Automated Assessment of Program and System Quality

Sukesh Patel, Rich Baxter, William Chu, Brian Sayrs, and Steve Sherman

Software Technology Center, Lockheed Palo Alto Research Labs
B/254E, O/96-10, 3251 Hanover Street, Palo Alto, CA. 94040

Abstract

Software quality issues play a dominant role in the development of large scale software systems. High quality software enhances reuse potential and facilitates software maintenance activities. However, at present, software quality assessment is a labor intensive, error prone, time consuming process. Automated quality assessment of software systems is a cost effective alternative to ensure compliance with system quality objectives. In this paper we discuss how reverse engineering technology can be leveraged to solve the needs of software quality assurance teams.

1. INTRODUCTION

The potential benefits of Software Reuse [2,9,24] has been extensively reported, both in commercial and defense publications and in conferences world wide. Libraries of mathematical functions (e.g., the IMSL library collection of mathematical routines [11]) and Booch components [4] are instances of successful (albeit limited) reuse programs. Typical reusable components are derived from two primary sources:

1. Reusable components are harvested [1] from existing software assets.

2. Reusable components are explicitly designed for reuse as a result of a domain analysis process [14,22].

Although it is widely recognized that quality considerations play a dominant role in reusable software, cost considerations and time limitations restrict the exhaustiveness of the certification process. At present we are actively involved with conducting research on technology that will provide capabilities for conducting cost effective reusability certification procedures. A more precise characterization of reusability certification is as follows:

Reusability Certification Procedures/Metrics: A set of documentation, coding, quality and reusability checks that instill confidence in reusable software.

It is thus essential that reusable components be certified for quality and reusability characteristics. Fig. 1 displays our view of a reuse repository population process. Reusable components are submitted to the library for inclusion in a program/division/corporate related reuse library. The librarian evaluates the components and filters out components that fail to meet quality and reusability standards. Only components that pass the certification process are classified and cataloged in the reuse library. In general, reusable components are quality components and although our research has been motivated by the process model show in Fig. 1, its applicability certainly extends to the more general quality and cost considerations of current software engineering practices. It is for this reason that in this paper we shall target the more general area of software quality assurance for large Ada (Ada is a registered trademark of the U.S. Government, Ada Joint Program Office) software systems; it is assumed that quality considerations are necessary (but not necessarily sufficient) conditions for successful reusability certification processes.

This paper is organized as follows. In section 2 we present an overview of our reverse engineering technology that permits the development of customized certification algorithms. In Section 3, we provide samples of typical code quality checks and illustrate how reverse engineering technology has been utilized to automate such quality checks. Section 4 makes precise our notion of system structure quality and provides an overview of our measure of module cohesion. Finally, we summarize our experiences in Section 5 and briefly discuss the application of reverse engineering technology to automate several aspects of current software engineering practices.

2. ENABLING TECHNOLOGY

Recently, the software industry has seen a proliferation of Reverse Engineering and Ada Code analysis tools. Exam-
Fig 1: Component Quality and Reusability Assessment

With foobar;
With Text_io;
package FOO is
begin
X: integer:=0;
y: real:=1.0;
.
.
end foo;

Fig 2: Semantic analysis by Refine produces data that may be easily Accessed and integrated into applications.
An important aspect of reverse engineering tools deals with the provision and nature of the access to Ada semantic information. Anna (and also the Rational Design environment) provide access to the Ada DIANA tree, whereas systems like Adagen and Adamat provide little or limited access to the underlying Ada semantic information.

We have chosen reasoning Systems Refine tool as our reverse engineering tool because:

1. It provides access to semantic information along the lines of DIANA tree but in conjunction with an object oriented data base for easy access,
2. A powerful language (Refine) for merging query and algorithm processing, and
3. Multi-language support: semantic information in a structure (i.e., an attributed syntax tree) similar to DIANA is made available for three additional languages (Fortran, Cobol, and C).

Fig. 2 provides a overview of the Ada related services offered by the Refine tool. An Ada library is input to the Refine system. The system parses the compilation units in the library, performs semantic analysis on the parsed units, and after semantic analysis places the output of analysis into an object oriented data base for easy access. Thus as shown in Fig. 2, a Ada source unit is parsed to recover the structure as specified by the grammar. Subsequently, Ada semantic analysis is performed to identify the various components and their relationships. For example, package specification Foo with two other Ada units named Foobar and Text_IO. Moreover, once Foobar and Text_IO are parsed, it may be easily recognized that Foobar is an instance of an Ada subprogram, and that Text_IO is an instance of an Ada package.

