A Framework for Measuring and Evaluating Program Source Code Quality

Hironori Washizaki[1], Rieko Namiki[2], Tomoyuki Fukuoka[2], Yoko Harada[2], and Hiroyuki Watanabe[2]

[1] National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, Japan
washizaki@nii.ac.jp
[2] Ogis-RI Co., Ltd., MS-Shibaura Bldg., 13-23, Shibaura 4, Minato-ku, Tokyo, Japan
{Namiki_Rieko,fukuoka_tomoyuki,Harada_Yoko,Watanabe}@ogis-ri.co.jp

Abstract. The effect of the quality of program source code on the cost of development and maintenance as well as on final system performance has resulted in a demand for technology that can measure and evaluate the quality with high precision. Many metrics have been proposed for measuring quality, but none have been able to provide a comprehensive evaluation, nor have they been used widely. We propose a practical framework which achieves effective measurement and evaluation of source code quality, solves many of the problems of earlier frameworks, and applies to programs in the C programming language. The framework consists of a comprehensive quality metrics suiteC a technique for normalization of measured values, an aggregation tool which allows evaluation in arbitrary module units from the component level up to whole systemsC a visualization tool for the evaluation of resultsC a tool for deriving rating levels, and a set of derived standard rating levels. By applying this framework to a collection of embedded programs experimentally, we verified that the framework can be used effectively to give quantitative evaluations of reliability, maintainability, reusability and portability of source code.

1 Introduction

In today's world, where value is controlled in every corner of society by software systems from the embedded to enterprise level, demand is increasing for a system of technology to measure and evaluate system quality characteristics (e.g. reliability) to use the evaluation results to maintain and improve the system. In this paper we propose a quality evaluation framework based on quantitative quality measures, for software engineers involved in development, maintenance or procurement of software, or others involved in improvement of development processes. We deal with quantitative measures of quality that take measurements of program source code written in the C programming language.

There is great demand for practical technologies which can measure and evaluate quality with high precision and identify quality characteristics that will cause problems or will need improvement, because the quality of the source code has a significant effect on the overall system performance and cost of development and maintenance. In the past, various techniques for measuring quality

J. Münch and P. Abrahamsson (Eds.): PROFES 2007, LNCS 4589, pp. 284–299, 2007.

have been proposed, but they generally have not covered quality characteristics comprehensively and the metrics or measured results have not been widely used[1].

In response, we propose a framework which applies to source code written in the C programming language and implements quality measurements and evaluation effectively. The framework is independent of any person/evaluator properties, and resolves the problems of conventional approaches.

2 Problems with Conventional Quality Measurements

From a quality point of view, measurement methods can be classified into four types based on amount of information. A *Metric* contains the least amount of information, and simply measures a particular property without relating it to quality. A *Quality Metric* measures a property and includes a way to interpret the measurement result in terms of a quality characteristics. *Quality Metrics* is used to refer to multiple such metrics for a single quality characteristics and a *Quality Metrics Suite* treats several quality metrics and systematically summarizes the results of each.

Though many metrics have been proposed, it is generally difficult to select an appropriate one from among them or to interpret the measurement results[2]. Further, they and measured values have not been broadly useful[1]. For quality measures which apply to source code in particular, the main problems are summarized below.

(P_1) **Non-comprehensive suites:** In order to take into account tradeoffs between different quality characteristics (e.g. time-behaviour vs. analysability), it is desirable to be able to measure and evaluate all quality characteristics, which effect the final system's quality in use, at the same time. However, most of the existing metrics which apply to source code do not relate measurement values to quality, or provide a quality metric which measures quality based on only a single characteristic. There are a few suites which handle source code, including REBOOT[3], QMOOD[4], SPC suite[5], the suite from Ortega[6], the EASE project result[7] and the ISO/IEC TR 9126-3 reference implementation[8]; however, they all require additional input besides the source code (e.g. a design model) and/or they lack comprehensive coverage of the measurable and assessable source code characteristics specified by the ISO9126-1[9] (or equivalent) quality model.

(P_2) **Lacking in ability to break-down or overall evaluation:** Generally, source code written in a high-level programming language has a layered structure, with inclusion relationships between multiple logical and physical modules. For example, in the C programming language, generally functions are included in files, files in directories, and directories in the system, giving a four-layer structure. In this case, it is desirable to measure and evaluate the quality of individual module units according to various objectives, such as comparing the entire integrated system quality with another system and evaluating the quality

of individual small modules in order to identify problematic parts. However, no quality metrics suite has been proposed which can measure source code quality for arbitrary units from component up to the overall system.

