

DTIC COPY

AD-A223 064

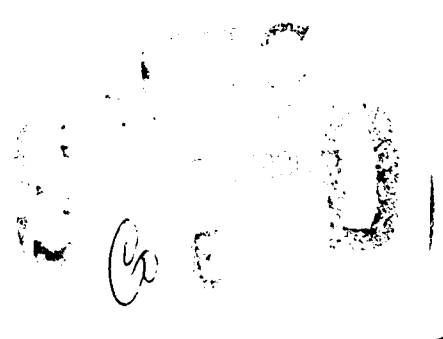
RADC-TR-90-14
Final Technical Report
March 1990



KNOWLEDGE BASED QUALITY ASSURANCE TOOLS

Northwestern University

Stephen S. Yau, Gwo-Long Huang, Jinshuan Lee, Yeou-Wei Wang



APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

90 06 20 037

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Services (NTIS) At NTIS it will be releasable to the general public, including foreign nations.

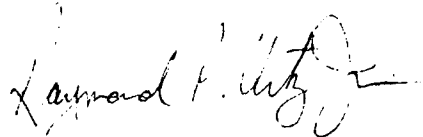
RADC-TR-90-14 has been reviewed and is approved for publication.

APPROVED:



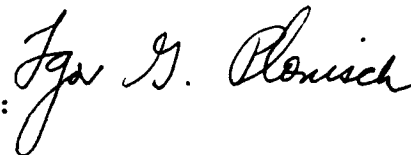
JOSEPH P. CAVANO
Project Engineer

APPROVED:



RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER:



IGOR G. PLONISCH
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COEE) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b RESTRICTIVE MARKINGS N/A	
2a SECURITY CLASSIFICATION AUTHORITY N/A		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.	
2b DECLASSIFICATION/DOWNGRADING SCHEDULE N/A		5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-90-14	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A		5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-90-14	
6a. NAME OF PERFORMING ORGANIZATION Northwestern University	6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COEE)	
6c. ADDRESS (City, State, and ZIP Code) Department of Electrical Engineering and Computer Science Evanston IL 60208-3118		7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center	8b. OFFICE SYMBOL (If applicable) COEE	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-81-C-0185	
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700		10. SOURCE OF FUNDING NUMBERS	
		PROGRAM ELEMENT NO. 615581	PROJECT NO. 5581
		TASK NO. 20	WORK UNIT ACCESSION NO. P7
11. TITLE (Include Security Classification) KNOWLEDGE BASED QUALITY ASSURANCE TOOLS			
12. PERSONAL AUTHOR(S) Stephen S. Yau, Gwo-Long Huang, Jinshuan Lee, Yeou-Wei Wang			
13a. TYPE OF REPORT Final	13b. TIME COVERED FROM Nov88 TO Nov 89	14. DATE OF REPORT (Year, Month, Day) March 1990	15. PAGE COUNT 100
16. SUPPLEMENTARY NOTATION N/A			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB-GROUP	
12	05		
		Truth Maintenance System Quality Measurement, Expert Systems	
		Quality Assurance	
19. ABSTRACT (Continue on reverse if necessary and identify by block number)			
A long range approach for integrating software quality information with knowledge based engineering technology was developed. A Software Quality Assurance Expert System framework was proposed to plan software quality assurance activities, evaluate systems designs, balance mutually conflicting quality factors, and make design refinement suggestions. This effort determined the basic system architecture and interaction among system components. The proposed expert system framework would include data objects of an Object-Oriented Data Base, a Rule Set, Meta Rules, and a Dependency-Based Truth Maintenance System. To help illustrate how such a system could be used, examples were provided to show how the expert system could assist Software Quality Assurance activities for software reliability. The DOD community will benefit from the results of this work; particularly, anyone attempting to use expert systems to improve the quality of their software.			
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED//UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Joseph P. Cavano		22b. TELEPHONE (Include Area Code) (315) 330-4476	22c. OFFICE SYMBOL RADC (COEE)

UNCLASSIFIED

UNCLASSIFIED

Contents

1	Introduction	1
1.1	The Objective of the Project	1
1.2	Organization of the Report	2
2	Background	4
2.1	Software Quality Assurance (SQA) Framework	4
2.2	Knowledge Representation	6
2.3	Inference	7
2.3.1	Rules, Meta Rules, and Planning	8
2.3.2	Hypothetical Reasoning - Truth Maintenance Systems	9
2.4	Current Expert System Building Tools	10
3	An Expert System Framework for Software Quality Assurance	14
3.1	System Architecture	15
3.2	Characteristics of Our Expert System	18
4	Object-Oriented Data Representation for SQA	20
4.1	Advantages of Object-Oriented Data Representation	21
4.2	Features and Terminologies in the Object-Oriented Representation	23
4.3	Objects and Classes	25
5	Quality Assurance Activities with Rules	30
5.1	Rule Set (RS)	31
5.2	Meta Rules (MR)	37
5.3	Examples in the Quality Assurance Activities	39
6	Dependency-Based TMS (DTMS)	41
6.1	Motivations for Using the DTMS	41
6.2	Definitions	42
6.3	DTMS Approach to Efficient Search	44
6.4	DTMS Algorithm	47
7	An Integrated Example	49
8	Development Issues	58
8.1	Expert System Building Tool Selection	58
8.2	Application Scope for SQAESF	60

9	Conclusions and Future Work	62
9.1	Summary of the Results	62
9.2	Future Work	64
A	Metric Questions for Reliability Factor	67
A.1	Software Requirement Analysis Phase	68
A.2	Detailed Design Phase	70
A.3	Coding and CSU Testing	74
B	Detailed Design and Coding for GEM	78
B.1	Detailed Design of GEM	78
B.2	Coding of GEM	80
	Bibliography	87

List of Figures

2.1	A software quality measurement framework.	5
3.1	A software quality assurance expert system framework.	16
4.1	Phase and level relationship for software development.	22
4.2	Software components hierarchy.	26
4.3	Software quality information hierarchy.	26
7.1	The object hierarchy for the GEM example.	52
7.2	The quality values for the GEM example in SRA Phase.	53
7.3	The dependency-directed backtracking paths of the GEM example. .	56

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



List of Tables

7.1	The metric values for the GEM example in the Software Requirement Analysis Phase.	51
7.2	The revised metric values for the GEM example in the Software Requirement Analysis Phase.	57

Chapter 1

Introduction

1.1 The Objective of the Project

The objective of this project is to develop a long range approach for integrating information available from the software quality framework [1-2] with knowledge based engineering technology. In order 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, such as reliability, portability, efficiency, maintainability, etc.

The purpose of this project is to provide knowledge based assistance for performing quality assurance, validation and verification throughout the entire software development cycle to aid software acquisition managers and software developers. Various metrics for software attributes, such as accuracy, anomaly, simplicity, consistency, traceability, modularity, simplicity, will be collected and stored in an object-oriented data base for all phases of the software development cycle, such as software requirement analysis phase, preliminary design phase, detailed design phase, coding and testing phase, and maintenance phase. The results can be applied during the entire software cycle to assist the software developer and maintainer in predicting the quality expected for the final product in the early stages so that error correction, optimization, functional and performance enhancement, and improvement of efficiency can be taken early when they are more effective and economical.

One of the most important goals for software engineering researchers is to improve the quality of software product. Software Quality Assurance (SQA) is an activity to improve the software quality [3-7]. The requirements for a software quality program in DOD-STD-2168[7] are evaluation of software, evaluation of software documentation, evaluation of the processes used in software development, evaluation of the software development library, etc. According to the ANSI/IEEE standards [4], SQA has been

defined as "planned and systematic pattern of all actions necessary to provide adequate confidence that the software conforms to establish technical requirement". A number of solutions for improving software product quality were proposed in the past decades [3]. Three kinds of issues regarding to the software quality assurance activities were discussed in [3]. They are managerial issues concerning the planning, organization, control, standards, practices, and conventions; technical issues concerning the requirement specification, design, programming, testing and validation; and issues concerning the use of software development tools in the software development process.

In order to increase the effectiveness of software management, we are dealing with the questions, such as how can we do a better job of project planning? and what type of development process should we choose? Since the software quality depends upon the skill and performance of all the programmers who work on it, we are also concerned with the standards, such as documentation standards, programming standards, and SQA standards. In the technical issues, we are concerned with the software construction methodology, i.e., the construction of requirement specification, design, and coding (programming). After the software products have been developed, we are concerned with the testing and validation of such software products against the initial requirement specification. Finally, we can have a set of tools assisting the development of a software product throughout the entire software development cycle. In general, there are tools for all phases of the software development cycle (i.e., tools for requirement analysis, tools for design, tools for implementation, and tools for testing), tools for planning, tools for key development support systems (i.e., approaches to the software engineering environments).

Software metrics [8-11] 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 software requirement and by predicting and measuring software quality during various phases of software development cycle. The concepts can improve quality since they are based on achieving positive influence on the product. In order to assure software quality, a framework for software quality measurement was proposed [1], and an Automated Measurement System (AMS) [12] was developed for collecting information on various software quality metrics during the software development cycle.

1.2 Organization of the Report

In this project, a Software Quality Assurance Expert System Framework (SQAESF) is presented. This system is used to plan the software quality assurance activities, evaluate systems designs, balance mutually conflicting quality factors, and make design refinement suggestions.

In this report, we will present a framework for developing such an expert system for software quality assurance. In Chapter 2, the background information used in this report is given. A comparison of the current expert system technologies according to different knowledge representations and inference methods will be made. Also, some of the technologies used in our framework including the use of object-oriented data base, rules and meta rules as the inferential knowledge, and truth maintenance system, will be described. In Chapter 3, an overview of our expert system framework for quality assurance is presented. In this chapter, the basic system architecture and interactions among system components will be described. In Chapters 4 to 6, details of this expert system framework are presented. They include data objects of the Object-Oriented Data Base (OODB), the Rule Set (RS), the Meta Rules (MR), and the Dependency-Based Truth Maintenance System (DTMS). In Chapter 7, an integrated example will be given to illustrate how our framework can be used at various phases of the software development cycle. Specifically, the requirement analysis phase will be used to show how a requirement specification can be developed following the metric question guidelines. In Chapter 8, we will discuss what would be needed for implementing this expert system framework. Comparisons of the current expert system building tools will be presented and suggestions in selecting an appropriate expert system building tool will be made. In Chapter 9, we will discuss what are needed in order to fully utilize the potential and what the difficulties and limitations will be. Finally, in Appendix A, metric questions for some phases of software development cycle will be listed and Appendix B will contain the detailed design and coding for the integrated example described in Chapter 7.

For illustration purpose, the software reliability factor will be used to show how the expert system can assist the Software Quality Assurance activities. The reliability of computer-based systems (particularly embedded systems) within the Department of Defense (DoD) has been a subject of considerable concern for a number of years [13]. For most DoD systems, the reliability of a system is critical to effective mission performance.

Chapter 2

Background

2.1 Software Quality Assurance (SQA) Framework

Software quality is a combination of many conflicting factors, such as reliability versus efficiency and efficiency versus integrity [2]. In the software quality framework proposed by Cavano and McCall [1], software quality is described as a hierarchical model. This model, as shown in Figure 2.1, has its highest level as a set of software quality factors which are user/management-oriented terms and represent the characteristics which comprise the software quality. At the next level, for each quality factor, there is a set of criteria which are the attributes that, if present in the software, provide the characteristics represented by the quality factors. At the lowest level of the model are the metrics which are quantitative measures of the software attributes defined by the criteria. Each metric is represented by a set of metric elements usually in the form of check lists. Several metric elements, completed at several points in the software development cycle, may be combined in calculating for a single metric.

There are 13 user-oriented quality factors of software product: reliability, correctness, efficiency, integrity, survivability, usability, maintainability, verifiability, expandability, flexibility, interoperability, portability, and reusability. 29 quality criteria which are software-oriented attributes, are defined to provide a more detailed representation of what a particular software quality factor means. Each criterion consists of one or more quality metrics which are defined by one or more metric elements. Metric elements are detailed questions concerning the software products.

For example, reliability evaluates software failures. The formula used is in terms of the total number of software errors in total executable lines of code. The reliability consists three criteria: Accuracy, Anomaly, and Simplicity. Accuracy means the attributes of the software which provide the required precision in calculations and

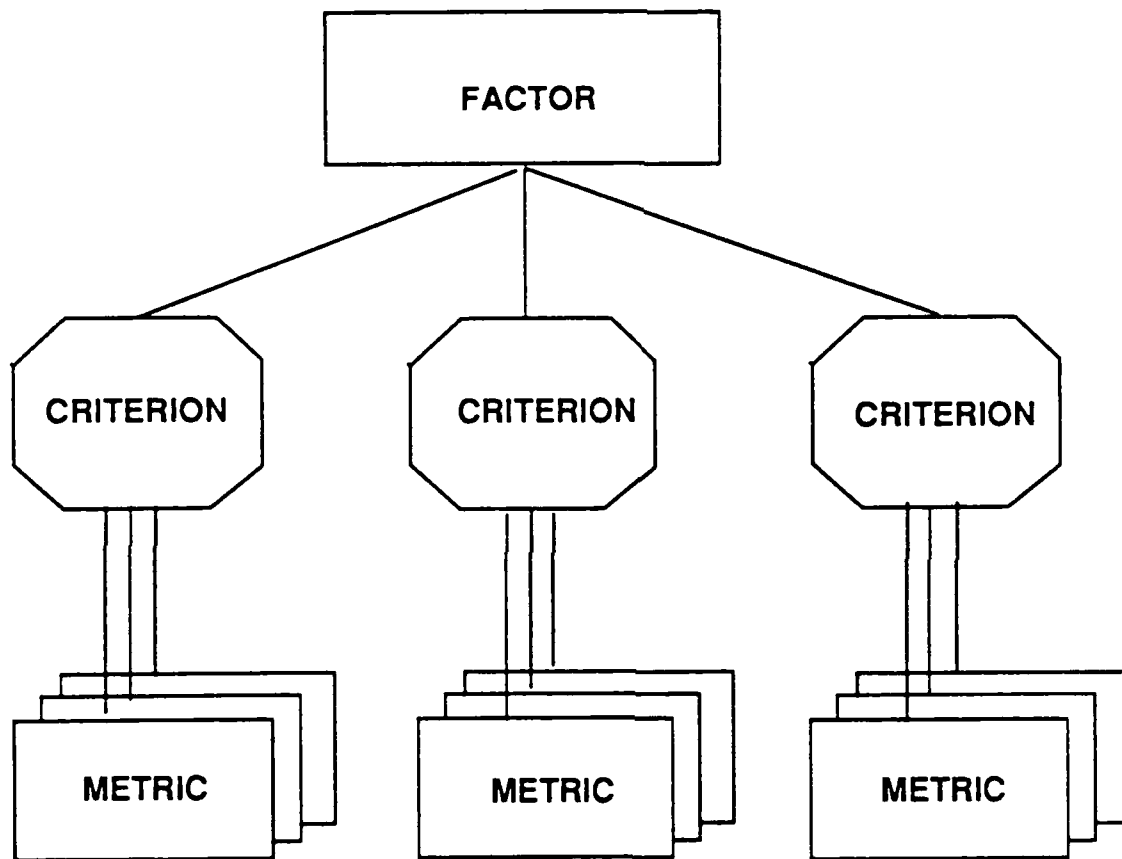


Figure 2.1: A software quality measurement framework.

output. Anomaly means those attributes of the software which provide for continuity of operations under and recovery from nonnominal conditions. Simplicity means those attributes of the software which provide for the definition and implementation of functions in the simplest and understandable manner. Each criterion is evaluated by some questions which best describe the quantitative measures of the attributes.

This hierarchical model for software quality offers several advantages. First, it covers all the 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. Second, it is a complete measurement of the software quality and offers a wide range of standpoints for the software quality from the user-oriented, software-oriented, or quantitative-oriented points of view.

2.2 Knowledge Representation

There are many different ways to encode the facts and relationships that constitute knowledge [15-16]. The most widely used approaches are semantic networks, context-parameter-value triplets, frames, logical expressions, and rules.

Most general representational scheme for knowledge is the semantic network (or semantic net) [16]. A semantic network is a collection of objects represented by nodes, which are connected together by arcs or links. Ordinarily, both the links and the nodes are labeled. There are no absolute constraints as to how nodes and links are named.

Another common way to represent knowledge information is context-parameter-value (or object-attribute-value) triplets [17]. In this scheme, contexts may be physical entities such as a door or a transistor, or they may be conceptual entities such as a logic gate, a bank loan, or a sales episode. Parameters are general characteristics or properties associated with contexts. The final member of the triplet is the value of a parameter, which specifies the specific nature of a parameter in a particular situation.

Frames [18] provide another method for representing knowledge information. A frame is a description of an object that contains slots for all of the information associated with the object. Slots, like parameters, may store values. Slots may also contain default values, pointers to other frames, sets of rules, or procedures by which values may be obtained. The inclusion of these additional features makes frames different from context-attribute-value triplets. From one perspective, frames allow for richer representations of knowledge.

Logic [19] provides another way to represent knowledge information. There are several logical notations. Two most common forms are propositional logic and predicate calculus. Propositional logic is a common logic system. Propositions are statements that are either true or false. Propositions that are linked together with connectives, such as AND, OR, NOT, IMPLIES, and EQUIVALENT, are called compound statements. Propositional logic is concerned with the truthfulness of compound statements.

Predicate calculus [19] is an extension of propositional logic. The elementary unit in predicate logic is a term (an object). Statements about objects are called predicates. Each predicate is either true or false. Predicates can address more than one term. Ordinary connectives can be used to link together predicates into larger expressions.

Rules [20] are used to represent relationships. One of the simplest rule is an if-then rule. In an if-then rule, there is an expression or an if clause as the premise part. The other single expression or then clause is taken as a conclusion.

2.3 Inference

Inference and control strategies (inference engine) in a knowledge based system use the facts and rules stored in the knowledge base. The inference engine of an expert system stands between the user and the knowledge base. The inference engine performs two major tasks: First, it examines existing facts and rules and adds new facts when possible. Second, it decides the order in which inferences are made.

Several different inference mechanisms have been explored. Forward chaining, hypothetical reasoning, and meta-control are widely used approaches. In our expert system, the forward chaining (rules) and the hypothetical reasoning (the truth maintenance) are used [21].

Forward chaining starts with data to be input or with the situation currently present in a global data base. The data or the situation is matched with the antecedent conditions if part in each of the relevant rules to determine the applicability of the rule to the current situation. One of the matching rules is then selected. For example, meta rules help the user determine the order in which the rules should be tried. The rule's consequents then part are used to add information to the data base or to actuate some procedure that changes the global situation.

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, certain assumptions may be found that are invalid and therefore have to be retracted. This makes the data base of the hypothetical reasoning nonmonotonic; that is, a hypothetical reasoning is also a nonmonotonic reasoning. This hypothetical

reasoning in our expert system is handled by the truth maintenance approach. This approach is to keep track of the assumptions that support the current search path and to backtrack to the appropriate branch point when the current path is invalid.

