Identifying Redundancy in Source Code using Fingerprints

J Howard Johnson

Abstract

A prototype implementation of a mechanism that uses fingerprints to identify exact repetitions of text in large program source trees has been built and successfully applied to a legacy source of over 300 megabytes. This prototype system has provided useful information as well as establishing the scalability of the technology. The approach will form the basis of a suite of tools for the visualization and understanding of programs and will complement other approaches currently under investigation.

1 Introduction

A large body of source code can be intimidating when first viewed. As the size approaches tens of thousands of lines it becomes infeasible to learn much by starting at the beginning and attempting to trace through various possible flows of control. Looking at a randomly chosen file in a large source provides about the same amount of global understanding as closely studying one tree in a forest. If the tree is like other trees, at best one can learn a lot about the structure of trees. However, many of the subtleties of the forest ecosystem will be missed because they involve complex interactions involving many trees. Clearly other techniques are needed.

One approach to understanding a large amount of source code involves mechanizing part of the process using tools that

- mechanically review and analyse the source and produce summary reports that provide insight into its structure, or
- transform the source into a more readable form, or
- allow interactive browsing to improve the effectiveness of the reader, or
- allow queries against the content to facilitate finding the relevant information.

Conventional compiler technology suggests a number of levels at which analysis can be done:

- *Raw Text:* The body of source is considered as a collection of files each of which is a sequence of characters formed into lines. In large systems, the files are organized into a file-system hierarchy or stored in a database to facilitate the management of module structure, versions for multiple platforms, and releases over time [1].
- *Preprocessed Text:* In languages with textual-level preprocessors (e.g., C or PL/I), the effect of applying the preprocessor to the collection of files causes macros and inclusions to be expanded. The content of individual files is now parsable by a compiler but a significant amount of structure may have been lost (e.g., manifest constants, inline functions, sharing of inclusions). The organizational structure now reflects how the compiler and loader will compile and build the applications.
- *Sequence of Lexemes:* The lexical analysis phase of a compiler has been applied to the

The IBM contact for this paper is Erich Buss, Software Engineering Process Group, 1 Park Centre, 895 Don Mills Rd., North York, Ontario M3C 1W3

NRC 37063

individual files, replacing their content by sequences of lexical items (lexemes).

- *Syntax Tree:* The sequences of lexical items have been analysed by a parser that interprets globally (within each file) a syntactically valid file-level module and produces a syntax tree [10, 12].
- *Annotated Syntax Tree / Symbol Table:* The syntax tree has been transformed to remove syntactic information that does not affect the meaning. Declarations have resulted in entries in the symbol table that identify properties of semantic entities. Occurrences of the entities in the syntax tree are correlated with the symbol table [4, 5].
- *Abstract Execution:* Particular control and data flows through the logic of the program are studied to arrive at an understanding of what the code does [9].

A number of issues affect the utility of analysis at each level:

- The more syntactic and semantic analysis is done, the more the understanding obtained corresponds to the *meaning* of the code and the less to the *form* and *structure* provided by the original designers. The nature of the insight required affects the choice of stopping point.
- Incidental artifacts appropriate for archeological design recovery [2, 6] are removed by syntactic and semantic analysis; for example, artifacts of cut and paste activities are progressively hidden by the above steps. For such purposes, less analysis is better.
- Syntactic and semantic analysis requires the existence of tools that understand the programming language and environment. Such tools cannot easily be taken to an environment where the syntax or semantics are different. Sometimes even small changes in language or environment can make such tools totally unusable.
- Some forms of source understanding (e.g., control flow and data flow analysis) require access to the semantics of the code and are not possible without deep analysis.

This research is focused on a *full-text* view of source using either the raw or preprocessed text. Tools that treat source simply as text are useful. Such tools look at the source the way the programmer does, retain more of the design information as understood and expressed by the programmer, and retain more artifacts of the implementation processes when compared with other methods. Although full-text tools do not support the whole process of source text understanding, they do support an important part of it.

2 Understanding Programs using Redundancy Analysis

2.1 Objective

The overall objective of this research is to help software maintainers understand the source code of large legacy systems for the purpose of reliable maintenance or re-engineering to improve maintainability or portability [2, 3, 4, 5, 6].

Repetition or redundancy provides a useful metric for several practical tools for program understanding. Noting which text occurs multiple times in a large source tree can facilitate understanding of the source.

2.2 What is Redundancy in Source?

We will give a general definition of redundancy before discussing the particular form exploited by the prototype implementation. This is partly to show how we are extending existing approaches and partly to indicate the opportunities that remain for further extension.

Definition: *Redundancy* is any characteristic of the source that could be used by a data compression algorithm to encode the source in fewer bits.

By characteristic we mean any asymmetry or repetition of content or structure that a data compression algorithm could detect and encode in a different way that requires fewer bits of information. Some examples of redundancy are:

- *Asymmetric use of the letters in the character set:* Since not all characters occur equally often in the source, a variable length encoding based on the observed frequencies will reduce the overall size of the source. Huffman and

arithmetic encoding exploit this type of redundancy.

