# *Early detection of ripple propagation in evolving software systems*

Richard john Turner

# Early Detection of Ripple Propagation in Evolving Software Systems

**Richard John Turver M.Sc. (Dunelm)**

Thesis submitted for the requirements of the degree of Doctor of Philosophy

School of Engineering and Computer Science
Faculty of Science
University of Durham

**1993**

## Abstract

*Ripple effect analysis is the analysis of the consequential knock on effects of a change to a software system. In the first part of this study, ripple effect analysis methods are classified into several categories based on the types of information the methods analyse and produce. A comparative and analytical study of methods from these categories was performed in an attempt to assist maintainers in the selection of ripple effect analysis methods for use in different phases of the software maintenance process. It was observed that existing methods are most usable in the later stages of the software maintenance process and not at an early stage when strategic decisions concerning project scheduling are made.*

*The second part of the work, addresses itself to the problem of tracing the ripple effect of a change, at a stage earlier in the maintenance process than existing ripple effect analysis methods allow. Particular emphasis is placed upon the development of ripple effect analysis methods for analysing system documentation. The ripple effect analysis methods described in this thesis involve manipulating a novel graph theory model called a Ripple Propagation Graph. The model is based on the thematic structure of documentation, previous release information and expert judgement concerning potential ripple effects.*

*In the third part of the study the Ripple Propagation Graph model and the analysis methods are applied and evaluated, using examples of documentation structure and a major case study.*

## Acknowledgements

This work was supported by a CASE studentship from the Science and Engineering Research Council and TSB Bank plc.

I am grateful to my supervisor Malcolm Munro for his encouragement and guidance throughout this study. I am grateful to Professor K.H. Bennett for the facilities provided. I am grateful to my parents who have encouraged me throughout this venture.

I would also like to thank the following staff associated with TSB Bank plc Retail Banking Technology.

Mr B. Edisbury for his support and provision of help in developing the cooperation between myself and TSB Bank plc and for organising meetings which enabled me to rapidly establish contact with TSB staff. Mr J. Rudolph the Head of Technology Research, for discussing current research interests and the development and maintenance of the TSB core banking system. Mr R. Broughton the Head of Implementation and Testing, for the provision of useful information and helping me refine the project focus. Mr S. Temple and Mr C. Evans for their time, helpful discussions and for selecting a suitable case study for the evaluation stage of the project. The author acknowledges with gratitude the permission of TSB Bank plc Retail Banking Technology to use their data to support the thesis.

I would also like to thank members of the Centre for Software Maintenance for invaluable discussions during the last three years.

This thesis has been produced using the LaTeXtext formatting system.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction and Aims

## 1.1 The Problem and Its Context

The accurate estimation of the resources required to implement a change in software is not a straight forward task. A method for achieving this should include the analysis of the impact of the change on the existing system. A number of methods for analysing the impact of a change on the source code have been described in the literature. Although these methods provide a good example of how to apply ripple effect analysis to source code, a weakness is that they can be difficult to apply in the early stage of a maintenance project. This is because the relationship between the change proposal and the source code is often not very well understood at this stage, since change proposals are written at a much higher level of abstraction than the code. It is therefore often the case, that in practice subjective impact analysis methods are used for project investment appraisal. The under-estimated resources for dealing with the **ripple effects**[1] of a change can result in project schedules becoming so tight that only the minimum quality is achieved.

Software often needs changing to reflect changes in real world situations. These changes create the need for new software products and changes to existing software products. Changes to software can be problematic since internal coupling, interconnections and dependencies which superficially

[1]In this thesis, words in bold typeface are defined in the glossary given in Appendix B

appear localised may even cause changes to impact the code elsewhere in a program. Thus when changes are made to the code, deviations from absolute correctness will occur and unexpected side effects will appear. These errors are likely to necessitate further corrective action [53].

In a study by Collofello and Buck [19] it was observed that more than 50% of errors were introduced by previous changes. In the article "Kill that Code!", [90] Weinberg describes his private list of the world's most expensive program errors. The top three disasters were caused by a change to exactly one line of code: "each one involved the change of a single digit in a previously correct program." Since the change was only to one line the usual mechanisms for change control were not used and the results were disastrous. Weinberg offers a partial explanation: "unexpected linkages," i.e., the value of a modified variable was used in some other place in the program. Changes with unexpected linkages annoy users and consume precious personnel time and resources [59]. Therefore impact analysis is an important research topic. In particular research investigating the detection of ripple effects at the beginning of a maintenance project has significant strategic importance.

