Requirements Management Metrics

Vanita Shroff

Master's in Software Engineering Comprehensive Project Report

Department of Electrical and Computer Engineering,

University of Calgary, Calgary,

Alberta, CANADA

© Vanita Shroff, 2001

Committee for the Final Exam on
20th December, 2001

Dr. Armin Eberlein: ________________________

Dr. Y.Wang: ______________________________

Dr. Rob Kremer: ___________________________

Abstract

In the last few decades, software projects have encountered major difficulties. Most software engineering projects tend to be late and over budget. Several of the causes of these failures are related to requirements engineering issues such as requirements creep, poorly documented requirements, requirements that were impossible to satisfy, and requirements that failed to meet the needs of the user. Good requirements management practices help improve customer satisfaction, lower the system development costs, and increase the chance of having successful project. Requirements metrics, when incorporated in requirements management, assist in analyzing the quality of requirements and identifying the reasons for software reengineering/failure. Requirements metrics define the output measures of the software process.

Some recent reports indicate that the project success rate has slightly increased over the last years. This success is a result of defining a process and use of tools like requirements management tools. There are several requirements management tools available in the market. These tools focus on information management aspects of requirements management namely traceability and organization. However, they also offer extended capabilities of collecting requirements metrics. The aim of this project is to analyze one such requirements management tool - DOORS for its metric collection capabilities and identify how to implement a requirements engineering process using those metrics and DOORS.

Table of Contents

Chapter 1: Introduction 7

1.1 Objective 7

1.2 Project Structure 9

Chapter 2: Literature Review 10

2.1 Requirements 10

2.2 Requirements Engineering 18

2.3 Requirements Traceability 20

2.4 Requirements Management 24

2.5 Requirements Metrics 29

2.6 Formal Methods in Requirements Engineering 33

2.7 UML in Requirements Engineering 35

Chapter 3: Requirements Management Tools 38

3.1 RequireIt (Telelogic AB) 40

3.2 RequisitePro (Rational Software Corp.) 43

3.3 RTM (Integrated Chipware) 46

3.4 SLATE (SDRC) 49

3.5 DOORS and DOORSNet (Telelogic AB) 52

Chapter 4: DOORS Tool 57

4.1 DOORS and DOORSNet Architecture 57

4.2 Collecting Metrics using DOORS 60

Chapter 5: Requirements Engineering Process Using Metrics 73

5.1 Requirements Elicitation 73

5.2 Requirements Analysis 74

5.3 Requirements Specification 75

5.4 Requirements Management 76

Chapter 6: Conclusions and Recommendations 78

References: 81

Appendix A – Tool Comparision 86

Appendix B – Telelogic Vendor Report 87

Appendix C – DXL Scripts 89

Appendix D – Sample Output 109

Appendix E – Date Data Input File 115

List of Figures and Tables

Figure 1: Relative Cost of Fixing Errors in Project Lifecycle 10

Figure 2: Abstraction of Requirements [Source: Weigers, 1999].................................... 18

Table 1: RM Tools Classified Based on Type of Database Used.................................... 40

Figure 3: Input Formats Supported by DOORS [Source: Telelogic] 53

Figure 4: DOORSNet and DOORS Architecture – University of Calgary 57

Table – 2: Access Rights in DOORS depending on User Type 59

Figure 5: List Returned by Setting Filter for “shall” 61

Chapter 1: Introduction

1.1 Objective

"*Experience makes a man perfect"*. Most engineering projects come in on time and within budget because of the years of experience gained from developing detailed designs. For example, the contractor has little flexibility of changing the specifications when building a dam using this *frozen* design. The scenario is different with software engineering projects. There are no laws to govern the quality or quantity of software *per se*, to which the software should conform. The risk increases with the ever-changing competitive technology and change in user needs. As per the Standish Group [Standish Group *Chaos* Report, 1995], an alarming 31.3% of software projects are canceled before they get completed and 52.7% of projects cost 189% of their original estimates. These projects fail at the cost of millions of dollars, thousands of jobs, and sometimes with the loss of life. A tragic example of this is London's automated ambulance transportation system that left patients waiting for about half an hour in the emergency callers queue [Grinter, 1996].

The Software Engineering Institute's (SEI) Capability Maturity Model (CMM) for software describes that such failures happen in organizations that do not have an objective basis for determining software cost, software schedule, or for judging software quality. In general, these factors can be attributed to requirements engineering issues such as requirements creep, poorly documented requirements, requirements that were

impossible to satisfy, and requirements that failed to meet the needs of the user. Proper management of requirements may form the basis for estimating, planning, and tracking the project progress and may reduce the chances of such failures.

Requirements management has evolved as an important aspect of the software development process and has been an area of research in recent years. In broad perspective, requirements management involves information storage, organization, traceability, analysis, visualization, change management and documentation. SW-CMM (Software Capability Maturity Model) Level 2 specifies Requirements Management as a KPA (Key Process Area). The purpose of Requirements Management is to establish a common understanding between the stakeholders on the requirements that will be implemented in the software project [Paulk et. al., 1993]. The agreement forms the basis for estimating, planning, performing, and tracking service level delivery and project progress. Good requirements management practices can lead to higher customer satisfaction, lower system development costs, and increase the chance of having successful software.

One aspect of good requirements management practices is to measure and collect requirements metrics, rather than relying on gut feeling. Requirements metrics assist in analyzing the quality of requirements [Rosenberg et. al., 1998] and identify the reasons for software re-engineering/failure. While requirements metrics can help in understanding and improving requirements management, implementing a metrics program is a challenge. Several requirements management tools are available today on the market.

These tools require a high degree of knowledge for potential application and use of the tool. One such tool in the market is DOORS (Dynamic Object Oriented Requirements System) from Telelogic AB. DOORS is a requirements management suite that can assist managers, developers and end-users. It currently has more than 30% of the industry market and is recognized by Standish Group [Standish Group's Report, 2000] as world leader in Requirements Management. The purpose of this project is to analyze the DOORS tool, and assess the degree of metrics collection capability offered by automated tools like DOORS.

1.2 Project Structure

Chapter 2 describes the State of the Art. It consists of background information on requirements engineering, requirements management, and requirements metrics.

Chapter 3 describes various requirements management tools available in the market.

Chapter 4 introduces the DOORS tool that will be used for this project, describes its requirements metrics collection capability.

Chapter 5 gives a brief outline of how to use these metrics during the requirements engineering process with the DOORS tool.

Chapter 6 concludes the project. It concentrates on how this project satisfied the aim and objectives and provides recommendations for further research.

Chapter 2: Literature Review

2.1 Requirements

According to the waterfall model [Pressman, 2001], the software lifecycle typically consists of five phases. **Phase-1**: Requirements analysis and specification, **Phase-2**: Design, **Phase-3**: Implementation, **Phase-4**: Testing, and **Phase-5**: Maintenance. Phase-1 software requirements define the business functions, the design attributes, and the performance characteristics of an application. Recent surveys suggest that 44% to 80% of all defects are inserted in the requirements phase [Eberlein, 1997]. On the other hand, as shown in Figure-1, the cost required to fix an error later in the life cycle increases exponentially: it costs 5-10 times more to repair errors during coding phase and between 100-200 times more during maintenance phase than during the requirements phase [Moore, 2000; Wiegers, 1999].

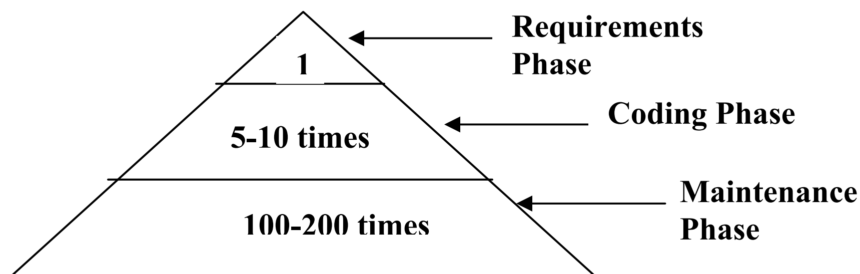

Figure 1: Relative Cost of Fixing Errors in Project Lifecycle

Inspite of known statistics it is not uncommon to hear the managerial attitude of "There's no time to stop for gas, we're already late". Organizations that start the design and implementation of the project before the requirements are analyzed, in order to release the product as soon as possible, may produce poor software.

2.1.1 Background

There are many different definitions for "Requirements". Sommerville I. and Sawyer [Sommerville and Sawyer, 1997] specify, "Requirements are the description of how the system should behave, or of a system property or attribute. They are capabilities and objectives to which software must conform. Or in other words, they are constraints on the development process and describe (1) User-level facilities (2) General system properties (3) Constraints on the system and on the development of the system". Sawyer P. [Sawyer, 2001] defines it as "A property that the software must exhibit in order for it to adequately perform its function". The IEEE Standard 729 defines it as [Davis, 1993]:

1. "A condition or capability needed by a user to solve a problem or achieve an objective".
2. "A condition or capability that must be met or possessed by a system – to satisfy a contract, standard, specification, or other formally imposed document."

In simple words, Requirements are *"something that the stakeholders want the software to do or to conform to"*. Typically requirements start out abstractly. Hence,

identifying this *something* that the stakeholders want is the most challenging part. Identifying the requirements right the first time determines the success of a project.

The complete description of *what* the software will do has to be documented. This document is known as the **Software Requirements Specifications (SRS)** document and should be non-ambiguous, complete, consistent, and traceable. Wieringa R.J. [Wieringa, 1995] defines SRS as "The description of the objectives that a product must satisfy, and the observable behavior that the product must have in order to fulfill these objectives". SRS is often part of the contract between the customer and vendor. SRS aids in estimating cost, planning team activities, performing tasks, and tracking the team's progress throughout the development activity [Paulk et. al., 1993]. SRS is sometimes developed in two distinct steps: the preliminary requirements specification and the final requirements specification. The former is generally the one that often provides the base for a contract between customer and vendor. The latter is the document that supports the design effort.

2.1.2 Importance of Requirements

Project failures can be attributed to two types of thinking: (1) those who thought but never did and (2) those who did and never thought. Ariane Flight 501 is a typical example of the second type of failure in software requirements process. On June 4, 1996, the maiden flight of Europe's Ariane 5 rocket ended in catastrophic failure with a complete destruction of the rocket and its payload. The cause was a software error, perhaps the most expensive software error on record. The root cause of this

error was not in the software design or coding processes. The root cause of the error was a failure due to breakdown in the requirements process. Within the requirements process, there were problems with elicitation, analysis, and verification and validation [CRSIP, 1999]. The software process was carefully organized and planned. But a 10-year old software module from Ariane 4 was reused without having precise specifications. This is a typical example of failures in the software industry attributed to " those who did and never thought".

Field T. [Field, 1997] describes another instance where a project failed due to unclear requirements. In 1995 Galileo International's Denver-based unit began work on project Agile that was budgeted at $400,000 and was scheduled to be completed in just a matter of months. However, the project was cancelled in late 1996 due to no definition of scope, no clear deliverables, and no customer focus. The project was then restarted with a new development team by identifying users and setting system requirements. The project was then up and running in six weeks.

These real-world examples identify the need and importance of project requirements. Requirements establish a shared vision, goals, and expectations. They are necessary to ensure that functionality built is really what the stakeholders want. They define achievable quality expectations and help in removing errors at the early stage. Requirements form the foundation of the software development process. Just as a weak building foundation causes the whole building to collapse, poorly written requirements statements collapse the whole software project.

2.1.3 Characteristics of Requirements

To ensure that the customer gets what s/he wants, requirements must exhibit certain characteristics. Desired characteristics of good requirements are: (1) Necessary – the stated requirement is essential (2) Concise - the requirements statement includes only one requirement stating what must be done (3) Implementation free - the requirement states what is required, not how the requirement should be met (4) Attainable - the stated requirement can be achieved by one or more developed system concepts at a definable cost (5) Complete - the stated requirement is complete and does not need further amplification (6) Consistent - the stated requirement does not contradict other requirements (7) Unambiguous - each requirement has one and only one interpretation (8) Verifiable – every requirement should be verifiable either through inspection, analysis, demonstration or test.

Requirements should have unique identifiers that can be continuously used throughout the software development process. These are known as the attributes of a requirement. Attributes yield significant information about the state of the system. Based on the attributes of requirements, queries can be made on the status of requirements. These attributes can help to plan, communicate, track the project's activities, and collect requirements metrics throughout the project lifecycle. Some of the requirements attributes are: requirement source, requirement authors, requirement rationale, requirement version number, requirement relative

importance, an assignee (to whom the requirement is assigned in the organization), comments, status, time it was created, due date by which the requirement must be provided, method of verification (qualification type to be used to verify that a requirement has been met), relationships to other requirements, and test number (identification number of method of verification). Davis and Leffingwell [Davis and Leffingwell, 1999] emphasize that as each project has unique needs, proper selection of the requirements attributes is critical to the success of the project.

2.1.4 Classification of Requirements

Sawyer P. [Sawyer, 2001] classifies requirements on the basis of levels at which requirements are formulated: (1) **User Requirements** and (2) **System requirements**. He defines users as the people who use or otherwise have a stake (stakeholders) in the software. User requirements define the results and qualities the user wants, and are gathered from the users of the system or product. The deliverable at the end of the user requirements phase is the User Requirements Document (URD) that forms the basis of software project acceptance. System requirements consist of user requirements, requirements of other stakeholders (such as regulatory authorities) and requirements that do not have an identifiable human source (e.g. some services of the system or act to constrain the system). The requirements engineer elaborates the system requirements into a number of more detailed and rigorously defined requirements that more precisely describe what the software must do. These are the '**Software Requirements**'. Software requirements provide an abstract model that shows that it solves every part of the problem.

Software requirements define what the software must do to achieve system requirements and must all trace back to the system requirements, which made them necessary.

Bahill and Dean [Bahill and Dean, 1999] further classify system requirements into two: (1) **Mandatory requirements** and (2) **Preference requirements**. They define mandatory requirements as the necessary and sufficient conditions that a minimal system must have in order to be acceptable (usually expressed with "shall" and "must"). According to them mandatory requirements have the scoring of only pass or fail (must not use scoring functions such as 1 for highly preferred functionality, 2 for good to have functionality, and 3 for not necessary, but can be an add-on in next release functionality) and are not susceptible to trade-offs between requirements. They define preference requirements as conditions that would make the customer happier (often expressed with "should" and "want"). According to them the preference requirements use scoring functions to produce figures of merit and are evaluated with a multicriteria decision technique, as none of the feasible alternatives would likely optimize all the criteria (susceptible to trade-offs).

Sommerville and Sawyer [Sommerville and Sawyer, 1997] classify requirements on a qualitative basis into (1) **Functional requirements** and (2) **Non-functional requirements**. According to them, functional requirements specify a function that the software system is capable of performing. These are statements of services that the system should perform. For example, functional requirements might state that a

system must provide some facility for authenticating the identity of a system user. Non-functional requirements on the other hand are not functions, but qualities or behavior that users want. Such requirements "constrain" the system within acceptable operational bounds. For example, a non-functional requirement might state that the authentication process should be completed in four seconds or less. The non-functional requirements are linked to several functional requirements. However, expressing non-functional requirements quantitatively is more important and challenging than expressing the functional requirements.

Sommerville and Sawyer [Sommerville and Sawyer, 1997] identify three categories of requirements: (1) **Product requirements** – related to performance, reliability, usability, and portability of the system (2) **Process requirements** – related to standards, programming languages etc. for the system (3) **External requirements** – related to interoperability, cost etc. of the system. Requirements are best verified when expressed quantitatively.

Hence, the classification of requirements depends on the level of abstraction at which the organization views its project. Figure-2 identifies one such abstraction of requirements.

