2003

Analyzing the impact of changing software requirements: a traceability-based methodology

James Steven O'Neal
Louisiana State University and Agricultural and Mechanical College

ANALYZING THE IMPACT OF CHANGING SOFTWARE REQUIREMENTS: A TRACEABILITY-BASED METHODOLOGY

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
James Steven O'Neal
B.S., Mississippi College, 1987
M.S., Clemson University, 1989
December 2003

Acknowledgments

The completion of this dissertation was only possible with the contributions of many people. Foremost is the generous support, guidance, and patience of my advisor, Dr. Carver, who brought forth this research and allowed me to extend my education beyond the formal studies leading to this dissertation. Thanks to Dr. Kraft for his mentoring, for his encouragement, and for recommending the papers that sparked a fuzzy way of thinking. I also thank Dr. Iyengar, Dr. Kak, and Dr. Lou for serving as members of my doctoral committee.

A special acknowledgment and thank you is given the members of the software engineering class who participated in the experimental application of my ideas. I am truly indebted to them for their extra work during that semester.

Finally, behind the scenes are my friends and family who were always there for me during my doctoral studies. I am beholden to you all for your fellowship, reassurance, and support. This is especially true of my niece and nephews, who simply allowed me to share in their play.

Table of Contents

List of Tables

List of Figures

Abstract

Software undergoes change at all stages of the software development process. Changing requirements represent risks to the success and completion of a project. It is critical for project management to determine the impact of requirement changes in order to control the change process. We present a requirements traceability based impact analysis methodology to predictively evaluate requirement changes for software development projects. Trace-based Impact Analysis Methodology (TIAM) is a methodology utilizing the trace information, along with attributes of the work products and traces, to define a requirement change impact metric for determining the severity of a requirement change. We define the Work product Requirements trace Model (WoRM) to represent the information required for the methodology, where WoRM consists of the models Work product Information Model (WIM) for the software product and Requirement change Information Model (RIM) for requirement changes. TIAM produces a set of classes of requirement changes ordered from low to high impact. Requirement changes are placed into classes according their similarity. The similarity between requirement changes is based on a fuzzy compatibility relation between their respective requirement change impact metrics. TIAM also identifies potentially impacted work products by generating a set of potentially impacted work products for each requirement change. The experimental results show a favorable comparison between classes of requirement changes based on actual impact and the classes based on predicted impact.

Chapter 1
Introduction

Software undergoes change at all stages of the software life cycle. That is, changes to requirements may occur at the requirements definition phase, requirements specification phase, design phase, implementation phase, and maintenance phase. Managing changes to a software product is frequently critical to the success of the product [Glass, 1998]. Accepting too many changes will cause delays in the completion of product, whereas failure to implement critical changes can affect the success of the product. According to Brooks, "Clearly a threshold has to be established, and it must get higher and higher as development proceeds, or no product ever appears," [Brooks, 1995]. Additionally, the expense of implementing changes in a software product becomes greater with each subsequent phase in the software life cycle [Boehm and Papaccio, 1988]. To effectively manage change in software development projects, methods are required to provide information about changes, such as how will the change impact a development schedule or what changes will have the greatest impact on a product. With information about changes, appropriate planning can be performed by project management for implementing or deferring changes.

Software development frequently starts with a customer that has a need that a software product may satisfy. Initial discussions with the customer and members of a software development group will usually yield a requirements definition document. The requirements definition document records the requirements of the software system in a manner that is understandable to the customer and to the designers of the development group. Each requirement in the definition should be

identified in such a manner that the requirement can be referenced by subsequent development work or by future changes in the requirements. The requirements definition document details the needs of the customer and related details of the customer's processes that the software system is to satisfy. This document serves as the mechanism for discussing the goals of the software system with the customer and may function as part of a contract between the customer and the software development group [Andriole, 1998].

The next step of development is the requirements specification. While the requirements definition document defines the requirements usually in natural language, the requirements specification document defines the requirements using more technical language the developer can use. The requirements specification should be unambiguous, therefore formal languages and other formalized techniques are often used in the requirements specification. Formal languages and techniques provide a strict or more exact model of the requirements definition to eliminate any ambiguity. Since formal languages may not be understood by the client, each individual requirement specification should be traceable to a corresponding requirement definition to facilitate future discussions of the requirements.

After the requirements specification is completed, the design of the software system begins. For most large software projects, a high level design of the system, called the architectural design, is initially performed. The architectural design is the first step in deciding how a product will be structured into modules to implement the requirements. Detailed design further defines the internal structures of modules and the interactions between modules. The detailed design of the product defines how the product is to work.

The design is then implemented, or coded, in a programming environment. The source code is organized into software modules according to the software architec-

ture. Test cases are then written to verify the basic operations of the modules. The software modules are the source code of the software product. Individual testing of each module is called unit testing.

Software modules are integrated to form the complete software product. Verification must be done at this point to ensure the modules interact as designed. This testing is called integration testing. Testing is also performed to verify that the product modules are assembled correctly.

When the product has passed integration testing, the functionality of the product must be tested and demonstrated. Function testing uses test cases that validate that the requirements have been satisfied in the complete product. Successful function testing marks production of a validated software product, ready for customer acceptance testing.

After the requirements definition, requirements specification, architectural design, detailed design, implementation, and integration phases, the product is validated to meet the needs of the customer. The product then moves into the operational phase where the product is used by the customer.

Most products will undergo changes after they become operational. Frequently, these changes will be incorporated into a new development cycle that uses the existing product as the base. These subsequent development cycles create new versions of the product that supersede the previous version. Changes for a new release can either correct or enhance the product. Frequently, it is not possible to include all needed changes and still release the next version in a timely manner. The decision as to what changes are incorporated into a release is a trade-off between development time and changes critical to the success of the product. Some changes may be saved for a future release of the product in order to deliver a version of the product that includes more important changes in a timely manner.

The requirements definition, specification, design, implementation, and operational phases are the general software development phases. The exact manner in which these phases are executed depends on the development model.

The most straight forward development model is the waterfall model [Royce, 1970]. Each phase of development is completed before the next phase is started, as shown in Figure 1.1. In this model the entire product is delivered at the end of development to the customer. The benefit of the waterfall model is the delineation of development activity into the separate phases, each with clearly defined outputs. However, the model is fairly rigid, requiring the activity of each phase to be completed before moving to the next phase.

The incremental life cycle model delivers successive builds of the product to the customer, until the entire product is completed [Schach, 1999]. Once the requirements definition, requirements specification, and architectural design are completed, multiple iterations of detailed design and implementation occur, where each iteration's build adds a subset of the functionality of the product until the complete functionality is delivered. This model provides the customer with a view of the product before complete functionality is delivered. Developing the product by this model requires the product to be designed in a manner that each build easily includes the newly completed functionality. This model often results in a product with an open architecture that facilitates the addition and modification of functionality. Figure 1.2 shows the incremental model.

To address the risks associated with major software products, the spiral life cycle model incorporates strict verification and risk analysis tasks between each phase [Boehm, 1988]. The risk analysis is performed to determine the objectives, alternatives, and constraints at each phase. Then, risks are identified, alternatives are evaluated, and risk resolution is attempted. If the risks cannot be resolved,

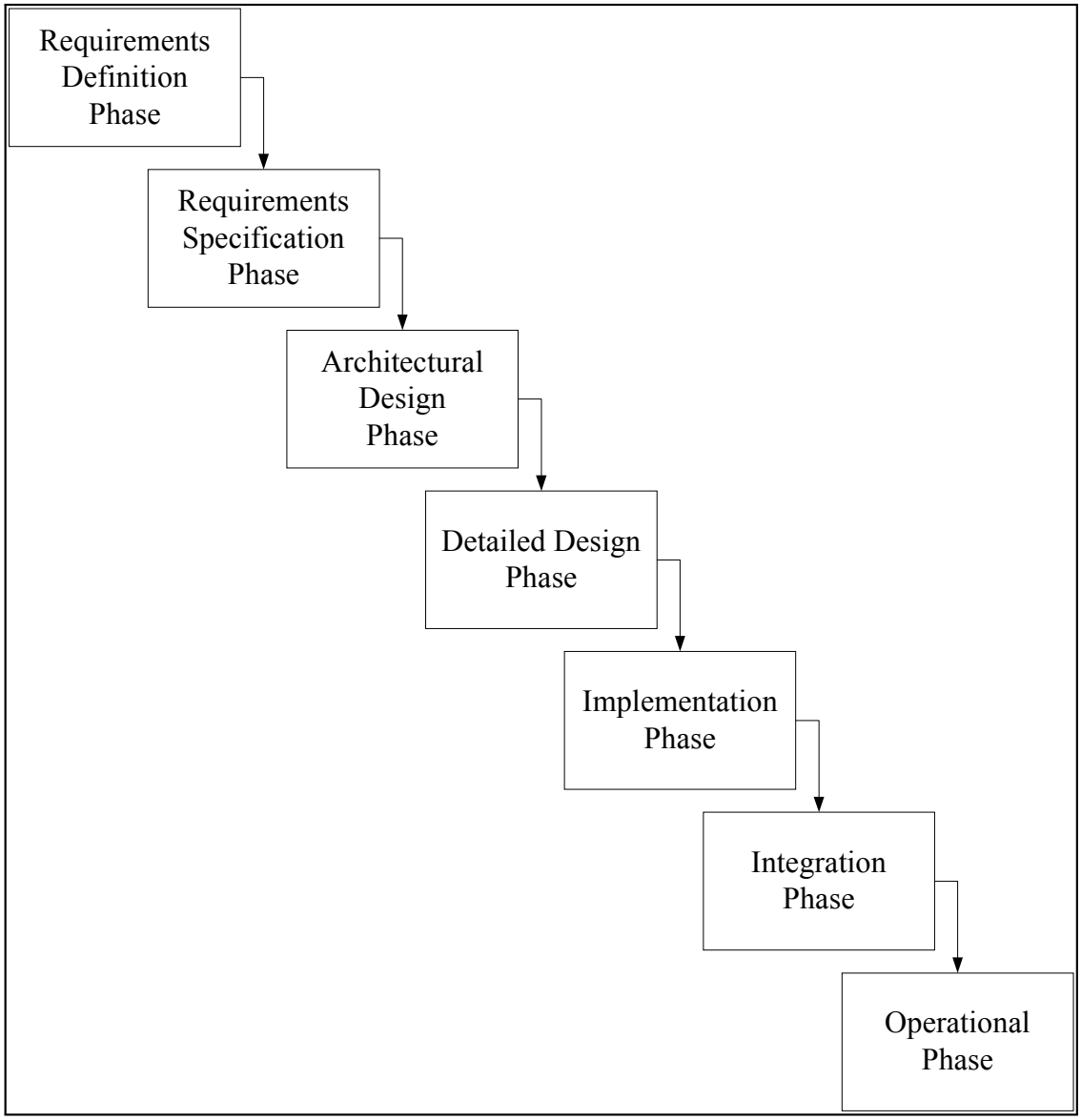

FIGURE 1.1. Waterfall life cycle model

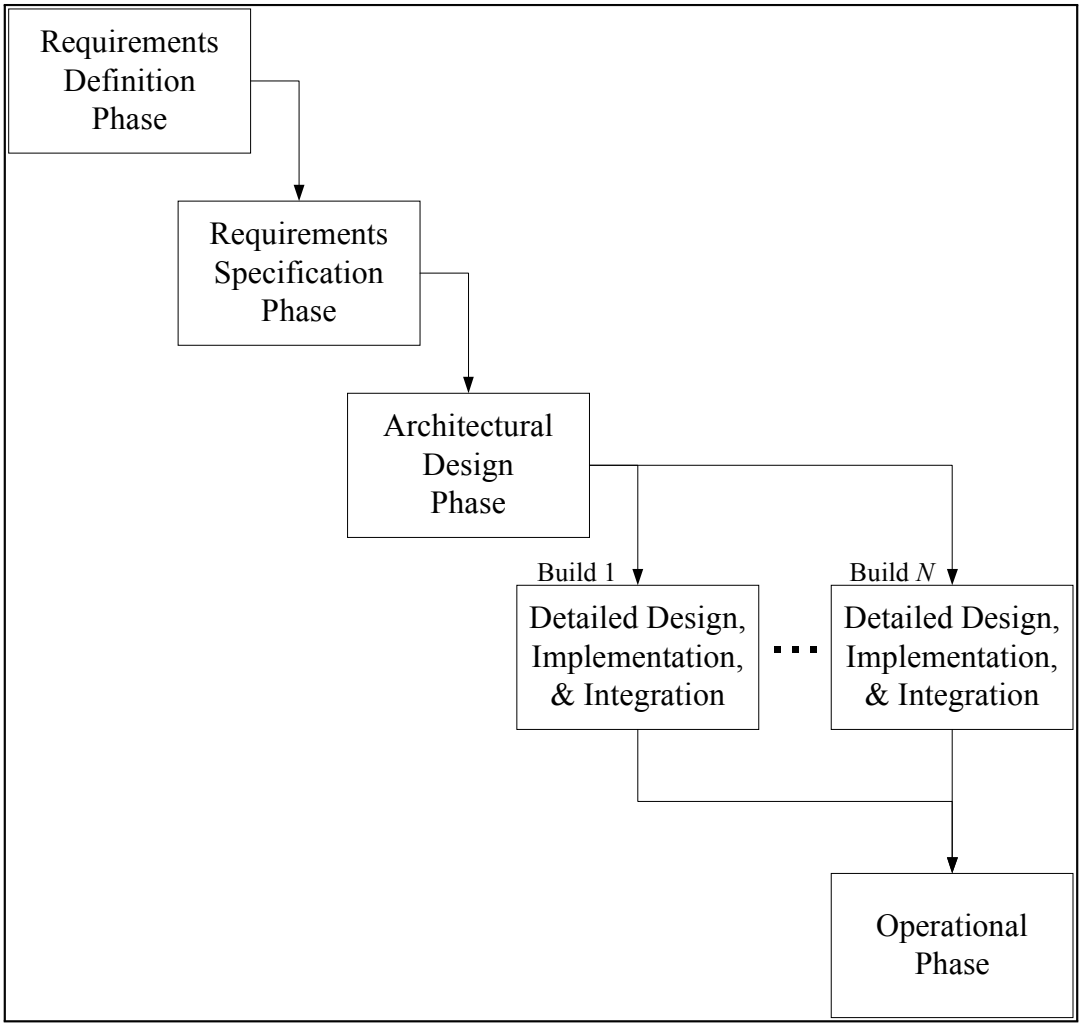

FIGURE 1.2. Incremental life cycle model

the project can be ended at that point, saving additional expense on the project. Failure to adequately manage risks associated with software development is one reason for large cost overruns in some software projects [Boehm, 1988].

The development of object oriented software is highly iterative, with the products in each phase are refined on each iteration. This practice has been described in the fountain life cycle model [Henderson-Sellers and Edwards, 1990]. The phases of the software life cycle overlap. Some activities performed in the phases are performed in parallel. Often discoveries in a latter phase require the revisiting of activities performed in an earlier phase.

In practice, variations of these models are used to fit the needs and the experience of the development group and customer. Two emerging models, extreme programming [Beck, 1999] and synchronize and stabilize [Cusumana and Selby, 1997], are examples of models created for specific types of projects and corporate cultures. In the extreme programming model, the development team selects a small set of requirements that are related to a feature, or story, to implement. The product is rapidly built to support this story and released to the customer. This process repeats until all the features required by the customer are implemented. Adaptability to vague and changing requirements is a key feature of the extreme programming model.

The synchronize and stabilize model, used by development organizations within Microsoft, is also similar to the incremental model. In this model, as soon as a feature is implemented, the code is integrated into the product during the daily build. Testing and debugging is performed after each daily build. Periodically the product is stabilized by halting the addition of features and working to remove the remaining flaws in the product. The synchronize and stabilize model allows

concurrent testing and program development along with the ability to incorporate customer feedback during development.

Regardless of the type of model used, the completed software product includes product documentation, source code, test suites, as well as the executable product. The product documentation is a collection of the documentation used in development of the product, from requirements through design to source code documentation. This information is important for future changes to the product so that appropriate analysis and caution is used when making a change to the product [Basili, 1990]. A developer making a change to a product that is being used by a customer should understand the design and implementation of the product. There are several reasons to make sure a developer understands the architecture and design of the product in sufficient detail. One is to ensure that the change is completely implemented within the product. Another is to prevent new faults from being introduced in the product as a result of changes in the product. For example, when an interface is changed between two modules, if other modules that use the called module are not changed to reflect the new interface, new faults would be introduced into the product. One other reason for the importance of attention to the product documentation is to make sure that the architecture of the product is not compromised and future maintenance made more difficult. An example could be a developer adding code to handle a unforeseen exception condition and not utilizing a sophisticated exception processing facility designed into the product that should have been invoked.

1.1 Managing Software Changes

The ability to manage change during software product development is critical for completing the product and satisfying the needs of the client requesting the prod-

uct. According to Brooks, changeability is one of the essential difficulties of software production [Brooks, 1987]. Software is frequently the most malleable component of a system, and therefore it is most likely to be changed. If changes to a software development project are not restrained, expectations for meeting estimates are not realistic. However, changes must be allowed so that the product satisfies evolving needs of the client [Kotonya and Somerville, 1998].

In the early stages of development during requirements definition, a set of requirements is created that identifies the needs of the client. During product development, changes to this set of requirements may be needed. These requirement changes are modifications to existing requirements or new requirements that may or may not affect existing requirements. It is naive to expect that no changes would be made after requirement specification. IBM's Santa Teresa Laboratory reported that an average of 25% of the requirements for an average project will change before completion of the project [Boehm, 1981]. A number of reasons exist for requirements changes, including: changes in customer needs or wants, clarification of requirements, changes in the target operating environment, and correcting errors in requirements [Bohner, 2002].

Requirements changes must be considered so that the product will satisfy the needs of the client; however, not all changes are equal [Bach, 1999]. Some changes may be critical to the success of the product, whereas some changes may be optional and thus should be included in a particular version of the product only if time allows. The ability to determine the risks that the change has on the potential to complete the product within established schedules represents a key aspect of project management [Kotonya and Somerville, 1998].

It is vitally important that project managers have information available to enable appropriate decisions to be made with respect to changes introduced during

product development. In general, management has three choices when present a change. First, the change can be incorporated in the current product development. This may require more resources to be allocated to the project and cause delays in delivering the product. Second, the change can be deferred from current project, to be included in the product at a future date. Lastly, the change can be rejected.