The assessment of the impact of a proposed change on an existing system at the beginning of a maintenance project is an important aspect of the software maintenance process [12]. However it is generally not practised at an early stage in the maintenance process. There are strong economic arguments for planning the contents of software releases based on impact analysis information in addition to planning a release using the business justification for a project. However the contents of releases are often customer driven only.

Given that impact analysis is accepted as a useful maintenance tool and that it has clear economic benefit, then it may be asked why is it not practised. The reason why impact analysis is the exception rather than the rule is technical. Most impact analysis methods are source code based which implies that the source code is well understood when these methods are applied. This is not the case at the start of a maintenance project. Therefore if impact analysis methods are to be used at this early stage they must address the part of the software which is in a level of language similar to that of the change proposal, such as documentation.

The principle of this work is that the impact of a change can be detected at an earlier stage in the maintenance process than existing methods allow. This thesis addresses this issue.

## 1.2 Research Method and Objectives

The research method used in this thesis is based on the engineering research method which consists of, problem, theory, design, prototype, evaluation, feedback and iterative refinements [8]. Other approaches considered were the mathematical approach or experimental approach. However a mathematical approach is more suited to investigations into formal methods such as program correctness, formal specification or computability. The experimental approach is also not particularly suitable for this investigation as it would involve data collection from real maintenance projects over a number of years.

The main objective and the research question addressed are:

*How can the impact of a change be detected at an earlier stage in the maintenance process than existing methods allow?*

As the investigation progressed and a solution was developed, other smaller research questions presented themselves. These were the following :

1. What information is available to management to make decisions about change proposal potential impacts at an early stage in the maintenance process?
2. What information, produced as a by-product of the software maintenance process, can be re-used from previous maintenance projects to help in tracing the impact of a proposed change, without increasing the costs of developing new software?
3. How can the thematic structure of a documentation system and system release information be used to understand the impact of a change?
4. How can the thematic structure of a documentation system and system release information be modelled with graph theory?
5. How can such a model be created and analysed objectively?

## 1.3 The Criteria for Success

The criteria for the success of the investigation are the following :

1. The production of an interconnection graph usable by management.
2. The ability to be able to trace user change requirements through to operational software.
3. The production of a method for building and maintaining the graph.
4. The provision of impact information using the graph slicing methods.
5. The evaluation of the information produced by the model with respect to other impact analysis methods.

## 1.4 The Limitations of the Approach

Although the approach is a general approach, based on neither a proprietary documentation method nor on a particular documentation standard, there are limits to the situations to which the approach should be applicable. These limits can be stated as follows:

1. Domain: The approach has been developed in the context of commercial data processing systems. The approach cannot be used to model documentation which describes real time software systems.
2. Documentation Style: The approach has not been evaluated in the context of graphical notations. Instead the approach addresses written documentation.

## 1.5 Author's Contribution

The contribution to knowledge of this work is a method for the early detection of the impact of a change. In particular, the method is based on the analysis of a new model of interconnectivity within documentation and source code [89]. The novelty of the model is that it attempts to capture abstract system properties such as thematic structure of documentation, expert judgement concerning potential ripple effects and software release information. These extra features lead to the development of a set of model analysis methods which help in tracing the ripple effect.

## 1.6 Organisation of Thesis

Chapter 2 introduces the software engineering context of the work and develops a new classification scheme for impact analysis methods. Two important observations were made at this stage. Existing impact analysis methods are only applicable near the end of a maintenance project and few people have explored the early detection of the impact. Chapter 3 analyses the models of system interconnectivity upon which existing impact analysis methods are based. The evidence suggests that existing models of interconnectivity are not suitable for early detection of the impact as it is difficult to link source code components with requirements documents. This chapter substantiates the observations made in the previous chapter.

Chapter 4 argues that a model can be devised and also develops the requirements for such a new model of software system structure. The philosophy of the model is introduced before formally defining the model. Chapter 5 describes the methods developed for constructing the graph model. Chapter 6 formally defines how the model can be analysed to derive impact information.

Chapter 7 describes a prototype implementation of a software tool called MAGENTA (MAnaGemENt Technical impact Analyser) to support ripple effect analysis, that is, the recording of structural information about software and the analysis of this information. Chapter 8 describes the application of the model and model analysis methods, to examples of software system structure and also to a major case study. In particular, the use of the information produced by the method is discussed.