Given a DIANA tree as input, one may traverse the DIANA to recover Ada specific semantic information. However, the Refine tool places such information into a object oriented data base and hence provides easy access to the underlying semantic information. Thus one need not traverse a DIANA tree to extract the information to answer the following question: What units are required as resources by package specification Foo? In Refine, one merely queries the closure of the "with relation" for package specification Foo in order to determine the answer. Space limitations preclude in-depth discussions of Refine's complete capabilities and we shall illustrate additional features as and when the need arises.

3. TYPICAL CODE QUALITY CHECKS

Typical organizations mandate software development to be performed in conformance with the organizations' "Software Engineering Standards and Practices Handbook." A significant aspect of standard software engineering practice (see for example Boehm [3]), is a review and audit process for ensuring compliance with coding standards. In Table 1 we list a sample of some coding checks that must be performed on deliverable Ada software as part of the quality assurance process.

Let us systematically analyze these rules to arrive at some approximate indication of the time requirements for enforcing such rules. In rule 1, the auditor is required to scan the source file (effectively by a editor such as EMACS) looking for instances of "case statement". Once a case statement is located, the reviewer checks for the presence of a "when others" clause and if found, determines whether the action part consists of a null action. If a null action is found, the offending statement must be marked. Clearly, rule 1 is relatively easy to enforce and does not require extraordinary resources (one may assume about 2 seconds to locate and check a single case statement). Although, easy to enforce, tracking violations of the rule 1 is a time consuming and monotonous activity for large systems.

Rule 2 also does not require extraordinary cognitive abilities. Here all compilation units "withed" by the unit being audited must be collected. Next a withed unit is selected from the list, and the auditor must then collect all visible declarations (e.g., visible functions, procedures, data types, etc. for a package or the specification for a subprogram) from this unit (this may require the auditor to search the whole library to find the required unit). The auditor must then check the specification and the body of the unit being checked for references to visible declarations. If a reference is found, the auditor knows that the current withed unit has at least one reference to its visible objects, and hence is needed within the unit under audit. If no references are found, we mark the with statement as unnecessary. These same steps must be repeated for all with packages. Here we must admit that rule 2, although not difficult to enforce, requires significant time to enforce.

Rule 3 is relatively difficult to enforce for non-trivial programs. Here the required number of paths to be checked over

---

1. Adagen is a registered trademark of Mark V Systems Limited, Encino, CA.
2. AdaMAT is a registered trademark of Dynamics Research Corporation, Andover, MA.
3. Logiscope is a registered trademark of Verilog SA, Toulouse, France.
4. Rational Environment is a trademark of Rational, Mountain View, CA.
5. Refine is a trademark of Reasoning Systems, Palo Alto, CA.
6. This sample list is by no means compete, but only serves to indicate the current availability of tools.
7. The Descriptive Intermediate Attributed Notation for Ada (DIANA) is a complex abstract data type (ADT) where objects in the ADT are a representation of an intermediate form of the corresponding Ada program [10].
8. For accuracy it must be mentioned that he Rational environment provides a front end to the underlying DIANA that facilitates access to Ada semantic information.
9. Such rules form a part of the quality assurance program enforced by typical programs at a major DoD contractor.
(1) **Null “when others” clauses in case statements:** While this is not necessarily a logic or coding error, it can conceal an error in logic. A better method of duplicating the desired processing in a safer manner has been identified, as a result, an audit of these instances would be beneficial.

(2) **Extraneous With Statements:** “Many Times, a Compilation unit may “with” other, units, but does not actually reference some of them. This introduces unnecessary dependencies in the ACS library structures, causing extraneous recompilation of units when these withed packages are changed.”

(3) **Unset Out Variables:** “Unset OUT mode variables in procedures. When an procedure is written with parameters with mode “out”, the programmer should ensure that all logic paths which result in a normal exit load valid values to “out” parameters.”

### Table 1: A Sample of Coding Standards

<table>
<thead>
<tr>
<th>Code Audit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule 1:</td>
<td>Null “when others” clauses in case statements</td>
</tr>
<tr>
<td>Rule 2:</td>
<td>Extraneous With Statements</td>
</tr>
<tr>
<td>Rule 3:</td>
<td>Unset Out Variables</td>
</tr>
</tbody>
</table>

Our cognitive capacities and prevents accurate assessment. For example, if a program has 5 nested “if” statements, a total of 32 distinct paths must be surveyed to identify paths where the out variable has not been set. Moreover, such a computation must be repeatedly performed for every executable program body within the unit being audited (e.g., for a package ever subprogram body must be checked). It must be admitted that enforcing rule 3 is both a time consuming, and an error prone process.

