1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

Development and Application of Reverse Engineering Measures in a Re-engineering Tool

S. Zhou, H. Yang and P. Luker
Department of Computer Science
De Montfort University
England

William C. Chu
Department of Information Engineering
Tung Hai University
Taiwan

Abstract

If software metrics are useful in a forward software engineering environment, they are vital in a reverse engineering one. We are endeavouring to develop suitable metrics for software engineers who urgently need them for reverse engineering legacy systems. We propose that the metrics for reverse engineering in our re-engineering tool, the Re-engineering Assistant (RA), are developed under five categories: complexity, abstractness, object orientation, economics and reusability. The main task of reverse engineering is to extract a representation of existing systems at a high level of abstraction. This main characteristic is reflected in our study. Complexity measures are used to indicate how complex it is to reverse engineer a piece of existing code. Abstractness measures indicate at what level of abstraction the existing code is and whether the code is abstract enough to understand. Object orientedness measures indicate how object oriented the code is for those re-engineers who are hoping to transform myriad conventional procedural systems into object-oriented ones via reverse engineering. Economics (cost estimation) measures indicate the cost to reverse engineer the existing code. Reusability measures indicate to what extent the reverse engineered existing code can be reused. Several measures under each of the above categories have been developed or adopted, and "justified" in this study. Examples are also presented.

Keywords: metrics, measure, reverse engineering, re-engineering, wide spectrum language.

1. Introduction

Metrics are fundamental to any engineering discipline, and software engineering is no exception. Currently, there exists a large number of systems which include many potentially reusable components, most of which systems are in need of refinement. For example, conventional procedural languages are rapidly being replaced by object-oriented languages, yet many of the existing systems referred to above have been implemented in traditional procedural languages. It would be desirable to convert these old systems so that they exhibit object-oriented characteristics and are, importantly, more maintainable. The transformation from a conventional program to an object-oriented one needs to contain the functionality of the old one, which can be accomplished through reverse engineering and re-engineering. In our research, we concentrate on developing reverse engineering metrics to measure changes of these characteristics, in reverse engineering in particular, changes through transformations from existing programs to specifications. We then implement these measures in a metriç tool (Metric Facility in the Re-engineering Assistant) and develop this tool simultaneously for practical use.

2. Reverse Engineering and Metrics

In this section, the definition and main characteristics of reverse engineering will be described together with the main concepts of relevant existing software metrics.

2.1 Characteristics of Reverse Engineering

Software reverse engineering is "the process of analysing a subject system to: identify the system's components and their interrelationships, create representations of the system at a higher level of abstraction" [Chikofsky90]. In short, reverse engineering supports the understanding of existing systems through extracting its specifications. The main incentive for reverse engineering can be attributed to six goals: coping with complexity, generating alternate views, recovering lost information, detecting side effects, synthesising higher level abstraction and facilitating reuse.

2.2 Metrics Definition

Measurement is fundamental to the software engineering discipline (which includes re-engineering) as a whole. Since reverse engineering is an initial and essential part of re-engineering, we are going to concentrate on measurement for reverse engineering phase of re-engineering. Software metrics refers to a broad range of tools for measuring computer software. Within the software engineering context, "a measure provides a quantitative indication of the extent, amount, dimensions, capacity, or size of some attribute of a product or process." [Pressman97] "Measurement is the act of determining a measure. Software metrics is a quantitative measure of the degree to which a system, component, or process possesses a given attribute." [IEEE93] In fact, measurement in the process of reverse engineering in principle can be carried out in the same way as it is carried out for other software engineering processes. Compared to metrics for forward engineering, metrics for reverse engineering must help engineers to *understand* an existing system, because the focus of reverse engineering is always existing systems. Moreover, an existing system normally lacks necessary documents and there are few, if any records about the existing system with which we want to reverse engineer. So reverse engineering metrics are not only used to measure products and procedures throughout the reverse engineering process but are also used to gain necessary knowledge of existing systems (resources of reverse engineering) before the main reverse engineering process begins. However, almost all existing software metrics approaches, methods and models have been designed for forward engineering. In the next section, a short overview of developments in software metrics is given.

2.3 Relevant Existing Metrics

The history of software metrics is not a long one. The first *long-term* software metrics research effort is described in Halstead's paper [Halstead72]. In fact, the earliest paper in this field appears to be *"Quantitative Measurement of Program Quality"* by Rubey and Hartwick. Other early studies include Knuth's study in which a sample program was examined in order to capture quantitatively "What programmers really do" [Knuth71], and Sammet's work related to measuring programming languages [Sammet71]. In the early 1970s, following Halstead's foundation for the new science, *software science*, much work was begun which related to software metrics. Much of the early research was focused on measuring software complexity. The most well-known work based on control paths is the often-cited complexity metric of McCabe [McCabe76]. Another major effort was established at NASA's Goddard Space Flight Centre in co-operation with the University of Maryland by Basili and Zelkowitz [Basili77]. Much of Basili's early work focused on examining relationships among various product and process attributes. More work was subsequently done, such as Albrecht's definition of function points [Albrecht79] and McClure's design complexity metric based on the complexity of control variables and modules [McClure78], Kemerer and Chidamber (KC)'s OO metrics suite [Kemerer94]. Zuse's book [Zuse91] gives an overview of all the major complexity measures of the time.