- *Asymmetric use of program identifiers or symbols:* A variable length encoding of identifiers can be used. Huffman or arithmetic encoding on lexemes can be used in this case.
- *Local serial correlation in the usage of identifiers, symbols, or general substrings:* Dynamic Huffman encoding, dictionary-based schemes based on self-organizing data structures, and variants of the Ziv-Lempel algorithm are based on this type of redundancy.
- *Program schemas, idioms, and clichés representing patterns that recur in programs in the given language:* Compilers and source management tools often use a concise encoding of source.
- *Complete files that are expected to have many lines in common:* Source control systems take advantage of this type of redundancy to store multiple versions of a source in a fraction of the space needed for the unencoded source.

The amount of redundancy of a particular type can be measured by applying an appropriate data compression algorithm and noting the effect on the size. A finer breakdown of redundancy corresponding to particular identifiers, particular idioms, or particular pairs of files can be calculated by restricting the compression algorithm appropriately. By this, the structure of redundancy can be examined to provide useful insights.

This study involves looking globally at source redundancy to include long substrings in different parts of the source agreeing exactly. The corresponding data compression algorithms would reduce space by storing the repeated text once and replacing occurrences by pointers to the single copy.

The focus of this study is the comprehension of large source texts. We are interested in the location and structure of the long matching substrings that provide information about a number of aspects of source useful for program understanding [1].

2.3 Tools based on Redundancy Analysis

To describe tools based on textual redundancy is perhaps premature; however, program understanding might be supported by tools such as the following:

- *Analysis of cut-and-paste:* How much cut-and-paste activity is evident in the text? Is it affecting maintenance?
- *Analysis of the effect of preprocessing:* The compiler sees the result of preprocessing the source whereas the programmer sees the unprocessed source. How are they related? This could involve an assessment of the amount and location of change effected by the preprocessor or provide a basis for navigating the two views simultaneously.
- *Measurement and understanding of change between versions or platforms:* How different are the implementations for two platforms? How much change occurred between two versions and where was it located?
- *Generalizations of "diff":* A generalization of diff that allows interchanges of large blocks of code and analyses the contents of directories would be useful.
- *Generalizations of cross-referencing or KWIC (KeyWord In Context):* Cross-referencing tools expose dependencies of identifiers across modules. More general tools based on source text redundancy would allow other kinds of information to be cross-referenced.
- *Similarity-based navigation of the source tree:* Directory navigation tools are based on the structure of the directory tree. Similarity-based navigation would involve a visual presentation of the similarity of content of files and would facilitate moving among files that are similar.
- *Improved file browsing:* As the content of a file is browsed, blocks of repeated code could be highlighted and provide an intuitive access point to files sharing that block.
- *Understanding of lack of abstraction mechanisms:* Redundancy will often occur as a result of constraints imposed by poor source language abstraction mechanisms. Procedural abstraction (subroutines) and data abstractions are not adequate mechanisms for removing all kinds of source redundancy.

In addition to program understanding, global source text redundancy has other uses:

- *Data compression:* As outlined above, a data compression algorithm based on repetition of substrings content is easy to construct. This kind of algorithm would be quite effective whenever many similar versions of source are to be stored.
- *Information measures:* A more accurate measure of source text change would result from discounting the absolute count of number of lines changed by any increase in the overall redundancy in the source (as might occur through simple cut-and-paste).
- *Distributed configuration management:* If source is being maintained on more than one platform tools are necessary to identify and track differences efficiently.

Thus, the analysis of redundancy in source provides a framework for a number of useful tools.

3 Description of Prototype Implementation

3.1 Equality Testing using Fingerprints

The term *fingerprint* has been used to refer to a short string that can be used to stand in for a larger data object for comparison purposes. Although only the time and space performance of algorithms are affected through this substitution, the savings are large enough that problems that would otherwise exceed current technology can be addressed.

Definition: A *fingerprinting function* $f(x)$ maps data objects from some data-object domain D into a set of fingerprints F so that $f(x) \neq f(y)$ implies $x \neq y$ and $f(x) = f(y)$ implies $x = y$ with high probability. The set of fingerprints F is usually chosen to be short bit strings.

By deterministically calculating fingerprint values that are guaranteed to be the same for all equivalent data objects and are likely to be different if the data objects differ, one can make a single pass over the set of data objects to compute their fingerprints and thereafter use the fingerprints to test for equality. The smaller size of the fingerprints with respect to the original data objects makes the equality test much faster. To guard against a false match when fingerprints agree, the original data objects must be compared; however, with an appropriate choice of fingerprinting function the probability of this false matching can be made extremely low.