How much time is required to flag every violation of rule 1, 2, and 3 on average for a typical Ada compilation? Although we have no statistical data to back our claims, we do suggest that 1 hour per unit is not a unreasonable estimate (“guessedimate” is a more accurate term) for units the size of Booch components [4]. Even if we cut the time estimates by 50% we still require considerable resources for enforcing code quality standards on large Ada systems. Large here means greater that 100,000 lines of undocumented Ada source.

We have encoded these rules as Refine applications (in addition to several other checks) and have successfully tried these automated checks on about 12,000 lines of production quality Ada source. Fig. 3 displays some sample outputs produced by our analysis tool. We have tested these rules and verified them manually. Rules that require all logical paths to be traversed for a given subprogram body are the most time consuming and require about 10 minutes of execution time for 12,000 lines of commented Ada code.

Note that the code auditor produces output in a form that pin-points the source of the error. For example, the line number and the file name are output in the case of rule 1, whereas the package specification/body name and the file name is output for rule 2. This is essential in applications where (a) the code audit system must itself be validated, and (b) identified errors must be manually verified and corrected. As a side benefit, automated code audits permit the collection of metrics regarding violations without undue difficulty thereby permitting evaluations of design methods, tools, and processes.

### 4. SYSTEM LEVEL ATTRIBUTES

Recognize that the checks that are enforced as part of current quality assessment practices emphasize program quality but do not adequately address system structure quality. Coherence and Coupling are two primary attributes of system structure [7,8,20]. The Software Technology for Adaptable Reliable Systems (STARS) Reusability Guidelines [23] state that for maximum reusability of software components; cohesion should be high between the subprograms comprising the component, and coupling should be low between the component and other components/subprograms in the system being reused.
Cohesion is a quality attribute of a module that indicates the singleness of purpose of a module. Technically, cohesion may be regarded as a measure of the quality of a n-ary relation between a (abstract or logical) component and its constituents. Coupling, on the other hand, is a measure of the quality of a binary relation between components that are, for the purposes of the measurement, treated as independent units. If a module is defined to be a procedure or a function, cohesion can be defined as the quality attribute that describes the degree to which different elementary actions contribute to a unified function. On the other hand, if a module is defined to consist of two or more subprograms, then module cohesion can be defined as the degree to which individual subprograms contribute to the performance of a unified task.

Emerson proposed a graph theoretic based discriminant for computing the cohesion of a single program (a C function, Fortran subroutine etc.) in [8]. Intuitively, given that there are m maximally independent paths and n variables in a program, Emerson's measure associates a high value of cohesion to the program if each path references each variable. As the number of paths that do not reference one or more variables increases the cohesion decreases.

A technique that provides a measure of module cohesion where a module consists of one or more programs is needed. On the one hand, such a technique would extend Emerson's work in that cohesion could also be measured as a quality indicator for composite modules as well as for individual programs. Our organization has developed a measure for composite module cohesion. Very briefly, composite module cohesion is defined as follows.

Given a set of programs P = {p1, p2, ..., pm} the cohesion of module P (i.e., a composite module that includes all members of the set) is given by:

\[
\text{Cohesion}(P) = \frac{\sum_{i \neq j}^{m} \text{Sim}^2(p_i, p_j)}{\sum_{i=1}^{m-1} \sum_{j=i+1}^{m}}
\]

where given two n-dimensional vectors \(X = (x_1, x_2, ..., x_i, ..., x_{n-1}, x_n)\) and \(Y = (y_1, y_2, ..., y_i, ..., y_{n-1}, y_n)\); \(x_i (y_i)\) is the coefficient of vector \(X (Y)\) in the \(i^{th}\) dimension, the similarity of the two position vectors \(X\) and \(Y\) is given by:

\[
\text{Sim}^2 (X, Y) = \frac{\sum_{i=1}^{n} x_i y_i}{\left(\sum_{i=1}^{n} x_i^2\right)^{1/2} \times \left(\sum_{i=1}^{n} y_i^2\right)^{1/2}}
\]

Here \(X\) and \(Y\) represent programs \(p_i\) and \(p_j\) where \(x_i (y_i)\) is the frequency of occurrence of the \(i^{th}\) data type used in program \(p_i (p_j)\). In (1) the numerator computes the summation of the similarity between distinct pairs of programs; the
i > j condition insures that the similarity computation between a pair is only considered once within the summation. The denominator computes the total number of possible pairs for which the similarity metric is computed in the numerator; for a m × m matrix this number is equal to the number of entries lying above the diagonal. Note that composite module cohesion is computed to be the average of the similarity measures over distinct pairs of programs in the module (see [21] for details).