2.3.1 Rules, Meta Rules, and Planning

The word *planning* refers to the process of computing several steps of a problem-solving procedure before executing any of them [22]. Generally, a planning system can perform the following functions:

- Choose the best rule to apply next based on the best available heuristic information. One way of doing this is first to isolate a set of differences between the desired goal states and the current state, and then to identify those rules that are relevant to reducing those differences (like in the means-ends analysis method exploited by General Problem Solver [23]). This process can also be assisted by using meta rules in guiding which of the rules is to be applied next.
- Apply the chosen rules to compute the new problem state that arises from its application. After the rules are selected for execution, applying rules are easy.
- Detect if a solution has been found. A planning system has succeeded in finding a solution to a problem when it has found a sequence of operators that transforms the initial problem state into the goal state. In our expert system, the goal of the planning system is a high quality software product.
- Detect deadends so that they can be abandoned and the system's effort can be directed to more fruitful direction to find a solution. As the planning system is searching for a sequence of operators to solve a particular problem, it must be able to detect the path that can never lead to a solution. In our case, when the quality planning system detects a plan which leads to a low quality value, the exploring path should be terminated and a new plan should be generated.

There are many planning approaches capable of carrying out actions in the real world in order to achieve some definite purposes. One of the earliest techniques to be developed for solving compound goals that might interact with each other was the use of a goal stack. This was usually referred to as linear planning, and was used by STRIPS [24] in the robot's world. In this method, the planner makes use of a single stack that contains both goals and operators that have been proposed to satisfy those goals. Data base and a set of operators (PRECONDITION, ADD, and DELETE) are used to control the actions. Although it offers simplicity in the planning of the robot's world, it has a very limited application domain.

Another planning approach, the case-based planning [25], formulating new plans and debugging old ones by using experiences stored in the memory. Since the experiences and past plans are so important to the case-based planning, it usually involves some theory of learning and memory organization. Since this approach relies heavily on the past experience to make new plans, the application areas in using case-based planning are still very limited.

Many AI programs have had the ability to break a problem into subproblems, that is, to find a solution by a divide-and-conquer strategy. However, a program uses a hierarchical approach only if it has the additional capability to defer consideration of the details of a problem. *Hierarchical planning* [27-28] is one of the planning approaches which distinguishes between important considerations and details. The hierarchical planning creates descriptions of abstract states and divides its planning task into subproblems for refining the abstract states. This is also referred as the *least commitment planning* and has been used successfully in MOLGEN (plans gene cloning experiments in molecular genetics). In this approach, the decisions are deferred as long as possible, and thus the planner's option is kept open.

Since in the hierarchical planning, the detailed decisions are often deferred as long as possible, a meta-plan [28] is usually used to perform the planning process. The meta-plan provides a framework for partitioning control knowledge into layers so that flexibility is achieved without the complexity of a large monolithic system. The hierarchical planning can reduce exponential problems to problems with linear complexity [29].)

In order to define the actions using the planning system, there must be rules for performing actions and rules for guiding which rules to be used. We define those rules for performing actions (improving the quality of the software quality) as quality rules, and those rules guiding which rules to be used as *meta rules* [30]. In Chapter 5, we will see that the rules are used as a guideline to assist the software quality assurance activities at various phases of the software development cycle. The meta rules are strategies to guide the use of knowledge in the application domain. The meta rules can be used for rule selection and justification.

2.3.2 Hypothetical Reasoning – Truth Maintenance Systems

Computer reasoning programs usually construct computational models of situations. To keep these models consistent with new information and changes in the situations being modeled, the reasoning programs frequently need to remove or change portions of their models. These changes sometimes lead to further changes, for the reasoner often constructs some parts of the model by making inferences from other parts of the model.

One approach to keep computational models consistent is to record the reasons for believing or using each program beliefs, inference rules, or procedures. In [31] a program called the Truth Maintenance System (TMS) which supports a powerful form of backtracking called dependency-directed backtracking [32] is used to determine the current set of beliefs from the current set of reasons, and to update the current set of beliefs in accordance with new reasons in an incremental fashion.

Several systems based on the TMS model have been produced [33-36]. There is, however, no one common implementation strategy. Instead, all these systems have to be modified to satisfied the needs of the particular application domain. The exact form of the data structure of these systems depends on the design representation they used. These truth maintenance systems adopt the same view of problem solving as a conventional TMS.

2.4 Current Expert System Building Tools

Expert systems are also called knowledge based systems. For consistency, they will be called expert systems throughout this report. Expert system building tools (expert system shells) can be divided into four categories: inductive, simple rule-based, structured rule-based, and hybrid system building tools [14,15,21,37]. We will review them briefly.

- *Inductive tools.* Inductive tools generate rules from examples. These tools come in two sizes: large inductive tools, which run on mainframes and PCs, and small inductive tools, which run only on PCs. They are derived from experiments conducted in machine learning. These tools are useful for simple tasks that rely on examples, but cannot be used to develop complex knowledge representations.
- *Simple rule-based tools.* Simple rule-based tools use **if-then** (situation-action) rules to represent knowledge. They can be run on PCs. They lack context trees as well as some other editing features commonly available in structured rule-based tools.
- *Structured rule-based tools.* Structured rule-based tools can be run on mainframes, such as VAX's, LISP machines, UNIX workstations, and PCs. They tend to offer context trees, multiple instantiation, confidence factors, and more powerful editors. These tools use **if-then** rules arranged into sets. These rules are like separate knowledge bases. These systems are more desirable in case where a large number of rules are involved, if the rules can be subdivided into sets.

- *Hybrid tools.* Hybrid tools can be run on LISP machines, VAX's, UNIX workstations, and PCs. These tools use object-oriented programming techniques to represent elements of each problem and support hypothetical reasoning. The system will work on objects. An object can contain facts, **if-then** rules, or pointers to other objects. They are designed to build systems that contain 500 or several thousand rules and may include the features of several different consultation paradigms.

The first two categories of expert system building tools (expert system shells) are the inductive tools and the simple rule-based tools. The expert system building tools of the first category are useful for simple tasks that rely on examples, and those of the second categories are good for systems with rules less than 500 rules. Since large-scale software is too complex to be described using these tools, they are not suitable for software quality assurance application and will not be discussed further here.

The other two categories of expert system building tools (expert system shells) are structured rule-based tools and hybrid tools. The structured rule-based tools are divided into two types: mid-size rule-based tools and large rule-based tools. M.1 and KES are considered mid-size rule-based tools, and S.1 is considered a large rule-based tool. Finally, ART, KEE, and Knowledge Craft are considered hybrid tools. Because these tools are suitable for our use we will discuss them briefly.

M.1 [38] is a product of Teknowledge Inc. M.1 was introduced in 1984 as a tool to assist knowledge engineers in prototyping knowledge systems. In M.1, facts are represented as attribute-value pairs (attributes are called expressions) with accompanying confidence factors. Production rules represent heuristic knowledge. The M.1 inference engine is a simple back-chainer in the tradition of MYCIN. Users interact with M.1 by tying answers to questions posed by the system. There are explanation facilities as well as trace functions for knowledge based debugging. On-screen menus are available that list appropriate responses. M.1 was originally implemented in PROLOG and has since been released in C programming language. M.1 is available for IBM PC XT and IBM mainframes.

Knowledge Engineering System (KES) [39] is a production of Software Architecture and Engineering (Software A&E). Facts in KES are represented as attribute-value pairs with associated confidence factors or probabilities. Attributes and values can be arranged in hierarchies. Relations among facts are represented as production rules, using statistical pattern classification techniques, or with hypothesis test cycles. Users communicate with KES through a line-oriented interaction. Multiple-choice questions provide a list of alternatives, and users choose one by number. KES is implemented in Wisconsin LISP, a dialect of LISP. It is also available written in FranzLISP for DEC VAX running under UNIX operating system, for CDC CYBER running under A-LISP, and for Apollo workstation running under Portable Standard LISP (PSL). KES is also implemented in IQLISP for IBM PC XT. C version is also available.

S.1 [40] is a knowledge engineering tool offered by Teknowledge, Inc. S.1 is an integrated package of software, training, sample knowledge systems, documentation, maintenance, and support. Facts in S.1 are stored as object-attribute-value triplets with associated confidence factors. Objects (also called classes) can be grouped into class types. Other relationships are represented as production rules. Existential and universal quantifiers may be included in the premises of rules. The S.1 inference engine is backward chaining. Users communicate with S.1 by selecting items from a menu with a mouse, or by entering information with the keyboard. Help and explanation facilities are available. S.1 is implemented in LISP and is available on Xerox 1100, Symbolics 3600, and VAX computers.

Automated Reasoning Tool (ART) [41] from Inference Corporation is a tool kit for knowledge system development. The kit contains four major components: a knowledge language, a compiler, an applier, and a development environment. ART provides a number of representations to store and maintain facts. One of the representations is the traditional O-A-V triplets. A second means of representation, called a fact, is a proposition with a truth value and a scope. The inference engine or knowledge applier is described as being an opportunistic reasoner. This means that ART can reason with both forward and backward chaining, or with explicit procedural commands. Rules affect the direction of inference. In this way the inference engine moves opportunistically, depending on the pattern of intermediary results. ART also supports confidence ratings. ART has a wide variety of interface features, all oriented toward helping the knowledge engineers to develop an expert system. The tool is flexible enough that a skilled knowledge engineer can use ART to develop whatever user interface he or she desires. ART is written in LISP and runs on TI's Explorer, DEC VAX, Sun workstation, IBM PC RT, and LISP machines (produced by Xerox and Symbolics).

Knowledge Engineering Environment (KEE) [42] is an integrated package of software tools available from IntelliCorp (formerly IntelliGenetics). In 1983 IntelliCorp began selling KEE, a hybrid tool derived from its work with genetic engineering software. KEE's basic knowledge representational paradigm is frames, which unify the procedural and declarative expression of knowledge. The inference scheme for KEE is quite flexible. It can be programmed to behave as a backward chainer or a forward chainer. Values in slots can be manipulated, and results ripple throughout the logical structure of the knowledge base. KEE can take advantage of a truth maintenance system for planning or simulation problems [43]. There is no sharp distinction between the consumer and the creator of KEE knowledge base. The user interface provides a number of graphics features that aid the knowledge engineer in the development and debugging of a knowledge based system. KEE is a hybrid system and therefore can be extended by the knowledge engineer. KEE is implemented in LISP and is available on Xerox 1100 machine, Symbolics, LMI LAMBDA, TI Explorer, Apollo, Sun, and IBM PC RT workstation.

Knowledge Craft [15] was first introduced in 1985 by the Carnegie Group. Knowledge Craft is based on a semantic network approach. The basic knowledge representation paradigm in Knowledge Craft is the schema network. Each schema can have any number of associated slots. By making extensive use of the meta-information associated with the slots in a schema, Knowledge Craft allows the developer to provide default values, demons, cardinality restrictions, and range and domain restrictions. Knowledge Craft provides object-oriented programming techniques to permit data abstraction, object specialization, and the passing of information via message. In Knowledge Craft, inheritance is specified at the relation level. It means that a user can specify which slots and values can be included and which can be excluded in any particular relationship at the time the relationship is established. Knowledge Craft provides a context mechanism that allows different versions of the knowledge base. This is used to model and test alternative situations. Knowledge Craft lacks truth maintenance. Knowledge Craft provides both forward and backward chaining. Knowledge Craft is written in Common Lisp and is available on Symbolics, TI's explorer, DEC VAX, HP AI workstation, and IBM PC RT workstation.

The rule capacities of an expert system building tools (expert system shells) should be considered for different scales of software projects. Hybrid expert system building tools are suitable for large scale software project. Further discussions and comparisons of expert system building tools for our use are given in Chapter 8.

Chapter 3

An Expert System Framework for Software Quality Assurance

The primary purpose of using software quality information is to improve the quality of software product by specifying software quality requirements and by measuring software quality. To achieve this objective, an expert system framework for software quality assurance is presented. This framework provides software quality monitoring, software development evaluation, conflicting factors balancing, and refinement suggestions in each phase of the software development cycle.

Traditionally, a data base system only stores facts. They do not have the ability to store inferential knowledge. With the advent of knowledge base systems, facts and inferential knowledge can be stored together.

An expert system consists of two parts: knowledge base and inference engine. The inference engine uses the knowledge in the knowledge base to do inferences. Rules, meta rules, and facts are the major components in the knowledge base. Rules and meta rules contain the control information for the expert system. Facts represent the data information for the expert system. The inference engine uses the control strategy specified in rules and meta rules to retrieve, insert, delete, or modify facts. In our approach, the data base used to represent facts is object-oriented. The reasons for choosing object-oriented data base are stated in Chapter 4. Rules, meta rules and a truth maintenance system are used to represent the inferential knowledge. Instead of developing an inference engine, we will select an off-the-shelf expert system building tool, and suggestions for the selection of expert system building tools will be given in Chapter 8.

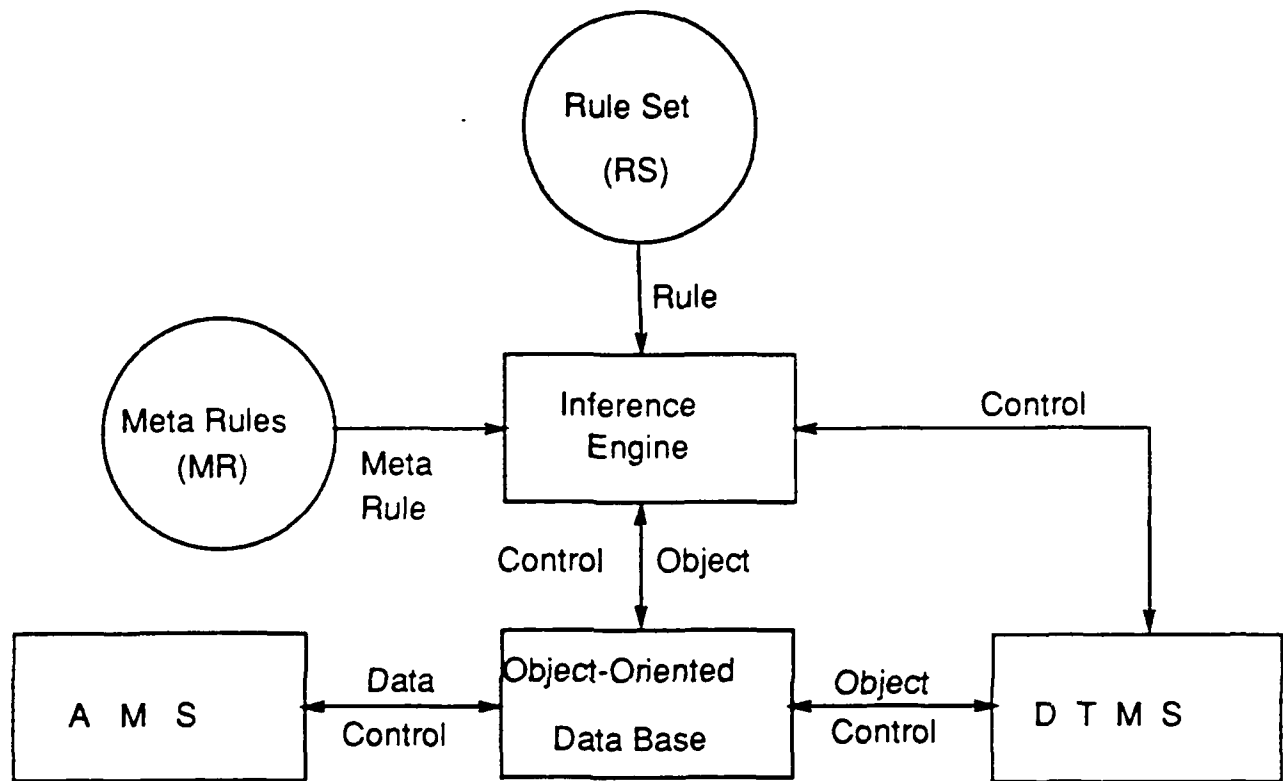
3.1 System Architecture

Our expert system framework, which is shown in Figure 3.1 consists of the following major components: the Object-Oriented Data Base (OODB), the Inference Engine using the Rule Set (RS) and the Meta Rules (MR) as inferential knowledge, and the Dependency-Based Truth Maintenance System (DTMS). Functions of each component are stated as follows:

- The Object-Oriented Data Base holds all software quality information, such as factors, criteria, and metrics values. The software quality information is obtained from the AMS.
- The Inference Engine is used as the control center to manage the operation flows of the system. The Inference Engine provides only non-hypothetical reasoning. It is the main interface of this expert system to interact with other tools. The Inference Engine uses inferential knowledge stored in the Meta Rules and the Rule Set:
 - The Meta Rules (MR) contain the meta rules provided by our expert system to specify the priority of the rules to be activated.
 - The Rule Set contains rules, such as Quality Guideline Rules, Quality Monitoring Rules, Automatic Quality Collection Rules, User's Goal Satisfied Rules, and Quality Improvement Rules (described in Chapter 5). These rules in the Rule Set are provided by our expert system. When a rule from the Rule Set (RS) is interpreted in the Inference Engine, some actions will be performed.
- The Dependency-Based Truth Maintenance System (DTMS) uses hypothetical inference strategy. It is used to maintain truth in the Object-Oriented Data Base and provides suggestions for software quality improvement.

We will present an expert system framework for software quality assurance during the software development cycle. Meta rules are selected by the Inference Engine to control where the system is to start and which rule the system is to select from the Rule Set (RS) to be executed. When our expert system is invoked by the user, the Inference Engine will choose a rule from the RS based on the current development phase (i.e., requirement specification phase, design phase, coding phase, etc.) to activate other components in the system.

For example, the Inference Engine will select one of the Quality Guideline Rules (QGRs) in the Rule Set (RS) based on the meta rules from the MR to provide design guidelines before starting each major phase of the software development cycle. After some milestone in the software development cycle, the Inference Engine will select one of the Automatic Quality Collection Rules (AQCRs) in the RS based on the meta



DTMS - DEPENDENCE-BASED TRUTH MAINTENANCE SYSTEM

AMS - RADCAUTOMATED MEASUREMENT SYSTEM

Figure 3.1: A software quality assurance expert system framework.

rules from the MR. The Inference Engine will execute the rule and send a command to the Object-Oriented Data Base to retrieve quality factor information from the AMS. The Inference Engine will also select one of the Quality Monitoring Rules (QMRs) in the RS based on the meta rules from the MR to constantly monitor the current software product quality values using the information stored in the Object-Oriented Data Base. Whenever the system detects a low quality value, one of the Quality Improvement Rules (QIRs) in the RS will be selected by the Inference Engine to activate the DTMS. Otherwise, one of the User's Goals Satisfied Rules (UGSRs) in the RS will be selected by the Inference Engine to inform users that the goals of the software quality have been achieved.

In the software quality framework [1], quality factors are measured at the end of each phase. The quality factor information will be acquired from different levels of software items (system level, unit level, etc.). The quality factor information represents the integrated quality factor information for the software product at the end of each phase. The Object-Oriented Data Base transforms data (quality factor information) from the AMS to object forms under the control of the Inference Engine. The Inference Engine will use quality factor information to make inference and the DTMS will use quality factor information to do backtracking.

The responsibility of the Object-Oriented Data Base is to provide our expert system with a reliable storage for quality factor information. The Object-Oriented Data Base obtains the commands (retrieve, store, etc.) from the Inference Engine (or the DTMS). When it receives the commands, the control of our expert system will be passed to the Object-Oriented Data Base. After the commands are executed, the control will be returned from the Object-Oriented Data Base to the Inference Engine (or the DTMS).

