An Integrated Expert System Framework for Software Quality Assurance*

Stephen S. Yau
Software Engineering Research Center and
Computer and Information Sciences Department
University of Florida
Gainesville, Florida 32611

Yeou-Wei Wang, Jules G. Huang and Jinshuan E. Lee
Department of Electrical Engineering
and Computer Science
Northwestern University
Evanston, Illinois 60208

Abstract

The objective of this paper is to present a software quality assurance framework using knowledge based engineering technology. It can provide knowledge based assistance for quality assurance throughout the entire software development cycle. To ensure high quality software and achieve cost-effective software development and maintenance, software metrics are used during the entire software development cycle to measure and predict the quality of software products. Various metrics for software attributes for all phases of the software development cycle will be collected and stored in an object-oriented database. The framework uses the metrics and the Dependency-Based Truth Maintenance System to detect poor software quality factors, and provides suggestions for improvements.

Index Terms—Software quality assurance, software development cycle, software metrics, quality prediction, improvement suggestions, expert systems, object-oriented database.

1 Introduction

To ensure high quality software and achieve cost-effective software development and maintenance, software metrics should be applied during the entire software development cycle to measure and predict the quality of software products. Software metrics \[1-3\] are normally used to characterize the essential features of software quantitatively so that classification, comparison, and quantitative analysis can be applied. The primary purpose of using software quality metrics is to improve the quality of the software product by specifying it in the software requirement and by predicting and measuring software quality during various phases of the software development cycle.

In this paper we will present a framework to integrate the information available from the software quality framework \[4-6\] with knowledge based engineering technology. This system is used to plan the software quality assurance activities, evaluate systems designs, balance mutually conflicting quality factors, and make design refinement suggestions. It can provide knowledge based assistance to software developers for performing quality assurance throughout the entire software development cycle. Various metrics for software attributes will be collected and stored in an object-oriented database for all phases of the software development cycle.

2 An Integrated Expert System Framework

In the software quality framework proposed by Cavano and McCall \[4\], software quality is described as a hierarchical model shown in Fig. 1 covering all phases of the software development cycle. At each key milestone in the development process, the quality of the software can be evaluated and an early feedback can be obtained to let the developers know whether to continue or stop the development process. In addition, it is a complete measurement of software quality and offers a wide range of standpoints for software quality from the user-oriented, software-oriented, and quantitative-oriented points of view.

Our expert system framework is shown in Fig. 2, and the functions of its components are summarized as follows:

- The Object-Oriented Data Base (OODB) holds all software quality information, such as factors, criteria, and metrics values, which is obtained from the Automated Measurement System (AMS) \[7\].
- The Inference Engine is used as the control center to manage the operation flows of the system. It is the main interface of this framework to interact with other components. The Inference Engine uses inferential knowledge stored in Meta Rules (MR) and Rule Set (RS).
- The Dependency-Based Truth Maintenance System (DTMS) is used to maintain the truth in the OODB and provides suggestions for software quality improvement.

Our framework offers the following advantages: 1) Expressiveness: rules and facts are more expressive than programs and data in the traditional approach. 2) Expandability: the developer can add his/her knowledge into the expert system at any time. 3) Integration: rules are very useful tools as situation-
action mechanism for integrating the whole system. 4) Information hiding; object-oriented data base provides three different granularity of objects, i.e. factors, criteria and metrics. The developers only have to understand the meaning of factors while the structure of criteria and metrics is known by software quality assurance engineers only.

3 Object-Oriented Data Representation

There are four levels of software items in the software quality framework [4], which are SYS (system), CSCI (computer software configuration item), CSC (computer software component) and CSU (computer software unit). System Specification documents, System Design documents, etc. are software document items which belong to objects in the SYS level. Software Requirement Specification documents, Software Design documents, etc. are software document items which belong to objects in the CSCI level. Memory and processing time allocation are described for each CSC in the Software Design documents. The description for each CSC appears as a paragraph in the Software Design documents. The basic coding unit is in the CSU level. Each software code module represents a CSU.

