AN AUTOMATED TOOL
FOR REQUIREMENTS VERIFICATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

YAŞAR TEKİN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE
OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF INFORMATION SYSTEMS

SEPTEMBER 2004

Approval of the Graduate School of Informatics

Prof. Dr. Neşe YALABIK
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Onur DEMİRÖRS
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Onur DEMİRÖRS
Supervisor

Examining Committee Members

Prof. Dr. Semih BİLGEN

Assoc. Prof. Dr. Onur DEMİRÖRS

Assoc. Prof. Dr. Col. Kadir VAROĞLU

Assist. Prof. Dr. Erkan MUMCUOĞLU

Dr. Altan KOÇYİĞİT

ABSTRACT

AN AUTOMATED TOOL
FOR REQUIREMENTS VERIFICATION

Tekin, Yaşar
M.S., Department of Information Systems
Supervisor: Assoc. Prof. Dr. Onur DEMİRÖRS

September 2004, 102 pages

In today's world, only those software organizations that consistently produce high quality products can succeed. This situation enforces the effective usage of defect prevention and detection techniques.

One of the most effective defect detection techniques used in software development life cycle is verification of software requirements applied at the end of the requirements engineering phase. If the existing verification techniques can be automated to meet today's work environment needs, the effectiveness of these techniques can be increased.

This study focuses on the development and implementation of an automated tool that automates verification of software requirements modeled in Aris eEPC and Organizational Chart for automatically detectable defects. The application of reading techniques on a project and comparison of results of manual and automated verification techniques applied to a project are also discussed.

Keywords: software testing, verification & validation, reading techniques, automated verification

ÖZ

GEREKSİNİM DOĞRULAMASI İÇİN
OTOMATİK BİR ARAÇ

Tekin, Yaşar
Yüksek Lisans, Bilişim Sistemleri
Tez Yoneticisi: Doç. Dr. Onur Demirörs

Eylül 2004, 102 sayfa

Günümüz dünyasında hiç şüphe yok ki sadece sürekli olarak yüksek kalitede ürünler üreten yazılım organizasyonları başarılı olabilir. Bu durum, hata önleme ve tespit etme tekniklerinin etkin kullanımını gerekli kılar.

Yazılım geliştirme süreci boyunca kullanılan en etkin hata tespit etme tekniklerinden biride gereksinim mühendisliği safhasının sonunda uygulanan yazılım gereksinim doğrulamasıdır. Eğer varolan doğrulama teknikleri bugünün iş ortamı ihtiyaçlarını karşılayacak şekilde otomatik hale getirilebilirse, bu tekniklerin etkinliği artırılabilir.

Bu çalışma Aris eEPC ve Organizational Chart ile modellenmiş yazılım gereksinimlerinin otomatik doğrulanabilir hatalarını tespit eden bir aracın geliştirme ve gerçekleştirilmesi hakkındadır. Okuma tekniklerinin bir projeye uygulanması ve el ile ve otomatik yapılan doğrulama tekniklerinin bir projeye uygulanma sonuçlarının karşılaştırılması ile ilgili olarakta çalışılmıştır.

Anahtar Kelimeler: Yazılım Sınaması, Doğrulama & Geçerlilik Denetimi, Okuma Teknikleri, Otomatik Doğrulama.

ACKNOWLEDGMENTS

I express sincere appreciation to my advisor Assoc. Prof. Dr. Onur Demirörs for his guidance throughout the research.

I also would like to express my appreciation to my family who always give me their love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVATIONS AND ACRONYMS

ARM	:	Automated Requirements Measurement
DTD	:	Document Type Definition
eEPC	:	Extended Event-Driven Process Chain
FIX	:	Feature Interaction Extractor
GSFC	:	Goddard Space Flight Center
IBM	:	International Business Machines
IEEE	:	Institute Of Electrical And Electronics Engineers
NASA	:	National Aeronautics And Space Administration
RFP	:	Request For Proposal
RSM	:	Requirements State Machine
RSML	:	Requirements State Machine Language
SAMM	:	Systematic Activity Modeling Method
SATC	:	Software Assurance Technology Center
SRS	:	Software Requirements Specifications
TCAS	:	Traffic Alert & Collision Avoidance System
UML	:	Unified Modeling Language
XML	:	Extensible Markup Language

CHAPTER 1

INTRODUCTION

1.1 Context

The development of software products is increasing at an unpredictable rate. By the effect of increased complexity and time to market pressures, the need for software quality is also increasing.

The quality of software systems is increased by defect detection and defect prevention activities. Effective defect prevention can be realized by increasing the quality of software development process that produces the software systems. Effective defect detection can be realized by increasing the quality of software testing process that detects the problems in the software systems.

Software testing has a life cycle that parallels the software development life cycle. It begins with the determination of software requirements and continues until the submission of end product. The main purpose of testing is to detect the defects as soon as possible and prevent migration of defects to later stages.

The initial stage of the software testing process involves careful review of the software requirements specification. Requirements specification is the description of the needed functionality and performance characteristics of the software product. A complete specification is essential to the success of any project. Omissions, inconsistencies, ambiguities, or contradictions not discovered during the initial investigation will propagate through the software life cycle and can result in either an improperly functioning system or an expensive and time-consuming redesign.

Early detection and correction of defects in the software requirements specification is essential to keep development costs down and to build correct and reliable software that satisfies the customer’s needs.

1.2 Problem Statement

It is known that the majority of errors in the software systems are injected during the requirements engineering phase of the software development process and correcting them can be costly if they are detected late in software lifecycle. So, it is essential to improve the quality of requirements specifications and detect the requirements errors in that phase.

There are numerous techniques for the specification of requirements such as object oriented modeling, view point oriented modeling and formal methods. Despite the fact that the modeling techniques provide some positive improvements in the quality of requirements specifications, verification of requirements remains as an important issue to increase the quality.

Requirements verification is the final stage of requirements engineering phase. The purpose of requirements verification is to assure that the requirements specification document states the correct description of the system.

Inspections and walkthroughs are the techniques used for verification of requirements specification documents. Because these techniques require qualified personnel and time, automation of verification process reduces the verification expenses and eliminates human failures.

In this thesis, an RFP project is considered which models The Turkish Army’s organizational structure and command-control system. Organizational structure of The Turkish Army and activity of each organizational unit in the command-control system is modeled by using Aris eEPC and organizational chart modeling techniques.

For the verification of the project, it was seen that some of the defects in the models could be detected by an automated tool. Then, because there was no such tool that can be used for the verification of eEPC and organizational chart modeling techniques, we decided to develop a tool for the verification of these techniques.

1.3 Approach

The aim of this study is to reduce the need for classified personnel and time required during requirements verification by developing an automated tool which detects automatically detectable defects in requirements specifications which use Aris eEPC and organizational chart as modeling techniques. To fulfill the aim of this thesis, first, reading techniques used in verification process are researched and checklists and scenarios related to requirements verification are obtained. Then, a coherent part of the project is manually verified by applying different reading techniques to different parts of the project. Based on the results, defect types are categorized and the defects which can be automatically detected are identified. Next, a software tool is developed and applied to the project for automated detection of defect types. Lastly, effectiveness of manual verification and automated verification tool are compared.

1.4 Thesis Structure

Chapter 2 provides a description of software testing, software verification techniques, reading techniques used in the software inspection, automated tools developed for requirement specifications verification and eEPC and Organizational Chart modeling techniques used to model the organizational structure and command-control system of The Turkish Army.

In chapter 3, information about the tool is given. Aris xml export utility, Xerces java parser, SWI Prolog, Prolog structures used in the application and the scripts written are explained in detail. Lastly, an example is given to illustrate the functioning of the tool.

In chapter 4, first, information about manual verification is given. The reading techniques used for the inspection process, result of each application and comparison of the results are given in detail. Defects detected during manual verification and categorization of them is explained. Second, information about automated verification is given. Lastly, comparison of manual and automated verification techniques is discussed.

Chapter 5 provides a conclusion to the study and includes directions for future work for automated verification.

CHAPTER 2

RELATED RESEARCH

This chapter presents an overview of software testing, software inspection, reading techniques used in software inspection and automated requirements verification tools. Additionally, eEPC and Organizational Chart modeling techniques are explained which are used to model business processes in the project under consideration.

2.1 Software Testing

To prevent the software to have errors, software developers need to understand and effectively apply software testing techniques. They have to try to detect the errors as soon as possible and test the software throughout the software development life cycle.

The IEEE/ANSI definition for testing is:

“The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspects of the system or component.” [1]

Edward Kit [2] states that this definition is validation oriented and need to include verification part of the testing. He gives the definition of testing as follows;

Testing = Verification + Validation.

2.1.1 Verification

The IEEE/ANSI definition for Verification is:

“The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. (contrast with validation)” [1]

There are two widely used methods for software verification which are inspection and walkthrough.

The main principle of inspection is to detect the defects during individual inspection. Each inspector is given a role and individual inspection is required before the inspection meeting. Most of the defects are detected during individual inspection. During the meeting, some additional defects are tried to be detected.

Walkthrough is less formal than inspection. It has no individual inspection phase. A meeting is organized and the participants gather without any preparation. The presenter reads the document and the participants try to detect the defects during the meeting.

Fagan stated the comparison of key properties of inspection and walkthroughs as given in table 1[3]:

Table 1 Comparison of Inspection and Walkthroughs

Properties	Inspection	Walkthrough
Formal moderator training	Yes	No
Definite participant roles	Yes	No
Who “drives” the inspection or walkthrough	Moderator	Owner of material
Use “how to find errors” checklists	Yes	No
Use distribution of error types to look for	Yes	No
Follow up to reduce bad fixes	Yes	No

Table 1 (cont.)

Properties	Inspection	Walkthrough
Less future errors because of detailed error feedback to individual programmer	Yes	Incidental
Improve inspection efficiency from analysis of results	Yes	No

2.1.2 Validation

The IEEE/ANSI definition for Validation is:

“The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. (contrast with verification)” [1].

There are two basic validation strategies which are black-box and white-box testing. In black-box testing, tests are applied according to functional specification of the system. Internal structure of the program is not tested. In white-box testing, tests are applied according to internal design specification. Internal structure of the program is tested.

2.2 Inspection

Inspection is one of the verification techniques used in software testing. This technique was developed and reported by Michael E. Fagan at IBM in 1976 [3]. Some subsequent improvements are made and today it’s a widely used technique in software development industry.

The basic objectives of inspections are [4]:

- To find errors at the earliest possible point in the development cycle
- To assure that the appropriate parties technically agree on the work
- To verify that the work meets predefined criteria

- To formally complete a technical task
- To provide data on the product and the inspection process

There are different inspection types in use today. Short descriptions of these inspection types are given below:

2.2.1 Fagan Inspection

Fagan inspection [3,5] is a detailed review of work in progress. Inspectors study work product independently and then gather to examine the work in detail. There are formally defined stages in the inspection process as shown in figure 1.

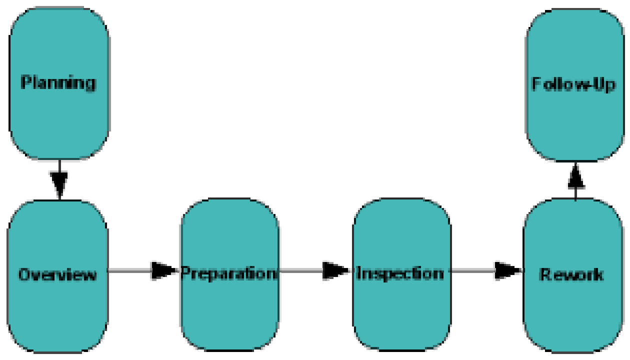


Figure 1 Fagan Inspection

- Planning:

Documents are checked if they meet the inspection entry criteria. Availability of the right participants and suitable meeting place is checked and meeting time is arranged.

- Overview:

The producer first describes the overall area being addressed and then the specific area he has produced in detail. Documentation is distributed to all inspection participants.

- Preparation:

Participants, using the documentation, work on the product to understand its intent and logic.

- Inspection:

A “reader” is chosen by the moderator who reads the work product. Every piece of work product is covered at least once by the reader. The finding of errors is done during the reader’s discourse. They are noted by the moderator, its type is classified, severity is identified and the inspection is continued.

- Rework:

All errors or problems noted in the inspection report are resolved by the producer.

- Follow-Up:

It is responsibility of the moderator to see that all issues, problems and concerns discovered in the inspection operation have been resolved.

2.2.2 Gilb Inspection

Gilb inspection is the improved version of Fagan inspection as shown in figure 2. Each stage in Gilb inspection is formally defined [6].

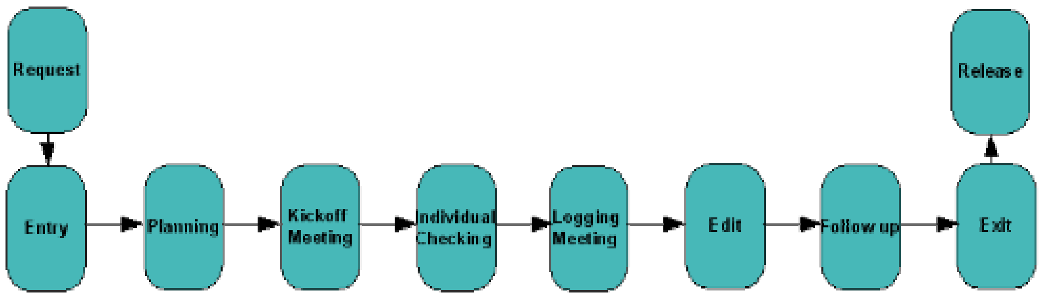


Figure 2 Gilb Inspection

- Request: initiating the inspection Process

The inspection process begins with a request by the author or owner of the work product. This is given to the responsible quality authority who finds a suitable inspection leader or directly to the inspection leader if one is already determined.

- Entry: Making sure loser inspections don't start

The Leader checks the product and its source documents against relevant entry criteria. The purpose of the entry criteria is to reduce the probability that the team will waste time and resources for a work product which it is impossible for the work product to satisfy "exit" criteria. Some entry criteria used in inspection process are:

  - The applicable set of generic and specific rules for the task which produced the product is available in writing
  - Ordinary text documentation shall have been cleaned up by a spelling checker before submission
  - The author or editor agrees to participate as a checker

- Planning: Determining the present inspection's objectives and tactics

The inspection leader plans where to gather, who should participate, and other details. This is the planning phase and results in a master plan for all the people in the inspection team.