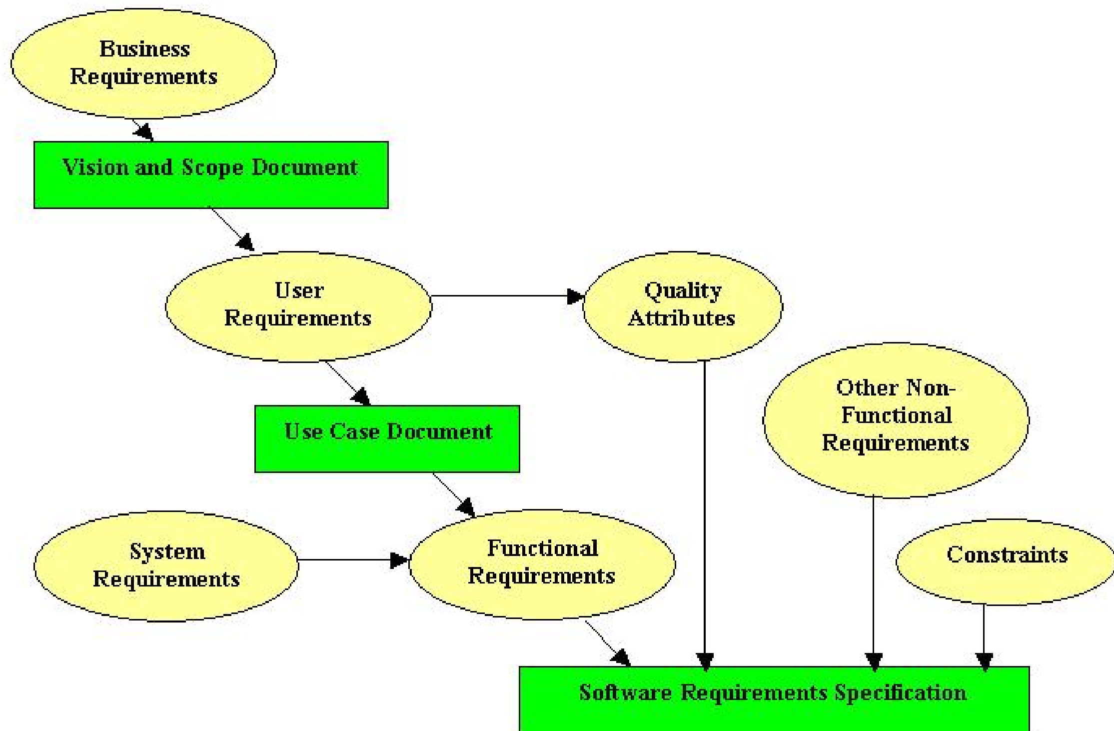

Figure 2: Abstraction of Requirements [Source: Weigers, 1999]

2.2 Requirements Engineering

Kotonya and Sommerville [Kotonya and Sommerville, 1997] define requirements engineering as "The systematic process of eliciting, understanding, analyzing, and documenting the requirements". Macaulay L.A. [Macaulay, 1996] defines it as "The process of defining what needs to be designed rather than how it is to be designed". But, as identified by Brooks F.P. [Brooks, 1987] "The hardest single part of building a software system is deciding what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems.

No part of the work so cripples the resulting systems if done wrong. No other part is more difficult to rectify later".

Often management fails to allocate the necessary resources required for the requirements engineering process. Allocation of sufficient time and resources on requirements phase may define the success or failure of a software project. The more the time spent on the requirements engineering process, the less time is required for the whole development process since less time is spent on rework. The proper management of the requirements engineering process can improve and accelerate the whole project development. Organizations can well plan their available budget by following minimalist processes that force to do just enough requirements management for the problem. Various surveys suggest that for large hardware/software systems, 15% [Kotonya and Sommerville, 1997] of the total budget is taken up by the requirements engineering activities and 50% [Embury, 2001] of the total budget is taken up by the software maintenance activities. Proper allocation of sufficient budget on requirements engineering may lower the maintenance budget.

The requirements engineering process can be defined as consisting of five aspects: requirements elicitation, requirements analysis, requirements specifications, requirements validation, and requirements management. Capturing of user requirements and analyzing them forms the first phase of the requirements engineering process. Once captured, these requirements are then categorized and prioritized in the requirements analysis phase. Categorization helps group

requirements into logical entities for planning, reporting, and tracking. Prioritization establishes the relative importance and risk of each requirement to help effectively manage the project. User requirements further lead to more detailed system requirements and development constraints on the system. At the requirements specification stage, the information collected during requirements elicitation is structured into a set of functional and non-functional requirements for the system. However, the customers cannot always specify accurate and complete requirements at the start of the process. Requirements change and cost or schedule constraints are placed on a development process. Capturing requirements is necessary, but the real challenge is to keep them current. Removing obsolete requirements, adding new ones, and modifying others are part of a never-ending process during the software development lifecycle. It then becomes extremely important to maintain traces from each requirement to its more abstract predecessor requirements and to its more detailed successor requirements. Traceability aids in assessing the impact of changes and is fundamental to the requirements management process (Requirements traceability is discussed in detail in Section 2.3). Requirements management on the other hand, ensures that changes are maintained throughout the software development lifecycle. Requirements Management is discussed in detail in Section 2.4.

2.3 Requirements Traceability

Gotel and Finkelstein [Gotel and Finkelstein, 1993] define requirements traceability as "The ability to describe and follow the life of a requirement, in both forward and

backward direction (i.e. from its origin, through its development and specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration in any of these phases)". This implies that traceability allows a requirement to be traced to its initial source as well as to its ultimate implementation throughout the entire life cycle of the project. In general, a requirement is traceable if it is possible to identify who suggested the requirement, why the requirement exists, what requirements are related to it, and how that requirement relates to other information such as system designs, implementation, and user documentation.

Traceability assists in assessing the impact of change in requirement. A single requirement change may affect several design elements. As well, a single design modification can sometimes affect several requirements. Traceability assists in planning changes, estimating efforts in changes, determining schedules, making changes, tracing through effects of changes, determining the affected people, sending change notifications, reducing the risk of failure of the project, and supporting project co-ordination.

Traceability has been classified into various categories by different researchers. Davis A. [Davis, 1993] classifies traceability into the following four categories:

- ❖ Backward-from traceability – linking requirements to their sources in other documents or people.

- Forward-from traceability – linking requirements to the design and implementation components.
- Backward-to traceability – linking design and implementation components back to requirements.
- Forward-to traceability – linking other documents (which may have preceded the requirements document) to relevant requirements.

In practice, the traceability information commonly maintained during requirements management is requirements-to-requirements traceability. Requirements-to-requirements traceability allows following the requirement trail from the source to the complying elements and generating documents as an output of the process rather than using the documents as the heart of the process. However, Davis A. [Davis, 1993] does not consider requirements-to-requirements traceability as a category for traceability.

A widely accepted classification is the one given by Gotel and Finkelstein [Gotel and Finkelstein, 1993]. They classify traceability into two categories: Pre-RS (requirements specification) and Post-RS (requirements specification). As per them, Pre-RS traceability is concerned with those aspects of a requirement's life prior to its inclusion in the requirements specification. And Post-RS traceability is concerned with those aspects of a requirement's life from its inclusion in the requirements specification.

In general traceability answers the following questions [ASEA Report, 1994]:

- What is the impact of changing a requirement?
- Where is a requirement implemented?
- Are all requirements allocated?
- What mission need is addressed by a requirement?
- Why is this requirement here?
- Is this requirement necessary?
- What design decisions affect the implementation of a requirement?
- Why is the design implemented this way and what were the other alternatives?
- Is the implementation compliant with the requirements?
- Who stated this requirement?
- Who changed this requirement and why?
- What acceptance test will be used to verify a requirement?
- Is this design element necessary?
- How do I interpret this requirement?
- Are we done?

The costs and efforts associated with requirements traceability result in better quality of the product, and the systems development and maintenance process becomes easy with potentially lower life cycle costs. However, currently requirements traceability is complex and costly overhead for companies. First of all, there is no comprehensive model of what information should be captured and used as part of traceability scheme [Ramesh et. al., 1995]. Moreover, it requires

integrated tool support and a project team that thoroughly understands the software development process.

2.4 Requirements Management

"A recent survey of 300 projects found that the main causes of failures were management and requirements issues. Five out of the top eight issues were related to poor handling of requirements. None was a technical issue" [Stevens, 1996]. It has become common to hear from customers' statements like this: "I paid the money but did not get what I wanted." Unsatisfactory products are mainly due to undefined requirements or requirements management policies. As per AnalystPro [AnalystPro], a software development company, many software project cost overruns and schedule delays are attributed to poor requirements management. Good requirements management practices help increase customer satisfaction and lower the system development costs.

There are many different ways in which Requirements Management has been defined in literature. Each definition covers a different scope and a different level of abstraction. Requirements management is a broad term and its scope needs to be specified for a specific project.

Grinter R.E. [Grinter, 1996] defines requirements management as "A process of assessing the impact of a change before it is made, identifying and managing the multiple versions of items which a change generates, rebuilding derived elements

after source elements are changed, and keeping track of all the changes that are made to a system". This definition limits the scope of requirements management to change impact analysis, configuration management and requirements traceability.

Eberlein A. [Eberlein, 1997] defines requirements management as "The overall process of requirements development including information storage, organization, traceability, analysis, visualization, and documentation". This definition covers everything related to requirements, except for configuration management, as a part of requirements management process.

Satta J. [Satta, 1999] defines requirements management, as "The process of capturing and communicating the users' needs to all the team members at the start of a project and throughout its lifecycle, in a clear and concise way". As per him, requirements management is a process of proper communication within the team and it has the potential of dramatically increasing the success of a project.

Davis and Leffingwell [Davis and Leffingwell, 1999] define it as "The process of eliciting, documenting, organizing, and tracking changing requirements, and communicating this information across the project team to establish a common understanding (i.e. an agreement) between the customer and the software project of the customer's requirements". As per them, requirements management ensures that iterative and unanticipated changes are maintained throughout the project lifecycle.

Kotonya and Sommerville [Kotonya and Sommerville, 1997] define it as "The process of managing large amounts of information and ensuring that it is delivered to the right people at the right time". This is a very broad definition where the scope of “managing large amounts of information” is undefined.

CMM, KPA Level 2 specifies, “The purpose of Requirements Management is to establish a common understanding between the stakeholders on customer's requirements that will be addressed by the software project. It involves establishing and maintaining an agreement with the customer on technical and non-technical requirements [Paulk et. al., 1993]. The agreement forms the basis for estimating, planning, performing, and tracking service level delivery and project progress. It helps in detecting the requirements errors and avoiding unnecessary costs associated with implementing wrong requirements”. CMM defines requirements management as a process of communicating, planning, performing and tracking requirements. As CMM does not define how the process is achieved, it does not consider change management, configuration management, and impact analysis as part of requirements management. As per AnalystPro [AnalystPro, 2000] for every organization striving to reach CMM level 2, or simply trying to improve the way they do business, requirements management is one of the indispensable project management controls. However, the scope of requirements management needs to be defined by the organization.

A generalized definition would then be:

Requirements Management is a process of eliciting, documenting, analyzing the impact of change, tracing, storing, organizing, visualizing the requirements, identifying and managing multiple versions of items, and communicating the changed requirements to all the team members.

It is up to the software development team to limit the scope of requirements management for their project. With the remarkable speed at which projects are being developed, with sky-high expectations and ever-increasing exposure and risk, managing project requirements effectively and efficiently is key to delivering a successful product.

Grinter R.E. [Grinter, 1996] mentions an incidence where a project failed due to poor configuration management practices. The London Stock Exchange wanted to modernize its operations by introducing new technologies both within the exchange itself and in the organizations associated with it. The multi-million pound Taurus system project started with all the optimism of any new venture sold as revolutionizing an industry. However, in 1993, after years spent in development the head of the London Stock Exchange announced that the Taurus system had failed. A highly regarded firm of computer consultants, Ovum Consultancy, suggested that the failure of Taurus was a direct result of poor configuration management practices. Taurus was a highly distributed system; teams of developers worked together on individual parts of the projects. In software engineering terms, relationships exist between pieces of code; this relationship is called dependencies.

The project failed due to lack of coordination by the people working on *dependency codes*.

Until 1980's system engineers used tools such as pencil and paper to manage requirements for complex systems such as those of space, military, and medical systems. Requirements of complex systems are large in number and have many inter-dependency links. The tools like pencil and paper were time-consuming for maintaining large number of requirements and they did not allow system engineers to explore their capabilities of analysis and design. After the 1980's, system engineers started using word processors, spreadsheets, and databases to develop complex systems. With the improvement in computer technology, system engineers now use automated tools to manage requirements. There are various requirements management tools available in the market. Some of these requirements management tools, besides managing requirements, also allow collecting requirements metrics. As per Standish Group [Standish Group, 2001], 28% of the projects succeeded in 2000 – on time, on budget with all required features. This was up from 16% in 1994. As per them, this is a direct result of defining a process and use of system tools. DOORS, the requirements management tool being analyzed in this project, is one such requirements management tool that allows collection of requirements metrics.

2.5 Requirements Metrics

The literature review presented in Section 2.1 showed that the development of good requirements is essential to quality product design because even after having a well-written code, software based on inaccurate requirements will be unsatisfactory. Requirement metric analysis assists in analyzing the quality of requirements [Rosenberg et. al., 1998], getting the right requirements, and identifying the reasons for software re-engineering/failure. Metrics require measuring something to meet a specific goal. Measuring and collecting requirements metric rather than relying on gut feeling is characteristic of good requirements management practice. However, as reported by Asbrand D. [Asbrand, 1998], in 1998, fewer than 9% of companies used metrics to measure and monitor software development in the United States.

There are various metrics available to analyze the quality of requirements [Costello and Liu, 1995]. Some of these metrics are:

- ❖ Lines of text – measures the actual number of lines of text in the requirements document. This metric allows the user to estimate the functionality and degree of testing required for that software [Rosenberg et. al., 1998].
- ❖ Imperatives – measuring the actual number of imperatives in different categories like "shall", "must", and "will". The number of imperatives gives a rough estimate of degree of design functionality required in the software.

It also gives an estimate of degree of testing required to satisfy these imperatives.

- Continuances – measuring the actual number of continuances such as "the following" after the imperative. Continuances indicate that the requirements are hierarchically well structured.
- Weak Phrases – measuring the actual number of weak phrases such as "large", "fast", "enough", etc. Weak phrases indicate non-testable, and loose-design requirements.
- Completeness metrics – measuring the actual number of incomplete requirements such as "TBD"(to be determined) and "TBS" (to be specified). These are requirements that are still uncertain.
- Option phrases – measuring the actual number of optional phrases such as "can", "may", " I/we think" etc. These indicate possibly non-satisfiable requirements.
- Effort – measure of total effort input to manage the requirements.
- Volatility and change metrics – measuring the number of requirements added, deleted, and modified, classified by reason for change.
    - Number of initially allocated requirements – measure the number of initially allocated requirements. This includes all technical and non-technical requirements as originally provided by the customer. This metric along with the number of final allocated requirements as well as the number of changes per allocated requirements describes the level of requirements volatility [Jones, 1998].

- Number of finally allocated requirements - measure the number of finally allocated requirements. This includes all technical and non-technical requirements that were used to build the final software product. This metric along with the number of initially allocated requirements as well as the number of changes per allocated requirements describes the level of requirements volatility.
- Number of changes per requirement – track the number of changes made to each requirement. Besides describing the level of volatility of requirements, this metric also describes the impact of changing requirements on the software process.
- Number of changes in specific time period – number of changes per week, for example. This describes the degree of volatility of requirements. This number should decrease towards the end of the software lifecycle (indicating convergence of requirements).
- Cause of change – categorize the cause of changes. This helps in identifying the most common causes of change in the software process and can be used to improve the software process.
- Who requested the change – Identify the source of change, reason for implementing a specific functionality, and anticipate the source of changes in future.

❖ Traceability metrics – measures the number of requirements traced to/from each specification, number of requirements untraced, number of

requirements inconsistently traced, and number of upward and downward linkages for each requirement.

- The number of requirements that trace consistently to the next level up.
- The number of requirements that trace consistently to the next level down.
- The number of requirements that trace consistently to the next level in both directions.
- Depth coverage metrics for coverage to the lowest level of specification.
- Height coverage metrics for coverage to the highest level of specification.
- Inconsistent link coverage metrics that includes the number of requirements that have at least one inconsistent traceability link upwards or downwards.
- Untraceable linkage metrics that has no traceability links upward and downward.

