2005

Using active database for management of requirements change

Haipeng Ge
University of Windsor

Using Active Database for Management of Requirements Change

By
Haipeng Ge

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada
2005

Your file Votre référence
ISBN: 0-494-09744-2
Our file Notre référence
ISBN: 0-494-09744-2

Abstract

Software system development projects experience numerous changes during their life cycle. These changes are inevitable and driven by several factors including changes to a system's environment and changes of customers' needs.

Requirements change has been reported as the major contributing factor for poor quality or even failures of software projects. This indicates that management of requirements change still remains a challenging problem in software development.

A critical part of the requirements change management process is impact analysis. To carry out impact assessment, traceability information is needed. Over two decades, requirements traceability has been an important research topic in software research, but the actual practice of maintaining traceability information is not always entirely successful.

In this thesis, a new traceability technique was presented for mapping dynamic behaviors of requirements into Active Databases. The technique keeps requirements and their related artifacts synchronized with respect to their states. It automatically maintains traceability links between requirements and related artifacts when a requirement is changed. This approach can not only efficiently handle basic and necessary traceability functions, but also centralize reactive behavior by using Active Database to ensure no one bypass traceability policies.

Acknowledgment

I would like to express my appreciation to my previous advisor, Dr. Li, Liwu, whose support and guidance make this work possible.

I am very appreciative of Dr. Wu, Dan for being my advisor after Dr. Li passed away. His helps are unforgettable.

Also, I would specially thank Dr. Lu, Jianguo for being the internal reader. His suggestions are the great helps to this thesis.

I would like to express my gratitude to Dr. Kao, Diana for being the external reader from her busy schedules and her suggestions and comments.

Specially, thank Dr. Rueda, Luis for serving as the chair of the committee.

Behind the scenes are my family and friends who were always there for me. I am beholden to you all for your encouragement and support.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Introduction

Requirements problems are expensive and plague almost all systems and software development organizations [SSV99]. Software undergoes changes at all stages of its life cycle. That is, changes to requirements may occur at the requirements elicitation stage, requirements specification stage, design stage, implementation stage, and maintenance stage. Management of requirements change is frequently critical to the success of the software product.

Karl Wiegers and David Card [SSV99] pointed out that despite a half-century of progress in the development of software systems, many organizations continue to struggle with the elicitation, specification, and management of requirements. The foremost reason is that requirements engineering is not only a technical issue, but also a social issue. Much of the information that requirements engineers need is embedded in the social worlds of users and managers, and is extracted through interactions with these people, e.g. through interviews and questionnaires. At its source, this information tends to be informal and highly dependent on its social context for interpretation [GOU94].

As a system's environment changes and customers develop a better understanding of their real needs, requirements change is inevitable. Requirements management is the process in Requirements Engineering to manage changes to a system's requirements.

The principal concerns of requirements management are: managing changes to agreed requirements; managing the relationships between requirements; and managing the dependencies between the requirements document and other documents produced during the systems and software engineering process [KS02].

The relationships and dependencies between requirements and between requirements and other software engineering *artifacts** are needed by impact analysis of *proposed changes** to requirements. This is usually called traceability information. Requirements management is essentially a process of managing those large amounts of traceability information and ensuring that it is delivered to the right people at the right time [KS02].

*An artifact is a piece of information produced or modified as part of the software process [RJ01].

*A proposed change implies that impact analysis should be performed to determine how change would impact the existing system [CCC03].

Therefore, requirements cannot be managed effectively without requirements traceability. Requirements traceability refers to the ability to describe and follow the lifecycle of a requirement and its related software artifacts in both a forwards and a backwards direction, ideally through the whole system lifecycle [GF94]. Requirements traceability captures the relationships between requirements, software design, and system implementation of a project [RPSE95]. All the system components, including hardware, software, personnel, manuals, policies, and procedures created at various stages in the development process are linked to requirements [RPSE95].

In the past two decades, the concern of requirements change in the development and maintenance process of large-scale, complex software projects has increased considerably. Weak engineering discipline in requirements management has become the leading cause of software failures [SG95].

Software engineering researchers have focused on identifying more effective strategies and methods to handle changing requirements [NZW04]. From the traditional methods, such as matrices, hypertext links, graph-base approaches, word processors, and spreadsheets, to commercial tools such as DOORS, Requisite Pro, Cradle and Slate, all these techniques and tools support traceability by establishing direct links between requirements and other traceable artifacts [CCC03].

Jane Cleland-Huang et al. [CZL04] proposed a method for requirements traceability, named Event-Based Traceability (EBT), based on event-notification to establish loosely coupled relationships between artifacts. EBT techniques can be used to trace performance requirements.

Jane Cleland-Huang et al. [CZL04] also proposed a "Best-of-Breed" approach to traceability, in which the return-on-investment of the requirements traceability effort is maximized through strategic deployment of a heterogeneous set of traceability techniques. Those techniques include matrix, information retrieval (IR), Event-based traceability, tracing non-functional requirements (NFR) through design patterns, and other traceability techniques.

In 1994, Gotel and Finkelstein [GF94] published an extensive survey of traceability problems, in which they identified several contributing factors such as insufficient of the allocation of time, staff, and resources, lack of clarity concerning roles played by individuals in the traceability process, failure to follow standard practices, lack of

ongoing cooperation and coordination between people responsible for various traceable artifacts, difficulty in obtaining necessary information in order to support the traceability process.

All these problems have been around for such a long time without being well solved [CCC03]. One main reason is that no single technique can cover all the concerns, but a combination of many techniques will burden practitioners.

1.2 Motivation

This research is motivated by the continuing need to increase the efficiency of management of requirements change. To improve management of requirements change we need to be able to collect current and correct traceability information between requirements and related artifacts. Traceability techniques are used to identify all artifacts that should be updated when a change is introduced. Unfortunately, there is a tendency in even the best traceability schemes for links to fail to keep pace with the evolving system, resulting in the gradual erosion of the traceability infrastructure and its eventual failure to reliably represent the current state of relationships [CCC03]. Those contributing factors include unclear traceability policies and failure to follow standard practices; insufficient resources, time, and support allocated to traceability; lack of clarity concerning roles played by individuals in the traceability process; inappropriate traceability methods; lack of monitoring mechanism for traceability maintenance processes.

Here, I argue that a good solution for requirements traceability should focus on following factors:

- Requirements traceability model should be easy to understand by all stakeholders;
- Providing horizontal and vertical traceability;
- Requirements traceability system implementation should be easy and based on common software;
- To release time-pressure on software engineering practitioners, the system should be easy to use and provide automation as much as possible.

In this thesis, I propose a new technique for requirements traceability which considers the above factors by using active database. When a change is made to a requirement, the active database management system (ADBMS) can automatically update the state of related artifacts based on the types of change to maintain links in an efficient and

consistent manner.

1.3 Research Objective

The research objective is to develop an effective technique for management of critical functional requirements change. Critical functional requirements are the central mission of software system. In fact in certain critical system in which safety is an overarching objective, traceability of critical components must be achieved despite its cost [CZL04]. Therefore, the critical functional requirements change should be traced at a fine-grained level.

However, when we trace critical functional requirements, there is a tendency for the traceability infrastructure to erode over its lifetime, as time-pressured practitioners fail to consistently and systematically update each and every link when changes occur [CCC03]. Then it will fail to reflect the current and accurate state of relationships between requirements and their related artifacts.

In the technical side, the proposed method introduces the new concept of *internal change* into requirements traceability scheme which minimizes the possibility of missing traceability links. In the social aspect, the proposed method places centralized constraints by using active database on traceability maintenance process to ensure no one bypass traceability policies. It also keeps requirements and their related artifacts in a *consistent state**. The well integration of these two aspects will demonstrate that the proposed method is suitable for tracing critical functional requirements.

1.4 Organization of the Thesis

This thesis is organized as follows: Chapter 1 explains the motivation and objective of this thesis. Chapter 2 presents background and reviews related research in requirements traceability. The features of active databases systems are introduced in Chapter 3. Chapter 4 presents the proposed system architecture for mapping the dynamic behavior of requirements into active database. Chapter 5 shows the interactions among requirements, designs and system components by using Triggered Message Sequence Charts. The effectiveness of the proposed method is assessed in Chapter 6. Chapter 7 presents the conclusions.

*An artifact is in a consistent state when its state and the state of its related links accurately represent the current state of the system configuration.

Chapter 2 Background and Related Research

2.1 Introduction

Requirements engineering (RE) is one branch of software engineering that has emerged to facilitate the development of software that truly meets the needs of the client [ZAV97]. Requirements engineering process activities shown in Figure 2.1 include requirements elicitation, requirements analysis and negotiation, requirements documentation and requirements validation [RE04]. In parallel with all of the above processes is a process of requirements management which is concerned with managing changes to the system requirements. The principal requirements management activities are *change control* and *impact analysis*. Change control is concerned with establishing and executing a formal procedure for collecting, verifying and assessing changes; Impact analysis is concerned with assessing how proposed changes affect the existing system. To carry out these activities, information about requirements dependencies, requirements rationale and the implementation of requirements should be maintained. This is usually called traceability information. This research presents a new method for requirements traceability (RT). Sections 2.2 - 2.6 review related research in requirements engineering, requirements management, requirements traceability, and traceability techniques, respectively.

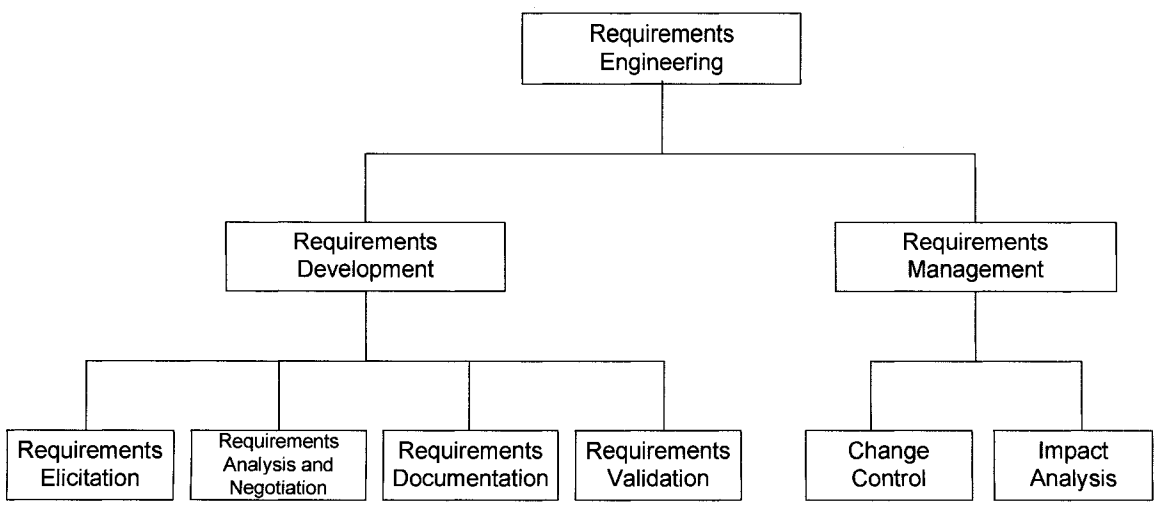

Figure 2.1 Requirements Engineering Process

2.2 Requirements Engineering

Requirements engineering is concerned with the identification of the goals to be achieved by the envisioned system [LAM00]. It is important to realize that it is impossible to develop a computer-based system without knowing its goals.

2.2.1 Requirements Engineering Process

The stage that precedes system design is called requirements engineering. Its aim is to ensure that the delivered system satisfy customer's needs. Normally, requirements engineering is a complex process, because many people involved in it may have different background, views, needs, and interests. The activities in the requirements engineering process are as follows:

(a). Requirements Elicitation

In this process, the system requirements are discovered through consultation with stakeholders, from system documents, existing domain knowledge, and market research. The stakeholders analyze the problems, the needs, and the domain characteristics. Based on that analysis, they decide the changes to be introduced in the domain and the functions that should be performed by the system.

(b). Requirements Analysis and Negotiation

In this process, stakeholders analyze the requirements in detail and different stakeholders negotiate to decide on which requirements are to be accepted. This process is necessary because there are inevitably conflicts between the requirements from different sources, information may be incomplete or the requirements expressed may be incompatible with the budget available to develop the system.

(c). Requirements Documentation

In this process, the agreed requirements are documented at an appropriate level of detail. In general, the requirements document should be understandable by all system

stakeholders. This usually means that the requirements must be documented using natural language and diagrams. More detailed system documentation, such as system models may also be produced.

(d). Requirements Validation

In this process, requirements specifications are checked with respect to customers' needs. It must be ensured that users get a complete understanding of how the future system will be before it is built. This is also a crucial process that can be done well only if requirements have been described explicitly.

2.2.2 Requirements Engineering Process Model

In practice, there are no distinct boundaries between these activities, the activities are interleaved and there are many iteration and feedback from one activity to another activity.

Figure 2.2 shows a sequence of theses phases.