- Kickoff Meeting: Training and motivating the team

A kickoff meeting is usually held to ensure that the inspectors know what to do in the inspection process. The kickoff meeting may include the distribution of documents, role assignments, training in inspection procedures, etc. The kickoff meeting may be held for the following specific purposes:

  - familiarize checkers with their tasks
  - agree on their individual special defect-searching role assignments which were suggested by the planner of the inspection in the master plan
  - hand out the recently produced materials as well as their source materials, relevant rules and checklists
  - ask any general questions about the documents being checked
  - obtain group or individual instruction on how to do the inspection work
  - inform team about current logging rates and effectiveness
  - identify and agree to use suitable new tactics for meeting their improvement targets

- Individual Checking: The search for potential defects

The inspectors work individually on the work product using the source documents, and the rules, procedures and checklists provided. The purpose of each inspector is to find the maximum number of defects.

- Logging Meeting: Log issues found earlier and check for more potential defects

A logging meeting is held for three purposes:

  - To log the issues which have already been identified by each inspector during individual inspection process
  - To detect more defects during the meeting
  - To identify and log ways of improving the development of the inspection process like improvement suggestions to procedures, rules or checklists

- Edit: improving the product

Someone, usually the producer, is given the log of issues to resolve. He or she works on the work product and resolves all the problems.

- Follow up: Checking the editing

The inspection leader checks that all logged issues are resolved by the producer and process improvement suggestions are sent to the inspection process management.

- Exit: Making sure the product is economic to release

The exit process is performed by the inspection leader using specific exit criteria. For example, follow up must be complete and the number of errors left in the document should be below a quality threshold.

- Release: The close of the inspection process

The product is made available, as officially exited with an estimate of the remaining major defects in a warning label.

2.2.3 Other Inspection Techniques

In this part, information about other inspection techniques proposed to increase the effectiveness of inspections is given.

2.2.3.1 N-Fold Inspection

N-Fold inspection uses formal inspections but replicates these inspection activities using N independent teams. The same software product is given to N inspection teams. Each inspection team performs formal inspection process and analyzes the software product. All results of the independent inspection processes are recorded in a database.

The primary use of N-Fold inspection is to identify defects that might not be detected by a single inspection team [7].

2.2.3.2 Two-Person Inspection

Two-person inspection uses the formal method of Fagan inspection with a two-person team which does not require initial resources of Fagan inspection technique. It eliminates the role of the moderator, assigns one person for tester and one person for producer roles [8].

2.2.3.3 Phased Inspection

A phased inspection consists of a series of coordinated partial inspections called phases. Each phase is applied to ensure that the product has a specific property. For example, phases can be used to ensure that a work product has characteristics such as portability, reusability or maintainability. The properties checked during phases are ordered so that each phase can assume the existence of properties checked in preceding phases [9].

2.3 Reading Techniques

Reading techniques provide a systematic and well-defined way of inspecting a document, allowing feedback and improvement [10]. It is a sequence of steps guiding the individual analysis of a text-based document in order to achieve the goals of a particular inspection task.

Different reading techniques are offered for the success of inspection process. General classifications of reading techniques used for the inspection process are given below:

2.3.1 Ad-Hoc Reading

Ad-hoc reading is not really a guided reading technique. The work product is given to the inspector without any guidelines, checklists and help. Therefore the success of the defect detection strongly depends on the skills and experience of the inspector.

2.3.2 Checklist-Based Reading

In the checklist-based reading technique, the inspector is given a checklist consisting of several questions. These questions are answered during the inspection by the inspector.

The inspector has to match the questions to the tasks he performs during the defect detection. The checklist reminds him which parts are to be checked and what aspects he should think of.

Heuristics that are commonly suggested for creating an effective inspection checklist include [11]:

- Checklists should be regularly updated based on defect analysis
- Checklists should not be longer than a single page

- Checklist items should be phrased in the form of a question
- Checklist items should not be too general

2.3.3 Scenario-Based Reading

The scenario-based reading technique provides guidance to the inspector by a scenario. The scenario can be a set of questions or a more detailed description of work. Usually, scenarios focus on certain details of the document, not the whole document. Therefore, to cover the whole document, different scenarios must be provided to each inspector. The teams' effectiveness can be increased by this way.

Scenario-based reading is a family of inspection techniques. These techniques are given below:

2.3.3.1 Defect-Based Reading

Defect-based reading is focused on different defect types. For each type of defect, there is a scenario or questions which guide the inspector in detecting them.

2.3.3.2 Perspective-Based Reading

Perspective-based reading defines roles for the inspectors. Each inspector inspects the document with this role. Therefore, each inspector inspects the documents with different point of views. This provides the individual inspector to spend more inspection time for his/her part of the documents and read it more carefully.

Table 2 given below presents some characteristics of reading techniques according to the following criteria [12]:

• Is systematic. Are the specific steps of the individual review process definable?

• Is focused. Must different reviewers focus on different aspects of the document?

• Allows controlled improvement. Based on feedback, can reviewers identify and improve specific aspects of the technique?

• Customizable. Can reviewers customize the technique to a specific project or organization?

• Allows training. Can reviewers use a set of steps to train themselves in applying the technique?

Table 2 Characteristics of Reading Techniques

Technique	Systematic	Focused	Controlled improvement	Customizable	Training
Ad hoc	No	No	No	No	No
Checklist	Partially	No	Partially	Yes	Partially
Defect-based reading	Yes	Yes	Yes	Yes	Yes
Perspective-based reading	Yes	Yes	Yes	Yes	Yes

The choice of defect detection method significantly affects inspection performance. To discover the effectiveness of reading techniques, several experiments are conducted and several different results are achieved. One experiment states that Ad Hoc is less efficient or similar than Checklist, which is less efficient than Scenario [13]**.** Another experiment states that the Scenario detection methods resulted in the highest defect detection rates, followed by Ad Hoc detection methods, and finally by Checklist detection methods [14]. As both experiments stated, scenario based reading is the most effective technique among the reading techniques.

2.4 Automated Tools

In this part, information about automated tools developed for requirements verification is provided.

2.4.1 Automated Requirements Measurement (ARM)

The Goddard Space Flight Center's (GSFC) Software Assurance Technology Center (SATC) has developed the tool for assessing requirements that are specified in natural language [15]. The SATC's mission is to assist National Aeronautics and Space Administration (NASA) projects to improve the quality of software that they acquire or develop. The tool developed searches the documents for terms the SATC has identified as quality indicators. The reports produced by the tool are used to identify specification statements and structural areas of the requirements specification document that need to be improved.

The tool uses indicators of quality attributes to evaluate the requirements documents. These indicators are grouped into two classes. Those related to the examination of individual specification statements which are Imperatives, Continuances, Directives, Options and Weak Phrases and those related to the total requirements document which are Size, Specification Depth and Text Structure.

- Imperatives

Imperatives are words and phrases that command something must be provided. The ARM report uses Shall, Must or must not, Is required to, Are applicable, Responsible for, Will and Should imperatives and lists the total number of times imperatives were detected.

- Continuances

Continuances are phrases such as Below, As follows, Following, Listed, In particular and Support which introduce the specification of requirements at a lower level. They are found to be an indication that requirements were organized and structured.

- Directives

Directives are the category of words and phrases such as Figure, Table, For example and Note which point to illustrative information within the requirements document. The data and information pointed to by directives strengthens the document’s specification statements and makes them more understandable. A high ratio of the total count for the Directives category to the documents total lines of text appears to be an indicator of how precisely requirements are specified.

- Options

Options are the category of words such as Can, May and Optionally which give the developer latitude in satisfying the specification statements that contain them. This category loosens the specification, reduces the acquirer’s control over the final product, and establishes a basis for possible cost and schedule risks.

- Weak Phrases

Weak Phrases is the category of clauses that are apt to cause uncertainty and leave room for multiple interpretations. Use of phrases such as “adequate” and “as appropriate” indicate that what is required is either defined elsewhere or the requirement is open to subjective interpretation. Phrases such as “but not limited to” and “as a minimum” provide a basis for expanding a requirement or adding future requirements. The total number of weak phrases found in a document is an indication of the extent that the specification is ambiguous and incomplete.

- Size

Size is the category used by the ARM tool to report three indicators of the size of the requirements specification document. They are:

  - lines of text
  - imperatives
  - subjects of specification statements

The number of lines of text in a specification document is accumulated as each string of text is read and processed by the ARM program. The number of subjects used in the specification document is a count of unique combinations and permutations of words immediately preceding imperatives in the source file. This count appears to be an indication of the scope of the document. The ratio of lines of text to imperatives provides an indication of how concise the document is in specifying the requirements.

- Specification Depth

Specification Depth is a category used by the ARM tool to report the number of imperatives found at each of the document's levels of text structure. This data is significant because it reflects the structure of the requirements statements as opposed to that of the document's text. Differences between the Text Structure counts and the Specification Depth were found to be an indication of the amount and location of text describing the environment that was included in the requirements document. The ratio of the specification depth category to document's total lines of text appears to be an indication of how concise the document is in specifying requirements.

- Text Structure

Text Structure is used by the ARM tool to report the number of statement identifiers found at each hierarchical level of the requirements document. These counts provide an indication of the document's organization, consistency, and level of detail. The text structure of documents judged to be well organized and having a consistent level of detail were found to have a pyramidal shape. Documents that exhibited an hour-glass shaped text structure were usually those that contain a large amount of introductory and administrative information. Diamond shaped documents indicated that subjects introduced at the higher levels were addressed at different levels of detail.

2.4.2 Feature Interaction Extractor (FIX)

FIX was developed to be used in telecommunication services. First a formal specification language is presented based on temporal logic. In this language, features of the system are defined. As an example, a telephony feature, such as call waiting or call forwarding, typically specifies the behavior over time of one or more entities in terms of their current state and a set of input events. The informal specification for call forwarding "If entity x has call forwarding enabled and calls to x are to be forwarded to z then, whenever x is busy, any incoming call from y to x is eventually forwarded to z." can be expressed with predicates call_forwarding_enabled(x), forward_from_to(x,z), forwarded_call_from_to(y, x, z), busy(x), and incoming_call_from_to(y, x).

Feature conflict is defined as mutually inconsistent properties; that is, no program exists that can implement both features. Consider the two features A and B defined below.

A : calls(a, b) => connected(a, b) v disconnect(a)
("Whenever a calls b, a and b are connected, unless a disconnects"),
B : calls(a, b) => forwards(a, b, c) v disconnect(a)
("Whenever a calls b, the call is forwarded to c, unless a disconnects").

These features are conflicting because forwarding from b and connecting to b should not both happen for the same call [16].

2.4.3 TCAS II

TCAS II was developed to be used in avoidance systems required on all commercial aircrafts. The method ensures that the verified properties hold for the specification by using functional composition rules.

A high level specification language RSML (Requirements State Machine Language) was developed by adding some features of state charts to RSM. The RSML is composed of Super states, AND decomposition and transition definitions.

- Super states

In RSML, states may be grouped into super states as shown in figure 3. Such groupings reduce the number of transitions by allowing transitions to and from the super state rather than requiring explicit transitions to and from all of the sub states. There are two ways to a super state. First, the transition to the super state may end at the super state's border. In this case, a default state must be specified within the super state. Alternatively, the transition may be made to a particular state inside the super state.

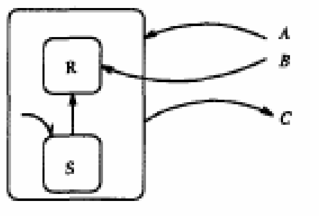


Figure 3 Superstates

- AND decomposition

It contains two or more state machines separated by dashed borders as shown in figure 4. When a parallel state is entered, each of the state machines within it is entered. All state machines are exited when any transition is taken out of the parallel state. The use of parallel states greatly reduces the size of the specification.

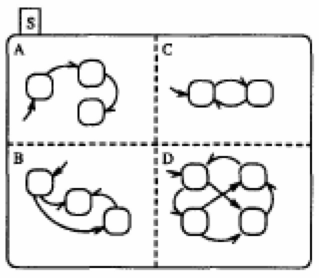


Figure 4 And composition

• Transition definition

Transitions are taken upon the occurrence of the trigger event, provided that the guarding condition is true. The guarding condition defines what must be true before the transition can be taken and is specified using AND/OR tables. Output actions identify events that are generated when the transition is taken.

The rules for union, parallel, and serial composition can then be applied to show that the behavior of the entire hierarchical and parallel machine is complete and consistent.

• Union Composition:

Union composition requires that the domains of the functions describing the transitions involved in the composition must be disjoint, i.e., no two transitions out of the same state can be satisfied at the same time. In addition, functions require that the entire domain must be covered. Thus, there must be a satisfiable transition out of every state independent of what input arrives at the model boundary.

• Serial Composition:

Serial composition of functions requires that if an event is generated, there must always be a transition elsewhere in the model ready to be triggered by this event.

• Parallel Composition:

Parallel composition occurs when two or more transitions in parallel state machines are triggered by the same event. If the truth value of the guarding condition of one transition can be affected by a state change caused by a parallel transition, then there exists a possibility of non determinism and the transitions are said to conflict with each other [17].

2.4.4 SAMMDF

The motivation for the development of SAMM was originated in response to the need to perform an analysis of current aircraft manufacturing processes. SAMM is based on the Human Directed Activity Cell Model developed by Hori in 1971 [18].

The purpose of SAMM is to model a system through a layered structure of activities and data flow. The SAMM representation scheme is comprised of three elements which are tree structure, activity diagram and condition chart.

• Tree structure

In SAMM, the nodes of trees represent activities which are a generalized concept to represent any action performed by a machine, or people, or combination of both to accomplish a task as shown in figure 5. The tree structure is used to organize the semantic refinement of an activity into its subordinate sub activities. An activity and its subordinates form the basis of an activity diagram.