In [Jones, 1995], creeping requirements, which includes changes to current requirements, are identified as one of the top ten factors associated with the success or failure of a software project. Projects that fail to control changes to requirements and quantify the impact of changing requirements imbibe a risk to the successful completion of the project. Since requirement changes can be costly, frequently a formal business case must be made to justify the requirement change in a software project [Maciaszek, 2001].

Impact analysis is used in many different forms to manage changes to software. Impact analysis, as defined by Arnold and Bohner, "is the activity of identifying what to modify to accomplish a change, or of identifying the potential consequences of a change," [Arnold and Bohner, 1993]. There are several areas of research that are investigating how to measure the impact of change. Some techniques are source code based. The research using this approach focuses on the dependencies that exist in the source code in order to determine what program elements may be affected by a change. Various types of program dependency techniques have been proposed, including control and data dependency [Podgurski and Clarke, 1990; Loyall and Mathesen, 1993] and program slicing [Horwitz *et al.*, 1990; Chen *et al.*, 1996]. A second research direction uses traceability relationships between all work products or documents created during development. Bianchi, et.al., use several types of traces to predict impacted components of a software system [Bianchi *et al.*, 2000]. Other traceability approaches view work products as documents, incorporating traceabil-

ity as hypertext links or as interfaces between documents [Garg and Scacchi, 1990; Horowitz and Williamson, 1986]. These approaches attempt to identify what may be modified to make a change.

One problem resulting from incorporating changes in a software product after the requirements definition is that the change may only be reflected in work that has yet to be completed. When this occurs, requirements, design documents, and other product documentation become incorrect and outdated. Product documentation that does not reflect the actual product makes product maintenance difficult as the product source code must be used as the arbitrator of what the product does. Unfortunately in practice, developers rarely update existing documentation due to limited resources. As noted by Lindvall and Sandahl, experienced developers usually assume that the documentation is outdated and rely mainly on source code when making changes[Lindvall and Sandahl, 1998a]. The appropriate implementation of changes includes the updating of the product specification and design to reflect changes to the source code of the product [Maciaszek, 2001]. It is important to include the additional work that keeps the product documentation current as part of impact analysis.

1.2 Change Scenarios

To emphasize the need to manage changes, we present the following scenarios. The first scenario involves management's need to determine the impact of planning for a more flexible information technology infrastructure, specifically allowing a choice of database management systems. The second scenario describes a policy change that must be incorporated into the supporting software application.

The first scenario involves a company that is developing a new inventory control system. The company's current information systems are based on a single vendor's

database management system (DBMS). Since the current software development organization is experienced with using the existing DBMS, it is a requirement for the new inventory control system to use the proprietary database interface to the existing DBMS. Management has decided to investigate minimizing the risk of depending on the vendor of the existing DBMS. A standard is evolving to uncouple database applications from specific DBMS, called Open Database Connectivity (ODBC). Since the existing DBMS has an ODBC interface, management wishes to know if the new inventory control system under development can be changed to use the ODBC interface rather than the proprietary database interface. Implementing this change would allow the company to choose a new DBMS in the future without significant changes to the inventory control system.

Determining the impact of the ODBC migration scenario depends on how far the development of the inventory control system has progressed and the architecture of the inventory control system. If the development of the system has not reached the design phase, the impact of the change on the product should be minimal, since there is nothing to change except for requirements documents. However, if the system has been designed and possibly source code modules completed, then any design documents and source code modules including a database interface would be impacted and require modification. If the architecture of the inventory control system isolates the DBMS access from the business application functions, the impact would be less than if many modules implementing application functions independently access the DBMS.

Impact analysis of this scenario should be able to give information on what may be impacted and by to what degree, if the change was to be made, without regard to what point development has progressed on the product or dependent on the product architecture. Management would use this information to decide if it was

cost effective to migrate to ODBC at this time or defer the migration to a later date. Management may determine that the costs are to high to migrate at this point, or they may find that the costs of the migration would be offset by the savings incurred by using another vendor's DBMS.

The second scenario involves a university. To enroll in classes that have prerequisite courses, a student must have passed the prerequisite courses with the required grade or the student is automatically withdrawn from the course at the start of the term. A student is allowed to retake a prerequisite course only twice. The university's enrollment system implements this restriction. A new policy has been made that removes the limit to the number of times a student may retake a course, but the student must have approval from the dean of the college in which the course is offered when enrolling in a course after the second time.

In this scenario, the change must be implemented. Management of the university enrollment system requires information from impact analysis to determine the extent of the impact in order to schedule resources for implementing the policy change. Additionally, impact analysis should be performed after changes have been implemented to ensure that no other restrictions in the enrollment system are affected.

1.3 Research Objective

The research objective is to develop a predictive impact analysis technique that identifies classes of requirements changes that have similar impact levels. By predicting the impact that requirement changes may have, the effects of making a requirement change can be compared to other requirement changes with respect to the predicted effort to implement the change. This information can be used

as a criterion in a process that selects which changes can be implemented within schedule constraints.

We develop formal models and a general methodology for applying requirement traceability to impact analysis. Next, we develop a specific instance of the general methodology to use for impact analysis. Finally, the specific instance of the methodology is applied to a software development project to evaluate capability of the methodology for impact analysis.

The general methodology for using requirement traces for impact analysis includes the definition of formal models that represent a software development project and a set of requirement changes. The models include information on attributes of the work products and traces. The attributes are complexity, effort, phase, and influence. The methodology defines an impact metric that is directly related to the effort predicted to implement a requirement change. With these results the requirement changes are grouped into classes of changes that have similar impact.

The specific instance of the impact analysis general methodology defines the attribute levels that are used to describe the work products and traces. The instance is applied to a software development project in which the actual impact with respect to effort is documented. The actual impact information is used to classify changes into groups with similar impact. These results are compared with the results from our impact analysis methodology.

1.4 Motivation

This research is motivated by the continuing need to increase the efficiency of software evolution. Previous work has focused on impact analysis for maintenance changes to completed software product using traceability. The use of traces between work products is an appropriate basis for impact analysis for software prod-

ucts [Strens and Sugden, 1996]. Research providing impact analysis for software products during development is needed [Zave, 1997]. The traceability approach does not depend on the internal structure of the work product, therefore is suitable to apply to a software project that is in progress, where work products may not be developed to the degree that would allow the internal structure to be used for any analysis. Also, previous work has focused on the work products that may be impacted but not on the effect of the change on resources.

Further, to improve software evolution we need to be able to evaluate a set of proposed changes to software product in order to help assess the degree of impact a change may have on the software project with respect to the other proposed changes [Nuseibeh and Easterbrook, 2000]. This information would allow management of the project to initially determine potential impacts on resources and what changes may require significant changes in resources to implement.

1.5 Summary

Requirement changes during the development of a software product threaten the timely completion of the product and the proper documentation of the product. The decision to incorporate changes during development should allow for the modification of work products that have been completed so that all documentation is correct and so that those who perform future maintenance will not be dependent solely on the source code to determine what the product does.

This research determines the impact that a change has on existing work for a software product. It derives a comparative relationship of severity of impact between changes, and evaluates changes that are modifications to existing requirements or new requirements that affect existing requirements. The evaluation is a prediction of the amount of effort required to modify existing work to make the

changes. We categorize the changes into groups of varying impact, and we order the groups based on low to high impact.

Chapter 2
Background and Related Research

2.1 Introduction

The field of software engineering is concerned with delivering quality software, within budget and schedule constraints, that satisfies the needs of the client and user. Since changeability is a essential aspect of software [Brooks, 1987], much of the research has focused on change. Techniques used for requirements engineering seek to manage change by documenting sources of information, and providing formal methods to introduce changes. Methods of impact analysis have been used during the maintenance phase of software life cycle to determine the effect of a change or predicting the effect of a potential change. This research is concerned with distinguishing the severity between changes. In Section 2.2 we define terms used in this research. Sections 2.3 - 2.7 review related research in requirements engineering, traceability, impact analysis, graph theory, and fuzzy set theory, respectively.

2.2 Terminology

In this research, we use the following terms:

Impact analysis is the task of identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change [Arnold and Bohner, 1993].

A **requirement** is a statement of a system service or constraint [Maciaszek, 2001].

A **requirement change** is either a modification to an existing requirement or a new requirement that may or may not affect existing requirements. Requirement changes occur after the finalization of the requirements specification.

Requirements traceability is the ability to describe and follow the life of a requirement, in both a forward and backwards direction (i.e., from its origins, through its development and specification, to subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases) [Gotel and Finkelstein, 1994].

A **work product** is a concrete, planned result of the development process [IBM Object-Oriented Technology Center, 1997]. In this research, work products are software artifacts that must be maintained or recreated when the details on which the artifact was based change [Gotel and Finkelstein, 1997]. Examples include a use case, a class model, an object collaboration diagram, other UML diagrams, a source code module, and a test case.

2.3 Requirements Engineering

Requirements engineering is one branch of software engineering that has emerged to facilitate the development of software that truly meets the needs of the client [Zave, 1997]. Requirements elicitation and specification, identification of the stakeholders of a software product, and verification that the needs of the client have been meet are key issues in requirements engineering [Nuseibeh and Easterbrook, 2000]. As requirements evolve during the course of software development, the management of risk to the product from changing requirements is an important function of requirements engineering [Keil *et al.*, 1998].

One problem identified by Zave is, "Many requirements are not absolute; they can be satisfied partially, or only if resources permit," [Zave, 1997]. Some requirements can be categorized as imprecise. An imprecise requirement can be satisfied to a range of satisfaction degrees. Design decisions between design alternatives made during product development determine the degree that a requirement can

be satisfied. Yen and Tiao present a framework that allows design alternatives to be evaluated by determining relationships between requirements and design alternatives [Yen *et al.*, 1996], [Yen and Tiao, 1997]. Additionally, the rationale for the selection of specific design alternatives can be captured. When there are a number of imprecise requirements and multiple design alternatives that can be chosen, a design alternative may increase the satisfaction of one or more requirements at the cost of decreasing the satisfaction of one or more other requirements. Yen and Tiao's approach utilizes fuzzy logic to find the set of design alternatives that maximizes the satisfaction of requirements for the complete product.

Frequently there are requirements that should be implemented only if the inclusion of such requirements does not adversely impact the delivery of the product. In this case, project management should employ a method for prioritizing requirements [Siddiqi and Shekaran, 1996; Karlsson and Ryan, 1997] or negotiating requirements [Olphert *et al.*, 1994]. Research into the prioritization of requirements focuses on maximizing value or satisfaction and minimizing cost. For many projects, trade-offs must be made in selecting requirements so that the product may be completed within given time constraints. The value of a requirement is determined by the client and is relative to the other requirements. Karlsson and Ryan present an approach that ranks requirements at three levels, from one to three, where one denotes the highest level [Karlsson and Ryan, 1997]. A requirement's cost is determined relative to other requirements, with the sum of all requirements equal to 1. This work prioritizes requirements according to the cost-value ratio. The cost-value ratio facilitates the identification of requirements that have a high cost and a low value to the product. Failure to deliver requirements in a high cost, low value group can decrease development cost without jeopardizing client satisfaction. Jung [Jung, 1998] presents a knapsack approach based on the cost

and value determination of Karlsson and Ryan's work. This approach finds a set a requirements that maximizes value while minimizing cost.

The management of changing requirements must be actively performed during a software development project to ensure the a product is delivered and that the product is usable by the customer [Sugden and Strens, 1996]. First steps of risks management are the identification of risks and the evaluation of the impact of risks on the project [Williams *et al.*, 1997]. Properly documenting a requirement change as a risk is appropriate and should be part of the software development process. Then impact analysis can be performed to understand the degree of risk that a requirement change poses. With this information, management is able to perform a risk assessment and plan a course of action [Lam and Shankararaman, 1999]. Risks from requirement changes can be identified by analyzing requirements and determining which requirements are particular sensitive to change [Strens and Sugden, 1996]. These requirements are called volatile requirements and represent potential risk to a project. Appropriate risk mitigation includes accounting for the volatility of these requirements in the product design to minimize the risk to the project. The WinWin system, [Boehm and In, 1996], can be used for the management of requirement changes by following a scheme that evaluates the value of the change to the stakeholders and the risks to the project and seeks to resolve risks to the satisfaction of all stakeholders.

Requirements engineering is still an evolving discipline [Zave, 1997]. Resources for requirement engineering are increasing. Tools and processes are being developed and explored [Juristo *et al.*, 2002]. However, these tools cannot be used in isolation from the overall software development process [Nuseibeh and Easterbrook, 2000].

2.4 Impact Analysis

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" [Bohner, 1991]. Risk assessment is also associated with the task of impact analysis [Pleeger and Bohner, 1990]. The motivation behind impact analysis is to identify work products that may be affected by a change. With information on the objects that may be affected, plans can be made to determine what actions must be undertaken with respect to the change. Any change deemed necessary to implement will involve some rework of existing work products or tasks.

There are two main approaches to impact analysis, dependency analysis and traceability analysis. Dependency analysis is the analysis of relationships between program source code statements. Traceability analysis uses the relationships between work products for analysis, including design, program source code, and documentation. Dependency relationships can be found by automatic evaluation of source code, whereas traceability relationships may have to be explicitly expressed between work products [Spanoudakis, 2002].

Dependency analysis determines impact by exploring the internal structure of software code modules to identify relationships between the modules [Bohner and Arnold, 1996]. This approach focuses on the dependencies that exist in the source code in order to determine what program elements may be affected by a change. Various types of program dependency techniques have been used, including control and data dependency, [Podgurski and Clarke, 1990; Loyall and Mathesen, 1993] and program slicing, [Horwitz *et al.*, 1990; Chen *et al.*, 1996].

Control dependency uses a program's conditional structures for analysis and data dependency uses a program's variable usage for analysis [Podgurski and Clarke, 1990]. A control dependency exist if a statement contains a conditional

that controls the execution of several alternative program execution paths. Data dependency is established when a statement modifies a variable that is used in a subsequent statement [Loyall and Mathesen, 1993]. When a program statement is modified, statements which have either dependency on the modified statement may be impacted by the modification.

Program slicing limits the scope that is considered for impact analysis by considering only the sections of the original program which can be affected by a change. A slicing criteria, $S(v, n)$, defines a code slice on variable v at statement n in a program producing the sections of the program that affected the value of v before the execution of statement n, [Weiser, 1984]. The slices are determined by control and data information from the program. By excluding code that is not affected by the change, analysis and testing of the change is simplified [Gallagher and Lyle, 1991].

Traceability analysis uses relationships between types of work products for analysis, from requirements through design and code modules to testing and documentation. Dependency analysis allows a more detailed analysis than traceability analysis due to the explicit nature of source code, but is limited to the analysis of code modules. Typically the lack of detailed information between work products limits the effectiveness of traceability analysis. However, if traces are available a more extensive view of an impact on the software project as a whole can be determined.

2.5 Traceability

Traces can document the life cycle of a requirement, from the original client needs that created a requirement to the work products that lead to a software product that satisfies the requirement. A trace establishes a link from one work product

to another. Initially, requirements traceability was established as a mechanism for ensuring that the objectives, that is the requirements, of a product can be shown to have been satisfied. Today, traceability is used to additionally record the processes, stakeholders, and products involved in the work product's production. Current uses of requirement traces include quality assurance, reuse of components, testing, maintenance, change impact analysis, and process control [Pohl, 1996].

By the requirements specification document, requirements traceability is separated into two areas, pre-traceability and post-traceability. Pre-traceability links information regarding the sources of a requirement before the requirement is included in the requirements specification. Traces joining aspects of a requirement after placement into the requirements specification are called post-traceability links [Gotel and Finkelstein, 1994].

Forward to requirements and backwards from requirements links are pre-traceability links. Pre-traceability provides a method to document the source of requirements, specifically the business needs and political contexts in which they were created [Jarke, 1998]. Figure 2.1 illustrates pre-traceability links.

Post-traceability provides forward from requirements and backward to requirements links. Forward from requirements links allow the satisfaction of a requirement to be exhibited. Backward to requirements links facilitate finding the sources of a work product, providing the ability to find why and for what requirement a work product exists. These links provide information needed to ensure that requirements have been implemented in the product and that all features in the product are verified to satisfy the requirements [Jarke, 1998]. Figure 2.2 diagrams post-traceability links.

Much of the work on impact analysis using traceability has been limited to source code analysis or has been integrated into software development environ-

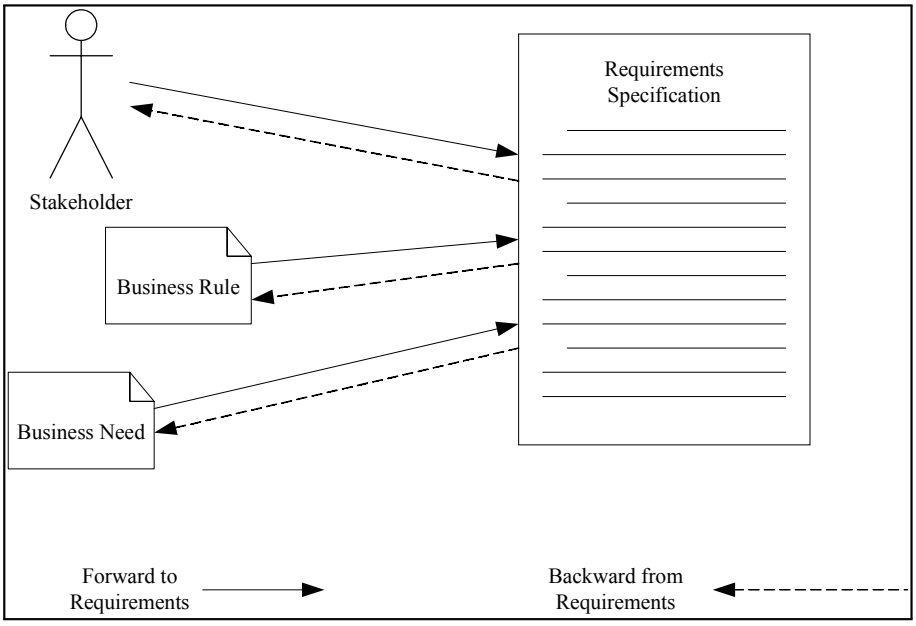

FIGURE 2.1. Pre-traceability links

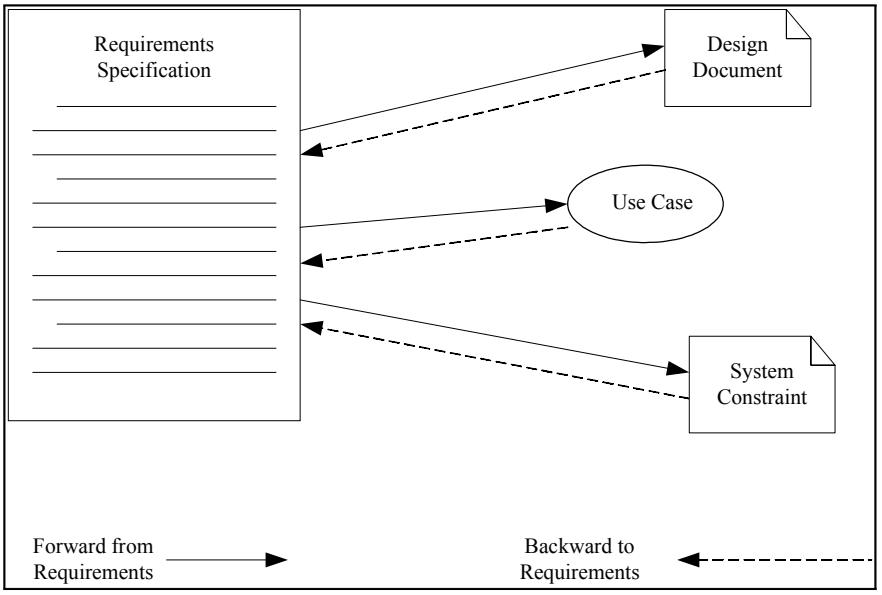

FIGURE 2.2. Post-traceability links

ments, [Bohner and Arnold, 1996; Queille and Voidrot, 1994]. Source code analysis is usually done after a change is made [Yau, 1988].