As far as metrics for reverse engineering are concerned, there are still very few measures specially designed for reverse engineering. In his systematic treatments of software metrics [Fenton91] [Fenton96], Fenton gives an explanation of the whole metrics subject, but, again, mainly from a forward engineering perspective. Boehm's COCOMO model is the typical cost estimation model of the software metrics area, which is mainly designed for forward engineering, but it must be remarked that he has proposed estimating the effects of adapting existing software and software maintenance estimation which refer to metrics for reverse engineering [Boehm81]. Some other authors have mentioned metrics in reverse engineering to a greater or lesser extent. As Pfleeger said: "Measurement is essential for understanding, managing and controlling the software development and maintenance process." [Melton96]. It's easy to see that reverse engineering metrics is still virgin territory in the field of software metrics.

2.4 Summary

Although some authors have referred to metrics in reverse engineering to a lesser or greater extent, we still have been unable to find any systematic research for reverse engineering metrics until now. In particular, software metrics such as object-orientation and abstractness measures for reverse engineering are almost non-existent. Therefore, developing reverse engineering metrics will be very meaningful work in the software engineering area.

Another problem is that reverse engineers always have to find and adapt metrics from the forward engineering domain by themselves so as to meet their need for assessing and controlling a reverse engineering project from the beginning. Meanwhile, they still can't find powerful metrics tools that support their measurement actions when reverse engineering. Because re-engineering is more and more widely in demand, there is no doubt that there is an urgent need to develop suitable reverse engineering metrics for both process and product management in re-engineering.

3. Development of Reverse Engineering Metrics

The main objectives of reverse engineering metrics are to measure complexity, abstractness level, object-orientedness, understandability, transportability and economic characteristics (i.e. reverse engineering duration, effort and productivity, etc.) Based on these facts, measures for reverse engineering are classified into five different categories of reverse engineering metrics. Reverse engineering metric measures can be adapted, redefined from existing measures and developed within each of the five categories. Because when reverse engineering a large program or system, engineers always decompose or slice it into several small program or system segments (modules) and then reverse engineer these segments or modules separately, this leads to a feasible and economic way of reverse engineering existing systems. As for applying the metrics themselves, it's nearly impossible to measure the whole system or program without measuring segments or components of it one by one. As a result, the essential characteristics of reverse engineering and reverse engineering metrics decide that reverse engineering metric measures must be simple measures, which can be executed quickly. More complex measures can only reduce the efficiency of reverse engineering and increase the cost of the whole reverse engineering project. Therefore, we try to use simple, yet effective and efficient measures to occupy the five categories.

In the following parts of this section, five categories of metrics and their application will be described separately. Then all metric measures defined in the study will be implemented in a re-engineering tool, Re-engineering Assistant, thereby strengthening it considerably.

3.1 Complexity Measures

Complexity is one of the most pertinent characteristics of computer software. Complexity measures are the fundamental element of software metrics. Many existing complexity measures can meet the need of reverse engineering. In forward engineering, complexity measures are mainly used for measuring how complex a system is. In a reverse engineering process, people mainly want to understand the existing program through transforming the original program to less complex specifications, because the less complex a program is, the easier people can understand it. Before those transformations, complexity measures can help people to know the general complexity level of the object program and predict how hard it will be to reverse engineer this object program. Among transformations, complexity measures can let reverse engineers control transformation procedures with the direction of always reducing complexity of the object program, but the main usage of complexity measures is still to give an overview to managers and engineers. Several complexity measures are adopted for use for reverse engineering.

Metric 1: McCabe Complexity (MCCM)

Definition: The number of linearly independent circuits in a program flowgraph [McCabe76]. It is calculated as the number of predicates plus one.

Viewpoints: The bigger the MCCM number, the more transformation steps will be required to extract specifications. The more predicates the program comprises, the more difficult to transform the object program.

Metric 2: Structural (STRUCT)

Definition: The sum of the weights of every construct in the program. The construct is defined subjectively according to experience gained by engineers and managers.

Viewpoints: In different systems, constructs always have different weight levels. Those constructs with high weight levels will let transformations become more difficult to execute. In practice, high weight level constructs such as loops are more difficult to transform than assignment statements and other low weight level constructs. Calculating the sum of the weights of every construct can generally help engineers know the overall level of difficulty of forthcoming transformations.

Metric 3: Lines of Code (I) (LOC)

Definition: The number of statements in the program.

Viewpoints: Simply, this measure crudely estimates the overall size of the object program by relating to the scale of the forthcoming transformations directly.

Metric 4: Control-Flow and Data-Flow Complexity (CFDF)

Definition: The number of edges in the flowgraph plus the number of times that variables are used (defined and referenced).