The approach taken in this research is to compute fingerprints for substrings of source text such that

- The probability of a false match occurring is so low that it can be ignored.
- The fingerprint calculation is portable and yields the same results on different platforms.
- The fingerprint calculation is capable of efficient implementation.

3.2 Searching for Redundancy

Fingerprinting can be used efficiently to find matches in source text using the following methodology:

- Fingerprint an appropriate set of substrings in the source. Record in a *fingerprint file* the location of each substring in the source together with the fingerprint of the substring.
- Sort the fingerprint file by fingerprint value and remove any records with fingerprints that occur only once.
- We are left with information about source text redundancy in the form of fingerprint values, each with multiple references into the source.

There remain several problems to be addressed:

- What is an appropriate set of source substrings?
- How can we efficiently calculate the fingerprints?
- How can the reduced fingerprint file be further processed to make it useful?

These questions will be considered in the remainder of this section.

3.3 Karp-Rabin Fingerprinting

Karp and Rabin suggest the following efficient way of calculating fingerprints for all length n substrings of a text [7, 8, 11]:

- Each sequence of n characters is considered as a numeral in some base r. Typically r equals 256 for 8-bit characters but other values are possible.
- The numeral encodes an integer value that can be calculated by multiplying the integers stored in individual bytes by appropriate powers of r and adding them up.
- The fingerprint is the residue (or remainder) of this integer when divided by a (fixed) number p.
- The value of the fingerprint will be an integer in the range 0 to $p-1$ and can be encoded in $\lceil \log_2 p \rceil$ bits.

The values of p and r can be chosen to reduce the probability of fingerprints agreeing when the substrings do not. Karp and Rabin choose r to be 256 and change p whenever a false match occurs as the text is being scanned. This is possible in their case because they are scanning a large text for the occurrence of a single character string. We do not have the option of changing p and so must be careful about the choice of p and r (see Appendix A).

When fingerprints are being calculated for all substrings of length n in a text, an incremental algorithm is much more efficient than the obvious separate evaluation approach. It is based on the observation that the integer calculated from the substring at some offset i is easily computed from the integer calculated at offset $i-1$. Because of the properties of residue arithmetic, this calculation can be done on residues to produce the new fingerprint.

More precisely, consider the sequence of characters starting at position i:

$$c_i c_{i+1} \cdots c_j \cdots c_{i+n-1}$$

This corresponds to the integer value:

$$c_i \cdot r^{n-1} + c_{i+1} \cdot r^{n-2} + \cdots + c_j \cdot r^{(i+n-1)-j} + \cdots + c_{i+n-1}$$

When this is reduced mod p we get the ith fingerprint f_i:

$$f_i = \left(\begin{array}{l} c_i \cdot r^{n-1} + c_{i+1} \cdot r^{n-2} + \cdots \\ \qquad + c_j \cdot r^{(i+n-1)-j} + \cdots + c_{i+n-1} \end{array} \right) \bmod p$$

Then f_i can be computed incrementally from f_{i-1} using:

$$f_i = \left(\left(f_{i-1} \cdot r \right) + c_{i+n-1} - c_{i-1} \cdot \left(r^n \bmod p \right) \right) \bmod p$$

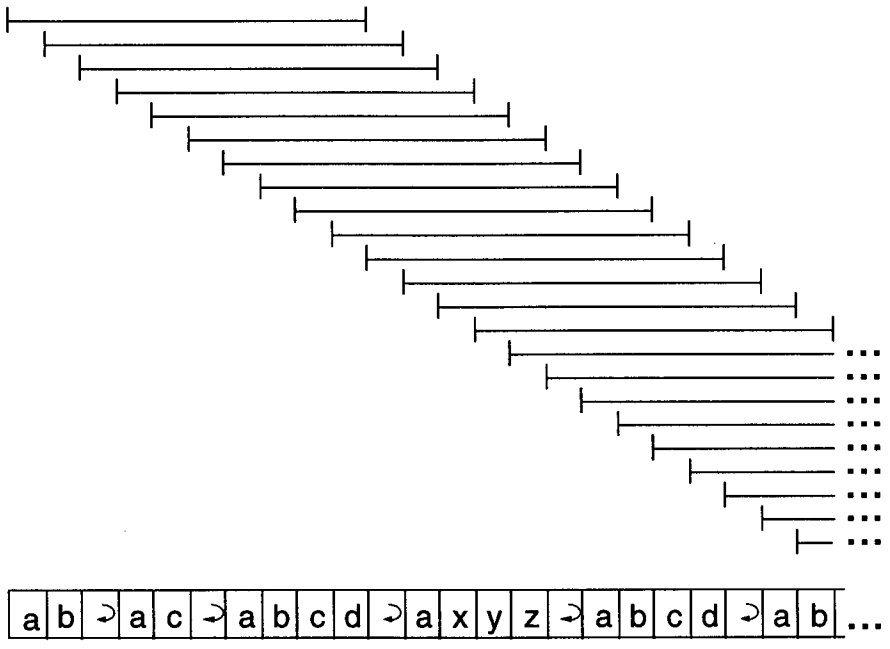

Figure 1. Snips of length 10 characters

Here $(r^n \bmod p)$ is a constant that can be precomputed.