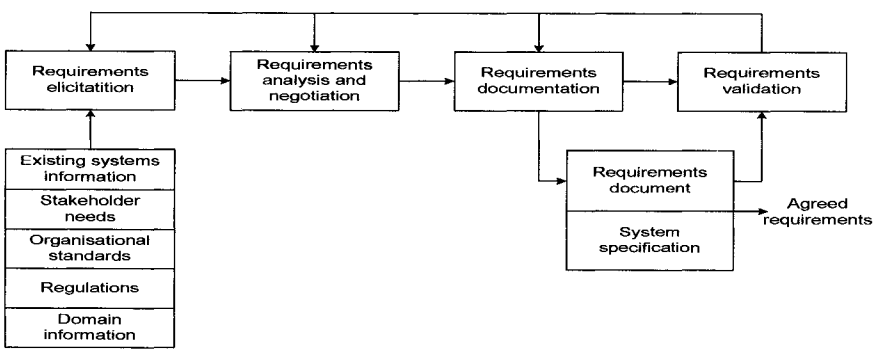

Figure 2.2 Coarse-grain activity model of the requirements engineering process [RE04]

In Figure 2.3, it shows that the different activities in requirements engineering are repeated until a decision is made. If a problem in requirements document is found, the elicitation, analysis and negotiation, documentation, and validation spiral is re-entered. This continues until an acceptable document is produced or until external factors such as schedule pressure or lack of resources mean that the requirements development process should end. A final agreed requirements document then is produced. Any further changes to the requirements are then part of the requirements management process.

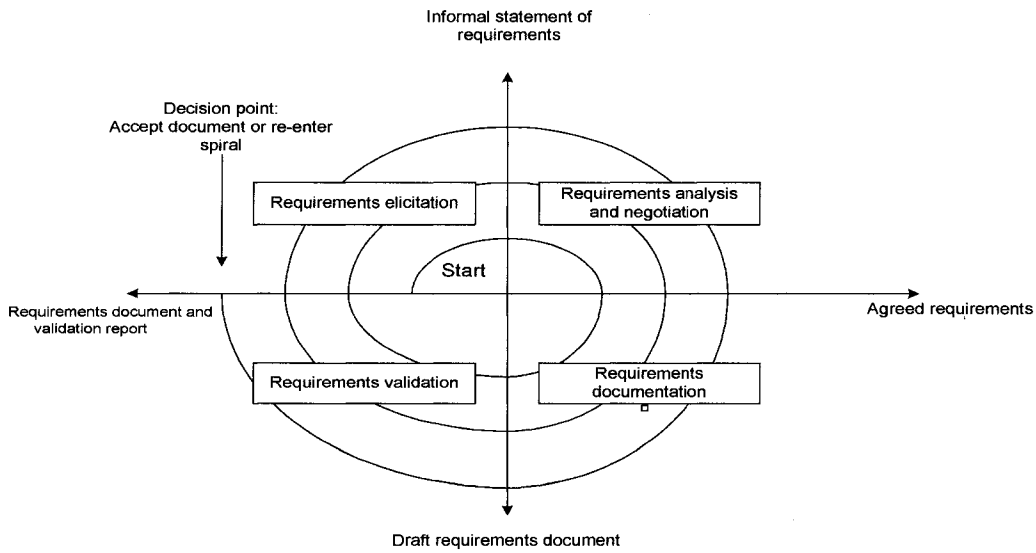

Figure 2.3 A spiral model of the requirements engineering process [RE04].

2.3 Requirements Management

2.3.1 Introduction

The management of requirements is an essential element of software development to ensure program success. As software systems become increasingly large, the management of their requirements becomes increasingly challenging [PN98].

As computer based system has been involved large application domain, management of these system development has been more complicated and sometimes uncontrollable. The successful management of a large system development requires strict control over the requirements specification, the documentation and code constituting the product [PN98].

2.3.2 Requirement Management Functions

There are two important activities in requirements management process: impact analysis and change control. Impact analysis is defined by Bohner and Arnold as "identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change" [BOH91]. Impact analysis uses relationships between

requirements and related artifacts. Lack of detailed information between requirements and related artifacts limits the effectiveness of impact analysis.

Change control is concerned with the procedures, processes and standards which are used to manage changes to a system's requirements. Change control ensures that traceability information is collected for each proposed change and overall judgment is made about the costs, possibility and benefits of proposed changes. Without formal change control, it is impossible to ensure that proposed changes to the requirements fulfill the fundamental business goals.

2.3.3 Efficient Requirements Management

The aim of requirements management is to reduce requirements related errors and to ensure requirements traceability throughout all development phases. Following are some important issues that efficient requirements management should consider:

• Ensure Requirements Traceability.

Requirements traceability is very important to ensure that the software is produced in accordance with stakeholders expectations and that the stakeholders receive what they have paid for, no more and no less.

We should well organize all collected requirements and make sure all of them are handled, even though some are not necessarily implemented. This implies that each requirement and all other components need to have their own, unique identifier in order to perform requirements tracing. And the storage of information in a table or database can ensure full traceability.

• Using Appropriate Attributes of Requirements.

When we store the information of requirements traceability, we should find a way to get a good representation of requirements. That means what kind of attribute of a requirement should be selected. And this selecting should make it easier to sort requirements and to search for requirements with specific properties. By using appropriate attributes, we can view and analyze requirements from many different points of view. Some potential attributes are:

• Source

- Priority
- Requirement types
- Status
- Date
- Created by
- Updated information

• Choosing Appropriate Requirements Management (RM) tools.

There are many advantages when we use RM tools to support requirements traceability. The tools provide various methods by which stakeholders are able to view, update requirements traceability links. Some tools can also produce statement of compliance from customer and product requirements, keep track of testing progress, evaluate cost and so on.

• Making Sure Developers Get Enough Training When Using RM Tools

The requirement management tools available today 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. After you get appropriate RM tool, the next thing is how to use it, or how to utilize RM tool to maximize your return-on-investment. Some requirements management tools are complicated, and need time to learn how to use it. All practitioners must get training and must follow the procedures of RM tools.

2.4 Requirements Traceability

2.4.1 Introduction

One of the aims of requirements management is to ensure requirements traceability throughout system development life cycle. Within system development life cycle, requirements must be traced both forward and backward to assure that the correct system is being designed and produced [PAL97]. Requirements traceability, then, is defined as the ability to describe and follow the life of a requirement, in both a forward and backward direction, ideally through the whole systems life cycle [GF94].

*Traceability management** applies to the entire development life cycle from project initiation through operation and maintenance as shown in Figure 2.4.

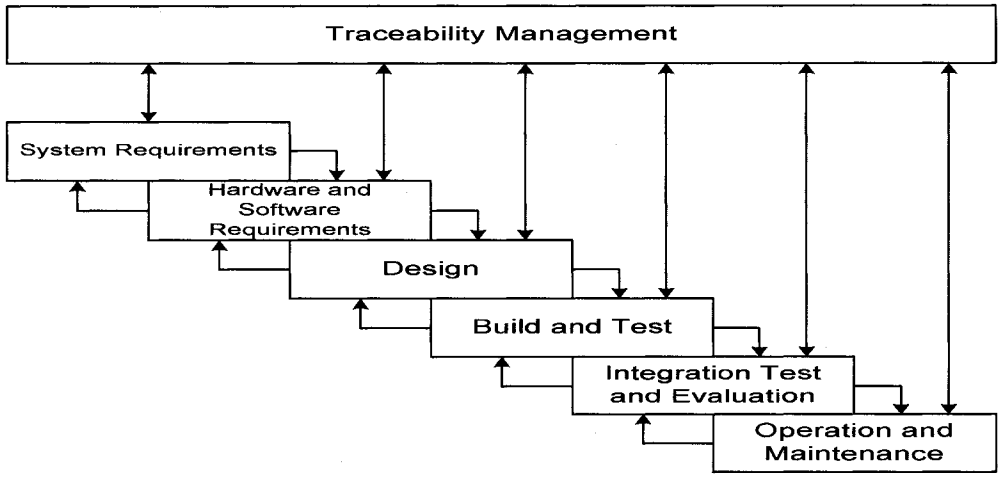

Figure 2.4 Traceability management across the system development life cycle [PAL97]

Successful software system development depends on the ability to satisfy stakeholder needs and to reflect these satisfactions in the delivered system. Requirements and their related artifacts that are in a correct, current and consistent state, play a major role in ensuring that the delivered system truly meets customer needs.

Large-scale complex software systems are initiated by customer expectation. From this beginning, system requirements are elicited to broadly outline the expectation, which, in turn, are investigated to ascertain feasibility and examine trade-offs. Once the feasibility of the desired system have been determined to be necessary and sufficient to launch a new system, design is completed and systems are constructed, tested, and implemented. It is essential to maintain traceability from the system requirements to related artifacts to assure that the delivered system meets the customer's needs.

Traceability gives essential assistance in understanding the relationships that exist within and across software requirements, design and implementation and is critical to the development process by providing a means of ascertaining how and why system development products satisfy stakeholder requirements, especially for large complex systems[PAL97]. Traceability provides a means to validate and verify system requirements to assure the delivered system truly meets customer's needs.

*Traceability management controls and directs tracing from top level through to design and code [PAL97].

However, traceability is often misunderstood, frequently misapplied, and seldom performed entirely successful [PAL97]. There are many challenges to achieving traceability, particularly the absence of automated technique to assist in establishing traceability links between requirements and between requirements and their related artifacts.

2.4.2 Traceability Links

Davis [DAV90] has classified traceability information into four types. Figure 2.5 shows four kinds of traceability links with respect to their relationships to requirements:

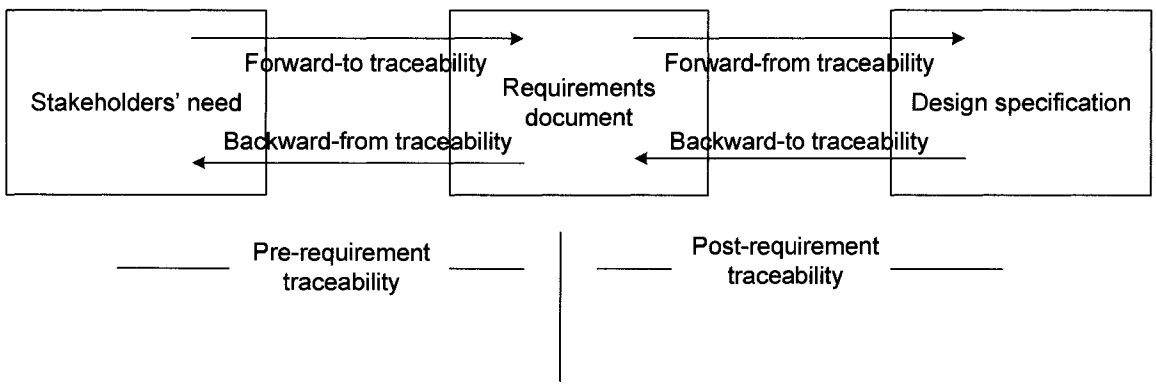

Figure 2.5 Traceability Links

(a). Forward to traceability

Changes in stakeholder needs, as well as in technical assumptions, may require a radical reassessment of requirements relevance.

(b). Forward from traceability

Responsibility for requirements achievement must be assigned to system components, such that accountability is established and the impact of requirements change can be evaluated.

(c). Backward to traceability

Compliance of the system with requirements must be verified, and gold-plating must be avoided.

(d). Backward from traceability

The contribution structures underlying requirements are crucial in validating requirements, especially in highly political settings.

The first two traceability types are called post-requirements traceability. They link requirements to design and implementation, documenting responsibility assignment, compliance verification, or impact analysis of a requirement. The latter two types are called pre-requirements traceability. They document the rationale and sociopolitical context from which the requirements emerge. It is fair to say that post-traceability is much better understood than pre-traceability, even though only pre-traceability will really provide the often demanded linkage between the business and IT.

2.4.3 Issues Involved in Building Traceability Model

A primary concern in the development of large-scale, real-time, complex, computer intensive systems is ensuring that the performance of system meets the specified requirements [RE93]. It is necessary to build a comprehensive scheme for maintaining traceability that all system components, created at various stages of the development process, are linked to the requirements. These components include software, hardware, standards, business policies, personnel, and procedures. The following figure is an example of requirements traceability model, which is from the ADIP project (a flight control project) of DoD of the U.S. [RPSE95].

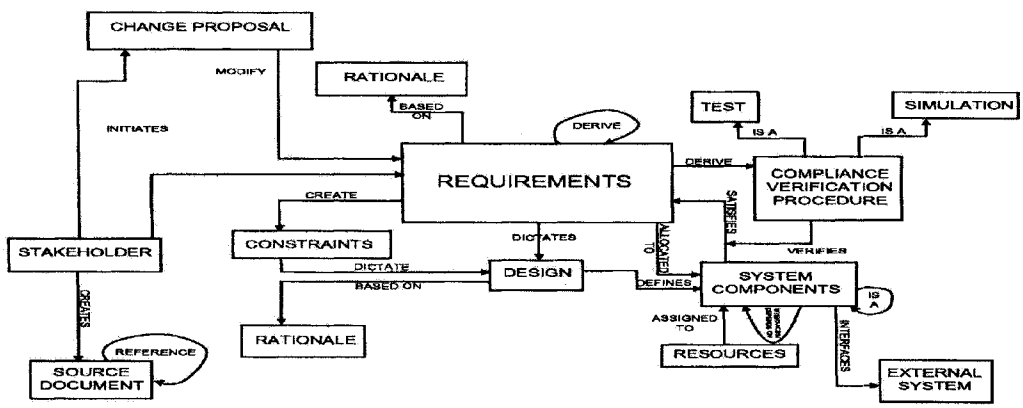

Figure 2.6 A Requirements traceability model [RPSE95].