❖ Correctness metrics – assesses the correctness of specified requirements. It is a subjective attribute and requires defect density metrics for analysis.

Some of the limitations of using metrics are:

❖ All these metrics do not have a same unit. It is difficult to compare these metrics on different unit scale. For example, what is lines of text?, what is the time unit?

- All these metrics may not have the same context. For example, number of changes in one year for one document may be 50 and for the other document it may be 100. But the 50 changes in the first document may have actually occurred in one day as against the 100 changes in the other document distributed equally over the time span of one year.
- It is difficult to interpret the raw metrics data. For example, one document may have 50 lines of text and the other may have 100 lines of text. The document with 50 lines of text may be written in very professional technical language, but a junior requirements engineer may write the other document of 100 lines of text. There is no objective means of comparing these two documents. These metrics need to be analyzed on a statistical scale for further inference.
- The most important question to answer after having this raw data is what do we want to do with the results? These results need to be further analyzed for example, to compare the productivity of software processes.

2.6 Formal Methods in Requirements Engineering

Formal methods are mathematical approaches to software and system development that support the rigorous specification, design and verification of computer systems. A formal specification is a specification expressed in a language whose vocabulary, syntax and semantics are formally defined. It cannot be based on natural language; it is based on mathematics. In the analysis stage of development user requirements are expressed in the natural language of the client (e.g. English). These are often

ambiguous and hard to analyze. Moreover, it is difficult to trace these requirements to the resulting software. Formal specifications avoid the speculations of meaning of phrases in imprecisely worded prose. Because the notations used in formal specifications are unambiguous, it can be used as part of the 'contract' between the customer and the development team. Hence, formal specifications improve the accuracy, understandability, and productivity of software development.

Advantages of formal specifications are:

- They can be easily analyzed for their correctness, completeness, consistency, and verifiability. Correctness here is the correctness of requirements and not correct requirements, i.e., validation of the requirements is still necessary to ensure that the requirements represent what the client wants.
- They can be used as a guide for creation of test cases for any particular component of the system.
- It is easy to establish links between the specifications and the resulting code.
- IBM estimated that formal methods reduced the number of problems per line of code by a factor of 60% [Deb, 1995].
- IBM also estimated that formal methods reduced the code production cost by 9% [Deb, 1995].

- In the INMOS T800 (now SGS-Thomson Microelectronics) project, formal methods uncovered the faults in the IEEE floating-point standard and in other hardware implementations used for testing purposes.
- Inmos estimated that the development work was completed in less than 50% of the time required for informal methods [Deb, 1995].

Disadvantages of formal specifications are:

- The notations are difficult to read and understand.
- It has a steep learning curve.
- The mathematical notation is hard for clients to understand and may intimidate the client.
- It is difficult to express constraints such as processing speed, reliability and efficiency, in some formal specification languages.
- It forces to express a significant amount of functionality in the early life cycle.
- It is expensive to implement.

2.7 UML in Requirements Engineering

The Object-Oriented (OO) approach can be used for identifying requirements (using use cases), Object-Oriented Analysis (OOA), Object-Oriented Design (OOD), and OO programming (OOP). The Unified Modeling Language (UML)

developed by James Rumbaugh, Grady Booch and Ivar Jacobson is one of the object modeling language that tailors this approach.

The biggest challenge in software development is that of building the right system. The difficult part of the software process is to identify these right requirements to build the right system. It requires a very good communication channel between the stakeholders of the system. Use cases provide the basis of communication between sponsors and developers in planning the project. Use-case techniques give the external picture of the system by explaining what the system will do and who are the stakeholders of the system. It gives a good understanding of who the stakeholders are and what they want. Use cases appear first in the requirements model and are used to generate a domain object model.

Developers have always used typical scenarios to try to understand what the requirements of a system are and how a system works. Use cases are a technique for formalizing the capture of these scenarios. Use cases are tools to acquire requirements from the users of a system as they correspond very closely to the users view of the functional requirements of the system. They can be used to capture the functional and certain non-functional requirements of the system. They also make requirements available for review. Reviewing allows the identification of misunderstood problems in the early requirements stage. This avoids any implementation bias in the requirements. Use cases can also be used to categorize the requirements, rank the requirements, and trace the requirements to implementation using automated tools. The biggest advantage of use cases as

compared to formal methods is that they are easy to review with the end-users of the system.

Use cases also partition the system into single atomic business functions, which may be used as a basis for costing the system, or for planning a phased system delivery. In this case each successive phase would deliver further batches of Use Cases. Because use cases are not expressed in descriptive language (they are expressed in diagrams), further information is still required, however, to tie down the detail of what each business function does.

After defining the use cases of the system and ranking them, the domain objects and classes can be identified. This leads to the conceptual model of the system.

Chapter 3: Requirements Management Tools

Requirements Management tools are software tools that facilitate the management of requirements. They focus on capturing requirements, managing, and producing requirements specifications. An ideal requirements management tool should [as per Literature Review in Section 2.3]:

- Support common functionalities such as: requirements identification, viewing and editing requirements, tracing requirements to their origin, change impact analysis, completeness and consistency checking, metrics collection, change control (keeping track of any additions, deletions, or changes to existing requirements), view requirement subsets (sort, filter, or query the database to view subsets of the requirements that have specific attribute values), and report generation.
- Should also assist in communicating requirements to all the team members of the system under development throughout the project lifecycle.
- Should be easy to configure to suit the organization's existing process.
- Should be based on commercially available, scalable and proven database technology. The use of commercial database allows the use of industry standard database query languages such as SQL and allows the use of various add-on utilities available from the commercial vendor. The use of commercial database is like using a testsed technology instead of reinventing the wheel.
- Should support heterogeneous operating platforms.
- Should support configuration management.

There are various requirements management tools available in the market. Most of these tools focus on the information management aspect of requirements management namely traceability and organization. These tools require a high degree of knowledge not only in the potential application of the tool, but also in the actual use of the tool base itself. Hammer and Huffman [Hammer and Huffman, 1998] describe an incidence where a NASA project demonstrated a common failing in the implementation of a requirement management tool. The failure was because the features of the tool were not properly understood and were not properly activated. The project consisted of groups. Each group was made responsible for the data in its own domain without consideration of how it related to other data sets. The result was that the common information remained contained in each class (class here is an organizational element). Hence, it is essential to understand the functionality of requirements management tools before using it.

Requirements Management tools are enablers that drive the existing requirements management process. They do not define the requirements management process. They are tools to assist existing requirements management processes. A common term associated with requirements management tools in the software industry is "silver bullet syndrome". The user becomes enamored with a particular tool and expects it to solve all the problems based on the marketing pitch of a particular vendor. However, the user does not realize that a tool is only a tool. If people write poor requirements, test badly, and haphazardly do impact assessment, the tool is not going to help. The tool cannot solve communication problems between the programmer, manager, marketing, etc. Any requirements management tool is only as good as the discipline of the people using it.

Requirements management tools have been classified into three categories for the purpose of this project, based on the type of database used to store requirements: Word-Processor based tools, relational database based tools, object-oriented database based tools [refer Table-1].

Word-Processor Based Tools	**Relational Database-Based Tools**	**Object-Oriented Database-Based Tools**
RequireIt	RequisitePro	DOORS
	RTM	SLATE

Table 1: RM Tools Classified Based on Type of Database Used

This chapter briefly describes the following requirements management tools: RequisitePro, RequireIt, RTM, SLATE, and DOORS. A comparison of these different tools is presented in Appendix-A.

3.1 RequireIt (Telelogic AB)

RequireIt by Quality Systems Software (now Telelogic AB) is a Requirements Management Tool, based entirely on Microsoft Word. It is a basic tool aimed at novice users. This tool provides the benefit to the user of utilizing an existing, familiar interface. The tool does not use any database to store requirements. Requirements are stored as a document with hyperlinks. The positive side of not leveraging the power of a database is that it eliminates the need for costly database

administration overheads. The negative side of not having a database is that the tool is only suitable for small projects with 500-1000 requirements per document.

Requirements can be captured in two ways by this tool: either by highlighting desired text and then marking it using the RequireIt toolbar, or automating the process by doing keyword searches on typical requirement words such as "shall". The captured requirement shows up as a highlighted item within Microsoft Word.

The tool allows storing the attribute information along with the requirements. The attributes are stored as either text, numeric, date, YES/NO, High/Medium/Low. How it actually does this, is proprietary in nature (as per vendor). For example, if there is a requirement, "The household shall receive audible confirmation that the main door is fully closed," one attribute that might be useful to have for this requirement is a Pass/Fail test. Highlighting the requirement, expanding its attributes list, and adding it can add this attribute to the requirement. Now this requirement can be tracked on its Pass/Fail status.

The tool also assists in change impact analysis by establishing links between documents (i.e. within the same project). For example, the above-mentioned user requirement can be linked to a software requirement contained in another document. Once this link is created, any changes made in this requirement can be traced to other affected requirements to ensure that there is no disconnect or lost requirement. As the tool is primarily MS-Word-based, it only supports uni-directional traceability with hyperlinks.

The tool has three reporting features: Requirements report, traceability report, and suspect requirement trace. The requirements report shows the marked-up requirements and their attributes. The traceability report shows the link relationships between requirements. The suspect requirement trace report shows the source and target requirement information for any suspect requirements in the current document.

The tool also provides templates designed specifically for the software development life cycle.

The advantages of the tool are:

- It is a simple tool based entirely on familiar user interface of MS-Word.
- It eliminates the need of database administration overheads.
- It assists in change impact analysis.
- The attribute information is stored along with the requirements.
- It has templates designed for software development life cycle.

The disadvantages of this tool are:

- It does not manage artifacts other than requirements.
- It does not support bi-directional traceability. The tool only supports uni-directional traceability for establishing relationships or links between artifacts in the repository. However, tracing is accomplished in either direction by a simple change in perspective in the view.

- It does not have a database with global view that can support sharing of artifacts across different projects.
- It does not support cross-project traceability.
- It does not support a change management process.
- It does not support configuration management.
- It does not provide multi-user requirements management, i.e. only one team member can work on the same information at the same time.
- It does not integrate with other testing, design and project management tools.
- It does not support any other file format except MS-Word.
- It has limitations of MS-Word: It does not allow manipulation of graphical objects and the parsing is imperfect unless the user is diligent about using text styles or keywords, such as "shall," when writing the requirements.

3.2 RequisitePro (Rational Software Corp.)

RequisitePro by Rational Software Corp. is also a requirements management tool that has the MS-Word-familiar front-end. However, unlike RequireIt, RequisitePro leverages the power of a relational database to allow requirements prioritization and organization. Many requirements documents are written in a word processor type of format. They are written out by putting a number of "shall", "will", and "must" type statements together. RequisitePro uses Microsoft Word as the primary user interface for managing requirements. The tool then automatically parses through the

document on user-specified keywords (such as "shall") and extracts the requirements from the document. Each requirement in the document is then stored as a tuple in the database. RequisitePro supports databases such as Oracle, Microsoft SQL Server, and Microsoft Access. The tool is aimed at novice users who have less experience in requirements management tools.

RequisitePro is build upon a simple conceptual model consisting of projects, documents, requirements, and attributes. Projects are top-level objects that serve as 'containers' for documents and are subject to revision management and archiving. Each project is maintained in a sub-directory and consists of a database file and the project documents. The database file has the information on requirements, attribute values, traceability relationships, requirement types, attribute definitions, document types, saved views, revision histories, security information, and user and group data. Project documents are word documents whose instances can contain product requirements, requirement specifications, use cases, test cases, or any other user-specified requirement types.

The advantages of RequisitePro are:

- It supports MS-Word file format and comma-separated value (CSV) file format.
- It has the ability to classify/categorize requirements during identification.
- It recognizes graphics and OLE objects as requirements.
- The tool supports a matrix view that provides visual feedback on what software requirements were derived from which system requirements.

- The tool supports traceability tree view that shows requirements in a hierarchical fashion. This view graphically shows relationships between requirements.
- The tool supports change impact analysis.
- The tool supports bi-directional traceability.
- The tool supports cross-project traceability.
- It partially supports team collaboration and configuration management by providing integration with other Rational software lifecycle tools such as Rational Rose for use-case modeling, Rational ClearQuest for change request, Rational Clearcase for version control, and Rational TestManager for test case creation. It also integrates with INTERSOLV PVCS Version Manager, Microsoft Source Safe, and Microsoft Project

The disadvantages of RequisitePro are:

- Graphics and OLE objects have MS-Word limitation where specific points in an object cannot be referenced and changes cannot be tracked.
- It has total lack of support for linking to external documents, graphs, spreadsheets and the like.
- Artifacts cannot be shared across projects, as RequisitePro does not have a global view of the repository.
- RequisitePro does not have the ability to handle code artifacts or to link to artifacts in non-Word documents.
- The tool is only suitable for small/medium-sized projects.

- The parsing is imperfect unless the user is diligent about using text styles or keywords, such as "shall," when writing the requirements.

3.3 RTM (Integrated Chipware)

RTM (Requirements and Traceability Management) from Integrated Chipware is a requirements management tool aimed at large integrated hardware/software systems. RTM supports both the requirements engineering and the full lifecycle traceability processes. As per the vendor, it supports the “Enterprise Engineering Process” where they define “Enterprise Engineering Process” as the process of taking an idea from 'concept to delivery'.

Requirements can be documented in RTM using word-processing packages from Adobe, Interleaf, and Microsoft. The documented requirements are then automatically captured into the RTM database by the tool by selecting requirements based on keywords, batch, paragraph, graphics, tables, etc. from source documents. RTM uses Oracle – a relational database, to store requirements. The database is held on a 'server' and 'client' computers have access to it over a network. RTM has a relational database transactional property where multi-users can simultaneously access the database with object-level security (has locking control). The access privileges can be controlled on a per-user basis down to the attribute level within an object. As RTM is oracle-based, it offers good data integrity and scalability. However, RTM must be configured, before use, to define the schema of the

database, i.e. define the types of requirements and the ways by which they will be linked together.

RTM supports the change management and the project management process. It allows the user to: (1) enter requirements into the database, (2) browse, (3) edit and manipulate the information; (4) assess the magnitude of change and do a full impact analysis (5) manage traceability of diverse objects (6) generate reports, and (7) generate documentation. The advantages of the tool are:

- RTM tool is available in all types of environments, on any network, and on over 90 platforms (as per the vendor).
- Requirements can be allocated and can be linked to a variety of project work products such as test cases, source code, and analysis and design specifications.
- System attributes such as document and paragraph ID, and user-defined attributes, can be associated with each requirement.
- The use of the commercial database Oracle extends support to industry standard database query languages such as SQL (Structured Query Language). Various utilities built around these SQL engines like PowerBuilder and Report Maker can be used with the RTM tool. Moreover, the commercial database vendor automatically provides heterogeneous support; with proprietary databases this has to be written in an ad hoc manner.
- The tool has a built-in configuration management system.

- The tool is integrated with popular desktop publishing interfaces such as Adobe FrameMaker, Microsoft Word, and Interleaf. This allows users to publish information in any desired format.
- The tool can be integrated with design tools such as Sterling Software's Teamwork and Object Team, Rational Software's Rose and ClearCase, Aonix's StP, Silicon Valley Networks' TestExpert, and I-Logix's Statemate.
- The oracle database is part of the tool.
- It supports sharing of artifacts across projects.
- It allows reuse of requirements from one successful project to another.

The disadvantages of the tool are:

- It limits the selection of database to Oracle.
- As Oracle is a relational database, RTM does not support object-oriented properties like inheritance.
- As per the vendor, the tool can be used for any project size. However, relational databases are normally more efficient for a large number of records that have only few links between them. The requirements databases have relatively few records (hundreds rather than hundreds of thousands) each of which includes many links such as links to documents, text files and other requirements. Maintaining these links is possible with a relational database but it is inefficient. It requires operations on several different tables. Hence, for very large numbers of requirements, relational databases may be too slow [Kotonya and Sommerville, 1997].

- ❖ RTM is the key market competitor of DOORS and is more expensive than DOORS.

3.4 SLATE (SDRC)

SLATE (System Level Automation Tool for Engineers) from SDRC is a requirements management tool suited for projects that involve both software and hardware components. It is suitable for hardware components too because it has special modules to support such situations. For example, the SLATE tool is suitable for embedded control systems that require managing software requirements such as how fast the software should operate and the hardware requirements such as cooling requirements.