3.4 Handling Line Boundaries

In practice, we are not interested in *all* substrings in a source text (as in Figure 1). Source code is usually organized in lines (in Figure 1 indicated by arrow characters). A program that analyses source should recognize this reality and allow parameters to be specified in terms of lines and information to be reported in terms of lines; thus, substrings that start and end on line boundaries are preferred (as in Figure 2). The strategy should behave reasonably when very long lines or very short lines occur.

- the minimum acceptable number of characters m in a snip.

In practice, since source lines are of various lengths, the optimal size of a snip is neither a fixed number of lines nor a fixed number of characters. Short lines increase the desired number of lines in a snip; long lines reduce it.

Each of the following will be a snip:

- every sequence of l lines that is between m and M characters long,
- every sequence of more than l lines that is shorter than M characters but that cannot have the first line removed without falling below m characters.
- every length M substring of sequences of l lines that exceed M characters,
- every length M substring of sequences of more than l lines that exceed M characters but cannot have the first line removed without falling below m characters, and
- the whole file if it has fewer than l lines and M characters.

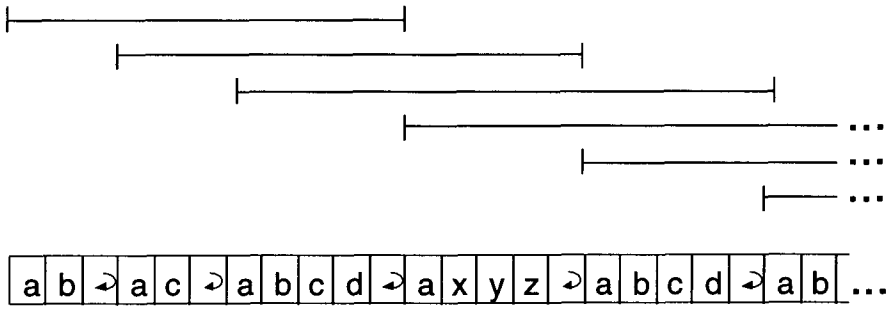

Figure 2. Snips of length 3 lines

To facilitate discussion, we will introduce a new term:

Definition: A *snip* [1] is an identified sequence of characters in a source file. It is identified by file name, beginning offset, ending offset, and has as content the substring so identified.

Definition: Two snips *match* if their contents agree.

Line boundaries will be brought into the system by defining using three parameters the set of snips that will be fingerprinted:

- the desired number of lines l for a snip,
- the maximum acceptable number of characters M in a snip, and

Several observations can be made:

- Setting the $M = m$ gives the original Karp-Rabin character strategy.
- Setting the m to zero and M very large gives a pure line count strategy.
- This strategy can be accommodated using an extension to the Karp-Rabin incremental fingerprinting algorithm.

Two other modifications have been made to the original Karp-Rabin fingerprinting algorithm:

- Each character is incremented by one before being used in the fingerprint to avoid the equality of fingerprints for snips that differ by a leading block of zero characters. Adding one

[1] "A small piece or slip, esp. of cloth, cut off or out; a shred. A small amount, piece, or portion, a little bit (of something) 1588.", *Shorter Oxford English Dictionary*.

to each character implicitly encodes the length of the string into the fingerprint.

- End-of-line characters or indicators (platform-specific) are replaced by a platform-independent code to enhance portability of the fingerprints.

Observations:

- Any snip that is shorter than the maximum number of characters forms a number of complete lines.
- Any snip that is formed from an exact number of lines will be produced independent of the context.
- Any snip that is of the maximum length will be produced independent of the context.

In summary, the production of snips is independent of the context.

3.5 Culling the Fingerprint File

The fingerprint file contains records with information identifying the snip together with its fingerprint. If a snip were generated for every position in the file, as occurs when $M = m$, this file would be extremely large. The fingerprint file must be culled while it is being produced and only a small selection retained.

Any culling strategy will result in some matches being lost. What is desired is that only short matches will be lost completely and long matches will lose only precision in the boundaries and exact length of the match.; otherwise, the results will not be very useful.

A number of technical issues affect the choice of a culling strategy:

- Since culling is done out of practical necessity, and some information will be lost, the amount of culling must be tunable.
- Culling must be done in a way that depends only on the local context so that the snips corresponding to long matches are culled in the same way.
- The guarantees provided about how much precision in matches is lost must be understandable to the user.
- The strategy must work for all snips.

The culling strategy is easiest to explain in terms of substrings of files and covering snips.

Definition: A snip *covers* a substring occurring in a file if all of the substring is within the snip.