After all the facts (in object forms) are stored in the Object-Oriented Data Base, based on current quality factor value, one rule in the RS will be fired by the Inference Engine. This rule compares the user's expected quality factor values with the actual quality factor values. If any of the actual quality factor values is smaller than the user's expected corresponding quality factor value, one of the QIRs in the RS will be invoked and the control will be passed to the DTMS for quality improvement suggestions. If all the design quality factor values (from the Object-Oriented Data Base) are greater than the user's expected corresponding quality factor value, one of the UGSRs in the RS will inform the developer that the goal of software quality has been achieved and the operation will stop.

After the DTMS is invoked by the Inference Engine, the DTMS will activate the dependency-directed backtracking for quality improvement suggestions. Based on the data objects stored in the Object-Oriented Data Base, the dependency-directed backtracking searches the cause (a metric element) for low quality factors. Each time the dependency-directed backtracking is invoked by the DTMS, a metric element that causes low quality factors will be found. After all the metric elements resulting in low quality factors are found, control will be returned from the DTMS to the Inference

Engine and quality improvement suggestions will be stored in the Object-Oriented Data Base.

The Inference Engine informs the developer of the quality analysis of the software product during each phase. Based on the quality improvement suggestions in the Object-Oriented Data Base, if the quality factor values are not satisfied, the Inference Engine will ask the developer to modify the design.

3.2 Characteristics of Our Expert System

Our expert system will have the following characteristics which will make our expert system capable of handling large scale software development projects:

- Expressiveness: Traditionally, the knowledge for software development process is usually stored in a data base using development tools. The tools are usually written in procedural languages. In our system, the knowledge for software development and software quality assurance is represented by rules and facts. The concepts of rules and facts can be understood and modified more easily than tools written in procedural languages. The project manager can easily understand what a fact is and what a rule is.
- Expandability: The situation-action knowledge for software development and software quality assurance is expressed in the form of rules. Knowledge with structured properties are represented as facts. The project manager can expand this knowledge by incorporating his/her own knowledge in the expert system and use the expert system to assist him/her during the software development process.
- Granularity: In our expert system framework, many kinds of objects are proposed. Each one gives a different level of details of the whole software product and consequently gives us abstractions of the data structure by hiding details. Therefore, the activities for performing software quality assurance are more manageable and reliable. For example, factors, criteria, and metrics represent three kinds of granularity objects. A project manager only needs to consider what the software quality (i.e., quality factors) is, while a quality engineer has to consider the detailed structure of the quality hierarchy, factors, criteria, and metrics.
- Different kinds of inference: 1) Deduction is used in rules and meta rules. 2) Direct connection is used in structured objects. 3) The dependency-based truth maintenance system is used to support non-monotonic inference. These three inference methods allow the software developers to present their inferential knowledge for software development and software quality assurance easily. Thus, the

inference engine of our expert system for software quality assurance is more powerful.

- Integration: The situation-action rules act as a monitor to control the progress of the product for project managers. For example, using this mechanism, the AMS can be activated automatically. Many tools can be integrated with this system, e.g., planning packages, administration packages, and packages that track the discrepancy reports. By adding rules in connection with development tools, the development tools can be automatically invoked at appropriate time. Integration of all the tools through situation-action rules allows project managers to have the whole picture of the status of the project.
- Well-defined quality status: Information describing the quality status of the software product is incorporated into this framework. Any low quality parts of the product can be located as soon as possible. The project manager can get an early warning for any low quality parts during the software development process.
- Interactive Capability: At any time of the software development process, the project manager can modify the software quality assurance activities by modifying the rules. This interactive capability provides the managers with the control over the course of software quality assurance activities.

Chapter 4

Object-Oriented Data Representation for SQA

Activities of the software development cycle generate much data (facts), including:

- Documents/programs: Data items in Software Life Cycle Support Environment (SLCSE) [6], such as requirements, functional specifications, designs, user manuals, source codes, and object codes;
- Internal forms of programs: control flow graphs, data flow graphs, program slice and ripple effect information;
- Test data: input to and expected output from tests;
- Software quality information: quality factors, quality criteria, and quality metrics.

Each piece of the data is a development item generated during the software development cycle. The idea to have a data base is to have all the information generated at the specification, design, implementation and testing phases available to the maintenance personnel in a complete, structured and traceable form. In this project, we emphasize the usage of software quality information. The same techniques can be applied to manage and manipulate other information.

There are four levels of software items in the software quality framework [1,6, 44], which are SYS (system), CSCI (computer software configuration item), CSC (computer software component) and CSU (computer software unit). We call the items in these levels SYSLIs (SYS level items), CSCILIs (CSCI level items), CSCLIs (CSC level items) and CSULIs (CSU level items). 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 Software Design documents. The basic coding unit is in the CSU level. Each software code module represents a CSU.

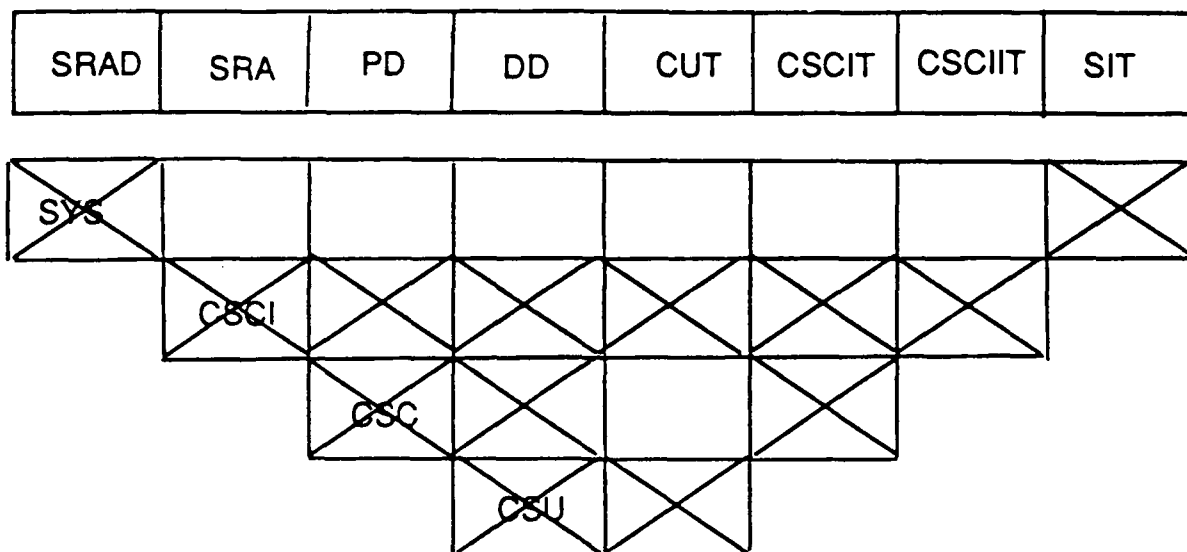
In the software quality framework [1], quality factors are measured at the end of each phase. The quality information will be acquired from different levels of software items. The quality factor information represents the integrated quality information for the software product at the end of each phase. The detailed acquisition stages for different levels of software items are described in Figure 4.1.

4.1 Advantages of Object-Oriented Data Representation

In the data base and knowledge base community, data (fact) models can be classified into two categories: set theoretic models and graph theoretic models. A typical example for set theoretic models is the relational data model. A typical example for graph theoretic models is the object-oriented data model.

The relational data model provides set operators: union, set difference, projection, cartesian product, and selection. The basic unit in the relational data model is a relation. We have the operators, selection and projection, to manipulate a relation, and the remaining operators work on inter-relations. The most important benefit of using the relational data model is that it provides a very simple but expressive model. Relational systems generally perform well when accessing a field in all tuples of a relation, but perform poorly when accessing individual records. This is because set-oriented operators can express full-fledged power on a set instead of a record. One of the derived operators, natural join, is used very often for inter-relations. The natural join is applicable only when both two relations have columns that are named by the same attributes. The natural join operation is generally performed poorly because it has to match the same values in the same attribute of two relations. Matching is a time-consuming job, so relational systems are generally perform poorly on inter-relation operations.

The concept of object-oriented representation has been developed during the last decade. It extracts the features from the *Object-Attribute-Value representation*, *semantic networks* and *frame based representation* in the artificial intelligence community. It also extracts the *type* concept from the programming language community. It provides another dimension to overcome the deficiencies of the relational data model. Instead of using matching such as in the relational data model, it provides direct



SRAD - SYSTEM REQUIREMENT ANALYSIS/DESIGN

SRA - SOFTWARE REQUIREMENT ANALYSIS

PD - PRELIMINARY DESIGN

DD - DETAILED DESIGN

CUT - CODING AND CSU TESTING

CSCIT - CSC INTEGRATION AND TESTING

CSCIIT - CSCI TESTING

SIT - SYSTEM INTEGRATION AND TESTING

SYS - SYSTEM

CSCI - COMPUTER SOFTWARE CONFIGURATION ITEM

CSC - COMPUTER SOFTWARE COMPONENT

CSU - COMPUTER SOFTWARE UNIT

Figure 4.1: Phase and level relationship for software development.

connectivity between a single object and many related objects. Because of this feature, it can access individual records very fast. Its inter-object operation is through direct connectivity instead of matching. It runs faster than relational systems when the number of inter-object (or inter-relation) queries is greater than that of the set oriented queries. The major disadvantage of the object-oriented data model is that it loses the powerful set operators.

A query is an action to retrieve the data (facts). In the software quality assurance knowledge base, it seldom asks a query like "Find reliability factors of *all* the projects in the knowledge base". Instead, it often asks a query like "Find the criteria of the reliability factor in *a* project". We feel that we can exploit all the capabilities provided in an object-oriented knowledge base and give users a unique view of software quality information.

4.2 Features and Terminologies in the Object-Oriented Representation

Traditionally, the Entity-Relationship (E-R) model is used as the conceptual data model for the data base. Relational data model is used as an internal representation for the data base in computer systems. Database Administrator should take the responsibility for conversion from the E-R model to the relational data model. In this section, we will discuss how an object-oriented data model can play a role in the conceptual data model as well as internal computer representation. No conversion between the conceptual data model and the internal representation is required for an object-oriented data representation. The basic constructs for the E-R model are entity and relationship. In an object-oriented data model, we have more constructs, such as IS-A, PART-OF, etc. which will be discussed in detailed in this section.

The ideas, such as *IS-A*, *PART-OF*, *default values* and *procedure attachment* in our object-oriented representation is used in Artificial Intelligence. The ideas, such as *abstract data types*, *hierarchical types* (type and subtype), and *instance-of*, are from object-oriented programming languages. With these features, it is easy to organize a large amount of knowledge into a hierarchical and structured knowledge base. It is more complicated than the Object-Attribute-Value structure mentioned before. The loss of simplicity in the object-oriented representation may complicate the implementation of inference engines and knowledge acquisition routines. However, it is easier to express a complex and highly structured collection of facts and relationships. This is the reason that the object-oriented data model is used as a representation for software quality information.

The term *object* represents a software component having a hidden state and a set of operations or capabilities for transforming the state. Although the term "object-oriented" has been most closely associated with object-oriented programming language, like Smalltalk [45], the concept "object" is used very often in semantic networks and frames. A frame in frame based representations, and a node in semantic networks are similar to an object here. An object is the primitive element of an object-oriented system.

Messages represent interactions between the objects. Every object is an instance of a *class*. All instances of a class have the same message interface; the class describes how to carry out each of the operations available through that interface. Each message is processed by an operation which is described by a *method*. A method is a procedure which describes how to perform the operation. The encapsulation of slots (variables) and methods is similar to abstract data type in programming languages, like Ada. They can provide a modular design for software product. Some object-oriented programming languages still use function calls instead of message passings. Because the message passing method can provide a more modular structure than the function call method, we will use it in this project.

The slots of an object are determined by the class of the object. Each slot can hold one value. The slot options for each slot can specify additional functionalities, such as default values, pointers to its sub-objects (*PART-OF*), and procedure attachments. As we mentioned before, slots in frames have the same abilities as slots in objects.

A *class* is an object that describes the implementation of one or more *similar* objects. A *class* object determines the structure and behavior of a set of other objects, which are called its *instances*. *Classification/Instantiation* is a term to describe this kind of abstraction in which an object class is defined as a set of instances. Classification represents an *instance-of* relationship between an object and its class. When a class is defined to have a slot with default-value slot option, every object in this class will inherit this default value defined in this class.

A *subclass* is a class that inherits structure (slots) and behavior (methods) from an existing class. A *superclass* is the class from which slots and methods are inherited. Hence, if A is B's superclass, B must be A's subclass. The relationship between superclasses and subclasses is called *generalization/specialization*. A subclass must provide a new class name for itself, but it inherits the slot declarations, slot options, and methods of its superclass. Methods can be added or overridden to subclasses, but slots can only be added to subclasses. The subclass concept described here is similar to hierarchical types (type and its subtypes) in programming languages. A class hierarchy represents the IS-A relationship between a superclass and its subclass.

Composite object is composed of two or more objects. Their existence depends on the existence of their constituent objects. The composite object represents the *aggregation/decomposition* concept. A composite object hierarchy captures the *PART-OF* relationship between parent class and its component classes. A composite object has a

single root object, and the root object references multiple child objects, each through a slot.

Procedure attachment is built through the slot option for a slot. Fetching or storing a value in the slot causes the attached procedure to be invoked. This feature lets a procedure constantly monitor the value in the slot. It allows us to set constraints on the value in the slot, propagate the changes to other objects, or other activities. These can all be specified through the attached procedures. The implementation of Truth Maintenance System (TMS) that will be described later is based on this feature.

4.3 Objects and Classes

In each phase of the software development cycle, each 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. For an object of the II class, it will point to its own quality information objects, its documents and its subcomponents. For an object in II class in the SYS level, it will consist of all 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. Figure 4.2 is an illustration for this hierarchy.
- Development Phase Item (DPI): For an object in the DPI class, its phase of the software development cycle and software quality factors are represented. A software product will go through different phases of the software development cycle. An information item will have several development phase items as its quality information sub-objects.
- 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; that means it has no sub-objects. Figure 4.3 shows an example for quality information hierarchy.

The detailed description of each class for their slots and method is given as follows:

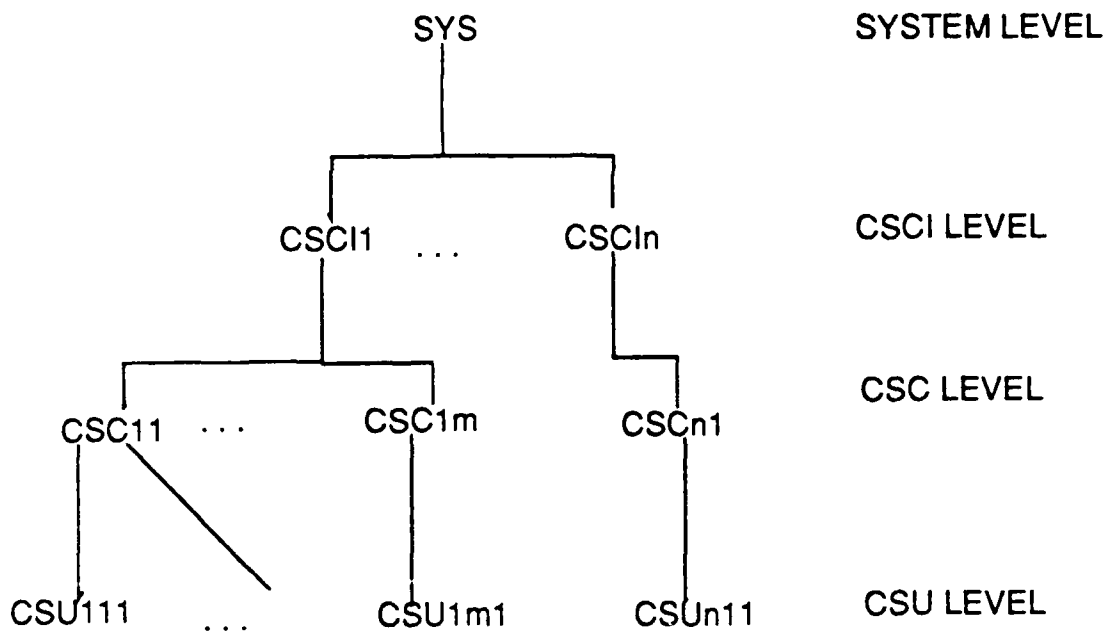


Figure 4.2: Software components hierarchy.

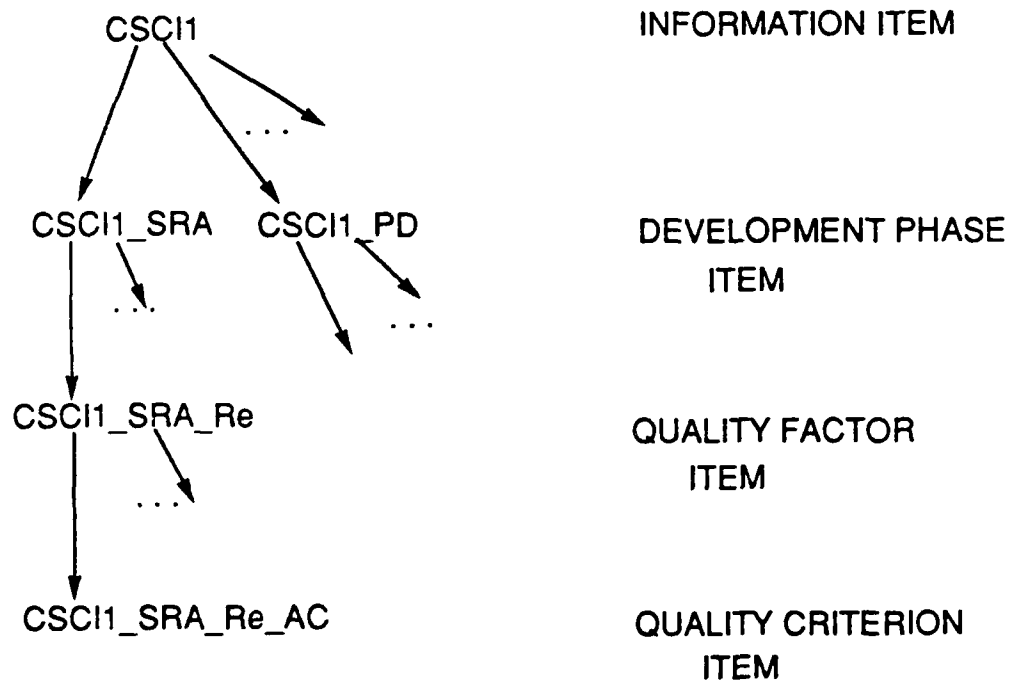


Figure 4.3: Software quality information hierarchy.

- Information Item (II)

CLASS II

(slot)Name:	name of this object.
(slot)Body:	a set of pointers in which each pointer points to the program or document body (only the CSU level components have program body, while other level objects have document).
(slot)Subcomponents:	a set of pointers in which each pointer points to this object's components. It is empty for objects at the CSU level.
(slot)Level:	one of the items in (SYSTEM, CSCI, CSC, CSU).
(slot)Phases:	a set of pointers in which each pointer points to a CSCI level object's development phase items.
(method)Get-subcomponents	get all the names and level information for this object and its subcomponents.
(method)Get-quality-information:	get all the quality factors of this object.

- Development Phase Item (DPI)

CLASS DPI

(slot)Development-phase-name:	name of this object
(slot)Factor-information:	a set of pointers in which each pointer points to this object's quality factor items.
(method)Get-factor	retrieve a quality factor. Which factor to be retrieved depends on the message this object receives. The message must be one of the thirteen factors.