The users can define their product with graphical building blocks called abstraction blocks. Requirements can be captured in Microsoft Word or Adobe Framemaker publishing system or in an ASCII text file. The SLATE Requirement Identifier examines the document's structure and creates a requirement for each sentence, paragraph, or numbered paragraph at the user's discretion. Each requirement is assigned as a requirement object in the database. A link is created between the source paragraph and its defined requirement. Requirements can also be parsed based on defined keywords.

As every requirement is an object, every requirement can have object-oriented properties like inheritance. Users can link these requirements to one or more

abstraction blocks and source documents. Accordingly, once a requirement is linked to an abstraction block in a hierarchy, the children of the abstraction block automatically inherit the requirement.

Attributes can be defined for all objects in the database when they are parsed/created. This allows requirements to be grouped into logical groups for easy categorizing, parsing, identifying, and analyzing the pool of requirements. Tables and activators can also be attached to these database objects. Moreover, a note object can be attached to any object in the database and can store typed in text, text loaded from external files and graphics or file references to graphics. Notes can be categorized as rationale, assumptions, decisions, phone calls, email, questions or answers. For example, every requirement can have attributes such as customer assumptions and requirements rationale associated with them.

The tool is built on the commercially available object-oriented client-server database – Versant. Versant is a multi-user database that allows storing requirements and all rationale, assumptions and decisions associated with requirements analysis. As Versant is a multi-user database, it provides persistence of design data, and provides object level locking during multi user access to the common database. The object-oriented database enables the tool to be used for large projects. The tool has been tested with 800,000 requirements per document.

SLATE consists of a number of modules. Each module is able to access the same SLATE database through different capabilities. Licenses need to be purchased for each module. The various modules of the SLATE product line are: SLATE

Architect, SLATE REquire, SLATE reVIEWer, SLATE VIEWer, SLATE Sim, SLATE Activator/Authoring, SLATE Sentry, and translate Web Access. More information on these modules is available from the vendor's website.

The advantages of the SLATE tool are:

- The tool allows the design teams to create a series of graphical building blocks to describe and build complex products on a systems basis.
- Customer requirements document paragraphs can be linked to derived design specifications.
- It supports document-to-document traceability and requirements-to-requirements traceability. Requirements-to-requirements traceability allows following the requirement trail from the source to the complying elements and generating documents as an output of the process rather than using the documents as the heart of the process.
- The tool provides folders for organizing information. For example, there can be a "Safety Requirements Folder" within an overall "Requirements Folder". This allows arranging folders and other SLATE objects in hierarchies.
- It supports change impact analysis.
- It can be integrated with SDRC product knowledge management suite Metaphase.
- Links are objects and are bi-directional.

- It provides team support. Many people can work on the same database at the same time from geographically different locations.
- It supports full product lifecycle.
- As projects usually start from an existing requirements document, SLATE supports importing a requirements document from Microsoft Word or Adobe FrameMaker.
- It supports bi-directional traceability.
- SLATE currently supports the following CASE environment interfaces: Structured Analysis Tools Teamwork SA/SD, Software-Thru-Pictures (STP), Object-Oriented Analysis Tools GD-Pro, Rational Rose, ObjecTeam, and ObjecTime.

The disadvantages of the SLATE tool are:

- It does not support document-to-document traceability.
- It uses object-oriented database. Object-oriented database does not provide data independency like the deductive object-oriented databases. Hence, business rules need to be encoded and duplicated in various application programs.

3.5 DOORS and DOORSNet (Telelogic AB)

DOORS (Dynamic Object Oriented Requirements System) is a requirements management suite based on a proprietary object-oriented database. The tool uses the

X-Windows (the DOORS client) user interface on the Intel PC architecture. It is a tool that addresses the specific needs of all the stakeholders: managers, developers, and end-users, throughout the project lifecycle. It is a tool that supports communication to reduce project risk, collaboration to increase productivity, and traceability to enable validation.

Requirements can be imported into the DOORS tool in any of the formats supported by the tool (see Figure-3). A programmable parser can then be used to extract requirements objects based on keywords, structure, unique identifiers, etc.

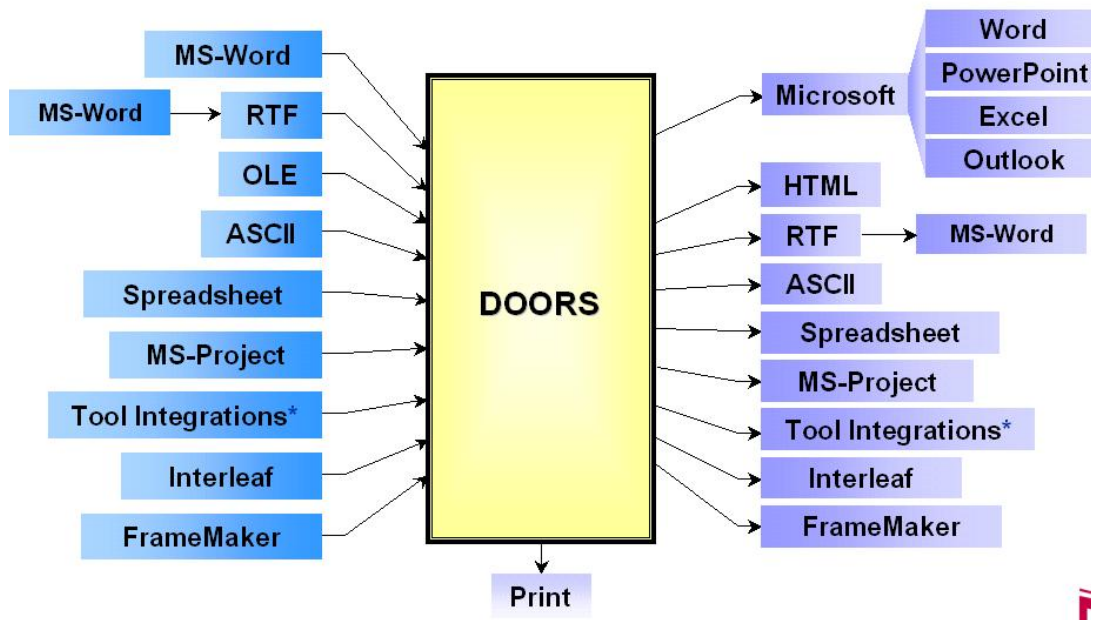

Figure 3: Input Formats Supported by DOORS [Source: Telelogic]

The tool uses proprietary object-oriented database to store data. The tools database allows storing more than 10,000 requirements per document making it suitable for

large projects that require requirements support throughout the whole lifecycle. The information in a DOORS database is stored in modules. Each module is like a document in a word processor or a worksheet in a spreadsheet program. Each module has items of data that are called objects. Objects can be a reference document, a paragraph, a sentence, a quantitative number, an entire table, or a single table cell. The objects have properties that express the characteristics of these objects. These are known as attributes. Objects have some pre-defined attributes such as object heading, object text and who has modified the object. The tool allows defining any number of additional attributes such as priority, cost, etc.

The tool allows: (1) capturing the requirements by directly entering them in DOORS, (2) parsing of requirements after importing files of the different formats mentioned in Figure-3 (3) managing requirements, (4) analyzing requirements and (5) tracing requirements.

The advantages of the tool are:

- It has the capability to classify/categorize requirements during identification.
- It supports multi-user access at the database, project, document, attribute, view, and object level.
- It allows bi-directional traceability.
- It allows change impact analysis. For example, user can trace the impact of a change to a single piece of data on the rest of the system.

- It allows baselining of modules. A baseline is a read-only version of a module. The baseline cannot be deleted until the module is deleted.
- It contains a complete change proposal system.
- It allows creating links between modules in different projects or folders and creating traceability reports that show how one project impacts another (allows cross-project traceability).
- DOORS includes a C-like scripting language for customizing or developing extensions to the basic product. For example, the user can create a DXL script to check which projects are not meeting their targets.
- It also includes direct interfaces to Microsoft Project, Teamwork, and Rational Rose, and the user can create interfaces to many other tools through an open API (Application Programming Interface).
- It has a partition-rejoin feature. For example, a partition can be created for each contractor, leaving a program manager with a locked read-only copy of the partitioned-out data. If the contractor does not have the DOORS client software, he can email a comma or tab separated value file representing a table of current progress on the requirements. The program manager can then use the DOORS spreadsheet import tool to load the data into appropriate formal module of the contractor.
- It allows having multiple views of the same project. So a program manager can have a different view than a developer.

- The tool provides an interface to a few formal methods toolkits. This allows DOORS to be used for safety critical systems.

The disadvantages of the tool are:

- Provides a (proprietary) single-database repository.
- As per the views of some DOORS tool users, DOORS configuration controls do not work well with high requirements churn i.e. if 70% or more of the requirements in a document are changing in short periods of time.
- There are limitations on the import and export of data. A character limit in Excel (Excel has a limit of 255 characters per cell) table cells can cause data loss, and exports of large DOORS data files to Word are slow.
- The tool requires training. Effective setup of the database with regards to attributes and traceability is necessary for proper storing of project data. Engineers also can get caught up in features and playing (reformatting, rearranging, customizing colors/fonts) without formal training.

DOORSNet is a web-based front end to DOORS. It allows using a web browser to access information stored in the DOORS database. Appendix-B shows that currently the web-based front-end DOORSNet supports few of the DOORS functionalities. As DOORSNet supports limited functionality, this project aims at exploring the DOORS tool for metric collection. Chapter 4 discusses the DOORS tool in detail.

Chapter 4: DOORS Tool

4.1 DOORS and DOORSNet Architecture

The DOORSNet and DOORS architecture as installed at the University of Calgary is shown in Figure-4. The figure describes the propagation of data in one direction. The response data is returned in the opposite direction.

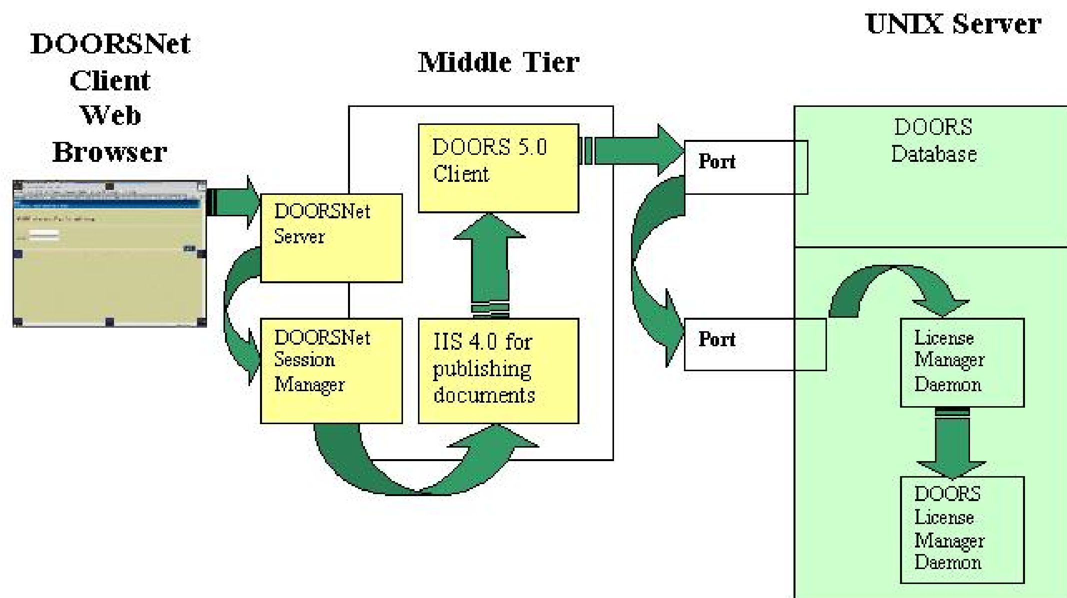

Figure 4: DOORSNet and DOORS Architecture – University of Calgary

The following steps describe the flow of data using the DOORSNet client. If using the DOORS client to access the database, skip steps 1 to 5.

1. The authenticated web client accesses the DOORSNet server through URL http://webserver-name/doorsnet using the standard web browser Internet Explorer 5.0 and higher or Netscape Navigator 4.7 and higher.

2. The DOORSNet Server component running in the middle tier services requests from web users and passes the request to the DOORSNet Session Manager.
3. The DOORSNet Session Manager component running in the middle tier, controls web user sessions.
4. The IIS (Internet Information Server) 4.0 running in the middle tier has the link to projects, modules, and views published by the DOORS administrator.
5. The IIS communicates with the DOORS 5.0 client installed in the middle tier.
6. The DOORS 5.0 client communicates with the DOORS database installed on the UNIX server. The database port first communicates with the license server to check the availability of licenses.
7. For a team working on projects over the network (different geographical areas), organizations prefer to buy floating licenses from the vendor. Floating licenses allow the specified number of users use the product at the same time. The users can be on any computers on the network. The FLEXlm license server from GLOBETrotter Inc. is used to manage the Telelogic DOORS vendor-issued floating licenses. Information about floating licenses is stored in a license file on the computer that the FLEXlm license server is running on. The license file has the information of port number on which FLEXlm license manager daemon is running. The license file also contains a list of FEATURE lines that describe features that the FLEXlm license server can provide floating license for. The FLEXlm license manager daemon starts, stops, and restarts the Telelogic DOORS vendor daemon. The Telelogic DOORS vendor daemon keeps track of how many licenses are being used and by whom.

8. If there is no valid license available, the Telelogic DOORS vendor daemon refuses the request, and the DOORS suite program fails to start, giving a message saying that it could not get a license.
9. If a license is available, DOORS client 5.0 fetches the user information from the DOORS database.
10. The users can have any of the following access rights authorized by the DOORS administrator:

Type of Users	Access Rights
Standard users	They can work with the DOORS database but cannot do any management tasks such as archiving data or creating new users.
Project Managers	They can partition and archive data, and create and manage groups. They cannot create new users. But they can create new groups, add users to groups, remove users from groups and so on.
Database Managers	They can do everything that Project Managers can do and can additionally create projects and users, and manage the database.
Custom Users	Can have any combination of the above rights.

Table – 2: Access Rights in DOORS depending on User Type

11. Depending on the access rights of the user, related projects, modules, and views are displayed in the user web browser.
12. If the user has the right to edit/modify the data, the DOORS 5.0 client communicates to server where the DOORS database is responding.
13. The DOORS database constantly communicates with license daemon installed on the server to check the validity of the floating license.

4.2 Collecting Metrics using DOORS

The DOORS tool has various on-screen menus that allow the listing of specific data.

a. Tools -> Filter – Filters lets you control what data is displayed or your screen. You can use them to filter out data you do not want to see. For example, you can filter out all objects except those that contain the word "steering". Or you can filter out all objects except those that have links. There are two types of filter, simple and advanced. With a simple filter, you can either filter on the contents of every attribute of type text or string, filter on the value of a single attribute of any type, filter on the basis of whether the object has links, filter on the basis of whether the object is either the current object or a leaf object. With an advanced filter, you can combine together simple filters to create a complex filter. For example, you can only display objects that contain the word "steering" and that also have links. You can also use the And, Or and Not buttons below the list of rules to define complex search criteria. The filter

menu provides the results in list format. For example, if 100 objects in the document contain “TBD”, then 100 objects are returned in the list format without their count in number (refer Figure-5).