Definition: A snip *represents* a substring

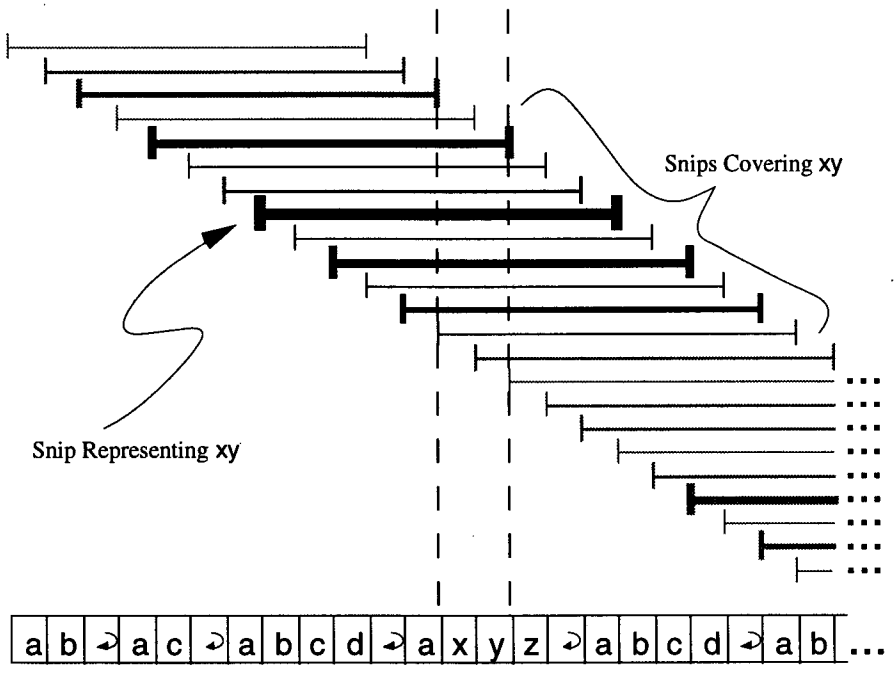

Figure 3. Culling snips

occurring in a file if it has the largest fingerprint (treated as a binary numeral) of the snips that cover the substring.

The culling procedure for a particular cull parameter, c, keeps only snips that represent one or more substrings of the source of length c. Thus, each substring in the file "elects" a snip to represent it and the culling effect is achieved through the probability that the elected snips will represent many substrings of length c.

Setting $c = 1$ causes the most culling to be done and as c is increased more and more records in the fingerprint file are retained. Setting c to M or larger causes no culling to be done.

Figure 3 shows the election process for the characters xy with $c = 2$. Nine snips cover xy; the fourth is elected because it has the largest binary value (indicated in the figure by the heaviness of the line). Figure 4 shows the collection of snips that represent the whole text. Note that the snip that represents xy also represents several other pairs of characters.

This culling process has the following useful property:

Any match longer than $2 \cdot M - c$ occurring in the source will contain at least one matching fingerprint.

This can be seen by observing that the snip representing the middle c characters can, in the extreme, end with these characters or begin with these characters. Since the maximum length of a snip is M, the total snip must fall within the area of match.

Note that culling can be done incrementally as the fingerprint file is being produced.

3.6 Data Reduction

After the culled fingerprint file is further reduced by removing records whose fingerprint occurs only once, we are still left with information that is difficult to process or understand.

The most obvious difficulty is that long matches are represented in the file be a large number of tiny matches, each of which corresponds to a snip. This makes the file large and difficult to understand or process by another program.

To alleviate this problem, the strategy that has been implemented assumes that the boundaries on the snips can be adjusted as long as the overall information about matches is not changed.

For example, imagine two matching snips with content x that are each followed immediately (overlapping or touching) by matching snips with content y. Assume also that no other snips with content x or y occur elsewhere in the source. Then combining each pair of overlapping snips x,y results in four snips being replaced by two and no loss in information about where matches occur.

If, on the other hand, a third snip occurs having content x and it is not followed by a snip with content y, then the above combining of snips cannot be done without loss of information of the partial match of the three x's. However, if only these three snips have x and only these two snips have content y, then the situation is described more naturally by keeping the x snips but shortening the y snips so that there is no overlap.

The snip combining algorithm, not described here, is a systematic application of this principle. It has been very effective in reducing the bulk of

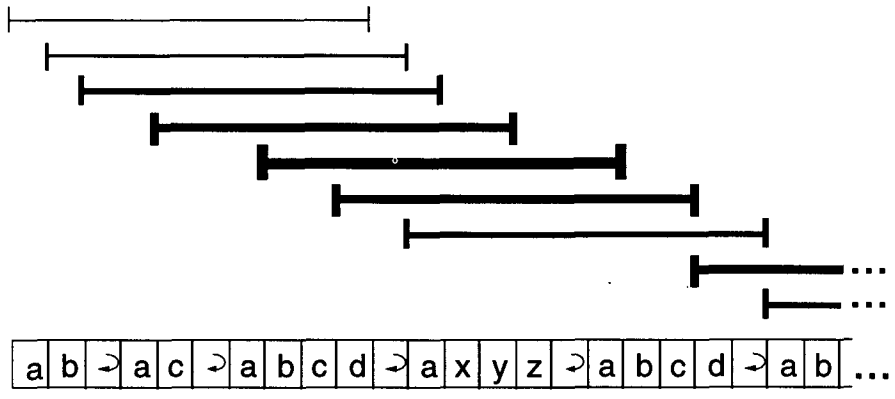

Figure 4. Snips selected by culling

the data and improving the readability of the results.