- Quality Factor Item (QFI)

CLASS QFI

(slot)Quality-factor-name:	name of this object.
(slot)Value:	value for this factor.
(slot)Criterion-information:	a set of pointers in which each pointer points to object's quality criterion items.
(method)Get-and-calculate	get all the criteria for this factor and calculate the factor value.

- Quality Criterion Item (QCI)

CLASS QCI

(slot)Quality-criterion-name:	name of this object.
(slot)Value:	value for this criterion.
(slot)Metric-information:	a set of metric values
(method)Collect-metric	collect the values of the metrics. This method calls AMS to collect metrics.
(method)Get-and-calculate	get all the metrics of the criterion and calculate it.

In the software quality framework[1], each factor or criterion has a formula to calculate its value. The same criterion for different software products will use the same formula to make the calculation. By putting this formula into a class, all the instances of this class can share the same formula. As we mentioned before, the instances of the same class will have the same structure and behavior. The modularity of the data representation will be strongly supported by this design.

Another benefit of this design is that it uses the composite object concept, where a composite object consists of sub-objects. When the Get-factor is invoked, it will automatically invoke Get-and-calculate for each of the sub-objects which are in its Factor-information slot. This kind of granularity design will facilitate the maintenance of the knowledge base.

In the software quality framework [1], the equations for calculating factor values from criteria in different development phases are the same. For example, the reliability factor value is obtained from the average of the accuracy, anomaly and simplicity,

regardless whether it is for the SRA, PD, or DD phase. What happens to the expert system framework, when the software quality framework is changed, such as different equations are used in different development phases? A simple way to do this is to create subclasses of QFI for each development phase. Each subclass will inherit the structure (slots) of QFI and add its own Get-and-calculate (equation calculation) method to the subclass. In the object-oriented approach, methods are overridden in the subclasses if the names of the methods are the same as the names of methods in their superclasses. The objects in the subclass will be calculated using the new equation in the subclass instead of the equation in QFI. With the ability provided in the object-oriented approach, we have a more flexible way to handle future enhancement.

Chapter 5

Quality Assurance Activities with Rules

In Chapter 4, we have discussed how the knowledge in our expert system framework (quality information from the AMS) is represented in the Object-Oriented Data Base. In this chapter, we are going to explore another component of our expert system framework, namely, the Inference Engine. In this chapter, we are going to use logical rules as the inferential knowledge and a set of meta rules as the control portion of the inferential knowledge to show how the quality of a software product can be assured using our knowledge-based system.

The application of logical rules is the most common inference strategy used in knowledge based systems. When A is known to be true and if there is a rule states, "If A then B", then it is valid to conclude that B is true. The logical rules are chosen to be the inference strategy here because they are simple, so reasoning based on them is easily understood. The use of logical rules also makes the change of the inference strategy easy because a user of the system may easily modify the rules available in the inferential knowledge. In Section 5.1, the rules by converting from the quality framework metric elements will be described.

The control portion in the inferential knowledge generally should solve the two basic problems [14]: First, the knowledge based system must have a way to decide where to start. Rules and facts usually reside statically in the knowledge base. The Inference Engine must decide where the reasoning process is to begin. Second, the inference engine must resolve conflicts that occur when alternative lines of reasoning emerge. Therefore, in Section 5.2, we are going to use meta rules as the control portion of the inference engine to control where the system is to start and which rules are selected to be executed.

5.1 Rule Set (RS)

The Rule Set (RS) in this system are stored in the knowledge base. Based on available data, the RS will provide design suggestions, invoke the Dependency-Based Truth Maintenance System (DTMS) to evaluate designs, inform the users of satisfied results, or provide improvement suggestions. Users of this system can add, delete or update the RS according to their needs. The RS will be centrally controlled by the Inference Engine based on the meta rules in the Meta Rules (MR) that are described in the next section.

There are basically the following five categories of rules in the RS:

- Quality Guideline Rules (QGRs):

Quality Guideline Rules (QGRs) are used as quality improvement guidelines. During the software development process (including software requirement analysis phase, preliminary design phase, detailed design phase, coding and CSU testing phase, CSC integration and testing phase, CSCI testing phase, and system integration and testing phase), the design decisions will be guided by the QGRs to assure high quality design. These rules serve as an assistant in making design decisions at each software development phase. For example, before the software requirement analysis phase, how to incorporate the accuracy, anomaly, and simplicity of the requirements document to the reliability factor; before the detailed design phase, how to incorporate completeness, consistency, and traceability of the requirements document to the correctness factor.

- Quality Monitoring Rules (QMRs):

Quality Monitoring Rules (QMRs) are rules for monitoring the quality information for each quality factor/criterion/metric relation. Even if a user's expected quality is achieved, some metric values may still be lower than the expected. The QMRs will send a warning message to the user and the Quality Improvement Rules (QIRs) will be invoked to use the DTMS for quality improvement (the QIRs are described later in this section).

- Automatic Quality Collection Rules (AQCRs):

These rules help program developer to evaluate software quality at key milestones. After all the metric values are collected, the AQCRs invoke factor/criterion calculation program in Class II to compute factor/criterion values. These rules are useful after the metric values are collected automatically. These rules are invoked first by the Inference Engine using some meta rules in the Meta Rules (MR) at each development key milestone. For example, anomaly management should be evaluated at software requirements review, preliminary design review, detailed design review, and unit level during coding review.

- User's Goals Satisfied Rules (UGSRs):

User's Goals Satisfied Rules (UGSRs) work similar to the Quality Monitoring Rules. If the user's expected quality factors are achieved and the design does not have metric values lower than the critical values, the UGSRs inform the user that the goals have been achieved.

- Quality Improvement Rules (QIRs):

When the design at some phases of the development process cycle is detected to have low quality values, the DTMS is invoked for design evaluation and the Quality Improvement Rules will be used for quality improvement.

In this chapter, we are going to elaborate the Quality Guideline Rules (QGRs) to show how these rules are derived and used in our expert system framework. As mentioned in Chapter 2, software quality factors are represented by sets of criteria, each of which is represented by one or more metric elements. Most of the metric elements are evaluated using questionnaire (i.e., checklist) format. There are two basic forms of questions. In one form, the user can answer either "Yes" or "No". This form of questions usually checks the existence of some characteristics that the software product has. The characteristics may be in any phase of the software development cycle.

In another form, a numeric value is expected to be filled. This form of questions comes either as a pair of questions or single question with a numeric value as its answer. For the pair of questions, the quotient of these two values is used toward the final computation of the criterion and factor values. For example, in the anomaly management (AM) questionnaire, there are questions for checking the error-tolerance code such as: a) How many error conditions are required to be recognized? b) How many recognized error conditions require recovery or repair?. The value of $b \text{ div}^* a$ (div^* is either $/$, $\text{div}0$, or $\text{div}1$) is then used toward the quality framework metric equations to compute the anomaly management criterion, which then is used to compute the reliability factor of the software quality. div^* is either $/$, $\text{div}0$ (division, return 0 if divisor is 0), or $\text{div}1$ (division, return 1 if divisor is 0). For those single questions, there are quality framework metric equations defined for calculating the metric values. Those quality framework metric equations use regular mathematical operators like: $+$, $-$, $*$, $/$, $\text{div}0$, and $\text{div}1$. The quality framework metric equation also supports two functions: ABS and AVE. The ABS function returns the absolute value of the single argument and the AVE function returns the average of a list of elements. There are also other forms of questions involving quality framework metric equations, which can be converted into rules accordingly.

Finally, any criterion is computed using the operators, functions and quality framework metric equations as defined in the last paragraph. For those questions with "Yes" or "No" as their answer, if "Yes" is the answer, a numeric value 1 is assigned to the metric, otherwise, 0 is assigned to the metric. All the criteria from a software factor are then assigned the same weight and we will find the factor value by multiplying

the weight with each criterion value (or simply taking the average of all of the values of the criteria to be the value of the factor). In this scheme, the factor values are obtained at the end of each development phase.

One question ("Yes" or "No" question) for each criterion is chosen from the metric element sets to illustrate the software-oriented view of the software product in the next paragraph. In [12], a set of metric questions has been developed, and under each criterion there are questions for each development phase.

Accuracy (AC) Are there quantitative accuracy requirements for all applicable outputs associated with each mission critical system function?

Anomaly (AM) Are there requirements to check all critical output data before final outputting?

Application Independence (AP) Is the unit free from computer architecture references?

Augmentability (AT) Are all variable dimensions and dynamic array sizes defined parametrically for this unit?

Autonomy (AU) Are all processes and functions portioned to be logically complete and self-contained to minimize interface complexity?

Commonality (CL) Are there requirements for communication with other systems?

Completeness (CP) Are the flow of processing (algorithms) and all decision points (conditions and alternate paths) in that flow described for all system functions?

Consistency (CS) Is there a requirement to standardize all design representations? (For example, representing control flow, data flow.)

Distributedness (DI) Is a graphic portrayal (figures, diagrams, tables) provided which identifies all software functions and functional interfaces in the system?

Document Accessibility (DO) Are current versions of all software documentation related to the project, free from access control (i.e. any member of the current project or other projects may access a copy of any document)?

Effectiveness Communication (EC) Have performance requirements and limitations for system communication efficiency been specified for each system function?

Effectiveness Processing (EP) Have performance requirements and limitations for processing efficiency been specified for each system function? (For example, flow time for processes, execution time.)

- Effectiveness Storage (ES)** Have performance requirements and limitations for storing data to efficiently utilize primary and secondary storage been specified for each system function?
- Functional Scope (FS)** Are there requirements to construct functions in such a way to facilitate their use in other similar system applications?
- Generality (GE)** Is each unit free from machine-dependent operations?
- Independence (ID)** Is there a requirement to use a standard subset of the implementation language(s) for the system?
- Modularity (MO)** Are all software functions and CSCIs developed according to structured design techniques?
- Operability (OP)** Have the operating characteristics of the system been specified (i.e. the normal and alternate procedures and actions performed by the system)?
- Reconfigurability (RE)** Are there requirements to ensure communication paths to all remaining nodes / communication links in the event of a failure of one node / link?
- Self Descriptiveness (SD)** Has the specific standard been established that each unit prologue contain the unit's function, author, version number, version date, inputs, outputs, algorithms, assumptions and limitations?
- Simplicity (SI)** Are there diagrams identifying all CSCI functions in a structured fashion? (For example, top-down hierarchical.)
- System Accessibility (SS)** Are there requirements to control user input/output access in the system? (For example, user access is limited by identification and password checking.)
- System Clarity (ST)** Is the unit interface established solely by arguments in the calling sequence parameter list?
- System Compatibility (SY)** Does the design use the same I/O transmission rate as the interoperating system(s), in accordance with the specified requirements?
- Traceability (TC)** Is there a table(s) tracing all of the CSCI's allocated requirements to the parent system or subsystem specification(s)?
- Training (TN)** Are there requirements to provide lesson plans and training materials for operators, end users, and maintainers of the system.
- Virtuality (VR)** Are the system implementation details transparent to the user?
- Visibility (VSOB)** Are all specified performance requirements of the CSCI to be tested?

Some of the questions in [12] requiring numerical values are listed as follows: (For those questions forming a pair, we want to maximize the quotient of the two values; and for those single questions, we want to maximize the value to its answer.)

- a) How many instances of different processes are there which are required to be executed at the same time?
b) How many instances of concurrent processing are required to be centrally controlled?
- a) How many error conditions are required to be recognized?
b) How many recognized error conditions require recovery or repair?
- a) How many identified data items are documented with regard to their source, meaning, and format?
b) How many data items are identified?
- a) How many input parameters can the users "self describe" along with the parameter value?
b) How many total parameters must the user input?

For the above questions, assign the value of $b \div a$ to some metric element; if $b/a < 1$, then "No" is the answer, otherwise "Yes" is the answer. Checklists from the quality factor-criterion-metric element sets in different phases of the software development cycle are converted into rules guiding the design of the software product. There are two basic forms of rules converted from the metrics elements. In general, the two types of questions discussed previously have the following two formats:

1. Are there <functions> [on <components>] at <milestone_X>?
2. a) How many entities are there to perform <function_A>
on <component> ?
b) How many entities are there to perform <function_B>
on <component> ?

The first type of questions are concerned with the existence of certain important considerations during the development of the software product. Each factor contributing to the software quality depends upon a number of quality functions, and hence we would like to have the software product process as many such functions as possible. The second type of questions uses the quotient of the two questions asked

to be the value of the metric element value, and hence we want to maximize the answer of the second question. Both types of rules can be executed in various phases of the software development cycle. In order for those rules to be executed, the user of the system must provide an indication of which milestone is the development process currently at. In this way, different milestones will trigger different types of rules to be executed. They are converted into rules having the following format:

1. If <functions> are not presented [on <components>]
at <milestone_X>,
Then add <functions> to that component.
2. Maximize the entities in a component at <milestone_X> performing
<function_B> while maintaining the entities at <milestone_X> performing
<function_A>

These rules when converted from the metric element questions in [12] will be stored in the knowledge base to provide the software developer guidelines to build high quality software. Some of the Quality Guideline Rules (QGRs) providing guidelines to increase the reliability value of the software being developed during the software requirement analysis phase are given below:

In the accuracy criterion (AC) the questions: "Are there quantitative accuracy requirements for all applicable inputs associated with each mission critical CSCI function?" and "Are there quantitative accuracy requirements for all applicable outputs associated with each mission critical CSCI function?" can be converted into the following rules:

AC1.4b If there are no quantitative accuracy requirements for all applicable inputs associated with each mission critical CSCI function, then add quantitative accuracy requirements for all inputs.

AC1.5b If there are no quantitative accuracy requirements for all applicable outputs associated with each mission critical CSCI function, then add quantitative accuracy requirements for all outputs.

...

In the anomaly criterion (AM) the questions: "In how many instances are different functions allowed to execute at the same time?" and "In how many instances is concurrent processing centrally controlled?" and "How many error conditions are identified?" and "How many identified error conditions are provided with processing instructions for recovery or repair of the error?" can be converted into the following rules:

AM1.1b Maximize the number of concurrent functions that are centrally controlled, while maintaining the number of concurrent functions in the software requirement analysis phase.

AM1.3b Maximize the number of identified error conditions that are provided with processing instructions for recovery or repair of the error, while maintaining the number of error conditions being identified in the software requirement analysis phase.

...

In the simplicity criterion (SI) the questions: "Are there diagrams identifying all CSCI functions in a structured fashion?" and "Are there requirements for a programming standard?" can be converted into the following rules:

SI1.1b If there are no diagrams for identifying all CSCI functions in a structured fashion, then add diagrams in a top-down hierarchical during the software requirement analysis phase.

SI1.9b If there is no programming standard specified during the software requirement analysis phase, then specified some programming standards for the requirements.

...

The notation of **AC1.4b** means this rule is converted from the fourth question related to requirement analysis phase in the Accuracy (AC) criterion 1 as appeared in [12].

5.2 Meta Rules (MR)

As we mentioned at the beginning of this chapter, the control portion of the Inference Engine should decide where to start and resolve conflicts that occur 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 ways in controlling what action is to be taken. In the software quality assurance activities, the goal is to have high-quality software product. The quality (measured by 13 quality factors and 29 criteria) of the software product is measured by metric element sets which have the questionnaire as the metric elements. To control which quality factor/criterion/metric is most important, we will have some meta rules in assisting the developer to make decisions.

A meta rule is the knowledge about knowledge itself. This knowledge at meta-level controls how the quality assurance activities are going to use the knowledge to achieve their goal. In developing software, there are three meta rules in Meta Rules (MR) that will help the developer to choose a rule from the Rule Set (RS) to be applied next. The first kind of meta rule from the MR will be a rule to control the invocation of rules to be activated. This meta rule is used in controlling quality rule invocation at each milestone of the development process. The Rule Set (RS) is used for the system component interactions as explained in Chapter 3. The second meta rule from the MR 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 from the MR is concerned with the temporal ordering among the rules to be chosen. These rules are used to select which quality rule is to be used. They are especially useful for selecting the Quality Guideline Rules (QGRs) described in Section 5.1. The following are the three meta rule formats:

Meta-Rule1:

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

The first meta rule states that at each development milestone (i.e., after the software requirement analysis phase or after the detailed design phase), the Automatic Quality Collection Rules (AQCRs) and the Quality Monitoring Rules (QMRs) should be invoked to collect and evaluate the metric values. The Quality Guideline Rules (QGRs) should be invoked between every two software development phases to provide quality improvement guidelines. If the quality values are lower than what we are expected, the Quality Improvement Rules (QIRs) should be invoked or User's Goals Satisfied Rules (UGSRs) should be invoked to indicate goals achieved.

Meta-Rule2:

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

Meta-Rule3:

When $\langle \text{condition(s)} \rangle$ is true.

Rules which do{not} have $\langle \text{characteristics_X} \rangle$ should be used $\langle \text{order} \rangle$
with $x\%$ certain than

Rules which do{not} have $\langle \text{characteristics_Y} \rangle$.

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

5.3 Examples in the Quality Assurance Activities

Meta-Rule1 is used by the Inference Engine to control which rule in the Rule Set is to be invoked. For example, the Inference Engine will select a rule in the Quality Guideline Rules (QGRs) using Meta-Rule1 to provide design guidelines before each major phase of the software development cycle. After each milestone in the software development cycle, the Inference Engine will select a rule from the Automatic Quality Collection Rules (AQCRs) to collect quality information using Meta-Rule1. The Inference Engine will execute the rules selected from the RS and send a command to the Object-Oriented Data Base to retrieve quality factor information from the AMS. The Inference Engine will also select a rule from the Quality Monitoring Rules (QMRs) using Meta-Rule1 to constantly monitor the current software product quality values using the information stored in the Object-Oriented Data Base. Whenever the system detects a low quality value, the Quality Improvement Rules (QIRs) in the RS will be selected by the Inference Engine to activate the DTMS. Otherwise, the User's Goals Satisfied Rules (UGSRs) in the RS will be selected by the Inference Engine to inform the user that the goals of the software quality have been achieved.

For example, assume that we are at the detailed design phase. After the Automatic Quality Collection Rules and Quality Monitoring Rules are being executed, a message "get.factors" is sent to the CLASS DPI to collect the software quality factor values. Let us also assume that there is a module with the reliability value of 0.75 which is lower than the expected reliability value of 0.95. This low reliability value does not satisfy the user's expected reliability value, and hence a rule in the Quality Improvement Rules is executed to call the Dependency-Based Truth Maintenance System (DTMS) for quality improvement.

In another example, before the software requirement analysis phase, we have decided that one of the most important goals is to develop a reliable requirement specification. Using Meta-Rule2, we can decide which rule in the Quality Guideline Rules (QGRs) is more useful than others for improving the reliability factor. Using Meta-Rule3, we can determine the order of different rules in the Quality Guideline Rules (QGRs) to improve the reliability. Since the reliability depends upon the Anomaly Management, rules in the Quality Guideline Rules increasing the Anomaly Management are important. For example, using Meta-Rule2, we can determine the usefulness of the following rules in the Quality Guideline Rules (the certainty factor is assigned 0.9 indicating that it is very likely the rules will be useful):

Under the condition that we want to increase the reliability,
If there are rules that can increase the Anomaly Management,
Then it is likely (0.9) that each of these rules will be useful.