Code evaluated after a change has been implemented does not account for the overall impact to a software project. Work products such as design documents and test cases should be kept up-to-date and should be considered as part of the impact analysis.

Pfleeger and Bohner address the consequences of making a change by identifying complexity metrics that use traces to analyze the consequences of making a change. Traceability analysis was used to evaluate the impact of performing software maintenance. This work, which uses forward from requirements traces, defines two types of traces. Horizontal traces are traces that link work products in subsequent development phases, and vertical traces link work products within the same development phase. Complexity metrics based on these horizontal traces and vertical traces were defined. The objective was to ensure that only changes that did not increase the complexity metrics would be incorporated, with the goal that the product's design would evolve to minimize the impact of changes, [Pleeger and Bohner, 1990; Bohner, 1991].

A few software development environments have included some ability for traceability analysis in varying degrees, including ALICIA, SODOS, PMBD, and System Factory [Arnold and Bohner, 1993]. ALICIA stores the existence of software work products, but does not the store the content of work products. Navigation, via trace relationships is provided. SODOS enables the traceability and navigation of software documentation by the establishment of hyper-link between documents. PMDM is an integrated software engineering database which supports traceability but has no direct support of impact analysis. System Factory is similar to SODOS in that it supports a hypermedia view of software documentation. None

of these environments enable vertical traceability, limiting the ability of the systems to even identify all potentially impacted objects. Additionally the granularity of these systems varies from the document level (e.g. the requirement specification) to the statement level of a code module. Current traceability tools, including DOORS and RTM, allow vertical traceability and horizontal traceability between work products. These newer tools allow the entities, relationships, and attributes of a development project's artifacts be defined for traceability analysis. Therefore, any work product can be represented as a traceable entity in these tools, but the work products may actually be stored external to the tool.

Table 2.1 lists existing impact analysis methods. Most trace-based impact analysis methods predict the work products that may be modified because of a change to the product. This results in the estimated impact set [Arnold and Bohner, 1993]. Implicitly, those work products not in the estimated impact set, are not predicted to be modified. After a change is made to a product, the set of actually modified work products and the set of actually unmodified work products can be determined. The predicted outcome can be compared to the actual, resulting in four sets, shown in Figure 2.3, [Lindvall and Sandahl, 1998b]:

- Modified work products that are predicted to be modified.
- Unmodified work products that are not predicted to be modified
- Modified work products that are not predicted to be modified.
- Unmodified work products that are predicted to be modified.

The first two sets indicate when the impact analysis was correct. The latter two sets represent when the impact analysis prediction was incorrect. In [Lindvall and Sandahl, 1998b], using prediction of impacted work products by experienced developers, two common results were found. The predicted to be modified set was

too small, that is not enough work products were predicted to be modified. The other result was that some work products were predicted incorrectly.

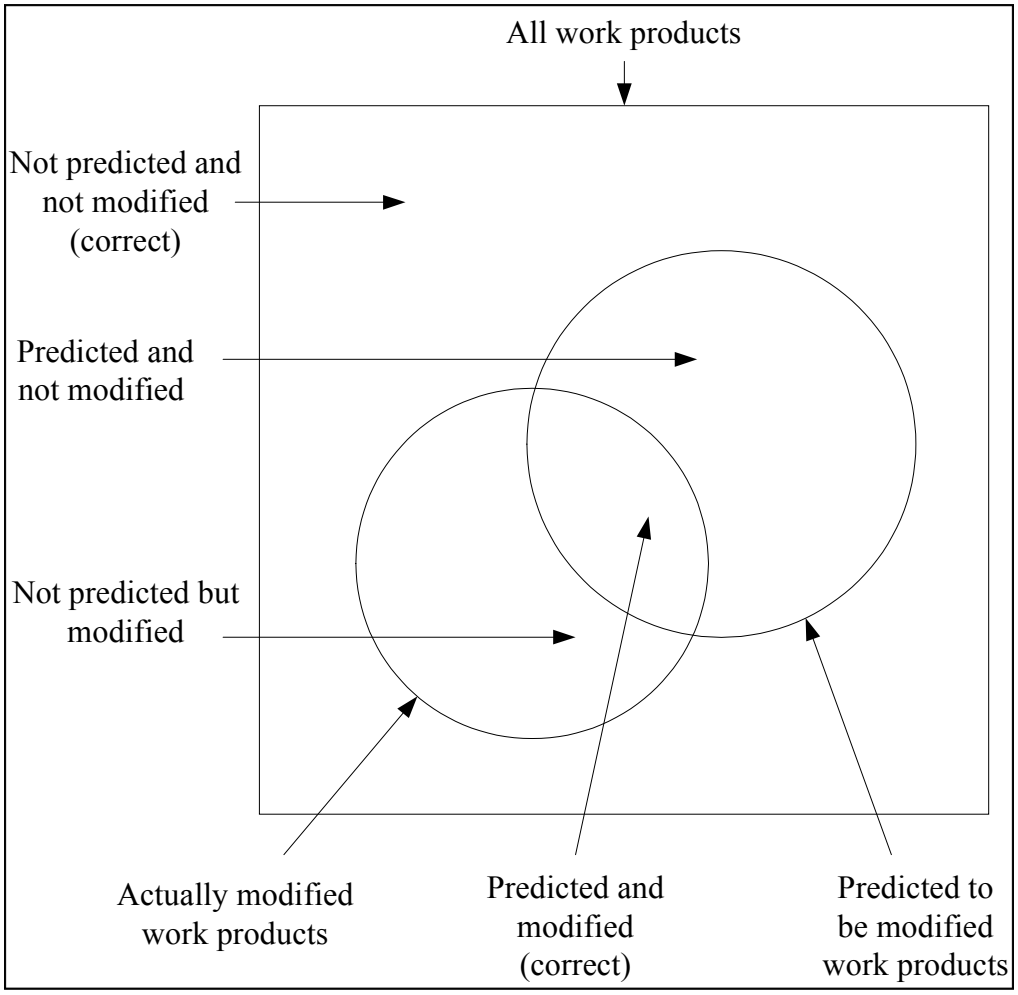

FIGURE 2.3. Diagram showing relationship between predicted impact set and actual impact set

Barros,et al., developed an Impact Analysis System (IAS) as a tool for impact analysis based on representing a software system as a system of typed objects, work products, and the dependency links between work products [Barros *et al.*, 1995]. The dependency links include composition links, life-cycle traceability links, version links, and documentation links. Composition links are discovered when a work product is defined or encompassed by another work product and version links

are created when the source code diverges due to multiple releases of the product. The work products and associated links form a dependency graph which is used for impact analysis. IAS allows a detailed description be made of desired modifications. The potential impact is determined by tracing potentially impacted work products via the dependency graph and propagation rules. There is a high level of interaction with the user of IAS to validate or invalidate the resulting impact set following a propagation. IAS is proposed as a tool to assist a maintenance developer with determining potentially impacted work products rather than performing actual modifications to the software system or providing any cost or schedule impacts.

Lindvall and Sandahl investigated using traceability for impact analysis for planning activities of new releases of a software product, [Lindvall and Sandahl, 1998b; Lindvall and Sandahl, 1998a]. They evaluated the availability of the following four types of traces for impact analysis:

1. Traces via explicit links, explicit traces, usually within the development environment, provided between work products in successive stages of development.
2. Traces by using references, which are textual references within a work product to another work product.
3. Name tracing, which can be used when a naming scheme allows searching for related work products.
4. System knowledge and domain knowledge provided by experienced software developers applying implicit knowledge of the software product.

They found that system knowledge and domain knowledge are the primary sources used by developers to predict which classes are impacted by changes. Their work uses only class source code modules as the set of work products that may be impacted by a change. Typically the number of actually impacted classes was

under-predicted by the impact analysis by developers. However, for each change, 60% to 100% of the predicted impacted sets of classes were actually changed.

Bianchi, et al., [Bianchi *et al.*, 2000] published a case study of the effectiveness of traceability models. This is a small exploratory case study based on a class project utilizing structural, name tracing, and cognitive links. The structural links are requirement traces and the cognitive links are based on system and domain knowledge. Additionally, the study was conducted using two different granularities of work products. The coarse granularity defined a work product as the entire source code class definition, whereas the finer grained work products were a subset of methods and attributes from a class definition. For both levels of granularity the actually impacted set of work products was included within the predicted impact set of work products. The number of predicted impacted work products that were not modified was higher for the finer grained model; however, this set of work products was a smaller percentage of the entire system than the coarser grained model. The analysis effort was also higher for the finer grained model. They discovered that when changes where implemented in the finer grained model, fewer faults were found. Even though greater analysis effort is required for the finer grained model, the authors indicate that the effort results in fewer faults when the modifications are implemented.

The decision to use requirements traceability must be made by management and integrated into the development process, [Gotel and Finkelstein, 1994; Ramesh *et al.*, 1995]. The types of traces that should be implemented should be determined by the use of the traces and the quality of the analysis that can be obtained from the traces [Cimitile *et al.*, 1999]. At the current time, requirements traceability is a manually intensive process, however there is research to provide tools and

TABLE 2.1. Comparison of Impact Analysis Approaches

Approach	Types of Links	Results
Pfleeger & Bohner	Requirement Traces	Complexity Measures
Barros, et al	Dependency and Requirement Traces	Interactive identification of potentially impacted work products
Lindvall & Sandahl	System and Domain Knowledge	Potentially impacted classes
Bianchi, et al.	Structural, Naming, and Cognitive	Finer grain work products produce better impact analysis, but at a higher effort

processes to facilitate the automatic creation of traces, [Spanoudakis, 2002; von Knethen, 2002].

2.6 Graph Theory

Many problems exist in the real work that can be represented visually by a diagram of points connected by lines that represent some relationship between the points. Diagraming work relationships between employees of a company could be done by representing the employees by a set of points and drawing a connecting line between two points if the employees represented by the points work on the same project. Another example is a subway system. The points would represent the stations of the subway system and a line connects two points if there is train service from one station to the other with no stops. Such representations provide a mechanism to determine whether employees work on the same project or whether there is direct train service between two specific train stations. These and similar concepts can be represented as a graph.

A graph, G, is defined as the tuple $\langle V(G), E(G), \psi_g \rangle$, where $V(G)$ is a non-empty set of vertices, $E(G)$ is a set of edges such that $V(G) \cup E(G) = \emptyset$, and an incidence function ψ_g which associates each edge in $E(G)$ with a pair of vertices in $V(G)$

[Bondy and Murty, 1976]. The vertices, or nodes, are the points in the previous example and the lines are edges, or arcs. Consider the graph shown in Figure 2.4.

$G = \langle V(G), E(G), \psi_G \rangle$

where:

$V(G) = \{v_1, v_2, v_3, v_4\}$

$E(G) = \{e_1, e_2, e_3\}$

and ψ_G is defined by

$\psi_G(e_1) = v_1v_2$, $\psi_G(e_2) = v_2v_3$, and $\psi_G(e_3) = v_2v_4$.

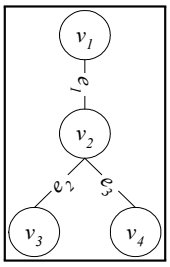

FIGURE 2.4. Graph G

Several special classes of graphs are of interest in this research. A complete graph is a graph in which every pair of distinct vertices is connected by an edge. A graph H is a subgraph of G, if $V(H) \subseteq V(G)$, $E(H) \subseteq E(G)$, and ψ_H is restricted to ψ_G defined for $E(H)$. A clique is a subgraph that is also a complete graph.

2.7 Fuzzy Set Theory

Fuzzy set theory is useful for modeling imprecise concepts. In conventional set theory, an element is either a member of a set or not a member of a set. These sets are called crisp sets. Membership in a set, S, can be determined by mapping the elements in an universal set, U, to 0 or 1 with the membership function m, where $m : U \rightarrow \{0, 1\}$. Therefore, for every $x \in U$, $x \in S$ iff $m(x) = 1$. Fuzzy sets are used to represent imprecision about set membership. Each element in a fuzzy

set has a degree of membership associated with it. The degree of membership is defined by a membership function, A, where each element x of U is mapped to a number in the closed interval $[0, 1]$, $A : U \rightarrow [0, 1]$ [Klir *et al.*, 1997].

One application of a fuzzy set is the concept of a long book. Some readers would consider a book of more than 100 pages to be long, while other readers may not think a book is particularly long unless it at least 500 pages long. These differing views can be modeled by a fuzzy set with the membership function L:

$$L(x) = \left\{ \begin{array}{cc} 1 & \text{when } pages \geq 500 \\ \frac{pages-100}{400} & \text{when } 100 < pages < 500 \\ 0 & \text{when } pages \leq 100 \end{array} \right\}$$

To select subsets of a fuzzy set that have at least some degree of membership, an α-cut, ${}^{\alpha}L$, of the fuzzy set can be used. The α-cut of a fuzzy set is the set of all elements whose membership degrees are greater than or equal to the value of α. The α-cut of ${}^{0.90}L$ would be the set of books that are 460 pages or more in length.

Fuzzy binary relations may be defined for the elements of a set. Fuzzy relations can be used to describe the similarity between two elements. If the fuzzy relation is reflexive and symmetric, but not transitive, it is a fuzzy compatibility relation. An α-cut of a fuzzy compatibility function can be used to partition a set into fuzzy compatibility classes. The members within each compatibility class are considered similar. The similarity between four items given by a fuzzy relation R, is shown by the matrix in Equation 2.1 and diagramed in Figure 2.5.

$$\hat{R} = \left[\begin{array}{cccc} 1 & .4 & 1 & .8 \\ .4 & 1 & .8 & 1 \\ 1 & .8 & 1 & .6 \\ .8 & 1 & .6 & 1 \end{array} \right] \tag{2.1}$$

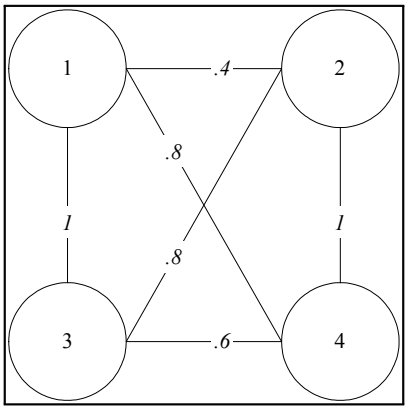

FIGURE 2.5. Representation of a fuzzy compatibility relation

The compatibility classes for this relation of α-cuts for $\alpha = 1, 0.8, 0.6, 0.4$ are:

$\alpha = 1.0 : \{1, 3\}\{2, 4\}$.

$\alpha = 0.8 : \{1, 3\}\{1, 4\}\{2, 3\}\{2, 4\}$.

$\alpha = 0.6 : \{1, 3, 4\}\{2, 3, 4\}$.

$\alpha = 0.4 : \{1, 2, 3, 4\}$.

The value of α determines the degree of similarity between the members of a compatibility classes.

2.8 Summary

We presented key research initiatives that provide a foundation for this research. We focused on traceability, which is the basis for the impact analysis method used in this research. We also included background information on graph theory and fuzzy set theory, with specific details on compatibility classes since this technique will be applied to the impact analysis results to distinguish the impact between proposed requirement changes.

Chapter 3
Modeling Software Changes

3.1 Introduction

Impact analysis requires a representation of the work products that are created to implement the software product. In this research, we define models representing the information required for performing impact analysis.

The work products created during software development include requirements, design diagrams, source code modules, and test cases. The relationship among these work products can be identified by using requirement tracing. Forward from requirements traces identify the work products that satisfy a requirement and therefore the work products that are most likely to be changed if the requirement is changed.

In order to perform impact analysis, we must have an understanding of the items that are created during a software development project and the relationships between those items. The formal representation for a software development product and its requirement changes consists of information about the items, work products that are part of the development project, and the developmental relationships linking the work products. The models, which are designed to capture information about the development project, assume traceability as a part of the development process.

3.2 Software Change Models

We define two models to capture information about the software development project and the proposed requirement changes, the Work product Information Model (WIM) and the Requirement changes Information Model (RIM). These two

models together are referred to as the Work product Requirements trace Model (WoRM). WoRM is the basis for the Trace-based Impact Analysis Methodology (TIAM) that is presented in Chapter 4.

WIM encapsulates information about the work products that are created to develop a software product. The primary components of WIM are the forward from requirement traces and the work products. The model also includes information about attributes of work products and traces that are used in TIAM.

RIM contains information on the requirement changes that are the focus for the analysis. It represents a set of associations between requirement changes and the requirements to be modified, along with a measure of the severity of the change.

The elements of the models, WIM and RIM, are the work products, forward from requirement traces, attributes associated with each work product, and an attribute associated with each trace.