3.7 Group Analysis

If we form groups of snips that are related to one another directly or indirectly, we will be able to understand complex situations involving overlapping snips and matches.

Definition: Two snips *overlap* if they come from the same file and the intervals defined by their offsets overlap.

Definition: Two snips are *connected* if they overlap or their fingerprints match.

Consider the undirected graph whose nodes are snips and whose arcs join connected snips. The connected components of this graph form a useful partition of snips.

An algorithm that can efficiently identify connected components, subsequently referred to as groups, has been implemented and an ASCII text group listing file containing the grouped fingerprint data produced. It has a line for each snip (remaining after culling) and contains information necessary for identifying the snip in the context of its connected sub-graph: complete file name, beginning offset, ending offset, and fingerprint.

3.8 Prototype Viewer

To provide a better means for understanding the content of the group listing file, a viewing tool based on the Motif selection box and text widget was constructed. This tool displays a range of lines from the file as selection entries and, when a selection is made, parses the line and extracts information about where the snip is located. A text window containing the whole of the file is then opened with the snip highlighted. The window is positioned at the beginning of the snip. The window can be scrolled forward or backward to study the context in which the snip resides.

4 Case Study

4.1 GNU C Compiler v. 2.3.3

To properly show what can be done with this technology, a large source tree is required. The GNU C compiler is publicly available and when unpacked should appear the same on any platform. Thus any of the comments made here can be verified reasonably easily.

The file gcc-2.3.3.tar.Z (7,324,127 bytes) was obtained and unpacked. The software was run on the tree before any other processing was done. The source tree contains 642 files and 19 megabytes of text.

The parameters were set to:

- lines: l = 50
- maximum size of snip: M = 10000
- minimum size of snip: m = 50
- culling parameter: c = 10000

These parameters mean that all sequences of exactly 50 lines will be fingerprinted. The software required about 10 minutes to process the 19 megabytes on an IBM RS/6000 530[2] workstation. After a short period of scrutiny of the output, several observations were made.

A number of large matches appear to be the result of cut-and-paste operations, where a new file was copied from an existing file and tailored. Several of these occurred in the config directory:

- between 63 and 95 lines shared by 3b1.h, crds.h, fx80.h, hp320.h, mot3300.h, news.h, and tower-as.h
- 60 lines: mot3300.h and tower-as.h
- several blocks involving fx80 and m68k files: (.h, .c, and .md)
- first 92 lines: aix386.h and aix386ng.h
- 54 lines: elxsi.h and vax.h
- 71 lines: i386mach.h and i386rose.h
- 62 lines: merlin.h and tek6000.h
- first 50 lines: pa-hpux.h and pa-hpux7.h
- chunks of 137 lines and 111 lines: gmon-sol2.h and ../gmon.h
- 51 lines: i860.h and spur.h

Several matches occurred in the main source directory:

[2] IBM and RS/6000 are trademarks of International Business Machines Corporation.

- INSTALL matches gcc.info-4 with chunks of size 343, 59, 83, 108, and 83 lines and gcc.info-5 with 200 lines
- 121 lines: c-convert.c and cp-cvt.c
- 54 lines: c-decl.c and cp-decl.c
- 98 lines: c-lex.c and cp-lex.c
- several chunks of lines: c-parse.in, c-parse.y, and objc-parse.y
- 51 lines: dbxout.c and protoize.c
- 113 lines: fixinc.sco and fixinc.svr4
- 55 lines: math-3300.h and math-68881.h
- 53 lines: c-typeck.c and cp-type2.c
- chunks of 70, 63, and 65 lines: c-typeck.c and cp-typeck.c

Several large matches occurred within the same file:

- 52 lines: m68ksgs.h
- m68k.md: 50 lines
- we32k.md: 62 lines

A number of files were identical:

- mips-g5.h and mips-gn5.h
- t-decstatn and t-mips
- t-i386isc and t-i386sco
- x-i860v3, x-pa-hpux, and x-w32k
- c++ and g++ (Hard Linked)
- objc/Object.h and objc/object.h (Hard Linked)

A lot of matches resulted from the use of the Bison parser generator:

- c-parse.c, objc-parse.c, cexp.c, cp-parse.c: parser driver
- c-parse.c and c-parse.h; cp-parse.c and cp-parse.h: #defines for tokens
- cexp.y and cexp.x; cp-parse.y and cp-parse.x: input copied to output

In addition, the software found this example of redundancy:

- Sequence of 64 identical lines in mips.c used to initialize an array "mips_char_to_class"

The software has managed to identify a large number of interesting features of the source worthy of followup. The source clearly contains redundancy worth studying.