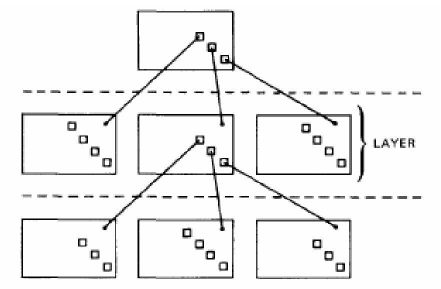
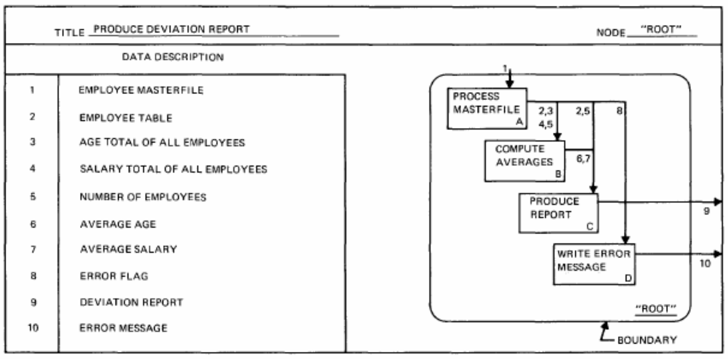


Figure 5 Tree structure

• Activity Diagram

An activity diagram consists of a description of sub activities and a data table. The description of sub activities is comprised of activity cells, a boundary, and data flows. The data table is comprised of data descriptions with indexes. A sub activity is referred to as an activity cell and is represented graphically by a rectangular box containing a descriptive verb phrase and a label, and data flows into and/or out of the cell.

In Figure 6, Box A is an example of an activity cell. The word "Process Masterfile" is the descriptive phrase and the letter "A" is the label. The arrow into the top of the cell with the numeral 1 by it indicates that the line is an input data flow. Numeral 1 refers to the first entry in the data description table titled "Employee Masterfile." Numbers 2, 3, 4, and 5 depicts data flowing from cell A to B.

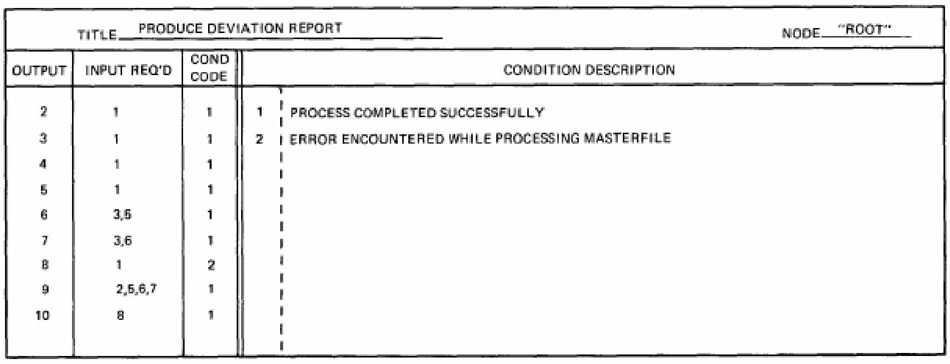


1	EMPLOYEE MASTERFILE
2	EMPLOYEE TABLE
3	AGE TOTAL OF ALL EMPLOYEES
4	SALARY TOTAL OF ALL EMPLOYEES
5	NUMBER OF EMPLOYEES
6	AVERAGE AGE
7	AVERAGE SALARY
8	ERROR FLAG
9	DEVIATION REPORT
10	ERROR MESSAGE

Figure 6 Activity Diagram

• Condition Chart

The purpose of a condition chart is to state for an activity diagram the input requirements of each output and to describe the behavioral aspects of the diagram. The entry 6|3, 5|1 in Figure 7 means output 6 (Average Age) is the result of inputs 3 (Age Total of All Employees) and 5 (Number of Employees) as shown in Figure 6 under condition 1 (Process Completed Successfully) from Figure 7 again.



TITLE PRODUCE DEVIATION REPORT NODE "ROOT"

OUTPUT	INPUT REQ'D	COND CODE	CONDITION DESCRIPTION	
2	1	1	1	PROCESS COMPLETED SUCCESSFULLY
3	1	1	2	ERROR ENCOUNTERED WHILE PROCESSING MASTERFILE
4	1	1		
5	1	1		
6	3,5	1		
7	3,6	1		
8	1	2		
9	2,5,6,7	1		
10	8	1		

Figure 7 Condition Chart

The automation of verification is accomplished in three ways: basic syntax and consistency checking; analysis of model, both on a diagram and global basis, utilizing graph-theoretic techniques; and by providing reports to utilize the human pattern-matching capability.

- Consistency Checking

This approach is taken on a diagram basis. When reviewing a diagram and its chart, certain questions must be asked. Are there indexes referencing undefined data or control descriptions? Are there data or control descriptions defined but not used? For each output item of a cell, is the output-input relationships specified on the chart? Does each diagram have at least one external input and one external output? Is the external data of a diagram related back to the data in the parent diagram?

- Connectivity Analysis

In analyzing the data flow graph, it is essential to determine that the graph is connected and each is accessible either from the input set node or the output set node.

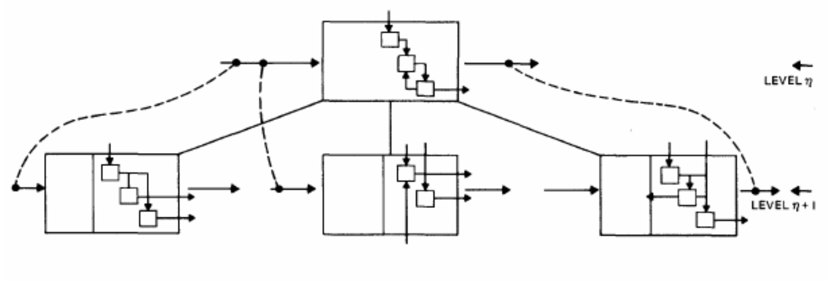


Figure 8 Data flow graph

The data flow graph is derived from a diagram or a layer of interrelated diagrams as shown in figure 8. A connection matrix for the graph is then constructed. A connection matrix is a square Boolean matrix where there is a row and column for each node in the graph. A one is placed in the connection matrix, C, at Cij if the node represented by column j can be reached directly from the node represented by row i. After the connection matrix is formulated, a reachability matrix is computed. The reachability matrix shows which nodes can be reached from other nodes. The information developed from the matrix is used to identify the sources and sinks of the graph.

- Computer-Assisted Checking

The objective of this approach is to present the different perspectives of the relationships found in a model. The perspectives are then detailed in a report which can be evaluated by a human for correctness.

A useful report produced is the data path report. The report provides insight into the behavior characteristics of the system and a scenario on how an external global output is transformed through the cells of a layer from its external global input.

A comparison of the tools, their usage areas, modeling techniques and methods used to detect the defects is given in table 3.

Table 3 Automated Tools

Tool Name	Usage Area	Modeling Technique	Method
Automated Requirements Measurement (Arm)	Nasa	Natural language	Individual indicators
Feature Interaction Extractor (Fix)	Telecom	W-Automata	Feature conflict
TCAS II	Avoidance systems	RSML	Union Composition, Serial Composition, Parallel Composition
SAMMDF	Aircraft manufacturing	SAMM	Consistency Check, Connectivity Analysis, Computer-Assisted Check

ARM tool assess requirement specifications written in natural language and searches the document for terms the SATC has identified as quality indicators. SATC use metrics to identify risks by using the indicators. ARM does not assess the correctness of requirements. But the tool we developed assesses completeness, consistency, ambiguity and semantic/syntax properties of models and detect the defects related to these properties.

Three major aspects comprise SAMMDF are model generation and modification, automated analysis and report generation. The tool we developed only focus on automated analysis of models.

SAMMDF uses consistency checking and connectivity analysis, TCAS II uses completeness and consistency checking and FIX uses conflict detection to verify the requirements. The tool we developed uses completeness, consistency, ambiguity and semantic/syntax properties of the models to verify the requirements.

Additionally, the verification tool we developed can be used with any modeling tool which provides eEPC and organizational chart modeling techniques if the tool uses the same DTD file with Aris.

2.5 eEPC and Organizational Chart

There are numerous modeling techniques that can be used to model the system requirements like data flow diagram and UML. In the project under consideration which models The Turkish Army's organizational structure and command-control system, eEPC and organizational chart modeling techniques in Aris are used. Organizational chart modeling technique is chosen because it is suitable to model the structure of an organizational hierarchy and eEPC modeling technique is chosen because it is suitable to model high level business processes, inputs, outputs and the relations between them.

In this part, information about eEPC and organizational chart modeling techniques is provided [19].

2.5.1 Extended Event-Driven Process Chain (eEPC)

In this model, the sequence of functions is illustrated in the form of process chains. The start and end events can be specified for each function. Events not only do trigger functions but are also results of functions.

Functions are displayed as rectangles with rounded corners and events are graphically represented as hexagons as shown in figure 9.

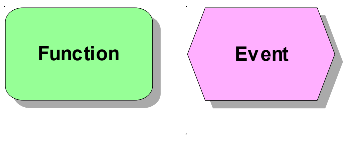


Figure 9 Function and Event

Since events determine which state or condition will trigger a function and which status will define the end of a function, the starting and end nodes of an eEPC are always events.

Several functions can be triggered from one event and a function can result in several events.

Functions and events are linked to each other by AND, OR or XOR operators.

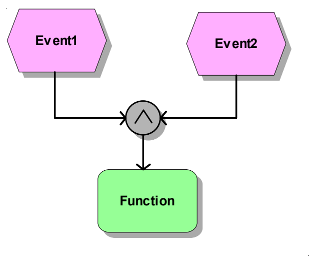


Figure 10 And Link Example

In the example given in figure 10, the starting events are linked by an AND operator. This means that the procedure Function is only started if the triggering events have been verified. Therefore both events must have occurred before the procedure can begin.

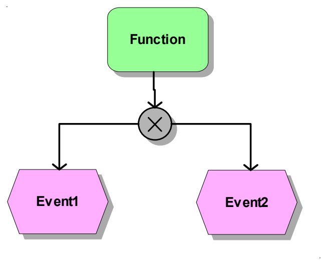


Figure 11 Xor Link Example

The second example given in figure 11 shows an exclusive OR operator. The Function may either result in Event1 or in Event2. Both results cannot occur at the same time.

Definitions of operators are given below:

AND operator: The outgoing can only be started after all incomings have occurred or the incoming results in all outgoings occurring.

OR operator: The outgoing can be started after at least one of the incomings has occurred or executing the incoming results in at least one of the outgoings occurring.

XOR operator (Either/Or operator): The outgoing is started after exactly one and only one incomings has occurred or executing the incoming results in one outgoing at the most occurring.

Event AND Links:

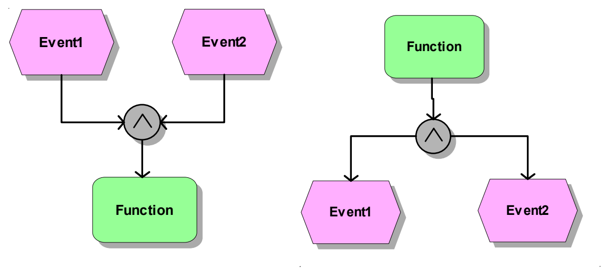


Figure 12 Event AND Links

Event OR Links:

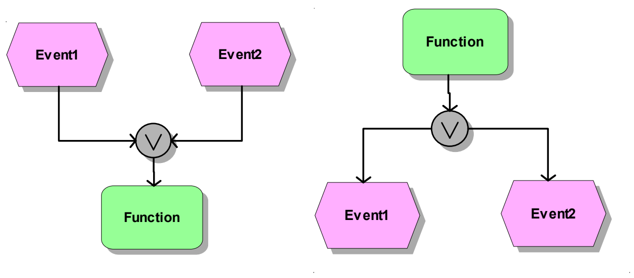


Figure 13 Event OR Links

Event XOR Links:

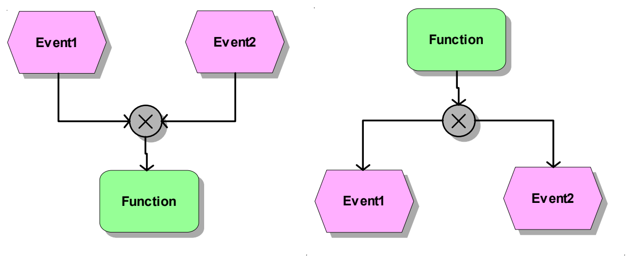


Figure 14 Event XOR Links

Function AND Links:

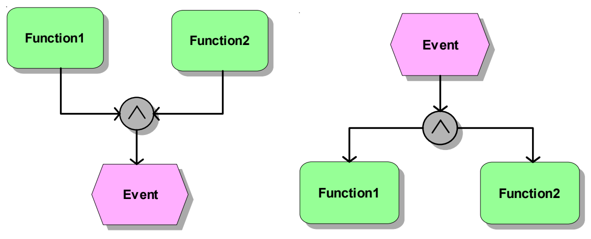


Figure 15 Function AND Links

Function OR Links:

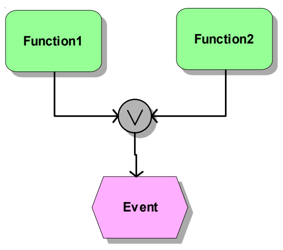


Figure 16 Function OR Link

Function XOR Links:

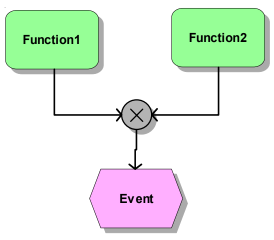


Figure 17 Function XOR Link

In figure 18, an example eEPC model from the project is given with the names changed to illustrate the use of eEPC in the project. In the figure, the triggering events Event1, Event2, Event3 and Event4 are linked by an Or operator. This means that the procedure Function is started if at least one of these events has been verified. The result events Event5, Event6 and Event7 are linked by an Or operator which is linked Event8 by an And operator. This means that the execution of procedure Function results in Event8 and at least one of the events Event2, Event3 and Event4.

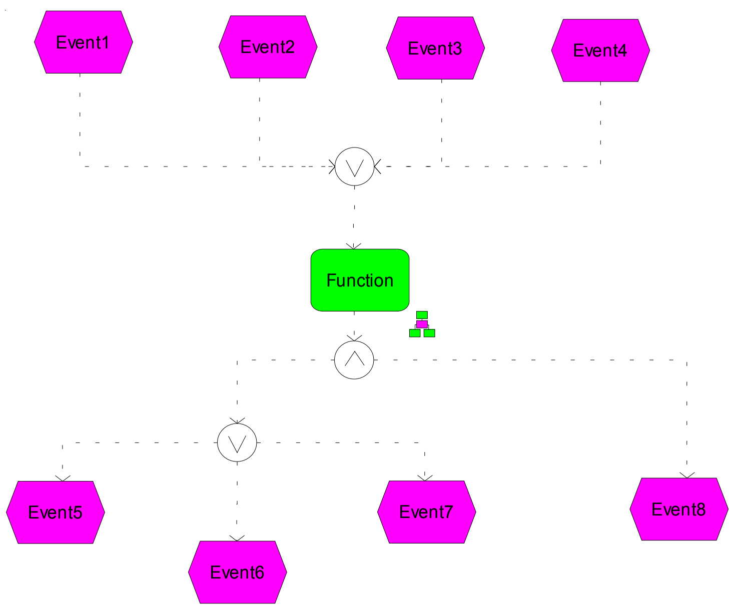


Figure 18 Example eEPC from the project

2.5.2 Organizational Chart

An organizational chart is a model used to represent organizational structures. It reflects the organizational units and their interrelationships. The relationships are the links between the organizational units.