CFDF= Number of edges in the flowgraph + Number of variables + Number of times that variables are referenced

This is a modification of the measure defined by Oviedo [Zuse91].

Viewpoints: Through measuring CFDF, reverse engineers can estimate the size of their forthcoming preliminary work, because a great amount of the basic work in reverse engineering procedures is to extract specifications to explain every node and variable. The bigger the value of CFDF, the more information will appear in specifications.

Metric 5: Branch-Loop Complexity (BL)

Definition: The number of non-loop predicates plus the number of loops.

$$BL=\sum \text{non-loop predicates} + \sum \text{loops}$$

This is a modification of the measure defined by Moawad and Hassan [Zuse91].

Viewpoints: This measure is sensitive both to branches and to loops. The bigger the value of BL, the more explanations will be added to specifications for pointing out structured and unstructured flowgraphs.

Metric 6: Function Points (FPs) Interface Complexity (FPIC)

Definition: Summarises the weighted adjusting functions which counts for the external interface files through which data is stored elsewhere by another application. The function complexity scores are simple, average and complex. This is a modification of the functions points measure [Albrecht83] [Banker94].

$$FPIC = \sum \sum_{c=1}^{3} Interface \quad Function \quad \times \quad Complexity \quad Score_{c}$$

Viewpoints: The bigger the value of FPIC, the more work will be done referring to other applications in reverse engineering procedures. It's more difficult to extract specifications for explaining connections with other applications. Sometimes, because of the lack of necessary documentation of other applications, it will be impossible to gain specifications about parts of interface files.

3.2 Abstractness Measures

Abstractness measures are also frequently used as important measures in the five categories, because abstraction is a significant notion in reverse engineering. Abstractness measures are mainly used to measure the abstractness of an object program. Moreover, abstractness measures can measure both product attributes and process attributes following reverse engineering procedures, abstraction procedures especially. For understanding an existing program, the main task in reverse engineering is to extract the specification of source code via abstraction. Throughout the whole abstraction process, engineers always want to know how abstract the program is and judge whether they have omitted at the correct abstraction level depending on the results of abstractness measurements and whether the program is abstract enough to understand. In other words, abstractness measures help engineers execute abstraction actions more effectively. Another reason is that by using abstractness measures, we can easily use other measures, because the more abstract the program is, the easier it is to measure [Yang97].

Metric 7: Abstractness based on McCabe's Cyclomatic Complexity Measure (ABST-MCCM)

Definition: The reciprocal of the number of linearly independent circuits in a program flowgraph, which is calculated based on McCabe's cyclomatic complexity measure [McCabe76].

$$ABST-MCCM = \frac{1}{\sum predicates + 1}$$

Viewpoints: The fewer the branching statements, the more abstract the program is. That is to say, the closer to 1 ABST-MCCM is, the more abstract is the program.

Metric 8: Abstractness based on Lines of Code (ABST-LOC)

Definition: The quotient of the number of statements (LOC) over the number of nodes (NON) in the abstract syntax tree.

$$ABST - LOC = \frac{\sum statements}{\sum nodes}$$

Viewpoints: This reflects the simplicity of the statements in the program and suggests that the fewer the nodes in each statement, the more abstract is the program. Numerically, the closer to 1 ABST-LOC is, the more abstract is the program.

Metric 9: Abstractness based on Control-Flow and Data-Flow Complexity (ABST-CFDF)

Definition: The reciprocal of the number of edges in the flowgraph plus the number of times that variables are used (defined and referenced), which is a modification of the measure defined by Oviedo [Zuse91].

Viewpoints: This suggests that the fewer the segments of the program and the fewer the variables used, the more abstract the program is. Consequently, the most abstract program is a program with a single edge and no variables. Once again, the closer to 1 ABST-CFDF is, the more abstract is the program.

Metric 10: Abstractness based on Number of Classes (ABST-NOC)

Definition: The reciprocal of the sum of the number of classes plus one.

Viewpoints: When all classes have been transformed into specifications, the abstractness of the object-oriented program must be at the highest abstraction level.

3.3 Object Orientation Measures

Following the strong trend towards object orientation, object orientation measures have become an unavoidable subset of software metrics required for reverse engineering. Nowadays, many software managers and engineers want to transform their huge number of conventional procedural systems into object-oriented systems via re-engineering. Object orientation measures for reverse engineering have special meanings to those managers and engineers. Object orientation measures can be used to measure characteristics of source programs, transitional programs, specifications and so on, which can help engineers transform procedural systems effectively and efficiently in that they exhibit object-oriented characteristics through measuring these attributes. Here we adapt Kemerer and Chidamber (KC)'s OO metrics suite for reverse engineering measures [Kemerer94].

Metric 11: Weighted Methods per Class (WMC)

Definition: Consider a Class C_1, with methods $M_1, \ldots M_n$ that are defined in the class. Let $c_1,\ldots c_n$ be the complexity of the methods. Then: WMC= $\sum_{i=1}^{n} C_i$. If all method complexities are considered to be unity, then WMC = n, the number of methods. Here the number of methods is calculated as the summation of McCabe's cyclomatic complexity of all local methods.