Using Meta-Rule3, a partial ordering can determine which rule from the Quality Guideline Rules should be applied first. The following rule indicates that when we want to increase the reliability while the Anomaly Management (AM) is high, it is likely (with certainty factor 0.8, assigned by the project manager) that we should apply the rules to increase the Accuracy (AC) first.

Under the conditions that we want to increase the reliability
and that the Anomaly Management (AM) is high,
If (1) there are rules to increase Anomaly Management (AM),
(2) there are rules to increase Accuracy (AC),
Then it is likely (0.8) the second type of rules should
be used before the first type of rules.

Chapter 6

Dependency-Based TMS (DTMS)

In [46], it is shown that the system maintenance activity could be benefited greatly if the knowledge of software development process has been captured and is used later to reason about the consequences of changing conditions or requirements. After the process knowledge for a design is recorded, if there is a need to make some designs that are similar to some existing designs, designers may not have to repeat these similar designs that were made by the original designers (possibly themselves). Designers can just modify these existing designs to match the new requirements.

Early design knowledge can be more useful to the maintainer than the final code, in which the effects of an early design decision may have become difficult to trace [47]. Without such a record, a maintainer may repeat the undocumented mistakes that were made by the original designer (possibly himself) or may undo earlier unrecorded decisions that are not manifest in the code.

6.1 Motivations for Using the DTMS

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 mutually related (based on some design decisions). Design decisions can be represented as *dependency* relations.

There are three kinds of dependency relations. The first is a metric/criterion (or criterion/factor) dependency relation. It exists between a metric and a criterion whose value depends on the value of that metric (or between a criterion and a factor whose value depends on the value of that criterion). For example, the value of the criterion Anomaly Measurement depends on the value of the metric Error Tolerance/Control (the value of the factor Reliability depends on the value of the criterion Anomaly

Measurement). The second is a data/module dependency relation. It exists between a set of data and a module which is needed to manipulate that set of data. For example, a sorting procedure is needed only when there is data that needs to be sorted; that is, the existence of a sorting procedure in a program depends on the unsorted data. The third is a phase/phase dependency relation which only occurs between two modules which are in adjacent software development cycle phases. For example, the sorting procedure in the detailed design phase depends on the sorting procedure in the preliminary design phase. Development for the first dependency relation will be discussed in this report, the other two dependency relations will be discussed in the future.

The dependency relations are all 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. In order 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 [21]. In our expert system, assumptions can be users' design decisions or users' expected quality factor values.

This hypothetical reasoning can be handled in a variety of ways. The first approach is referred to as viewpoints [48], contexts, and worlds in different tools. This approach reduces the difficulty of the computation by carrying along multiple solutions representing different hypotheses in parallel. This approach will also discard inappropriate hypotheses when they contradict the facts. Difficulty for this approach is that it is not possible to have all different hypotheses (design decisions) tried for a piece of software. Some software quality factors are mutually conflicting, and they may not be improved at the same time. Another approach has been referred to by a name like 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, which removes derived beliefs when their conditions are no longer true. Two primary features of the TMS—persistent relations representation and dependency-directed backtracking—make it possible to represent and update the persistent dependency relations of metric and criterion (or criterion and factor). Hence, the truth maintenance system is chosen for our expert system.

6.2 Definitions

The data objects of our expert system are mutually 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 results ripple throughout the logical structure of the system. Such values are called **active values**.

Definition 1. An *active object* is an object whose value attribute affects another object's value attribute.

Definition 2. A *passive object* is an object whose value attribute is affected by another object's value attribute.

In our expert system, 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 expert system may be a passive object or an active object or both.

The organization of objects of our expert system can be viewed as an abstraction mechanism. Those abstraction mechanisms that are considered in this subsystem 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. However, the PART-OF relationships of our expert system are not quite the same as the traditional ones, e.g. fingers PART-OF hands. Here, the A PART-OF B means not only A is part of B, but also B's value is derived from A. For example, Anomaly Measurement PART-OF Reliability.

Thus, the other way of describing data objects of our expert system is that they are bound together by either the IS-A relationship or the PART-OF relationship. In this subsystem, the IS-A relationships exist between classes and subclasses. The PART-OF relationships are used to represent the hierarchical relationships of software quality factors.

Objects of our expert system have the *inheritance* feature. Therefore, when an object is an instance of a class it inherits attributes from it. This object inherits values from a class only when it has the same attributes as its class. In our expert system, however, when an object (a component object) is PART-OF another object (the parent object), the component object does not inherit attribute values from that parent object. On the contrary, attribute values of parent object are derived from the component objects. For example, A and B are both objects, and A PART-OF B. A does not inherit from B. On the contrary, some of B's attribute values are derived from A.

Definition 3. A *Dependency-Based Truth Maintenance System (DTMS)* is a system which does dependency-directed backtracking and incrementally updates its beliefs when dependency relations are modified.

Definition 4. A *culprit* represents a metric that lowers a quality factor value of the software.

Definition 5. 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 our expert system is applied, a culprit will be found. Our expert system can be iteratively applied to find culprits for a software quality factor or factors.

6.3 DTMS Approach to Efficient Search

A simple example is used to show the efficient search that applied by the DTMS.

Let $x, y,$ and $z \in \{1,2\}$. $a = \log(x)$, $b = \log(y)$, $c = \log(z)$, $a \neq b$, and $b \neq c$. Find valid solutions for this question.

The simplest search strategy (brute force) is exponential: Enumerate all the possibilities and try each one until a solution is found or all solutions are found. In this example, there are 3 binary selections giving tentative solutions to test. ('YES' indicates a valid solution and 'NO' indicates a contradiction). As each solution requires 3 computations, there are totally 24 computations required.

$x = 1,$	$y = 1,$	$z = 1, \dots\dots$	NO
$x = 1,$	$y = 1,$	$z = 2, \dots\dots$	NO
$x = 1,$	$y = 2,$	$z = 1, \dots\dots$	YES
$x = 1,$	$y = 2,$	$z = 2, \dots\dots$	NO
$x = 2,$	$y = 1,$	$z = 1, \dots\dots$	NO
$x = 2,$	$y = 1,$	$z = 2, \dots\dots$	YES
$x = 2,$	$y = 2,$	$z = 1, \dots\dots$	NO
$x = 2,$	$y = 2,$	$z = 2, \dots\dots$	NO

The search space for the brute force search is:

- (1) $x = 1$
- (2) $x = 1, \quad y = 1$
- (3) $x = 1, \quad y = 1, \quad z = 1, \dots\dots$ NO
- (4) $x = 1$
- (5) $x = 1, \quad y = 1$
- (6) $x = 1, \quad y = 1, \quad z = 2, \dots\dots$ NO
- (7) $x = 1$
- (8) $x = 1, \quad y = 2$
- (9) $x = 1, \quad y = 2, \quad z = 1, \dots\dots$ YES
- (10) $x = 1$
- (11) $x = 1, \quad y = 2$
- (12) $x = 1, \quad y = 2, \quad z = 2, \dots\dots$ NO
- (13) $x = 2$
- (14) $x = 2, \quad y = 1$
- (15) $x = 2, \quad y = 1, \quad z = 1, \dots\dots$ NO
- (16) $x = 2$
- (17) $x = 2, \quad y = 1$
- (18) $x = 2, \quad y = 1, \quad z = 2, \dots\dots$ YES
- (19) $x = 2$
- (20) $x = 2, \quad y = 2$
- (21) $x = 2, \quad y = 2, \quad z = 1, \dots\dots$ NO
- (22) $x = 2$
- (23) $x = 2, \quad y = 2$
- (24) $x = 2, \quad y = 2, \quad z = 2, \dots\dots$ NO

Comparing with the brute force search, chronological backtracking requires the additional machinery of a stack of variable bindings, but has better efficiency. The expression order of this search is presented below. If the expression is a selection, then try the first one; otherwise, evaluate the equation. If the expression is inconsistent, back up to the most recent selection with a remaining alternative and resume processing expressions from that point [36]. In this example, the search space of assumptions is:

- (1) $x = 1$
- (2) $x = 1, y = 1$
- (3) $x = 1, y = 1, z = 1 \dots\dots$ NO
- (4) $x = 1, y = 1, z = 2 \dots\dots$ NO
- (5) $x = 1, y = 2$
- (6) $x = 1, y = 2, z = 1 \dots\dots$ YES
- (7) $x = 1, y = 2, z = 2 \dots\dots$ NO
- (8) $x = 2$
- (9) $x = 2, y = 1$
- (10) $x = 2, y = 1, z = 1 \dots\dots$ NO
- (11) $x = 2, y = 1, z = 2 \dots\dots$ YES
- (12) $x = 2, y = 2$
- (13) $x = 2, y = 2, z = 1 \dots\dots$ NO
- (14) $x = 2, y = 2, z = 2 \dots\dots$ NO

The extra machinery (a stack) for controlling the search is well worth it because only 14 computations are required, while the brute force technique requires 24 computations. The chronological backtracking is the central control mechanism of PROLOG.

By checking the search space of the chronological backtracking, 8 out of the 14 computations are easily avoided:

Futile backtracking: Steps 4 and 14 are futile. When a contradiction is discovered, the search should backtrack to an assumption which contributes to the contradiction, not to the most recent assumption made. For the contradiction found in Step 3 (or 13), the contradictions depend on the values of x and y . The selection of $z \in \{1, 2\}$ has no effect on the contradiction in Step 3 (or 13), so Steps 4 (or 14) can be ignored.

Rediscovering contradictions: Step 10 is futile. When Step 3 is contradicted, the backtracking strategy should have been determined that the contradiction depended on $y = 1$ and $z = 1$. Therefore, Steps 10, which has the same values for y and z should never have been tried.

Retrying inferences: The function computations of Steps 6, 7, and 9 to 14 are unnecessary. Chronological backtracking erased the earlier computations, but if the previous results have been remembered, each function computation would only need to be computed once.

The dependency-directed backtracking [32] is a solution to avoid the above three defects because it maintains records of the dependency of each inference on earlier ones and records the reasons for contradictions (the no-good sets). When it encounters

a contradiction, it consults these dependency records to determine which selection to backtrack to and records the no-good sets. Consider Step 3 as an example. When the contradiction happens, the dependency records indicate that $(x = 1, y = 1)$ and $(y = 1, z = 1)$ contribute to the contradiction. Then the system can backtrack to the most recent selection which actually contributed the contradiction.

Dependency records are bidirectional, linking antecedents to consequents as well as consequents to antecedents. Thus, the problem of retrying inference is also avoided. Therefore, if $x = 1$ is believed, $a = \log(x)$ is also believed. Whenever some assumption is included in the current set, the dependency records are consulted to reassure the previously derived consequents of that assumption. This can be done by storing them in a data base. If they are not derivable from the current set of assumptions, they can be marked as temporarily disbelieved. Besides, before any new assumption set is added, it is checked to see whether it contains any known contradictions or not.

These techniques are the basis of all the TMSs. A scheduling strategy for the problem-solver-TMS has been proposed in [49]. The scheduling strategy cannot only solve the three defects, but also eliminate incorrect ordering problem. Therefore, it can avoid Step 3 as well, which cannot be avoided by using the dependency-directed backtracking. However, all possible assumptions need to be known before this scheduling strategy is applied. This is not quite possible in software design.

It is important to note that all these strategies are ultimately equivalent. They will find as many consistent solutions as pure enumeration. The purpose of all these strategies is to improve efficiency without sacrificing completeness.

6.4 DTMS Algorithm

The purpose of the DTMS is to find the culprit(s), i.e. metric(s) that lowers the quality of the evaluated design. The mechanism that the DTMS applies to find the culprit(s) is dependency-directed backtracking.

Algorithm:

- Step 1. *Target Selection*: If there are two or more quality factors that did not achieve the expected standard, the user should pick up the target, one factor, according to his preferences or design purposes, to ask for software quality improvement suggestions.
- Step 2. *Invocation*: The DTMS is invoked by the Inference Engine for verifying and improving the software quality.

Step 3. *Dependency-Directed Backtracking and Culprit Finding*: The DTMS invokes dependency-directed backtracking to find the culprit for the chosen quality factor.

Step 4. *Improvement Suggestions Passing*: Signal and pass software quality improvement suggestions to the Inference Engine.

Step 5. *Users Actions*: Users check quality checklists and correct errors. Then the system quality is remeasured.

Step 6. *Exit*: Exit conditions of our expert system are 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 factors values of two adjacent versions of 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 our expert system 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 3.

The DTMS can be iteratively called by the Inference Engine. Each time the DTMS is called, it invokes the dependency-directed backtracking and makes a design improvement suggestion. Due to the potential contradictions inherent in some of the software quality metrics, a software improvement suggestion may increase one quality factor and decrease another quality factor at the same time. Fortunately, those mutually conflicting factors are not absolutely conflicting (the amount of changes of the increased quality factor is not as large as that of the decreased quality factor). By iteratively choosing different quality factors as targets, the DTMS can help balance conflicting factors of the software.

Chapter 7

An Integrated Example

In this chapter, we are going to give an example to illustrate how the expert system would work based on our framework, which includes the Object-Oriented Data Base, Inference Engine (consisting of the Meta-Rules and Rule Set), and the DTMS. This example is to develop a software system to solve a set of linear system equations using the Gaussian Elimination Method (GEM). In this example (GEM), only the reliability quality factor is considered. The user's expected reliability factor values during the SRA (Software Requirement Analysis phase), DD (Detailed Design phase), and CUT (Coding and CSU Testing phase) are all assumed to be 0.95. We are going to show how the Quality Guideline Rules (QGRs) are used to provide design guidelines, how the quality information is stored in the Object-Oriented Data Base, and how the DTMS can find the culprits in the metric elements and provide improvement suggestions.

Before the software requirement analysis phase, the Quality Guideline Rules (QGRs) will be used to provide guidelines in writing the requirement specification for GEM. Since only the reliability factor is used, the metric questions for the reliability factor during each development phase are given in Appendix A. Here, we will exercise all the QGRs in the Accuracy (AC), Anomaly (AM), and Simplicity (SI) criteria as described in Chapter 5. Those QGRs, when used, provide guidelines to improve the reliability (of the requirement document in the software requirement analysis phase) that can be summarized as follows [12]:

- Accuracy Criterion (AC)
 - AC1 (Accuracy) – There are statements in the requirement document, which provide quantitative accuracy requirements for all applicable inputs, outputs, and constants associated with each mission critical function. Existing mathematic library routines shall be planned for use in the CSCI, which will provide enough precision to support accuracy objectives.

- Anomaly Criterion (AM)

- AM1 (Error Tolerance/Control) - All concurrent processing should be centrally controlled. All identified error conditions must provide processing instructions for recovery or repair. Standard error handling routines shall be provided such that all error conditions are passed to the calling function or some software element. All instances of parallel/redundant processing should be centrally controlled.
- AM2 (Improper Input Data) - All error tolerances should be specified for all applicable external input data to the CSCI.
- AM3 (Computational Failures) - There should be requirements for detecting (or recovering from) all computation failures, for range-testing all loop and multiple transfer index parameters, and subscript values before use, and for checking all critical outputs to verify that they are reasonable before final outputting.

- Simplicity Criterion (SI)

- SI1 (Design Structure) - There should provide diagrams identifying all CSCI functions in a structured fashion. Programming standard should be established and used.
- SI2 (Structured Language or Preprocessor) - There should be requirements to use a structured language or preprocessor for implementation.

The requirement document written following the guidelines provided by the QGRs is given 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 by using the data in Table 7.1 are given as follows:

$$Accuracy(AC) = AVE(AC1) = 1.00$$

AC		AM		SI	
AC1	1.00	AM1	0.50	SI1	0.67
		AM2	1.00	SI2	1.00
		AM3	0.75	SI3	N/A
		AM4	N/A	SI4	N/A
		AM5	N/A	SI5	N/A
		AM6	N/A	SI6	N/A
		AM7	N/A		

Table 7.1: The metric values for the GEM example in the Software Requirement Analysis Phase.

$$Anomaly(AM) = AVE(AM1, AM2, AM3, AM4, AM5, AM6, AM7) = 0.75$$

$$Simplicity(SI) = AVE(SI1, SI2, SI3, SI4, SI5, SI6) = 0.835$$

Finally, we calculate the reliability value (RL) after the software requirement analysis phase as follows:

$$RL = AVE(AC, AM, SI) = 0.86$$

Now, we are going to show how the quality information is stored in the Object-Oriented Data Base. Figure 7.1 is the quality information hierarchy for object GEM. Figure 7.2 shows the obtained reliability quality information.

There are many CSCI level items for a large scale software product, hence there are many objects in the Class II. 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↑, GEM_PD↑, ...)

The software project GEM has to flow through the software development cycle. Figure 7.1 shows there are six objects in the Class DPI: GEM_SRA, GEM_PD,

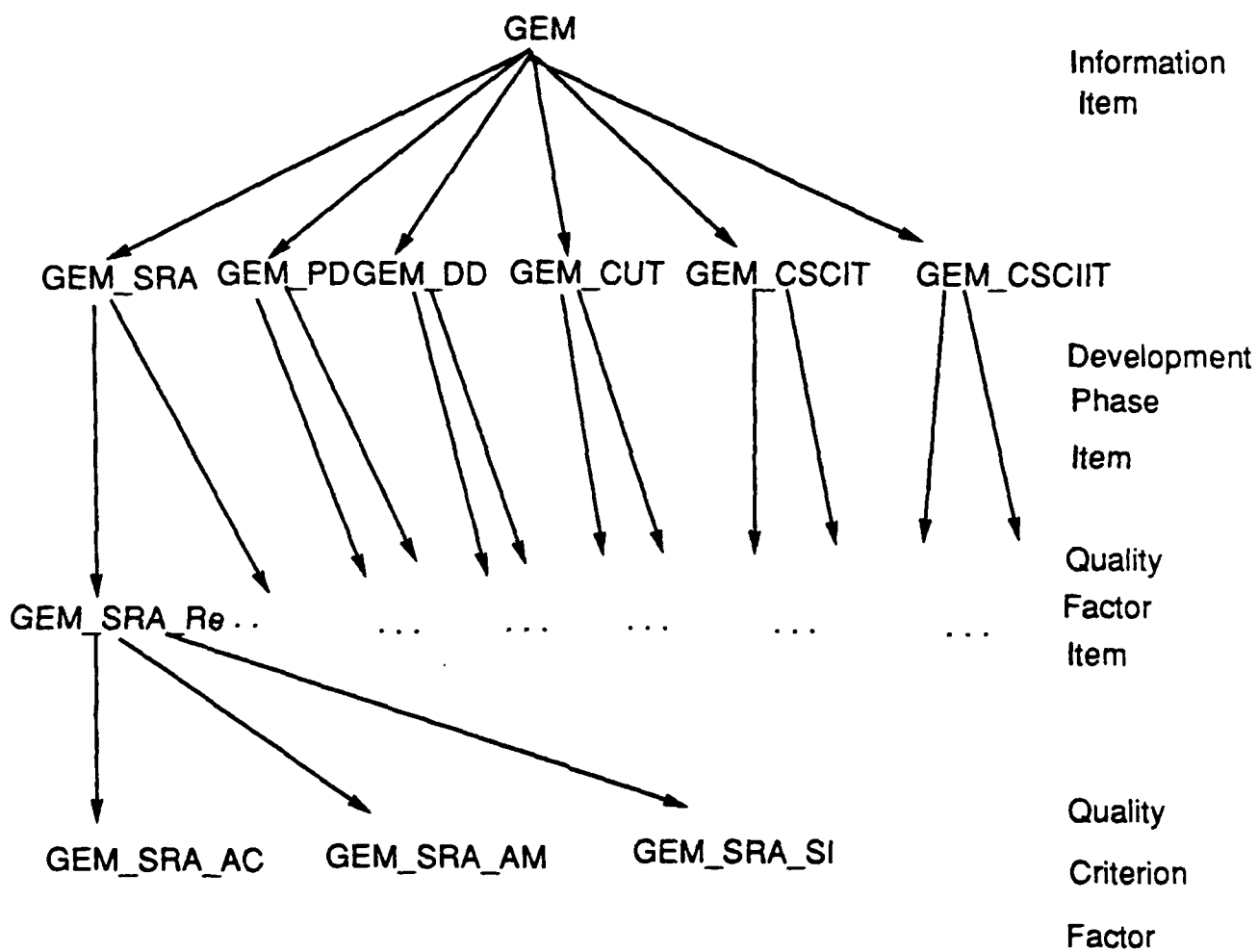


Figure 7.1: The object hierarchy for the GEM example.