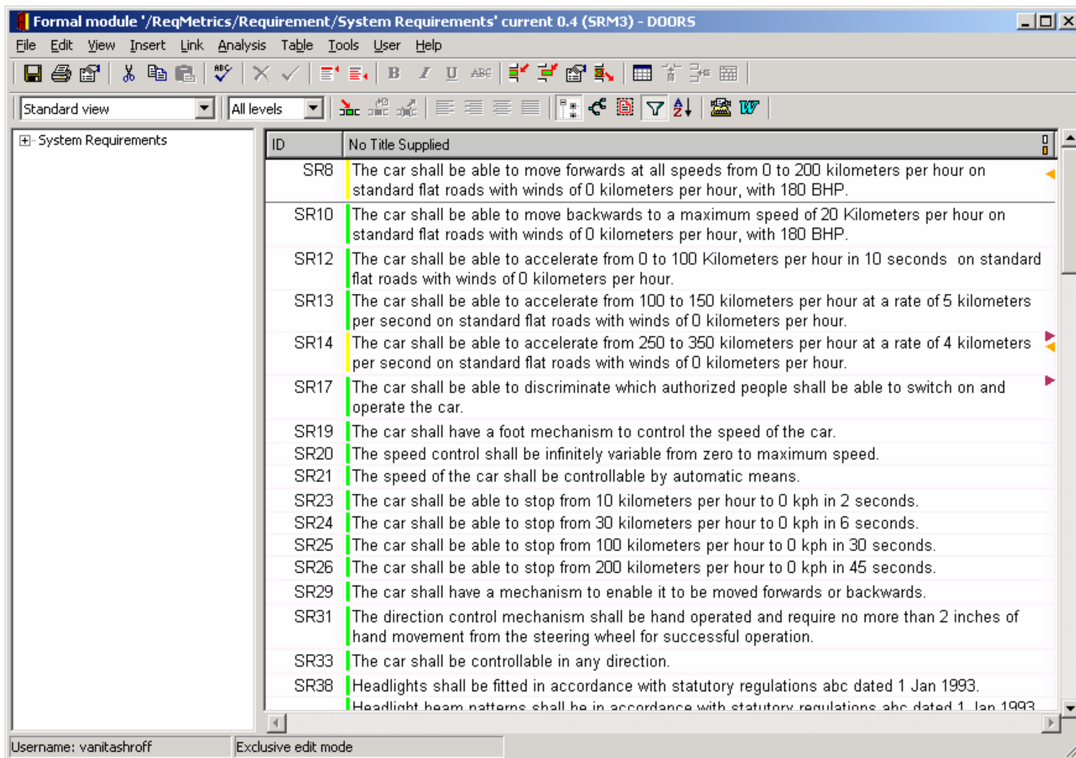

Figure 5: List Returned by Setting Filter for “shall”

b. Tools -> Sort – Allows sorting on basis of selected attributes in ascending or descending order. For example, you can sort on the attribute number of modifications in descending order. This will list all the objects with the object modified most number of times at the top of the list, and the object modified least number of times at the bottom of the list. However, this also returns the result in list form.

c. File -> Module Properties -> Statistics – On refreshing, this menu gives:

i. Total Number of objects in the module.
ii. Number of words in the Object Heading, Object Text and Object Short Text attributes.
iii. Number of characters in the Object Heading, Object Text and Object Short Text attributes.
iv. Number of words in other attributes of type text or string.
v. Number of characters in other attributes of type text and string.
vi. Total number of words in attributes of type text and string, including the Object Heading, Object Text and Object Short Text.
vii. Total number of characters in attributes of type text and string, including the Object Heading, Object Text and Object Short Text.

This menu returns the data in count format. However, the metrics returned by this menu have not been used in this project.

d. Tools –> Functions -> Statistics - In the Attribute box, select the attribute you want to look at. The frequencies of the attribute's values for all objects in the current view are displayed graphically.

As truly identified by Lord Kelvin (1889) *"When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind"*. Most of the in-built menus (a to d) return the results in list format, and not in numbers. There are certain vendors like

MetricCenter (2001) that offer add-on products to collect metrics. However, this project uses the DXL scripting library, instead of add-on products, to collect the required metrics.

Metric - 1: ***Lines of Text*** (Unit: Number of lines of Text/document). The script calculates the total count of lines of text based on the number of end of line (\n) and return characters (\r) (refer to Appendix C for script code). However, as each requirement is considered as an object in DOORS, this script returns the total number of objects instead of total number of end of line (\n) and return characters (\r) (refer to Appendix D for sample output).

Metric - 2: ***Imperatives to measure the actual number of "shall", "must", and "will"*** **(Unit: Number of Imperatives/document)**. This script calculates the total count of imperatives in this module (refer to Appendix C for script code). It uses the DXL function ***contains*** to check if the object contains the string imperatives (refer to Appendix D for sample output).

(refer to Appendix C script code for variables description)

```
int totalShall = 0
        .
        .
string s3
s3 = "shall"
Buffer b = create
        .
int iRetShall = contains(b, s3, offset)
        .
        .

while ( iRetShall != -1)
```

```
{
        totalShall++
        offset = iRetShall
        iRetShall = contains(b, s3, offset)
}
```

The disadvantage of finding imperatives with this script is that statements like "The car shall be compatible with all standard fuel supply mechanisms in the countries to which it will be sold." will return "will" as one imperative count. The advantage is that if any statement has two imperatives “shall” defined, it will return it as two counts instead of one.

Metric - 3: Metric to calculate the total number of continuances like "the following" **(Unit: Number of Continuances/document)**. This script calculates the total count of continuances in this module (refer to Appendix C for script code). It uses the DXL function ***contains*** to check if the object contains the string of continuances as identified in the script code (refer to Appendix D for sample output).

Metric - 4: Metric to calculate the total number of weak phrases like "large", "fast", "enough" **(Unit: Number of weak phrases/document)**. This script calculates the total count of weak phrases in this module (refer to Appendix C for script code). It uses the DXL function ***contains*** to check if the object contains the string of weak phrases as identified in the script code (refer to Appendix D for sample output).

Metric - 5: Metric to calculate the total number of incomplete sentences like "To be Determined" and "To be Specified" **(Unit: Number of incomplete sentences/document)**. This script calculates the total count of incomplete sentences (refer to Appendix C for script code). The script gives the total count of "To be Determined", "TBD", "To be Specified", and "TBS". It uses the DXL function ***contains*** to check if the object contains the string of words such as "TBD" as identified in the script code (refer to Appendix D for sample output).

Metric - 6: Metric to calculate the total number of option phrases like "can", "may", and "I think" **(Unit: Number of option phrases/document)**. This script gives the total count of option phrases (refer to Appendix C for script code). It uses the DXL function ***contains*** to check if the object contains the string of words such as "can" as identified in the script code (refer to Appendix D for sample output).

Metric - 7: Metric to calculate the total number of changes to an object. **(Unit: Total Number of changes to an object/document)**. This is a volatility and change metric that gives the total number of times each object has been modified (refer to Appendix C for script code). DOORS has an object attribute called **No. of Modifications**. This attribute returns the number of modifications to the object as recorded in the history. History records only those object changes that go through the change proposal process. Hence, if the object has been modified without going

through the change proposal process, this script does not consider it as a change to the object.

Metric - 8: Metric to calculate the total number of changes in a specific time period **(Unit: Total Number of changes /document/specified time period)**. This metric uses the ***Date*** function of the DXL library to compare the last modified date of the object to the dates passed in (start date and end date) (refer to Appendix C for script code). If the object has been modified ten times, the last modified date is the date of the 10th time. And this will show up as one modification in this metric. However, the previous metric (Metric-7) that shows the number of modifications to the object will show 10. Also, if that particular object has been modified 10 times, but the last modified date is greater than the end date passed in as parameter, it will not show up as modified object. For example, if the object has been last modified on 15th November, 2001 and we are calculating the number of changes between 10th November, 2001 to 12th November, 2001, this object will not be counted in this metric. As the DXL library does not have any function to input the date parameters through the console, this script reads the two date parameters (start date and end date between which the changes are made) from a text file d:\datedata.txt (refer Appendix-E for datedata.txt file).

Metric - 9: Metric to calculate the cause of change in a specific time period **(Unit: Count of total additions or modifications/document/specified time)**. This metric returns the total number of additions/modifications made in a specific time period

(refer to Appendix C for script code). It looks at the change proposal module for the cause of change. If the cause of change is modification, it goes through the history session of that module and prints the “who” attribute of that cause of change. Similarly if the cause of change is “addition”, it goes through the history session and prints the “who” attribute of that cause of change.

(refer to Appendix C script code for variables description)

```
if (s2 == "Modification")
{
        string sChange = identifier(o21)
        string sProposer
        HistorySession hs
        for hs in current Module do
        {
                sProposer =  who(hs)
        }
        print sChange " Modified by: " sProposer
        print  " --- " s3 "\n"
        iCauseOfChange++
}
```

This metric calculates only those objects in the change proposal system to which the proposed changes have been applied. If a change is proposed, but not approved or applied yet, then this piece of code does not consider it as a change.

The important thing to consider is that DOORS does not open any closed module required for the metrics count.. As the calculation of metrics requires data from the change proposal module and the change proposal module may not be open all the time, the following piece of code opens the change proposal module:

```
Module Proposals =
        read("/ReqMetrics/Change Proposal System/Proposals 1", false)
```

where ReqMetrics is the project that contains the Change Proposal System folder and Change Proposal System folder contains Proposals 1 module.

Metric - 10: Metric to calculate the total number of finally allocated requirements **(Unit: Total Number of Final Requirements/document)**. The total number of finally allocated requirements is the total count of “shall”, “will”, and “must” in the current open module (refer to Appendix C for script code). The DXL library requires that the current module be explicitly read (loaded in the memory) before performing any metric calculation on it. The script reads the current module as:

```
read("/ReqMetrics/Requirement/System Requirements", false)
```

and returns the total count of imperatives (“shall”, “will”, and “must”) (refer to Appendix D for sample output).

Metric - 11: Metric to calculate the total number of initially allocated requirements **(Unit: Total Number of Initial Requirements/document)**. The total number of initial requirements is the count of “shall”, “will”, and “must” in the first baseline document (refer to Appendix C for script code). The initial baseline of the document is read using the ***load*** function of the DXL library:

```
Module m = load(baseline(0,1,"SR"), false)
```

where,

0 – major version number of the baseline

1 – minor version number of the baseline

SR – baseline suffix

false – load the module but do not open it on display

After loading the baseline, the date of the baseline is obtained using the ***Date*** function of the library.

```
Baseline b = baseline(0,1,"SR")
Date d
d = dateOf(b)
```

If the date of the "Created On" attribute of the object in the current module is less than or equal to the date of baseline, and has any of the imperatives ("shall", "will", "must") then it is counted as initial requirement (refer to Appendix D for sample output).

Metric - 12: Metric to calculate number of incoming links in the module **(Unit: Total Number of Incoming Links /document)**. To calculate the total number of incoming links into the module, all the modules that have direct or indirect (at different levels) link to this module have to be open (refer to Appendix C for script code). For example, in Figure - 6, all the documents such as User Requirements Document, System Requirements Document, Architecture Document, CI Structure Document, and Test Document should be open before counting the incoming links to User Requirements Document.

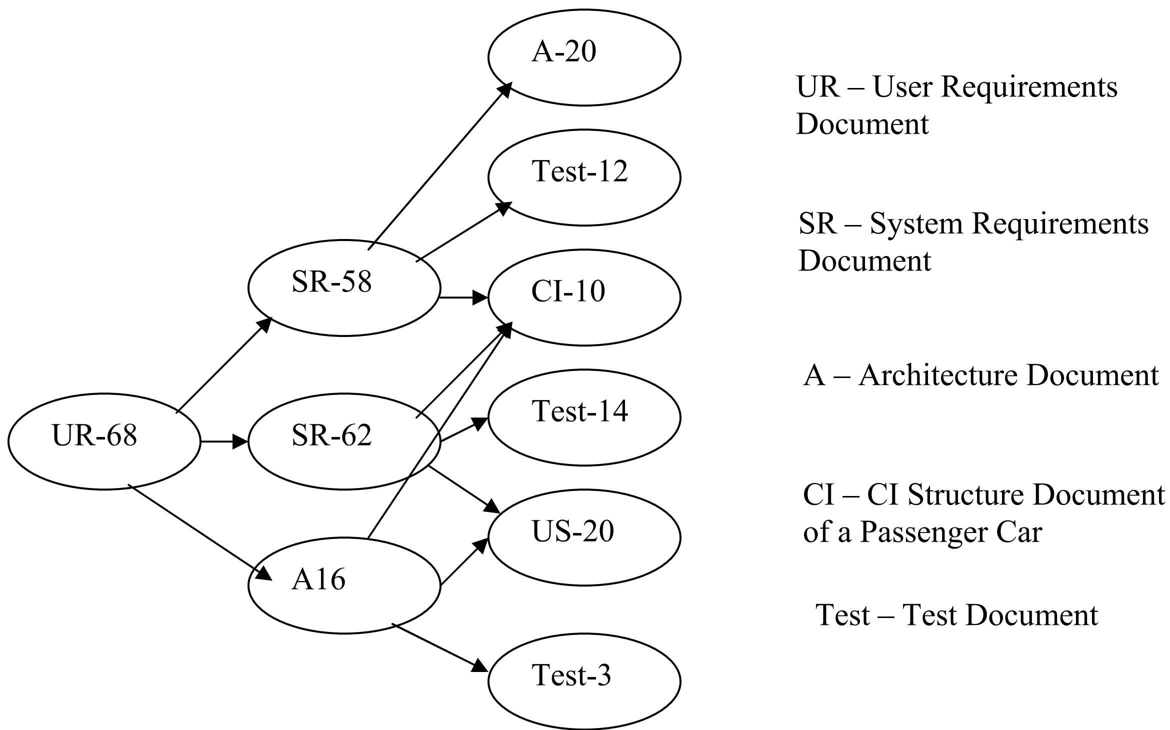

Figure 6: Traceability links example

The script reads all the modules of the current folder using the ***read*** function as follows:

```
read("/ReqMetrics/Requirement/User Requirements", false)
read("/ReqMetrics/Sub-systems/Architecture", false)
read("/ReqMetrics/Sub-systems/Car Use Scenario", false)
read("/ReqMetrics/Sub-systems/CI Structure", false)
read("/ReqMetrics/Test/Verification Methods", false)
read("/ReqMetrics/Requirement/System Requirements", false)
```

Then the "for" loop is used for checking all the incoming links to the object. The link's object identifier, object text, and object heading is then printed to the output screen (refer to Appendix D for sample output).

Metric - 13: Metric to calculate number of outgoing links from the module **(Unit: Total Number of Outgoing Links /document)**. Similar to Metric-12 (Incoming Links), to calculate the total number of outgoing links from a module, all the modules that have direct or indirect (at different levels) links to this module have to be open (refer to Appendix C for script code). For example, in Figure - 6, all the documents such as User Requirements Document, System Requirements Document, Architecture Document, CI Structure Document, and Test Document should be open before counting the outgoing links from the User Requirements Document. After reading all the documents, a recursive function is used to check if the object's target has links.

(refer to Appendix C script code for variables description)

```
int recurIt(int iOutgoing, int iLevel, Object o25)
{
        bool bsuccess = false;

        Link l
        for l in (o25) -> "*" do
        {
                .
                .
                .
        }

}
```

This function prints the object identifier and the level at which the object is found. For example, in Figure-6, A16 is an object at Level-2 (refer to Appendix D for sample output).

Chapter 5: Requirements Engineering Process Using Metrics

5.1 Requirements Elicitation

The requirements elicitation process consists of

- ❖ Identifying the business objectives and aligning the use cases and functional requirements with the business objectives.
- ❖ Identifying the State and Federal guidelines, policies, regulations, and legislation.
- ❖ Developing the use cases.
- ❖ Identifying the systems/sub-systems from other projects that can be re-used for the current project.
- ❖ Identifying the available technology.
- ❖ Identifying the external interfaces as to how the system interacts with people, other hardware, other software, and other agencies.
- ❖ Identifying the constraints on the system.

The requirements elicitation phase will give an informal requirements document expressed in the customer's own words. This document can be entered into the DOORS database, or any other requirements management tool, as the first informal requirements document with use cases.

5.2 Requirements Analysis

The requirements analysis phase consists of analyzing the requirements for ambiguity, conflicts, inconsistencies, missing or extra requirements. Organizing and prioritizing the requirements technique is usually used to resolve conflicts between requirements.Organizing requirements consists of forming groups of requirements to analyze the similarities and contrasts between the grouped requirements. The two techniques normally used for organizing the requirements are: (1) Affinity Analysis (2) User Scenarios.