We have analyzed software components developed in Ada and C for determining the appropriateness of the proposed measure. The Ada code selected to test our methods is the Common Ada Missile Packages (CAMP) developed as an experiment in software reuse by the McDonnell Douglas Astronautics Division for the United States Air Force [13]. CAMP software represents a suite of reusable components which may be applied to the construction of missile avionics systems. For our analysis we selected six Ada packages from CAMP. The packages were selected on the basis that their intended functionality were very similar (i.e., Bounded Stack and Unbounded Stack), thus perhaps leading to misplaced or unrelated subprograms within the packages. The six packages selected were Bound_FIFO_Buffer, Unbounded_FIFO_Buffer, Nonblocking_Circular_Buffer, Unbounded_Priority_Queue, Bounded_Stack, and Unbounded_Stack.

The results of the analysis performed on existing Ada code is presented in Table 2, Table 3, and Table 4. In Table 2 the range of computed cohesion values is shown for each package analyzed. For instance, the first row of the Table 2 indicates that package Bounded_FIFO_Buffer contains 8 programs and that the similarity between any two programs in the package was at least 0.89 and at most 1.0; a similarity of 1.0 occurs when the similarity is computed for identical vectors. The rightmost column in each row displays the cohesion of the package as a whole as computed by equation 1. Table 3 displays the similarity measurements for the Bounded_FIFO_Buffer package and Table 4 compares similarity measures for programs contained in different packages. Notice the low values of similarity for programs contained in different packages and the relatively higher values of similarity between programs in the same package. In Table 4 for instance, the intersection of the first row and the second column indicates that the similarity between any program in the Bounded_FIFO_Buffer package and any program in the Unbounded_FIFO_Buffer package is at most 0.25.

We suggest that the proposed measure can be effectively used to determine the cohesive strength of Ada packages. Thus, given a Ada package, the module cohesion is measured and if the measure is below some threshold (say 0.85) the implementor must reorganize the components of the module and rename some type definitions in an attempt to meet the 0.85 threshold. Such a process ensures that modules and their sub-components correspond to cohesive conceptual entities.

5. CONCLUSIONS

In this paper we have demonstrated how current reverse engineering technology can assist in reducing the costs associated with software quality assurance activities. As evident from the discussions of Section 3, we have extensive experience with automating typical code audit processes. Our experience clearly indicates that automating such procedures can result in significant cost and time benefits (this should be clear from the discussion in Section 3 also). However, it is important to mention that the applications of reverse engineering technology are not restricted to automated quality assurance. Indeed some excellent opportunities for automation via reverse engineering technology is being pursued by our organization with considerable success:

1. Language Translation: Automating C to Ada and Fortran to Ada translation.
### Table 3: Similarity Computations for CAMP package Bounded_Fifo_Buffer

<table>
<thead>
<tr>
<th>Programs</th>
<th>Clear_Buffer</th>
<th>Add_Element</th>
<th>Buffer_Len</th>
<th>Peek</th>
<th>Buffer_Status</th>
<th>Retrieve_Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear_Buffer</td>
<td>1.0</td>
<td>0.98</td>
<td>0.95</td>
<td>0.97</td>
<td>0.95</td>
<td>0.94</td>
</tr>
<tr>
<td>Add_Element</td>
<td>0.98</td>
<td>1.0</td>
<td>0.94</td>
<td>0.98</td>
<td>0.94</td>
<td>0.90</td>
</tr>
<tr>
<td>Buffer_Len</td>
<td>0.95</td>
<td>0.94</td>
<td>1.0</td>
<td>0.94</td>
<td>0.89</td>
<td>0.96</td>
</tr>
<tr>
<td>Peek</td>
<td>0.97</td>
<td>0.98</td>
<td>0.94</td>
<td>1.0</td>
<td>0.95</td>
<td>0.89</td>
</tr>
<tr>
<td>Buffer_Status</td>
<td>0.95</td>
<td>0.94</td>
<td>0.89</td>
<td>0.95</td>
<td>1.0</td>
<td>0.87</td>
</tr>
<tr>
<td>Retrieve_Element</td>
<td>0.94</td>
<td>0.90</td>
<td>0.96</td>
<td>0.89</td>
<td>0.87</td>
<td>1.0</td>
</tr>
</tbody>
</table>