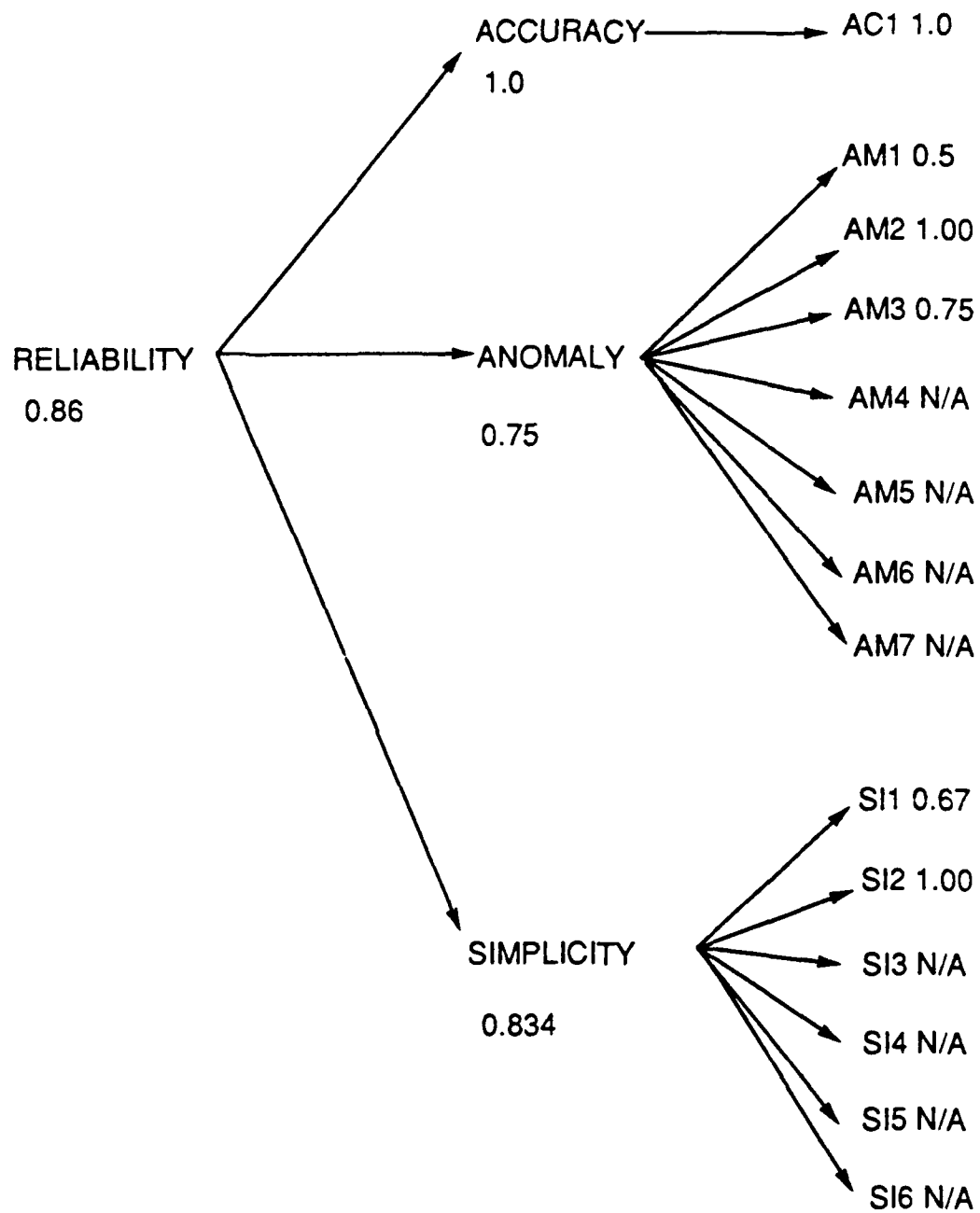


Figure 7.2: The quality values for the GEM example in SRA Phase.

GEM_DD, GEM_CUT, GEM_CSCIT and GEM_CSCIIT. These objects are all sub-objects of the object GEM. The following object represents the GEM_SRA which is the object for the Software Requirement Analysis phase.

CLASS DPI

object

Development-phase-name: GEM_SRA

Factor-information: (GEM_SRA_Re↑, GEM_SRA_Ef↑,...)

In the Software Requirement Analysis phase, all the software quality factors should be evaluated based on the requirement specification. In this example, because only the reliability factor is considered, 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↑, GEM_SRA_AM↑,
GEM_SRA_SI↑)

There are three objects in the Class QCF for the software project GEM in the Software Requirement Analysis phase: GEM_SRA_AC, GEM_SRA_AM, and GEM_SRA_SI. Each of the three objects represents a software criterion for the reliability factor. They are shown as follows:

CLASS QCI

object

Quality-criterion-name: GEM_SRA_AC

Value: 1.0

Metric-information: (1.0)

CLASS QCI

object

Quality-criterion-name: GEM_SRA_AM

Value: 0.75

Metric-information: (0.5, 1.00, 0.75, NA, NA, NA, NA)

CLASS QCI

object

Quality-criterion-name: GEM_SRA_SI

Value: 0.834

Metric-information: (0.67, 1.00, NA, NA, NA, NA)

The three objects described above are stored in the Object-Oriented Data Base. Object constructions for the software project GEM in the Detailed Design phase (DD)

and the Coding and CSU Testing phase (CUT) are described in Appendix B.

Now we are going to show how the DTMS works. To make the example easier to understand, unrelated object details are skipped and the dependency-directed backtracking is represented by the $\Rightarrow$ sign.

Operation flows of the DTMS is as follows:

Step 0. Based on the quality data of the example, data objects hierarchy for object GEM are constructed.

Step 1. Since the Reliability factor does not achieve the expected standard, it is chosen as the target.

Step 2. DTMS is invoked for verifying and improving software quality.

Step 3. Dependency-Directed Backtracking is invoked to find the culprit, and the Dependency-Directed Backtracking path is
From (GEM, ..., GEM_SRA, GEM_PD, ...)
 $\Rightarrow$ (GEM_SRA, ..., GEM_SRA_Re $\uparrow$)
 $\Rightarrow$ (GEM_SRA_Re, 0.86, GEM_SRA_AC $\uparrow$, GEM_SRA_AM $\uparrow$, GEM_SRA_SI $\uparrow$)
 $\Rightarrow$ (GEM_SRA_AM, 0.75, 0.5, 1.0, 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 picked as the culprit, and the dependency-directed backtracking paths are shown in Figure 7.3.

Step 4. 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 5. The user checks the Error Tolerance/Control metric checklists, and find that 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 repair 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 6. After our expert system compares the new quality values with the expected quality values, our expert system finds that the new quality values are still smaller than the expected quality values. The operation goes to Step 3.

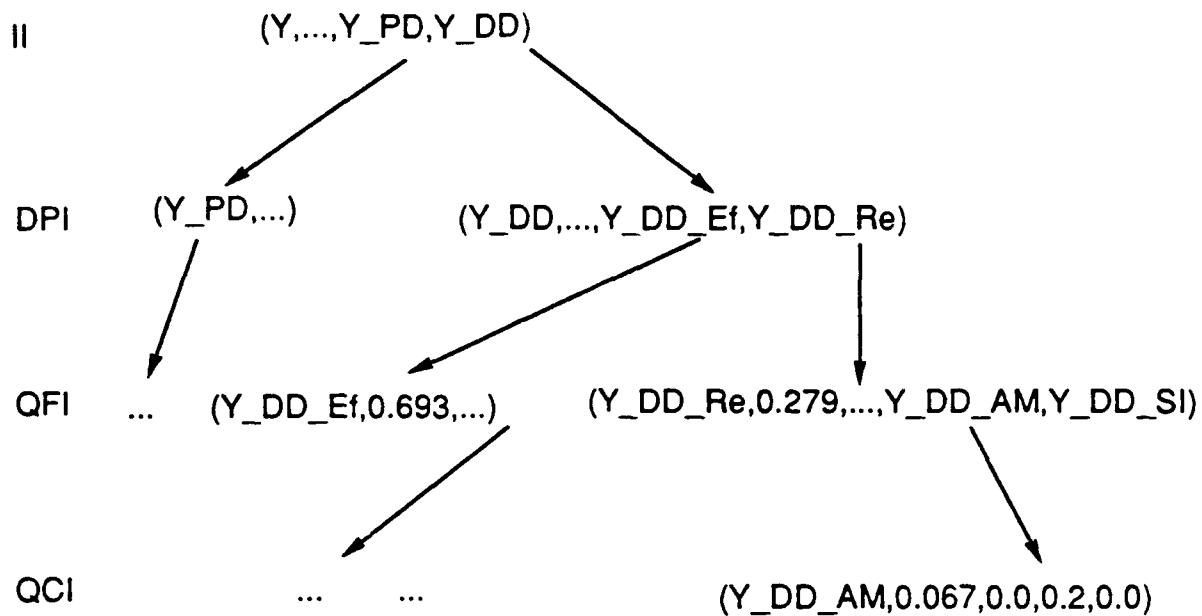


Figure 7.3: The dependency-directed backtracking paths of the GEM example.

The second invocation of the 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 real Reliability factor value is greater than the expected value (0.95), the DTMS stops.

After the DTMS, the requirements were modified to include a standard for handling errors such that all error conditions are passed to the calling function element, and to include diagrams for identifying all CSCI functions in a structured fashion. The newly revised requirement document is given 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 error conditions will be*

AC		AM		SI	
AC1	1.00	AM1	1.00	SI1	1.00
		AM2	1.00	SI2	1.00
		AM3	0.75	SI3	N/A
		AM4	N/A	SI4	N/A
		AM5	N/A	SI5	N/A
		AM6	N/A	SI6	N/A
		AM7	N/A		

Table 7.2: The revised metric values for the GEM example in the Software Requirement Analysis Phase.

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 7.2 are given as follows:

$$Accuracy(AC) = AVE(AC1) = 1.00$$

$$Anomaly(AM) = AVE(AM1, AM2, AM3, AM4, AM5, AM6, AM7) = 0.92$$

$$Simplicity(SI) = AVE(SI1, SI2, SI3, SI4, SI5, SI6) = 1.00$$

The newly revised reliability (RL) after the software requirement analysis phase:

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

After the software requirement analysis phase, the same process for developing quality software program will continue to the later phases of the software development cycle. In Appendix B, the detailed design and the code for GEM will be implemented using a similar method as described in this chapter.

Chapter 8

Development Issues

8.1 Expert System Building Tool Selection

The minimum capabilities of an expert system building tool for software quality assurance activities are to represent facts and rules, and do inferences. The rule capacities of an expert system building tool should be considered for various scales of software projects. Hybrid expert system building tools are suggested to be used for a large scale software project. For a small scale software project, any tools mentioned in Chapter 2 can be used. The object-oriented approach is not the only representation for storing information of the software quality assurance activities for building expert systems. The object-oriented approach is selected because of its efficiency to retrieve information and capability to integrate with all the information in the software development environment. The DTMS is recommended for its efficiency in backtracking. Without applying the DTMS, many rules need to be added. It is easier to maintain rules and facts in the expert system using the Object-Oriented Data Base and the DTMS.

There are two methods to implement our expert system framework. One method is to use high level languages, especially AI languages, like LISP, PROLOG, etc. The other method is to use current expert system building tools. Using high level language to implement this framework, every component of this framework (Object-Oriented Data Base, Rule Set, Meta Rules, and TMS) needs to be implemented by the developers. An expert system building tool is recommended for implementing this framework for its multiple features. A suitable expert system building tool should be able to supply the following capabilities:

- Rules and meta rules inference engine.
- Object-oriented knowledge representation.

- Dependency-directed backtracking.

Since our expert system framework is designed to provide assistance throughout the software development cycle, more than 500 rules will be created. Due to the large number of rules created, the mechanism of knowledge representation chosen, and the inferential knowledge used in our expert system framework, hybrid expert system building tools are more appropriate than others. Three widely used hybrid expert system building tools, ART, KEE, and Knowledge Craft, will be compared here [15-16,22,38]. Although a hybrid expert system building tool may have a variety of advantages and disadvantages; only those advantages and disadvantages related to our software quality assurance activities are discussed.

ART:

- Advantages:
 - ART is the most powerful tool for doing rule and meta rule inference.
 - ART's approach to truth maintenance and viewpoints will be very useful to establish logical dependencies to update the system dynamically as facts change.
 - ART has the fastest execution time.
- Disadvantages:
 - ART keeps its knowledge primarily in rules. But in the software development process, not every problem can be conceptualized in terms of rules.
 - Maintenance can become a significant problem as the number of rules increase. Significant maintenance problems begin when systems have over 2,000 rules.
 - ART does not provide the best knowledge interface and lacks good graphic editing facilities.

KEE:

- Advantages:
 - KEE has the best knowledge interface environment with superior graphic editing facilities and good on-screen menus.
 - Active values support data-directed reasoning and allow the system to recognize and monitor changing conditions.
 - Object-oriented programming allows convenient modularization of the expert system.

- Disadvantages:

- KEE has system-defined inheritance and will not allow users to tailor inheritance for special situation.
- KEE does not use complete backward chaining.

Knowledge Craft:

- Advantages:

- Knowledge Craft has the most powerful schema representation language. It offers such features as dynamic inheritance, meta-information, user-defined inheritance search patterns, and user-defined dependency relationships.
- Knowledge Craft's context mechanism allows users to create multiple hypotheses systems.
- Knowledge Craft's agenda mechanism allows users to tailor knowledge base processing to various applications.
- Object-oriented programming permits the conceptualization of problems in terms of objects and relationships.

- Disadvantages:

- Knowledge Craft's components integration is poor.
- Knowledge Craft lacks good interface.
- Knowledge Craft lacks truth maintenance.

We recommend that KEE be used as the expert system building tool for implementing our Software Quality Assurance Expert System Framework (SQAESF). KEE provides all the foundation that SQAESF needs, an inference engine, rules and meta rules representation, object-oriented knowledge representation, and dependency-directed backtracking. KEE allows users to develop prototypes more rapidly than other expert system building tools and is available for all varieties of hardware: mini/mainframe, Sun, PC, etc.

8.2 Application Scope for SQAESF

Our Software Quality Assurance Expert System Framework (SQAESF) is not limited to only the reliability factor in the software quality framework. In fact, the way we store the software quality factors in the Object-Oriented Data Base and use the Dependency-Based Truth Maintenance System to find the culprits has nothing to do with which factors we are choosing from the software quality framework. The

only difference is in the rules, where we should convert metric elements for all quality factors into rules and consider more meta rules for controlling the usage of those rules.

When all the software quality factors are considered, their metric elements will be converted into rules for quality assurance activities. They can all be combined according to the parent-child hierarchy relationship [12] to determine the final software quality. In this parent-child hierarchy relationship, each software factor is considered as the child of the software quality parent. By default, the children of the parent software quality are given the same weight (i.e., since there are 13 factors, each factor is assigned a weight of $1/13$). The user of the system can change the weights of different factors to reflect the various degrees of importance of individual factors to their applications. Thus, this scheme can provide an overall view of the software quality.

There are some limitations to this expert system framework. First, it cannot solve the ripple phenomenon. There are such cases that a software quality factor has achieved the user's expected value, but some of its criteria values are not high enough. In this case, this software quality factor will pass the design evaluation review, but it will cause the problem (low criteria values) to be carried over to the next phase of the software development cycle. Second, this framework cannot automatically correct the design problems, which can be solved by automatic programming [51]. Whenever a culprit is found, the framework only gives design improvement suggestions. The users still need to change the design manually according to the suggestions provided by the expert system framework.

Chapter 9

Conclusions and Future Work

In this project, a knowledge based quality assurance system framework (Software Quality Assurance Expert System Framework) has been presented. It can be integrated with the information available from the Software Quality Measurement Framework, especially the AMS, for quality assurance. This expert system 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. Software quality assurance activities can be much improved with the use of knowledge based systems.

In the following sections, we will summary our results and discuss what need to be developed further.

9.1 Summary of the Results

In this project, we have shown that knowledge based systems can greatly enforce software quality assurance. We have presented a knowledge based system framework for software quality assurance, which is highlighted as follows:

- Our knowledge based quality assurance system framework is an integrated system which uses the information from the software quality measurement framework [1] and the knowledge based system features. First, the quality information from the software quality measurement framework is stored as objects in the Object-Oriented Data Base. Second, the quality metric elements served as a guideline are converted into rules. The control portion of the Inference Engine is controlled using meta rules for rule selection and justification in case multiple

choices of reasoning emerge. With rules in the RS in this framework, it can express the knowledge that traditional data base systems cannot.

- **Object-Oriented Data Base:** All the quality information and their calculation formulas are stored in the Object-Oriented Data Base. The encapsulation of quality information and calculation formulas makes our system modular. The Inference Engine can send messages to the Object-Oriented Data Base to retrieve the information. The Inference Engine can also send messages to the Object-Oriented Data Base to invoke formula calculations. With the message passing facilities, the Object-Oriented Data Base can be integrated with the Inference Engine. By supporting attachment capability of procedures, the Object-Oriented Data Base can easily be integrated with Dependency-Based Truth Maintenance System. All these features are supported by the object-oriented data base. Traditional Entity-Relationship model or Relational Data Model does not support these features. By creating new classes or subclasses, the object-oriented data base provides flexibility for modifying the structure of quality information. The user can easily add quality information or modify the structure of quality information.
- **Rule Set (RS) and Meta Rule (MR) as the inferential knowledge:** There are two major components in an inference engine, namely the inference and control. In our expert system, we use the logical rules as our inference part of the inference engine. The logical rules have the **If-Then-Else** structure, and are converted from the software quality metric elements. Instead of bonly using the metric elements to evaluate the design of a software system, those metric elements are converted into a design guideline to help the user of our expert system improve the software quality. The second part of the inference engine, the control portion, takes care of the problems such as how to start the system and how to resolve the conflicts when there are many different lines of reasoning. This control portion is implemented using a set of meta rules. Meta rules are the rules about rules themselves, and they are used for rule selection and rule justification when there are choices of alternatives.
- **Dependency-Based Truth Maintenance System (DTMS):** Data objects which are constructed and stored in the Object-Oriented Data Base are used as inputs of our system. The outputs of our system are design improvement suggestions. The search mechanism applied here, which efficiently retrieves facts and rules, is the dependency-directed backtracking. The DTMS communicates with the user through the Inference Engine. The Inference Engine invokes the DTMS whenever any software quality factors of the evaluated design do not achieve the user's expected values. The DTMS can be iteratively called. Each time when it is called, a culprit is found and a design improvement suggestion is made. It stops when the goal is achieved or no significant improvement can be made.