(P_3) **Difficulty in deriving standard rating levels:** In order to evaluate quality from measurement results, rating levels which determine the allowable range of values for each type of measurement (i.e. thresholds), and assessment criteria which evaluate the results in units of quality characteristic and can combine them into an integrated result is required[10]. Generally, to derive rating levels, a set of samples are used, which are then divided into superior and inferior groups based on some criteria (e.g. a particular usage scenario[5] or qualitative evaluations[11,12]). The distributions of measured values in both groups are compared, and the upper and lower bounds of the range which statistically contains most of the measurements from the superior group are used as the thresholds. However, traditional techniques have required additional information, such as usage scenarios or qualitative evaluations, in addition to the source code and it has not always been easy to derive rating levels.

3 Proposed Framework for Quality Evaluation

We propose a practical framework which effectively measures and evaluates the quality of programs from source code written in the C programming language. The overall scheme, the solutions to the problems described above, and details of each of the elements of the framework are described below.

3.1 Overall Approach and Solutions to Problems

The structure of the framework is shown in Figure 1. It is made up of five elements: the quality metrics suite, an aggregation tool, a visualization tool, a rating levels derivation tool, and actual derived rating levels. Note that the framework does not include actual measurement tools. Rather, we make use of existing tools (e.g. QAC[13] and Logiscope[14]) which both apply to C source code and cover the metrics specified in the suite.

- Quality metrics suite: We extended the ISO9126-1 quality model to create a more comprehensive model, and built a collection of metrics together with associated quality characteristics based on this quality model by defining the interpretation of measurements from a quality standpoint. This resolved the problem P_1 of conventional approaches.
- Aggregation and visualization tools: We resolved the problem P_2 by implementing a technique for normalizing the results of each measurement in the suite to a value from 0 to 100 based on rating levels, and a mechanism for aggregating and summarizing module scores in step-wise gradations. The visualization tool transforms the collection of totaled scores into an evaluation report that is easier to understand intuitively.

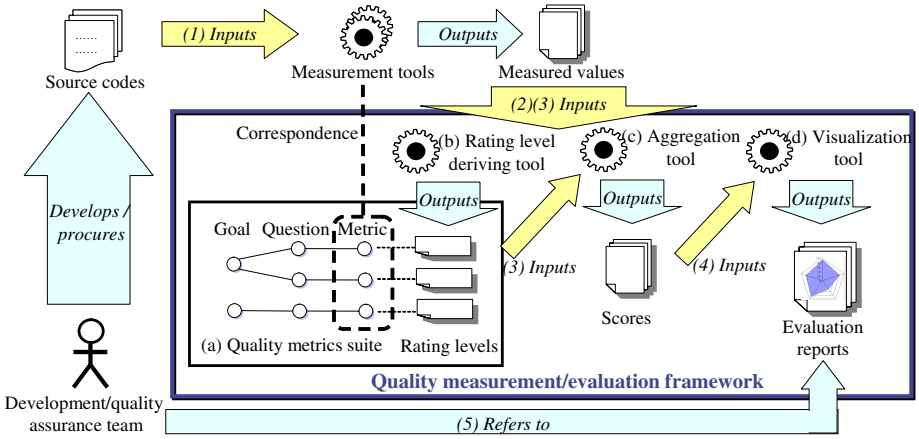

Fig. 1. Structure of the quality measurement/evaluation framework

– Rating level derivation tool and reference values: We resolved the problem P_3 by implementing a mechanism to derive rating levels statistically. The mechanism requires a set of source codes that are acceptable from a quality point of view. Then, by applying our metrics suite and the above three tools to several existing embedded software programs, we derived some actual rating levels and included them in the framework as reference values for software in the same domain.

The process flow for using the framework is shown below. Step (2) is not always required after rating levels have been derived for the target problem domain; however the rating levels should be continuously improved by iterating the process and accumulating measurement results, because the rating levels highly depend on the set of source codes used for the level derivation.

(1) Measurements (measured values) are obtained by applying measurement tools that handle the metrics specified in the quality metrics suite to the source code being measured.
(2) If rating levels are to be derived, this is done by applying the rating level derivation tool to measurements of source code that are acceptable from a quality point of view. If such codes are not available, the framework uses source codes which has been improved from a quality point of view, without any significant changes in functionality.
(3) The measurements are entered into the aggregation tool to get the aggregate result of all of the scores. Internally, the aggregation tool uses the quality metrics suite as well as the derived rating levels.
(4) An evaluation report is created by entering the aggregate results into the visualization tool.
(5) The report is used to identify problematic parts or quality characteristics that need improvement and can be useful in resolving them.

3.2 Details of Structural Elements

The details of the elements and techniques that compose the framework are described below.

(a) Quality metrics suite
Using the ISO9126-1 general quality model as a starting point, we repeatedly interviewed several software professionals to narrow-down to the internal quality characteristics (static, not dynamic, qualities that can be measured and evaluated) of program source code that are not dependent on a particular programming language. The resulting quality model is shown in Figure 2. Note that functionality and usability have been excluded because they are difficult to measure using the source code only.