Viewpoints: The number of methods and the complexity of methods involved is a predictor of how much time and effort is required to reverse engineer the class. The larger the number of methods in a class, the greater the potential impact on children, since it will be difficult to extract specifications from this class. Classes with large numbers of methods are likely to be more application specific, which limits the ease of reverse engineering.

Metric 12: Depth of Inheritance Tree (DIT)

Definition: Depth of inheritance of the class is the DIT metric for the class. In cases involving multiple inheritance, the DIT will be the maximum length from the node to the root of the tree.

Viewpoints: The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to gain specifications for it. Deeper trees constitute greater design complexity, since more methods and classes are involved which cause difficulty for reverse engineering tasks. The deeper a particular class is in the hierarchy, the more reverse engineering steps will be performed on it.

Metric 13: Number of Children (NOC)

Definition: Calculates the number of immediate sub-classes subordinated to a class in the class hierarchy.

Viewpoints: The greater the number of children, the greater the likelihood of improper abstraction of the parent class. If a class has a large number of children, it may be a case of misuse of sub-classing. Therefore, it's easy to reverse engineer those parent classes. The number of children gives an idea of the potential influence a class has on the design. If a class has a large number of children,it may require more abstraction steps.

Metric 14: Coupling between Object classes (CBO)

Definition: As for a class, it is a count of the number of other classes to which it is coupled. It relates to the notion that two classes are coupled when methods in one class use methods or instance variables defined by another class.

Viewpoints: The more independent a class is, the easier it is to extract its object and specifications and to transform it. The larger the number of couplings, the higher the sensitivity to changes in other parts of the design, and therefore transformation procedures will become more difficult. The higher the inter-object class coupling, the more rigorous specifications will be added.

Metric 15: Average Parameters per Method (APM)

Definition: Calculated as the number of method parameters compared to the total number of methods.

$$APM = \frac{\textit{The Number of Method Parameters in the program}}{\textit{Total Number of Methods}}$$

Viewpoints: Parameters require more effort of clients. Higher numbers of APM will put a heavier burden on reverse engineers.

Metric 16: The weight of Input/Output Classes (IOC)

Definition: Calculated as the number of classes referring to external actions compared to the total number of classes.

$$IOC = \frac{The\ Number\ of\ Class\ with\ IO\ Methods}{NOC}$$

Viewpoints: Because Input/Output classes refer to other applications, more steps of reverse engineering will be used to gain specifications for them.

3.4 Economics (Cost Estimation) Measures

Economics (cost estimation) measures indicate the cost of reverse engineering the existing code. For controlling the reverse engineering process and reducing the cost of obtaining specifications, engineers and managers must be able to estimate relevant quantities. We present reverse engineering estimation• metrics to address two kinds of quantity: reverse engineering value measures and object based measures. The first 4 measures listed below are reverse engineering value measures, which are all based on the number of thousands of transformed and merged source code instructions (TSI). TSI is calculated as "TSI= Source Lines of Code – Lines of Specifications." The other measures, being object-based, are mainly used to assess the effort of transforming a procedural program into an object-oriented one.

Metric 17: Effort Assessment based on Man-Days (EMD)

Definition: This measure is a modification of the basic COCOMO model defined by Boehm [Boehm81]. Nineteen work days per month is used here.

$$Effort : ManDays = \frac{2.4(TSI / 1000)^{1.05}}{19}$$

Viewpoints: Through this measure, engineers can obtain how many man-days cost units they spend on all source lines of code which have been transformed.

Metric 18: Reverse Engineering Duration (RET)

Definition: $RET(Days) = 2.5(EMD)^{0.38}$

This measure is a modification of the COCOMO model defined by Boehm [Boehm81].

Viewpoints: It can be used to estimate the duration of the whole reverse engineering process. Clearly, the smaller the value of RET, the shorter the duration.

Metric 19: Productivity of Reverse Engineering (REP)

Definition: $REP = \frac{TSI}{EMD \times 1000}$

This measure is another modification of the COCOMO model [Boehm81] [Conte86].

Viewpoints: This measure gives engineers an overview the efficiency of the reverse engineering process. Clearly, the smaller the value of REP, the higher efficiency engineers have realised.

Metric 20: The number of objects (NOO)

Definition: The number of objects is extracted from source code in specifications.

Viewpoints: This is a meaningful measure when engineers transform a procedural system into an object-oriented one. Engineers can know how many objects they have extracted form the old system in preparation for transforming the old system to object-oriented code in the next step. It is a central component of the measure that follows.

Metric 21: Effort: Man-Days of Obtaining Each Object (OMD)

Definition: This gives how many man-days cost units are used to transform one object from source code.

$$OMD = \frac{EMD}{NOO}$$

Viewpoints: This measure gives a general view of the reverse engineering process for transforming a procedural program. It tells that how many man-days cost units were used to extract one object from the raw program.