This model shows us what kind of information should be captured when we apply requirements traceability scheme in our project. In this model, stakeholders create source documents and requirements. Stakeholders can initiate change proposal to modify requirements. Higher level requirements are iteratively refined to derive lower level detail requirements. Requirements that identify system constraints and dictate the system design

activity are explicitly identified. Also, traceability on how the requirements are allocated to the system components is captured [RPSE95].

A principal challenge in building a requirements traceability model is that it should represent and provide the semantics of various traceability linkages or relationships between requirements and system components. One must consider following important issues [RE93] when designing a traceability model:

• Bi-directional Traceability

Bi-directional traceability includes both forward and backward traceability. Bidirectional traceability can make sure that stakeholders' needs are satisfied by system components and the delivered system is what the stakeholders expect, no more and no less.

• Criticality of Requirements

To identify critical requirements is to relate them to the major task of the system. This needs a mechanism to represent the elaboration and refinement of requirements. Because we do not want to record linkages between every requirement and every output created during the system design process related to it. This will lead an uncontrollable amount of information.

• Design Rationale

Traceability linkages to represent rationale would capture the why or reason for design decisions. Tracking relationships among design objects, and understanding how and which of those objects is affected by change, is vital in the maintenance of the system. Traceability could be very useful for justifying why you did something the way you did it [RE93].

• Project Management

Traceability ensures customer satisfaction by providing us a documented means by which to prove to the customer that all of the stated requirements are met and that the job is completed [RPSE95].

Project manager can use traceability links such as status, completion date, and authorization between various components of the system for scheduling, continuity, and

security. Traceability provides a means to them to fully control the project. System engineers can utilize traceability information to capture the engineer's design rationale explaining why the system was designed the way it was. This information could prove invaluable throughout life cycle maintenance and on the development of similar systems [RPSE95].

• **Accountability**

Some accountability information should be captured to provide a better means in maintaining and revising a system. The information include: design elements designed by, validated by, and modified by development personnel.

• **Horizontal and Vertical Traceability**

Horizontal traceability refers to the traceability between objects of same type, and vertical traceability refers to the traceability between object of different types.

• **Automated Support for Traceability**

Maintaining traceability links will be extremely time-consuming and error-prone if we manually capture traceability information. So an automated requirements management tools is very important.

2.4.4 Traceability Policies

The fundamental problem with maintaining traceability information is the high cost of collecting, analyzing and maintaining that information. To help software engineers who are responsible for requirements management, it is helpful if an organization maintains a set of traceability policies which defines the traceability information to be maintained. These should normally include the following:

a) What kind of traceability information should be maintained?
b) What kind of traceability techniques should be used for maintaining traceability links?

c) A description of when the traceability information should be collected during the requirements engineering and system development processes, the roles of the people who are responsible for maintaining the traceability information.

d) The process used to ensure that the traceability information is updated after the change has been made.

Traceability policies usually have to be specialized for each project. However, whatever traceability policies are specified, it is very important that they should be realistic. Maintaining traceability information is tedious, time-consuming and labour-intensive. Very comprehensive traceability policies may be fine in principle but, if they cannot actually be implemented, they are useless.

2.5 Requirements Storage

Requirements are stored in a repository provided by computer systems such as word processor, spreadsheet and database system.

A word processing system or spreadsheet system is usually used to create the initial version of the requirements document. The requirements are stored as one or more files. Most organizations which produce requirements for small and medium-sized systems maintain their requirements in this way.

Relational databases are now the most commonly used type of database. Relational databases were designed for storing and managing large numbers of records which have the same structure and minimal links between them. A requirements database can be used to link requirements and related artifacts.

Object-oriented databases have been developed relatively recently and are structurally more suitable to requirements management. They are better than relational databases when there are many different types of entity to be managed and where there are direct links between different entities in the database. They allow different types of information to be maintained in different objects and managing links between objects is fairly straightforward.

2.6 Traceability Methods

This section examines current traceability practices and methods and discusses some of the difficulties and weaknesses related to each approach. This discussion provides a clear direction as to the types of problems that a new traceability method should attempt to solve.

2.6.1 Traceability Matrix

The traceability matrix [PAL97] is the most frequently used method for establishing traceability. In a basic traceability matrix the cells that represent relationships between the entities defined in the current row and column are marked with an "X". It is also possible for an entity to be displayed as both a row and a column so that intra-entity relationships can be defined.

The following table shows that traceability was established to design specification, class diagrams, java code and test cases.

Requirements Number	Description	Traces To
RS15	Players shall be able to move pieces on the board	DS08,CD13,JV03,TC05
RS16	Players shall be able to write comment on the whiteboard	DS08,CD13,JV04,TC06
RS18	The board shall support multiple players	DS09,CD14,JV06,TC08

Table 2.1 Traceability Matrix

The advantage of traceability matrices is that they are simple to construct and when they carry only the limited amount of information shown in these example, they are relatively simple to maintain. However, in reality, traceability links are numerous and extend between many different types of products, which make link maintenance extremely difficult.

2.6.2 Graph-based Approaches

Pinheiro and Goguen [PG96] designed TOOR for tracing requirements. TOOR stands

for "Traceability of Object-Oriented Requirements". TOOR is derived from its use for object-oriented development and its object-oriented implementation, which allows the definition of classes and subclasses of objects and relationships among objects. Of course, the requirements themselves are not necessarily object-oriented.

TOOR supports the linking of requirements to design documents, specifications, programming code, and various other artifacts through the use of relations instead of simple links [JH02]. These relationships are user-definable.

Figure 2.7 represents the configuration of objects and relations with people involved. Requirements are associated with project specification through either PartOf (e.g. TR0001) or direct relationship (e.g. requirement TR0005).

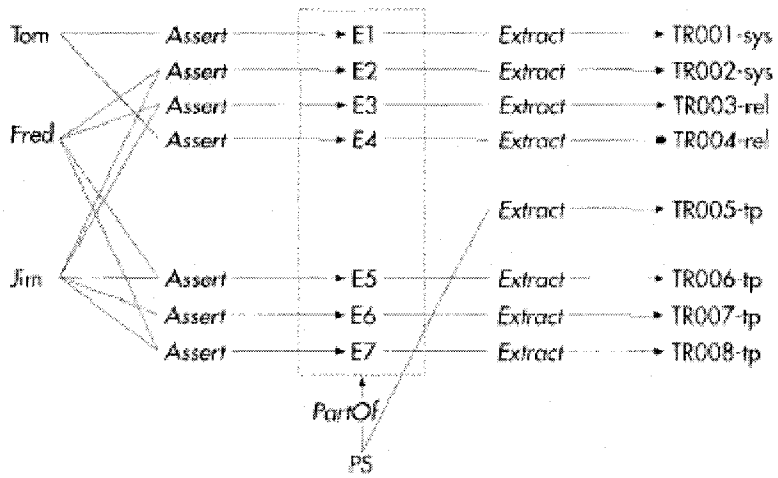

Figure 2.7 Relations in TOOR [PG96].

This approach is very powerful and expressive. It supports an extremely rich traceability scheme. But it needs professionals to give the definition of objects and relationship, and the maintenance of those links could be very complex.

2.6.3 Contribution Structures

This approach identified the inadequate pre-requirements traceability, caused by the paucity and unreliability of information about requirements production, was uncovered as a likely reason for requirements traceability problems in the longer-term [GF95]. Concepts from the social sciences are applied into requirements traceability to address some problems. It links tangible RE artifacts (contributions) to details of agents who have contributed to their production (contributors) using contribution relations.

Contribution structure refers to all the contribution relations defined for an artifact.

The relation between agents and artifacts could be defined using terms like "contributed to" and "contributed by". Contribution format defines the nature of the contribution relations, and which includes some attributes: principal, author, documentor, approved by, pending approval by, not approved, and so on.

This approach extends conventional forms of artifact-based requirements traceability with accompanying contribution structures, which thereby offers a way to accommodate the diverse forms of personnel-based requirements traceability [GF95].

2.6.4 Information Retrieval (IR)

Information Retrieval can be used in certain situations to dynamically generate links in place of user-defined explicit links [CZL04]. Most of the documentation that accompanies large software systems consists of free text documents expressed in a natural language. Examples include requirements and design documents, user manuals, logs of errors, maintenance journals, design decisions, reports from inspection and review sessions, and also annotations of individual programmers and teams [ACCD00]. Therefore, a query can be constructed from the keywords of the requirement to be traced, and based on the similarity of the query with artifacts in the search space, the retrieval algorithm then returns a set of likely links to the user [CZL04].

The following figure shows the process of traceability link recovery using IR:

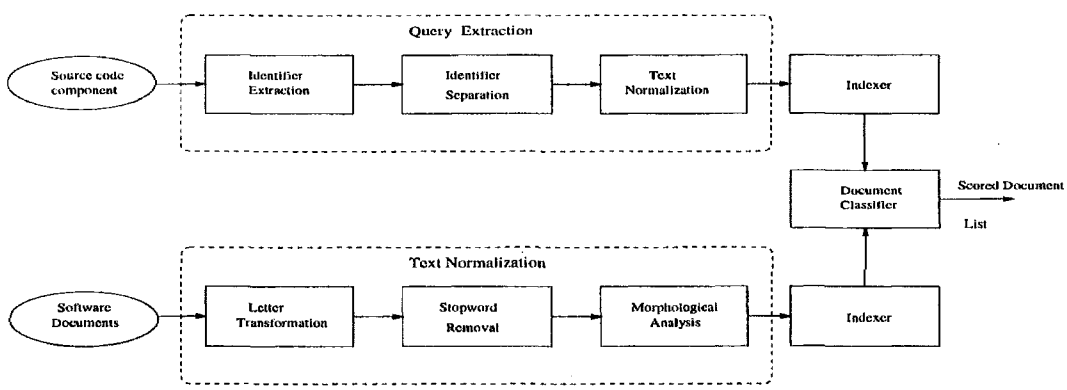

Figure 2.8 Traceability Recovery Process Using IR [ACCD00]

In the first path which is at the bottom of the picture, software documents are indexed based on a vocabulary that is extracted from the documents themselves, the second path builds and indexes a query for each source code class, finally, a classifier computes the

similarity between queries and documents and returns, for each class, a ranked list of documents [ACCD00].

The primary advantage of IR traceability techniques is that they eliminate the need for maintaining links, and when applicable can eradicate the problem of outdated and incomplete links. But, it can only work effectively if and when there is a high lexical correlation between the requirement and its related artifacts.

2.6.5 Event-based Traceability

Event-based requirements traceability [CCC03] is based upon event-notification and builds loose coupled relationship between artifacts through an event service. Figure 2.9 shows Event-based traceability architecture. It contains three main components. The requirements manager is responsible for managing requirements, monitoring changes to those requirements, and for publishing change event messages to the event server. The event server is responsible for establishing traceability by handling initial subscriptions placed by dependent entities. It also listens for event notifications from the requirements manager and forwards event messages to relevant subscribers. The subscriber manager listens on behalf of the subscribers that it manages for event notifications forwarded by the event server. Depending upon event and subscriber type, the manager either stores the incoming event message in an event log for later human-supported resolution, or else processes it automatically according to a set of predefined rules.

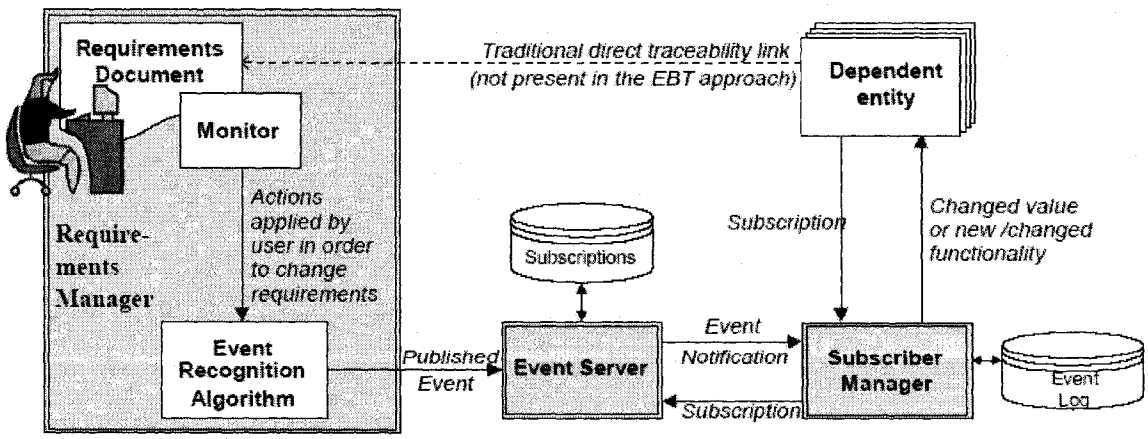

Figure 2.9 EBT Architecture [CCG02]

Event-based requirements traceability addresses several of the identified causes of

traceability failure, such as the problems related to the need for close coordination between team members, lack of visibility into the current state of the dependencies, lack of training, and the tendency of developers to fail to maintain links because of a perceived lack of immediate benefits [CCC03]. Event-based requirements traceability can also support tracing certain types of NFRs, through the use of design patterns as intermediary objects.