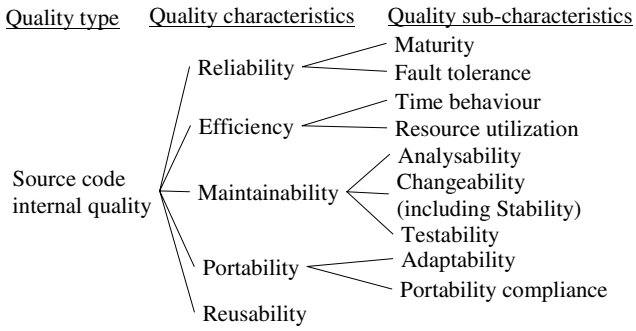

Fig. 2. Quality model

- Reliability: The ability to maintain specified performance levels when used under the specified conditions[9]. We take maturity and fault tolerance as sub-characteristics, while recoverability and reliability compliance (not the "reliability" itself) are excluded because they are difficult to measure and evaluate from the source code only.
- Efficiency:The ability to provide appropriate performance relative to the amount of resources consumed when used under clearly specified conditions[9]. Sub-characteristics are time behaviour and resource behaviour, while efficiency compliance has been excluded.
- Maintainability: The ease to which modifications can be made[9]. Sub-characteristics are analysability, changeability and testability, while maintainability compliance has been excluded. Also, to avoid duplication, the stability characteristic in ISO9126-1 is included under changeability.
- Portability: The ability to be transferred from one environment to another[9]. Sub-characteristics are adaptability and portability compliance, while installability, co-existence and replaceability have been excluded because they are difficult to measure and evaluate from the source code alone.

– Reusability: The extent to which a system or module-unit parts can be reused in a different environment. This is not regulated in ISO9126-1, but considering its importance, particularly with respect to development efficiency within the same problem domain, we have added it as another quality characteristic separate from portability.

Next, we applied the Goal-Question-Metric (GQM) method[15], and assigned a metric to each quality sub-characteristic in the quality model. The GQM method is a goal-oriented method for mapping a goal to a metric by using a question which must be evaluated in order to determine whether the goal has been achieved or not. It is used within the framework to assign the metrics to the quality characteristics being evaluated.

We posed questions so that the evaluation could be made independently of the programming language of the source code being evaluated, and with the goals being to measure and evaluate each of the quality sub-characteristics. Finally, we narrowed down possible (programming-language dependent) metrics to those which could provide answers to the questions by measured values. If a given question was at a relatively high abstraction level, or was more removed from the available programming-language-dependent metrics, we handled it by dividing into sub-questions. In this way, the suite is structured in four layers, with the goals and questions being independent of language, and the sub-questions and metrics being basically language-dependent. This raises the reusability of the framework by clearly separating the fixed part of the framework (i.e. commonality) from the part which may require modification (i.e. variability).

An excerpt from the suite is shown in Figuresuite[1]. The suite is made up of 47 questions, 101 sub-questions and 236 metrics. As metrics, we used those which are supported by existing tools for the C programming language (such as QAC and Logiscope), the degree of conformance to existing coding style guides for the C language (such as MISRA-C[17] and IPA/SEC's guide[18]), and other metrics which seemed necessary for particular questions or sub-questions where currently available measurement tools did not apply. 19% of the metrics could not be measured using currently available tools. One such example is the very specialized measurement, "number of branches due to macros." Further, 29% of the questions (either directly or via sub-questions) could not be assigned a metric at all. In the future, we will reduce or eliminate the proportion of metrics and questions that are not covered by developing new measurement tools. A selection of metrics from the full list is shown in Table 1. The table includes the following details to help the evaluator understand each metric.

– Type of measurement scope: System (ID: `MSyXXX`), Directory (`MMdXXX`), File (`MFlXXX`) or Function (`MFnXXX`).
– Type of rating level[19]: Threshold (quality is interpreted to be best when the measurement value is a particular value, or within a particular range), Minimal (the smaller the better) or Maximal (the larger the better).

[1] The entire suite is published in [16].

– Scale type[20]: Nominal, Ordinal, Interval, or Ratio.
– Programming language dependency type: Not (not dependent on language), Not-OO (non-object-oriented language dependent), OO (dependent on object-oriented language), C (C programming language), C++ (C++), or C&C++ (C or C++).