In order to show the individual positions, a separate Position object type is available.

Organizational units are displayed as ellipses and positions are graphically represented as rectangles as shown in figure 19.

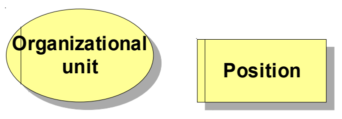


Figure 19 Organizational Unit and Position

One organizational unit can be linked to multiple positions. The connection means that the position is the manager of that organizational unit.

In figure 20, an example organizational chart model from the project is given with the names of the objects changed to illustrate the use of organizational chart in the project. In the figure, Unit1 is the root of the model. Unit2, Unit3, Unit5, Unit6 and Unit7 are linked Unit1 and Unit4 is linked Unit3. This means that Unit4 is a sub unit of Unit3 and the other organizational units are the sub units of Unit1. Position1 is linked Unit1 and Position2 is linked Unit4. This means that Position1 is the manager of Unit1 and Position2 is the manager of Unit4.

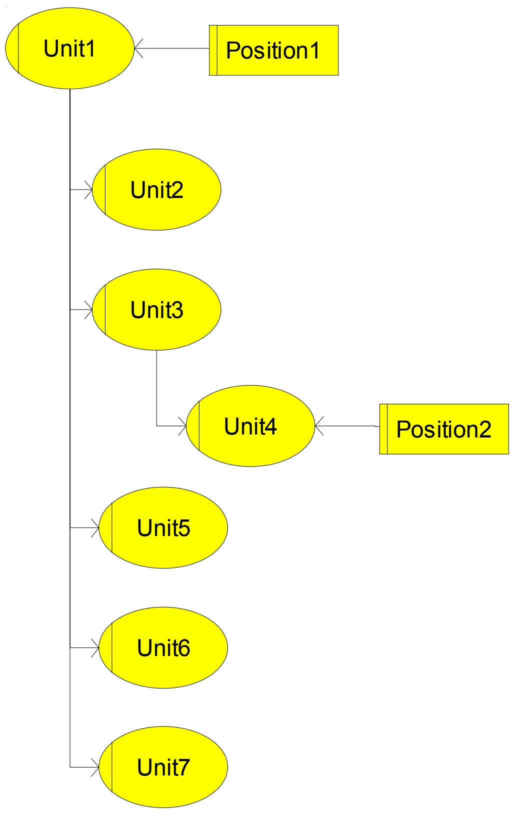


Figure 20 Example Organizational Chart from the project

CHAPTER 3

THE TOOL

This chapter gives information about the tool that is developed to detect the defects in a requirements specification modeled by eEPC and organizational chart.

3.1 Defect Detection Steps

First, models are exported to an XML file by using Aris XML export utility. Then these XML files are converted into given prolog structures by using “xerces-2_6_2” XML Parser. A class is written which examines the elements, gets the specified values and writes them to a file. Then these prolog files are consulted to a prolog interpreter with prolog scripts written to find the defects in the models as shown in figure 21.

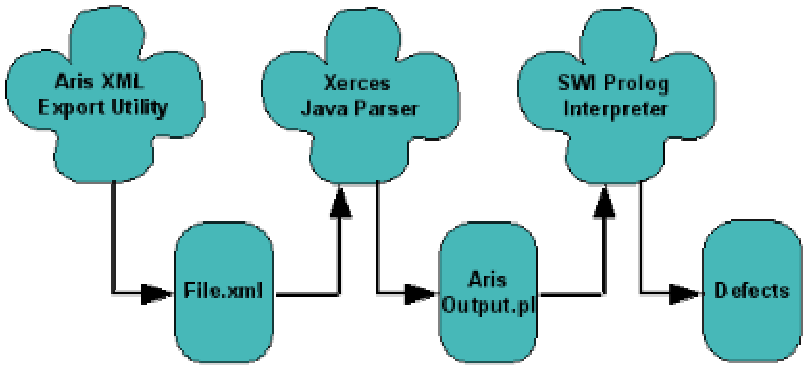


Figure 21 Application Process

3.2 Xerces XML Parser

An XML parser parses XML documents and sends data to display in a browser or to write to a file. The parser makes sure that the document meets the predefined structures which are defined in the form of a DTD or a schema [24].

The Xerces2 java parser is a free software which is a member of Apache parser family. For the thesis, Xerces 2.6.2 release is used. A class named Writer.java is written which converts Aris XML files to prolog files.

3.3 SWI Prolog

SWI-Prolog is a free software prolog compiler. Its development started in 1987 in the SWI department of the University of Amsterdam. Being free, small and standard compliant, SWI-Prolog has become very popular for education [25].

The SWI-Prolog executable plwin.exe can be started from the Start Menu or by opening a .pl file holding Prolog program text from the Windows explorer. After loading a program, one can ask Prolog queries about the program.

3.4 Prolog Structures

By examining ARIS-Export.dtd file, two prolog structures are defined which are object and model structures.

```
object(
        assigned model id,
        object id,
        [incoming connections],
        object type,
        object name,
        [outgoing connection, target object]).

model(
        model id,
        model type,
        model name,
        [model objects]).
```

The object structure defines each object in a model and the model structure defines a model and its components. Explanations of components of structures are:

assigned model id: if the object has a sub model assigned to it, id of that sub model is written. Otherwise component gets ‘no’ value.

object id: Id of the object is written to that component.

incoming connections: if the object has incoming connections, id’s of these connections are written as a list. Otherwise component gets empty list.

object type: Type of the object is written.

object name: Name of the object is written.

[outgoing connection, target object]: if the object has outgoing connections, id’s of these connections and their target objects are written as a list of lists. Otherwise component gets empty list.

model id: : Id of the model is written to that component.

model type: Type of the model is written.

model name: Name of the model is written.

model objects: ids of all objects in the model is written as a list.

3.5 Prolog Scripts

In this part of the thesis, scripts written in prolog are explained and pseudo code of each script is given.

• No_manager_for_organizational_units (rule 1)

This script finds the organizational units in the organizational charts which have no connection to a manager. The script is given in appendix A.

Pseudo code of the script:

```
Select organizational unit type objects as a list
For each organizational unit, find incoming object connections
For each connection, check the source object
If the source object is a position type, append this organizational unit type object to a list
Compare organizational unit type objects list with the list of organizational unit type objects connected to a position
If an organizational unit is detected that is in organizational unit type objects list but not in organizational unit type objects connected to a position list, write an error message
```

• No_manager_for_functions (rule 2)

This script finds the functions in eEPC models which have no connection to a manager. The script is given in appendix B.

Pseudo code of the script:

```
Select function type objects as a list
Select position type objects as a list
Select application system type objects as a list
Get the function type objects that are connected to a position by using position type objects list
Get the function type objects that are connected to an application system type by using application system type objects list
Append these two connected functions lists
```

```
Compare connected functions list with function type objects list
If a function is detected that is not in connected functions list but in function type objects list, write an error message
```

- Assigned_model (rule 3)

This script finds the function and sub-function relations which have inconsistencies with incoming or outgoing event names, info carriers and performing organizational units. The script is given in appendix C.

Pseudo code of the script:

```
Select a function type of object
Get the object’s incoming and outgoing objects
Get assigned-model object ids of this function type object
Convert assigned-model object ids to names
Compare function type object’s incoming and outgoing object names with assigned-model object names
If an object name is detected that is in incoming and outgoing objects list but not in assigned-model objects list, write an error message
```

- Assigned_model_name (rule 4)

This script finds the function and sub-function relations which have inconsistencies with names given to the function and sub-function assigned to it. The script is given in appendix D.

Pseudo code of the script:

```
Select a function of type object and get its name
Get the name of assigned-model of this function type object
Compare the names
If they are not the same, write an error message
```

• Rule_must_be_used (rule 5)

This script finds the objects which have more than one same kind of incoming or outgoing objects with no rule connection. The script is given in appendix E.

Pseudo code of the script:

```
Select a function type of object
For each incoming connection, find the type of source object and append it to a list
For each outgoing connection, find the type of destination object and append it to a list
Check each lists individually
If they have more than one same kind of object type, write an error message
```

• Have_same_name (rule 6)

This script finds the functions that the names of a function and its incoming or outgoing event are the same. The script is given in appendix F.

Pseudo code of the script:

```
Select a function type of object and get its name
Get this object’s incoming objects and their names
Get this object’s outgoing objects and their names
Compare function type object’s name with the names of incoming and outgoing objects
If they are the same, write an error message
```

• One_in_one_out_rule (rule 7)

This script finds the rules with only one incoming and one outgoing objects. The script is given in appendix G.

Pseudo code of the script:

```
Select a rule type object
Get the number of incoming connections for this rule type object
Get the number of outgoing connections for this rule type object
If both numbers are equal to 1, write an error message
```

3.6 Example

In this part of the thesis, an example is given to illustrate the functioning of the tool. First, an example eEPC model is constructed as shown in figure 22 and 23 which has all kinds of defects defined in the thesis. The defects in the example model are listed below:

a) Function31 has no connection to a manager
b) Function32 has no connection to a manager
c) Function33 has no connection to a manager
d) Function5 has no connection to a manager
e) Function51 has no connection to a manager
f) Function52 has no connection to a manager
g) Function53 has no connection to a manager
h) Function7 has no connection to a manager
i) Function5 has two outgoing events Event3 and Event4 which are not in its sub model
j) Function3 has a connection to Position2 which is not in its sub model
k) Function5 has a sub model named Function Error which is different from the function's name
l) Function1 has two incoming events with no rule connection
m) Function3 has two outgoing rules with no rule connection

n) Function4 has two outgoing functions with no rule connection

o) Function5 has two outgoing events with no rule connection

p) Function7 has an outgoing event named Function7 which is the same with the function’s name

q) Function7 has an incoming rule with only one incoming and one outgoing objects

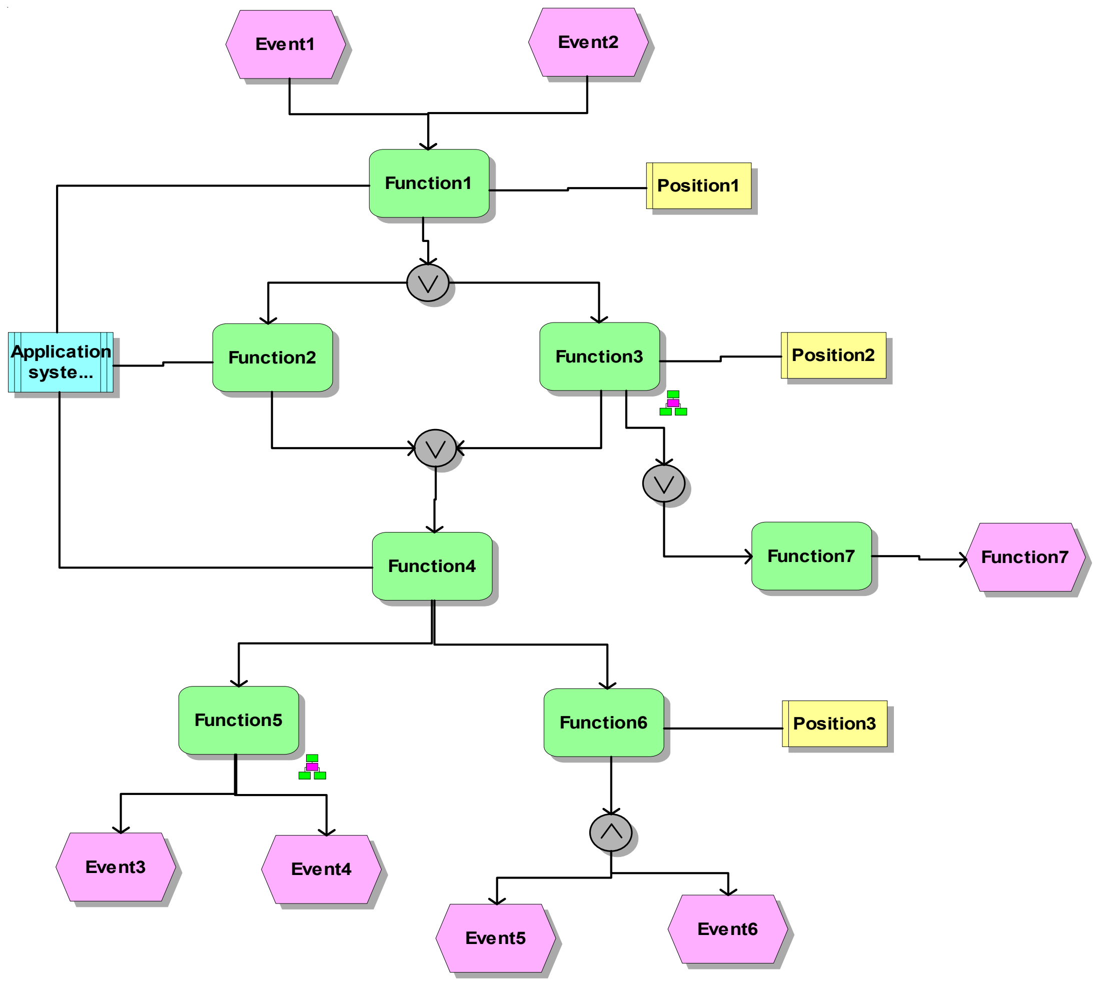


Figure 22 Example EEPC Main Model

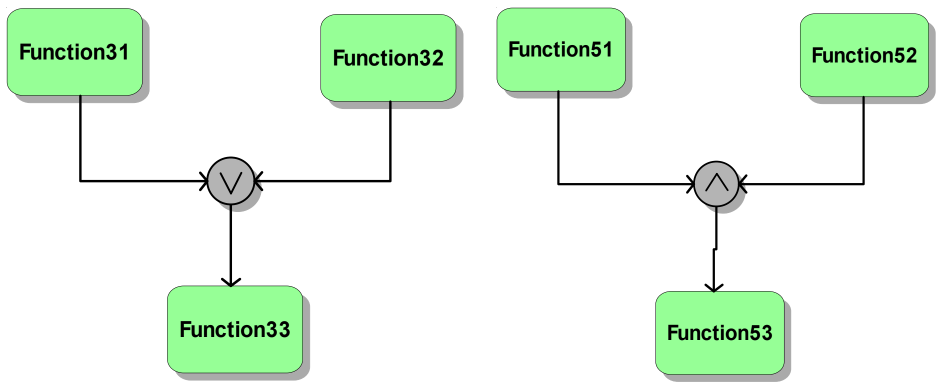


Figure 23 Example EEPC Sub-Models (Function 3 and Function Error)

Then the XML file is exported by using Aris XML Export utility. After that, by using Xerces java parser, information in the xml file is converted into two prolog structures which are explained in the preceding part and written in a file named arisOutput.pl. This file is given in appendix H. This prolog file is consulted to the prolog interpreter with the prolog scripts written to find the defects and the interpreter is run. The defects detected by using the scripts are given below:

- Defects detected by No_manager_for_functions script

```
Function5 does not have a manager
Function51 does not have a manager
Function52 does not have a manager
Function7 does not have a manager
Function33 does not have a manager
Function31 does not have a manager
Function53 does not have a manager
Function32 does not have a manager
```

This script finds the defects a, b, c, d, e, f, g and h given above.