The integration of our knowledge base with the software quality framework [1] provides a wide range of supports 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. Some of the advantages and contributions to the software engineering community are discussed in the following paragraphs.

A knowledge based system is designed instead of using traditional procedural packages. In the conventional approaches, knowledge about problems and procedures for manipulating that knowledge to solve the problems is mixed together. When developers want to change their development environment, they should ask other people to modify the software that constitute the development environment. In designing a software quality assurance system, we separate knowledge from inference and control. All the quality information knowledge is put in the Object-Oriented Data Base. All the situation-action knowledge is put in the Rule Set (RS). Software quality engineers can easily modify their own quality information structure 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. Good quality software can be easily developed with the help of our system.

The software quality assurance activity is a comprehensive view of the whole software development process rather than a restrictive one. In other words, the SQA is not restricted to the function of a software quality group, but rather it includes all necessary activities that may contribute to the quality of software during the entire software development cycle of the product. 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 any application domain. The developers who use our system can use the knowledge from the Object-Oriented Data Base and the Rule Set for quality design improvement. In the case of low quality design, the Dependency-Based Truth Maintenance System can provide design suggestions. Our system shall play the most important role in monitoring software assurance activities and providing suggestions for improvement to the software developers.

9.2 Future Work

Besides the implementation of our system, the following issues need further investigation:

- **Model Refinement:** In this project, we use the software quality framework [1] to build an expert system framework for software quality assurance. In this framework, regression analysis is used to determine the relationship between factors and criteria. After the regression analysis, the relationship becomes a mathematical form. The mathematical form provides a deterministic relationship that is static and cannot be changed no matter what kind of activities we are performing on the software product. There are no relationships between development phases. We should provide a way to modify the existing framework to support probability reasoning that can change the relationships between factors and criteria and should provide a way for reasoning between each phase of the software development process. The probability relationships are closer to dealing with the real world problem than the deterministic relationship. For example, if we change the software product in phase A, then to some extent this change will affect another development phase, say phase B. The same conditions exist between two factors in consecutive development phases and two different factors in the same development phase.

Before the requirement specification phase, the expected quality factors for each development item are specified. At each development milestone, the values of quality factors are evaluated and compared with the expected values. In our current scheme, values of the evaluated quality factors must be higher than or equal to the expected values in every milestone, which basically is not concerned by project managers. What the project managers consider is the quality of the final products, not necessarily the quality of intermediate products, although a deficiency in the early phase of the software development cycle will ripple through the software development cycle and affect the quality of the final product.

Under the probability reasoning model, the ripple phenomenon can be represented by rules. These rules are used to describe the relationships between items in different development phases. When the expected values for quality factors are specified, but the real values for quality factors are unknown, it is impossible to fire the ripple phenomenon rules to predict the quality of the product. One possible solution is to use the assumption-based truth maintenance system (ATMS) [36] because the ATMS allows multiple assumptions. Parallel reasoning is used to detect any wrong assumptions about the quality factors in every phase. When the development proceeds, the real quality factors are obtained and the ATMS can tell which quality factors in the next phase should be achieved if the quality factors in the current phase do not meet the expected quality factors.

- **Build the Data/Module Dependency Relation on the DTMS:** The data/module dependency relation exists between a set of data and a module which is needed to manipulate that set of data. Recording the data/module dependency relation can help simplify the maintenance activities. After the data/module dependency relation is recorded, using the dependency-directed backtracking any change of the data can be propagated to its related modules. Furthermore, if there is a need to manipulate some data that has been processed before, its

related modules can be retrieved by tracing the data/module dependency relations. That is, the dependency relation can be used to establish a software development library [7] to improve software reusability.

- Build the Phase/Phase Dependency Relation on the DTMS: The phase/phase dependency relation only exists between two modules which are in adjacent phases of software development cycle. It happens when a module in the later phase is derived from a module in the former phase. After the phase/phase dependency relation is established, using the dependency-directed backtracking (forward chaining) any change of the former (later) phase can be propagated to its related modules in the later (former) phase.
- Extend the Object-Oriented Representation Contents: In the Software Life Cycle Development Environment (SLCSE), not only software quality information, but also other information such as programs/documents, internal forms of programs, and test data, are important. These data items are described in DOD-STD-2167A [6]. Relational data bases do not support representation of variable-length of data, operators for manipulating flow graph, and operators for test data. A good representation scheme should cover all information described above. The object-oriented approach does not have these limitations. The variable-length of data can be stored in a slot of an object. With methods defined in each class, operators for different types of data can be defined. Object-oriented approach seems to be the best choice for a unified representation. This representation can help integrate the software development environment with expert system technologies. The object-oriented representation which covers all the information in the software development cycle should be further investigated.
- Extending the concept of the software quality assurance activities as a planning process to improve the software quality. Because software quality assurance is defined as "planned and systematic pattern of all actions necessary to provide adequate confidence that the software conforms to establish technical requirement" [4], we can treat the software quality assurance activities as a planning process with a well defined goal (high software quality value). In this planning process, a plan for each development phase handling the development ordering in computer software units (CSU), computer software components (CSC), and computer software configuration items (CSCI) should be considered. The planning knowledge will be derived from both the software developers and software tools available in the development environment. This development plan will be useful when developing a large scale software system. Since there will be many software components and software units, the order of the development will play an important role in the entire software system. Which component or unit should be developed (for example, in a top-down or bottom-up fashion) first will affect the quality for the final software product. This plan can be assisted with the meta rules described in Chapter 5, and is subjected for replan in the case of situation changes such as conflicts between different software factors.

Appendix A

Metric Questions for Reliability Factor

Appendix A contains the metric questions for reliability factor during Software Requirement Analysis phase (SRA), Detailed Design phase (DD), and Coding and CSU testing (CUT) phase proposed in [12]. Other development phases like the Preliminary Design (PD) and CSC Integration and Testing phase are not included here because there are no significant number of metric questions in [12]. Each metric is numbered according to the order appeared on the AMS metric question list. For example, metric question 4.b under the criterion Accuracy (AC1) represents that it is the fourth metric question related to the Software Requirement Analysis phase in that metric sets, namely AC1.

In [12], the metric elements are numbered consecutively under each metric and annotated with lowercase letters to indicate the phase which they should be applied. The lowercase letters for the development lifecycle are:

- a. System Requirement Analysis/Design
- b. Software Requirement Analysis
- c. Preliminary Design
- d. Detailed Design
- e. Coding and CSU Testing
- f. CSC Integratinon and Testing
- g. CSCI Testing

h. System Integration Testing

We extract the metric elements from [12] and follow their numbering convention in this chapter. Since we divide metric elements by phases into sections, the metric elements in each phase would not be consecutive. Because we only consider the software development lifecycle, we will start from Software Requirement Analysis phase in the next section.

A.1 Software Requirement Analysis Phase

The followings are questions for reliability measurement on Software Requirement Analysis Phase:

CRITERION: ACCURACY (AC)

Metric: AC.1 – Accuracy

[4.b] Are there quantitative accuracy requirements for all applicable inputs associated with each mission critical CSCI function?

[5.b] Are there quantitative accuracy requirements for all applicable outputs associated with each mission critical CSCI function?

[6.b] Are there quantitative accuracy requirements for all applicable constants associated with each mission critical CSCI function?

[7.b] Do the existing math library routines planned for use in the CSCI provide enough precision to support accuracy objectives?

CRITERION: ANOMALY (AM)

Metric: AM.1 – Error Tolerance/Control

[1.b] In how many instances are different functions allowed to execute at the same time (concurrent processing)?

[2.b] In how many instances is concurrent processing centrally controlled.

[3.b] How many error conditions are identified?

[4.b] How many identified error conditions are provided with processing instructions for recovery or repair of the error?

[5.b] Is there a standard for handling errors such that all error conditions are passed to the calling function/software element (CSCI, TLCSC, unit)?

[6.b] How many instances of the same function are required to execute more than once for comparison purposes? (For example, polling parallel or redundant processing results.)

[7.b] How many instances of parallel/redundant processing are centrally controlled?

Metric: AM.2 - Improper Input Data

[1.b] Are error tolerances specified for all applicable external input data to the CSCI?

Metric: AM.3 - Computational Failures

[1.b] Are there requirements for detection of and recovery from all computational failures?

[2.b] Are there requirements to range-test all loop and multiple transfer index parameters before use?

[3.b] Are there requirements to range-test all subscript values before use?

[4.b] Are there requirements to check all critical outputs to verify that they are reasonable before final outputting?

Metric: AM.4 - Hardware Faults

[1.b] Are there requirements to recovery from all detected hardware faults?

Metric: AM.5 - I/O Device Errors

[1.b] Are there requirements to recover from all I/O device errors.

Metric: AM.6 - Communication Errors

[1.b] Are there requirements to recover from all communication transmission errors?

Metric: AM.7 - Communication Failures

[1.b] Are there requirements to recover from all failures to communicate with other nodes/systems?

[2.b] Is there a requirement to periodically check adjacent nodes for operational status?

[3.b] Is there a requirement to provide a strategy for alternate routing of messages?

CRITERION: SIMPLICITY (SI)

Metric: SI.1 – Design Structure

[1.b] Are there diagrams identifying all CSCI functions in a structured fashion? (For example, top-down hierarchical.)

[9.b] Are there requirements for a programming standard?

[10.b] Has a programming standard been established?

Metric: SI.2 – Structured Language or Preprocessor

[1.b] Are there requirements to use a structured language or preprocessor for CSCI implementation?

Metric: SI.3 – Data and Control Flow Complexity

Metric: SI.4 – Coding Simplicity

Metric: SI.5 – Specificity

Metric: SI.6 – Halstead's Level of Difficulty

A.2 Detailed Design Phase

The followings are questions for reliability measurement on Detailed Design:

CRITERION: ACCURACY (AC)

Metric: AC.1 – Accuracy

CRITERION: ANOMALY (AM)

Metric: AM.1 - Error Tolerance/Control

[5.d] When an error condition is detected, is its resolution determined by the calling unit?

Metric: AM.2 - Improper Input Data

[2.d] Are values of all applicable inputs range-specified?

[3.d] Are all applicable inputs range-tested?

[4.d] Are conflicting requests and illegal combinations of all applicable inputs identified and checked?

[5.d] Are all inputs checked and all errors (resulting from those inputs) reported before processing begins?

[6.d] Is there a check to see if all data is available before processing begins?

Metric: AM.3 - Computational Failures

[1.d] Is recovery provided for all computational failures within the unit?

[2.d] Are all loop and multiple transfer index parameters range-tested before use?

[3.d] Are all subscript values range-tested before use?

[4.d] Are all critical outputs checked for reasonableness before final outputting?

Metric: AM.4 - Hardware Faults

Metric: AM.5 - I/O Device Errors

Metric: AM.6 - Communication Errors

Metric: AM.7 - Communication Failures

CRITERION: SIMPLICITY (SI)

Metric: SI.1 - Design Structure

[1.d] Does the design of the CSCI reflect a structured design approach? (For example, top-down design.)

[2.d] Is the unit independent of the source of input and destination of output?

[3.d] Is the unit independent of knowledge of prior processing?

[4.d] Does the unit description/prologue include input, output, processing, and limitations?

[5.d] How many entrances into the units?

[6.d] How many exit from the unit?

[7.d] How many unique data items are in common blocks in this CSCI?

[8.d] How many unique common blocks in this CSCI?

[11.d] Does this unit description identify all interfacing units and interfacing hardware?

Metric: SI.2 - Structured Language or Preprocessor

[1.d] How many units are implemented in a structured language or using a preprocessor?

Metric: SI.3 - Data and Control Flow Complexity

[1.d] How many conditional branch statements are there in the unit? (For example, IF, WHILE, REPEAT, DO/FOR, loop, CASE.)

[2.d] How many unconditional branch statements are there in the unit? (For example, GOTO, CALL, RETURN.)

Metric: SI.4 - Coding Simplicity

[1.d] Is the unit's flow of control from top to bottom (i.e., control does not erratically jump)?

[2.d] How many estimated lines of source code excluding comment lines and blank line are there for the unit?

[3.d] How many negative boolean and compound boolean expressions are used in the unit?

[4.d] How many loops are used in the unit? (For example, WHILE, REPEAT, DO/FOR.)

[5.d] How many loops have unnatural exits? (For example, jumps out of loop, return statements.)

[6.d] How many iteration loops are used in the unit (DO/FOR loops)?

[7.d] In how many iteration loops are indices modified to alter the fundamental processing of the loop?

[8.d] Is the unit free from all self modification of code? (For example, the unit does not alter instructions, overlays of code, etc.)

[10.d] What is the maximum nesting level in the unit?

[11.d] How many total branches (conditional and unconditional) are used in the unit?

[12.d] How many data declaration statements are there in the unit?

[13.d] How many data manipulation statements are there in the unit?

[14.d] How many total data items (local and global) are used in the unit?

[15.d] How many local data items are used in the unit? (For example, variables declared locally and value parameters.)

[16.d] Does each data item in the unit have a single use? (For example, each array serves only one purpose.)

[18.d] Does the CSCI avoid repeated and redundant code by utilizing macros/procedures/functions?

Metric: SI.5 – Specificity

[1.d] How many data items are used as input to the unit?

[2.d] How many data items are used for output by the unit?

[3.d] How many parameters in the unit's calling sequence return output values?

[4.d] Does the unit perform a single, non-divisible function?

Metric: SI.6 – Halstead's Level of Difficulty

- [1.d] How many unique operators are in the unit?
- [2.d] How many unique operands are in the unit?
- [3.d] How many total operands are in the unit?

A.3 Coding and CSU Testing

The followings are questions for reliability measurement on Code/Unit Testing (e):

CRITERION: ACCURACY (AC)

Metric: AC.1 – Accuracy

CRITERION: ANOMALY (AM)

Metric: AM.1 – Error Tolerance/Control

[5.e] When an error condition is detected, is its resolution determined by the calling unit?

Metric: AM.2 – Improper Input Data

[3.e] Are all applicable inputs range-tested?

[4.e] Are conflicting requests and illegal combinations of all applicable inputs identified and checked?

[5.e] Are all inputs checked and all errors (resulting from those inputs) reported before processing begins?

[6.e] Is there a check to see if all data is available before processing begins?

Metric: AM.3 – Computational Failures

[1.e] Is recovery provided for all computational failures within the unit?

[2.e] Are all loop and multiple transfer index parameters range-tested before use?

[3.e] Are all subscript values range-tested before use?

[4.e] Are all critical outputs checked for reasonableness before final outputting?

Metric: AM.4 – Hardware Faults

Metric: AM.5 - I/O Device Errors

Metric: AM.6 - Communication Errors

Metric: AM.7 - Communication Failures

CRITERION: SIMPLICITY (SI)

Metric: SI.1 - Design Structure

[2.e] Is the unit independent of the source of input and destination of output?

[3.e] Is the unit independent of knowledge of prior processing?

[4.e] Does the unit description/prologue include input, output, processing, and limitations?

[5.e] How many entrances into the units?

[6.e] How many exit from the unit?

[7.e] How many unique data items are in common blocks in this CSCI?

[8.e] How many unique common blocks in this CSCI?

[11.e] Does this unit description identify all interfacing units and interfacing hardware?

Metric: SI.2 - Structured Language or Preprocessor

Metric: SI.3 - Data and Control Flow Complexity

[1.e] How many conditional branch statements are there in the unit? (For example, IF, WHILE, REPEAT, DO/FOR, loop, CASE.)

[2.e] How many unconditional branch statements are there in the unit? (For example, GOTO, CALL, RETURN.)

Metric: SI.4 - Coding Simplicity

[1.e] Is the unit's flow of control from top to bottom (i.e., control does not erratically jump)?

[2.e] How many lines of source code, excluding comment lines and blank lines, are there for the unit?

[3.e] How many negative boolean and compound boolean expressions are used in the unit?

[4.e] How many loops are used in the unit? (For example, WHILE, REPEAT, DO/FOR.)

[5.e] How many loops have unnatural exits? (For example, jumps out of loop, return statements.)

[6.e] How many iteration loops are used in the unit (DO/FOR loops)?

[7.e] In how many iteration loops are indices modified to alter the fundamental processing of the loop?

[8.e] Is the unit free from all self modification of code? (For example, the unit does not alter instructions, overlays of code, etc.)

[9.e] How many statement labels are used in the unit, excluding labels for format statements?

[10.e] What is the maximum nesting level in the unit?

[11.e] How many total branches (conditional and unconditional) are used in the unit?

[12.e] How many data declaration statements are there in the unit?

[13.e] How many data manipulation statements are there in the unit?

[14.e] How many total data items (local and global) are used in the unit?

[15.e] How many local data items are used in the unit? (For example, variables declared locally and value parameters.)

[16.e] Does each data item in the unit have a single use? (For example, each array serves only one purpose.)

[17.e] Is this unit coded according to the required programming standard?

[18.e] Does the CSCI avoid repeated and redundant code by utilizing macros/procedures/functions?

Metric: SI.5 - Specificity

[1.e] How many data items are used as input to the unit?

[2.e] How many data items are used for output by the unit?

[3.e] How many parameters in the unit's calling sequence return output values?

[4.e] Does the unit perform a single, non-divisible function?

Metric: SI.6 - Halstead's Level of Difficulty

[1.e] How many unique operators are in the unit?

[2.e] How many unique operands are in the unit?

[3.e] How many total operands are in the unit?

Appendix B

Detailed Design and Coding for GEM

Appendix B contains the detailed design and coding for the GEM example discussed in Chapter 7. The following paragraphs are the reliability metric/criterion/factor values corresponding to the detailed design and coding of GEM.

B.1 Detailed Design of GEM

The detailed design of the Gaussian Elimination Method (GEM):

1. Augment the $n \times n$ coefficient matrix with the vector of righthand sides to form an $n \times (n + 1)$ matrix.
2. Interchange rows if necessary to make the value of a_{11} the largest magnitude of any coefficient in the first column.
3. Create zeros in the second through the n th rows in the first column by subtracting a_{i1}/a_{11} times the first row from the i th row. Store the a_{i1}/a_{11} in a_{i1} , $i = 2, \dots, n$.
4. Repeat Steps (2) and (3) for the second through the $(n - 1)$ st rows, put the largest-magnitude coefficient on the diagonal by interchanging rows (considering only rows j to n), and then subtract a_{ij}/a_{jj} times the j th row from the i th row so that in all positions of the j th column below the diagonal are zeros. Store the a_{ij}/a_{jj} in a_{ij} , $i = j + 1, \dots, n$. At the conclusion of this step, the system is upper-triangular.