Affinity analysis is a technique to group entities with merely a logical affinity. It is a technique used to find natural groupings of information, which can be used when structuring business needs. It provides an initial view of the structure of requirements that can further be refined and finalized in the business needs hierarchy. This technique supports identifying innovative ways of implementing the business processes. This group technique is used to structure or understand a complex issue. When successful, it leads to getting a common view of the structure of the problem domain.

User scenarios are specific sequence of interaction between the actor and the system. They are very natural, as stakeholders tend to use them spontaneously. Use cases give the high-level view of user scenarios.

Prioritizing of requirements (1) determines the degree of importance of each requirement to the customer (2) implements the most critical first, as there may not be enough time or resources to implement all requirements (3) identifies conflicting requirements (4) helps in planning successive releases of a product by identifying which requirements should be done first, and which should be left to successive releases.

5.3 Requirements Specification

A requirements document is a specification of what the system should do. Failure to document the requirements results in process failure, interaction failure, and expectation failure. The system requirements should be documented in any of the formats supported by the DOORS tool (see Figure-3, p. 52) to form the System Requirements Specification (SRS) document. The SRS can then be imported in a DOORS database where each requirement is stored as an object. This first SRS produced by documenting the requirements should be baselined and should be used to calculate the metrics such as:

- Lines of text
- Imperatives such as “shall”, “will”, “must”
- Continuances such as the “the following”
- Weak Phrases such as “large”, “fast”, “enough”
- Incomplete sentences such as “TBD” and “TBS”
- Optional phrases such as “can”, “may”
- Initially allocated requirements

These metrics will give a rough estimate of number of satisfiable requirements and the number of unsatisfiable requirements. This will allow to estimate the degree of functionality required by the system, degree of testing required for the system, non-testable requirements, incomplete requirements, non-satisfiable requirements, and the hierarchical structure of requirements. Using these metrics indirectly in any of the project management models, will allow estimating the size, cost, staffing and effort for the developing system. Further research is required in the direction of using these metrics in the requirements specification phase of requirements engineering.

5.4 Requirements Management

SRS is a ***live*** document that changes throughout the software life cycle. Requirements management tools like DOORS allow organizing the different documents in different folders. For example, a project is a special kind of folder in DOORS that contains all the data for a particular project. Projects can contain folders and folders can contain modules. “System Requirements” can be one of the modules of the folder “Requirements” in the project “Car”. Requirements Management tools like DOORS also support configuration management where modules being modified can be locked and different users of the system can be given different access rights. This allows the team to work simultaneously on the project and change the SRS document. The changed SRS document can then be baselined at different levels throughout the software life cycle. The metrics collected during this phase such as:

- Number of changes to an object
- Number of changes in a specific time period
- Cause of change
- Number of incoming links in the module
- Number of outgoing links from the module
- Finally allocated requirements

assist in identifying how far is the development of the system in the software life cycle. For example, if the number of changes in a specific time period has reduced by 70%, it can be judged that the requirements of the system are stable and probably well defined. On the other hand, if the number of changes in a specific time period has increased by 30%, it can be judged that there is a huge scope creep in the system.

Chapter 6: Conclusions and Recommendations

Requirements are the foundation upon which the entire system is built. The high failure rate of software indicates that often these requirements are not satisfied. This results in either fixing of deficiencies or accepting the fact that certain functionality will not be there in the developed system. The best approach is to get the requirements right the first time. Measuring and collecting requirements metrics throughout the software lifecycle is characteristic of good requirements management practice. The various requirements management tools available in the market make the task of managing requirements easy. However, these tools do not necessarily provide the metrics for the requirements engineering process. The DXL scripting library of DOORS has been used as an example in this project, to collect the requirements metrics. DOORS has very good DXL reference manual available along with the software that made the task of writing scripts for collecting metrics easy.

The various requirements metrics collected in this project are lines of text, number of imperatives, number of continuances, number of weak phrases, number of incomplete sentences, number of optional phrases, number of changes to an object, number of changes in a specific time period, cause of change, number of finally allocated requirements, number of initially allocated requirements, number of incoming links in the module, and the number of outgoing links from the module. As shown in the sample output, the numbers produced by running the script give a detailed overview of the

whole requirements phase of the project. This overview gives the management a perspective into how far they are into the software development process. Requirements metric collection is cheaper, faster and more reliable with requirement management tools. It requires commitment from the organization to adopt and maintain it as part of the organizational process.

Some of the limitations of the DOORS tool used for this project are:

- ❖ The tool requires that all the required modules that are referenced in the metric script should be loaded in the memory.
- ❖ The tool does not accept any data input from the command console.

A few recommendations for further work are:

- ❖ Currently, the script prints the result on the console. The script can further be extended to output the quality trends using metrics graphs at different levels throughout the requirements phase.
- ❖ The script is currently returning metric numbers for a specific module. The script can further be extended to produce results for specific folders and specific project.
- ❖ The current project uses DOORS 5.0. The DOORSNet can further be used to analyze the metrics collection capability of DOORSNet.
- ❖ The project does not identify certain requirements elicitation metrics such as number of requirements gathered in a specific time period. This needs to be further defined in future work.

- This project does not define the requirements engineering process using these metrics. Complete requirements engineering process needs to be defined using these metrics and requirements management tools.
- The project calculates the raw metric data. It does not define the objective scale for measuring these metrics. For example, if one project has 100 lines of text and the other project has 1000 lines ot text, there is no objective scale to define which project is better. Further research needs to be done in the area of defining an objective metric scale.
- The scripts defined in this project return the raw metric data. Statistical analysis and meta-data needs to be defined from this raw metric data.
- There are various other tools available in the market. They need to be analyzed for their capability to collect metrics.

References:

1. AnalystPro Software Development, Requirements Management (2000). Retrieved from the World Wide Web http://www.analysttool.com/requirements.htm.
2. Asbrand D. (1998). IT Metrics For Success, Information Week Online, News in Review issue of August 17, 1998.
3. ASEA Report (1994). Analysis of Automated Requirements Management Capabilities developed in support of Advanced System Engineering Automation (ASEA), CSC-2.7 Requirements/Design Manager (Contract N0. F30602-93-C-0123), Prepared for Rome Laboratory, Air Force Materiel Command C3CB 525 Brooks Rd. Griffiss AFB, NY 13441 by Software Productivity Solutions, Inc. 122 4th Avenue Indalantic, FL. 32903.
4. Bahill, A.T. and Dean, F.(1999). Discovering system requirements, Chapter 4 in the Handbook of Systems Engineering and Management, A.P. Sage and W.B. Rouse (Eds), John Wiley & Sons, 175-220.
5. Brooks, F.P. (1987). Essence and Accidents of Software Engineering, IEEE Computer, Vol. April 1987, pp. 10-19
6. Costello R.J. and Liu D. (1995). Metrics for Requirements Engineering, Journal of Systems Software, Vol. 29, pp. 39-63
7. CRSIP (Computer Resources Support Improvement Program) (1999). A Gentle Introduction to Software Engineering, Technical Report Update published by U.S. Air Force's Software Technology Support Center (STSC), Ogden Air Logistics Center, 7278 4th Street, Hill Air Force Base, Utah.

8. Davis A.M. (1993). Software Requirements – Objects, Functions, and States, Prentice Hall Publication, New Jersey.
9. Davis A. M. and Leffingwell D. A. (1999). Making Requirements Management Work for You, The Journal of Defense Software Engineering CrossTalk issue of April 1999.
10. Deb S. (1995). Formal Methods Within the Development Cycle, 27-320 Software Engineering course offered by Prof. Stefan C. Kremer at the University of Guelph in Winter 2000.
11. Eberlein A. (1997). Requirements Acquisition and Specification for Telecommunication Services, PhD Thesis, University of Wales, Swansea, UK.
12. Embury S. M. (2001). Software Maintenance, CM-0206 course notes on Software Engineering offered at the Cardiff University, Cardiff.
13. Field T. (1997). When Bad Things Happen to Good Projects, CIO Magazine of October 15, 1997.
14. Gotel O.C.Z. and A.C.W. Finkelstein (1993). An Analysis of the Requirements Traceability Problem. Technical Report Tr-93-41, Department of Computing, Imperial College of Science, Technology and Medicine, London, U.K., August.
15. Grinter R.E. (1996). Understanding Dependencies: A Study of the Coordination Challenges in Software Development. A dissertation submitted for the degree of PhD in Information and Computer Science, University of California, Irvine.
16. Hammer T. and Huffman L.(1998). Automated Requirements Management - Beware HOW you Use Tools - An Experience Report, presented at the 1998

International Conference on Requirements Engineering (ICRE'98), Colorado Springs, CO.

17. Integrated Chipware. http://www.chipware.com
18. Jones B.C. (1998). Requirements Management Measurement Plan prepared for A.D.Ho and Company, retrieved from the World Wide Web http://www.cpsc.ucalgary.ca/~jonesb/seng/623/requirementsPlan.html.
19. Kotonya G. and Sommerville I. (1997). Requirements Engineering - Processes and Techniques, John Wiley and Sons Ltd.
20. Lord Kelvin (1889). Counting the Numbers, The Journal of Electronic Publishing, Published by the University of Michigan Press, Retrieved from the World Wide Web http://www.press.umich.edu/jep/06-02/glos0602.html.
21. Macaulay L.A. (1996). Requirements Engineering, Springer-Verlag, London.
22. MetricCenter (2001). Metrics Details for Telelogic DOORS, Retrieved from the World Wide Web http://www.distributive.com/Documents/MetricDetail_DOORS.pdf.
23. Moore D. L (1999-2000). A look at the role of software requirements specifications in the changing world of software development, published in Systems - Enterprise Computing Monthly, December/January 1999-2000.
24. Paulk M. C., Charles V. Weber, Suzanne M. Garcia, Mary Beth Chrissis, Marilyn Bush (1993). Key Practices of the Capability Maturity Model, Version 1.1, Technical Report published by Software Engineering Institute at Carnegie Mellon University, Pittsburgh, Pennsylvania, February 1993.

25. Pressman R. S.(2001). Software Engineering: A Practitioner's Approach, 5th ed.; McGraw-Hill, 2001.

26. Ramesh B., Lt. Curtis Stubbs, Lt. Cmdr. Timothy Powers, and Edwards M. (1995). Lessons Learned from Implementing Requirements Traceability, The Journal of Defense Software Engineering CrossTalk issue of April 1995.

27. Rational Software Corp. http://www.rational.com

28. Rosenberg L.H., Hammer T.F., Huffman L.L. (1998). Requirements, Testing, and Metrics, presented at the 16th Pacific Northwest Software Quality Conference, UTAH, October 1998, presented by the NASA Software Assurance Technology Center.

29. Rosenberg L.H., Hammer T.F., Shaw J. (1998). Software Metrics and Reliability, presented at the 9th International Symposium, "BEST PAPER" Award, November, 1998, Germany, presented by NASA Software Assurance Technology Center.

30. Satta J. (1999). Gathering Requirements - a Product Manager's Perspective, Telelogic (http://www.telelogic.com/) NewsByte Magazine of September 16, 1999.

31. Sawyer P.(2001). Software Requirements Engineering - An Introduction and Overview, CSC-321 course on Requirements Engineering offered at the Computing Department, Lancaster University.

32. Sommerville I. and Sawyer P. (1997). Requirements Engineering – A good practice guide, John Wiley and Sons.

33. SDRC – http://www.sdrc.com

34. Standish Group *Chaos* Report (1995). Report Published by The Standish Group International Inc., in 1995.

35. Standish Group Report (2000). Report Published by The Standish Group International Inc., in 2000.

36. Standish Group Report (2001). "Extreme Chaos", Report Published by The Standish Group International Inc., in 2001.

37. Stevens R. (1996). Requirements Management, lecture offered by founder and technical director of QSS Ltd. at the Staffordshire University, The Octagon, Beaconside,

38. Telelogic AB. http://www.telelogic.com/

39. Wieringa R.J. (1995). Requirements Engineering - Frameworks for Understanding, John Wiley and Sons.

40. Wiegers K.E. (1999). Software Requirements, Practical Techniques for Gathering and Managing Requirements Throughout the Product Development Cycle, Microsoft Press.

41. Wiegers K. E. (1999). In Search of Excellent Requirements, a brochure obtained from the vendor Integrated Chipware.

Appendix A – Tool Comparision

No.	Features	RequisitePro	RequireIt	RTM	DOORS	SLATE
1	Database the tool uses	Oracle, MSSQL, MSAccess	None	Oracle	Proprietary Database	Versant OODB
2	Magnitude of project size the tool can handle	small/medium	small(500-1000 reqts/document)	All	large (>10000 reqts/document)	Tested at 800,000 reqs/doc.
3	Primary user interface for the tool	MS-Word	MS-Word	MS-Word, Adobe Frame-Maker	Proprietary	Proprietary
4	Capability to parse the source document	Yes	Yes, if project data is maintained in one folder	Yes	Yes	PARTIAL
5	Cross-project traceability	Yes	No	Yes	Yes	Yes
6	Manage artifacts other than requirements	Yes	No	Yes	Yes	Yes
7	Allow uni-directional traceability	Yes	Yes		Yes	
8	Allow bi-directional traceability	No	No	Yes	Yes	Yes
9	Support sharing of artifacts across projects	No	No	Yes	Yes	
10	Support requirements management	Yes	Yes	Yes	Yes	Yes
11	Handle code artifacts or link to artifacts in non-word documents	No	No	Yes	Yes	
12	Store attribute information along with the requirements	Yes	Yes	Yes	Yes	
13	Support full life-cycle traceability processes	Yes	Yes, if data can be captured in MS Word	Yes	Yes	Yes
14	Support change management and project management process	Yes	No	Yes	Yes	Yes
15	Support user-defined configuration management system	Partially	Partially	Yes	Yes	Yes
16	Support multi-project requirements management system		No	Yes	Yes	Yes
17	Support multi-user requirements management system	Yes	No	Yes	Yes	Yes
18	Support requirements pre-traceability		No		Yes	Yes
19	Support document-to-document traceability		Yes	Yes	Yes	No
20	Support requirements-to-requirements traceability		Yes	Yes	Yes	Yes
21	Integrates with other tools such as testing, design and project management	Yes	No	Yes	Yes	Yes
22	Support Manipulation of graphics objects	No	No	No	Yes	Yes
23	Integration with UML Tool	Yes Rational Rose	NO	YES Rational Rose	YES GDProd & Rational Rose	YES Rational Rose
24	Operating Systems Supported	Windows 95, 98, 2000, ME & NT 4.0	Windows 95, 98, 2000 & NT 4.0	Windows NT, Unix	Win 95, 98, NT,2000, HP-UNIX, Sun Solaris	Sun Solaris/SunOS, HP-UX, Win 95/98/NT

Appendix B – Telelogic Vendor Report

(As received from the Vendor, Telelogic Inc.)

Feature/Capability	DOORS	DOORSnet	DOORSrequireIT
Auto parsing for requirements	X		X
Pre-defined user attributes			X
Multiple object attribute display	X	X	
Multiple object attribute edit	X		
Attribute setting from spreadsheet	X		
Change history creation	X	X	
Change history viewing	X		
Baseline creation	X		
Baseline viewing	X		
Access controls	X	X	
Multi-user document editing	X	X	
Hierarchical project organization	X	X	
Hierarchical text w/ inheritance	X	X	
Document explorer	X		X
Column displays	X	X	
View saving and loading	X	X	
Link creation	X		X
Link viewing	X	X	X
Link filtering	X		
Drag-and-drop linking	X		
Attributes on links	X		
Pop-up link tips	X	X	
Multi-level link views	X	X	
Traceability explorer	X		
Link matrix	X		

Simple filtering	X	X	X
Advanced filtering	X	X	
Independent filter saving		X	
Simple sorting	X	X	X
Advanced sorting	X	X	
Pre-defined reports			X
Customizable reports	X		
Change Proposal System	X	X	
Suggestion logging	X	X	
Threaded discussion	X		
e-mail capability	X	X	
Table cells as requirements	X	X	
Pictures as requirements	X	X	X
Graphical hierarchy display	X		
Templates	X		X
Scaleable sizes: req per document	>10,000	N/A	500-1000
Customizable with scripts	X		
Interfaces with other tools	X		
Live web access	X	N/A	

© 1998 QSS, Inc and affiliated companies. All rights reserved. DOORS is a trademark of QSS, Inc.