3.5 Reusability Measures

The last category of the five is reusability measures. Managers and engineers always use them before other processes begin. There are more risks in reverse engineering than in forward engineering. As forward engineers begin with detailed specifications, they have very helpful, distinct directions together with necessary explanations. In contrast, reverse engineers have to retrieve those "necessary things". They may have to work hard to obtain specifications of programs with low reusability. Therefore, reusability measures are classified as an isolated category. Also, reverse engineering is the first stage of re-engineering. Therefore, it can't be avoided that the reusability of an object program must be proved before every re-engineering project is begun. Reverse engineering always occurs in a software reuse domain. Reusability measures can help engineers and managers know what makes software "reusable" and how it is reusable.

Reusability includes many attributes, such as transportability, understandability, readability, testability, correctness, and confidence. In reverse engineering, we mainly measure the transportability and understandability of object software.

Metric 22: The Weight of Interfaces based on Lines of code (WOIL)

Definition: This is a measure calculated by lines of interfaces compared to total lines of code.

$$WOIL = \frac{Lines\ of\ Interfaces}{Source\ Lines\ of\ Code}$$

Viewpoints: The lower the value of WOIL, the less relationship the program has with its environment, since the program is more easily transformed and reused.

Metric 23: Human interaction level referring to Lines of code (HILL)

Definition: This calculates human action level by lines of commands.

$$HILL = \frac{Lines\ of\ User's\ Commands}{Source\ Lines\ of\ Code}$$

Viewpoints: The lower the human interaction level is of the program, the higher the portability of the program.

Metric 24:Environment Independence Level (EIL)

Definition: This assesses the weight of parts which must run under the old environment in the object program.

$$EIL = \frac{Lines\ of\ System - Dependent\ Code + Lines\ of\ Hardware - Dependent\ Code}{Source\ Lines\ of\ Code}$$

Viewpoints: The less the program depends on features of its environment, the more flexible it is.

Metric 25: Comments Density for Methods (CDM)

Definition: This gives the weight of methods with comments compared to all methods.

$$CDM = \frac{The\ Number\ of\ Methods\ with\ Comments}{Total\ Number\ of\ Methods}$$

Here the number of methods is calculated by McCabe's cyclomatic measure [McCabe76].

Viewpoints: It is easier to analyse and transform those methods with comments than those without.

Metric 26: The Weight of Reuse on Lines of code (WORL)

Definition: This assesses the weight of reused parts of the object program depending on documentation.

$$WORL = \frac{The\ Number\ of\ Reused\ Lines\ of\ Code}{Source\ Lines\ of\ Code}$$

Viewpoints: The greater the value of WORL, the more the existing program contains "good" components, and the higher the reusability of the old program. This measure is normally used to measure procedural programs.

4. Metrics Application

4.1 The Re-engineering Assistant

RA (Re-engineering Assistant) is a tool designed for covering aspects of reverse engineering, software maintenance, reuse and re-development. More details can be found in [Yang97].

Reverse engineering is first addressed in RA by extracting high-level specifications of existing software from code level. RA will take existing software written in low-level procedural languages, through a process of successive transformations, and turn it into an equivalent high-level abstract specification expressed in terms of a non-procedural abstract specification language (R-WSL), as the schematic of the prototype of RA shows in Figure 1.

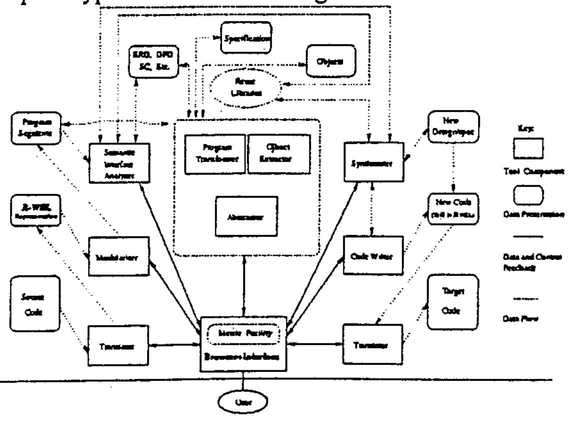

Figure 1: Prototype of Re-engineering Assistant

RA is designed to take the source code (in any language) and translate it into its equivalent R-WSL. A user uses the Browser Interface to control the whole tool, i.e. the Browser calls each tool component and displays the results on the interface. The Modulariser checks the program, chops it into smaller programs, which are of manageable size, and saves them in a database called Program Segments. Then, the user will take a segment of code from the database to work on. The Browser allows the user to look at and alter the code under strict conditions and the user can also select transformations to apply to the code. The program transformer works in an interactive mode, presenting R-WSL on screen in a pretty-printed format. Its main task is to search a catalogue of proven transformations to find applicable transformations for any selected piece of code. Once a transformation has been selected it is automatically applied. The code is then transformed to a form at a higher level of abstraction.