Table 1. List of metrics used (excerpt)

ID	Metric name	Rating	Scale	Dependency
MSy021	Number of recursive passes	Minimal	Ratio	Not
MMd027	Number of elements located directly below the directory	Threshold	Ratio	Not
MFl003	ELOC	Threshold	Ratio	Not
MFn072	Cyclomatic number	Minimal	Ratio	Not

Characteristic	Sub-characteristic	Goal	Question	Sub-question	Metric
Reliability	Maturity	Purpose： Evaluate Issue： the frequency of faults Object: source code Viewpoint: end-user	Q0100: Is the code not prone to faults?	Q0101 Has memory been initialized properly?	MFl134: Number of un-initialized const objects. MFl107: Number of arrays with fewer initialization values than elements. MFl133: Number of strings which do not maintain null termination. MFl169: Number of enumerations not adequately initialized.
				……	……
			Q0200: Is the scope not too large?	Q0201: Is the number of partition elements appropriate?	MMd027: Number of sub elements MMd008: Number of functions MFl003: Effective number of lines.
			Q0400: Is it possible to estimate the size of resources to be used?	Q0401: Is there not any recursive call?	Msy021: Number of recursive paths.
	Fault tolerance	……	……	……	……
Maintainability	Analysability	Purpose: Evaluate Issue： the easiness of identifying styles, structure, behaviour and parts for maintenance Object: source code Viewpoint: developer	Q3700 Are the functions not too complicated?	Q3701 Is the function-call nesting not deep?	MFn095 Depth of layers in call graph
				Q3702 Is the logic not too complex?	MFn066 Max. nesting depth in control structure. MFn072 Cyclomatic number. MFn069 Estimated no. of static paths.

Fig. 3. Quality metrics suite (excerpt)

For example, Figure 3 gives several language-independent questions (e.g. Q3700) which help to evaluate how easy the source code is to analyze. Q3700 is quite abstract and difficult to measure directly, so it is broken-down into several sub-questions, including Q3701 and Q3702. Finally, metrics are assigned to each sub-question to make it possible to obtain data to answer them. The single metric, MFn095, is assigned to Q3701, and three metrics, MFn066, MFn072 and MFn069, are assigned to Q3702. By making these assignments, source code quality can be evaluated in quality-sub-characteristic units from the measurement

values. For example, it is clear in Table 1, that the measured value for the cyclomatic number of a function[21] can be used to evaluate the analysability of the source code. Also, since the type of rating level for cyclomatic number is "Minimal", the smaller the measurement value is, the better the analysability is for that part of the code.

(b) Technique and tool for deriving rating levels

In order to evaluate the permissible range of values for a particular quality characteristic from the measurements obtained using the suite, a rating level for each metric is required. Within the framework, such a rating level is derived using a collection of existing program source codes (denoted as the "acceptable set") that are acceptable from a quality point of view.

If such codes are not available, the framework uses source codes (denoted as the "after-improvement set") to which some quality improvements have been made while not altering the functionality, as an alternative to the acceptable set. Due to tradeoffs between different quality characteristics, there might be quality characteristics that have got worse compared to the before-improvement set, among all characteristics. However, we think the after-improvement set can be regarded approximately as an acceptable one, because some developers or clients accepted the set instead of the corresponding before-improvement in fact.

Measurements were made on the acceptable set or the after-improvement set, and rating levels of three different types were derived using the upper and lower hinges (i.e. 75th and 25th percentiles) of the statistical distributions of measurements from each metric as described below:

- Minimal: The rating level is below the upper hinge.
- Maximal: The rating level is above the lower hinge.
- Threshold: The rating level lies between the upper and lower hinges.

This derivation technique was implemented using spreadsheet software functions on the Excel worksheet. We applied this technique to three quality-improved, industrial embedded software S_1, S_2, S_3 (automobile or internal printer software) that we were able to obtain. The measurement values for program scale, before and after quality improvements, for the total of six programs are shown in Table 2. Comparing the programs before and after improvements, the improved versions appear to be implemented with a larger number of functions but the number of lines of code in each function is smaller.

As an example of another metric, the distribution of results from metric `MFn072`, "Cyclomatic number," before and after improvement, are shown in

Table 2. Scale totals for samples used

Type	Number of files	Number of functions	ELOC
Before improvement	603	3,269	174,650
After improvement	842	4,873	116,015
Total	1,445	8,142	290,665

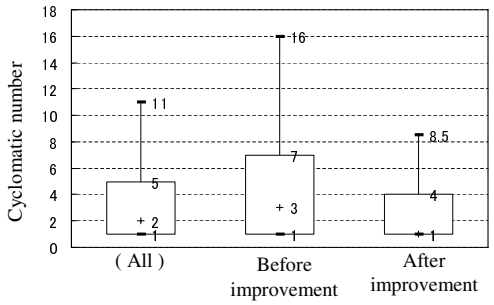

Fig. 4. MFn072 – Distribution of cyclomatic numbers (box-plot diagram)

Figure 4 as a box-plot diagram. In the box plot, the upper and lower edges of the rectangle indicate the upper and lower hinges, the value in the box is the median value, and the lines above and below the box give upper and lower adjacents. Figure 4 shows that compared to the before-improvement set, the after-improvement set tends to yield smaller values. Since the rating level type for the cyclomatic number values is "Minimal" (as shown in Table 1), the rating level allows values below the upper hinge of 4.