3.2.1 Work Products and Traces

As defined in Section 2.2, a work product is a clearly defined piece of work that is to be maintained and whose developmental influences can be traced [Gotel and Finkelstein, 1997; IBM Object-Oriented Technology Center, 1997]. The granularity of the work products determines the effectiveness of the impact analysis. Granularity refers to the amount of information contained in each work product. Each work product should encapsulate as few details as possible, producing a fine granularity for effective analysis. Each requirement should be a separate work product. A design document such as a complete object table for an entire product is too coarse and should be subdivided into work products of class or object diagrams with one or more classes that are hierarchically or operationally related. Likewise for source code, a source code work product should be at most one class definition

and methods. A source code work product may contain only extensions to a class, which are a set of related methods added to a class.

Forward from requirements traces show the satisfaction of a requirement by linking products such as requirements to design and design to source code work products. These traces can be represented by a directed graph as described by [Pleeger and Bohner, 1990; Pfleeger, 1998; Bianchi *et al.*, 2000; Turver and Munro, 1996].

3.2.2 Attributes for Work Products and Traces

The model uses external attributes about each work product. External attributes are used so that all work products in a development project are treated in an uniform manner.

The external attributes for each work product are:

- Complexity of a work product
- Effort required to produce the work product, measured in person-hours
- Development phase in which the work product was created.

Developers assign values to the attributes as part of the development process. Examples of typical levels for the attributes are presented in Section 3.2.4.

The complexity of a work product is the degree of difficulty required to create a work product. The attribute value should be selected from a set of predefined levels for the development project. The number of levels and associated values could be extracted from cost models such as Boehm [Boehm, 1981] or from historical data about the development team. We provide an example in Section 3.2.4.

The effort attribute represents the amount of resources required to develop the work product. It must apply equally to work products in all phases of software

development. We use the number of person-hours required to create a work product as the value of the attribute.

The development phase is used to account for the increased effort required to change work products late in the product development cycle [Schach, 1999]. The inclusion of this function stems from observations that changes are more costly to make during later phases than in earlier phases. The factors for each phase can be chosen from studies such as Boehm [Boehm, 1981] or from historical data about the development team.

In addition for each trace, an external attribute representing the influence of the source work product on the target work product is assigned by the developer. The influence attribute accounts for the fact that target work products may have multiple sources, where each source may have a different influence on the target. The attribute value should be selected from a set of predefined levels for the development project.

3.2.3 WoRM

WoRM is composed of the two models, WIM and RIM. Figure 3.1 shows the relationship between the two models, where the work products in common are those requirements that are to be changed.

WIM represents the elements of a software development project that are to be used for impact analysis. In the graph-based model, work products are the nodes, or vertices, and the traces are represented directed edges between the nodes. Functions are defined for each attribute to associate the work product or trace attribute with the assigned level. In Section 3.2.4 we formally define WIM.

RIM contains information on the requirement changes that are the focus for the analysis. It is also graph based, with requirement changes represented as nodes and

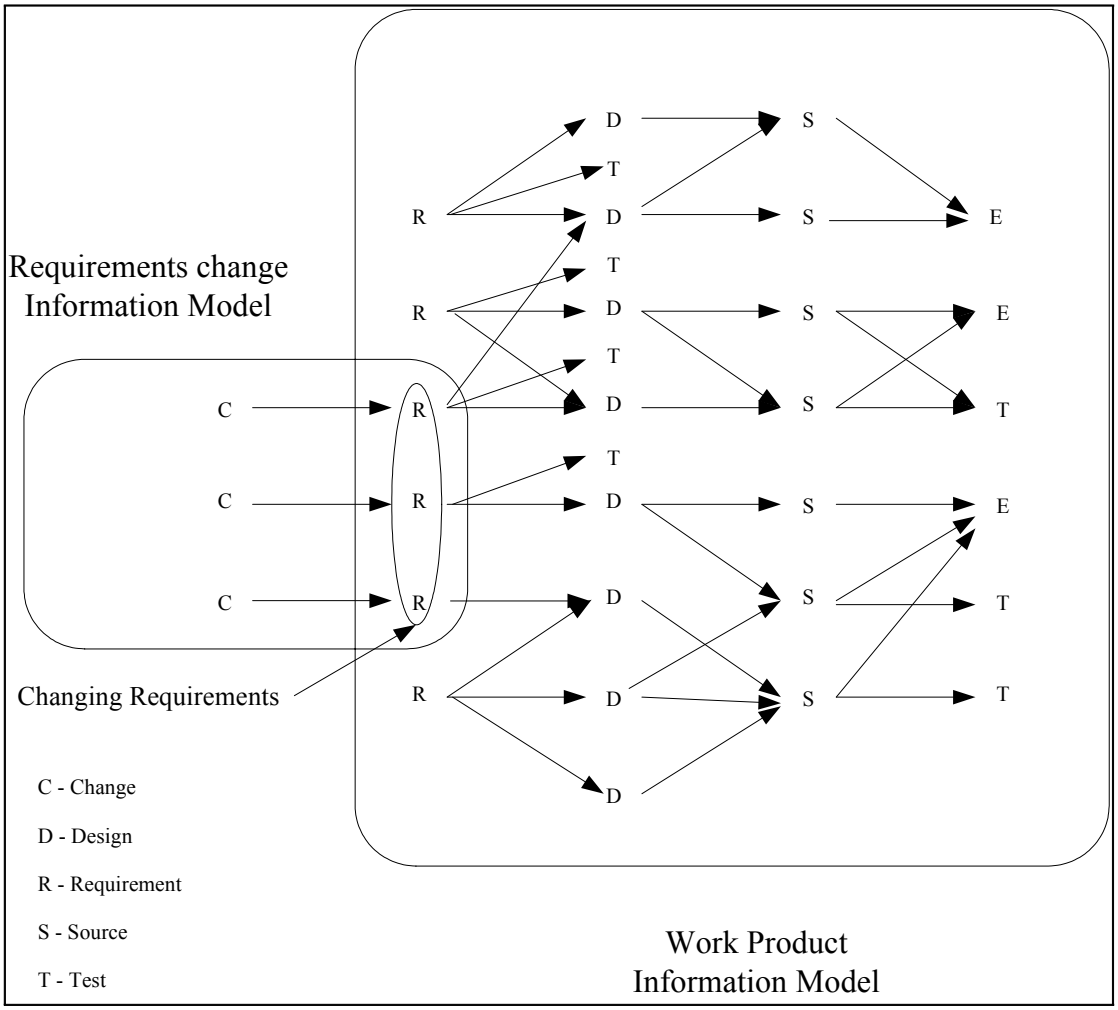

FIGURE 3.1. Work products Requirements trace Model

directed edges represent traces from requirement changes to affected requirements. A function associates each edge with the assigned severity of the change. Section 3.2.5 contains the formal definition for RIM.

3.2.4 WIM

WIM is defined as:

$$WIM = \langle Nodes, Traces, i, c, e, p \rangle \tag{3.1}$$

where:

- $Nodes$ represents the set of work products produced during product development.
- $Traces$ represents the set of edges representing requirement traces from source work products to target work products. It is a binary relation on $Nodes$: $Traces \subseteq Nodes \times Nodes$.
- The influence function i is defined as $i : Traces \rightarrow InfluenceSet$. It is the degree of influence that a source work product has on subsequent target work products.

  The $InfluenceSet$ is a set of influence levels. For example, if two levels of influence are desired then $InfluenceSet = \{strong, weak\}$ could be selected as the candidate levels. The labels *strong* and *weak* represent numeric values; thus the $InfluenceSet \subseteq \Re$.
- The complexity function c is defined as $c : Nodes \rightarrow ComplexitySet$. It is the estimated complexity of the work product.

  Like the $InfluenceSet$, the $ComplexitySet$ is a set of complexity levels, where $ComplexitySet \subseteq \Re$.
- The effort function e is defined as $e : Nodes \rightarrow \Re$. It is the development effort, in person-hours, required to create the work product. Any variability

between the productivity of developers should be managed by normalizing this value.

- The phase cost function p is defined as $p : Nodes \rightarrow PhaseSet$. It maps a work product to a factor associated with each phase. The $PhaseSet$ is dependent on the phases used by a development team, e.g. $\{Requirements$, $Analysis$, $Design$, $Implementation$, $SystemTest\}$.

3.2.5 RIM

RIM is defined as:

$$RIM = \langle ChangeSet, RequirementNodes, ChangeTraces, i \rangle \tag{3.2}$$

where:

- $ChangeSet$ is the set of requirement changes.
- $RequirementNodes$ represents the set of work products in WIM that are requirement work products.
- $ChangeTraces$ is the set of edges linking a requirement change to the associated node representing a requirement work product. It is a binary relation on $ChangeSet$ and $Nodes$: $ChangeTraces \subseteq ChangeSet \times RequirementNodes$.
- The function i is defined as $i : ChangeTraces \rightarrow InfluenceSet$. It is the degree of influence that the requirement change has on the changed requirement.

3.2.6 Definitions

The following definitions are used in TIAM:

Definition 3.1. *The **weight** w(n) of a node, or work product, is defined as:*

$$w(n) = e(n) \cdot c(n) \cdot p(n)$$

where $n \in Nodes$

and $e(n), c(n), p(n)$ are defined in WIM (3.3)

Definition 3.2. *The **normalized influence** of an edge, or trace, is defined as:*

$$i'(\{a,b\}) = \frac{i(\{a,b\})}{\sum_T i(\{c,b\})}$$
where $\{a,b\} \in Traces \cup ChangeTraces$

and $T = \{c | \{c,b\} \in Traces \cup ChangeTraces\}$. (3.4)

Definition 3.3. *The **work product impact metric**, $wpImpact$, is defined as:*

$$wpImpact(a) = w(a) + \sum_U (i'(\{a,b\}) \cdot wpImpact(b))$$
where $U = \{b | \{a,b\} \in Traces\}$ (3.5)

Definition 3.4. *The **requirement change impact metric**, $reqImpact(a)$, is defined as:*

$$reqImpact(a) = \sum_S (i'(\{a,b\}) \cdot wpImpact(b)),$$
where $S = \{b | \{a,b\} \in ChangeTraces\}$. (3.6)

3.3 WoRM Example

We provide a straight forward example of WIM using Figure 3.2. In WIM:

$Nodes = \{$Req A, Req F, Design B, Design C, Source J, Source K, Source L$\}$

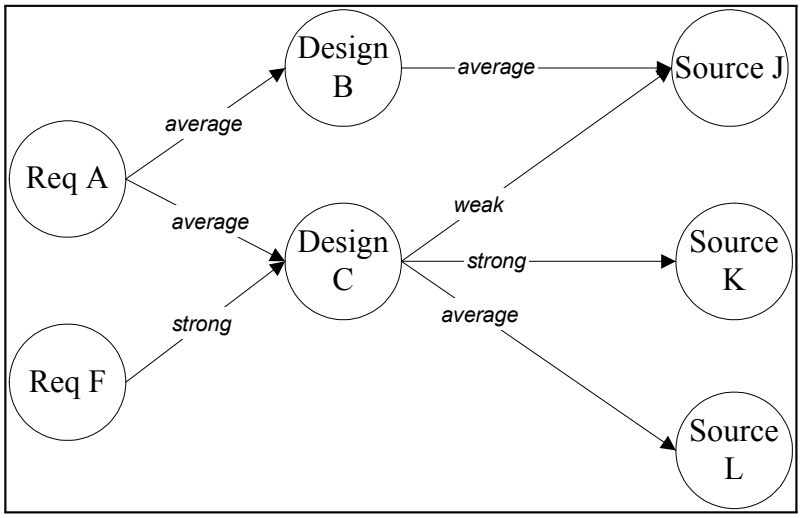

FIGURE 3.2. Product development diagram

$Traces = \{\{$Req A, Design B$\}, \{$Req A, Design C$\}, \{$Req F, Design C$\}, \{$Design B, Source J$\}, \{$Design C, Source J$\}, \{$Design C, Source K$\}, \{$Design C, Source L$\}\}$

Trace influence attribute information is assigned as given in Table 3.1, and work product attribute information is assigned as given in Table 3.2.

TABLE 3.1. Trace Influence Attributes

Trace: $\{a, b\}$	Influence: $i(\{a, b\})$
{Req A, Design B}	average
{Req A, Design C}	average
{Req F, Design C}	strong
{Design B, Source J}	average
{Design C, Source J}	weak
{Design C, Source K}	strong
{Design C, Source L}	average

Assume two changes, Change 1 and Change 2, which are changes to Req A and Req F, respectively. The new development diagram is given in Figure 3.3. The resulting RIM consists of:

$ChangeSet = \{$Change 1,Change 2$\}$

$RequirementNodes = \{$Req A, Req F$\}$

TABLE 3.2. Work product Attributes

Work product: a	Complexity: $c(a)$	Effort: $e(a)$	Phase: $p(a)$
Req A	medium	6	requirements
Req F	medium	8	requirements
Design B	medium	4	design
Design C	high	15	design
Source J	low	5	implementation
Source K	high	21	implementation
Source L	medium	7	implementation

$ChangeTraces = \{\{\text{Change 1, Req A}\}, \{\text{Change 2, Req F}\}\}$

The change trace influence attribute information is shown in Table 3.3.

TABLE 3.3. Requirement Influence Attribute

ChangeTrace: $\{a, b\}$	Influence: $i(\{a, b\})$
{Change 1, Req A}	weak
{Change 2, Req F}	average

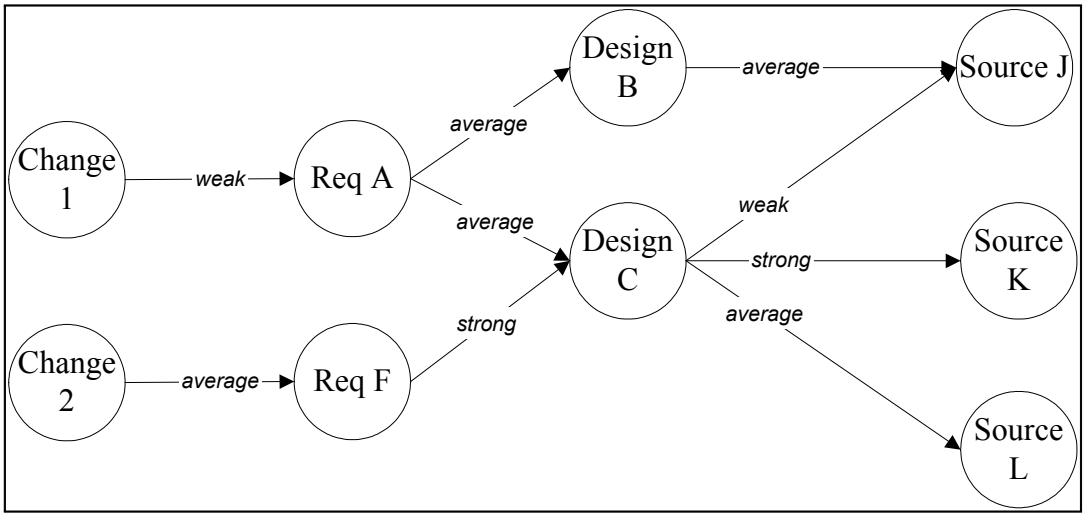

FIGURE 3.3. Product development diagram showing requirement change influences

3.4 Summary

We formally defined the WIM and RIM as the models that encapsulate the information required for the impact analysis and together form WoRM. We also defined basic formulas that are a part of TIAM. We describe TIAM in Chapter 4.

Chapter 4

A Methodology for Requirement Change Impact Analysis

4.1 Introduction

TIAM evaluates a set of requirement changes. The methodology is applied during the stage of planning the incorporation of a set of requirement changes into a version of a software product under development. The information provided by this methodology predicts the effects that modifications to the established requirements may have on work that has already been completed.

4.2 Overview

TIAM utilizes a multistep approach to impact analysis. The first step uses requirements traceability to find the predicted impact set of work products. With the attribute information from the traces and work products, an impact metric is computed for each requirement change. The requirement changes are grouped into classes of similar estimated impact using the technique of fuzzy compatibility classes. These classes are ordered with respect to increasing impact, providing sets of low to high impact requirement changes. With this information, management then evaluates the risk of implementing each requirement change dependent on the value of each requirement change to the overall success of the product.

The requirement change impact metric predicts the effect of the implementing a change on existing completed work products. This impact metric is computed using the attribute information of the predicted impact set of work products and the trace influence attribute. The requirement change impact metric is a fuzzy metric. It is computed from attribute levels assigned by the developers. Since the assignment for the levels of complexity and influence attributes are likely to vary

from developer to developer, the use of fuzzy techniques provide a mechanism to incorporate such differences. The impact metrics for a set of requirement changes are used to group requirement changes into fuzzy compatibility classes according to the differences in the impact metric. These classes are ranked according to the mean of the impact metrics in the class. This ordering prioritizes the classes by the impact that the changes may have on completed work, from low impact to high impact when ordered by ascending values of the mean of the class.

4.3 Assumptions

This methodology and its associated models are based on the following assumptions:

- Functional requirements of the product are traceable.
- Forward from requirements traceability is available.
- All work products (requirements, design documents, source code modules, documentation, and test cases) are available. All development work products should be current.
- Developers assign attribute levels for the trace link influence attribute and work product complexity attribute.
- Product is not monolithic in design or implementation, that is, the product is designed and implemented in modules. Preferably the project should utilize an object-oriented development paradigm, since good object-oriented development tends to maximize the independence of the modules.

4.4 TIAM Definition

TIAM is shown graphically in Figure 4.1 and algorithmically in Algorithm 4.1. The input for TIAM is a WoRM, composed of a WIM and the associated RIM. Outputs of the methodology are sets of potentially impacted work products for each change

and the ordered compatibility classes of requirement changes. Each requirement change is associated with a set of potentially impacted work products for that change labeled PI_i, where i is the requirement change. PPI, is a collection of all the sets of potentially impacted work products for each requirement change; therefore for every $i \in ChangeSet$, the corresponding $PI_i \in PPI$. The compatibility classes are collected in the list CCI where each compatibility class $CI_j \in CCI$ is the j^{th} class in the list. The computation of the impact metrics for a set of changes gives a vector V, where v_i is the impact metric for change $i \in ChangeSet$.

Algorithm 4.1 TIAM

Input: aWoRM, an instance of WoRM
α, degree of similarity within a compatibility class
Output: PPI, sets of potentially impacted work products for each change
CCI, a list of ordered compatibility classes of requirement changes.