The Semantic Interface Analyser then analyses information in the Program Segments, in the high abstraction level form (e.g., ERD) and in objects, gives them formal attributes and saves them in the Reuse Libraries. When the engineering process starts, the Synthesiser uses the requirements in the NEW Design/Spec to make queries to the Reuse Libraries. The Synthesiser will make best use of the reusable components and invoke the Code Writer to generate the part of code that was required by the new system but was not available in the component libraries. The synthesised code will be saved in New Code but still in R-WSL. Finally, this R-WSL code is translated into the target code in the required language. The Metric Facility of RA is described in more detail in the next section.

4.2 Metric Facility Implementation

During the whole RA process mentioned in the last section, a tool component called Metric Facility, was designed and built to measure the object program, which the user is working on. The Metric Facility can be invoked by the user at any time to measure the object program, or a component in particular.

By using the menu named "Metrics" in the interface of RA, a user can calculate any one, or all of the metrics, applied either to the current program item on which he or she is working or to the whole program. During the process of transforming a program, the metrics at each stage can be recorded and the results can be plotted as and when required.

The objectives of using metrics in RA are to help the user to select transformations (to help develop heuristics on what the final form the measured component should be), to measure the progress made in optimising the component and to measure the resulting quality of the component being transformed.

4.3 Experiment

One specific example has been selected from the many programs with which we have experimented to illustrate the use of the Metric Facility in RA. This example (main step 0) is a short program in R-WSL shown below:

```
comment: "It's a program to calculate factorial and exponentiation";
cal (x, y) { int: x, y, exp, fac;
 !p input (x,y var std_in);
 if (x<>0)
 then call proc exponentiation
else call proc factorial;
fi; }
comment: "To calculate factorial";
proc factorial( In y, Out fac);
{ fac:=1;
if (y==0) or (y==1) then fac:=1 fi;
while (y>1) do fac:=fac*y; y:=y-1 od;
if (y<0) then fac:= -100 fi;
lp print (fac var std_out); }
comment: "To calculate exponentiation";
proc exponentiation( In x, In y, Out exp);
{ <int: n, n:=1>; exp:=1;
if (y<0) then exp:= -100 fi;
if (y==0) then exp:=1 fi;
while (n<=y) do exp:=exp*x; n:=n+1 od;
lp print (exp var std_out); }
```

We extract specifications from the raw program through applying the elementary abstraction rule via a number of transformations. Here we give the main abstract steps using ITL [Moszkowski86, Cau96, Zedan96]. Unimportant steps are omitted, such as those to delete unnecessary comments. Among them, the main program is defined as Φ_{cal} procedure proc factorial is defined as Φ_{fac} and procedure proc exponentiation is defined as $\Phi_{\exp}$. These definitions are all under the ITL rules.

Main step 1

$$\begin{array}{l}
\Phi_{cal} \succ read\ (x, y); (x <> 0 \wedge \Phi_{fac}) \vee (x = 0 \wedge \Phi_{fac}) \\
\Phi_{fac} \succ \{y, fac\} : fac = 1; ((((y = 0) \vee (y = 1)) \wedge fac = 1) \\
\vee ((y <> 0) \wedge (y <> 1))); \\
((y > 1) \wedge (fac = fac * y); y = y - 1; \Phi_{w})); ((y < 0) \wedge \\
fac = -100); pr\ int(fac)) \\
\Phi_{exp} \succ \{x, y, exp\} : n = 1; exp = 1; ((y < 0) \wedge exp = -100) \\
\vee ((y >= 0); \\
(((y = 0) \wedge exp = 1) \vee (y > 0); ((n <= y) \wedge (exp = exp * x; \\
n = n + 1; \Phi_{w}) \vee ((n > y); pr\ int(exp))))
\end{array}$$

Main step 2

$$\begin{array}{l}
\Phi_{cal} \succ (x <> 0 \wedge \Phi_{exp}) \vee (x = 0 \wedge \Phi_{fac}) \\
\Phi_{fac} \succ \{y, fac\} : fac = 1; (((y = 0) \vee (y = 1)) \wedge fac = 1 \\
((y > 1) \wedge (fac = fac * y); y = y - 1;))^{y-1}; (y < 0) \wedge \\
fac = -100) \\
\Phi_{exp} \succ \{x, y, exp\} : n = 1; exp = 1; ((y < 0) \wedge exp = -100); \\
((n <= y) \wedge (exp = exp * x; n = n + 1;)^{n+1} \vee ((n > y);)))
\end{array}$$

Main step 3

$$\begin{array}{l}
\Phi_{cal} \succ (x <> 0 \wedge \Phi_{exp}) \vee (x = 0 \wedge \Phi_{fac}) \\
\Phi_{fac} \succ \{y, fac\} : fac = 1; ((y > 1) \wedge (fac = fac * y); y = \\
y - 1;))^{y-1}; (y < 0) \wedge fac = -100) \\
\Phi_{exp} \succ \{x, y, exp\} : ((y < 0) \wedge exp = -100); \\
((n <= y) \wedge (exp = x^{y}))
\end{array}$$

Main step 4

$$\begin{array}{l}
\Phi_{cal} \succ (x <> 0 \wedge \Phi_{exp}) \vee (x = 0 \wedge \Phi_{fac}) \\
\Phi_{fac} \succ \{y, fac\} : ((y > 0) \wedge fac = y!) \vee (y < 0) \wedge \\
fac = -100) \\
\Phi_{exp} \succ \{x, y, exp\} : exp = x^{y}
\end{array}$$