- Defects detected by Assigned_model script

Event3 can not be found in sub-model
Event4 can not be found in sub-model
Position2 can not be found in sub-model

This script finds the defects i and j given above.

- Defects detected by Assigned_model_name script

Function5 is not the same with assigned sub-model name

This script finds the defect k given above.

- Defects detected by Rule_must_be_used script

Function5 must use a rule connection for incoming or outgoing objects
Function3 must use a rule connection for incoming or outgoing objects
Function1 must use a rule connection for incoming or outgoing objects
Function4 must use a rule connection for incoming or outgoing objects

This script finds the defects l, m, n and o given above.

- Defects detected by Have_same_name script

Function7 has an incoming or outgoing with the same name

This script finds the defect p given above.

- Defects detected by One_in_one_out_rule script

Rule outgoing from Function7 has one incoming and one outgoing

This script finds the defect q given above.

When the application results of the scripts are compared with the defects in the example model listed above, it is seen that all the defects in the model are detected by the tool.

CHAPTER 4

EXPERIMENTAL STUDY

In this chapter, we give information about the experiment that is conducted to assess the effectiveness of verification techniques. First, application of reading techniques and their comparison are considered. Second, the application results of automated tool are given. Lastly, comparison of manual and automated verification techniques is given.

4.1 Manual Verification

As part of our approach for the development of the tool, we have performed a manual verification experiment. In this experiment, a coherent part of a project is taken and inspected by using different reading techniques. The results are collected, categorized and used for the development of the tool.

The manual verification of the project is executed by one person. First, reading techniques for requirements verification are surveyed and checklists and scenarios are obtained from past experiences to be used in verification procedure. The selected model is divided into four parts to apply each technique on a different part of the model. Ad-Hoc Reading, Checklist-Based Reading, Defect-Based Reading and Perspective-Based Reading techniques are used for the application and the result of each technique is classified according to the characteristics of Software Requirements Specifications defined by IEEE [20].

4.1.1 Application of Ad-Hoc Reading

During this application, no guidelines, checklists and scenario are provided. Result of the application is given in table 4:

Table 4 Application Results of Ad-Hoc Reading

Defect Type	Class	Defects
There are some organizational units which does not have performing organizational unit.	Incompleteness	17
There are some functions which does not have performing organizational unit.	Incompleteness	22
There are some functions named “registering to related file” which does not have any related file info.	Incompleteness	1
There are some function and sub-function relations which have inconsistencies with incoming or outgoing events names, info carriers, performing organizational units and connected decimal files	Inconsistency	1
There are some function and sub-function relations which have inconsistencies with names given to the function and sub-function assigned to it	Inconsistency	4
Some objects have more than one same kind of incoming or outgoing objects with no rule connection.	Ambiguous	41
There are some functions with the names of the function and its incoming or outgoing event are the same	Ambiguous	1
There are some rules with only one incoming and one outgoing event.	Semantic /Syntax	2

It is observed from the results of application that the technique is applicable to the project. The success of the defect detection strongly depends on the skills of the inspector and defects detected are only related to completeness, consistency, ambiguity and Semantic/Syntax.

4.1.2 Application of Checklist-Based Reading

The checklist used for the application is given in appendix I [21].

Result of the application is given in table 5:

Table 5 Application Results of Checklist-Based Reading

Defect Type	Defect
Complete	0
Correct	(not applicable)
Precise, unambiguous and clear	1
Consistent	0
Relevant	(not applicable)
Testable	(not applicable)
Traceable	(not applicable)
Feasible	(not applicable)
Free of unwarranted design detail	(not applicable)
Manageable	(not applicable)

It is observed from the results of application that the technique is not applicable to the project. The questions in the checklists are about low level details of specification written in natural language and because the project works on high level specification of business work flow, only completeness, consistency and ambiguity characteristics of the project can be examined. Other questions related to the other characteristics are not applicable.

4.1.3 Application of Defect-Based Reading

The defect types used for the application are given in appendix J [22].

Result of the application is given in table 6:

Table 6 Application Results of Defect-Based Reading

Type	Defect	Class	Defects
Omission	There are some functions or org. Units which does not have performing organizational unit.	Incompleteness	15
Ambiguous information	There are some function and sub-function relations which have inconsistencies with incoming or outgoing events names, info carriers, performing organizational units etc.	Inconsistency	2
Ambiguous information	There are some function and sub-function relations which have inconsistencies with names given to the function and sub-function assigned to it.	Inconsistency	1
Ambiguous information	Some objects have more than one same kind of incoming or outgoing objects with no rule connection	Ambiguous	14
Incorrect fact	Not applicable		-
Extraneous	No defect		0
Other defects	There are some rules with only one incoming and one outgoing event	Semantic/ Syntax	1

It is observed from the results of application that the technique is applicable to the project. It gives the same results with ad hoc reading and it has no additional advantage over ad hoc reading technique.

4.1.4 Application of Perspective-Based Reading

Only Perspective-Based Reading User-based Reading scenario is applied to the model. The other scenarios (Design based Reading scenario and Test based Reading scenario) are not applicable because of lack of sufficient detail. The questions used for the application is given in appendix K, L and M [22].

Result of the application is given in table 7:

Table 7 Application Results of Perspective-Based Reading

Defect Type	Class	Defects
Is there anything that prevents you from writing this operational scenario? (position omissions)	Incompleteness	0
Are all the functions necessary to write this operational scenario specified in the requirements or functional specifications?	Correctness	0
Are the initial conditions for starting up this operational scenario clear and correct? (All functions are triggered by an event)	Ambiguous	0
Are the interfaces well defined and compatible, e.g., do the inputs of one function link to the outputs of the previous function?	Not applicable	-
Are the effects of the operational scenario specified in the requirements or functional specifications under all possible circumstances?	Not applicable	-
Might some portion of the operational scenario give different answers depending on how a requirement or functional specification is interpreted?	Not applicable	-
Does the requirement or functional specification make sense from what you know about the application or from what is specified in the general description?	General Question	-
Can you get into a state of the system that must be avoided, e.g. for reasons of safety or security?	Not applicable	-

It is observed from the results of application that the technique is not applicable to the project. The project does not contain the necessary information to answer the questions in the scenarios. Only some questions from user based reading technique can be applied which check the defects related to work flow.

A comparison of the reading techniques, their applicability and reasons for that is given in table 8.

Table 8 Reading Techniques

Reading technique	Applicability	Reason
Ad-Hoc Reading	applicable to the project	-
Checklist-Based Reading	not applicable to the project	questions in the checklist are about low level details of specification written in natural language
Defect-Based Reading	applicable to the project	-
Perspective-Based Reading	not applicable to the project	the model of the project does not contain the necessary information to apply different viewpoints

4.2 Automated Verification

For the categorization of defects, we used characteristics of a good SRS. In IEEE Recommended Practice for Software Requirements Specifications [20], characteristics of a good SRS are defined as:

- Correct
- Unambiguous
- Complete
- Consistent
- Ranked for importance and/or stability
- Verifiable
- Modifiable
- Traceable.

4.2.1 Defects

As the result of manual verification, eight different types of defects are detected. Then they are grouped according to characteristics of a good SRS given above. The defects detected and the characteristics they belong to are given below:

- Incompleteness

  - There are some organizational units which does not have performing position (No_manager_for_organizational_units, rule 1)
  - There are some functions which does not have performing position (No_manager_for_functions, rule 2)
  - There are some functions named “registering to related file” which does not have any related file info (no rule can be written)

- Inconsistency

  - There are some function and sub-function relations which have inconsistencies with incoming or outgoing event names, info carriers, performing organizational units and connected decimal files (Assigned_model, rule 3)
  - There are some function and sub-function relations which have inconsistencies with names given to the function and sub-function assigned to it (Assigned_model_name, rule 4)

- Ambiguous

  - Some objects have same kind of incoming or outgoing objects with no rule connection (Rule_must_be_used, rule 5)
  - There are some functions whose names are the same with its incoming or outgoing events (Have_same_name, rule 6)

- Semantic/Syntax

  - There are some rules with only one incoming and one outgoing event (One_in_one_out_rule, rule 7)

The names given in parenthesis are the names of scripts written to detect the defects.

For automated verification, seven of detected defects are selected. The defect “There are some functions named registering to related file which does not have any related file info” is not selected because it can not be detected by an automated tool.

Then, prolog scripts are written for each defect type and applied to the same project models which are used for manual verification. The results of the application are given in table 9:

Table 9 Application Results of Automated Verification

	Rule 1	rule 2	rule 3	rule 4	rule 5	rule 6	rule 7
Total	**17**	**40**	**4**	**6**	**76**	**1**	**3**

4.3 Comparison of Results

In table 10, application results of manual verification and in table 11, comparison of manual and automated verification results are given.

Table 2 Application Results of Manual Verification

	Rule 1	rule 2	rule 3	rule 4	rule 5	rule 6	rule 7
Ad-Hoc Reading	17	22	1	4	41	1	2
Checklist-Based Reading	-	-	-	-	1	-	-
Defect-Based Reading	-	15	2	1	14	-	1
Perspective-Based Reading	-	-	-	-	-	-	-
Total	**17**	**37**	**3**	**5**	**56**	**1**	**3**

Table 31 Comparison of Results

	rule 1	rule 2	rule 3	rule 4	rule 5	rule 6	rule 7	total
Manual	**17**	**37**	**3**	**5**	**56**	**1**	**3**	**122**
Automated	**17**	**40**	**4**	**6**	**76**	**1**	**3**	**147**

As given in the table above, number of defects detected during automated verification (147) is higher than the number of defects detected during manual verification (122 + the defect that can not be automated). When the defects detected during each verification technique are compared, it is seen that all the defects detected during manual verification are also detected during automated verification except the defect type that can not be automated. During automated verification, some additional defects are detected which could not detected during manual verification.

All defect types can not be detected by the automated tool. As an example, we can not automate the defect “There are some functions named registering to related file which does not have any related file info”.

The time spent for manual verification is approximately three days. This time includes defining the defect types in the models. For automated verification, it takes thirty minutes to complete the verification for the same models.

Main difference between the manual and automated verification occurred at “Rule_must_be_used” script. Reasons for this difference are considered as:

- Number of defects is high at this defect type.
- Because all the connection and object occurrences have to be checked in “Rule_must_be_used”, possibility of missing the defects during manual verification is the highest.

CHAPTER 5

CONCLUSION AND FUTURE DIRECTIONS

This thesis focused on the verification of requirements specifications by using manual and automated verification techniques, comparison of results and advantages and disadvantages of the automated tool we developed. This chapter gives how the tool we developed fulfilled the objectives and points out the future work for related issues.

5.1 Fulfillment of Objectives and Aim

In recent years, several tools have been developed for automated verification of software requirements. Some of them are focused on the assessment of requirements specified in natural language and some others on the assessment of graphical representations.

In this thesis, we focused on the verification of eEPC and organizational chart modeling techniques used in a military project.

First, we applied different reading techniques to different parts of the project and categorized the results of applications to determine the defect types which can be detected by an automated tool. The application results of reading techniques are summarized below:

Ad-hoc reading technique is applicable to the project because it requires no past experience such as checklist questions or scenarios.

Checklist-based reading technique is not applicable to the project because the checklists used in the application have questions related to requirements specifications written in natural language. So, for the success of checklist-based reading technique, a new checklist must be written with the experience of this work.

Defect-based reading technique is applicable to the project because the defect types given in the example are general classifications. So, during an application, reviewers can check different kind of issues under each defect class and detect different kind of defects.

Perspective-based reading technique is not applicable to the project because the modeling techniques used in the project does not contain low level details of different view points.

After determination of defect types, we developed an automated tool to automatically detect these defect types by using an xml parser and a prolog interpreter. Then, we executed automated verification of the manually verified project models by using the tool and compare the results of manual and automated verification.

The tool developed for automated detection of defects has significant advantages over manual verification of models. First, during manual verification, the time spent for verification is approximately three days. This time includes defining the defect types in the models and searching the models for these defect types. For automated verification, it takes thirty minutes to complete the verification process for the same models. Second, during manual verification, total number of automatically detectable defects found is 122. But, for automated verification, total number of defects found is 147. It means that, by manual verification, only 83 percent of total defects could be found.

The tool proposed here also has some disadvantages over manual verification. First, all defect types can not be detected by the automated tool. As an example, we can not automate the defect “There are some functions named registering to related file which does not have any related file info”. Second, the tool is specific to Aris Tool eEPC and organizational chart modeling techniques. It can not be used with other modeling tools unless the same DTD file is used.

5.2 Future Work

The result of the application of the tool suggests that the tool is applicable for verification of software requirements modeled using Aris eEPC and organizational chart modeling techniques. But some further experiments must be conducted to compare the verification results of manual verification and the tool we developed which shows the effectiveness of the tool.

For widespread use of the tool, new defect types for eEPC and organizational chart must be defined and new scripts must be written. This provides a higher number in detection of automatically detectable defects.