The phase and level relationship for the software development cycle is shown in Fig. 3. In each phase, the software component at each level has its own software quality factors, criteria, and metrics. There are the following four classes of objects in our data base:

- Information Item (II): Each software component at each level is represented by an object of the II class. The object points to its own quality information objects, its documents, and its subcomponents. For an object in the II class in the SYS level, it consists of the documents that describe the whole system, e.g. System Specification documents, System Design documents and Interface Requirement documents. A SYS level II object also consists of the names of its subcomponents in the CSCI level. An example for this hierarchy is given in Fig. 4.
- Development Phase Item (DPI): For an object in the DPI class, its phase of the software development cycle and software quality factors are represented. An information item will have several development phase items as its quality information sub-objects.

![Figure 3: Phase and level relationship for software development.](image)

- Quality Factor Item (QFI): Each object in the QFI class has its corresponding quality-criterion-item sub-objects. The value of the factor is stored here. An object in the DPI class has several QFI objects.
- Quality Criterion Item (QCI): Each object in the QCI class has its own metrics and the value of the criterion. This is the lowest level of the four classes.

An example for quality information hierarchy is given in Fig. 5. Detailed description of the slots and method of each of these classes is given as follows:

- Information Item (II)
  CLASS II
  Name: Name of this object.
  Body: A set of pointers point to programs or document body (only the CSU level components have program body, while other level objects have document).
  Subcomponents: A set of pointers point to this object's...

![Figure 4: Software components hierarchy.](image)

- Development Phase Item (DPI)
  CLASS DPI
  Development-phase-name: Name of this object.
  Factor-information: A set of pointers point to this object's quality factor items.
  Get-factor: Retrieve a quality factor. Which factor to be retrieved depends on the message this object receives.

- Quality Factor Item (QFI)
  CLASS QFI
  Quality-factor-name: Name of this object.
  Value: Value for this factor.
  Criterion-information: A set of pointers point to object's quality criterion items.
  Get-and-calculate: Get all the criteria for this factor and calculate the factor value.

- Quality Criterion Item (QCI)
  CLASS QCI

162
First, the knowledge based system must decide where to start the reasoning process. Second, the Inference Engine must resolve conflicts when alternative lines of reasoning emerge. There are usually a list of actions with temporal information constraining the order of actions in the quality assurance activities. Since there are many quality factor/criterion/metric relations, there must exist some way in controlling what action is to be taken. To control which quality factor/criterion/metric is most important, meta rules are used in assisting the developer to make decisions.

There are three meta rules in MR that will help the developer to choose a rule from RS to be applied next. The first meta rule is used to control the invocation of rules to be activated, and is used in controlling quality rule invocation at each milestone of the development process. The second meta rule is concerned with the usefulness of a rule, and is used to tell the developer whether a particular rule is useful in improving the software quality. The third meta rule is concerned with the temporal ordering among the rules to be chosen and to select which quality rule is to be used. They are especially useful for selecting QGRs. Following are the formats of the three meta rules:

Meta-Rule1:
When <some.milestone> is reached,
Invokes Automatic Quality Collection Rules,
Quality Monitoring Rules,
Quality Improvement Rules, and
User's Goals Satisfied Rules.

Meta-Rule2:
When <condition(s)> is true,
If rules do{not} have <characteristics>,
Then they will be z% certain they are useful.

Meta-Rule3:
When <condition(s)> is true,
Rules which do{not} have <characteristics.X> should be used <order> with z% certain than
Rules which do{not} have <characteristics.Y>.

Meta-Rule2 and Meta-Rule3 are used to resolve conflicts when alternative lines of reasoning emerge. When we develop a large scale software system, RS for that software system is generally very large. Choosing the best rules from RS to be applied can depend upon Meta-Rule2 and Meta-Rule3. For Meta-Rule2 and Meta-Rule3, <condition(s)> is a boolean expression that this meta rule has to be satisfied, <characteristics> is a set of characteristics that the meta rule has, and <order> is a partial or
A certainty factor dering which indicates the ordering of the rules from RS to be chosen. A certainty factor 2% can be included by the project manager to change the importance of believe depending on different situations.

5 Dependency-Based TMS

The progression of a single piece of software under development can be regarded as a graph, in which the nodes are artifacts (data or modules), and the links are derivation paths (design decisions). Artifacts are related based on some design decisions. Design decisions can be represented as dependency relations.