Main step 5

$$\begin{array}{l}
\Phi_{cal} \succ (x <> 0 \wedge \Phi_{exp}) \vee (x = 0 \wedge \Phi_{fac}) \\
\Phi_{fac} \succ \{y, fac\} : fac = y! \\
\Phi_{exp} \succ \{x, y, exp\} : exp = x^{y}
\end{array}$$

All the above transformations were executed using the RA tool. Following these transformations, all measures were used to control them and validated them simultaneously. Results of the main metrics that are suitable for this sample program are given in table 1. Throughout the table, "COMP" means complexity measures, "ABST" means abstractness measures, "OO" means Object Orientation measures, "ECONOM" means economics measures and "RU" means reusability measures.

Table 1; Results of Measures for Six Main Steps

	Measures	Results of Measures					
	Name	Step 0	Step 1	Step 2	Step 3	Step 4	Step 5
C	MCCM	7	1	1	1	1	1
O	STRUCT	237	186	150	100	62	39
M	LOC	25	12	10	6	3	3
P	CFDF	68	58	42	24	15	7
A	ABST-MCCM	.143	1	1	1	1	1
B	ABST-LOC	.298	.333	.556	.600	.750	1
S	ABST-CFDF	.015	.017	.024	.042	.067	.143
T	ABST-STAT	0	.692	.700	.833	1	1
	WMC	2.3	-	-	-	-	-
O	NOC	3	3	3	3-	3	-3
O	CBO	2	2	2	2	2	2
	RFC	2	-	-	-	-	-
E	CRVL	1	2.09	2.50	4.17	8.33	8.33
C	EMD	0	.0013	.0015	.0020	.0023	.0023
O	RET	0	.200	.211	.236	.249	.249
N	REP	-	1.00	1.00	9.50	9.56	9.56
O	NOO	-	-	-	-	-	2
M	OMD	-	-	-	-	-	.0012
R	WOIL	0.04	-	-	-	-	-
	HILL	0.04	-	-	-	-	-
U	WOM	3.57	-	-	-	-	-

Through analysing several results of the table, it's easy to conclude that those measures give us numeric evidence to assess the reverse engineering process. As processing proceeds, the values of the complexity measures generally become smaller and smaller, which means that the object program has become easier to understand and reverse engineer. For example, after using specifications to represent all possible situations of predicates, the McCabe complexity became the smallest. In fact, transformations to predicates are executed step by step and the McCabe complexity level becomes lower and lower continuously. In the table, we only give the main transformation steps, which makes it seem that the McCabe complexity result becomes 1 suddenly. Other measures behave similarly.

By analysing abstractness measures, we can find that the abstractness level of a program becomes higher and higher, but it's clear that sometimes we don't need to transform the program to gain the highest level of abstraction when it's enough to understand the program. Sometimes, we can't get the highest abstractness measure value as the ideal number, or it is difficult for us to gain the ideal result, such as ABST-VOC equal to 1. We have to use such measures in combination with the results of other abstractness measures so that we can gain the correct assessment of the reverse engineering process. Moreover, those measures for which it is difficult to arrive at the ideal value are still effective in certain reverse engineering stages. This can be easily seen from the table.

As for object orientation measures, we suppose the sample program to be an object-oriented one in which every module is a class, so that we can use OO measures to validate the usefulness of these adapted and developing OO measures in this sample experiment. Economics measures verify that the sample program is a short one and that it's easy to gain its specifications in a short time and with low cost. Another conclusion is that economics measures based on objects are normally suitable for use at the final stages of reverse engineering a procedural system.

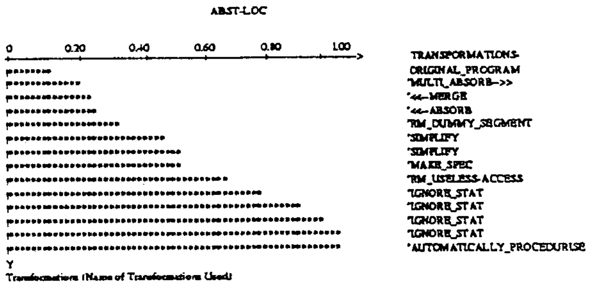

Figure 2: Sample Plot of an Abstractness Measure

Only three reusability measures are used in the experiment, because the program is designed to meet the need of this experiment, more reusability measures are not suitable for measuring the sample program. Through analysing the results of the measures, we can find that the reusability of the sample program is not good. In other words, the reusability measures are valid and useful to measuring the sample program. After deep validation using more, different object programs in Re-engineering Assistant, several measures will be added into the Metric Facility and used to measure practical programs and systems in the reverse engineering process. A sample plot of an experiment is given in Figure 2 for a single metric.