The same experiments must be conducted for other modeling techniques such as UML. This provides the usage of the tool with a wide variety of modeling techniques and increases the tool’s applicability in requirements engineering domain.

A study must be done to define new rules by using user interfaces instead of writing new scripts for each new defined defect type. And lastly, a better user interface might be a good idea to ease the use of the tool.

REFERENCES

1. IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, The Institute of Electrical and Electronics Engineers

2. Kit E., (1995), Software Testing In The Real World, Harlow, ACM Press Books

3. Fagan, M. E., (1976), Design And Code Inspections To Reduce Errors In Program Development, IBM Systems J., Vol. 15, No. 3, pp 182-211

4. Humphrey W. S., (1989), Managing The Software Process, Addison Wesley Publishing Company

5. Fagan. M. E., (1986), Advances In Software Inspections, IEEE Trans. Software Eng., Vol. 12, No. 7, pp 744-751

6. Glib T., Graham D., (1993), Software Inspection, Harlow, Addison Wesley Longman Limited

7. Martin J., Tsai W. T., (1990), N-Fold inspection: A Requirements Analysis Technique, ACM, Vol. 33, No. 2, pp 225-232

8. Bisant D. B., Lyle J. R., (1989), A Two Person Inspection Method To Improve Programming Productivity, IEEE Trans. Software Eng., Vol. 15, No. 10, pp 1294-1304

9. Knight J. C., Myers E. A., (1993), An Improved Inspection Technique, ACM, Vol. 36, No. 11, pp 51-61

10. Shull F., Rus I., Basili V., (2001), Improving Software Inspections By Using Reading Techniques, IEEE, pp 726-727

11. Brykczynski B., (1999), A Survey of Software Inspection Checklists, ACM SIGSOFT, Vol. 24, No. 1, pp 82-89

12. Shull F., Rus I., Basili V., (2000), How Perspective Based Reading Can Improve Requirements Inspections, IEEE, pp 73-79

13. Kirner T. G., Abib J. C., (1997), Inspection Of Software Requirements Specification Documents: A Pilot Study, Utah, ACM, pp 161-171

14. Porter A. A., Votta L. G., (1994), An Experiment To Assess Different Defect Detection Methods For Software Requirements Inspections, IEEE, pp 103-112

15. Wilson W. M., Rosenberg L. H., Hyatt L. E., (1997), Automated Analysis Of Requirement Specifications, Boston, ICSE, pp 161-171

16. Felty A. P., Namjoshi K. S., (2003), Feature Specification And Automated Conflict Detection, ACM, Vol. 12, No. 1, pp 3-27

17. Heimdahl M. P. E., Leveson N. G., (1995), Comleteness And Consistency Analysis Of State-Based Requirements, Seattle, ACM, pp 3-14

18. Stephens S. A., Tripp L. L., (1978), Requirements Expression And Verification Aid, pp 101-108

19. IDS Scheer AG., (2003), Aris methods, Saarbrücken

20. IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements Specifications, The Institute of Electrical and Electronics Engineers

21. Freedman D. P., Weinberg G. M., (1990), Walkthroughs, Inspections and Technical Reviews, New York, Dorset House Publishing

22. Laitenberger O., (1995), Perspective Based Reading: Technique, Validation And Research In Future, ISERN Technical Report

23. Williamson H., (2001), The Complete Reference XML, California, McGraw-Hill

WEB REFERENCES

24. Xerces2 Java Parser Read me, http://xml.apache.org/xerces2-j/index.html, Last visited in December 2004

25. What is SWI-Prolog?, http://www.swi-prolog.org , Last visited in December 2004

APPENDICES

APPENDIX A

No_Manager_For_Organizational_Units Script

```
nomanagerfororgunits:-

findall(Ot_org_unit,object(_,_,_,'OT_ORG_UNIT',Ot_org_unit,_),ListOutgoing
s),
  List2 = [],
  getfacts(ListOutgoings,ListOutgoings,List2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


getfacts(ListOutgoings,[],List):-
  results(ListOutgoings,List).

getfacts(ListOutgoings,[Head|List1],List2):-
  object(_,_,[Z|Tail1],_,Head,_),
  object(_,_,_,'OT_POS',Name2,[[X,Y]|Tail2]),
  nomanager([Z|Tail1],[[X,Y]|Tail2],[[X,Y]|Tail2]),
  append(List2,[Head],List3),
  getfacts(ListOutgoings,List1,List3).

getfacts(ListOutgoings,[Head|List1],List2):-
  object(_,_,[Z|Tail1],_,Head,_),
  object(_,_,_,'OT_POS',Name2,[[X,Y]|Tail2]),
  not(nomanager([Z|Tail1],[[X,Y]|Tail2],[[X,Y]|Tail2])),
  getfacts(ListOutgoings,List1,List2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

results([],List1).

results([Head|List1],List2):-
  member(Head,List2),
  results(List1,List2).

results([Head|List1],List2):-
  not(member(Head,List2)),
  write(Head),write(' does not have a manager'),nl,
  results(List1,List2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

nomanager([],[[X,Y]|Tail2],Tail2copy):- fail.

nomanager([Z|Tail1],[],Tail2copy):-
  nomanager(Tail1,Tail2copy,Tail2copy).

nomanager([X|Tail1],[[X,Y]|Tail2],Tail2copy):-!.

nomanager([Z|Tail1],[[X,Y]|Tail2],Tail2copy):-
  nomanager([Z|Tail1],Tail2,Tail2copy).
```

APPENDIX B

No_Manager_For_Functions Script

```
nomanagerforfunctions :-
  findall(Ot_func,object(_,_,_,'OT_FUNC',Ot_func,_),Listotfunc),
  findall(Ot_pos,object(_,_,_,'OT_POS',Ot_pos,_),Listotpos),

findall(Ot_appsystype,object(_,_,_,'OT_APPL_SYS_TYPE',Ot_appsystype,_)
,Listotappsystype),
  getotposfunctions(Listotpos,Listotposfunctions),
  getotappsystypefunctions(Listotappsystype,Listotappsystypefunctions),

append(Listotposfunctions,Listotappsystypefunctions,ListFunctionswithmana
gers),
  results(Listotfunc,ListFunctionswithmanagers).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getalloutgoings([],[]).

getalloutgoings([[Head1Listoutgoings|[Head2Listoutgoings]]|TailListoutgoings
],Listoutgoingsobjects):-
  getalloutgoings(TailListoutgoings,NewListoutgoingsobjects),
  object(_,Head2Listoutgoings,_,_,NameofFunction,_),
  append(NewListoutgoingsobjects,[NameofFunction],Listoutgoingsobjects).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getotappsystypefunctions([],[]).

getotappsystypefunctions([HeadListotappsystype|TailListotappsystype],Listot
appsystypefunctions):-

getotappsystypefunctions(TailListotappsystype,NewListotappsystypefunction
s),
  object(_,_,_,_,HeadListotappsystype,Listoutgoings),
  getalloutgoings(Listoutgoings,Listoutgoingsobjects),
append(NewListotappsystypefunctions,Listoutgoingsobjects,Listotappsystype
functions).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getotposfunctions([],[]).

getotposfunctions([HeadListotpos|TailListotpos],Listotposfunctions):-
  getotposfunctions(TailListotpos,NewListotposfunctions),
  object(_,_,_,_,HeadListotpos,Listoutgoings),
  getalloutgoings(Listoutgoings,Listoutgoingsobjects),
  append(NewListotposfunctions,Listoutgoingsobjects,Listotposfunctions).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

results([],List1).

results([Head|List1],List2):-
  member(Head,List2),
  results(List1,List2),!.

results([Head|List1],List2):-
  not(member(Head,List2)),
  write(Head),write(' does not have a manager'),nl,
  results(List1,List2).
```

APPENDIX C

Assigned_Model Script

```
assignedmodel1:-

object(AssignedModel,ObjectId,Listincomings,'OT_FUNC',Name,Listoutgoing
s),
  assignedmodel2(AssignedModel,Listincomings,Listoutgoings,Name).

assignedmodel2(AssignedModel,Listincomings,Listoutgoings,Name):-
  getincomings(Listincomings,Listinc,Name),
  getoutgoings(Listoutgoings,Listoutg,Name),
  append(Listinc,Listoutg,AllList),
  getAssignedObjects(AssignedModel,AssignedObjectsIds),
  convertIdsToNames(AssignedObjectsIds,AssignedObjectsNames),
  checkObjects(AllList,AssignedObjectsNames),
  fail.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

checkObjects([],AssignedObjectsNames):-!.

checkObjects([Head|Tail],AssignedObjectsNames):-
  not(member(Head,AssignedObjectsNames)),
  not(object(_,_,_,'OT_FUNC',Head,_)),!,
  write(Head),write(' can not be found in sub-model'),nl,
  checkObjects(Tail,AssignedObjectsNames).

checkObjects([Head|Tail],AssignedObjectsNames):-
  checkObjects(Tail,AssignedObjectsNames).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

convertIdsToNames([],[]).

convertIdsToNames([Head|Tail],AssignedObjectsNames):-
  convertIdsToNames(Tail,AssignedObjectsNames2),
  object(_,Head,_,_,Name,_),
  New = [Name],
  append(AssignedObjectsNames2,New,AssignedObjectsNames).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getAssignedObjects(AssignedModel,AssignedObjectsIds):-
  model(AssignedModel,_,_,AssignedObjectsIds).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getincomings([],[],Name).

getincomings([Head|Tail],List,Name1):-
  getincomings(Tail,NewList,Name1),
  object(_,_,_,TypeNum,Name,Listoutgoings),
  find([Head|Tail],Listoutgoings),
  New = [Name],
  append(NewList,New,List).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getoutgoings([],[],Name).

getoutgoings([[Head1,Head2]|Tail],List,Name1):-
  getoutgoings(Tail,NewList,Name1),
  object(_,Head2,_,TypeNum,Name,_),
  New = [Name],
  append(NewList,New,List).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

find([X|Tail1],[[X,Y]|Tail2]):-!.

find([Z|Tail1],[[X,Y]|Tail2]):-
  find([Z|Tail1],Tail2),!.

find([Z|Tail1],[]):-
  fail.
```

APPENDIX D

Assigned_Model_Name Script

```
assignedmodelname:-
  object(AssigedModel,_,_,'OT_FUNC',Name1,_),
  model(AssigedModel,_,Name2,_),
  not(Name1 = Name2),
  write(Name1),write(' is not the same with assigned sub-model
name'),nl,fail.
```

APPENDIX E

Rule_Must_Be_Used Script

```
rulemustbeused:-
  object(_,_,Listincomings,'OT_FUNC',NameMain,ListOutgoings),
  Listinc = [],
  Listoutg = [],
  getincomings(Listincomings,Listinc,NameMain),
  getoutgoings(ListOutgoings,Listoutg,NameMain),
  fail.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getincomings([],Listinc,NameMain):-
  check(Listinc,NameMain).

getincomings([Head|Tail],Listinc,NameMain):-
  object(_,_,_,TypeNum,Name,Listoutgoings),
  find([Head|Tail],Listoutgoings),
  New = [TypeNum],
  append(Listinc,New,NewListinc),
  getincomings(Tail,NewListinc,NameMain).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getoutgoings([],Listoutg,NameMain):-
  check(Listoutg,NameMain).

getoutgoings([[Head1,Head2]|Tail],Listoutg,NameMain):-
  object(_,Head2,_,TypeNum,Name,_),
  New = [TypeNum],
  append(Listoutg,New,NewListoutg),
  getoutgoings(Tail,NewListoutg,NameMain).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

find([],[[X,Y]|Tail2]).

find([X|Tail1],[[X,Y]|Tail2]):-!.
find([Z|Tail1],[[X,Y]|Tail2]):-
  find([Z|Tail1],Tail2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

check([],Name):-!.

check([Head|Tail],Name):-
  member(Head,Tail),!,
  delete(Tail,Head,NewTail),
  write(Name),write(' must use a rule connection for incoming or outgoing
objects'),nl,
  check(NewTail,Name).

check([Head|Tail],Name):-
  check(Tail,Name).
```

APPENDIX F

Have_Same_Name Script

```
havesamename:-
  object(_,_,Listincomings,'OT_FUNC',NameMain,ListOutgoings),
  Listinc = [],
  Listoutg = [],
  getincomings(Listincomings,Listinc,NameMain),
  getoutgoings(ListOutgoings,Listoutg,NameMain),
  fail.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getincomings([],Listinc,NameMain):-
  check(Listinc,NameMain).

getincomings([Head|Tail],Listinc,NameMain):-
  object(_,_,_,TypeNum,Name,Listoutgoings),
  find([Head|Tail],Listoutgoings),
  New = [Name],
  append(Listinc,New,NewListinc),
  getincomings(Tail,NewListinc,NameMain).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getoutgoings([],Listoutg,NameMain):-
  check(Listoutg,NameMain).

getoutgoings([[Head1,Head2]|Tail],Listoutg,NameMain):-
  object(_,Head2,_,TypeNum,Name,_),
  New = [Name],
  append(Listoutg,New,NewListoutg),
  getoutgoings(Tail,NewListoutg,NameMain).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

find([],[[X,Y]|Tail2]).

find([X|Tail1],[[X,Y]|Tail2]):-!.
find([Z|Tail1],[[X,Y]|Tail2]):-
  find([Z|Tail1],Tail2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

check([],Name):-!.

check([Head|Tail],Name):-
  member(Name,[Head|Tail]),!,
  write(Name),write(' has an incoming or outgoing with the same name'),nl.

check([Head|Tail],Name).
```

APPENDIX G

One_In_One_Out_Rule Script

```
oneinoneoutrule:-
  object(_,_,ListIncomings,'OT_RULE',Name1,[[Head1,Head2]|Tail]),
  getnumber(ListIncomings,X),
  getnumber([[Head1,Head2]|Tail],Y),
  X =:= 1,
  Y =:= 1,
  object(_,Head2,_,_,Name2,_),
  write('Rule outgoing from '),write(Name2),
  write(' has one incoming and one outgoing'),nl,fail.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getnumber([],0).

getnumber([Head|Tail],Number1):-
  getnumber(Tail,Number2),
  Number1 is Number2 + 1.
```