1: $V \leftarrow \emptyset$
2: $PPI \leftarrow \emptyset$
{Traverse the model for each change.}
3: **for each** requirement change a, such that $a \in ChangeSet$ **do**
4: $v_a \leftarrow$ RequirementImpactTraversal$(aWoRM, PPI, a)$
5: Add v_a to V
6: **end for**
{Determine maximal compatibility classes.}
7: CompatibilityClasses(V, α, CCI)
{Order compatibility classes by increasing impact.}
8: OrderClasses(CCI, V)

4.4.1 Traversal

The first step in TIAM is the traversal of the set of potentially impacted work products and the computation of the requirement impact metric for each requirement change. This step is done via a depth first traversal of WIM following the traces.

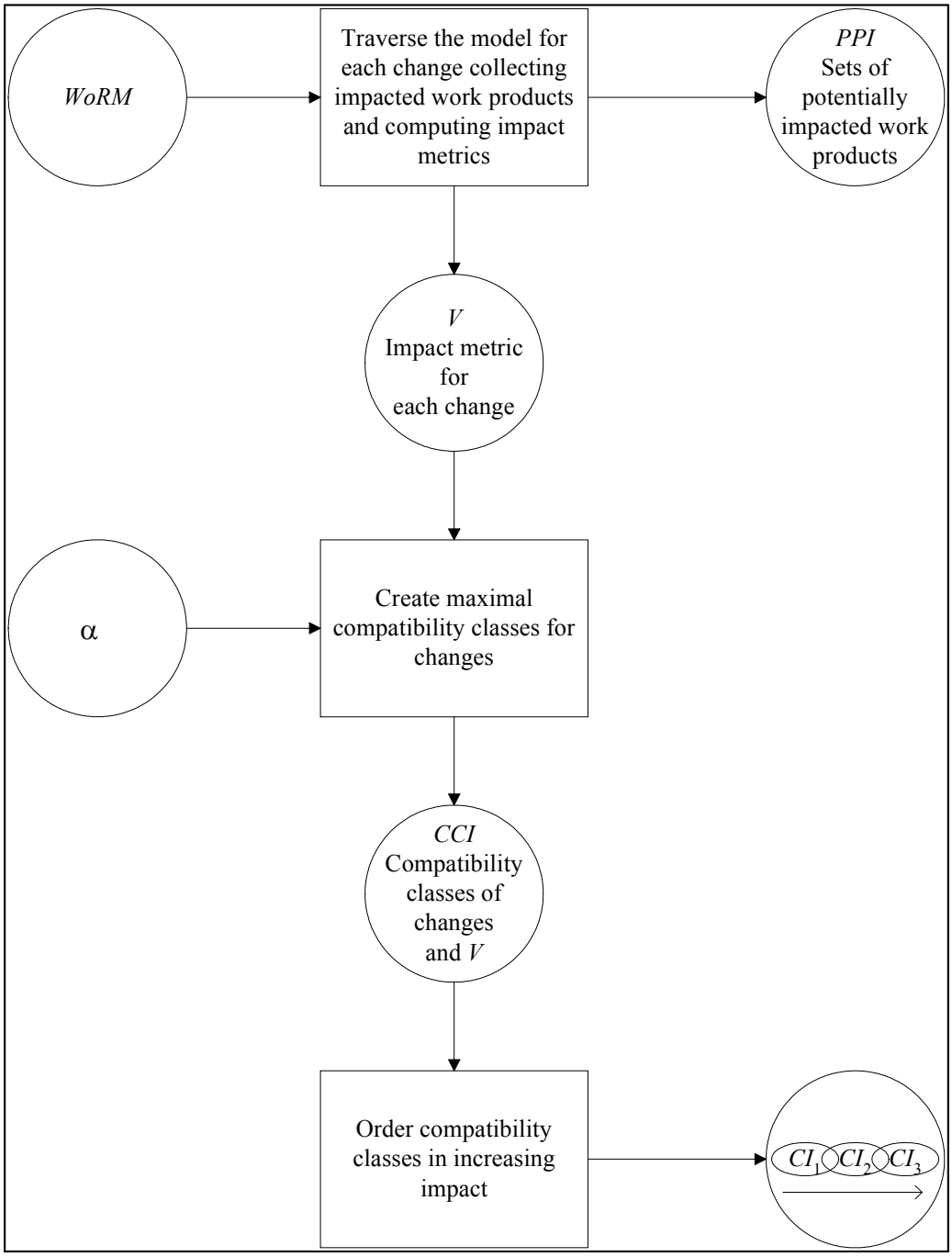

FIGURE 4.1. TIAM Diagram

Algorithm 4.2 implements the computation of Equation 3.6 and begins the collection of potentially impacted work products with the impacted requirements. The algorithm is invoked once for each requirement change, a. The set of potentially impacted work products for each change, PI_a, is added to the overall collection of potentially impacted work products sets, PPI. All impacted requirements are found by following the change traces from the requirement change. Each impacted requirement is added to PI_a. The requirement change impact metric, $reqImpact$, is the summation of the impact to each requirement affected by the requirement change. The impact for each requirement is the product of the normalized influence of the requirement change trace and the work product impact metric for each requirement.

Algorithm 4.2 RequirementImpactTraversal

Input: aWoRM, an instance of WoRM
PPI a set of potential impacted work products sets
a, a requirement change
Output: PPI
$reqImpact : \Re$

1: $PI_a \leftarrow \emptyset$
2: Add PI_a to PPI
3: $reqImpact \leftarrow 0$
4: **for each** b such that $\{a, b\} \in ChangeTraces$ **do**
5: add b to PI_a
6: $reqImpact \leftarrow reqImpact+$
$i'(\{a, b\})\cdot$ WorkProductImpactTraversal$(aWoRM, PI_a, b)$
7: **end for**
8: return $reqImpact$

The impact to work products from Equation 3.5 is implemented by Algorithm 4.3 which also collects the remaining potentially impacted work products. When invoked, the algorithm initializes a local work product impact metric to the weight of the work product. If there are any work products influenced by this work prod-

uct, as determined by the traces, the algorithm adds the traced work product to the set of potentially impacted work products, PI_a, and performs a summation of the impact to each traced work product added to the local work product impact metric. The impact for each traced work product is the product of the normalized influence of the trace and the returned value of a recursive call to the algorithm with the traced work product. If a traced work product has been reached before, the impact to the traced work product is the product of the normalized influence of the trace and the weight of the traced work product. The local work product impact metric is returned to the instance of the calling algorithm for use in its calculations.

Algorithm 4.3 WorkProductImpactTraversal

Input: aWoRM, an instance of WoRM
PI_a, a set of potential impacted work products
b, a work product
Output: PI_a
$wpImpactValue : \Re$

1: $wpImpact \leftarrow w(b)$
2: **for each** c such that $\{b, c\} \in Traces$ **do**
3: **if** $c \notin PI_a$ **then**
4: add c to PI_a
5: $wpImpact \leftarrow wpImpact+$
$i'(\{b, c\})\cdot$ WorkProductImpactTraversal$(aWoRM, PI_a, c)$
6: **else**
7: $wpImpactValue \leftarrow wpImpact + i'(\{b, c\})\cdot w(c)$
8: **end if**
9: **end for**
10: return $wpImpact$

4.4.2 Similarity

The second step in TIAM is the grouping of changes as fuzzy compatibility classes. Each change in a compatibility class has a predicted impact similar to the other

changes in the same class. CCI is the list of compatibility classes with each compatibility class $CI_i \in CCI$, the i^{th} class in the list.

We define the fuzzy compatibility relation R, given in Equation 4.1, to find classes of requirement changes that have a similar degree of impact. The relation is based on the distance between the impact metrics [Klir and Yuan, 1995].

$$R(v_i, v_j) = 1 - \delta(\mid v_i - v_j \mid),$$
$$\text{letting } \delta = \frac{1}{m}, \text{ where } m \text{ is the largest value in } V. \tag{4.1}$$

The resulting relationship matrix is used to find compatibility classes of requirement changes based on a specific α. As the value of α increases, the resulting classes are more refined. The value of α creates the α-cut of the set of requirement changes, that is the classes of changes that are related at least by the value of α. One way of finding the compatibility classes is to use a graph theory approach where each change is a vertex and two vertices, a and b, are adjacent if $R(a, b) >= \alpha$. Each clique formed in the resulting graph represents a compatibility class. The cliques can be found by visual inspection or algorithmically. The nodes for each clique i form a class CI_i that is added to CCI. With this approach of finding the classes, algorithms for finding cliques in a graph cannot be solved in deterministic polynomial time [Aho *et al.*, 1974; Horowitz and Sahni, 1978]. Generally, algorithms that cannot be solved in deterministic polynomial time cannot be solved for large sizes of problems but may be solved for small problem sizes in reasonable time or space.

However, we can find the compatibility classes in polynomial time by normalizing the impact metrics in the interval of [0,1] and finding the impact metrics that are within a specified distance from each other. The distance is $1 - \alpha$, for the selected value of α. Algorithm 4.4 implements this approach. This algorithm is bounded by

two nested loops, lines 10 through 17, which compare the distance between values, making the algorithm's upper bound complexity $O(n^2)$.

Algorithm 4.4 CompatibilityClasses

Input: V, a list of requirement impact metric values
α, degree of similarity within a compatibility class
Output: V, a list of requirement impact metric values
CCI, list of compatibility classes of requirement changes

```
1:  V' ← ∅
2:  CCI ← ∅
3:  m ← Max(V)
4:  d ← 1 − α
5:  for each v ∈ V do
6:     add v/m to V'
7:  end for
8:  Sort V', creating mapping P, such that P_i is the change associated with v'_i
9:  previousClass ← ∅
10: for i = 1 to |V'| do
11:    currentClass ← ∅
12:    add P_i to currentClass
13:    for j = i + 1 to |V'| do
14:       if v'_j − v'_i < d then
15:          add P_j to currentClass
16:       end if
17:    end for
18:    if currentClass ⊊ previousClass then
19:       add currentClass to CCI
20:       previousClass = currentClass
21:    end if
22: end for
```

4.4.3 Ordering

The third step in TIAM is the ordering of the compatibility classes from lowest to highest predicted impact. We rank the classes by sorting the list CCI in ascending order by the mean of the impact metrics associated with each change in a class. When Algorithm 4.4 is used, then the CI's in CCI are already in the proper order. If the fuzzy compatibility relation R is computed and the compatibility

classes found by visual inspection of the resulting graph, Algorithm 4.5 can be used to rank the classes.

Algorithm 4.5 OrderClasses

Input: CCI, list of compatibility classes of requirement changes
V, a list of requirement impact metric values
Output: CCI, list of ordered compatibility classes of requirement changes

1: **for each** $CI \in CCI$ **do**
2: compute average of impact metrics for changes in CI
3: **end for**
4: Sort each CI in CCI in ascending order of its average impact

4.5 Methodology Example

We use the trace and work product example from Chapter 3 to show an application of the methodology. A third change, Change 3, and additional work products are added to the example to elaborate the grouping of changes. In this example we associate the following values with attribute levels:

- $InfluenceSet = \{strong = 1, average = 0.6, weak = 0.3\}$
- $ComplexitySet = \{high = 1, medium = 0.6, low = 0.3\}$
- $PhaseSet = \{Requirements = 1,\ Design = 2,\ Implementation = 3\}$.

Figure 4.2 shows the computed weight of each work product beneath its label. Each dependency trace is labeled with its normalized influence value.

The first step of TIAM is the traversal of the model for each requirement change to find the potentially impacted work products and compute the requirement change impact metric. RequirementImpactTraversal, Algorithm 4.2, is invoked. $PI_{Change1}$ is initialized to empty and $reqImpact$ initialized to zero. Since there is a change trace from Change 1 to Req A, Req A is added to $PI_{Change1}$ and the computation of $reqImpact$ begins with a call to WorkProductImpactTraversal with the parameters:

aWorm = model instance passed to RequirementImpactTraversal

$PI_{Change1}$ = {Req A}

b = Req A.

When returning from the initial call to WorkProductImpactTraversal, the $wpImpact$ value of 51.5625 is then returned to RequirementImpactTraversal which is then returned to TIAM where $v_{Change1}$ is set to 51.6.

Calls to RequirementImpactTraversal for Change 2 and for Change 3, return 71.7 and 65 respectively, producing $V = \{51.6, 71.7, 65\}$ and PPI ={$PI_{Change1}$, $PI_{Change2}$, $PI_{Change3}$} where:

$PI_{Change1}$ ={Req A, Design B, Design C, Source J, Source K, Source L}

$PI_{Change2}$ ={Req F, Design C, Source J, Source K, Source L}

$PI_{Change3}$ ={Req G, Design H, Source I, Source M}.

The second step of TIAM can be completed by Algorithm 4.4 or by computing the relationship matrix. In order to demonstrate the grouping of changes into classes that have a similar impact metric we use the following relationship matrix computed from V:

$$R(v_i, v_j) = 1 - \delta(\mid v_i - v_j \mid) = \begin{bmatrix} 1 & 0.72 & 0.81 \\ 0.72 & 1 & 0.91 \\ 0.81 & 0.91 & 1 \end{bmatrix}$$

The value of α chosen for determining the fuzzy compatibility classes is the degree of similarity of the impact metrics for changes in a compatibility class. The determination of the appropriate α-cut is dependant on the parameters within the development organization. From the relationship matrix, it can be seen that for α = 0.7, we can make no distinction between the changes because all belong to the same maximal compatibility class. For this example, we select an $\alpha = 0.9$. Only

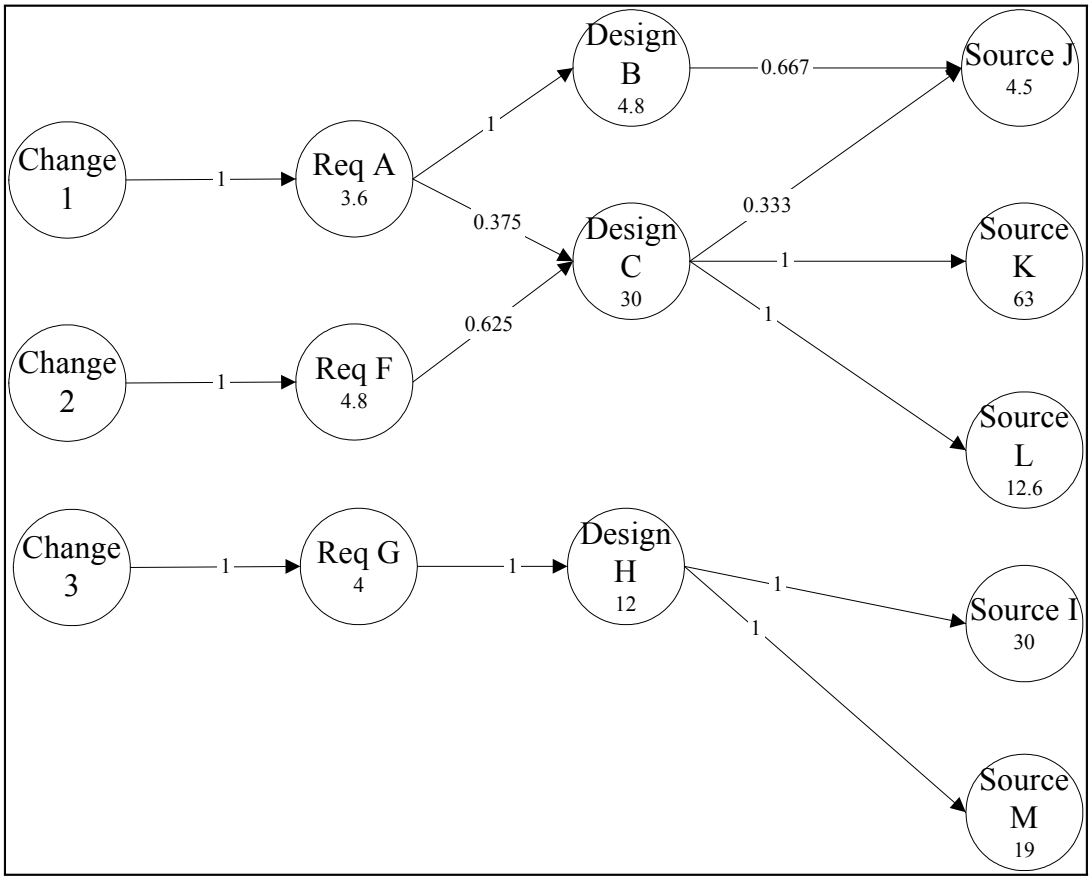

FIGURE 4.2. Work product diagram showing traces and attribute values

Change 2 and Change 3 have a similarity greater than 0.9, $R(v_{Change2}, v_{Change3}) = 0.92$. This α-cut partitions the changes into two compatibility classes, one with Change 2 and Change 3 and the other with Change 1. The compatibility classes produced are $CCI = \{CI_1, CI_2\}$, where:

CI_1 ={Change 1}

CI_2 ={Change 2, Change 3}.

The final step of TIAM is ordering the compatibility classes in increasing impact. Algorithm 4.5, OrderClasses, specifies the classes be sorted in increasing order according to the average of the impact metrics in each class. In this example, the compatibility classes in CCI are already in the appropriate order with the changes in CI_2 expected to have a greater impact than the change in CI_1.

4.6 Summary

TIAM is a predictive impact analysis method. For a set of requirement changes, the potentially impacted work products are identified, and an impact metric is computed. Next, the requirement changes are grouped into classes where the changes in each class have a similar predicted impact. Finally, the classes of changes are ordered in increasing impact. The application of TIAM characterizes changes based on the level of impact of a change.

Chapter 5
Experimental Results

5.1 Project Description

In this chapter, we present a case study in which a graduate level software engineering course project served as the case study. Students in the Computer Science graduate programs have the ability to program in a high level programming language and have taken undergraduate courses in data structures, operating systems, and programming languages. The class was divided into six teams, with each team consisting of three or four members. The same project and problem description were assigned to each team. The project was a web-enabled time tracking and billing system for a hypothetical software company. The problem description is provided in Appendix A. At the completion of the project, each team was required to submit a project portfolio with the following items:

- requirement work products document used to specify the functionality of the system
- all analysis work products
- all design work products
- source code module listing
- test plans and test work products
- user and operator instructions and guides
- work product attribute table
- work product traceability matrix
- requirement change impact analysis reports
- any instructions necessary to install, configure, and run the system.

During the implementation phase of development, six requirement changes were assigned to the teams. The requirement changes are given in Appendix B. Trace data and work product attribute data were collected from the teams to perform predictive impact analysis with TIAM. The teams were instructed to track the effort in person-hours expended while making these changes to existing work products. An impact analysis document for each change was required as part of the final documentation for the project. The impact analysis report for each change lists the requirements that were affected and the severity of the change, in terms of the influence attribute levels. It also lists the amount of effort that was required to make the changes to existing work products.