(c) Normalization/aggregation tool

To achieve an overall quality evaluation, we aggregate the measurements from the various metrics in the suite into quality-characteristic and module units by normalizing each measurement value, using the rating level, to a value from 0 to 100. More specifically, if a measurement value falls within the rating level as described above, it receives a score of 100. If it falls above the upper outer fence (upper hinge + 3·H-Spread) derived from the improved code set or below the lower outer fence (lower hinge - 3·H-Spread), it receives a score of zero. "H-Spread" (i.e. interquartile range) means the value of "upper hinge - lower hinge". A linear graph between these outer fences for each metric is created, and scores are interpolated linearly from the graph.

As an example, normalized values for measurements of the cyclomatic number are shown in Figure 5. According to Figure 4, the upper hinge for the improved code set is 4, and H-Spread is 3 (= 4-1), so a straight line from a measurement value of 4 to the value $4 + 3 \cdot 3 = 13$ is drawn in the score graph in Figure 5. Then, if the cyclomatic number is 2, the score taken from the graph is 100.

By transforming measurement values to normalized scores using a continuous, linear score graph in this way, small changes are reflected intuitively in the score, and values from different metrics can be compared with each other. It is also conceivable to construct the measurement normalization graphs using other forms such as non-linear curves or discontinuous step functions[22], but for the purpose of understanding overall trends in how small measurement values affect quality characteristics, these other graph types do not improve the results significantly, so linear graphs were selected as most appropriate.

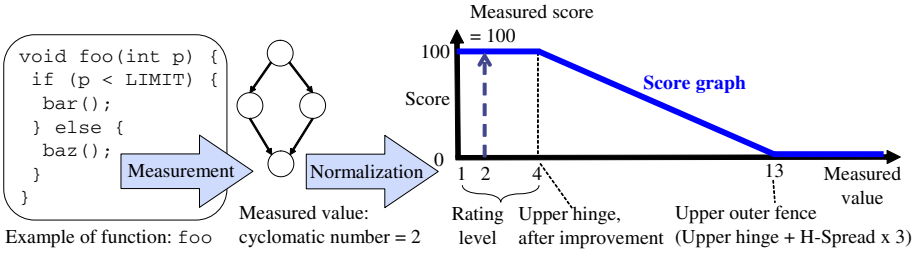

Fig. 5. Calculating the score for cyclomatic number

Next, a weighting is applied to each of the normalized scores, and they are aggregated in quality-characteristic, sub-characteristic and module units. The weighting mechanism allows for the influence of each of the questions or metrics to be adjusted but the standard settings simply use an even distribution (i.e. simply take the average). The aggregation model which forms the basis for aggregating the scores is shown as a UML class diagram in Figure 6. Also, the constraints related to scores in Figure 6 are given below in OCL[23].

```
context CharacteristicResult
 -- Score is a weighted sum of the scores from each of the sub-characteristics
 inv: score = SubCharacteristicResult->iterate( c:SubCharacteristicResult;
       result:Real=0 | result+c.score*c.SubCharacteristic.weight )
 -- The total of all sub-characteristic weights is 1
 inv: SubCharacteristicResult->SubCharacteristic->collect(weight)->sum()=1
context SubCharacteristicResult
 inv: Score=QuestionResult->iterate( q:QuestionResult;
       result:Real=0 | result+q.score*q.Question.weight )
 inv: QuestionResult->Question->collect(weight)->sum() = 1
context QuestionResult
 inv: Score=MeasurementResult->iterate( m:MeasurementResult;
       result:Real=0 | result+m.score*m.Metric.weight )
 inv: MeasurementResult->Metric->collect(weight)->sum()=1
context MeasurementResult inv:
 if underMeasurement.target <> Metric.target
  -- Score is the average of the total of the same metric's measurement
  -- result scores of all of the target program module unit's childs
  score=underMeasurement->child->collect(MeasurementResult)->select(Metric.
  id=self.Metric.id)->collect(score)->sum()/underMeasurement->child->size()
 else
  -- Score is equal to the measurement value normalized by using rating level
 endif
```

An example of score calculation and aggregation based on the aggregation model is shown as a UML object diagram in Figure 7. For simplicity, quality characteristics and sub-characteristics, rating levels, directories and functions have been omitted, and only evaluation of the reliability of the whole system,