APPENDIX H

Example eEPC Prolog File

```
object(
'no',
'ObjDef.di----4-----p--',
['CxnDef.jj----4-----q--'],
'OT_EVT',
'Event3',
[]).

object(
'no',
'ObjDef.a3----5-----p--',
['CxnDef.hu----5-----q--','CxnDef.h5----5-----q--'],
'OT_RULE',
'OR rule',
[['CxnDef.hv----5-----q--','ObjDef.fu----4-----p--']]).

object(
'Model.k9----4-----u--',
'ObjDef.fg----4-----p--',
['CxnDef.if----4-----q--'],
'OT_FUNC',
'Function5',
[['CxnDef.jh----4-----q--','ObjDef.h7----4-----p--'],
['CxnDef.jj----4-----q--','ObjDef.di----4-----p--']]).

object(
'no',
'ObjDef.h0----4-----p--',
['CxnDef.j1----4-----q--','CxnDef.jb----4-----q--'],
'OT_RULE',
'OR rule',
[['CxnDef.ib----4-----q--','ObjDef.cq----4-----p--']]).

object(
'no',
'ObjDef.hs----4-----p--',
['CxnDef.ir----4-----q--'],
'OT_EVT',
'Event5',
[]).

object(
'no',
'ObjDef.fn----4-----p--',
[],
'OT_FUNC',
'Function51',
[['CxnDef.ix----4-----q--','ObjDef.gm----4-----p--']]).

object(
'no',
'ObjDef.ev----4-----p--',
[],
'OT_FUNC',
'Function52',
[['CxnDef.jf----4-----q--','ObjDef.gm----4-----p--']]).

object(
'no',
'ObjDef.cc----4-----p--',
['CxnDef.jl----4-----q--'],
'OT_EVT',
'Function7',
[]).

object(
'no',
'ObjDef.hl----4-----p--',
['CxnDef.7a----5-----q--'],
'OT_FUNC',
'Function7',
[['CxnDef.jl----4-----q--','ObjDef.cc----4-----p--']]).

object(
'no',
'ObjDef.gm----4-----p--',
['CxnDef.jf----4-----q--','CxnDef.ix----4-----q--'],
'OT_RULE',
'AND rule',
[['CxnDef.hz----4-----q--','ObjDef.eo----4-----p--']]).

object(
'no',
'ObjDef.x-----5-----p--',
['CxnDef.7z----5-----q--'],
'OT_RULE',
'OR rule',
[['CxnDef.7a----5-----q--','ObjDef.hl----4-----p--']]).

object(
'no',
'ObjDef.e3----4-----p--',
['CxnDef.ip----4-----q--'],
'OT_EVT',
'Event6',
[]).

object(
'no',
'ObjDef.fu----4-----p--',
['CxnDef.hv----5-----q--'],
'OT_FUNC',
'Function33',
[]).

object(
'no',
'ObjDef.h7----4-----p--',
['CxnDef.jh----4-----q--'],
'OT_EVT',
'Event4',
[]).

object(
'Model.jn----4-----u--',
'ObjDef.f2----4-----p--',
['CxnDef.il----4-----q--','CxnDef.rq----5-----q--'],
'OT_FUNC',
'Function3',
[['CxnDef.7z----5-----q--','ObjDef.x-----5-----p--'],
['CxnDef.jb----4-----q--','ObjDef.h0----4-----p--']]).

object(
'no',
'ObjDef.d4----4-----p--',
[],
'OT_FUNC',
'Function31',
[['CxnDef.hu----5-----q--','ObjDef.a3----5-----p--']]).

object(
'no',
'ObjDef.dw----4-----p--',
['CxnDef.in----4-----q--','CxnDef.j3----4-----q--','CxnDef.i3----4-----q--
','CxnDef.i1----4-----q--'],
'OT_FUNC',
'Function1',
[['CxnDef.r0----5-----q--','ObjDef.ko----5-----p--']]).

object(
'no',
'ObjDef.eo----4-----p--',
['CxnDef.hz----4-----q--'],
'OT_FUNC',
'Function53',
[]).

object(
'no',
'ObjDef.c5----4-----p--',
[],
'OT_POS',
'Position2',
[['CxnDef.il----4-----q--','ObjDef.f2----4-----p--']]).

object(
'no',
'ObjDef.f9----4-----p--',
[],
'OT_POS',
'Position3',
[['CxnDef.ij----4-----q--','ObjDef.g1----4-----p--']]).

object(
'no',
'ObjDef.cx----4-----p--',
[],
'OT_EVT',
'Event1',
[['CxnDef.i3----4-----q--','ObjDef.dw----4-----p--']]).

object(
'no',
'ObjDef.ko----5-----p--',
['CxnDef.r0----5-----q--'],
'OT_RULE',
'OR rule',
[['CxnDef.rq----5-----q--','ObjDef.f2----4-----p--'],
['CxnDef.r1----5-----q--','ObjDef.he----4-----p--']]).

object(
'no',
'ObjDef.cq----4-----p--',
['CxnDef.ib----4-----q--','CxnDef.j7----4-----q--'],
'OT_FUNC',
'Function4',
[['CxnDef.ih----4-----q--','ObjDef.g1----4-----p--'],
['CxnDef.if----4-----q--','ObjDef.fg----4-----p--']]).

object(
'no',
'ObjDef.eh----4-----p--',
[],
'OT_FUNC',
'Function32',
[['CxnDef.h5----5-----q--','ObjDef.a3----5-----p--']]).

object(
'no',
'ObjDef.he----4-----p--',
['CxnDef.j5----4-----q--','CxnDef.r1----5-----q--'],
'OT_FUNC',
'Function2',
[['CxnDef.j1----4-----q--','ObjDef.h0----4-----p--']]).

object(
'no',
'ObjDef.g1----4-----p--',
['CxnDef.ij----4-----q--','CxnDef.ih----4-----q--'],
'OT_FUNC',
'Function6',
[['CxnDef.it----4-----q--','ObjDef.db----4-----p--']]).

object(
'no',
'ObjDef.g8----4-----p--',
[],
'OT_APPL_SYS_TYPE',
'Application system type1',
[['CxnDef.j5----4-----q--','ObjDef.he----4-----p--'],
['CxnDef.j7----4-----q--','ObjDef.cq----4-----p--'],
['CxnDef.j3----4-----q--','ObjDef.dw----4-----p--']]).

object(
'no',
'ObjDef.gt----4-----p--',
[],
'OT_EVT',
'Event2',
[['CxnDef.i1----4-----q--','ObjDef.dw----4-----p--']]).

object(
'no',
'ObjDef.gf----4-----p--',
[],
'OT_POS',
'Position1',
[['CxnDef.in----4-----q--','ObjDef.dw----4-----p--']]).

object(
'no',
'ObjDef.db----4-----p--',
['CxnDef.it----4-----q--'],
'OT_RULE',
'AND rule',
[['CxnDef.ip----4-----q--','ObjDef.e3----4-----p--'],
['CxnDef.ir----4-----q--','ObjDef.hs----4-----p--']]).

model(
'Model.k9----4-----u--',
'MT_EEPC',
'Function_hata',
['ObjDef.fn----4-----p--','ObjDef.ev----4-----p--','ObjDef.eo----4-----p--
','ObjDef.gm----4-----p--']).

model(
'Model.jn----4-----u--',
'MT_EEPC',
'Function3',
['ObjDef.fu----4-----p--','ObjDef.a3----5-----p--','ObjDef.eh----4-----p--
','ObjDef.d4----4-----p--']).

model(
'Model.kv----4-----u--',
'MT_EEPC',
'example_eepc',
['ObjDef.fg----4-----p--','ObjDef.gf----4-----p--','ObjDef.x-----5-----p--','ObjDef.cq-
---4-----p--','ObjDef.ko----5-----p--','ObjDef.cx----4-----p--','ObjDef.hl----4-----p--
','ObjDef.gt----4-----p--','ObjDef.e3----4-----p--','ObjDef.g1----4-----p--
','ObjDef.f2----4-----p--','ObjDef.db----4-----p--','ObjDef.cc----4-----p--
','ObjDef.he----4-----p--','ObjDef.h0----4-----p--','ObjDef.h7----4-----p--
','ObjDef.dw----4-----p--','ObjDef.di----4-----p--','ObjDef.f9----4-----p--
','ObjDef.g8----4-----p--','ObjDef.hs----4-----p--','ObjDef.c5----4-----p--']).
```

APPENDIX I

Checklist-Based Reading Inspection Checklist

1. Complete

All items that are needed for the specification of the requirements of the solution to the problem have been included?

2. Correct

Each item in the requirements specification is free from error.

3. Precise, Unambiguous and Clear

Each item in the requirements specification is exact and is not vague, there is a single interpretation of each item in the requirements specification, the meaning of each item in the requirements specification is understood, and the specification is easy to read.

4. Consistent

No item in the requirements specification conflicts with another item in the specification

5. Relevant

Each item in the requirements specification is pertinent to the problem and its solution.

6. Testable

During program development and acceptance testing, it will be possible to determine whether the item in the requirements specification has been satisfied.

7. Traceable

Each item in the requirements specification can be traced to its origin in the problem environment.

8. Feasible

Each item in the requirements specification can be implemented with the techniques, tools, resources and personnel that are available within the specified cost and schedule constraints.

9. Free of Unwarranted design Detail

The requirements specifications are a statement of the requirements that must be satisfied by the problem solution and they are not obscured by proposed solutions to the problem.

10. Manageable

The requirements specifications are expressed in such a way that each item can be changed without excessive impact on other items.

APPENDIX J

Defect-Based Reading Defect Classes Scenario

A defect in a requirements document is an omission, inaccuracy, inconsistency, ambiguity or anything that would lead to an unsatisfactory solution of the problem to be solved. It can fall into any of the following classes:

Omission (O)

Necessary information about the system for me to do my job has been omitted from the requirements document/functional specification.

Ambiguous Information (A)

Information within the requirements document/functional specification is inconsistent or ambiguous with other information.

Incorrect fact (I)

Some sentence contained in the requirements document/functional specifications asserts a fact that cannot be true under the condition specified in the requirements document/functional specifications.

Extraneous(E)

Information is provided that is not needed or used.

Miscellaneous (M)

Other defects

APPENDIX K

Perspective-Based Reading Test Based Scenario

For each requirement or functional specification (item), make up a test or set of tests that will allow you to ensure that the implementation satisfies the requirement. Use your standard test approach and test criteria to make up the test suite. For each requirement or functional specification, ask yourself the following questions:

1. Do you have all the information necessary to identify the item being tested and to identify your test criteria? Can you make up reasonable test cases for each item based upon the criteria?

2. Is there another requirement or functional specification for which you would generate a similar test case but would get a contradictory result?

3. Can you be sure that the test you generated should yield the correct value in the correct units?

4. Are there other interpretations of this requirement that the implementor might make based upon the way the requirement or functional specification is defined? Will this effect the tests you make up?

5. Does the requirement or functional specification make sense from what you know about the application or from what is specified in the general description?

APPENDIX L

Perspective-Based Reading Design Based Scenario

Given the requirements or functional specification generate a design using your standard design method. In so doing, ask yourself the following questions:

1. Are all the necessary objects (data, data structures, and functions) defined?

2. Is there sufficient information to specify the interfaces e.g., do the inputs of one function link to the outputs of the previous function?

3. Can all data types be defined e.g., are the required precisions and units specified?

4. Is all the necessary information available to do the design? Are all the conditions involving all objects specified is a requirement or functional specification missing?

5. Are there any points in which you are not clear about what you should do, because either the requirement or functional specification is not clear, not consistent or open to multiple interpretations?

6. Is there anything in the requirements or functional specifications that you can not design e.g., an infeasible constraint?

7. Does the requirement or functional specification make sense from what you know about the application or from what is specified in the general description/introduction?

APPENDIX M

Perspective-Based Reading User Based Scenario

Assume you are generating a user's manual for this system. Define the set of functions that the user should be able to perform. Define the set of input objects necessary to perform each function and the set of output objects that are generated by the function. This may be viewed as writing down as many operational scenarios or subsets of operational scenarios that the system should perform as possible. Start with the most obvious or nominal operational scenarios and proceed to the least common functions or special/contingency conditions. For each operational scenario ask yourself the following questions:

1. Is there anything that prevents you from writing this operational scenario?

2. Are all the functions necessary to write this operational scenario specified in the requirements or functional specifications e.g., are all the capabilities listed in the general description specified?

3. Are the initial conditions for starting up this operational scenario clear and correct?

4. Are the interfaces well defined and compatible e.g., do the inputs of one function link to the outputs of the previous function?

5. Are the effects of the operational scenario specified in the requirements or functional specifications under all possible circumstances?

6. Might some portion of the operational scenario give different answers depending on how a requirement or functional specification is interpreted?

7. Does the requirement or functional specification make sense from what you know about the application or from what is specified in the general description?