Appendix C – DXL Scripts

To run the DXL script in DOORS:

(1) Open any of the module for which you want to collect the metrics data.

(2) Currently the module name and path has been hard-coded in the script (this script has been tested for the System Requirements Module provided in Requirements folder of the DOORS test database). Modify the name and path of the module to the module for which you want to collect the data.

(3) Go to Tools->Edit DXL. Load the DXL script.

(4) Go Run in the DXL window.

DXL Script for calculating Metrics 1 to Metrics 13 are as follows:

```
/**
Metric - 1: Lines of Text

This piece of code is to print the number of lines in the current
module. The end of line is
identified by using the end of line character \n. However, the results
do not give number of lines.
It considers each object text line as one line. And the heading and
sub-headings are all considered
as one line
**/
print
"*******************************************************************
******""\n"
print "Metric - 1: Lines of Text. Calculates the total number of lines
of text" "\n"
print "based on the number of end of line and return characters.
Although, in DOORS" "\n"
print "it returns the total number of objects in the document.""\n"
print
"*******************************************************************
******" "\n"


Object o1 = first current Module
```

```
Object o2 = last current Module
int endOfLineChar =0
int charReturn = 0
int totalLines = 0
while (o1 != o2)
{

      string s1
      string s2
      string s3

      s2 = "\\n"
      s4 = "\\r"
      s1 = o1."Object Text"
      s3 = o1."Object Heading"

      int    offset
      int    len
      if ((matches(s2, s1)) || (matches(s2, s3)))
      {
            endOfLineChar++
      }

      else if ((matches(s4, s1)) || (matches(s4, s3)))
      {
            charReturn++
      }

      o1 = next o1

}
totalLines = endOfLineChar + charReturn
print "The total number of lines of text in this module is ="
print totalLines "\n"


/**
Metric - 2: Imperatives to measure the actual number of "shall",
"must", and "will"

This piece of code is to print the total number of imperatives in this
module. I came
across a statement like "The car shall be compatible with all standard
fuel supply mechanisms in the
countries to which it will be sold." Here "will" is recorded as a
requirement.
Moreover, statements which have multiple requirements defined in one
statement are
counted as one requirement statement.
**/
print "\n" "\n"
print
"**********************************************************************
******""\n"
print "Metric - 2: Imperatives to measure the actual number of shall,
must, and will" "\n"
```

```
print
"**********************************************************************
******" "\n"


Object o3 = first current Module
Object o4 = last current Module
int totalShall =0
int totalWill = 0
int totalMust = 0
bool bImperative = false

while (bImperative == false)
{

      string s1
      string s2
      string s3
      string s4
      string s5


      s1 = o3."Object Text"
      s2 = o3."Object Heading"
      s3 = "shall"
      s4 = "will"
      s5 = "must"

      int    len

      Buffer b = create
      b = s1
      char ch = '\n'
      int    offset = 0
      int    offset2 = 0
      int    offset3 = 0


      int iRetShall = contains(b, s3, offset)
      int iRetWill = contains(b, s4, offset2)
      int iRetMust = contains(b, s5, offset3)

      while ( iRetShall != -1)
      {
            totalShall++
            offset = iRetShall
            iRetShall = contains(b, s3, offset)
      }

      while ( iRetWill != -1)
      {
            totalWill++
            offset2 = iRetWill
            iRetWill = contains(b, s4, offset2)

      }
```

```
	while ( iRetMust != -1)
	{
		totalMust++
		offset3 = iRetMust
		iRetMust = contains(b, s5, offset3)

	}

	if (o3 == o4)
	{
		bImperative = true
	}

	o3 = next o3

}
print "Total number of shall in this module are:="
print totalShall "\n"
print "Total number of will in this module are:="
print totalWill "\n"
print "Total number of must in this module are:="
print totalMust


/**
Metric - 3: Metric to calculate the total number of continuances
like"the following"

**/

print "\n" "\n"
print
"*****************************************************************************
******""\n"
print "Metric - 3: Metric to calculate the total number of continuances
like the following " "\n"
print
"*****************************************************************************
******" "\n"


Object o5 = first current Module
Object o6 = last current Module
int totalContinuance =0
bool bContinuance = false

while (bContinuance == false)
{

	string s1
	string s2
	string s3

	s1 = o5."Object Text"
	s2 = o5."Object Heading"
	s3 = "the following"
```

```
        int     len

        Buffer b = create
        b = s1
        char ch = '\n'
        int     offset = 0

        int iRetContinuance = contains(b, s3, offset)

        while ( iRetContinuance != -1)
        {
              totalContinuance++
              offset = iRetContinuance
              iRetContinuance = contains(b, s3, offset)
        }

        if (o5 == o6)
        {
              bContinuance = true
        }

        o5 = next o5

}
print "Total number of continuances in this module are:="
print totalContinuance"\n"

/**
Metric - 4: Metric to calculate the total number of weak phrases
like"large", "fast",
"enough"
**/
print "\n" "\n"
print
"**********************************************************************
******""\n"
print "Metric - 4: Metric to calculate the total number of weak phrases
like large, fast, enough " "\n"
print
"**********************************************************************
******" "\n"


Object o7 = first current Module
Object o8 = last current Module
int totalLarge =0
int totalFast = 0
int totalEnough = 0
bool bWeakPhrase = false

while (bWeakPhrase == false)
{

        string s1
        string s2
        string s3
        string s4
```

```
        string s5


        s1 = o7."Object Text"
        s2 = o7."Object Heading"
        s3 = "large"
        s4 = "fast"
        s5 = "enough"

        int    len

        Buffer b = create
        b = s1
        char ch = '\n'
        int    offset = 0
        int    offset2 = 0
        int    offset3 = 0


        int iRetLarge = contains(b, s3, offset)
        int iRetFast = contains(b, s4, offset2)
        int iRetEnough = contains(b, s5, offset3)

        while ( iRetLarge != -1)
        {
              totalLarge++
              offset = iRetLarge
              iRetLarge = contains(b, s3, offset)
        }

        while ( iRetFast != -1)
        {
              totalFast++
              offset2 = iRetFast
              iRetFast = contains(b, s4, offset2)

        }

        while ( iRetEnough != -1)
        {
              totalEnough++
              offset3 = iRetEnough
              iRetEnough = contains(b, s5, offset3)

        }

        if (o7 == o8)
        {
              bWeakPhrase = true
        }

        o7 = next o7

  }
  print "Total number of weak phrase - Large are:="
  print totalLarge "\n"
  print "Total number of weak phrase - Fast are:="
```

```
print totalFast "\n"
print "Total number of weak phrase - Enough are:="
print totalEnough


/**
Metric - 5: Metric to calculate the total number of incomplete
sentences like
"To be Determined" and "to be specified" . It does not match the case.
**/

print "\n" "\n"
print
"*********************************************************************
******""\n"
print "Metric - 5: Metric to calculate the total number of incomplete
sentences like" "\n"
print "To be Determined and to be specified" "\n"
print
"*********************************************************************
******" "\n"


Object o9 = first current Module
Object o10 = last current Module
int totalTBD =0
int totalTBS = 0

bool bIncomplete = false

while (bIncomplete == false)
{

      string s1
      string s2
      string s3
      string s4
      string s5
      string s6


      s1 = o9."Object Text"
      s2 = o9."Object Heading"
      s3 = "To be determined"
      s4 = "TBD"
      s5 = "To be specified"
      s6 = "TBS"

      int    len

      Buffer b = create
      b = s1
      char ch = '\n'
      int    offset1 = 0
      int    offset2 = 0
      int    offset3 = 0
      int   offset4 = 0
```

```
        int iRetToBeDet = contains(b, s3, offset1)
        int iRetTBD = contains(b, s4, offset2)
        int iRetToBeSpecify = contains(b, s5, offset3)
        int iRetTBS = contains(b, s6, offset4)

        while ( iRetToBeDet != -1)
        {
              totalTBD++
              offset1 = iRetToBeDet
              iRetToBeDet = contains(b, s3, offset1)
        }

        while ( iRetTBD != -1)
        {
              totalTBD++
              offset2 = iRetTBD
              iRetTBD = contains(b, s4, offset2)

        }

        while ( iRetToBeSpecify != -1)
        {
              totalTBS++
              offset3 = iRetToBeSpecify
              iRetToBeSpecify = contains(b, s5, offset3)

        }

        while ( iRetTBS != -1)
        {
              totalTBS++
              offset4 = iRetTBS
              iRetTBS = contains(b, s5, offset4)

        }

        if (o9 == o10)
        {
              bIncomplete = true
        }

        o9 = next o9

}
print "Total number of TBD's :="
print totalTBD "\n"
print "Total number of TBS's are:="
print totalTBS "\n"


/**
Metric - 6: Metric to calculate the total number of option phrases like
"can", "may", and "I think" . It does not match the case.
**/

print "\n" "\n"
```

```
print
"*********************************************************************
******""\n"
print "Metric - 6: Metric to calculate the total number of option
phrases like" "\n"
print "can, may, and I think" "\n"
print
"*********************************************************************
******" "\n"



Object o11 = first current Module
Object o12 = last current Module
int totalCan =0
int totalMay = 0
int totalThink = 0
bool bOptionPhrase = false

while (bOptionPhrase == false)
{

      string s1
      string s2
      string s3
      string s4
      string s5


      s1 = o11."Object Text"
      s2 = o11."Object Heading"
      s3 = "can"
      s4 = "may"
      s5 = "I think"

      int    len

      Buffer b = create
      b = s1
      char ch = '\n'
      int    offset1 = 0
      int    offset2 = 0
      int    offset3 = 0


      int iRetCan = contains(b, s3, offset1)
      int iRetMay = contains(b, s4, offset2)
      int iRetThink = contains(b, s5, offset3)

      while ( iRetCan != -1)
      {
            totalCan++
            offset1 = iRetCan
            iRetCan = contains(b, s3, offset1)
      }

      while ( iRetMay != -1)
```

```
      {
            totalMay++
            offset2 = iRetMay
            iRetMay = contains(b, s4, offset2)

      }

      while ( iRetThink != -1)
      {
            totalThink++
            offset3 = iRetThink
            iRetThink = contains(b, s5, offset3)

      }

      if (o11 == o12)
      {
            bOptionPhrase = true
      }

      o11 = next o11

}
print "Total number of option phrases - Can are:="
print totalCan "\n"
print "Total number of option phrases - May are:="
print totalMay "\n"
print "Total number of option phrases - Think are:="
print totalThink

/**
Metric - 7: Volatility and change metrics - Metric to calculate the
total number of changes to an object
**/
print "\n" "\n"
print
"*********************************************************************
******""\n"
print "Metric - 7: Volatility and change metrics - Metric to calculate
the total" "\n"
print "number of changes to an object. DOORS has an object attribute
called No. of" "\n"
print " Modifications. I am using that attribute in this metric to
calculate the no." "\n"
print " of changes to a specific object. This attribute returns the no.
of modifications" "\n"
print " to the object as recorded in the history. History basically
records only those" "\n"
print " object changes that go through the change proposal process."
"\n"
print
"*********************************************************************
******" "\n"
```

```
Object o17 = first current Module
Object o18 = last current Module

bool bObjectChanges = false
int sChanges = 0

while (bObjectChanges == false)
{
      sChanges = o17."No. of Modifications"
      string sId = identifier(o17)

      if (sChanges > 0)
      {
            print "Requirement " sId " modified " sChanges " number of
times" "\n"
      }

      if (o17 == o18)
      {
            bObjectChanges = true
      }

      o17 = next o17
}


/**
Metric - 8: Volatility and change metrics - Metric to calculate the
total number of changes in a specific
time period. DOORS does not allow to input the data from its DXL input
command line. Hence, we need a
data input file to run this metrics. The file here is d:\\datedata.txt
**/

print "\n" "\n"
print
"*******************************************************************
******""\n"
print "Metric - 8: Volatility and change metrics - Metric to calculate
the total" "\n"
print "number of changes in a specific time period. This metric
compares if the last modified date" "\n"
print " of the object to the dates passed in. Hence, if the object has
been modified " "\n"
print " ten times, the last modified date is the date of the 10th time.
And this will " "\n"
print " show up as one modification in this metrics. However, the
previous metrics that " "\n"
print " shows the no. of modifications to the object will show 10.
Another drawback" "\n"
print " is that if that particular object has been modified 10 times,
but the last " "\n"
print " modified date is greater than the date specified, it will not
show up as modified" "\n"
print " object." "\n"
```

```
print
"***********************************************************************
******" "\n"



void getData(Date &d1, Date &d2)
{
      string filename = "d:\\datedata.txt"
      print "Reading from " filename "\n"
      Stream input = read filename
      string d3, d4
      input >> d3
      input >> d4
      print "Initial date is : " d3 "\n"
      print "Final Date is : "  d4 "\n"
      d1 = d3
      d2 = d4
      close input
}


Date d1, d2, d3
getData( d1, d2)

Object o19 = first current Module
Object o20 = last current Module

int iChangesBetTime = 0
bool bFlag = false
print "The following requirements have been modified : " "\n"
while (bFlag == false) {
      d3 = o19."Last Modified On"

      if ((d3>= d1) && (d3<= d2))
      {
            string s1
            s1 = o19."Object Text"
            string sIdentify = identifier(o19)

            print "\t" sIdentify  " --- " s1 "\n"
            iChangesBetTime++
      }
      if (o19 == o20)
            {
                  bFlag = true
            }

      o19 = next o19
}

print "\n" "The total number of changes in the specified time period
is:="
print iChangesBetTime


/**
```

```
Metric - 9: Volatility and change metrics - Metric to calculate the
cause of change in a specific
time period. DOORS does not allow to input the data from its DXL input
command line. Hence, we need a
data input file to run this metrics. The file here is d:\\datedata.txt.
The other most important thing
is that DOORS does not really close the open module. Hence, I have to
hard core to read a module every
time.
**/

print "\n" "\n"
print
"**********************************************************************
******""\n"
print "Metric - 9: Volatility and change metrics - Metric to calculate
the cause" "\n"
print "in a specific time period. This metric calculates only those
objects in the " "\n"
print " change proposal system to which the proposed changes have been
applied. " "\n"
print " If a change is proposed, but not approved or applied yet, then
this piece of " "\n"
print " code does not consider it as a change." "\n"
print
"**********************************************************************
******" "\n"



Module Proposals = read("/ReqMetrics/Change Proposal System/Proposals
1", false)

Date d5, d6, d7
getData( d5, d6)
Object o21 = first current Module
Object o22 = last current Module

int iCauseOfChange = 0
bool bCauseOfChange = false

while (bCauseOfChange == false) {
      d7 = o21."Last Modified On"
      if ((d7>= d5) && (d7<= d6))
      {

            string s1 = o21."CP Attr - Status"
            string s2 = o21."CP Attr - Change Type"
            string s3 = o21."Object Text"
            if (s2 == "Modification")
            {
                  string sChange = identifier(o21)
                  string sProposer
                  HistorySession hs
                  for hs in current Module do
                  {
                        sProposer =  who(hs)
```

```
                }
                print sChange " Modified by: " sProposer
                      print  " --- " s3 "\n"
                iCauseOfChange++
            }

            if (s2 == "Insertion")
            {
                string sChange = identifier(o21)
                string sAddProposer
                HistorySession hs
                for hs in current Module do
                {
                      sAddProposer =  who(hs)
                }

                print sChange " Added by: " sAddProposer
                print  " --- " s3 "\n"
                iCauseOfChange++
            }

      }
      if (o21 == o22)
            {
                bCauseOfChange = true
            }

      o21 = next o21
}

print "\n" "The number of changes due to modifications/additions are:="
print iCauseOfChange
close(Proposals, false)




/**
Metric - 10: Volatility and change metrics - Metric to calculate the
total number of finally allocated
requirements. The library does not give the handle to the current
baseline or current module after
opening the initial baseline document
**/

print "\n" "\n"
print
"*****************************************************************
******""\n"
print "Metric - 10: Volatility and change metrics - Metric to calculate
the total" "\n"
print "number of finally allocated requirements" "\n"
print
"*****************************************************************
******" "\n"
```

```
read("/ReqMetrics/Requirement/System Requirements", false)

Object o15 = first current Module
Object o16 = last current Module

int finalNoOfReqs = 0
bool bfinalNoOfReqs = false

while (bfinalNoOfReqs == false)
{

      string s1
      string s2
      string s3
      string s4

      s1 = o15."Object Text"
      s2 = "shall"
      s3 = "will"
      s4 = "must"

      int    len
      Buffer b = create
      b = s1
      char ch = '\n'
      int    offset1 = 0
      int    offset2 = 0
      int    offset3 = 0

      int iRet = contains(b, s2, offset1)
      int iRetWill = contains(b, s3, offset2)
      int iRetMust = contains(b, s4, offset3)

      while ( iRet != -1)
      {
            //print o15."Absolute Number" " --- " s1 "\n"
            finalNoOfReqs++
            offset1 = iRet
            iRet = contains(b, s2, offset1)


      }

      while ( iRetWill != -1)
      {
            //print o15."Absolute Number" " --- " s1 "\n"
            finalNoOfReqs++
            offset2 = iRetWill
            iRetWill = contains(b, s3, offset2)

      }

      while ( iRetMust != -1)
      {
            //print o15."Absolute Number" " --- " s1 "\n"
            finalNoOfReqs++
```

```
            offset3 = iRetMust
            iRetMust = contains(b, s4, offset3)

      }

      if (o15 == o16)
      {
            bfinalNoOfReqs = true
      }

      o15 = next o15

}
print "Total Number of Requirements in the Final Document = "
print finalNoOfReqs

/**
Metric - 11: Volatility and change metrics - Metric to calculate the
total number of initially allocated
requirements.
**/

print "\n" "\n"
print
"*****************************************************************
******""\n"
print "Metric - 11: Volatility and change metrics - Metric to calculate
the total" "\n"
print "number of initially allocated requirements" "\n"
print
"*****************************************************************
******" "\n"



int initialReqs = 0
bool bInitialReqs = false

Module m = load(baseline(0,1,"SR"), false)
Baseline b = baseline(0,1,"SR")
Date d
d = dateOf(b)

Object o13 = first current Module
Object o14 = last current Module


Date d10

while (bInitialReqs == false) {
      d10 = o13."Created On"
      if (d10<= d)
      {
            string s1
            string s2
            string s3
            string s4
```

```
            s1 = o13."Object Text"
            s2 = "shall"
            s3 = "will"
            s4 = "must"

            int    len

            Buffer b = create
            b = s1
            char ch = '\n'
            int    offset1 = 0
            int    offset2 = 0
            int    offset3 = 0

            int iRet = contains(b, s2, offset1)
            int iRetWill = contains(b, s3, offset2)
            int iRetMust = contains(b, s4, offset3)


            while ( iRet != -1)
            {
                  initialReqs++
                  offset1 = iRet
                  iRet = contains(b, s2, offset1)
            }

            while ( iRetWill != -1)
            {
                  initialReqs++
                  offset2 = iRetWill
                  iRetWill = contains(b, s3, offset2)

            }

            while ( iRetMust != -1)
            {
                  initialReqs++
                  offset3 = iRetMust
                  iRetMust = contains(b, s4, offset3)

            }
      }
      if (o13 == o14)
            {
                  bInitialReqs = true
            }

      o13 = next o13
}
//close(m, false)
print "Total Number of Initial Requirements is:="
print initialReqs



/**
```

```
Metric - 12: Traceability metrics - Metric to calculate number of
requirements traced to/from
each specification. The code piece below prints all the incoming links.
**/

print "\n" "\n"
print
"***********************************************************************
******""\n"
print "Metric - 12: Traceability metrics - Metric to calculate number
of incoming" "\n"
print "links to requirement object" "\n"
print
"***********************************************************************
******" "\n"

read("/ReqMetrics/Requirement/User Requirements", false)
read("/ReqMetrics/Sub-systems/Architecture", false)
read("/ReqMetrics/Sub-systems/Car Use Scenario", false)
read("/ReqMetrics/Sub-systems/CI Structure", false)
read("/ReqMetrics/Test/Verification Methods", false)
read("/ReqMetrics/Requirement/System Requirements", false)


Object o26 = first current Module
Object o27 = last current Module

bool bIncoming = false
int totalIncoming = 0

while (bIncoming == false)
{
      Link l
      for l in (o26) <- "*" do
      {
            totalIncoming++
            //string srcModName
            //for srcModName in o26<-"*" do print identifier(o26) "\n"
            string sId = identifier(o26)
            print "Object Identifier = " sId "\n"
            print "Text=="  o26."Object Text" "\n"
            print "  "   "Heading=="  o26."Object Heading" "\n"
            print "==================================================="
"\n"
      }

      if (o26 == o27)
      {
            bIncoming = true
      }
      o26 = next o26
}

print "\n" "Total incoming links to this module are: "
print totalIncoming
```

```
/**
Metric - 13: Traceability metrics - Metric to calculate number of
requirements traced to/from
each specification. The code piece below prints all the outgoing links.
**/

print "\n" "\n"
print
"***********************************************************************
******""\n"
print "Metric - 13: Traceability metrics - Metric to calculate number
of outgoing" "\n"
print "links to requirement object. In this metric it is important to
mention which " "\n"
print "module should be considered for calculating the outlinks. Hence,
the read line" "\n"
print "loads that particular module for calculating the outlinks." "\n"
print
"***********************************************************************
******" "\n"

read("/ReqMetrics/Requirement/User Requirements", false)
read("/ReqMetrics/Sub-systems/Architecture", false)
read("/ReqMetrics/Sub-systems/Car Use Scenario", false)
read("/ReqMetrics/Sub-systems/CI Structure", false)
read("/ReqMetrics/Test/Verification Methods", false)
read("/ReqMetrics/Requirement/System Requirements", false)



int totalOutgoing = 0

Object o23 = first current Module
Object o24 = last current Module


int recurIt(int iOutgoing, int iLevel, Object o25)
{
      bool bsuccess = false;

      Link l
      for l in (o25) -> "*" do
      {
            Object ooo = target(l)

            if (iLevel==1)
            {
                  print "\n" "\n"
                  print "iOutgoing == " iOutgoing "\n"
            }

            bsuccess = true
            totalOutgoing++
            string sId = identifier(ooo)
            print "Object Identifier = " sId "      "
            print "Level ="  iLevel "\n"
```

```
            print "Text=="  ooo."Object Text" "\n"
            print "  "   "Heading=="  ooo."Object Heading" "\n"
            recurIt(iOutgoing, iLevel + 1, ooo)
      }

      if ((iLevel==1) && (bsuccess == true))
      {
            print "== == == == == == == == == == == == == == == == ==
== " "\n"
      }
      return totalOutgoing
}


bool bOutgoing = false
int iOutgoing=0
int totalOut = 0
while (bOutgoing == false)
{
      iOutgoing++
      totalOut  = recurIt(iOutgoing, 1, o23)
      if (o23 == o24)
      {
            bOutgoing = true
            print "\n" "Total outgoing links from this module are: "
            print totalOut

      }
      o23 = next o23
}
```