For each team, the predicted impact for each change was computed by applying TIAM, based on the requirement trace information and work product attribute data reported by the two teams. The actual impact for each change is the effort required to make the change as reported by the teams. We used the fuzzy compatibility relation R, Equation 4.1, to define compatibility classes on the actual data in the same manner as we use it for the predicted impact method analysis. The purpose was to determine the degree of similarity between the actual impact classes and the predicted impact classes.

5.2 Case Study Details

Attribute levels for work product complexity were chosen from the set: {*low*, *nominal*, *high*}. For the requirement traces, the influence attribute levels were chosen from the set: {*weak*, *average*, *strong*}. The effort attribute for each work product, recorded by the team as each work product was being developed, is the number of person-hours expended in developing the work product. Work products

were reported from the requirements phase, design phase, and implementation phase of development at the time changes were introduced.

The six requirement changes are identified as a, b, c, d, e, and f. Each change was written as a change to an requirement that was expected to be developed from the original problem description.

To perform the computation of the impact metric, the initial values for the attribute levels were selected to represent a balanced effect across the range of levels. The complexity attribute levels were $\{low = 1, nominal = 2, high = 3\}$. The trace influence attribute levels selected were $\{weak = 0.3, average = 0.6, strong = 0.9\}$. Since the project was constrained to a semester the attribute levels for phase were selected to be $\{requirements = 1, design = 1, implementation = 2\}$.

5.3 Case Study Data

Table 5.1 summarizes the number of work products by the phase in which the work product was created. The granularity of work products appears to differ markedly between the groups. Since each team had completed the design phase when the changes were given to them, we expected that each team would have completed similar designs and have identified the same magnitude of design work products. Data was collected from each of the six teams at the conclusion of the project; however only two teams provided data in enough detail to be useful for analysis. Two teams provided incomplete impact analysis information, while the remaining two provided no reports. Upon inspection of the project portfolios of the four teams that did not provide enough impact data, the work products of these team were also too coarsely defined for effective analysis.

TABLE 5.1. Summary of Work Reported by Each Team When Changes Introduced

Number of	Team					
	1	2	3	4	5	6
Requirement work products	23	70	9	4	32	10
Design work products	34	28	5	4	8	4
Implementation work products	27	15	7	4	0	6
Number of traces	249	1101	32	32	210	83
Person-hours reported	140	114.5	72	41	54	94.5

We present the results of teams 1 and 2 with the impact analysis and the application of the impact prediction methodology, using the six changes given to each team. The actual impact to the each team project depends on the work that the team had completed at that point.

5.3.1 Results from Team 1

For team 1, two of the changes did not impact any existing work products because the changes were implemented in work products that were not created at the time the changes were introduced. Only one work product was changed that was not included in the set of potentially impacted work products. Table 5.2 provides a summary of the impact to the project for team 1 by showing the number of predicted impact work products and number of actually impacted work products.

TABLE 5.2. Summary of Predicted and Actual Modified Work Products

Work Products	Change					
	a	b	c	d	e	f
Changed and predicted impacted	0	8	1	2	4	0
Unchanged and predicted not impacted	48	39	45	43	50	45
Changed and predicted not impacted	0	0	1	0	0	0
Unchanged and predicted impacted	36	37	37	39	30	39

The actual impact for the set of changes, $\{a, b, c, d, e, f\}$, is $V = \{0, 6, 2, 2, 4, 0\}$, where $v_a = 0$, $v_b = 6, \ldots$, $v_f = 0$. V represents the actual time required to make

the changes. Applying the compatibility relation R on V, as defined in Equation 4.1, yields the following matrix:

$$R(v_i, v_j) = 1 - \delta(\mid v_i - v_j \mid) = \begin{bmatrix} 1 & 0 & .67 & .67 & .33 & 1 \\ 0 & 1 & .33 & .33 & .67 & 0 \\ .67 & .33 & 1 & 1 & .67 & .67 \\ .67 & .33 & 1 & 1 & .67 & .67 \\ .33 & .67 & .67 & .67 & 1 & .33 \\ 1 & 0 & .67 & .67 & .33 & 1 \end{bmatrix}$$

From this matrix, we obtain the following maximal compatibility classes:

$\alpha = 0.3 : \{\{a, f, c, d, e\}, \{c, d, e, b\}\}$

$\alpha = 0.5 : \{\{a, f, c, d\}, \{c, d, e\}, \{e, b\}\}$

$\alpha = 0.7 : \{\{a, f\}, \{c, d\}, \{e\}, \{b\}\}$

We show these classes graphically in Figure 5.1.

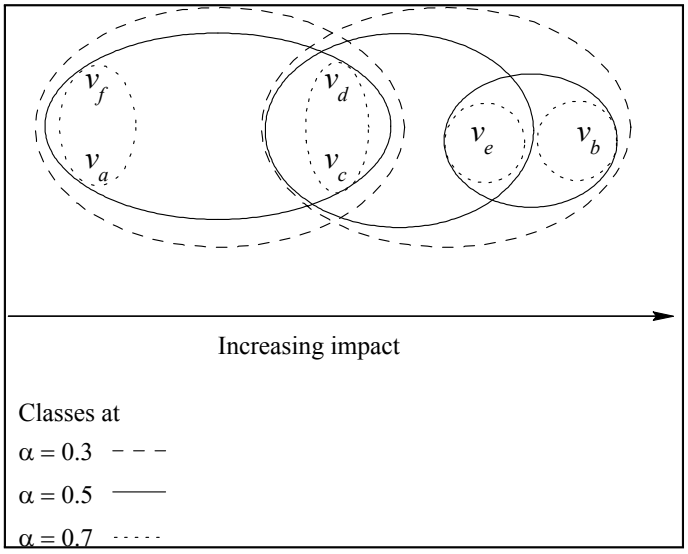

FIGURE 5.1. Compatibility classes for team 1, actual impact

We used TIAM to produce the sets of potentially impacted work products and the impact metric for each change. The number of potentially impacted work prod-

ucts for each change is summarized in Table 5.2. With one exception, the impacted work products were included in the sets of potentially impacted work products. The number of predicted not impacted work products that would not be impacted for each change was at least 50% of the total number of work products. As expected there were a number of work products that were predicted to be impacted that were not changed.

The predicted impact metrics for this set of changes are given by $\hat{V}$, where $\hat{V} = \{9.04, 39, 9.82, 17.08, 20.84, 16.8\}$. The computation of the predicted compatibility relationship $\hat{R}$ on $\hat{V}$ gives the following matrix and maximal compatibility classes:

$$\hat{R}(\hat{v}_i, \hat{v}_j) = 1 - \delta(\mid \hat{v}_i - \hat{v}_j \mid) = \begin{bmatrix} 1 & .23 & .98 & .79 & .7 & .8 \\ .23 & 1 & .25 & .44 & .53 & .43 \\ .98 & .25 & 1 & .81 & .72 & .82 \\ .79 & .44 & .81 & 1 & .9 & .99 \\ .7 & .53 & .72 & .9 & 1 & .9 \\ .8 & .43 & .82 & .99 & .9 & 1 \end{bmatrix}$$

$\alpha = 0.2 : \{\{a, c, d, f, e, b\}\}$

$\alpha = 0.24 : \{\{a, c, d, f, e\}, \{c, d, f, e, b\}\}$

$\alpha = 0.5 : \{\{a, c, d, f, e\}, \{e, b\}\}$

$\alpha = 0.6 : \{\{a, c, d, f, e\}, \{b\}\}$

$\alpha = 0.75 : \{\{a, c, d, f\}, \{d, f, e\}, \{b\}\}$

$\alpha = 0.9 : \{\{a, c\}, \{d, f, e\}, \{b\}\}$

In Figure 5.2, we show selected maximal compatibility classes for selected values of α graphically.

At an $\alpha = 0.5$ for the actual impact, we have the classes, $CCI = \{\{a, f, c, d\}, \{c, d, e\}, \{e, b\}\}$. For the predicted impact for $\alpha = 0.5$, the classes $CCI = \{\{a, c, d, f, e\}, \{e, b\}\}$ where found. The method predicted changes b and e would have a higher

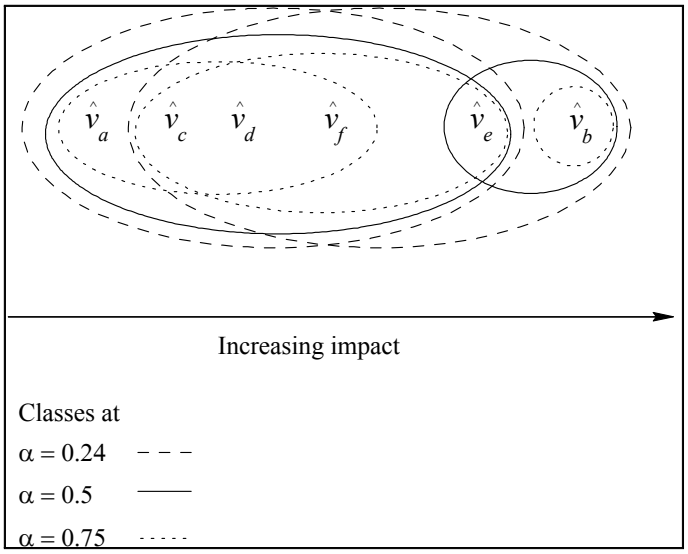

FIGURE 5.2. Compatibility classes for team 1, predicted impact

impact than the other changes. This prediction concurs with the actual results, even to predicting that change b would have the greatest impact of all changes. Other changes were predicted to have a lower impact and to be similar in degree. The method predicted that impact from changes a and f were similar to changes c and d. The actual impact analysis showed these two sets of changes were not similar, since no effort was reported to be required for changes a and f. If any work products are potentially impacted, our method will compute a nonzero impact metric.

5.3.2 Results from Team 2

We next applied TIAM to the data from Team 2. Two of the changes were reported as having no affect on requirements as specified by the team. As a consequence the methodology could not be applied to these two changes. In one of these cases, the design of the product allowed for the change without modification of the product. As expected there were a number of work products that were predicted to be impacted that were not changed. Table 5.3 provides a summary of the predicted impacted work products and the actually impacted work products to the project.

TABLE 5.3. Summary of Predicted and Actual Modified Work Products

Work Products	Change					
	a	b	c	d	e	f
Changed and predicted impacted	14	12	-	11	-	10
Unchanged and predicted not impacted	80	90	-	86	-	85
Changed and predicted not impacted	5	4	-	4	-	4
Unchanged and predicted impacted	20	13	-	18	-	20

TIAM produced the sets of potentially impacted work products and the impact metric for each change. The number of potentially impacted work products for each change is summarized in Table 5.3. Though four to five work products were changed that were not predicted to be impacted for each change, the majority of changed work products were predicted to be impacted. For each change, least 65% or more of the total work products were unchanged and predicted not be impacted. Again, there were a number of work products that were predicted to be impacted that were not changed.

For team 2, the actual impact for changes $\{a, b, d, f\}$ was $V = \{3, 2, 3, 4\}$, producing the compatibility matrix and maximal compatibility classes:

$$R(v_i, v_j) = 1 - \delta(\mid v_i - v_j \mid) = \begin{bmatrix} 1 & .75 & 1 & .75 \\ .75 & 1 & .75 & .5 \\ 1 & .75 & 1 & .75 \\ .75 & .5 & .75 & 1 \end{bmatrix}$$

$\alpha = 0.5 : \{\{b, a, d, f\}\}$

$\alpha = 0.7 : \{\{b, a, d\}, \{a, d, f\}\}$

$\alpha = 0.8 : \{\{b\}, \{a, d\}, \{f\}\}$

Selected classes are shown in Figure 5.3.

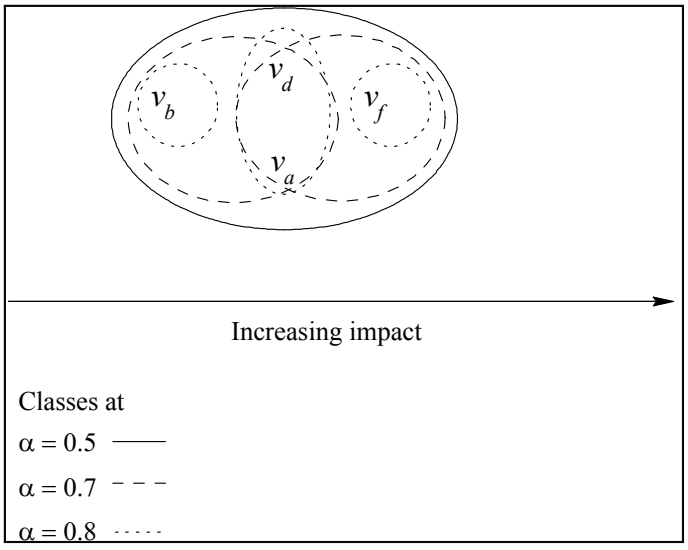

FIGURE 5.3. Compatibility classes for team 2, actual impact

This team reported two changes, c and e, as not affecting any requirements. We could not compute an impact metric for these changes. So we predict using the changes $\{a, b, d, f\}$. The predicted impact metrics for this set of changes is given by $\hat{V}$, where $\hat{V} = \{5.98, 4.48, 5.58, 3.8\}$. The computation of the predicted compatibility relationship $\hat{R}$ on $\hat{V}$ gives the following matrix and maximal compatibility classes:

$$\hat{R}(\hat{v}_i, \hat{v}_j) = 1 - \delta(\mid \hat{v}_i - \hat{v}_j \mid) = \begin{bmatrix} 1 & .75 & .93 & .63 \\ .75 & 1 & .82 & .89 \\ .93 & .82 & 1 & .7 \\ .64 & .89 & .7 & 1 \end{bmatrix}$$

$\alpha = 0.6 : \{\{f, b, d, a\}\}$

$\alpha = 0.7 : \{\{f, b, d\}, \{b, d, a\}\}$

$\alpha = 0.8 : \{\{f, b\}, \{b, d\}, \{d, a\}\}$

$\alpha = 0.9 : \{\{f\}, \{b\}, \{d, a\}\}$

We show selected maximal compatibility classes in Figure 5.4.

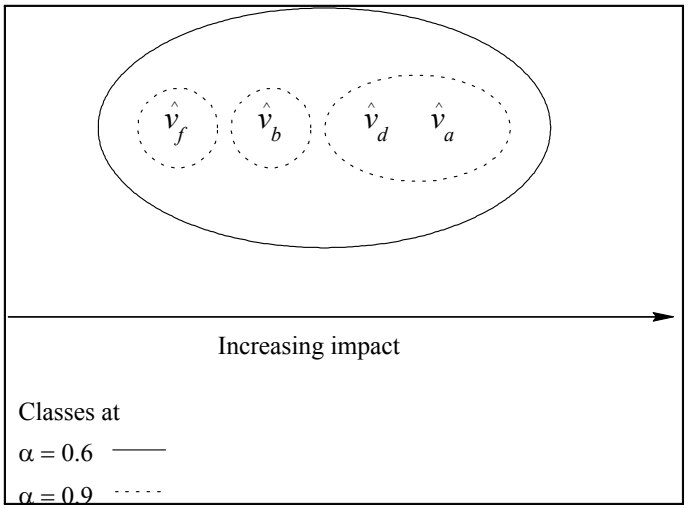

FIGURE 5.4. Compatibility classes for team 2, predicted impact

For team 2, at $\alpha = 0.5$ the actual and predicted impact for changes a, b, d, and f are similar. In this set of changes, no change seems to have a significantly higher impact than the other changes at this selected value. The method also found that changes a and d are similar at a high value of α, for the actual and predicted analysis.

5.4 Summary

The methodology results for team 1 identified two changes that were similar and had a predicted higher impact than the other changes. The actual impact agrees with the predicted impact for these changes. The remaining changes where predicted to be similar, where the actual impact for two of those changes was not similar and was less than the predicted impact. The actually changed work products were included in the sets of potentially impacted work products, except for one work product. The methodology results from team 2 found that all the changes would have a similar impact. The actual impact for these changes was also found to be similar. The majority of actually modified work products was included in the predicted to be modified sets of work products.

This case study demonstrates that the methodology creates classes of requirement changes. These classes are similar to classes generated from the actual impact to the projects. Our observation is that an α-cut of 0.5 is appropriate for the methodology, though we expect this to vary depending on the development group. The selection of α-cuts above $\alpha = 0.6$ did not yield useful predictive classes. This behavior is expected since the method is dependant on the impact metric which is fuzzy. The sets of potentially impacted work products effectively reduces the number of work products that would need to be considered for further impact analysis, with respect to the total number of work products.

In summary, the experimental results support the predictive ability of TIAM. We compared the application of TIAM on two software development projects to the actual impact on each project, and found that methodology has predictive value for finding classes of similar changes.

Chapter 6
Methodology Validation

6.1 Introduction

In order to validate the methodology, we view the compatibility classes as graphs and use a technique for comparing graphs for similarity. We then describe the similarity of the graphs from the actual impact classes with the graphs of the predicted impact classes. Finally, we examine the use of the attributes to better facilitate predictions.

6.2 Results

We represent the results of TIAM as a graph. Each change is a node in the graph, and each node is connected by an edge if the value of the fuzzy compatibility relation value is equal to or greater than the value of the selected α-cut. The maximal compatibility classes of the requirement changes are represented by the cliques formed in the representative graph, where a clique is a subset of nodes of a graph that form a complete graph [Bondy and Murty, 1976]. The graph accurately represents the compatibility class but the information as to which class was of greater impact is lost. We must consider this ordering separately from comparing the similarity of graphs.

6.2.1 Distance Between Graphs

When applying a metric to determine the similarity between two graphs, we refer to the distance between two graphs. If the distance between two graphs is 0, then an isomorphism exists between the two graphs. For an isomorphism to exist, there must be a one-to-one correspondence between the vertices and edges

such that the adjacencies of the vertices are preserved. Recent work on distance between graphs is given by [Chartrand *et al.*, 1998; Goddard and Swart, 1996; Papadopoulos and Manolopoulos, 1999]. Two categories of distance measures identified in [Papadopoulos and Manolopoulos, 1999] are feature-based distances and cost-based distances. Feature-based distances extract a set of features from the structural representation as a n-d vector. Cost-based distances between two graphs measures the number of modifications required to transform one graph into an isomorphism of the other graph.