All dependency relations are persistent relations; that is, whenever they are established, they will be there until users change their designs whereupon new dependency relations need to be established. To represent the persistent relations and efficiently rebuild dependency relations, hypothetical reasoning inference engines are considered. Hypothetical reasoning refers to solution approaches in which assumptions may have to be made to enable the search procedure to proceed. However, later along the search path, it may be found that certain assumptions are invalid and therefore have to be retracted [9]. In our framework, assumptions can be users' design decisions or users' expected quality factor values.

Hypothetical reasoning can be handled by nonchronological backtracking. It keeps track of the assumptions that support the current search path and backtracks to the appropriate branch point when the current path is invalidated. A related capability is truth maintenance [10,11] which removes derived beliefs when their conditions are no longer true. Two primary features of the Truth Maintenance System (TMS) - persistent relations representation and dependency-directed backtracking - make it possible to represent and update the persistent dependency relations of metrics and criteria (or criteria and factors).

The data objects of our framework are related. There is a dependency relation between every two related data objects: an object and its consequent. The value attribute of an object is affected by the value attribute of its antecedent, i.e. the quality value of an object is either inherited or manipulated from the quality value of its antecedent. For example, X and Y are both objects and Y is X's antecedent. Some value attributes of X are inherited or manipulated from the values of Y. Hence, the values in object slots can be manipulated and the results ripple throughout the logical structure of the system. Such values are called active values. An active object is an object whose value attribute affects another object's value attribute. A passive object is an object whose value attribute is affected by another object's value attribute. In our framework, a passive object has at least one active object, and an active object may affect more than one passive objects. Each data object of our framework may be a passive object or an active object or both.

The organization of objects of our framework can be viewed as an abstraction mechanism. The Dependency-Based Truth Maintenance System (DTMS) is a system which does dependency-directed backtracking and incrementally updates its beliefs when dependency relations are modified. The abstraction mechanisms considered in the DTMS are: generalization/specification, and aggregation/decomposition. The first abstraction mechanism generates a taxonomy of classes known as an IS-A hierarchy. The second abstraction mechanism captures the PART-OF relationship between a parent class and its component classes. Thus, the other way of describing data objects of our framework is either the IS-A relationship or the PART-OF relationship. In the DTMS, the IS-A relationships exist between classes and subclasses. The PART-OF relationships are used to represent the hierarchical relationships of software quality factors. A culprit represents a metric that lowers a quality factor value of the software. A ripple phenomenon is a phenomenon that a deficiency in the early phase of the software development cycle affecting the product quality of the later phase.

For a low software quality factor value, each time when the DTMS algorithm is applied, a culprit will be found. The DTMS algorithm can be iteratively applied to find culprits for a software quality factor or factors. The mechanism that DTMS uses to find the culprit(s) is dependency-directed backtracking. If there are two or more quality factors that do not achieve the expected standard, the user should pick up a factor according to his preferences or design purposes to ask for software quality improvement suggestions. The DTMS algorithm is summarized as follows:

Step 1: Invocation: DTMS is invoked by the Inference Engine for verifying and improving the software quality.

Step 2: Backtracking and Culprit Finding: DTMS invokes dependency-directed backtracking to find the culprit for the chosen quality factor.

Step 3: Improvement Suggestions Passing: DTMS will signal and pass software quality improvement suggestions to the Inference Engine.

Step 4: Users Actions: Users check quality checklists and correct errors. Then the system quality is regenerated.

Step 5: Exit: DTMS exits based on the comparison of the quality values with the expected quality values:
- Goal achieved: If all the quality values are larger than the expected ones, then returns control from the DTMS to the Inference Engine.
- No major changes: Due to some fatal design errors, some software may not be improved to achieve user's expected quality values. Therefore, after a fixed number of iterations, if the difference between the quality factor values of two adjacent versions of the improved software is smaller than a certain value, the system will stop making quality improvement suggestions for that quality factor. If all the unsatisfied quality factors have been processed, then DTMS will stop processing and inform the Inference Engine of the software quality analysis.
- Iterative: If the above two conditions are not satisfied, go to Step 2.