2.7 Summary

This chapter presents background and related research in requirements management and requirements traceability. Requirements traceability is fundamental for management of requirements and evolving requirements when developing and maintaining software systems. Traditional traceability techniques store traceability information in word processor, relational database or object-oriented database. In this thesis, I will present a new traceability technique which stores traceability information in active database.

Chapter 3 Active Rules in Active Database

Traditionally, database systems have been viewed as repositories that store the information required by an application and are accessed either by user programs or through interactive interfaces [PD99]. In such a context, database management systems (DBMS) are passive when external events happen and difficult to maintain in a consistent state.

Active database management systems (ADBMS) support mechanisms that enable them to respond automatically to events and centralize reactive semantics to increase data consistency. These advantages can be applied into requirements traceability system. By using reactive behavior in ADBMS, requirements changes can be captured automatically. By using centralized reactive semantics, requirements and their related artifacts can be maintained in a consistent state. In this chapter, active rules and rules analysis are described.

3.1 Features of Active Rules

As the scale and complexity of data management increased, interest has grown in bringing active behaviors into databases, allowing them to respond independently to data-related events. Typically these behaviors are described by event-condition-action (ECA) rules [GSS04].

ECA rules have up to three components: event, condition, and action. The *event* describes an external happening to which the rule may be able to respond. The *condition* examines the context in which the event has taken place. The *action* describes the task to be carried out by the rule if the relevant event has taken place and the condition has evaluated to true. In sum, if the specified event occurs and if the condition is true, the specified action is executed [GSS04].

There are several advantages in using ECA rules to implement reactive functionality compared to direct implementation in application code:

- ECA rules allow an application's reactive functionality to be specified and managed within a rule base rather than being encoded in diverse programs and, thus, enhance

the modularity, maintainability, and extensibility of applications.

- It promotes code reusability. Rather than replicating code in distinct applications, the code resides in a single place from which it is implicitly invoked. Such centralization accounts for increasing consistency because no application can bypass the policy, and maintenance is eased as changes to the policy are localized in a single piece of code.
- Moreover, in a client/server environment, centralized reactive behavior reduces network traffic, as the reaction associated with the event is executed locally as the single implicit invocation arises.
- ECA rules have a high-level, declarative syntax. They are amenable to analysis and optimization techniques, which cannot be easily applied if the same functionality is expressed directly in application code.
- ECA rules realize a generic mechanism that can abstract a wide variety of reactive behaviors, in contrast to application code that is typically specialized to a particular kind of reactive scenario.

3.2 Modeling

Active databases support certain applications by moving the reactive behaviors from the application into the ADBMS. Active databases are thus able to monitor and react to specific circumstances of relevance to an application. The reactive semantics is both centralized and handled in a timely manner [PD99].

An ADBMS must provide a knowledge model (i.e., a description mechanism) and an execution model (i.e., a runtime strategy) for supporting this reactive behavior [PD99].

3.2.1 Knowledge Model

The knowledge model of an ADBMS indicates what can be said about active rules in that system. The knowledge model of an active rule is considered to have three principal components, an event, a condition, and an action [PD99].

An **event** is something that happens at a point in time.

The role of a **condition** indicates whether it must be given. In ECA-rules, the condition is generally optional. When no condition is given for an ECA-rule, or where the role is none, an event-action rule results. In systems in which both the event and the

condition are optional, it is always the case that at least one is given.

The range of tasks that can be performed by an **action** is specified as its options. Actions may update the structure of the database or rule set, perform some behavior invocation within the database or an external call, inform the user or system administrator of some situation, abort a transaction, or take some alternative course of action using do-instead.

3.2.2 Execution Model

The execution model specifies how a set of rules is treated at runtime. The execution model of a rule system is closely related to aspects of the underlying DBMS (e.g., data model, transaction manager) [PD99].

The following figure shows the principal steps that take place during rule execution:

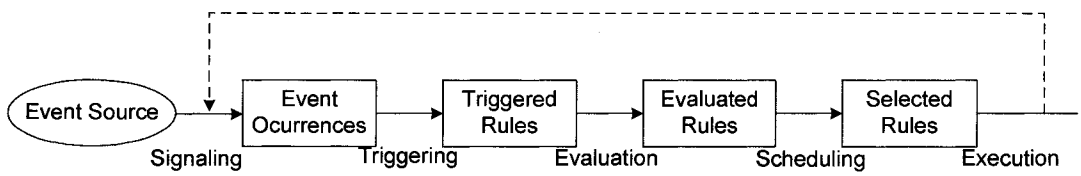

Figure 3.1 Principal steps of rule execution [PD99]

The signaling phase refers to the appearance of an event occurrence caused by an event source.

The triggering phase takes the events produced thus far, and triggers the corresponding rules. The association of a rule with its event occurrence forms a rule instantiation.

The evaluation phase evaluates the condition of the triggered rules. The rule conflict set is formed from all rule instantiations whose conditions are satisfied

The scheduling phase indicates how the rule conflict set is processed.

The execution phase carries out the actions of the chosen rule instantiations. During action execution other events can in turn be signaled that may produce cascaded rule firing.

3.3 Rule Analysis

Rules can be seen as an implementation mechanism, but implementation must be

preceded by analysis. Reactive behavior is based on business policies. Business policies are explicit statement of constrains placed on the business that concern both structural aspect and behavioral aspect. Structural aspect is concerned with the description of essential concepts, relationships, or states. Behavioral aspect is concerned with the procedures that govern how the business operates.

Recovering business policies focus on the structural aspect. It expresses the conditions that should hold in the domain. Causal business policies focus on the behavioral aspect. It reflects the procedural aspects of the organization. In this thesis, an E/R model is used to represent the structural aspect of requirements traceability, a state transition diagram is used to represent the behavioral aspect of a requirement. In this way, traceability policies are mapped into Active Database.

Chapter 4 Mapping Dynamic Behaviors of Requirements into Active Database

In this chapter, I present an approach to map the dynamic behaviors of requirements change into active databases. I shall use a state transition diagram to define the states or stages in the lifecycle of a requirement. An E/R model is used to represent the structural patterns of requirements traceability in active database. Lastly, I will present the system architecture for managing requirements changes.

4.1 Types of Requirements Change

Software development is a dynamic process, this causes software requirements change while development is still in process. Identifying and characterizing the nature of requirements changes could lead to more effective management of changing requirements [NZW04]. As system's environments change or customers develop a better understanding of their real needs, requirements changes are inevitable. This kind of change is considered as external change. However, during the development of software system, a requirement will undergo different development stages. In each stage, this requirement may have different set of related artifacts, or the relationship between this requirement and its related artifacts changes. This kind of change is considered as internal change. Figure 4.1 shows external change and internal change of a requirement in requirements traceability scheme.

The formal definitions of external change and internal change are given below:

External Change: Relating to, or connected with the outside or an outer part, such as system's environments or customer's new needs. It is explicit.

Internal Change: Of, or located within the surface, such as relationships between a requirement and its related artifacts during the development of a software system. It is implicit.

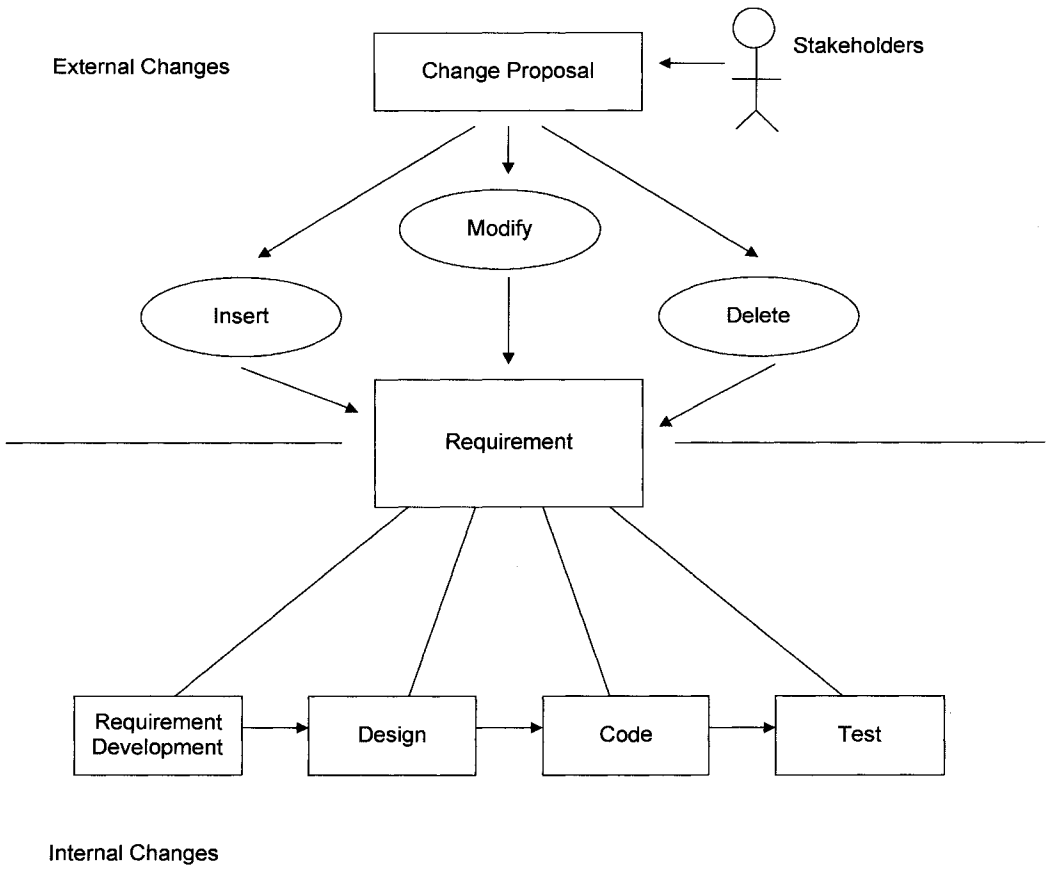

Figure 4.1 Requirements Change Types

Traditional traceability techniques such as matrix, IR and EBT have focused on external change. When we trace critical requirements which are the central mission of the system, there is a tendency for the traceability infrastructure to erode over its lifetime, as time-pressured practitioners fail to consistently and systematically update each and every link when changes occur [CCC03]. Then it will fail to reflect the current and accurate state of relationships between requirements and their related artifacts. Apparently, mechanisms of monitoring maintenance of requirements change are lacking in the traditional traceability techniques.

The introduction of internal change into requirements traceability scheme provides a means to monitor requirements traceability maintenance process. This mechanism ensures that practitioners must follow requirements traceability policies. Maintenance of requirements change are also monitored and controlled by active rules. In this thesis, the proposed method will consider both external and internal change.

4.2 The Attributes of Requirement, Design and Component Tables

Relational databases are now the most commonly used type of database. Requirements are maintained in a database with each requirement represented as one or more database entities. The facilities of the database can be used to link related requirements and it is usually possible to formulate fairly complex database queries to identify requirements groupings. The database may provide some version control facilities or, at least, provision for these facilities to be implemented.

In this thesis, a small project (Chess game software system) is used to validate the proposed method. First, the requirements of this system are elicited. Then, based on those requirements, the design specifications are defined and described. Finally, the system components are developed to satisfy those requirements.

In Chapter 2 section 2.3.3, I discussed that using appropriate attributes of requirements is very important for efficient requirements management. In this section, the attributes of requirements, design specifications and system components which will be used in Chess game software system are introduced below. The detailed data definition of this system is presented in Appendix A.

Table 4.1 shows data definition of requirements which have 11 attributes:

REQUIREMENT
Identifier: TEXT Statement: TEXT | GRAPHIC Date_entered: DATE Date_changed: DATE Rationale: Rationale_ID Stakeholder: Stakeholder_ID Design: Design_ID Status: STATUS Dependants: REQ_LIST Is_dependent_on: REQ_LIST Level: TEXT Comments: TEXT

Table 4.1 A Requirement Entity

A requirement has the following attributes:

1. Identifier

This is a simple text string which is assigned when a requirement is created in the database and it is the key of this table.

2. Statement

This is a statement of the requirement which may be natural language text or a diagram.

3. Date_entered

The date that the requirement was originally entered in the database.

4. Date_changed

The date of the last alternation to the requirement.

5. Rationale

This is a reference to a set of information which provides a rationale explaining why the requirement has been included. The associated information may include text, diagrams or photographs.