4.2 SQL/DS

The software was also applied to parts of the SQL/DS[3] source being used as the reference legacy source for the IBM Centre for Advanced Studies Program Understanding Project [4, 5]:

- When applied to the expanded (preprocessed) source, the most obvious information in the output is the redundancy caused by the expansion of inclusions.
- When applied to the expanded source together with the original source, the most obvious information in the output is how the expanded source is assembled through the preprocessing phase.
- When applied to a small sample of the original code (60 files, 51,655 lines, 2,983,573 characters) looking for matches of 20 lines or more, a number of matches were found that on inspection, appear to be cut-and-paste operations totaling about 727 copied lines in 13 files.
- When applied to about 300 megabytes of source, the software successfully ran in under two hours and revealed a number of interesting matches.

A more thorough analysis of the SQL/DS source is waiting better back-end analysis and visualization tools.

5 Conclusions and Future Directions

This preliminary research has established that information in source can be obtained by looking for repeated substrings; but, the technology needs a number of refinements before it can be used effectively by others:

- The information produced must be analysed more thoroughly before it is presented to the maintainer. Good visualization techniques in particular will make this information more accessible.
- The present system is based on exact matches. Similarity matches, using the same basic approach, can generalize this in a number of ways. For example, Baker's technique [1] for replacing identifiers by positional parameters is possible to implement without a major change.
- Some of the ideas for tools discussed in an above section have immediate utility and should be prototyped.

Work is continuing in each of these areas.

[3] SQL/DS is a trademark of International Business Machines Corporation.

References

[1] Brenda S. Baker, "A Program for Identifying Duplicated Code", Proceedings of Computing Science and Statistics: 24th Symposium on the Interface, (1992).

[2] Ted J. Biggerstaff, "Design Recovery for Maintenance and Reuse", *Computer* **22**(7), pp. 36–49, (July 1989).

[3] Ted J. Biggerstaff and Alan J. Perlis, eds. *Software Reusability Vol. I & II*, ACM Press, New York, USA (1989).

[4] Erich Buss and John Henshaw, "A Software Reverse Engineering Experience", *Proceedings of the 1991 CAS Conference,* pp. 55–73 (October 28–30, 1991).

[5] Erich Buss and John Henshaw, "Experiences in Program Understanding", *Proceedings of the 1992 CAS Conference,* pp. 157–189 (November 9–12, 1992).

[6] Elliot J. Chikofsky and James H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software* **7**(1), pp. 13–17 (January 1990).

[7] Richard M. Karp, "Combinatorics, Complexity, and Randomness", *Communications of the ACM* **29**(2), pp. 98–109, (February 1986).

[8] Richard M. Karp and Michael O. Rabin, "Efficient randomized pattern-matching algorithms", *IBM J. Res. Develop.* **31**(2), pp. 249–260, (March 1987).

[9] Kostas Kontogiannis, "Toward Program Representation and Program Understanding Using Process Algebras", *Proceedings of the 1992 CAS Conference,* pp. 299–317 (November 9–12, 1992).

[10] Santanu Paul, "SCRUPLE: A Reengineer's Tool for Source Code Search", *Proceedings of the 1992 CAS Conference,* pp. 329–345 (November 9–12, 1992).

[11] Robert Sedgewick, "Rabin-Karp Algorithm", pp. 289–291 in *Algorithms, 2nd ed.*, Addison-Wesley, New York, USA (1988).

[12] Wuu Yang, "Identifying Syntactic Differences Between Two Programs", *Software—Practice and Experience* **21**(7), pp. 739–755, (July 1991).

Appendix A

A.1 Length of Fingerprint String

How many bits should a fingerprint contain to guarantee a low false drop probability? If we will assume that the bits of a k–bit fingerprint are independent Bernoulli variates having value 1 with probability 1/2, the probability of a false drop occurring among n objects is 1 minus the probability of randomly choosing n different values out of 2^k:

$$1-\frac{\left(2^k\right)_{(n)}}{\left(2^k\right)^n}$$

$$=1-\left(\frac{2^k\cdot\left(2^k-1\right)\cdot\cdots\cdot\left(2^k-n+1\right)}{2^{nk}}\right)$$

$$=\frac{\sum_{i=1}^{n-1} i}{2^k}+O(\frac{n^4}{2^{2k}})=\frac{n(n-1)}{2^{(k+1)}}+O(\frac{n^4}{2^{2k}})$$

Suppose we choose $n=2^{32}=4,294,967,296$ and require the target probability to be smaller than $q=2^{-32}$. This value of n corresponds to the assumptions that objects can be identified by 32-bit integers. The value of q is small compared to the probabilities of other types of failure that are usually considered remote. Solving the above equation for k yields

$$\log_2\left(\frac{n(n-1)}{q}\right)$$

$$=\log_2\left(2^{32}\cdot 2^{32}\cdot\left(2^{32}-1\right)\right)-1$$

$$\approx 96$$

The value chosen was $k=128$. This results in an extra safety margin, double word alignment for arrays of fingerprint values, and agreement with the engineering decision in the length of "message digest" fingerprints produced by MD2 [A1], MD4 [A2, A3], and MD5 [A4].