Appendix D – Sample Output

Sample Output:

```
******************************************************************************
Metric - 1: Lines of Text. Calculates the total number of lines of text
based on the number of end of line and return characters. Although, in DOORS
it returns the total number of objects in the document.
******************************************************************************
The total number of lines of text in this module is =142


******************************************************************************
Metric - 2: Imperatives to measure the actual number of shall, must, and will
******************************************************************************
Total number of shall in this module are:=73
Total number of will in this module are:=3
Total number of must in this module are:=0


******************************************************************************
Metric - 3: Metric to calculate the total number of continuances like the following
******************************************************************************
Total number of continuances in this module are:=2


******************************************************************************
Metric - 4: Metric to calculate the total number of weak phrases like large, fast, enough
******************************************************************************
Total number of weak phrase - Large are:=0
Total number of weak phrase - Fast are:=1
Total number of weak phrase - Enough are:=0


******************************************************************************
Metric - 5: Metric to calculate the total number of incomplete sentences like
To be Determined and to be specified
******************************************************************************
Total number of TBD's :=3
Total number of TBS's are:=2


******************************************************************************
Metric - 6: Metric to calculate the total number of option phrases like
can, may, and I think
******************************************************************************
Total number of option phrases - Can are:=0
Total number of option phrases - May are:=1
Total number of option phrases - Think are:=0


******************************************************************************
```

Metric - 7: Volatility and change metrics - Metric to calculate the total
number of changes to an object. DOORS has an object attribute called No. of
Modifications. I am using that attribute in this metric to calculate the no.
of changes to a specific object. This attribute returns the no. of modifications
to the object as recorded in the history. History basically records only those
object changes that go through the change proposal process.
****

Requirement SR8 modified 2 number of times
Requirement SR14 modified 1 number of times
Requirement SR156 modified 2 number of times
Requirement SR155 modified 2 number of times

****

Metric - 8: Volatility and change metrics - Metric to calculate the total
number of changes in a specific time period. This metric compares if the last modified date
of the object to the dates passed in. Hence, if the object has been modified
ten times, the last modified date is the date of the 10th time. And this will
show up as one modification in this metrics. However, the previous metrics that
shows the no. of modifications to the object will show 10. Another drawback
is that if that particular object has been modified 10 times, but the last
modified date is greater than the date specified, it will not show up as modified
object.
****

Reading from d:\datedata.txt
Initial date is : 6 Nov 2001
Final Date is : 17 Nov 2001
The following requirements have been modified :

SR8 --- The car shall be able to move forwards at all speeds from 0 to 200 kilometers per hour on standard flat roads with winds of 0 kilometers per hour, with 180 BHP.

SR10 --- The car shall be able to move backwards to a maximum speed of 20 Kilometers per hour on standard flat roads with winds of 0 kilometers per hour, with 180 BHP.

SR14 --- The car shall be able to accelerate from 250 to 350 kilometers per hour at a rate of 4 kilometers per second on standard flat roads with winds of 0 kilometers per hour.

SR137 --- The car shall be assembled from modules by 2 person in 2 working day.

SR141 --- The car shall be able to travel at 80 kph with a passenger load of 90 kilograms and luggage of 0 kilograms on standard flat roads with a wind speed of 0 kph at a maximum fuel cost of 1.0 pence per kilometer at prices of 1 January 1994. The system may also do something else

SR142 --- The car shall be able to travel at up to 250 kph with a passenger load of up to 300 kilograms and luggage of up to 300 kilograms on standard flat roads with a wind speed of 0 kph at a maximum fuel cost of 4 pence per kilometer at prices of 1 January 1994.

SR144 --- The car shall be compatible with all standard fuel supply mechanisms in the countries to which it will be sold. To be specified

SR146 --- The car shall be fitted with ABS. TBS

SR148 ---

SR149 ---

SR151 --- The system shall be able to do the following: (1) Run fast (2) apply break - TBD

SR153 --- the system shall respond in 5 secs - To be determined

SR156 --- the system shall respond in 15 secs - to be determined

SR155 --- the system shall respond in 5 secs - To be determined

The total number of changes in the specified time period is:=14

****
Metric - 9: Volatility and change metrics - Metric to calculate the cause
in a specific time period. This metric calculates only those objects in the
change proposal system to which the proposed changes have been applied.
If a change is proposed, but not approved or applied yet, then this piece of
code does not consider it as a change.
****
Reading from d:\datedata.txt
Initial date is : 6 Nov 2001
Final Date is : 17 Nov 2001
CP SR1-2 Modified by: vanitashroff --- Modification of 'SR14' in
'/ReqMetrics/Requirement/System Requirements'
CP SR1-3 Modified by: vanitashroff --- Modification of 'SR142' in
'/ReqMetrics/Requirement/System Requirements'
CP SR1-4 Modified by: vanitashroff --- Modification of 'SR8' in '/ReqMetrics/Requirement/System
Requirements'
CP SR1-5 Added by: vanitashroff --- Insertion after 'SR153' in '/ReqMetrics/Requirement/System
Requirements'
CP SR1-6 Added by: vanitashroff --- Insertion below 'SR153' in
'/ReqMetrics/Requirement/System Requirements'

The number of changes due to modifications/additions are:=5

****
Metric - 10: Volatility and change metrics - Metric to calculate the total
number of finally allocated requirements
****
Total Number of Requirements in the Final Document = 76

****
Metric - 11: Volatility and change metrics - Metric to calculate the total
number of initially allocated requirements
****
Total Number of Initial Requirements is:=72

****
Metric - 12: Traceability metrics - Metric to calculate number of incoming
links to requirement object
****
Object Identifier = SR58
Text==A sun roof shall be able to be opened and closed by automatic means by the driver.
Heading==
==
Object Identifier = SR62
Text==The car shall be able to maintain stability of travel with maximum tilt from vertical of the car
being no greater than 4 degrees for the speed/ turning circle curves at reference xyz.
Heading==
==

Total incoming links to this module are: 2

****

Metric - 13: Traceability metrics - Metric to calculate number of outgoing links to requirement object. In this metric it is important to mention which module should be considered for calculating the outlinks. Hence, the read line loads that particular module for calculating the outlinks.
****

iOutgoing == 14
Object Identifier = UR68   Level =1
Text==Users shall be able to stop with the vehicle maintaining a straight track over the stopping distance when the steering is maintained to within + or - 10% of a straight line by the user.
  Heading==
Object Identifier = SR58   Level =2
Text==A sun roof shall be able to be opened and closed by automatic means by the driver.
  Heading==
Object Identifier = CI10   Level =3
Text==
  Heading==Fog lights
Object Identifier = Test12   Level =3
Text==Have the 4 smallest get into car and drive around test track. Each tester will then evaluate the comfort and ride of the car.
  Heading==
Object Identifier = A20   Level =3
Text==
  Heading==Power unit
Object Identifier = SR62   Level =2
Text==The car shall be able to maintain stability of travel with maximum tilt from vertical of the car being no greater than 4 degrees for the speed/ turning circle curves at reference xyz.
  Heading==
Object Identifier = US20   Level =3
Text==The user shall be able to luggage internally.
  Heading==
Object Identifier = Test14   Level =3
Text==Have 2 of the smallest, and 2 of the largest people get into car and drive around test track. Each tester will then evaluate the comfort and ride of the car.
  Heading==
Object Identifier = CI10   Level =3
Text==
  Heading==Fog lights
Object Identifier = A16   Level =2
Text==
  Heading==Sun roof
Object Identifier = Test3   Level =3
Text==Compare Final delivery date to delivery Requirement.
  Heading==
Object Identifier = CI10   Level =3
Text==
  Heading==Fog lights
Object Identifier = US20   Level =3
Text==The user shall be able to luggage internally.
  Heading==
== == == == == == == == == == == == == == == == == ==

iOutgoing == 17
Object Identifier = A14   Level =1

Text==
  Heading==Trunk/boot/Rear door

iOutgoing == 17
Object Identifier = Test8   Level =1
Text==Verify that the automobile is completely maintainable using metric tools.
  Heading==

iOutgoing == 17
Object Identifier = UR62   Level =1
Text==Users shall be able to stop safely.
  Heading==
== == == == == == == == == == == == == == == == == ==

iOutgoing == 58
Object Identifier = CI10   Level =1
Text==
  Heading==Fog lights

iOutgoing == 58
Object Identifier = Test12   Level =1
Text==Have the 4 smallest get into car and drive around test track. Each tester will then evaluate the comfort and ride of the car.
  Heading==

iOutgoing == 58
Object Identifier = A20   Level =1
Text==
  Heading==Power unit
== == == == == == == == == == == == == == == == == ==

iOutgoing == 62
Object Identifier = US20   Level =1
Text==The user shall be able to luggage internally.
  Heading==

iOutgoing == 62
Object Identifier = Test14   Level =1
Text==Have 2 of the smallest, and 2 of the largest people get into car and drive around test track. Each tester will then evaluate the comfort and ride of the car.
  Heading==

iOutgoing == 62
Object Identifier = CI10   Level =1
Text==
  Heading==Fog lights
== == == == == == == == == == == == == == == == == ==

Total outgoing links from this module are: 22

Appendix E – Date Data Input File

The datedata.txt file:

6 Nov 2001
17 Nov 2001