6. Stakeholder

This is a reference to stakeholders who are responsible for this requirement.

7. Design

This is a reference to a design entity which is related to this requirement.

8. Status

This is a variable representing the status of the requirement. The status may be 'new', 'waiting', 'satisfied', 'pending', 'testing' and 'inactivated'.

9. Dependants

This is a list of references to requirements which depend on this requirement.

10. Is_dependent_on

This is a list of references to requirements on which this requirement depends.

11. Level

Level number which shows the position of a requirement in a requirement tree structure.

12. Comments

This is any other information which may be useful. In this thesis, I use this field to record event.

Table 4.2 shows data definition of designs which have 12 attributes:

DESIGN
Identifier: TEXT Statement: TEXT | GRAPHIC Type: TEXT Date_entered: DATE Date_changed: DATE Stakeholder: Stakeholder_ID Requirement: Req_id Component: Component_ID Status: STATUS Dependents: DESIGN_LIST Is_dependent_on: DESIGN_LIST Level: TEXT Comments: TEXT

Table 4.2 A Design Entity

1. Type

The type of design specification.

2. Component

This is a reference to a system component which fulfills this design.

3. Status

The status may be 'null', 'designing' and 'designed'.

Table 4.3 shows data definition of components which have 12 attributes:

COMPONENT
Identifier: TEXT Statement: TEXT | GRAPHIC Type: TEXT Date_entered: DATE Date_changed: DATE Stakeholder: Stakeholder_ID Design: DESIGN_LIST Test: Test_ID Resource: Resource_ID Status: STATUS Dependents: COMPONENT_LIST Is_dependent_on: COMPONENT_LIST Level: TEXT Comments: TEXT

Table 4.3 A Component Entity

1. Test

This is a reference to a test case which verifies satisfaction with requirements.

2. Resource

A description of any and all resources that are managed, affected, or needed by this entity.

3. Status

The status may be 'null', 'developing' and 'developed'.

In practice, it is almost impossible to define a schema which covers everything. The proposed method will only focus on functional-requirements traceability.

4.3 State Transition Diagram

A state transition diagram [PVC98] specifies the lifecycle of a requirement. It shows the possible sequences of state transitions and the operations that make the state transitions. Nodes in the diagram represent the various states of a requirement, and arcs denote state transitions caused by events applicable to that requirement. The state transition diagram in Figure 4.2 is used to specify the lifecycle of software requirements.

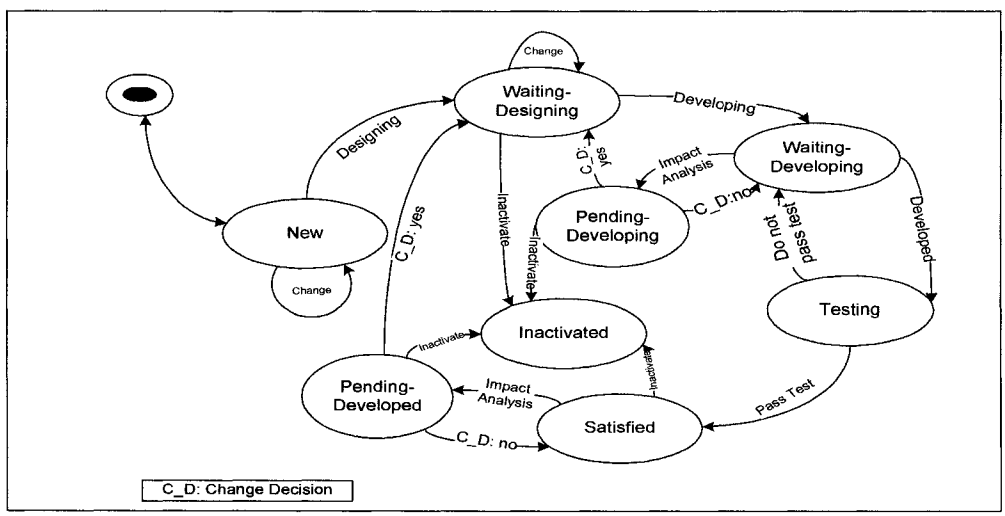

Figure 4.2 Requirements State Transition Diagram

As shown in Figure 4.2, a requirement can go through the following states:

1. *New*: A newly created requirement is in the *new* state. A requirement in the new state can be modified by a requirements engineer.
2. *Waiting*: After a requirement in the new state is assigned to design components or it is

modified, the requirement enters the *waiting* state. Modifying a requirement in the pending state also transfers the requirement to the waiting state.

3. *Testing*: After the development of the designed components that are related to a requirement is finished, the requirement enters the *testing* state. If the test of the software components against the requirement passes, the requirement enters the *satisfied* state; otherwise, the requirement returns the *waiting* state for re-engineering or re-development.
4. *Satisfied*: The requirement has been satisfied or realized by certain system components or the system components related to the requirement has passed specific test against the requirement.
5. *Pending*: When a change is made or proposed for a requirement that is in the satisfied state or waiting-developing state, we need an impact analysis for the change. The impact analysis may lead to one of three possibilities: The change is aborted, and the requirement is returned to the previous state; the requirement is changed as expected and it enters the waiting state; and the requirement is inactivated.
6. *Inactivated*: After a requirement is inactivated or disposed, the requirement enters the inactivated state.

The state transition diagram describes state changes of requirements based on events of action or operation execution for requirements. The state transition diagram in Fig. 5.2 uses the following events:

External change events:

1. *New*: Create a new requirement.
2. *Activate*: Activate an inactivated requirement.
3. *Inactivate*: Inactivate an active requirement.
4. *Change*: Modify a requirement when change decision is yes.
5. *Impact analysis*: Impact analysis is performed to determine how a change may impact on the existing system.
6. *Change Abort*: A change is aborted when change decision is no.

Internal change events:

1. *Designing*: Begin to design for a requirement.

2. *Designed*: The design for a requirement is finished.
3. *Developing*: Begin to develop for a requirement.
4. *Developed*: The development of related artifacts for a requirement is finished.
5. *Pass*: System components pass the test.
6. *Do not pass*: System components do not satisfy the related requirement and the requirement needs re-develop.

Based on the above states and events, two examples are given below to show the active rules for the impact analysis event and the change event.

RuleName: Impact Analysis
RuleStatus: Active
On a change proposal is introduced for a requirement
If State == *Satisfied or Waiting-Developing*
Do query all the related artifacts;
  set State to *Pending*;
  send the report to software engineers;

RuleName: Change
RuleStatus: Active
On update a requirement
If State == *Pending* and the decision of change == Yes
Do modify the requirement;
  Set State to *Waiting-Designing*;
  Set the State of related artifacts to *Designing or Null*;
  Notify the designers and developers;
If State == *Pending* and the decision of change == No
Do Set State to the previous state
  Notify the software engineers;
If State == *New or Waiting-Designing*
Do Modify the requirement;
  Notify the software engineers

These active rules are placed on requirements table to monitor requirements change. When an external or internal event happens, the active rule evaluates the conditions according to the type of the event, and then takes appropriate action to make sure the state of a requirement follows the sequence which is described in the state transition diagram.

4.4 E/R Model

Figure 4.3 presents a high-level entity-relationship (ER) model for modeling requirements traceability with active rules:

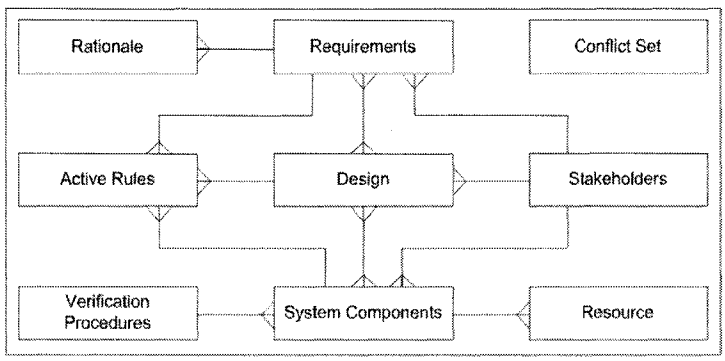

Figure 4.3 An E/R Model for Requirements Traceability

In the above figure, each of the entity types except active rules and conflict set is represented using a table in SQL. The SQL create table commands for the corresponding tables are provided in Appendix A.

A requirement is created by a stakeholder and a stakeholder can create many requirements. A requirement should base on rationales. A design specification is dictated by certain requirements and it should be satisfied by system components. Moreover, a system component relates to certain resources and should be verified by a verification procedure.

Stakeholders could be the program sponsor (customer), the project manager, the system analyst/designer, the test engineer, system maintenance personnel, or the end user of the system [RPSE95]. A major use of traceability is to provide accountability [RE93]. The accountability information include: design elements designed by, validated by, and modified by development personnel [RE93]. The availability of such information will be

indispensable in maintaining and revising a system [RE93].

Rationale information is another important component of traceability. Traceability linkages to represent rationale would capture the why or reason for design decisions [RE93].

Conflict Set is a queue of triggered rules waiting to be fired. Verification Procedures is to verify the satisfaction of system components with requirements. Resource is to record the information of hardware, software and so on that are allocated to a system component.

4.5 System Architecture

Based on the above descriptions of the dynamic behavior of a requirement and the structural feature of requirement traceability scheme, a full working version of requirements management tool can be developed. The following is the proposed system architecture.

The system architecture is illustrated in Figure 4.4. This architecture consists of the six main components: 1) The Requirements Manager; 2) The Design Manager; 3) The System Components Manager; 4) The Impact Analysis Manager; 5) The ECA Rule Engine; 6) The Database System – an ADBMS with trigger and stored procedures mechanism.

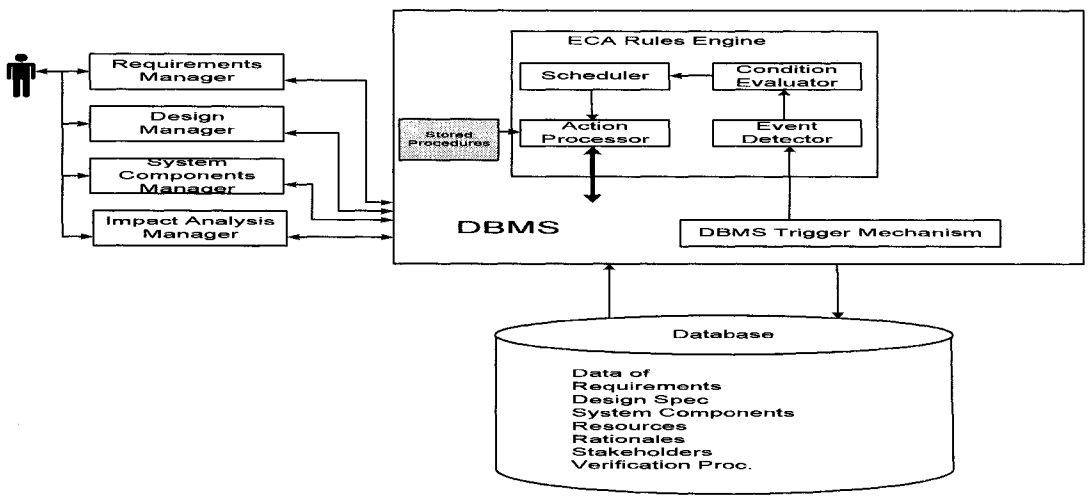

Figure 4.4 Requirements Management System Architecture

The Requirements Manager allows the creation, deletion and modification of

requirements. It permits the user to create, inactivate, decompose, refine, modify, merge and replace requirements.

The Design Manager allows the creation, deletion and modification of design specifications. It permits the user to modify the root design specification and create the sub-detailed design specifications.

The System Components Manager allows the creation, deletion and modification of system components. It permits the user to modify the root system components and create the sub-detailed system components.

The Impact Analysis Manager allows the query of related design specifications and system components when a change is introduced to a requirement. It also permits the user to input information of why to make such change, who approves it, who will do it, and when it will be done.

The ECA Rules Engine is the kernel of this system. It consists of event detector, conditions evaluator, scheduler, action processor.

The database serves the purpose of storing requirements, design specifications and system components. The trigger mechanism drives the execution of update of requirements.

In this thesis, two components of this architecture are implemented: ECA Rules Engine and Database.

Chapter 5 Interaction among Requirements, Designs, and Components

5.1 Triggered Message Sequence Charts

Triggered Message Sequence Charts (TMSC) [SC02] describe system scenarios in terms of the sequences of atomic actions (message sends and receives, and local actions) that each parallel process (or instance) may engage in [SC02]. In this section, a brief overview of the visual syntax of TMSC is given. For the details of the syntax and semantics of TMSC, readers are suggested to refer to [SC02].