### Table 4: Similarity Measures for Programs in Different Packages

<table>
<thead>
<tr>
<th>Packages</th>
<th>Bounded FIFO Buffer</th>
<th>Unbounded FIFO Buffer</th>
<th>Nonblocking Circular Buffer</th>
<th>Unbounded Priority Queue</th>
<th>Bounded Stack</th>
<th>Unbounded Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded FIFO Buffer</td>
<td></td>
<td>&lt; 0.25</td>
<td>&lt; 0.37</td>
<td>&lt; 0.3</td>
<td>&lt; 0.31</td>
<td>&lt; 0.20</td>
</tr>
<tr>
<td>Unbounded FIFO Buffer</td>
<td>&lt; 0.25</td>
<td></td>
<td>&lt; 0.1</td>
<td>&lt; 0.39</td>
<td>&lt; 0.1</td>
<td>&lt; 0.25</td>
</tr>
<tr>
<td>Nonblocking Circular Buffer</td>
<td>&lt; 0.37</td>
<td>&lt; 0.1</td>
<td>&lt; 0.1</td>
<td>&lt; 0.39</td>
<td>&lt; 0.1</td>
<td>&lt; 0.1</td>
</tr>
<tr>
<td>Unbounded Priority Queue</td>
<td>&lt; 0.3</td>
<td>&lt; 0.39</td>
<td>&lt; 0.1</td>
<td>&lt; 0.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bounded Stack</td>
<td>&lt; 0.31</td>
<td>&lt; 0.1</td>
<td>&lt; 0.39</td>
<td>&lt; 0.1</td>
<td>&lt; 0.1</td>
<td></td>
</tr>
<tr>
<td>Unbounded Stack</td>
<td>&lt; 0.20</td>
<td>&lt; 0.25</td>
<td>&lt; 0.1</td>
<td>&lt; 0.4</td>
<td>&lt; 0.1</td>
<td></td>
</tr>
</tbody>
</table>

(2) **Design Recovery**: Recovering the structure of a system’s design given just the source code as input.

(3) **Design Documentation**: The generation of design documentation given the source code as input.

(4) **System Restructuring**: Recognizing poor implementation decisions and restructuring the implementation to improve parameters of interest.

We have also significant experience with application classes (1) and (2). In particular, we are developing techniques that could be applied to convert C system into equivalent Ada systems. The process essentially consists of two steps: (i) a design recovery [13] process is first undertaken to extract the hierarchical C design, and (ii) a source to source translation of C to Ada must be performed given the recovered design.

At this point in time we are advocating the use of module cohesion for determining the quality of Ada modules for de-
liverable Ada systems. The experience so collected will help
to enhance and refine our measures for system structure. In
parallel with our efforts of further experimentation with
module cohesion, we are now focusing on establishing a vi-
able metric for module coupling.

REFERENCES

[1] Biggerstaff, T. J., "Design Recovery for Maintenance and
Volume 1: Concepts and Models, and Volume 2: Applications and
Experience, ACM Press, Addison–Wesley Publishing Company,
Tools, and Subsystems, The Benjamin/Cummings Publishing
[5] L. Constantine and E. Yourdon, Structured Design, Prentice-
pling for Abstract Data Types", Proc. 6th Phoenix Conf. on Com-
229–234.
[8] Emerson, T.J., "A Discriminant Metric for Module Cohes-
An Intermediate Language for Ada", In G. Goos and J. Hartmanis
(Eds). Lecture Notes in Computer Science, Vol. 161, Springer–Ver-
lag, Heidelberg, Germany, 1983.
(CAMP), Vol. 1: Overview and Commonality Study Results,
Software from Reusable Components", In P. Freeman (Ed.), Tu-
Crisis", Proc. IEEE Conference on Software Maintenance, San
mation Methodology for Sequential Runtime Checking of an
Anna Subset", Technical Report 86–301, Computer Systems Labora-
tory, Stanford University, June 1986.
mentation of Anna", In Ada in Use: Proceedings of the Ada Inter-
national Conference, Cambridge University Press, Cambridge,
[20] Stevens, W.P., Myers, G.J., and Constantine, L.L., "Struc-
115–139.
[21] Patel, S., Chu, W., and Baxter, R., "A Measure for Compos-
tive Module Cohesion", Submitted, 14-th Conf. Soft. Engr., Mel-
bourne Australia, 1992.
[22] Prieto-Diaz, R., "Domain Analysis for Reusability", In W.
Tracz (Ed.), Tutorial: Software Reuse: Emerging Technology,
[23] Unisys, STARS Q9 Baseline Ada Library: Reusability
Guidelines, Tech. Report GR–76761012 (NP), Unisys Defense