We apply a cost-based distance in the comparisons, primarily because in this application the vertices are identical in the graphs. The modifications, or transform operations are adding an edge, making two vertices adjacent, or deleting an edge and removing the adjacency property between two edges. Such transform operations are also referred to as vertex updates. The cost-base metric is straightforward to calculate by assigning a cost for the addition of an edge and a cost for the deletion of an edge. For our distance measures we assign a unit cost to each operation, weighting both operations equally.

Recall in Section 2.6, a graph, G, is defined as the tuple $\langle V(G), E(G), \psi_g \rangle$, where $V(G)$ is a non-empty set of vertices, $E(G)$ is a set of edges, and an incidence function ψ_g. We define a distance between two graphs if, and only if, the vertices are the same, $V(G_1) = V(G_2)$. An edge deletion and edge insertion have a unit cost and therefore we formally define the distance between two graphs, $Dist(G_1, G_2)$, as the number of edges not equivalent in each graph. Equivalence between edges is given in Definition 6.1, and distance between graphs in Definition 6.2.

Definition 6.1. *For two graphs, G_1 and G_2, edges $e_1 \in V(G_1)$ and $e_2 \in V(G_2)$ are equal if and only if $\psi_{G_1}(e_1) = \psi_{G_2}(e_2)$.*

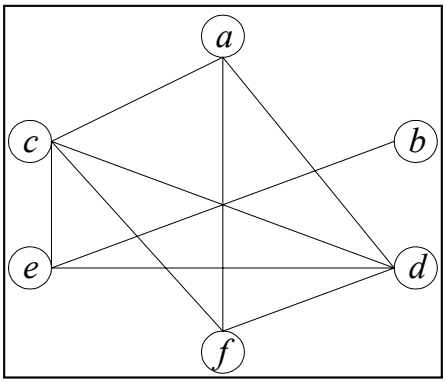

FIGURE 6.1. Team 1 actual impact graph

Definition 6.2. *For two graphs, G_1 and G_2, the distance between G_1 and G_2 is* $Dist(G_1, G_2) = |E(G_1) \cup E(G_2)| - |E(G_1) \cap E(G_2)|$.

6.3 Comparison of Case Study Results

In the case study, the compatibility classes formed at an α-cut of 0.50 provide the most favorable results for this case study. We use this α-cut for the compatibility classes that we compare in this section.

6.3.1 Team 1 Results

The graph that represents the compatibility classes for the actual impact for team 1 is shown in Figure 6.1. The graph of predicted compatibility classes is shown in Figure 6.2. To transform the predicted graph to the actual graph requires two edge deletion operations, removing (a, e) and (e, f). The distance measure between these two graphs is therefore 2.

The minimum distance between two graphs with six vertices is 0, if the graphs are isomorphic. The maximum distance between two such graphs is 15, the number of edges in a complete graph with 6 vertices, which would mean the two graphs had no adjacency properties in common. Therefore a distance of 2 indicates that the graphs are similar but not isomorphic to each other.

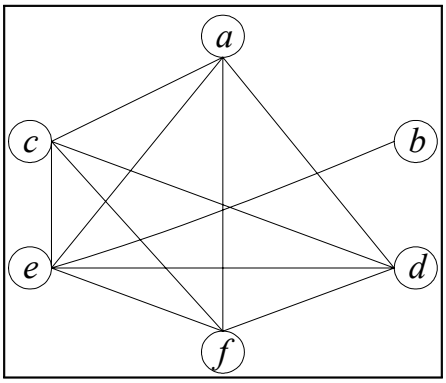

FIGURE 6.2. Team 1 predicted impact graph

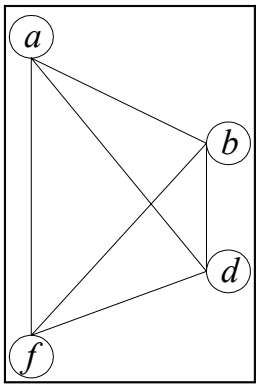

FIGURE 6.3. Team 2 actual and predicted impact graph

6.3.2 Team 2 Results

For team 2, the requirement changes of c and e are not included in the predicted requirement changes set, so we do not include these changes in the graph for the actual requirement changes. Figure 6.3 shows the graph representing the actual compatibility classes. It also represents the predicted impact classes. The distance between these two graphs is 0.

6.3.3 Conclusion

The distance between the actual classes graph and the predicted classes graph for team 1 is 2 and for team 2 is 0, demonstrating that the predicted classes are similar in the case of team 1 to the actual classes and the actual classes are the same as

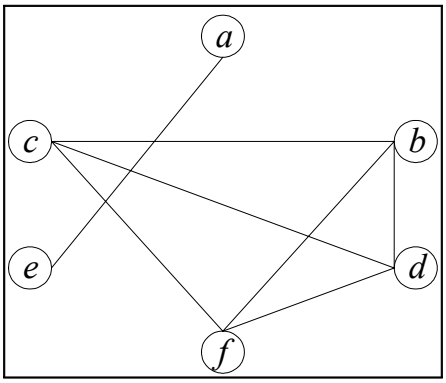

FIGURE 6.4. Team 1 number of work products graph

the predicted classes for the case of team 2. The similarity of the ordering was shown in Chapter 5.

6.4 Evaluation Using Selected Attributes

To investigate the effectiveness of the complete model, we examine several alternative approaches to calculating the requirement change impact metric. We used an α-cut of 0.50 to determine the compatibility classes in these examples.

6.4.1 Number of Work Products

Most impact analysis approaches using requirements tracing identify the potentially impacted work products. We calculate the requirement change impact metric for this example to be the number of potentially impacted work products found using requirements tracing. We do this to investigate if the number of potentially impacted work products of a change is an indication of the its degree of impact. Figure 6.4 shows the graph for team 1 and Figure 6.5 shows the graph for team 2.

To transform the graph in Figure 6.4 to the actual compatibility classes requires 6 edge additions, $\{(a,c),\ (a,f),\ (a,d),\ (b,e),\ (c,e),\ (d,e)\}$, and 4 edge deletions, $\{(a,e),\ (b,c),\ (b,f),\ (b,d)\}$. The total distance between the graphs for team 1 is 10, which we consider shows the graphs markedly dissimilar. The case of team 2 requires two edge additions, $\{(a,b),\ (a,d)\}$, for the graph shown in Figure 6.5 to

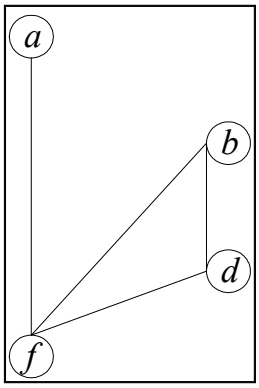

FIGURE 6.5. Team 2 number of work products graph

transform the graph to the actual compatibility classes. This transform makes the distance 2 for the graphs for team 2, a comparison in which the maximum possible distance is 6 for any two graphs with 4 nodes. Thus, predicting the impact using the number of potentially impacted work products is less accurate than the method we define in TIAM.

6.4.2 Effort and Complexity

As another alternative, we redefine the weight of a node as the product of the effort and the complexity attribute value, and the requirement change impact metric as the summation of the weights of the work products in the potentially impacted work product set. The graph for team one is the same as the graph derived from the number of work products as shown in Figure 6.4. The graph for team 2 is shown in Figure 6.6.

Again, the distance between the graphs for team 1 is 10. For team 2, the transform requires a single edge addition, (a, f), making the distance 1.

6.4.3 Conclusions

In the case of team 1, the distance is large between these graphs and the actual impact graphs. The distance between the graphs here for team 2 are not as different but the maximum distance is smaller. In the graphs for team 1, b and e were not in

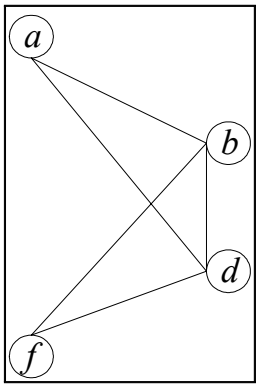

FIGURE 6.6. Team 2 effort and complexity graph

the same class as in the actual impact classes. Thus neither the use of the number of work products nor the use of only effort and complexity improve the results over the method defined in TIAM.

6.5 Conclusion

Table 6.1 shows the distances between the graphs using the approach in TIAM. The distance between the predicted impact classes graphs and the actual impact classes graph shows that TIAM produces classes similar to the actual classes. Simplifying the impact metric to represent only the number of potentially impacted work products or the effort and complexity of potentially impacted work products, produced graphs that were a greater distance from the actual impact classes graphs than the requirement change impact metric defined by TIAM produced. This analysis shows that the inclusion of the influence of requirement traces in TIAM yields a better prediction of the impact that a change may have on a software development project.

TABLE 6.1. Summary of Distances

Attributes Used to Generate Graphs	Distances	
	Team 1	Team 2
Complete impact metric	2	0
Number of work products	10	2
Effort and complexity of work products	10	1
Maximum possible distance	15	6

The distance metric between the graph representations of actual and predictive classes shows that the graphs are similar. We have also shown that the requirement trace influence in the model is key to the effectiveness of TIAM.

Chapter 7
Summary and Conclusions

The objective of this research was to develop a predictive impact analysis technique that identifies classes of requirements changes that have similar impact levels. To achieve this objective, we defined a requirements traceability methodology known as TIAM. We defined WoRM as the model to represent the information required for the methodology, where WoRM consists of the models WIM and RIM. WIM provides a representation of work products and the traces between work products in a software product. RIM provides information about requirement changes introduced to a software development project.

We defined a requirement change impact metric to predict the impact that a requirement change will have on the resources for a project. The requirement change impact metric is based on traces between work products and attributes of work products and traces, information that is contained in WoRM.

We defined algorithms that evaluate the impact of a set of requirement changes, resulting in a set of classes of requirement changes ordered from low to high impact. The results are classes of changes based on their similarity. The similarity between requirement changes is based on a similarity between their respective requirement change impact metrics. We also identify the potentially impacted work products by using traceability.

Using subjects from a software engineering course, we experimentally applied TIAM to the course project to predict the impact of a set of requirement changes for two teams. The teams provided the data that was encapsulated in WoRM, the

data on the number of person hours required to implement each change, and the work products modified for each change.

The potentially impacted work products predicted by TIAM were compared to the actually impacted work products. For one team, only one work product was predicted not to be modified that was actually modified. For the other team, three out of every four work products modified was predicted to be modified. In each case, TIAM identified at least half the work products in the project as work products that were unchanged and predicted not impacted.

Using the actual impact in person hours for each change, we created classes of changes with similar actual impact. These classes were compared to classes generated by TIAM. The actual and predicted classes, when represented as graphs, are shown to be similar using a graph distance measure. The highest impact requirement changes were also discovered by using TIAM.

This research builds on previous research in impact analysis using requirement tracing combined with an approach developed from research using fuzzy logic with respect to requirements satisfaction. Table 7.1, which summarizes trace based impact analysis techniques, extends Table 2.1 by including TIAM. Pfleeger and Bohner's trace based complexity measure provides a quality comparison between the current version of a product and a proposed changed version of the product [Pleeger and Bohner, 1990; Bohner, 1991]. TIAM uses similar traces to determine a quantity comparison between a set proposed changes to be used for resource planning. When a change is made to a work product, Barros, et al, use traces to determine what work products may be impacted. The analysis is performed interactively to enable a developer to completely implement a change [Barros *et al.*, 1995]. TIAM's impact analysis is intended for planning rather than ensuring that changes are thoroughly implemented. Lindvall and Sandahl used traceability based

TABLE 7.1. Comparison of Impact Analysis Approaches including TIAM

Approach	Types of Links	Results
Pfleeger & Bohner	Requirement Traces	Complexity Measures
Barros, et al	Dependency and Requirement Traces	Interactive identificaton of potentially impacted work products
Lindvall & Sandahl	System and Domain Knowledge	Potentially impacted classes
Bianchi, et al	Structural, Naming, and Cognitive	Finer grain work products produce better impact analysis, but at a higher effort
TIAM	Requirement Traces	Classes of changes with similar impact

on the domain knowledge of the system developers [Lindvall and Sandahl, 1998a; Lindvall and Sandahl, 1998b]. Only source code modules were included in the impact analysis. Traceability based on developers knowledge is limited by the availability and memory of the developer, but TIAM's models provide a way of codifying this knowledge and including more than source code work products. Bianchi, et al., included derived traces based the internal structure of work products to accurately identify the impacted work products of a change [Bianchi *et al.*, 2000]. TIAM provides a measure of the degree of impact that a change may have in addition to identifying potentially impacted work products. TIAM presents the results of impact analysis in a manner that clearly distinguishes between the impact of changes. These features increase the information available for decision making about which changes should be incorporated in a software product.

7.1 Contributions

This research makes the following contributions to the state of knowledge of impact analysis and change management:

1. TIAM provides a method to predict which requirement changes present the greatest risks to the project. These changes are ideal candidates for more

detailed impact analysis techniques. The inclusion of changes with higher risk should call for greater care and management when implemented in the existing project. This approach to using requirement tracing yields more useful information than the identification of potentially impacted work products alone.

2. TIAM adds another dimension to impact analysis techniques by determining not only the impact of a change but also providing a comparison of severity among a set of changes. Completing a software development project and shipping a product on schedule and within budget is a complex process. Frequently, decisions on what to include or what not to include in a version of a product must be made with respect to the available resources. This methodology provides a tool to aid in this decision process.
3. TIAM provides an impact analysis technique that is useful during the development of a software product. Since TIAM does not rely on the internal structure of work products, impact analysis may be conducted while many work products are incomplete or not yet created.
4. TIAM can be used for impact analysis for maintenance activities in products provided sufficient requirements traceability information was recorded during development. Failure to update design and other work products when changes are made to source code creates a source of difficulty when performing future maintenance. TIAM accounts for the impact to all work products and thus provides useful analysis for maintenance activities.
5. TIAM provides new models to represent requirements changes and work products information.

6. The application of TIAM is independent of any specific software life cycle model. TIAM is especially suitable as a tool for the risk analysis component of the spiral life cycle model.

TIAM benefits managers by providing assistance for resource planning. The impact metric for each change is an indication of the effort required to make changes in existing work products. For a given change, if the value of the requirement change impact metric is high enough, a determination should be made as to whether the change can be made without affecting the schedule. If further analysis shows that the schedule will be affected, a determination as to how much to change the schedule should be done. If the change is significant enough to invalidate the previous effort estimate to implement the requirement, then the management team should perform another planning exercise for the modified requirement to account for work products yet to be implemented.

For changes that are critical to the success of the product, the methodology can be used to determine which changes require the expense of more detailed impact analysis by revealing those critical changes contained within the higher impact classes. When more detailed impact analysis is required for changes that are in the higher impact classes, TIAM can reduce the expense of the overall impact analysis. The predicted impacted work products should encompass the set of the actually impacted work products, narrowing the number of work products that may require further impacted analysis. In a software development project, it is a more efficient use of the time of domain and system experts to examine the predicted higher impact work products and predicted severe requirement changes than examining all requested requirement changes for potential impact. This method provides for more focused impact analysis by developers and allow the developers to generate better analysis.

If there is flexibility in accepting changes or resources do not allow all changes to be included, management must decide which changes will be included in the product. If the requirement changes have different priorities or values to the client or users, TIAM can be used in conjunction with a process that incorporates the value of including a change to the satisfaction of the product. Project management can use this process to facilitate the determination of requirement changes are candidates not to be included in the current version. To determine whether or not a non-critical change should be included in the current release, management should determine the value of including the change on the overall success of the product in the time frame for the current and following releases. High impact changes with a low value could be candidates for deferring to a following release. Including low impact changes of high value in the current release could increase satisfaction with the product in a cost effective manner.

7.2 Future Work

Next steps will be further experimentation with TIAM in diverse environments. The computation of the impact metric could be refined with the availability of more actual impact data cases. The choice of attribute levels and associated values are likely dependent on specifics of a development organization; thus, experimentation with historical data from an organization is desirable.

One application of TIAM's use of requirement traces is to extend the process to identify interdependent changes for a product. If a set of changes has a significant number of potentially impacted work products in common, effort could be reduced by implementing the set of changes at the same time.

The requirement change impact metric in TIAM potentially could be used to the evaluate the risk of volatile requirements. Volatile requirements are requirements

that are likely to change. There is a risk to the product if such requirements have a strong influence on the design or implementation of the product or on significant interdependent work products. High risk volatile requirements may indicate a need to revisit design and architecture of the product, and they may represent a risk to the stability of the product. TIAM could be extended to determine the risk of volatile requirements to the software product. Design of a product without accounting for such requirements represents a risk to the entire product.

References

[Aho *et al.*, 1974] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, Reading, Mass, 1974.

[Andriole, 1998] S. Andriole. The politics of requirements management. *IEEE Software*, pages 82–84, November/December 1998.

[Antoniol *et al.*, 2000] G. Antoniol, G. Canfora, and A. De Lucia. Identifying the starting impact set of a maintenance request: A case study. *Proceeding of the Conference on Software Maintenance and Reengineering*, pages 227–230, 2000.

[Arnold and Bohner, 1993] R. S. Arnold and S. A. Bohner. Impact analysis - towards a framework for comparison. *Proceedings of the International Conference on Software Maintenance*, pages 292–301, 1993.

[Bach, 1999] J. Bach. Risk and requirements-based testing. *Computer*, pages 113–114, June 1999.

[Barros *et al.*, 1995] S. Barros, Th. Bohdhuin, A. Escudie, J. P. Queille, and J. F. Voidrot. Supporting impact analysis: a semi-automated technique and associated tool. *Proceedings of the International Conference on Software Maintenance*, pages 42–51, 1995.

[Basili, 1990] V. R. Basili. Viewing maintenance as reuse-oriented software development. *IEEE Software*, pages 19–25, January 1990.

[Beck, 1999] K Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, October 1999.

[Bianchi *et al.*, 2000] A. Bianchi, A. R. Fasolino, and G. Visaggio. An exploratory case study of the maintenance effectiveness of traceability models. *Proceeding of the International Workshop on Program Comprehension*, 8:149–158, June 2000.

[Boehm and In, 1996] B. Boehm and H. In. Identifying quality-requirement conflicts. *IEEE Software*, March:25–34, 1996.

[Boehm and Papaccio, 1988] B. W. Boehm and P. N. Papaccio. Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, 14(10):1462–1477, October 1988.

[Boehm, 1981] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1981.

[Boehm, 1988] B. W. Boehm. A spiral model of software development and enhancement. *Computer*, pages 61–72, May 1988.