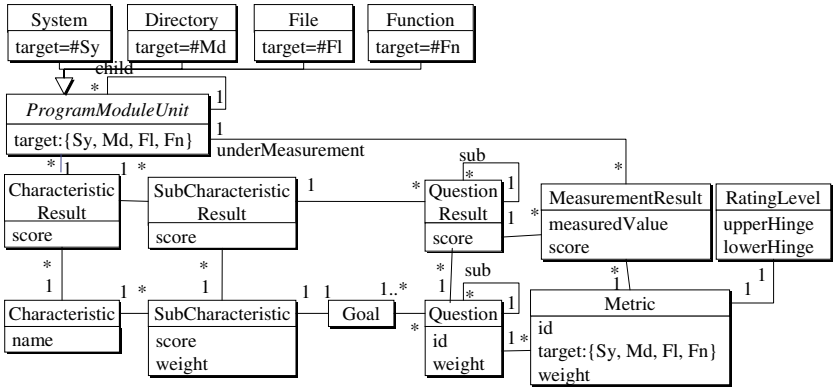

Fig. 6. Characteristic/module unit score calculation/aggregation model

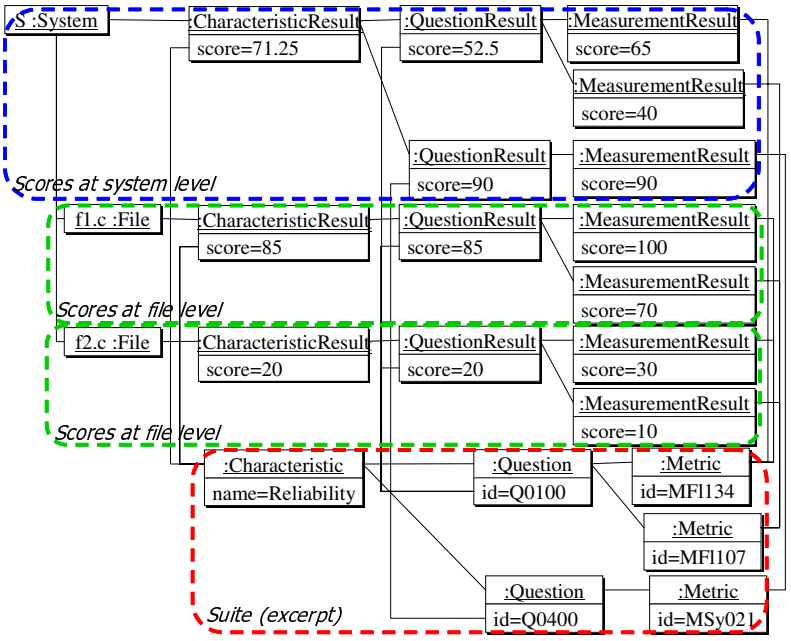

Fig. 7. Example of calculating the score using the aggregation model

S, is shown. Further, the evaluation is made based on only two files, `f1.c` and `f2.c`, and not on all directories. In Figure 7, the reliability score for S (71.25) is calculated for two questions from the scores from three metrics. Since the metrics `MF1134` and `MF1107` apply directly to the files, the average of the scores from measurement values from `f1.c` and `f2.c` was used. Figure 7 shows that detailed scores for each quality-characteristic or question can be obtained for the whole

system or for individual module (file) units (e.g. the reliability score for `f1.c` is 85). This normalization/aggregation technique has been implemented using Excel macros.

(d) Visualization tool and example of use
We implemented a visualization tool which displays the scores obtained from normalization and aggregation in module units for each quality characteristic, and allows detailed inspection based on module-inclusion relationships. The tool is implemented in Ruby and generates an evaluation report consisting of a collection of linked HTML pages using the scores calculated in Excel. An example is shown in Figure 8. The person evaluating the code can use the generated pages to get a comprehensive understanding of quality trends from the module level up to the overall system.

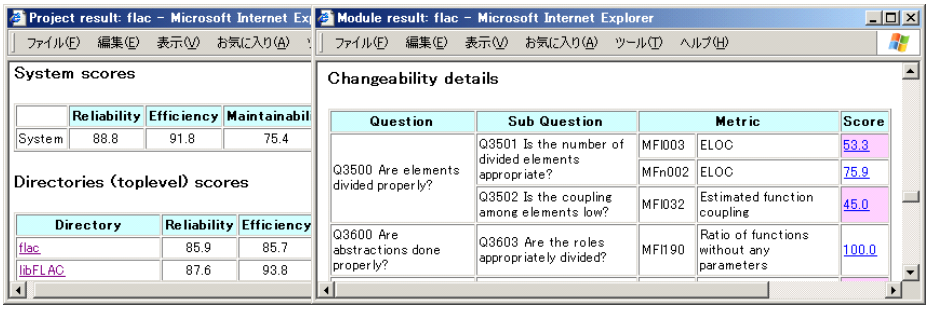

Fig. 8. Report examples (left: system/directory, right: characteristic in detail)

3.3 Applicable Scope of the Framework

Because the framework covers quality from the overall system down to a detailed level, it can be used to evaluate quality over a wide range, from management level down to the individual module developer. Specifically, the scores can be used to identify and prioritize problematic characteristics or parts that need quality improvements. Also, if a range of allowable scores (e.g. 75 to 100 points) is set as an assessment criterion for an organization or project, scores can be used as a non-functional requirement during the development or procurement process.