Note that the approximation made above is valid (and an upper bound) for the ranges of n and k that we are interested in. For $k=128$ and $n=2^{32}$, the first neglected term is

$$-\frac{\sum i\cdot j}{2^{2k}}$$

$$=-\frac{n(n-1)(n-2)(3n-1)}{24\cdot 2^{2k}}$$

$$\approx 2^{-131}$$

A.2 Choice of p and r for the Karp-Rabin Algorithm

The desired properties for p are

- p is a prime so that inverse elements mod p can be used to improve the performance of ancillary functions. The actual fingerprint calculation does not require inverses.
- $(p-1)$ has a known factorization to allow the calculation of the order of elements (the order of r is the minimum value x such that $r^x\equiv 1 \mod p$) and the identification of primitive roots (elements of order $p-1$). r should have large order mod p so that all the characters in each substring of length n have different multipliers mod p. If the order of r is x, then interchanging characters c_j and c_{j+x} leaves the fingerprint unchanged.
- $p\le 2^{128}$ so that the fingerprint fits in 128 bits.
- p is as close to 2^{128} as possible to ensure that the 128 bits are fully used.

The desired properties for r are

- r has large order so that each character c_j has a different multiplier and thus is represented differently in the fingerprint.
- $r\ge 256$ to avoid easy linear dependencies between adjacent characters.
- $r<\left(2^{16}-1\right)/15$ to facilitate calculation of the fingerprint.
- r is a power of 2 to allow shifting to be used when multiplying by r.
- There should be no simple linear combinations mod p that, if preserved, result in the fingerprint being unchanged.

Primality testing can be done efficiently and factoring is expensive. As a result, the approach taken was to look for the largest p less than 2^{128} such that $(p-1)$ can be easily factored using the Maple symbolic computation system:

$$p = 2^{128} - 275$$
$$= 340,282,366,920,938,463,$$
$$463,374,607,431,768,211,181$$

$$p - 1 = 2^{128} - 276$$
$$= 2 \cdot 2 \cdot 5 \cdot 1051 \cdot$$
$$16,188,504,610,891,458,$$
$$775,612,493,217,496,109$$

The 35-digit factor and p both pass Maple's primality test as well as the Miller-Rabin test provided as part of the Lenstra big number arithmetic package.

The value 2 can be shown to be a primitive root of p. As a result, or by direct verification the following can be shown:

r	Order
256	$(p-1)/4$
512	$(p-1)$
1024	$(p-1)/10$
2048	$(p-1)$
4096	$(p-1)/4$

The value of 512 was chosen to allow an extra bit to be available for encoding extra information.

References

[A1] B. Kaliski, "The MD2 Message—Digest Algorithm", *Network Working Group Request for Comments 1319,* (April 1992).

[A2] Ronald L. Rivest, "The MD4 Message Digest Algorithm", *Advances in Cryptology—Proceeding of Crypto '90,* pp. 303–311 (August 11–15, 1990), Springer-Verlag, Berlin, Germany.

[A3] Ronald L. Rivest, "The MD4 Message—Digest Algorithm", *Network Working Group Request for Comments 1320,* (April 1992).

[A4] Ronald L. Rivest, "The MD4 Message—Digest Algorithm", *Network Working Group Request for Comments 1321,* (April 1992).

About the author

Howard Johnson has been a Senior Research Officer with the Software Engineering Laboratory of the National Research Council since May 1992. His current research interest is Software Re-engineering and Design Recovery.

He received his BMath and MMath in Statistics from the University of Waterloo in 1973 and 1974 respectively. After working as a Survey Methodologist at Statistics Canada for four years, he returned to the University of Waterloo and in 1983 completed a PhD in Computer Science on applications of finite state transducers. Between 1983 and 1988 he was an assistant professor in the Department of Computer Science at the University of Waterloo. During this time he developed a prototype system (INR) that facilitated the definition of large finite state machines used in the computerization of the Oxford English Dictionary. Between 1988 and 1992 he was a manager of a software development team at Statistics Canada working on the generalized system for data collection and capture (DC2), a corporately funded project to reduce costs through software component reuse.

He can be reached at the Institute for Information Technology, National Research Council Canada, Montreal Road, Building M-50, Ottawa, Ontario K1A 0R6. His Internet address is johnson@iit.nrc.ca.