6 An Example

In this section, we will give an example to illustrate how our framework works. This example is to develop a software system that solves a set of linear system equations using the Gaussian Elimination Method (GEM). In this example (GEM), only the reliability factor quality is considered. The user's expected reliability factor values during the SRA, DD, and QFR phases are all assumed to be 0.95. We will show how the QGRs are used to provide design guidelines, how the quality information is stored in the OODM, and how DTMS can find the culprits in the metric elements and provide improvement suggestions.

Before the SRA phase, QGRs will be used to provide guidelines in writing the requirement specification for GEM. Since the reliability factor depends on the Accuracy (AC), Anomaly (AM) and Simplicity (SI) criteria, we will use the rules in QGRs to improve AM, AC, and SI. Following the guidelines provided by the QGRs, the requirement document of GEM is written as follows:

A mathematical function is needed to solve a system of linear equations. It is expected that the Gaussian Elimination Method is used and the function be implemented in a high level programming language like PASCAL. The function takes a set of real coefficients (with 5 decimal points precision) as input and a set of real values (with 5 decimal points precision) as output. There are no more than 500 system equations. All constants used inside this function have precision of 5 digits after the decimal point. All subscripts must be range-tested. All output must be verified before final outputting. Error conditions must be identified for invalid input/output values and wrong system equation sizes.

The criteria and reliability values calculated using the data in Table 1 are given as follows:
<table>
<thead>
<tr>
<th>AC</th>
<th>AM</th>
<th>SI</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC1</td>
<td>1.00</td>
<td>AM1</td>
</tr>
<tr>
<td>AM2</td>
<td>1.00</td>
<td>SI2</td>
</tr>
<tr>
<td>AM3</td>
<td>0.75</td>
<td>SI3</td>
</tr>
<tr>
<td>AM4</td>
<td>N/A</td>
<td>SI4</td>
</tr>
<tr>
<td>AM5</td>
<td>N/A</td>
<td>SI5</td>
</tr>
<tr>
<td>AM6</td>
<td>N/A</td>
<td>SI6</td>
</tr>
</tbody>
</table>

Table 1: The metric values for the GEM example in the Software Requirement Analysis phase.

$$\text{Accuracy}(AC) = \text{AVE}(AC_1) = 1.00$$

$$\text{Anomaly}(AM) = \text{AVE}(AM_1, AM_2, AM_3, AM_4, AM_5, AM_6, AM_7) = 0.75$$

$$\text{Simplicity}(SI) = \text{AVE}(SI_1, SI_2, SI_3, SI_4, SI_5, SI_6) = 0.835$$

Finally, we calculate the reliability value (RL) at the end of the SRA phase as follows:

$$RL = \text{AVE}(AC, AM, SI) = 0.86$$

There are many CSCI level items for a large-scale software project, and hence there are many objects in the Class II. Fig. 6 shows the quality information hierarchy for GEM. In this particular example, there is only one object GEM in the Class II. The "Body" of this object is the requirement specification. The object GEM is shown as follows:

**CLASS II**

object

Name: GEM

Body: documents for this object

Subcomponents:

Level: CSCI

Phases: (GEM.SRA\_AC, GEM.SRA\_PD, ...)

The software project GEM has to flow through the software development cycle. Fig. 6 shows there are six objects in Class DPI: GEM\_SRA, GEM\_PD, GEM\_DD, GEM\_CUT, GEM\_CSCI\_IT and GEM\_CSCI\_IT. These objects are all subobjects of the object GEM. The following object represents the GEM\_SRA which is the object for the SRA phase.

**CLASS DPI**

object

Development-phase-name: GEM\_SRA

Factor-information: (GEM\_SRA\_Re\_1, GEM\_SRA\_Ef\_1, ...)

In the SRA phase, all the software quality factors should be evaluated based on the requirement specification. In this example, because only the reliability factor is considered, the structure of the object GEM\_SRA\_Re is shown as follows:

**CLASS QFI**

object

Quality-factor-name: GEM\_SRA\_Re

Value: 0.86

Criterion-information: (GEM\_SRA\_AC\_1, GEM\_SRA\_AM\_1, GEM\_SRA\_SI)

There are three objects in Class QCF for the project GEM in the SRA phase: GEM\_SRA\_AC, GEM\_SRA\_AM, and GEM\_SRA\_SI. Each of the three objects represents a software criterion for the reliability factor, and is given as follows:

**CLASS QCI**

object

Quality-criterion-name: GEM\_SRA\_AC

Value: 1.0

Metric-information: (1.0)

To find the culprits using DTMS algorithm, unrelated object details are skipped and the dependency-directed backtracking is represented by the ⇒ sign. Operation flows of DTMS are summarized as follows:

Step 1. Based on the quality data of the example, the data object hierarchy for object GEM is constructed. Since the Reliability factor does not achieve the expected standard, it is chosen as the target.