The framework can be used in the following situations:

- Implementing or procuring C programs in the embedded software domain: Entire framework can be reused.
- Implementing or procuring C programs in the non-embedded software domain: All of the framework except for the derived rating level described in this paper can be reused if a collection of acceptable source code (or source code to which some quality improvements have been made) is available in the problem domain. If such a sample is not available, all of the framework

except the rating level and technique for deriving a rating level can be reused, and another technique for deriving a rating level can be incorporated in the framework.
- Implementing or procuring programs in other languages: The goals and questions within the suite which are language independent can be reused.

4 Experimental Evaluation

In the following, we evaluate the validity and usefulness (especially quality improvement reflection capability) of the framework by using several real programs.

4.1 Validity of the Framework in Quality Evaluation

We evaluate the validity of the framework by comparing two evaluation results for the same set of source codes: a qualitative evaluation by using a questionnaire, and a quantitative evaluation by using the framework.

First, we created a table of questions to evaluate each quality sub-characteristic with a four-level score (0, 50, 75 or 100 points), and applied it to the three embedded software programs (S_1, S_2, S_3) that were used to derive the rating level in section 3.2. The programmer in charge of each program before improvements or the person accepting the program after improvements was asked to perform this qualitative evaluation by answering the questions. Table 3 shows the average results of this evaluation in quality-characteristic units[2]. "Before" and "After" in the table show the results for the code before and after quality improvements were made. For two of the program, S_1 and S_2, the qualitative evaluation showed improvement for almost all quality characteristics.

Table 3. Qualitative quality results using the query table

	Reliability		Efficiency		Maintainability		Portability		Reusability	
	Before	After	Before	After	Before	After	Before	After	Before	After
S_1	92	92	80	83	75	95	69	100	92	100
S_2	59	79	67	71	54	78	76	88	60	83
S_3	–	92	–	78	–	75	–	88	–	83

Next, we compared the results of the qualitative evaluation described above with the quantitative results from the framework in order to verify the validity of the framework. The quantitative evaluation results for each of the programs, before and after improvement, are shown in Table 4. We examine the validity of the framework for each quality characteristic below:

- Reliability, maintainability and reusability: As with the qualitative evaluation results, the quantitative evaluation results for each of these characteristics

[2] Due to some reasons, the before-improvement qualitative results for S_3 were not available.

showed improvement, suggesting that the framework is valid for them. However, one program (S_3) did not show improvement in reusability afterwards, so it may be necessary to add additional metrics and corresponding quality mappings.
- Portability: The quantitative result for programs S_2 and S_3 showed improvement afterwards, so the framework may be useful for this evaluating this quality characteristic. However, the improvement seen in the qualitative evaluation of S_1 was not reflected in the quantitative evaluation, so it may be necessary to adjust or add to the metrics or quality mappings used.
- Efficiency: For all programs, the quantitative evaluation results showed different tendencies than the qualitative evaluation results for before and after quality improvements, suggesting that the metrics used were not appropriate. One reason for this may be that it is fundamentally difficult to estimate the final system's efficiency by only using the source code[24]. We will need to make revisions to the metrics used here.

From the above-mentioned results, it is found that the framework can be used effectively to give quantitative evaluations of reliability, maintainability, reusability and portability of source code.

Table 4. Quantitative quality evaluation results using the framework

	Reliability		Efficiency		Maintainability		Portability		Reusability	
	Before	After	Before	After	Before	After	Before	After	Before	After
S_1	79	83	96	92	80	88	87	86	80	92
S_2	88	99	99	96	74	89	94	96	0	95
S_3	85	90	96	86	67	75	77	82	0	0

4.2 Quality Improvement Reflection Capability of the Framework

We used another embedded program which controls a Japanese shelf of gods for verifying the quality improvement reflection capability of the framework. In an earlier version, the program had maintenance problems such as the heavy use of global variables and the very long `main()` function. To solve these problems, we restructured the program by shifting global variables to non-global variables (such as local variables) and by dividing long functions into small ones. The excerpts of the programs before and after improvements are the following.

```
/*************** shrine.c, before improvements ***************/
extern int mic_threshold; extern int show_mic_value; ...
void main(void) {
    MY_ADCSR.BYTE = 0x31; while(!MY_ADCSR.BIT.ADF);
    PADDR = 0x7F; PADR.BIT.B2 = 0;
    PADR.BIT.B3 = 0; PADDR = 0x0C | PADDR; ...
/*************** shrine.c, after improvements ***************/
int main() {
    start_microphone(); init_switch(); init_motor(); ...
```

Figure 9 shows the quantitative evaluation results using the framework for each of the programs. The result of the after-improvement reflects a significant improvement in maintainability since several metrics related to maintainability provide different values. Regarding this example, it is found that the framework has the quality improvement reflection capability. Note that efficiency has been slightly decreased in the after-improvement due to the division of functions; this is a typical example of tradeoff between maintainability and efficiency.