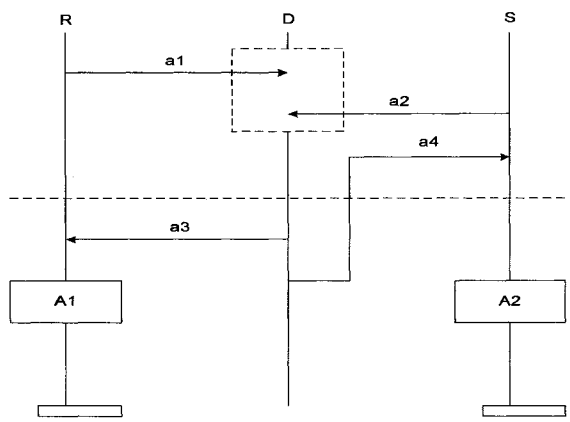

Figure 5.1 An Example TMSC

Graphically, TMSC can be represented as in Figure 5.1. The partitioning of the sequence of events of an instance into the trigger and action sequences is indicated by a horizontal line. For each instance, the sequence of events above the line constitutes its trigger, while the sequence below the line constitutes its action. The presence of a small bar at the foot of each instance indicates that the instance cannot beyond this point in the TMSC, while the absence means that behavior of this instance beyond the TMSC is left unspecified i.e. there are no constraint on its subsequent behavior. The TMSC in Figure 5.1 consists of three instances: a requirement R, a design D, and a system component S. The TMSC in Figure 6.1 may be read as follows:

If R sends a1 to D, then it should receive a3 from D. After R receives a3, it should perform the local-action A1 and terminate; if D receives a1 from R and a2 from S in any order, then it should send a3 to R and a4 to S, and its subsequent behavior is left

unspecified; if S sends a2 to D and receives a4 from D, then it should perform the local-action A2 and terminate.

5.2 Basic Interactions

In this section, an example is given to illustrate basic interaction between requirement (R), design (D) and system component (S) using Triggered Message Sequence Charts.

The example in Figure 5.2 shows the basic interaction between requirement, design and system component when a new requirement is created.

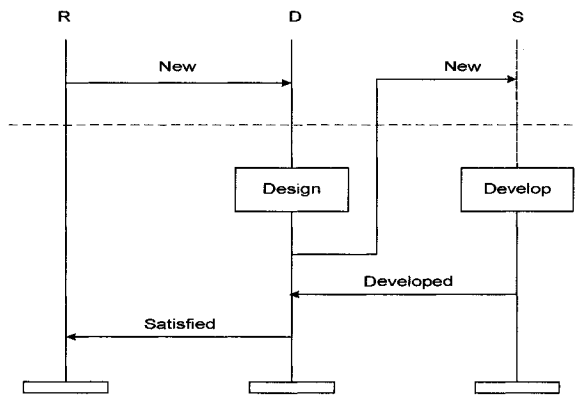

Figure 5.2 Interactions among requirements, designs and system components when a new requirement is created.

Chapter 6 Validation and Analysis

The basic structural and behavioral features of requirements in an active database have been introduced, now I discuss how these active rules are supported in concrete systems. In this chapter, first, the active rules in Oracle Database are described; secondly, the development processes of active rules in Oracle Database are introduced; thirdly, the proposed method is implemented in Oracle Database; finally, the case studies are conducted to validate the proposed method.

6.1 Active Rules in Oracle Database

Active rules are known as *triggers* in Oracle Database. Triggers are stored procedures that are invoked by Oracle in response to database INSERT, UPDATE and DELETE SQL statements. Triggers can execute PL/SQL statements, call stored procedures, and raise errors. The PL/SQL code executed within a trigger may include additional INSERT, UPDATE and DELETE statements that potentially invoke other triggers.

Oracle follows an event-condition-action approach for the description of triggers. For example:

```
create or replace trigger new_req
after insert on Requirement
for each row
when (new.status = 'new')
begin
        insert into DESIGN values
        (:new.req_no, 'NEW', NULL);
        DBMS_OUT.PUT_LINE('New requirement has just been created!');
end;
```

Every trigger has a name (in this case, new_req). The event definition describes the happening to which the rule may have to respond, such as the insertion, update or deletion of a tuple. The *"for each row"* clause indicates that this is a trigger with tuple_level transition granularity that has an immediate coupling mode. The condition is declared in the *when* clause, and it is a Boolean expression. The action is a PL/SQL block. PL/SQL blocks are delimited by begin and end.

6.2 Trigger Development Process

In this section, the processes of developing triggers in Oracle Database are described which include the following six steps:

1) Identify rules for procedural enforcement.
2) Construct a constraints violation/enforcement list (CVL/CEL).
3) Create the trigger functional description (TFD).
4) Identify errors raised for processing requirements.
5) Encapsulate functionality into a constraints package.
6) Write triggers and test.

1) Statement of rules

The origin of any software effort, including a simple constraint, begins with a statement of the requirement, or in this case, a statement of the rule. Some of the requirements/rules are:

- *A Requirement must have a unique identifier (rule 1).*
- *A Requirement Status must be in ("New", "Waiting", "Pending", "Satisfied", "Inactivated", "Testing") and the transition of status must follow the sequence stated in figure 4.2 (rule 2).*
- *A Component must have a valid Design Identifier (rule 3).*

2) Construct a constraints violation list

The second step is to construct a constraints violation list (CVL). This is a list of database actions that have the potential for violating a rule. For the rules stated above, we have the following CVL:

- *We can INSERT a requirement with a NULL identifier (rule 1).*
- *We can UPDATE the status of a requirement from "Testing" to "Pending" (rule 2).*
- *We can INSERT a Component with a NULL Design identifier (rule 3).*

The difference between a rule and the CVL is the rule written by someone who understands the requirement, where the CVL is written by someone who understands the database design and has an insight to how the rule can be violated.

3) Trigger functional description

The purpose of this step is to construct the trigger functional description. It describes the high-level logic of each trigger, its data requirements, and trigger level, for example, row or statement. There are three tables that need triggers:

- *Requirement*
- *Design*
- *Component*

The trigger functional descriptions are:

1. Requirement INSERT (rule 1)

Description: This trigger will detect a rule violation if the identifier is NULL or duplicated with other identifier.

Data requirements : :NEW.REQ_ID

LEVEL This is ROW because the trigger needs access the correlation values.

2. Requirement UPDATE (rule 2)

Description: The trigger will inspect the NEW:STATUS and OLD:STATUS correlation values. A violation will exist if the NEW:STATUS is "Testing" and OLD:STATUS is "Pending" .

Data requirements: :NEW.STATUS, :OLD.STATUS

LEVEL: This is ROW because the trigger needs access to the correlation values.

3. Component INSERT (rule 3)

Description: This trigger will detect a rule violation if the Design identifier is NULL.

Data requirements: :NEW.DESIGN_ID

LEVEL: This is ROW because the trigger needs access to the correlation values.

4) Error/Process Analysis

The purpose of this step is to identify the error messages for each rule violation. For each of these triggers, we can potentially raise an application error. Following is the package specification:

```
PACKAGE ERRORS IS
    REQ_ID_VIOLATION CONSTANT INTEGER := -30001;
    INVALID_STATUS_TRANS CONSTANT INTEGER := -30002;
    INVALID_DESIGN_ID CONSTANT INTEGER := -30003;

   REQ_ID VIOLATION_MSG VARCHAR2 (30) NOT NULL :=
   "CANNOT INSERT REQUIREMENT";
   INVALID_STATUS_TRANS_MSG VARCHAR2 (30) NOT NULL :=
   "CANNOT UPDATE THE STATUS";
   INVALID_DESIGN_ID_MSG VARCHAR2 (30) NOT NULL :=
   "DESIGN_ID NEEDED";
END ERRORS;
```

Not all triggers raise an application error in response to a constraint violation.

5) Encapsulate functionality into a constraints package

This step creates the specification and body of a constraints package. This is a PL/SQL package that encapsulates the rules of the procedural constraints. Each procedure in the package accepts arguments that are passed as trigger correlation values. The procedure determines if a violation has occurred and raises an application error.

6) Code the trigger

Having placed the trigger's functionality in a package, developing the trigger code is a straightforward process. Following is an example for the second constrains violation list trigger:

```
Create or replace trigger requirements_bus
Before update on requirements
Begin
    requirements_table_pkg.clear_table;
End;
```

In above *Before-Update-Statement* trigger, it clears correlation table in PL/SQL table package.

```
Create or replace trigger requirements_aur
After update on requirements
For each row
Begin
    requirements_table_pkg.insert_values;
End;
```

In above *After-Update-Row* trigger, it inserts correlation values into PL/SQL table.

```
Create or replace trigger requirements_aus
After update on requirements
Begin
    requirements_cons_pkg.enforce_rule_3;
End;
```

In above *After-Update-Statement* trigger, it calls enforce_rule_2 functions from Constrains package.

6.3 Implementation

In Chapter 5 section 5.2, the basic interactions between requirement, design and component using a simple example were illustrated. A requirement can be in one of 8 states: *New, Waiting-Designing, Waiting-Developing, Testing, Satisfied, Pending-Developing, Pending-Developed and Inactivated.* A design can be in one of 3 states: *Null, Designing* and *Designed.* A component can be in one of 3 states: *Null, Developing* and *Developed.* The possibilities of combination of states of three are 72, but only 8 of those are considered as useful. The following table shows that at any time the states of requirement, design and component must be in one of them.

	Requirement	Design	Component
1	New	Null	Null
2	Waiting	Designing	Developing
3	Waiting	Designed	Developing
4	Testing	Designed	Developed
5	Satisfied	Designed	Developed
6	Pending	Designed	Developing
7	Pending	Designed	Developed
8	Inactivated	Inactivated	Inactivated

Table 6.1 The possibilities of combination of states

For a *CHANGE* event, every time a change is introduced to a tuple in requirements table, there are six triggers to enforce the integrity of status between requirements and their related artifacts. Those triggers are *BEFORE-INSERT-STATEMENT* trigger, *BEFORE-INSERT-ROW* trigger, *AFTER-INSERT-STATEMENT* trigger, *BEFORE-UPDATE-STATEMENT* trigger, *BEFORE-UPDATE-ROW* trigger and *AFTER-UPDATE-STATEMENT* trigger. Every time a trigger is fired, the trigger will call functions and procedures that defined in the PL/SQL packages and constraint packages to maintain traceability links and synchronize the status of requirements and their related artifacts.

The following is an example which shows how triggers of requirements table handle requirements change:

1. SQL INSERT STATEMENT:

```
Insert into requirements values
('R30',                 /****Requirement Identifier****/
'The user must provide a self-image that will be used to represent them
during play',           /****Requirement Statement****/
SYSDATE,                /****Date-Entered****/
'',                     /****Date-Changed****/
RAT30,                  /****Rationale ID****/
STA5,                   /****Stakeholder ID****/
'',                     /****Design ID****/
'NEW',                  /****Status****/
'',                     /****Child****/
'R18',                  /****Parent****/
'LEVEL1',               /****LEVEL****/
'Refine R18'            /****Comment****/);
```

This SQL Insert statement is used to insert a new requirement to the traceability database.

2. UPDATE PARENT REQUIREMENTS

```
If rec.status='NEW' and rec.comments<>'LEVEL0'
   and rec_is_dependent_on<>''
Then
   update requirements set requirements.dependants=rec.is_dependent_on
   where requirements.req_id = rec.is_dependent_on;
   insert into req_list values
   (rec.is_dependant_on,'Its child is',rec.req_id);
end;
```

The above function is encapsulated in constrains package. When a new requirement is inserted into the traceability database, the trigger placed on the requirements table will be

fired and it will call this function in constrains package to update its related parent requirements.

3. UPDATE CHILD REQUIREMENTS

```
If rec.status='NEW' and rec.comments<>'LEVEL0' and rec.dependants<>''
Then
   update requirements set requirements.is_dependent_on=rec.dependants
   where requirements.req_id = rec.dependants;
   insert into req_list values
   (rec.req_id,'Its child is',rec.dependants);
   Delete from req_list
   where req_list.req_id=rec.is_dependent_on and
         req_list.next=rec.dependants;
end;
```

The above function is encapsulated in constrains package. When a new requirement is inserted into the traceability database, the trigger placed on the requirements table will be fired and it will call this function in constrains package to update its related child requirements.

4. UPDATE RELATED ARTIFACTS

```
req_id1      requirements.req_id%type;
req_id2      requirements.req_id%type;
lev_id       requirements.lev%type;

begin

    req_id2 := rec.req_id;

    loop
    select req_id into req_id1 from req_list where next = req_id2;
    select lev into lev_id from requirements where req_id=req_id1;
    If  lev_id = 'level0'
    Then exit;
    End if;
    req_id2 := req_id1;
    End loop;

    /**dbms_output.put_line(req_id1);**/

    update designs set status = 'Designing' where req_id = req_id1;

end;
```

The above function is encapsulated in constrains package. When a new requirement is inserted into the traceability database, the trigger placed on the requirements table will be fired and it will call this function in constrains package to update its related designs.

6.4 Validate the Proposed Method

To assess the effectiveness of the proposed method, two case studies are implemented to test the ability of the proposed method to perform forward and backward traceability and to manage requirements change. These case studies are conducted on the environment of Oracle Database 10g and Windows 2000 Server. A small project (Chess Game Software System) is used to test the proposed method. First, system requirements are elicited; second, design specifications are defined according to the related requirements; third, the simulated coding information is described. At the same time, the information of requirements, design specifications, coding and other related artifacts are put into the proposed system to establish traceability links.