Step 2. Dependency-directed backtracking is invoked to find the culprit, and the dependency-directed backtracking path is: From (GEM, ..., GEM\_SRA, GEM\_PD, ...) ⇒ (GEM\_SRA\_Re, GEM\_SRA\_AM) ⇒ (GEM\_SRA\_AM, 0.75, 0.5, 0.0, 0.75). We find that the Anomaly Management criterion (0.75) is the principal shortcoming that lowers the value of GEM's Reliability; the Error Tolerance/Control metric (0.5) is the major flaw that makes the Anomaly Management criterion value low. It is identified as the culprit, and the dependency-directed backtracking paths are shown in Fig 7.

Step 3. Through the Inference Engine, DTMS provides the user with the quality improvement suggestions, i.e. the Error Tolerance/Control metric in the Anomaly Management criterion should be improved.

Step 4. The user checks the Error Tolerance/Control metric checklists, and some errors are identified. But, no processing instructions for recovery or repair are provided and a standard for handling errors is needed. Therefore, the user provides processing instructions for recovery or repairs errors and builds a standard for handling errors,
which then increase the Error Tolerance/Control metric to 1.0. System software quality metrics are remeasured. The Anomaly Management criterion is then increased to 0.92, and the Reliability factor is increased to 0.92.

**Step 5.** After the new quality values and the expected quality values are compared, DTMS finds that the new quality values are still smaller than the expected quality values. Therefore, the operation goes to Step 2.

The second invocation of DTMS will take the Design Structure metric as the culprit. The user checks the checklists and includes diagrams for identifying CSCI functions in a structured fashion. The Design Structure metric is increased to 1.0, the Simplicity criterion is raised to 1.0, and the Reliability is improved to 0.97. Since the new Reliability factor value is greater than the expected value (0.95), DTMS stops.

After the DTMS algorithm, the requirements were appended with: The error conditions will be handled by a procedure that will handle all the error conditions in the system. Diagrams should be provided for identifying all CSCI functions in a structured fashion.

The revised criteria and reliability values calculated by using the data in Table 2 are given as follows:

$$RL = AVE(AC, AM, SI) = 0.97$$

After the SRA phase, the same process for developing quality software program will continue to the later phases of the software development cycle.

**7 Discussions**

In this paper, we have presented an framework for software quality assurance. This framework involves knowledge engineering technology which uses object-oriented data base to store the knowledge (the software quality information), rules and meta rules as its inferential knowledge. A subsystem, Dependency-Based Truth Maintenance System, based on the hypothetical reasoning is used for design evaluation of the software quality. The integration of our knowledge base with the software quality management system provides a wide range of support to the development of large-scale software systems. It can support all phases of the software development cycle. Different software quality factors are considered to validate the design decisions.

In designing a software quality assurance system, we separate knowledge from inference and control. The quality information knowledge is put in the OODB and the situation-action knowledge is put in the RS. Software quality engineers can easily modify their own knowledge and data. Project managers can easily change their working environment by modifying the rules and meta rules. Because any new tools can be easily integrated into our system by adding some rules as its interface, the automatic tool invocation can be achieved.

The software quality assurance activity is a comprehensive view of the whole software development process. In our integrated knowledge based quality assurance system, the software quality is measured by a wide range of software quality factors, and covers the entire software development cycle. This system is especially useful to those large-scale software development teams. Unlike some of the current software quality assurance tools which concentrate on limited activities or on a specific application domain, our system provides a wide range of activities and is not limited to a particular application domain. The developers who use our system can use the knowledge from the OODB and the RS for quality design improvement. Our system shall play an important role in monitoring the software quality assurance activities and providing suggestions to the software developers.

**References**