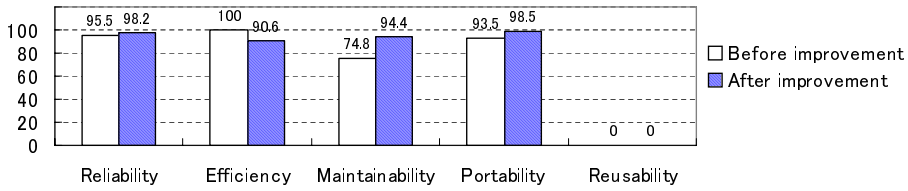

Fig. 9. Quantitative results using the framework for `shrine.c`

5 Conclusion and Future Work

In this paper, we propose a framework for evaluating the quality of program source code in order to resolve several problems faced by existing techniques. The framework focuses mainly on the C programming language and incorporates a quality metrics suite, a normalization and aggregation tool, a rating level derivation tool, and a set of actual rating levels. The framework is useful mainly for evaluating the quality of C language source code in module units from a detailed level up to whole systems and from individual quality sub-characteristics up to overall system quality. It would also be possible to apply the framework to source code in other languages by changing the sub-questions and metrics in the measurement suite. We verified that the framework can be used effectively to evaluate programs for reliability, maintainability, reusability and portability by applying it to several embedded software programs.

In further research, we plan to re-examine the metrics for some of the quality characteristics (particularly efficiency) to improve the accuracy of our quality evaluation by investigating the relation between the internal measured values (obtained by the framework) and possible external measurement values. Also, by applying the framework to many more programs, we will investigate how effective it is, and how it depends on the problem domain.

References

1. Ogasawara, H., et al.: Evaluating Effectiveness of Software Metrics, Union of Japanese Scientists and Engineers, 20SPC Research subcommittee report (2004)
2. Chaudron, M.: Evaluating Software Architectures, `http://www.win.tue.nl/mchaudro/swads/`

3. Sindre, G., et al.: The REBOOT Approach to Software Reuse, Journal of Systems and Software, 30(3) (1995)
4. Bansiya, J., Davis, C.G.: A Hierarchical Model for Object-Oriented Design Quality Assessment, IEEE Transactions on Software Engineering, 28(1) (2002)
5. Supervised by Kanno, A., et. al.: Software quality maintenance technology for the 21st Century, Union of Japanese Scientists and Engineers (1994)
6. Ortega, M., et al.: Construction of A Systematic Quality Model for Evaluating A Software Product, Software Quality Journal, 11(3) (2003)
7. Monden, A.: A Study of Data Collection using EPM and Analysis using GQM. In: 4th Empirical Software Engineering Workshop (2005)
8. ISO/IEC TR 9126-3: Software engineering – Product quality – Part 3: Internal metrics (2003)
9. ISO/IEC 9126-1: Information technology – Software product evaluation: Quality Characteristics and Guidelines for their use (2001)
10. ISO/IEC 14598-1: Information technology – Software product evaluation: Part 1: General overview (1998)
11. Washizaki, H., et al.: A Metrics Suite for Measuring Reusability of Software Components. In: Proc. 9th IEEE International Software Metrics Symposium (2003)
12. Hirayama, M., et al.: Evaluating Usability of Software Components, Information Processing Society of Japan Journal, 45(6) (2004)
13. Programming Research Ltd.: QAC, http://www.programmingresearch.com/
14. Telelogic: Logiscope, http://www.telelogic.com/corp/products/logiscope/
15. Basili, V.R., Weiss, D.M.: A Methodology for Collecting Valid Software Engineering Data, IEEE Transactions on Software Engineering, 10(6) (1984)
16. http://www.ogis-ri.co.jp/otc/consulting/euml/documents/QAFramework.html
17. The Motor Industry Software Reliability Association: MISRA-C: 2004 – Guidelines for the use of the C language in critical systems (2004), http://www.misra-c2.com/
18. IPA/SEC: C-Language Coding best practices for Embedded software Guide, Shoeisha Inc. (2006)
19. Emi, K., Lewerentz, C.: Applying Design-Metrics to Object-Oriented Frameworks. In: Proc. 3rd IEEE International Software Metrics Symposium (1996)
20. ISO/IEC 15939:2002, Software engineering – Software measurement process (2002)
21. McCabe, T.J., Watson, A.H.: Software Complexity, Crosstalk, Journal of Defense Software Engineering, 7(12) (1994)
22. Kazman, R., et al.: Making Architecture Design Decisions: An Economic Approach, CMU/SEI-2002-TR-035 (2002)
23. OMG: UML 2.0 OCL Specification, http://www.omg.org/docs/ptc/05-06-06.pdf
24. Washizaki, H., et al.: Experiments on Quality Evaluation of Embedded Software in Japan Robot Software Design Contest. In: Proc. 28th International Conference on Software Engineering (ICSE 2006), pp.551–560 (2006)