The following case studies demonstrate how the proposed method provides the traceability information for impact analysis when a change proposal is introduced and how the proposed method handles the requirements change when the change decision is made.

6.4.1 Case Study 1: Impact Analysis

In this case study, an attempt to enhance the registration of Chess Game Software System was made by introducing a new requirement that "R30: The user must provide a self-image that will be used to represent them during play". Before the introduction of this requirement, impact analysis should be performed to determine which artifacts would be impacted, and where functions should be added.

First, a requirement was identified which would be impacted by this introduction. The database was searched using a keyword of "register". The requirement identified is "R18: A user must register before being eligible to enter the board space". Next, a query was conducted to find out which artifacts would be impacted by refining this requirement. Finally, an analysis of dependencies on requirement R18 resulted in the dependency tree shown in following figure 6.1:

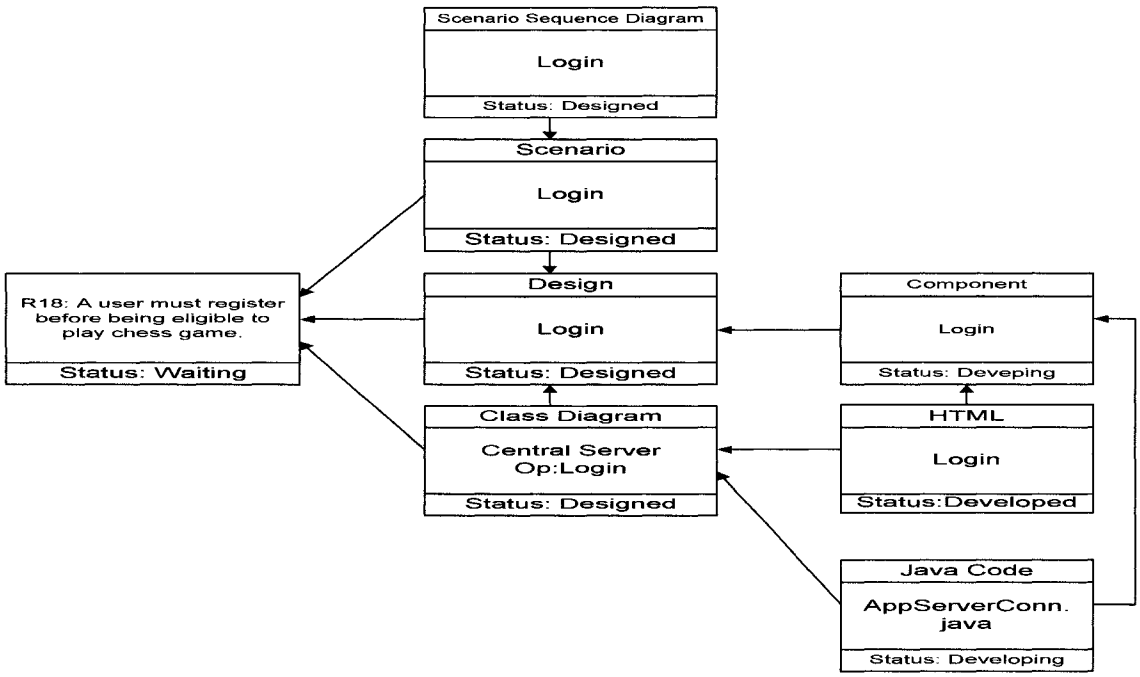

Figure 6.1 Artifacts impacted when a change is introduced.

This case study demonstrated that the proposed method has the ability to identify the artifacts that would be impacted by a change request. And the status of requirement and related artifacts are clear, it provides great insight for software engineers who will perform impact analysis.

6.4.2 Case Study 2: Change Management

This case study is to evaluate the effectiveness of the proposed method on handling requirements change. Following case study 1, after the change decision was made, the new requirement R30 was inserted into database and made it as a child of requirement R18. Then a set of query was conducted, dependency tree is shown in following figure 6.2:

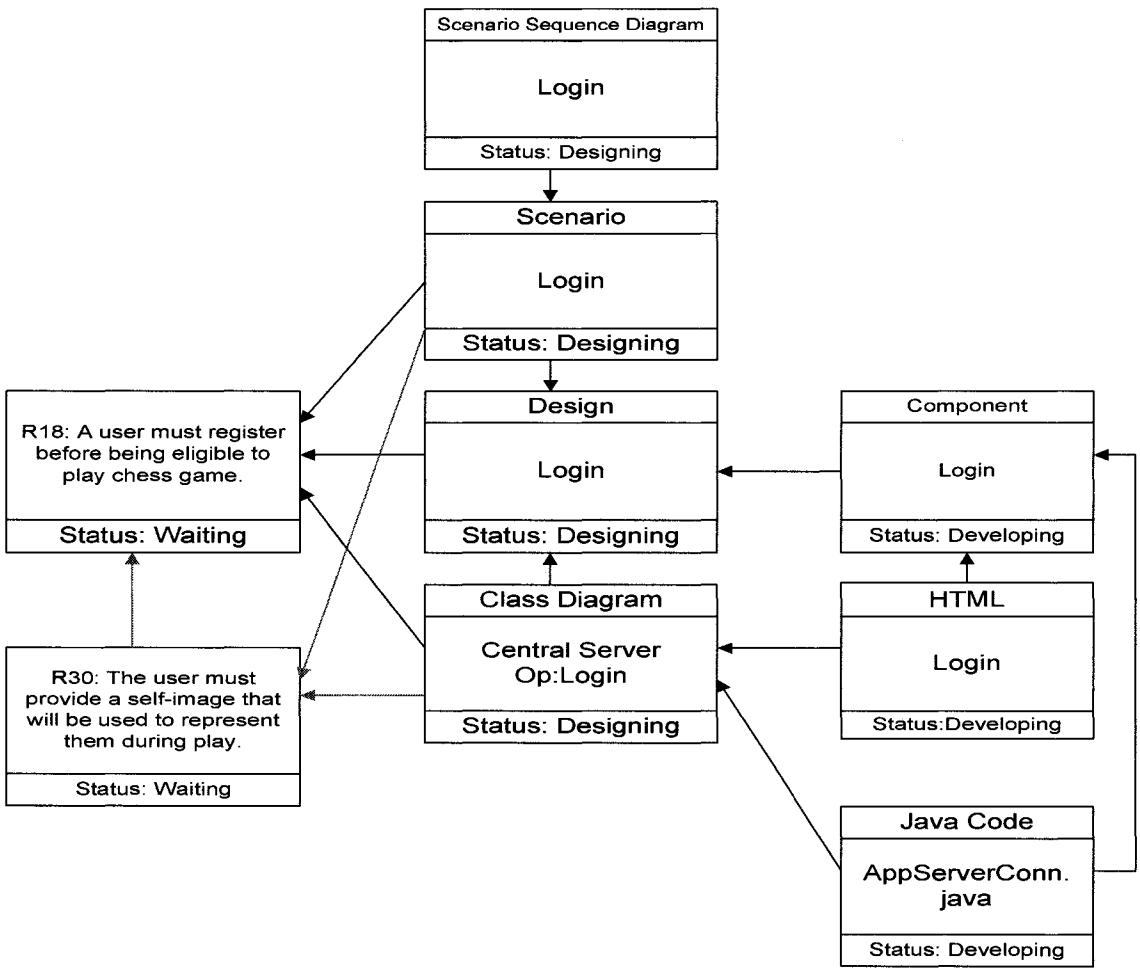

Figure 6.2 Change Management

This case study demonstrated that the proposed method has the ability to maintain traceability links when a requirement is changed. The requirements were put in the state of "Waiting", after designing, developing and testing phases, the requirements are satisfied. The whole process is monitored by active rules. Hence, the proposed method demonstrated the effectiveness of handling requirements change.

6.5 Analysis

In the previous sections, I discussed how to use Oracle trigger to implement the proposed method. From the results of experimentation, the proposed method demonstrated the effectiveness of providing forward and backward traceability and handling requirements change.

Requirements traceability supports critical tasks in both the development and maintenance phases of a software project, but to be effective, the scheme must be maintained in an accurate state. This approach maintain a traceability scheme over a long period of time by defining relationships which have the ability to evolve in response to not only external change events, but also internal change events. The status of requirements and related artifacts are clear, it provides great benefit for people who perform impact analysis.

Traceability policies usually have to be specialized for each project. It decides on exactly how traceability information should be represented, the responsibilities for traceability information collection. This approach uses Active Database to place constrains on traceability maintenance process, no one can bypass those policies.

However, maintaining traceability information is time-consuming, tedious and labour-intensive. The proposed method is a new technique for tracing critical functional requirements in finely-grained manner. It is relatively expensive, but if we pay short-term pain, we will get long-term benefit.

6.5.1 Compare with Event-based Traceability (EBT)

Event-based Traceability is based upon event notification and build loosely coupled relationships between requirements and their related artifacts. I have introduced EBT in the section 2.6.5. Table 6.2 shows the major differences between EBT and the proposed method:

	EBT	**The proposed method**
Relationship	**Loosely coupled**	**Tightly coupled**
Event monitor mechanism	**ECA + Database**	**Active Database**
Tracing	**Performance requirements**	**Critical functional requirements**
Suited for	**Coarsely-grained manner**	**Finely-grained manner**

Table 6.2 Major differences between EBT and the proposed method

6.5.2 Compare with Version Control

The output of the software process is information that may be divided into three broad

categories [ROG05]: (1) computer programs (both source level and executable forms); (2) work products that describe the computer programs (targeted at both technical practitioner and users), and (3) data. The items that comprise all information produced as part of the software process are collectively called a software configuration [ROG05].

Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process [ROG05]. A version control system has following major capabilities: (1) a project database that stores all relevant configuration objects; (2) a version management capability that stores all versions of a configuration object; (3) a make facility that enables the software engineer to collect all relevant configuration objects and construct a specific version of the software.

The Concurrent Versions System (CVS) is a widely used tool for version control. Originally designed for source code, but useful for any text-based file. Following table shows the major differences between CVS and the proposed method:

	CVS	The Proposed Method
Focus on	Source code	Requirement
Change control	No	Yes
Relationship	Horizontal	Vertical and Horizontal

Table 6.3 Major differences between CVS and the proposed method

6.5.3 Project Management

Traceability ensures customer satisfaction by providing us a documented means by which to prove to the customer that all of the stated requirements are met and that the job is completed [RPSE95]. In the process of developing large, complex systems or safety-critical systems, it is very important to minimize the possibility of missing a stated or derived requirement. For a critical requirement which is related to the central mission of system, missing even a single traceability link could be catastrophic when a change request is introduced to this requirement. The proposed method provides a means for tracing critical requirements in a finely-grained manner.

The state of requirements and their related artifacts provided by the proposed method

gives the project manager great insight of a project. The project manager is more concerned with the daily progress of the product. The proposed method is able to show that how many requirements are implemented and verified, how many requirements are implemented, but not verified, and how many requirements are not implemented. So, the project manager feels that she/he is in full control of the project.

6.5.4 Reconcile Technical and Social Aspects

Requirements engineering is not only a technical issue, but also a social issue. A new requirements traceability technique should consider both technical and social aspects. In the technical side, the proposed method introduces internal change into requirements traceability scheme which minimizes the possibility of missing traceability links. In the social aspect, the proposed method places centralized constraints on traceability maintenance process to ensure no one bypass traceability policies. The well integration of these two aspects proves that the proposed method suits to tracing critical requirements.

Chapter 7 Conclusion and Future Work

In this thesis, a new traceability technique for mapping the dynamic behavior of requirements into active relational databases was proposed. A modeling framework, architecture and its execution flow were presented.

A state transition diagrams is used to represent dynamic behavior of requirements. An E/R model represents structural behavior of requirements. The architecture proposed considers applications developed using Oracle Databases. The full working version of this system could be relatively complex, and it can be implemented on any database system which supports active rules, such as Oracle, DB2.

The main feature of this method is the extension of reactive capability supported by an underlying active database system in order to manage changing requirements. The advantages of this approach are that both static and dynamic aspects of requirements are integrated and mapped to the active database. In addition, requirements and related artifacts are stored in the relational database which makes the management of changing requirements efficient and effective. Moreover, it also provides requirements management and project management.

The proposed approach is well suited for the integration of requirement behavior with active relational databases, because of its simplicity and its ability to deal with the complexity of requirements change.

This research makes the following contributions to management of requirements change:

First, traditional traceability methods focus on external requirements change. In this thesis, a new concept of internal change is introduced into requirements traceability scheme. This approach extended requirements change to not only requirements themselves, but also status changes of requirements in the different development stages. It also provides a mechanism to monitor requirements change management process.

Second, it provides a method to map the dynamic behavior of requirements into active database which places constrains on traceability maintenance process to ensure no one bypass those policies.

This thesis only focuses on tracing functional requirements. Next steps will be further experimentation in enhancing pre-requirements traceability functions and tracing non-functional requirements.