5. Solve for x_n from the n th equation by formula:

$$x_n = a_{n,n+1}/a_{nn}.$$

6. Solve for $x_{n-1}, x_{n-2}, \dots, x_1$ from the $(n-1)$ st through the first equation by formula:

$$x_i = \frac{a_{i,n+1} - \sum_{j=i+1}^n a_{ij}x_j}{a_{ii}}$$

The following table is the metrics/criteria values for the reliability factor during the detailed design phase (DD) of the example GEM:

AC		AM		SI	
AC1	N/A	AM1	1.00	SI1	0.875
		AM2	0.80	SI2	1.00
		AM3	0.75	SI3	0.031
		AM4	N/A	SI4	0.882
		AM5	N/A	SI5	0.868
		AM6	N/A	SI6	0.92
		AM7	N/A		

$$Accuracy(AC) = AVE(AC1) = N/A$$

$$Anomaly(AM) = AVE(AM1, AM2, AM3, AM4, AM5, AM6, AM7) = 0.85$$

$$Simplicity(SI) = AVE(SI1, SI2, SI3, SI4, SI5, SI6) = 0.73$$

The reliability (RL) after the detailed design phase becomes

$$RL = AVE(AC, AM, SI) = 0.79$$

When this information is stored in the Object-Oriented Data Base, we will have one object (GEM_DD) in Class DPI, one object (GEM_DD_Re) in Class QFI to represent reliability factor, and two objects (GEM_DD_AM and GEM_DD_SI) in Class

QCF to represent the criteria anomaly and simplicity for reliability factor. The representations of these objects are shown as follows:

CLASS DPI

object

Development-phase-name: GEM_DD

Factor-information: (GEM_DD_Re↑, GEM_DD_Ef↑,...)

CLASS QFI

object

Quality-factor-name: GEM_DD_Re

Value: 0.79

Criterion-information: (GEM_DD_AM↑,
GEM_DD_SI↑)

CLASS QCI

object

Quality-criterion-name: GEM_DD_AM

Value: 0.85

Metric-information: (1.0, 0.80, 0.75, NA, NA, NA, NA)

CLASS QCI

object

Quality-criterion-name: GEM_DD_SI

Value: 0.73

Metric-information: (0.875, 1.00, 0.031, 0.882, 0.868, 0.92)

B.2 Coding of GEM

After the detailed design (DD) phase, the program was implemented using PASCAL programming language on a SUN workstation. The following table contains the reliability metric/criterion/factor values during the Coding and CSC Testing (CUT) phase (the program is given at the end of this appendix):

AC		AM		SI	
AC1	N/A	AM1	1.00	SI1	0.875
		AM2	0.80	SI2	N/A
		AM3	0.75	SI3	0.031
		AM4	N/A	SI4	0.896
		AM5	N/A	SI5	0.686
		AM6	N/A	SI6	0.92
		AM7	N/A		

$$Accuracy(AC) = AVE(AC1) = N/A$$

$$Anomaly(AM) = AVE(AM1, AM2, AM3, AM4, AM5, AM6, AM7) = 0.85$$

$$Simplicity(SI) = AVE(SI1, SI2, SI3, SI4, SI5, SI6) = 0.682$$

The reliability (RL) after the CSC integration testing phase:

$$RL = AVE(AC, AM, SI) = 0.766$$

All the quality information for GEM in the Coding and CSC Testing phase is represented as follows:

CLASS DPI

object

Development-phase-name: GEM_CUT

Factor-information: (GEM_CUT_Re↑, GEM_CUT_Ef↑,...)

CLASS QFI

object

Quality-factor-name: GEM_CUT_Re

Value: 0.766

Criterion-information: (GEM_CUT_AM↑,
GEM_CUT_SI↑)

CLASS QCI

object

Quality-criterion-name: GEM_CUT_AM

Value: 0.85

Metric-information: (1.0, 0.80, 0.75, NA, NA, NA, NA)

CLASS QCI

object

Quality-criterion-name: GEM_CUT_SI

Value: 0.682

Metric-information: (0.875, N/A, 0.031, 0.896, 0.689, 0.92)

The program written in PASCAL for the GEM example is given as follows.

```

program gaussian_elimination (input,output);

(*****)
(*
(* This module is used 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. The
(* number of system equations are less than 500. All constants used
(* inside this function are having 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.
(*
(*****)

const
    matrixsize = 500;
type
    matrixtype = array [1..matrixsize,1..matrixsize] of real;
    arraytype = array [1..matrixsize] of real;
var
    a : matrixtype;    (* a is the augmented matrix, including b *)
    n : integer;       (* number of rows and columns or matrix a *)
    p : integer;       (* number of columns of matrix b *)
                        (* We are given an n * n matrix a, n * p matrix b,
                        and n * p matrix x (result matrix) *)
    x : matrixtype;    (* x is used to store the resulting matrix *)
    i : integer;

procedure print (a : matrixtype; n, p : integer);

(* This procedure prints the input matrix a and b. Notes: the
   matrix a in program is an augmented matrix, but we print matrix a
   and matrix b separately. *)

var i,j : integer;
begin
    writeln;writeln;
    writeln(' This is matrix A:');
    writeln;

```

```

for i := 1 to n do
  begin
    write(' ':8);
    for j := 1 to n do
      begin
        write(a[i,j]:10:5);
      end;
    writeln;
  end;
writeln;writeln;
writeln(' This is matrix B:');
writeln;
for i := 1 to n do
  begin
    write(' ':8);
    for j := (n + 1) to (n + p) do
      begin
        write(a[i,j]:10:5);
      end;
    writeln;
  end;
end;

```

```

procedure printanswer (x : matrixtype; n, p : integer);

```

(* This procedure prints the answer of the problem. Notes: since the resulting matrix is a n by p matrix, the output will be printed as a general form. *)

```

var i, j : integer;
begin
  writeln;writeln;
  writeln(' This is the solution to the above system linear equation:');
  writeln;
  for i := 1 to n do
    begin
      write(' ':8);
      for j := 1 to p do
        begin
          write('x[' , i:1, ', ' , j:1, ' ] = ', x[i,j]:11:5, ' ');
        end;
      writeln;
    end;
  writeln;
  writeln('-----');

```

```

end;

procedure exchange (var a : matrixtype; row1, row2 : integer);

(* This procedure exchange the two rows in procedure Gaussian
   eliminaiton *)

var temp : arraytype;
    i : integer;
begin
    for i := 1 to matrixsize do temp[i] := a[row1,i];
    for i := 1 to matrixsize do a[row1,i] := a[row2,i];
    for i := 1 to matrixsize do a[row2,i] := temp[i];
end;

procedure findmax (var a : matrixtype; current : integer;
                  var row, col : integer);

(* This procedure finds the maximum pivot value in column 'col'. *)

var j : integer;
    max : real;
begin
    row := current;
    max := abs(a[current,col]);
    for j := (current + 1) to n do
        if abs(a[j,col]) > max then
            begin
                row := j;
                max := abs(a[j,col]);
            end;
    end;
end;

procedure gaussian_elim (var a : matrixtype; var n, p : integer);

(* This is the main procedure for gaussian elimination method *)

var
    i, j : integer;
    k : integer;
    row : integer;
    col : integer;
    tmp : real;

```

```

    m      : arraytype;
begin
  if n > matrixsize then writeln('Input matrix size error')
  else
    begin
      col := 0;
      for i := 1 to n do
        begin
          for j := 1 to matrixsize do m[j] := 0;
          col := col + 1;
          findmax(a,i,row,col);
          if row <> i then exchange(a,i,row);
          for k := (i + 1) to n do
            begin
              m[k] := a[k,col]/a[i,col];
            end;
          for k := (i + 1) to n do
            begin
              for j := col to (n + p) do a[k,j] := a[k,j] - (m[k] * a[i,j]);
            end;
          end;
        end;

      for i := 1 to p do x[n,i] := a[n,n+i]/a[n,n];
      for i := n-1 downto 1 do
        begin
          for j := 1 to p do
            begin
              tmp := 0;
              for k:= (i + 1) to n do tmp := tmp + a[i,k]*x[k,j];
              x[i,j] := (a[i,n+j]-tmp)/a[i,i];
            end;
          end;
        end;
      end;
    end;

begin (* main *)
  writeln;writeln;writeln;writeln;
  gaussian_elim(a,n,p);
  printanswer(x,n,p);
end.   (* end main *)

```

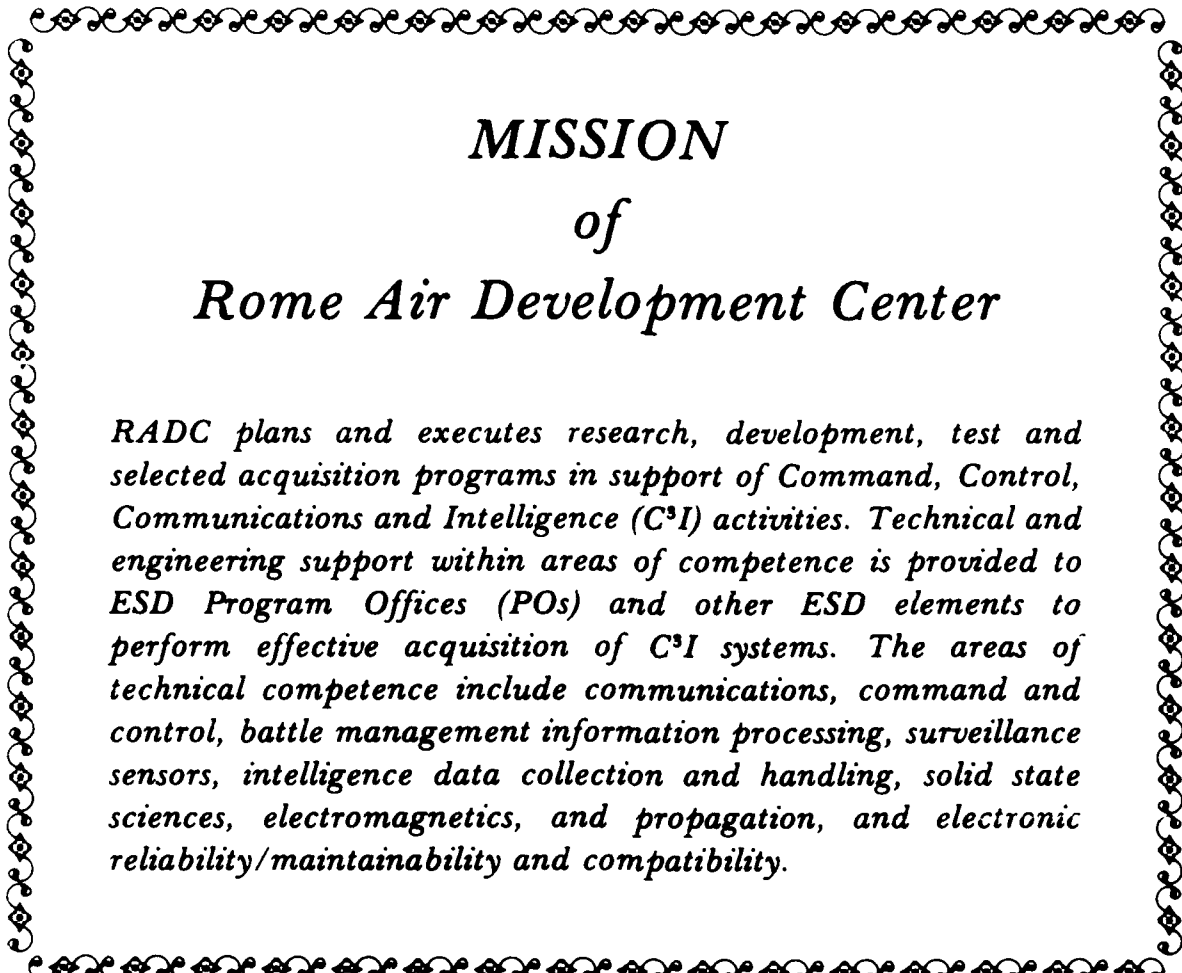
Bibliography

- [1] J. Cavano and J. McCall, "A Framework for the Measurement of Software Quality", *Proceedings of the ACM Software Quality Assurance Workshop*, November, 1978, pp. 133-139.
- [2] T. Bowen, J. Post, J. Tsai, E. Presson, and R. Schmidt, "Software Quality Measurement for Distributed Systems, Vol. I, II, III", *RADC-TR-83-175*, July 1983.
- [3] T. S. Chow (ed.), *Software Quality Assurance, A Practical Approach*, IEEE Computer Society Press, New York, NY 10017, 1985.
- [4] *Software Quality Assurance Plans, ANSI/IEEE Std 730-1984*.
- [5] *Software Quality Assurance Planning, ANSI/IEEE Std 983-1986*.
- [6] *DOD-STD-2167A*, Oct 1987.
- [7] *DOD-STD-2168*, Oct 1987.
- [8] S. D. Conte, H. E. Kunsmore, V. Y. Shen, *Software Engineering Metrics and Models*, The Benjamin Publishing Company, Inc., 1986.
- [9] S. S. Yau, J. S. Collofello, C. C. Hsieh, "Self-metric Software, A Handbook: Part I, Logical Ripple Effect Analysis", *NTIS AD-A086-291, RADC-TR-80-138*, April 1980.
- [10] S. S. Yau, and J. S. Collofello, "Some Stability Measures for Software Maintenance", *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 6, November 1980, pp. 545-552.
- [11] S. S. Yau, and J. S. Collofello, "Design Stability Measures for Software Maintenance", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 9, September 1985, pp. 849-856.
- [12] *Program Maintenance Manual for the Automated Measurement System*, Vol. I, II, Rome Air Development Center, 1987.
- [13] J. McCall, et al, "Methodology for Software Reliability Prediction", *RADC-TR-87-171, Vol I, II*, Science Applications International Corporation, November 1987.

- [14] P. Harmon and D. King, "Expert Systems", John Wiley & Sons, New York, 1985.
- [15] P. Harmon, R. Maus, and W. Morrissey, *Expert Systems, Tools and Applications*, John Wiley & Sons, New York, 1988.
- [16] R. Quillan, *Semantic Memory*, in *Semantic Information Processing*, M. Minsky (ed.), MIT Press, Cambridge, Mass. 1968.
- [17] E. H. Shortliffe, *Computer-based Medical Consultations: MYCIN*, Elsevier, New York, 1976.
- [18] M. Minsky, *A Framework for Representing Knowledge*, in *The Psychology of Computer Vision*, P. Winston (ed.), McGraw-Hill, New York, 1975.
- [19] D. H. Warren and L. M. Pereira, *Prolog: The Language and Its Implementation Compared with Lisp*, in *Proc. Symposium on Artificial Intelligence and Programming Languages*. SIGPLAN Notices 12(8) and SIGART Newsletter 64, 1977, pp. 109-115.
- [20] J. McDermott, *R1: A Rule-based Configurer of Computer Systems*, *Artificial Intelligence*, Vol. 19, Sept. 1982, pp. 39-88.
- [21] W. B. Gevarter, "The Nature and Evaluation of Commercial Expert System Building Tools", *IEEE Computer*, Vol. 20, No. 5, May 1987, pp. 24-41.
- [22] E. Rich, *Artificial Intelligence*, McGraw-Hill Book Company, New York, 1983.
- [23] A. Newell, H.A. Simon, "GPS, A Program that Simulates Human Thoughts," in *Computers and Thought*, E. A. Feigenbaum, J. Feldman (Ed.), McGraw-Hill, New York, 1963.
- [24] R. E. Fikes, and N. J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving", *Artificial Intelligence*, Vol 2, 1971, pp. 189-208.
- [25] K. J. Hammond, "Planning and Goal Interaction: The use of past solutions in present situations", *Proc. of American Association of Artificial Intelligence Conference 83*, August 1983, pp. 148-151.
- [26] M. J. Stefik, "Planning with Constraints", *Artificial Intelligence*, Vol. 16, 1981, pp. 111-140.
- [27] R. Wilensky, *Planning and Understanding*, Addison-Wesley Publishing Company, Reading Massachusetts, 1983.
- [28] M. J. Setfik, "Planning and Meta-Planning", *Artificial Intelligence*, Vol. 16, 1981, pp. 141-170.
- [29] R. E. Korf, "Planning as Search: A Quantitative Approach", *Artificial Intelligence*, Vol. 33, 1987, pp. 65-88.

- [30] R. Davis, "Meta-Rules: Reasoning about Control", *Artificial Intelligence*, Vol. 15, 1980, pp. 179-223.
- [31] J. Doyle, "A Truth Maintenance System", *Artificial Intelligence*, Vol. 12, 1979, pp. 231-272.
- [32] R. M. Stallman and G. J. Sussman, "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis", *Artificial Intelligence*, Vol. 9, No. 2, October 1977, pp. 135-196.
- [33] D. McAllester, "An Outlook on Truth Maintenance", Artificial Intelligence Laboratory, AIM-551, MIT, Cambridge, MA 1980.
- [34] D. McDermott, "Contexts and Data Dependencies: A Synthesis", *IEEE Trans. Pattern Anal. Machine Intelligence*, Vol. 5, No. 3, 1983, pp. 237-246.
- [35] J. P. Martins, "Reasoning in Multi Belief Spaces", Department of Computer Science, *Tech. Rept. No. 203*, State University of New York, Buffalo, NY, 1983.
- [36] J. de Kleer, "An Assumption-Based TMS", *Artificial Intelligence*, Vol. 28, No. 2, 1986, pp. 127-162.
- [37] J. F. Gilmore and K. Pulaski, "A Survey of Expert System Tools", *The Second Conference of AI Application*, 1985, pp. 498-502.
- [38] Teknowledge, M.1 product description. Teknowledge, 525 University Ave., Palo Alto, Calif., 1984.
- [39] Software Architecture and Engineering, Inc., Knowledge engineering systems. Artificial Intelligence Center, Suite 1220, 1401 Wilson Blvd., Arlington, VA 22209, November 1983.
- [40] L. Erman, A. C. Scott, and P. London, "Separating and Integrating Control in a Rule-Based Tool." *Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems*, IEEE Computer Society, IEEE Computer Press, Silver Spring, MD., 1984, pp. 37-43.
- [41] C. Williams, "ART: The Advanced Reasoning Tool." *Inference Corp. Report*, Inference Corp., 5300 W Century Blvd., Los Angeles, Calif., 1984.
- [42] J. C. Kunz, T. P. Kehler and M. D. Waterman, eds. "Applications Development Using a Hybrid AI Development System." *AI Magazine*, 5 (3), 1984, pp. 41-54.
- [43] R. Kempf and M. Stelzner, "Teaching Object-Oriented Programming with the KEE System", *OOPSLA-87 Proceedings of Object-Oriented Programming Systems, Languages, and Applications*, Vol. 22, No. 12, October 1987, pp. 11-25.
- [44] "User's Guide for the Automated Measurement System", *Rome Air Development Center*, 1987.

- [45] A. Goldberg and D. Robson, "Smalltalk-80: The Language and Its Implementation", Addison-Wesley, 1983.
- [46] V. Dhar and M. Jarke, "Dependency Directed Reasoning and Learning in Systems Maintenance Support", *IEEE Transactions on Software Engineering*, Vol. SE-14, No. 2, February 1988, pp. 211-227.
- [47] C. Potts and G. Bruns, "Recording the Reasons for Design Decisions", *Proc. of 10th International Conference on Software Engineering*, 1988, pp. 418-427.
- [48] C. Williams, "Managing Search in a Knowledge-Based System", unpublished, 1985.
- [49] J. de Kleer, "Problem Solving with the ATMS", *Artificial Intelligence*, Vol. 28, No. 2, 1986, pp. 197-224.
- [50] S. S. Yau and C. S. Liu, "An Approach to Software Requirement Specification", *COMPSAC 88 Proc.*, October 1988, pp. 197-224.
- [51] W. W. Agresti (ed.), *Tutorial, New Paradigms for Software Development*, IEEE Computer Society Press, 1986.



MISSION of Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C³I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C³I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.