5. Conclusions and Further Work

In this paper, the development of reverse engineering metrics has been discussed. The contribution of this study is to develop five categories of reverse engineering metrics and to adapt and develop reverse engineering metrics based on these five categories. The main objective of reverse engineering metrics is to measure and help acquire specifications from existing systems through abstraction so as to understand them. In particular, we have tried to:

- attempt to classify several reverse engineering metrics into five categories so as to meet different demands for individual reverse engineering stages and the whole process.
- adapt and develop reverse engineering metrics in different categories.
- design an experiment to implement five categories of reverse engineering metrics in Re-engineering Assistant tool so as to validate and refine them.
- develop the Re-engineering Assistant tool when implementing five categories of metrics and adding them to the Metric Facility of the Re-engineering Assistant.
- develop a prototype and process for studying five categories and several measures affiliated to them. This would allow researchers to experiment with real examples and build upon the initial research presented in this paper.
- develop more object orientation measures for reverse engineering, thereby giving new meanings to reverse engineering metrics when extracting objects from conventional procedural programs and transforming them to object-oriented programs.

We will continue to improve our reverse engineering metrics. For example, almost all key words in the current prototype have a weight of 1 for abstractness. As more experiments are carried out, we will be able to weigh the abstractness for each key word in R-WSL more precisely. Meanwhile, we will use further metrics, choose the most practical and powerful metrics from existing metrics and develop new metrics, then add them to the five categories and Metric Facility of RA.

References

[Albrecht83] A. Albrecht and J. Gaffney. Software Function, Source Lines of Code, and Development Effort Prediction - A Software Science Validation. *IEEE Transactions on Software Engineering* , 9(6):639-648, 1983.

[Albrecht79] A. Albrecht. Measuring Application Development Productivity. Technical report, *Proc. of IBM Applic. Dev.* Joint SHARE/GUIDE Symposium, Monterey,CA, 1979.

[Boehm81] B. Boehm. *Software Engineering Economics.* Prentice-Hall,Inc., 1981.

[Banker94] R. Banker, R. Kauffman and C. Wright. Automating Output Size and Reuse Metrics in a Repository Based Computer-aided Software Engineering. *IEEE Transactions on Software Engineering,* 20(3):169-187, 1994.

[Basili77] V. Basili and M. Zelkowitz. The Software Engineering Laboratory Objectives. In 5th Annual Computer Personnel Research Conference, pages 256-269. ACM, 1977.

[Chikofsky90] E. Chikofsky and J. CrossII. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software,* 7(1):13-17, Jan 1990.

[Conte86] H. Conte, S. Dunsmroe and V. Shen. *Software Engineering Metrics and Models.* The Benjamin/Cummings Publishing Company, Inc, 1986.

[Cau96] A. Cau, H. Zedan, N. Coleman, and B. Moszkowski. Using ITL and Tempura for Large Scale Specification and Simulation. In *Proceedings of 4th EUROMICRO Workshop on Parallel and Distributed Processing,* IEEE, pages 493-500, Braga, Portugal, 1996.

[Fenton91] N. Fenton. *Software Metrics--A rigorous approach.* Chapman-Hall, 1991.

[Fenton96] N. Fenton and S. Pfleeger. *Software Metrics--A rigorous approach.* International Thomson Computer Press, London, 2nd edition, 1996.

[Halstead72] M. Halstead. Natural Laws Controlling Algorithm Structure. *ACM SIGPLAN Notices,* 7, 1972.

[Kemerer94] C. Kemerer and S. Chidamber. A Metrics Suite for Object-Oriented Design. *IEEE Trans. Soft. Eng.,* 20((6)):476-493, June 1994.

[Knuth71] D. Knuth. An Empirical Study of Fortran Programs. *Software Practice and Experience,* 1971.

[McCabe76] T. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering,* SE-2(4):308-320, Dec 1976.

[McClure78] R. McClure and W. Edward. A Model for Program Complexity Analysis. In *Proc. 3rd International Conference on Software Engineering,* pages 149-157, 1978.

[Melton96] A. Melton. *Software Measurement,* chapter 5, page 53. Thomson Computer Press, 1996.

[Moszkowski86] B. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, Cambridge UK, 1986.

[Pressman97] R. Pressman. *Software Engineering: A Practitioner's approach.* McGraw-Hill Book Company, 4th edition, 1997.

[Sammet71] J. Sammet. Problems in, and a Pragmatic Approach to, Programming Language Measurement. In *AFIPS Conference Proc.* (Proc. of FJCC), volume 39, 1971.

[IEEE93] IEEE Software Engineering Standards, Std.610.12-1990, 1993.

[Yang97] H. Yang and P. Luker. Measuring Abstractness for Reverse Engineering in a Re-engineering Tool. In *IEEE International Conference on Software Maintenance,* Bari, Italy, October 1997.

[Zedan96] H. Zedan and H. Heping. An Executable Specification Language for Fast Prototyping Parallel Responsive Systems. *Computer Language,* 22:1--13, 01 1996.

[Zuse91] H. Zuse. *Software Complexity-Measures and Methods.* Walter de Gruyter, New York, 1991.