Pre-requirements traceability is concerned with those aspects of a requirement's life prior to its inclusion in the requirements specification [GF94]. Empirical findings identified that insufficient pre-requirements traceability is the main contributor to continuing requirements traceability problems. The main reason is the invisibility of the individuals and groups that gave rise to the requirements artifacts [GF95]. Our approach can be extended for providing more pre-requirements traceability by integrating with other techniques.

Non-functional requirements define the overall qualities or attributes of the resulting system. They are critical to the successful implementation of almost every nontrivial software system [CS03]. Traditional traceability techniques have focused upon the functional requirements of the system, however if non-functional requirements such as performance, reliability, and safety are not considered, then functional changes may introduce unexpected side effects which will degrade the system quality.

References

[ACCD00]G. Antoniol, G. Casazza and A. De Lucia, "**Information Retrieval Models for Recovering Traceability Links between Code and Documentation**", IEEE Trans. On Software Engineering, Vol. 28, No. 10, pp. 970-983.

[BOH91]S.A. Bohner, "**Software change impact analysis for design evolution**", Proceedings of the International Conference on Software Maintenance and Reengineering, 8:292—301, 1991.

[CHA95]S. Chakravarthy, "**Early Active Database Efforts: A Capsule Summary**", IEEE Transactions on Knowledge and Data Engineering, Vol. 7, No. 6, Dec. 1995.

[CS94]R. Chandra and A. Segev, **"Active databases for financial applications",** Research Issues in Data Engineering, 1994. Active Database Systems, Proceedings Fourth International Workshop on, 14-15 Feb. 1994. Pages: 6 – 52.

[CCG02]J. Cleland-Huang, C.K. Chang and Y. Ge, "**Supporting Event Based Traceability through High-Level Recognition of Change Events**", IEEE Proc. Int'l Computer Software and Applications Conf. (COMPSAC), Aug. 2002.

[CCC03]J. Cleland-Huang, C.K. Chang and M. Christensen, "**Event-Based Traceability for Managing Evolutionary Change**", IEEE Transactions on Software Engineering, Vol. 29, No. 9, Sept. 2003.

[CCW03]J. Cleland-Huang, C.K. Chang and Jeffrey C. Wise, "**Automating performance-related impact analysis through event based traceability**", Requirements Eng., 2003.

[CCY02]J. Cleland-Huang, Carl K. Chang and Yujia Ge, "**Supporting Event Based Traceability through High_level Recognition of Change Events**", Proceedings of the 26th Annual International Computer Software and Applications Conference, 2002.

[CLE02]J. Cleland-Huang, "**Robust Requirements Traceability for Supporting Evolutionary and Speculative Change**", PhD dissertation, Univ. of Illinois at Chicago, Mar. 2002.

[CZL04]J. Cleland-Huang, G. Zemont and W. Lukasik, "**A Heterogeneous Solution for Improving the Return on Investment of Requirements Traceability**", Proceedings of the 12th IEEE International Requirements Engineering Conference, 2004.

[CN93]A. Cornelio and S.B. Navathe, "**Using active database techniques for real time engineering applications",** Data Engineering, 1993. Proceedings. Ninth International Conference on, 19-23 April 1993. Pages: 100 – 107.

[DAV90]A.M Davis, "**The analysis and specification of systems and software requirements**", Systems and Software Requirements Engineering, IEEE Computer Society Press, 1990, 119-144.

[FE00]A. Finkelstein and W. Emmerich, "**The future of Requirements Management Tools**", in Information Systems in Public Administration and Law, Austrian Computer Society, 2000.

[GSS04]D. Goldin, S. Srinivasa and V. Srikanti, "**Active databases as information systems",** Database Engineering and Applications Symposium, 2004. IDEAS '04. Proceedings. International, July 7-9, 2004. Pages: 123 – 130.

[GF94]O. Gotel and A. Finkelstein, "**An Analysis of the Requirements Traceability Problem**", Proc. Of the 1st Int'l Conf. On Requirements Engineering, 1994, pp. 94-101.

[GF95]O. Gotel and A. Finkelstein, "**Contribution Structures**", Requirements Engineering, 1995, Proceedings of the Second IEEE International Symposium on 27-29 March 1995 Page(s): 100 – 107.

[GOU94]Joseph A. Goguen, "**Requirements engineering as the reconciliation of social and technical issues**", Requirements Engineering, Social and Technical Issues, 1994. Page: 165-199.

[JAR98]M. Jarke, "**Requirements Traceability**", Comm. ACM, vol. 41, no. 12, pp. 32-36, Dec. 1998.

[JH02]Jane Huang, "**Robust Requirements Traceability for handling Evolutionary and Speculative Change**", Ph.D. Thesis, University of Illinois at Chicago, 2002.

[KS02]G. Kotonya and I. Sommerville, "**Requirements Engineering: Processes and Techniques**", John Wiley & Sons, 2002.

[LAM00]Axel van Lamsweerde, "**Requirements Engineering in the Year 00: A Research Perspective**", Proceedings of the 22nd International conference on Software Engineering, June 2000.

[NDS04]N. Nurmuliani, Didar Zowghi and Sue Fowell, "**Analysis of Requirements Volatility during Software Development Life Cycle**", Proceedings of the 2004

Australian Software Engineering Conference, 2004.

[NZW04]N. Nurmuliani, D. Zowghi and S.P. Williams, "**Using Card Sorting Technique to Classify Requirements Change**", Proceedings of the 12th IEEE International Requirements Engineering Conference, 2004.

[OWE98]K.T. Owens, "**Building Intelligent Databases With Oracle PL/SQL, Triggers, and Stored Procedures**", 2nd ed. Upper Saddle River, NJ: Prentice-Hall, 1998.

[PAL97]J.D. Palmer, "**Traceability**", Software Req. Eng., R.H. Thayer and M.Dorfman, eds., 1997.

[PD99]N.W. Paton and O. Diaz, "**Active Database Systems**", ACM Computing Surveys, Vol.31, No. 1, 1999, pp.63-103.

[PCVM99]F.A.M. Porto, S.R. Carvalho, M.J. Viannae Silva and R.N. Melo, "**Persistent object synchronization with active relational databases",** Technology of Object-Oriented Languages and Systems, 1999. TOOLS 30. Proceedings, 1-5 Aug. 1999. Pages: 53 – 62.

[PG96]F.A.C. Pinheiro and J.A. Goguen, "**An Object-Oriented Tool for Tracing Requirements**", IEEE Software, Vol. 13, No. 2, Mar. 1996, pp. 52-64.

[PN98]Sooyong Park, Jongho Nang, "**Requirements management in large software system development**", Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on , Volume: 3 , 11-14 Oct. 1998, Pages:2680 - 2685 vol.3.

[PVC98]F. Porto, M.J. Vianna e Silva and S. Carvalho, "**Object life-cycles in active relational Databases",** Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings, 3-7 Aug. 1998, Pages: 168 - 179.

[RAM98]Ramesh, B., "**Factors Influencing Requirements Traceability Practice**", Comm. Of the ACM, Dec. 1998, Vol 41, No. 12.

[RE04]Gerald Kotonya and Ian Sommerville, "**Requirements Engineering: Processes and Techniques**", John Wiley & Sons Ltd, May 2004.

[RE93]B. Ramesh and M. Edwards, "**Issues in the Development of a Requirements Traceability Model**", Requirements Engineering, 1993, Proceedings of IEEE Internaitonal Symposium on, 4-6 Jan. 1993, pp. 256-259.

[RJ01]B. Ramesh and M. Jarke, "**Toward Reference Models for Requirements Traceability**", IEEE Trans. Software Eng., vol. 27, no. 1, pp. 58-92, Jan. 2001.

[RPSE95]B. Ramesh, Powers, C. Stubbs and M. Edwards, "**Implementing Requirements Traceability: A Case Study**", Requirements Engineering 1995, Porceedings of the Second IEEE International Symposium on, 27-29 March 1995, pp.89-95.

[ROG05]Roger S. Pressman, "**Software Engineering: A Practitioner's Approach**", 6th Edition, published by McGraw-Hill.

[SC02]B. Sengupta and R. Cleaveland, "**Triggered Message Sequence Charts**", ACM SIGSOFT 2002, 10th International Symposium on the Foundations of Software Engineering.

[SC03]B. Sengupta and R. Cleaveland, "**Refinement-Based Requirements Modeling Using Triggered message sequence charts**", Proceedings of the 11th IEEE International Requirements Engineering Conference, 2003.

[SDS99]A. Sundermier, S.W. Dietrich and V. Shah, "**An active database approach to integrating black-box software components",** Computer Software and Applications Conference, 1999. COMPSAC '99. Proceedings. The Twenty-Third Annual International, 27-29 Oct. 1999. Pages: 403 – 409.

[SG95]The Standish Group, **Chaos Reports**, 1995, www.standishgroup.com/visitor/chaos.htm.

[SSV99]Pete Sawyer, Ian Sommerville and Stephen Viller, "**Capturing the Benefits of Requirements Engineering**", IEEE Software, March/April 1999.

[WD01]B. Wu and Kudakwashe Dube, "**Applying Event-Condition-Action Mechanism in Healthcare: a Computerised Clinical Test-Ordering Protocol System (TOPS)**", IEEE Computer Society, pp.3-14, 2001.

[WIE99]K. Wiegers, "**Automating Requirements Management**", Software Development, 7(7), July 1999.

[ZAV97] P. Zave, "**Classification of research efforts in requirements engineering**", ACM Computing Surveys, 29(4):315—321, 1997.

[ZRP04]M. Zoumboulakis, G. Roussos and A. Poulovassilis, "**Active Rules for Sensor Databases**", Proceedings of the First Workshop on Data Management for Sensor Networks, Toronto, Canada, Aug, 2004.

APPENDIX A

A traceability database was established by using Oracle SQL statements. Following are those statements:

```
CREATE TABLE REQUIREMENTS
   (req_id                varchar2(10)      CONSTRAINT pk_requirements
                                            PRIMARY KEY,
    statement             varchar2(80),
    date_entered          date,
    date_changed          date,
    rationale_id          varchar2(10),
    stakeholder_id        varchar2(10),
    design_id             varchar2(10),
    status                varchar2(12),
    dependants            varchar2(10),
    is_dependent_on       varchar2(10),
    lev                   varchar2(10),
    comments              varchar2(40));

CREATE TABLE DESIGNS
   (design_id             varchar2(10)      CONSTRAINT pk_designs
                                            PRIMARY KEY,
    statement             varchar2(80),
    type                  varchar2(40),
    date_entered          date,
    date_changed          date,
    stakeholder_id        varchar2(10),
    req_id                varchar2(10),
    component_id          varchar2(10),
    status                varchar2(12),
    dependants            varchar2(10),
    is_dependent_on       varchar2(10),
    lev                   varchar2(10),
    comments              varchar2(40));

CREATE TABLE REQ_LIST
    (req_id               varchar2(10),
     description          varchar2(40),
     next                 varchar2(10));

CREATE TABLE DESIGN_LIST
    (design_id            varchar2(10),
     description          varchar2(40),
     next                 varchar2(10));

CREATE TABLE COMP_LIST
    (component_id         varchar2(10),
     description          varchar2(40),
     next                 varchar2(10));

CREATE TABLE COMPONENTS
   (comp_id               varchar2(10)      CONSTRAINT pk_components
                                            PRIMARY KEY,
    statement             varchar2(80),
```

```
    date_entered        date,
    date_changed        date,
    testcase_id         varchar2(10),
    stakeholder_id      varchar2(10),
    design_id           varchar2(10),
    resources_id        varchar2(10),
    status              varchar2(12),
    dependants          varchar2(10),
    is_dependent_on     varchar2(10),
    lev                 varchar2(10),
    comments            varchar2(40));


CREATE TABLE TESTCASE
   (testcase_id         varchar2(10)     CONSTRAINT pk_testcase
                                         PRIMARY KEY,
    statement           varchar2(80),
    date_entered        date,
    date_changed        date,
    stakeholder_id      varchar2(10),
    comments            varchar2(40));

CREATE TABLE RESOURCES
   (resources_id        varchar2(10)     CONSTRAINT pk_resources
                                         PRIMARY KEY,
    statement           varchar2(80),
    date_entered        date,
    date_changed        date,
    stakeholder_id      varchar2(10),
    status              varchar2(12),
    comments            varchar2(40));

CREATE TABLE STAKEHOLDERS
   (stakeholders_id     varchar2(10)     CONSTRAINT pk_stakeholders
                                         PRIMARY KEY,
    name                varchar2(80),
    date_entered        date,
    date_changed        date,
    department          varchar2(30),
    position            varchar2(30),
    status              varchar2(12),
    comments            varchar2(40));

CREATE TABLE RATIONALE
   (rationale_id        varchar2(10)     CONSTRAINT pk_rationale
                                         PRIMARY KEY,
    statement           varchar2(80),
    date_entered        date,
    date_changed        date,
    stakeholder_id      varchar2(10),
    status              varchar2(12),
    comments            varchar2(40));
```

VITA AUCTORIS

Haipeng Ge was born in 1970 in Anshan, China. He graduated from Northeastern University where he obtained a B.S. in Computer Science in 1994. He is currently a candidate for the Master's Degree in Computer Science at the University of Windsor and hopes to graduate in summer 2005.