8. Can you get into a state of the system that must be avoided e.g. for reasons of safety or security?

APPENDIX N

Aris Dtd File

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT AML (Header-Info, Language+, Prefix*, Database?, User*,
UserGroup*, FontStyleSheet*, FFTextDef*, OLEDef*, Group, Delete*)>

<!ELEMENT Header-Info EMPTY>
<!ATTLIST Header-Info
        CreateTime NMTOKEN #IMPLIED
        CreateDate NMTOKEN #IMPLIED
        DatabaseName CDATA #IMPLIED
        UserName CDATA #IMPLIED
        ArisExeVersion (61 | 62) #REQUIRED
>

<!--General elements, used by several other elements-->

<!ELEMENT Prefix (#PCDATA) >
<!ATTLIST Prefix
   Default (YES | NO) "NO"
>

<!ELEMENT Blob (#PCDATA)> <!-- Base64 encoded binary data -->

<!ELEMENT Flag (#PCDATA)>
```

```
<!ELEMENT GUID (#PCDATA)>
<!ELEMENT FilterGUID (#PCDATA)>
<!ELEMENT MasterGUID (#PCDATA)>

<!ELEMENT Pen EMPTY>
<!ATTLIST Pen
        Color NMTOKEN #REQUIRED
        Style NMTOKEN #REQUIRED
        Width NMTOKEN #REQUIRED
>

<!ELEMENT Brush EMPTY>
<!ATTLIST Brush
        Color NMTOKEN #REQUIRED
        Style NMTOKEN #REQUIRED
        Hatch NMTOKEN #REQUIRED
>

<!ELEMENT Size EMPTY>
<!ATTLIST Size
        Size.dX NMTOKEN #REQUIRED
        Size.dY NMTOKEN #REQUIRED
>

<!ELEMENT Position EMPTY>
<!ATTLIST Position
        Pos.X NMTOKEN #REQUIRED
        Pos.Y NMTOKEN #REQUIRED
>
<!--End: General elements-->

<!-- BEGIN: Database -->
```

```
<!ELEMENT Database (AttrDef+)>
<!-- END: Database -->

<!--Begin: Language-->

<!ELEMENT Language (LanguageName?, LogFont?)>
<!ATTLIST Language
        Language.ID ID #IMPLIED
        LocaleId NMTOKEN #REQUIRED
        Codepage CDATA #REQUIRED
>

<!ELEMENT LanguageName (#PCDATA)>

<!ELEMENT LogFont EMPTY>
<!ATTLIST LogFont
        FaceName CDATA #REQUIRED
        Height NMTOKEN #REQUIRED
        Width NMTOKEN #REQUIRED
        Escapement NMTOKEN #REQUIRED
        Orientation NMTOKEN #REQUIRED
        Weight NMTOKEN #REQUIRED
        Italic (YES | NO) "NO"
        Underline (YES | NO) "NO"
        StrikeOut (YES | NO) "NO"
        CharSet NMTOKEN #REQUIRED
        OutPrecision NMTOKEN #REQUIRED
        ClipPrecision NMTOKEN #REQUIRED
        Quality NMTOKEN #REQUIRED
        PitchAndFamily NMTOKEN #REQUIRED
        Color NMTOKEN #REQUIRED
>
<!--End: Languge-->
```

```
<!--Begin of User-Definition-->

<!ELEMENT User (GUID?, AttrDef+, FilterGUID*, Prefix?)>
<!ATTLIST User
        User.ID ID #REQUIRED
        isSystem (true | false) "false"
        Passwd NMTOKEN #IMPLIED
>
<!--End: User-->


<!--Begin of UserGroup-Definition-->
<!ELEMENT UserGroup (GUID?, AttrDef+, FilterGUID*, Prefix?)>
<!ATTLIST UserGroup
        UserGroup.ID ID #REQUIRED
        User.IdRefs IDREFS #IMPLIED
>

<!--End: UserGroup-->

<!--Begin of Font-Definition-->

<!ELEMENT FontStyleSheet (GUID?, AttrDef*, FontNode+)>
<!ATTLIST FontStyleSheet
        FontSS.ID ID #REQUIRED
>

<!ELEMENT FontNode EMPTY>
```

```
<!ATTLIST FontNode
        LocaleId NMTOKEN #REQUIRED
        FaceName CDATA #REQUIRED
        Height NMTOKEN #REQUIRED
        Width NMTOKEN #REQUIRED

        Escapement NMTOKEN #REQUIRED
        Orientation NMTOKEN #REQUIRED
        Weight NMTOKEN #REQUIRED
        Italic (YES | NO) "NO"
        Underline (YES | NO) "NO"
        StrikeOut (YES | NO) "NO"
        CharSet NMTOKEN #REQUIRED
        OutPrecision NMTOKEN #REQUIRED
        ClipPrecision NMTOKEN #REQUIRED
        Quality NMTOKEN #REQUIRED
        PitchAndFamily NMTOKEN #REQUIRED
        Color NMTOKEN #REQUIRED
>

<!--End: Font-Definition-->

<!ELEMENT ExtCxnDef (GUID?, AttrDef*, ExtCxnDef*)>
<!ATTLIST ExtCxnDef
        ExtCxnDef.ID ID #REQUIRED
        ExtCxnDef.Type NMTOKEN #REQUIRED
        ToDef.IdRef IDREF #REQUIRED
   Reorg (DELETE|NODELETE) "DELETE"
>


<!ELEMENT CxnDef (GUID?, AttrDef*, ExtCxnDef*)>
```

```
<!--Format for CxnDef.Type: CxnBaseType or FromObjType.CxnBaseType.ToObjType-->
<!ATTLIST CxnDef
        CxnDef.ID ID #REQUIRED
        CxnDef.Type NMTOKEN #REQUIRED
        ToObjDef.IdRef IDREF #REQUIRED
   Reorg (DELETE|NODELETE) "DELETE"
>

<!ELEMENT ObjDef (GUID?, MasterGUID?, SymbolGUID?, AttrDef*, CxnDef*, ExtCxnDef*)>
<!ATTLIST ObjDef
        ObjDef.ID ID #REQUIRED
        TypeNum NMTOKEN #REQUIRED
        LinkedModels.IdRefs IDREFS #IMPLIED
        ToCxnDefs.IdRefs IDREFS #IMPLIED
   Reorg (DELETE|NODELETE) "DELETE"
   SubTypeNum NMTOKEN #IMPLIED
   SymbolNum NMTOKEN #IMPLIED
>
<!--End: ObjDefs-Definition-->
<!--Begin: Attribute Definition-->
<!ELEMENT AttrValue (#PCDATA)>
<!ATTLIST AttrValue
        LocaleId NMTOKEN #REQUIRED
>
<!ELEMENT AttrDef (AttrValue+)>
<!ATTLIST AttrDef
        AttrDef.ID ID #IMPLIED
        AttrDef.Type NMTOKEN #REQUIRED
>
<!--End: Attribute Definition-->
```

```
<!ELEMENT SymbolGUID (#PCDATA)>

<!--Begin: ObjOcc-Definition-->
<!ELEMENT ObjOcc (SymbolGUID?, Pen?, Brush?, Position?, Size?,
CxnOcc*, AttrOcc*, ExtCxnOcc*)>
<!ATTLIST ObjOcc
        ObjOcc.ID ID #REQUIRED
        ObjDef.IdRef IDREF #REQUIRED
        ToCxnOccs.IdRefs IDREFS #IMPLIED
        Zorder NMTOKEN #IMPLIED
        SymbolNum NMTOKEN #REQUIRED
        Active (YES | NO) "YES"
        Shadow (YES | NO) "NO"
        Visible (YES | NO) "YES"
   Hints NMTOKEN #IMPLIED
>
<!--End: ObjOcc-Definition-->

<!--Begin: FFText-Definition-->
<!ELEMENT FFTextOcc (Position?)>
<!ATTLIST FFTextOcc
        FFTextOcc.ID ID #IMPLIED
        FFTextDef.IdRef IDREF #REQUIRED
        FontSS.IdRef IDREF #IMPLIED
        SymbolFlag (TEXT | SYMBOL | ATTRNAME |
ATTRNAME_AND_SYMBOL | POSTIT | SYMBOL_AND_POSTIT |
ATTRNAME_AND_POSTIT | ATTRNAME_AND_SYMBOL_AND_POSTIT)
#REQUIRED
        Alignment (LEFT | CENTER | RIGHT) "LEFT"
   Zorder NMTOKEN #IMPLIED
>
```

```
<!--End: FFText-Definition-->

<!ELEMENT AttrOcc EMPTY>
<!ATTLIST AttrOcc
        AttrOcc.ID ID #IMPLIED
        AttrTypeNum NMTOKEN #REQUIRED
        Port (CENTER | N | NE | E | SE | S | SW | W | NW | NONE |
UPPER_MIDDLE | LOWER_MIDDLE | PORT_FREE) #REQUIRED
        OrderNum NMTOKEN #REQUIRED
        Alignment (LEFT | CENTER | RIGHT) "LEFT"
        SymbolFlag (TEXT | SYMBOL | WIDTH_ATTR_NAME |
ATTR_NAME_AND_SYMBOL) #REQUIRED
        FontSS.IdRef IDREF #IMPLIED
   OffsetX  NMTOKEN #IMPLIED
   OffsetY  NMTOKEN #IMPLIED
>


<!ELEMENT ExtCxnOcc (Pen?, Position*, AttrOcc*, ExtCxnOcc*)>
<!ATTLIST ExtCxnOcc
        ExtCxnOcc.ID ID #REQUIRED
        ExtCxnDef.IdRef IDREF #REQUIRED
        ToOcc.IdRef IDREF #REQUIRED
        Zorder NMTOKEN #IMPLIED
        Active (YES | NO) "YES"
        Diagonal (NO | YES) "NO"
        Visible (YES | NO) "YES"
   Hints NMTOKEN #IMPLIED
>


<!--Begin: CxnOcc-Definition-->
<!ELEMENT CxnOcc (Pen?, Position*, AttrOcc*, ExtCxnOcc*)>
```

```
<!ATTLIST CxnOcc
          CxnOcc.ID ID #REQUIRED
          CxnDef.IdRef IDREF #REQUIRED
          ToObjOcc.IdRef IDREF #REQUIRED
          Zorder NMTOKEN #IMPLIED
          Active (YES | NO) "YES"
          Diagonal (NO | YES) "NO"
          Visible (YES | NO) "YES"
   Hints NMTOKEN #IMPLIED
>
<!--End: CxnOcc-Definition-->

<!--Begin: Lane-Definition-->
<!ELEMENT Lane (GUID?, Pen?, Brush?, AttrDef*)>
<!ATTLIST Lane
          Lane.ID ID #IMPLIED
          Lane.Type NMTOKEN #REQUIRED
          Orientation (VERTICAL | HORIZONTAL) #REQUIRED
          StartBorder NMTOKEN #REQUIRED
          EndBorder NMTOKEN #REQUIRED
>
<!--End: Lane-Definition-->


<!ELEMENT OLEDef (GUID?, Blob, Blob)> <!-- first blob is Metafile-BLOB;
second blob is Data-BLOB -->
<!ATTLIST OLEDef
          OLEDef.ID ID #REQUIRED
   Link CDATA #IMPLIED
>

<!ELEMENT OLEOcc (Position?, Size?)>
<!ATTLIST OLEOcc
```

```
   OLEOcc.ID ID #IMPLIED
   OLEDef.IdRef IDREF #REQUIRED
   Zorder NMTOKEN #IMPLIED

>

<!ELEMENT FFTextDef (GUID?, AttrDef+)>
<!ATTLIST FFTextDef
        FFTextDef.ID ID #REQUIRED
        IsModelAttr (TEXT | MODELATTR) "TEXT"
>
<!ELEMENT Group (GUID?, AttrDef*, Group*, (ObjDef | Model)*)>
<!ATTLIST Group
        Group.ID ID #REQUIRED
>

<!ELEMENT Polygon (Position*)>
<!ATTLIST Polygon
   FillStatus (FILLED | TRANSPARENT) "TRANSPARENT"
>

<!ELEMENT RoundedRectangle (Position)>
<!ATTLIST RoundedRectangle
   Shaded (YES | NO) "NO"
>

<!ELEMENT GfxObj (Pen?, Brush?, Position?, Size?, (Polygon |
RoundedRectangle))>
<!ATTLIST GfxObj
   GfxObj.ID ID #IMPLIED
   Zorder NMTOKEN #IMPLIED
>
```

```
<!ELEMENT Union (Union*)>
<!ATTLIST Union
   OLEObjOccs.IdRefs IDREFS #IMPLIED
   ObjOccs.IdRefs IDREFS #IMPLIED
   Gfxs.IdRefs IDREFS #IMPLIED
   TextOccs.IdRefs IDREFS #IMPLIED
   Zorder NMTOKEN #IMPLIED
>

<!--Begin: Model definition-->
<!ELEMENT Model (Flag?, GUID?, MasterGUID?, Lane*, AttrDef*, ObjOcc*,
FFTextOcc*, GfxObj*, OLEOcc*, Union*)>
<!ATTLIST Model
        Model.ID ID #REQUIRED
        Model.Type NMTOKEN #REQUIRED
        AttrHandling    (OVERLAP    |    RESIZESYM    |    BREAKATTR    |
SHORTENATTR) #IMPLIED
        CxnMode (ONLYVERTICAL | ANGULAR) #IMPLIED
        GridUse (NO | YES) #IMPLIED
        GridSize NMTOKEN #IMPLIED
        Scale NMTOKEN #IMPLIED
        PrintScale NMTOKEN #IMPLIED
        BackColor NMTOKEN #IMPLIED
   CurveRadius  NMTOKEN #IMPLIED
   ArcRadius  NMTOKEN #IMPLIED
>
<!--end of Modeldefinition-->

<!-- PCDATA is GUID of object that should be deleted -->
<!ELEMENT Delete (#PCDATA)>
<!ATTLIST Delete
        Type        (GROUP|MODEL|OBJDEF|USER|USERGROUP|CXNDEF)
#REQUIRED
```

APPENDIX O

Dtd Content Model Rules

This information is from an xml reference book named “The Complete Reference XML” by [23].

1. A | B
(Either A or B occurs, but not both)

2. A, B
(Both A and B occur, in that order)

3. A&B
(Both A and B occur, in any order)

4. A?
(A occurs zero or one time)

5. A*
(A occurs zero or more times)

6. A+
(A occurs one or more times)

Valid Types Of Content For Xml Elements

1. EMPTY

(Specifies that this element can contain no content whether that content is text or child element)

```
<! ELEMENT IMAGE EMPTY>
```

2. ANY

(Specifies that this element can contain any content whether that content is text, child elements or a combination of both)

<! ELEMENT ADDRESSBOOK ANY>

3. MIXED CONTENT

(Allows you to specify the exact content you wish the element to contain. You can specify only text data or a combination of text and specified child elements.)

<! ELEMENT ADDRESSBOOK (NAME | NICKNAME | #PCDATA)>

4. CHILDREN

(Specify child element(s) that can be found within the body of the identified element. This content can't contain any character data.)

<! ELEMENTCONTACT (NAME.STREET, CITY, PHONE)>

Attribute Data Types

1. CDATA

It allows the attribute to contain any string of text characters.

2. ENTITY, ENTITIES

The entity allows to link external unparsed entities that reference external binary files into the document. The entities is the same as the entity type, but it allows to reference multiple entities.

3. ENUMERATED

It allows to specify a list of possible text values for the attribute. Each value must be separated by a vertical bar or pipe symbol.

4. İD

It allows to identify a single element uniquely within the document. The nature of the id attribute type prohibits the use of the same name over multiple elements.

5. IDREF, IDREFS

It allows to refer to a previously used id attribute type value.

6. NMTOKEN, NMTOKENS

It is used to restrict the values of the attribute to well formed xml names. These are text strings that start with either a letter or underscore character, contain only letters , numbers or the underscore character, and do not have any white white space within them.

7. NOTATION

It allows the attribute to reference the name of a notation that has been declared within the xml document. Notations are used to identify the format used with non-xml information such as video, audio or image files.

Attribute Default Value Keywords

1. #REQUİRED

It allows to force the existence of the attribute within the document.

2. #IMPLIED

It is optional. it is used to allow, but not require, the existence of a particular piece of information.

3. #FIXED

It is used to set a default value for an attribute that is also the only value that is available far that attribute.