[Bohner and Arnold, 1996] S. A. Bohner and R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, 1996.

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

[Bohner, 2002] S. A. Bohner. Software change impacts - an evolving perspective. *Proceedings of the International Conference on Software Maintenance*, pages 263–272, October 2002.

[Bondy and Murty, 1976] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. Elsevier Science Publishing Company, Inc., New York, N.Y., 1976.

[Brooks, 1987] F. P. Brooks. No silver bullet. *Computer*, 20(4):10–19, April 1987.

[Brooks, 1995] F. P. Brooks. *The Mythical Man-Month, Anniversary Edition*. Addison Wesley Longman, Inc, Reading, Massachusetts, 1995.

[Chartrand *et al.*, 1998] G. Chartrand, G. Kubicki, and M. Schultz. Graph similarity and distance in graphs. *Aequationes Math.*, 55:129–145, 1998.

[Chen *et al.*, 1996] X. Chen, W. Tsai, H. Hunag, M. Poonawala, S. Royadurgam, and Y. Wang. Omega - an integrated environment for C++ program maintenance. *Proceedings of the International Conference on Software Maintenance*, pages 114–123, 1996.

[Cimitile *et al.*, 1999] A Cimitile, A. R. Fasolino, and G. Visaggio. A software model for impact analysis: A validation experiment. *Proceeding of the Sixth Working Conference on Reverse Engineering*, pages 212–222, 1999.

[Cusumana and Selby, 1997] M. A. Cusumana and R. W. Selby. How microsoft builds software. *Communications of the ACM*, 40(6):53–61, June 1997.

[Gallagher and Lyle, 1991] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[Garg and Scacchi, 1990] P. K. Garg and W. Scacchi. A hypertext system to manage software life-cycle documents. *IEEE Software*, 7(3):90–98, May 1990.

[Glass, 1998] R. L. Glass. *Software Runaways*. Prentice Hall, Inc., Upper Saddle River, New Jersey, 1998.

[Goddard and Swart, 1996] W. Goddard and H. Swart. Distances between graphs under edge operations. *Discrete Mathematics*, 161:121–132, 1996.

[Gotel and Finkelstein, 1994] O. Gotel and A. Finkelstein. An analysis of the requirements traceability problem. *Proceedings of the First International Conference on Requirements Engineering*, pages 94–101, 1994.

[Gotel and Finkelstein, 1997] O. Gotel and A. Finkelstein. Extended requirements traceability: Results of an industrial case study. *Proceedings of the Third IEEE International Symposium on Requirements Engineering*, pages 169–178, January 1997.

[Henderson-Sellers and Edwards, 1990] B. Henderson-Sellers and J. M. Edwards. The object-oriented systems life cycle. *Communications of the ACM*, 33(9):142–159, September 1990.

[Horowitz and Sahni, 1978] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, Rockville, Maryland, 1978.

[Horowitz and Williamson, 1986] E. Horowitz and R. C. Williamson. Sodos: A software documentation support environment - its definition. *IEEE Transactions on Software Engineering*, 12(8):849–859, August 1986.

[Horwitz *et al.*, 1990] S. Horwitz, T. Reps, and D. Brinkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[IBM Object-Oriented Technology Center, 1997] IBM Object-Oriented Technology Center. *Developing Object-Oriented Software: An Experience-Based Approach*. Prentice-Hall, Inc, Upper Saddle River, New Jersey, 1997.

[Jarke, 1998] M. Jarke. Requirements tracing. *Communications of the ACM*, 41(12):32–36, December 1998.

[Jones, 1995] C. Jones. Patterns of large software systems: Failure and success. *Computer*, pages 86–87, March 1995.

[Jung, 1998] H. Jung. Optimizing value and cost in requirements analysis. *IEEE Software*, pages 74–78, July/August 1998.

[Juristo *et al.*, 2002] N. Juristo, A. M. Moreno, and A. Silva. Is the European industry moving toward solving requirements engineering problems? *IEEE Software*, pages 70–77, November/December 2002.

[Karlsson and Ryan, 1997] J. Karlsson and K. Ryan. A cost-value approach for prioritizing requirements. *IEEE Software*, pages 67–74, September/October 1997.

[Keil *et al.*, 1998] M. Keil, P. E. Cule, K. Lyytinen, and R. C. Schmidt. A framework for identifying software project risk. *Communications of the ACM*, 41(11):76–83, November 1998.

[Klir and Yuan, 1995] G. J. Klir and B. Yuan. *Fuzzy Sets and Fuzzy Logic: Theory and Applications*. Prentice-Hall, Inc, Upper Saddle River, New Jersey, 1995.

[Klir *et al.*, 1997] G. J. Klir, U. T. St.Clair, and B. Yuan. *Fuzzy Set Theory: Foundations and Applications*. Prentice-Hall, Inc, Upper Saddle River, New Jersey, 1997.

[Kotonya and Somerville, 1998] G. Kotonya and I. Somerville. *Requirements Engineering: Processes and Techniques*. John Wiley & Sons Ltd, Chichester, West Sussex, England, 1998.

[Lam and Shankararaman, 1999] W. Lam and V. Shankararaman. Requirements change: A dissection of management issues. *Proceedings of the Euromicro Conference*, pages 2244–2251, 1999.

[Lindvall and Sandahl, 1998a] M. Lindvall and K. Sandahl. How well do experienced software developer predict software change? *The Journal of Systems and Software*, 43:19–27, 1998.

[Lindvall and Sandahl, 1998b] M. Lindvall and K. Sandahl. Traceability aspects of impact analysis in object-oriented systems. *Software Maintenance: Research and Practice*, 10:37–57, 1998.

[Loyall and Mathesen, 1993] J. P. Loyall and S. A. Mathesen. Using dependence analysis to support the software maintenance process. *Proceedings of the International Conference on Software Maintenance*, pages 282–291, 1993.

[Maciaszek, 2001] L. A. Maciaszek. *Requirements Analysis and System Design*. Person Education Limited, Harlow, England, 2001.

[Nuseibeh and Easterbrook, 2000] B. Nuseibeh and S. Easterbrook. Requirements engineering: A roadmap. *Proceeding of the conference on the future of Software Engineering*, pages 35–46, 2000.

[Olphert *et al.*, 1994] C. W. Olphert, Harker S. D. P., and K. D. Eason. Evaluating requirements for future information technology systems: A socio technical approach. *IRMA Conference Proceedings*, pages 263–264, 1994.

[O'Neal and Carver, 2001] J. S. O'Neal and D. L. Carver. Analyzing the impact of changing requirements. *Proceedings of the International Conference on Software Maintenance*, pages 190–195, November 2001.

[Papadopoulos and Manolopoulos, 1999] A. N. Papadopoulos and Y. Manolopoulos. Structure-based similarity search with graph histograms. *Proceedings of the International DEXA Workshop on Similarity Search*, pages 174–178, 1999.

[Pfleeger, 1998] S. L. Pfleeger. *Software Engineering: Theory and Practice*. Prentice-Hall, Inc, Upper Saddle River, New Jersey, 1998.

[Pleeger and Bohner, 1990] S. L. Pleeger and S. A. Bohner. A framework for software maintenance metrics. *Proceedings of Conference on Software Maintenance*, pages 320–327, November 1990.

[Podgurski and Clarke, 1990] A. Podgurski and L. A. Clarke. A formal model of program dependencies and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.

[Pohl, 1996] K. Pohl. *Requirement Engineering Processes.* Research Studies Press Ltd., Taunton, Somerset, England, 1996.

[Queille and Voidrot, 1994] J. P. Queille and J. P. Voidrot. The impact analysis task in software maintenance: A model and a case study. *Proceedings of the International Conference on Software Maintenance*, September:234–242, 1994.

[Ramesh *et al.*, 1995] B. Ramesh, T. Powers, C. Stubbs, and M. Edwards. Implementing requirements traceability: A case study. *Proceedings of the International Symposium on Requirements Engineering*, pages 89–95, 1995.

[Royce, 1970] W. W. Royce. Managing the development of large software systems: Concepts and techniques. *1970 WESCON Technical Papers, Western Electric Show and Convention*, pages A/1–1 – A/1–9, August 1970.

[Schach, 1999] S. R. Schach. *Object-Oriented and Classical Software Engineering.* WCB/McGraw-Hill, 1999.

[Siddiqi and Shekaran, 1996] J. Siddiqi and M. C. Shekaran. Requirements engineering: The emerging wisdom. *IEEE Software*, pages 15–19, March 1996.

[Spanoudakis, 2002] G. Spanoudakis. Plausible and adaptive requirement traceability structures. *Proceedings of the conference on Software Engineering and Knowledge Engineering*, pages 135–142, 2002.

[Strens and Sugden, 1996] M. R. Strens and R. C. Sugden. Change analysis: A step toward meeting the challenge of changing requirements. *Proceedings of the IEEE Symposium and Workshop on Engineering of Computer-Based Systems*, March:278–283, 1996.

[Sugden and Strens, 1996] R. C. Sugden and M. R. Strens. Strategies, tactics, and methods for handling change. *Proceedings of the IEEE Symposium and Workshop on Engineering of Computer-Based Systems*, March:457–463, 1996.

[Turver and Munro, 1996] R. J. Turver and M. Munro. An early impact analysis technique for software maintenance. *Software Maintenance: Research and Practice*, 6:35–52, 1996.

[von Knethen, 2002] A. von Knethen. Change-oriented requirements traceability: Support for evolution of embedded systems. *Proceedings of the International Conference on Software Maintenance*, pages 482–485, 2002.

[Weiser, 1984] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(7):352–357, 1984.

[Williams *et al.*, 1997] R. C. Williams, J. A. Walker, and A. J. Dorofee. Putting risk management into practice. *IEEE Software*, pages 75–82, May/June 1997.

[Yau, 1988] S. S. Yau. An integrated life-cycle model for software maintenance. *IEEE Transactions on Software Engineering*, 14(8), August 1988.

[Yen and Tiao, 1997] J. Yen and W. A. Tiao. Impact analysis of design alternatives on imprecise requirements. *Proceedings of the Sixth IEEE International Conference on Fuzzy Systems*, pages 1215–1220, 1997.

[Yen *et al.*, 1996] J. Yen, W. A. Tiao, and X. F. Liu. A systematic tradeoff methodology for acquiring and validating imprecise requirements. *Proceedings of the Fifth IEEE International Conference on Fuzzy Systems*, pages 349–354, 1996.

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

Appendix A
Case Study Problem Description

Fourteenth-Floor Software (FFS) is a medium-sized computer software and consulting company with expertise in object-oriented software development and detailed domain knowledge of the telecommunications industry. FFS assists major companies with application development by providing an experienced software development team on a contract basis. FFS also markets parts of its suite of development tools and specialized applications. Currently, FFS has over five hundred employees and twenty active clients.

FFS charges clients for the time and expenses that its employees spend providing services to the client. FFS also has several internal projects that employees work on which will become products that the company sells. Employees responsible for marketing and selling FFS's products are eligible to earn a commission on the sales they make.

FFS wishes to have a computer system that allows employees to report the hours worked, expenses, and to whom the time and expenses should be charged. Charges are made to accounts that represent a client contract, an internal project, or one of five pseudo-accounts (Training, Sales, Holiday, Vacation, and Leave). The system should gather and produce information to print payroll checks and related reports, billing information for clients, and reports for management on charges to accounts.

Currently all employees and contractors that are charging time and expenses to an account fill out a weekly time and expense sheet. Processing the sheets takes too much time and creates delays in billing. Paychecks are one pay period behind.

FFS wants to print paychecks one day after the pay period closes and mail invoices to clients as early as possible.

There are four levels of employees in FFS: junior, associate, senior, and partner. Junior level employees are the only hourly paid (nonexempt) employees. All other employees are salaried (exempt). Independent contractors may also be used by the company. Contractors are also paid hourly. Employees in sales earn a commission on sales. These employees may choose one of two methods of having their pay calculated:

1. Monthly base pay + commissions, where commissions are capped at 50% of base pay.

2. 50% of monthly base pay + commissions.

Time is charged to accounts for outside clients or internal projects. Time may also be charged to accounts for training and sales. Account types of Holiday and of Leave should also exist. Employees are paid for holidays but time charged to leave should not be paid to hourly paid employees and should be deducted from salaried employees at a rate of (yearly salary/2000) per hour, expect for employees that are at the partner level.

Junior level employees and contractors are required to provide actual work times and accounts to be charged for that time. Junior level employees and contractors must charge 100% of their time. These entries must be approved by their manager before processing in payroll. All other employees are required to enter the account and the number of hours and/or expenses to be charged for that account. Entries should immediately be validated for details such as number of hours worked and vacation time taken before the entry is accepted into the system.

	Billing Rate per hour	Commission Rate	Vacation Days per Year	Pay Period
Junior	$50	N/A	10	biweekly
Associate	$100	10%	15	monthly
Senior	$200	12.5%	20	monthly
Partner	$300	15%	N/A	monthly
Contractor	125% of actual pay rate	N/A	N/A	biweekly

A limit of 96 hours can be worked in one week. Vacation can be taken in four hour blocks (half days). Note: it would not make sense to have more than 40 hours of vacation or leave in one week.

FFS usually designates twelve holidays per year. Holidays may be taken at anytime. An employee beginning employment by March 31 is eligible for nine holidays. An employee beginning employment after March 31 but by June 30 is eligible for six holidays. An employee beginning employment after June 30 but by September 31 is eligible for three holidays. Any employee beginning employment after September 31 is ineligible for holidays that year.

The following table gives information on each employee level and contractors.

Accounting will manage the system and is not particular about the user interface for the functions that they will use. Employees and contractors should have a web enabled front end for making their time and expense entries. All of the company's systems reside on a private intranet and should be relatively secure. Reasonable security should be provided to ensure that each individual uses the system only for entering their own information. Additionally an audit trail is required to trace employee and contractor entries and manager approval.

Accounts for outside clients and internal projects may be in one of three states: active, closed, or limited. Active accounts may be charged any amount. Closed accounts may not have any charges applied to it. A limited account has a ceiling

Benefit Package	Employee Monthly Cost	Employee Bi-weekly Cost	Employer Monthly Contribution
A	$125	$57.50	$135
B	$167	$76.80	$135
C	$108	$49.60	$135

on the monthly cost or hours that may be applied to the account. The invoice for a client account that is limited may not exceed the ceiling. If the ceiling amounts are surpassed on any account, a detailed exception report for management should be automatically generated during normal account processing.

Employees or contractors who leave the company must remain in the system until year end tax forms are printed. After that point their information should be archived in some manner for future reference.

Employees may choose one of three benefit packages or no benefit packages at all. The company also offers a retirement plan in which the employee may deposit up to 8% of base pay with the company matching up to 4%. This plan is also optional and employees must have been employed for at least one year to enroll. The premiums for the benefit packages are deducted before taxes, where the retirement plan is deducted after taxes.

Taxes that are deducted from employee's pay are Federal income taxes, Social Security, and Medicare. FFS is based in a state and municipality that does not have income tax; however, this may not always be the case. The Social Security tax is 6.2% and Medicare tax is 1.45%. Only the first $76,200 of income per year is subject to the Social Security tax. Federal income tax is withheld according to the attached information from IRS Publication 15, Circular E, Employer's Tax Guide.

Note: Contractors are not employees and are not eligible for any holidays, benefits, retirement plans, and have no taxes withheld from pay.

The system should be flexible in allowing for changes in taxes and benefit plans.

At year end, all employee information that is shown in tax forms should be saved along with total retirement plan deposits for each employee.

Accounting will need to create and update information on employees, contractors, and accounts for internal projects and outside client contracts. Accounting also will enter sales figures for employees making sales.

Paychecks for all hourly paid employees and contractors are to be printed at the end of every bi-week (two week) period. All other paychecks are to be printed at the end of each month.

Each paycheck should include an attachment with the employee's name and social security number, and listing current and year-to-date totals for the following: gross pay, itemized deductions (taxes, benefits, etc.), total deductions, and net pay.

Monthly invoices are to be generated for each outside client account. The client is billed for the time charged to the associated account and incurred expenses. A detailed invoice listing time and expenses by person should be available at the client's request. Every client account should have a contact person to whom the invoice is sent.

Management wants a monthly report totaling hours and cost by internal project. A detailed report for each project listing by person should be available if requested.

A monthly report for management should be printed with totals for social security taxes withheld, medicare taxes withheld, federal income tax withheld, employee contribution and employer contribution for: benefit package A, benefit package B, benefit package C, and retirement plan.

Yearly, tax forms for employees and contractors will need to be printed. For employees, W-2 reports are needed for each employee listing employee name, address, and social security number, total taxable pay, total social security tax withheld,

total medicare tax withheld, and total federal income tax withheld. Contractors require a 1099 form which lists the contractor name, address, social security number, and total amount paid.

Social security numbers may not be used to reference employees or contractors. Everyone who works for FFS is issued a serial number when hired. The serial number is a letter followed by five digits. The serial number is used to identify workers in computer systems such as security badge readers and current accounting systems.

Appendix B
Case Study Requirement Changes

Due to changing business needs, the following changes are to be made to the system requested by Fourteenth-Floor Software. All work products (requirements, design, etc.) are required to be updated to reflect these changes.

Change A: Contractors are to be paid weekly.

Change B: Junior level employees may work as sales trainees, earning 5% commission up to 80 times their hourly rate per month. This is the only commission option for junior level employees.

Change C: Client's accounts may be set up to be sent detailed invoices automatically if the client so chooses.

Change D: The number of vacation days for each employee is based on level or length of service (time employed), whichever is greater.

Years of Service	Vacation Days
<5	10
5 to <10	15
10 or more	20

Change E: A manager will have another manager designated to approve hourly workers' time/expense entries if he or she is not available.

Change F: For accounts that are limited, the detailed exception report should be generated if the amounts reach 90% of the ceiling.

Vita

James Steven O'Neal finished his undergraduate studies at Mississippi College in December 1986 and graduated May 1987. During his undergraduate years, he worked for a subsidiary of General Motors developing local applications to support manufacturing and engineering processes. He earned a Master of Science degree in computer science from Clemson University in December 1989. For the next six years, he worked for International Business Machines in North Carolina. There, he worked as a software developer at the former Cary Programming Laboratory and as an object-oriented software development mentor and researcher while attached to the Research Triangle Park Programming Laboratory. He returned to academic life at Louisiana State University in Baton Rouge to pursue a doctorate. He is currently teaching computer science at Mississippi College in Clinton, Mississippi. The degree of Doctor of Philosophy will be conferred at the December 2003 commencement.