Chapter 9 discusses the results of the application of the model and model analysis methods and determines the worth of these impact analysis methods. In particular it investigates the extent to which the information produced by the method can be trusted. The strengths and weaknesses of the method are discussed and the method is compared and contrasted with other methods. Chapter 10 provides a summary of the investigations. The objectives which have been achieved process and the wider field.

Finally the appendices contain, firstly a glossary of mathematical notation and secondly a glossary of terminology used within this thesis. The third appendix describes the use of the prototype developed to test the ideas which evolved within this work, followed by a bibliography.

# Chapter 2

# The Software Maintenance Process

## 2.1 Introduction

In this chapter the subject of the thesis is placed in its software engineering context. The notion of software engineering process and software maintenance process are discussed. The reasons for the importance of impact analysis within the software maintenance process are argued. A detailed examination of the decisions made within the software maintenance process is presented. The results are used for developing a classification scheme into which existing impact analysis techniques can be placed. The classification scheme sorts existing impact analysis techniques according to how early in the software maintenance process the impact analysis techniques can be used to produce impact information.

## 2.2 The Software Engineering Context

Computer users first became aware of a software crisis 20 years ago. Many software projects were being delivered far behind the planned schedule, quality was poor and maintenance was expensive. As more complex applications were found, programmers fell further behind than the demand for new and modified software, a backlog of application development was created and their results were of poorer quality. The large backlog of application development, enhancement and low productivity increased the cost of producing systems. The cost of software development and maintenance is still growing [20]. During the eighties considerable attention was aimed at software tools. Software systems are large, commonly occurring and highly coupled to business organisational systems. The requirement for a systematic approach to software development and software maintenance has become increasingly important. Software Engineering is the field of study concerned with this emerging technology [29]. The techniques of software engineering have been introduced in an attempt to counteract these cost trends in the computer industry. Boehm defines software engineering in the following way [11] :

> *Software engineering involves the practical application of scientific knowledge to the design and construction of computer programs and the associated documentation required to develop, operate and maintain them.*

The IEEE *Standard Glossary of Software Engineering terminology* [46] defines software engineering as:

> *The systematic approach to the development, operation, maintenance and retirement of software.*

Both of these definitions suggest that methods, procedures, rules and principles are used in software engineering. In this thesis software is defined as computer programs, procedures, rules, and possibly any associated documentation and data pertaining to the operation of a computer system. Once software systems have been installed they are often changed to reflect changes in other sub systems with which they are connected [88]. Software maintenance has been defined by the IEEE [46] as:

> *The modification of a software product after delivery, to correct faults, to improve performance or other attributes or to adapt the product to changed environment.*

Over the total life of software the software maintenance effort has been estimated to be frequently more than 50% of the life cycle costs [55]. An update survey indicates the maintenance cost shows no sign of declining [66]. Software Maintenance can be classified into four categories [7] :

1. Perfective: changing the functional behaviour of a software system to reflect a revised set of system requirements.
2. Adaptive: changing the software in response to a change in the data environment or in the processing environment.
3. Corrective: diagnosing system incidents such as errors which cause the termination or incorrect behaviour of the software.
4. Preventive: improving the quality of a software system in areas where future maintenance is anticipated.

There are many factors involved in the high cost of software maintenance [66]. One of the reasons for this high cost of maintenance is the way in which requests for change are processed [4]. If change requests are serviced in the order in which they are made then extra costs may be involved. This is because some changes may overlap with others, resulting in duplication of work and increased complexity. Costs can be reduced by scheduling change requests and batching similar requests together. Detailed analysis of the system is required in order to determine the effect of each change on other programs and documentation. A software system should not be considered only in terms of its source code, it consists of many other related items such as specification and design documentation. Often a change may have system wide ramifications which are not obvious. When considering a change to the source code of a system it is important to assess the impact of that change not only on the source code but on the other elements of the system. In this thesis impact analysis is defined as :

*The analysis of the consequential effects on other parts of the system resulting from a change to a system where a system consists of all source code entities and documentation entities.*

In order to improve the maintenance process, by providing impact information at an early stage in the process, it is necessary to analyse and understand the process. This chapter presents a new model of the maintenance process for the purpose of evaluating existing impact analysis techniques. In particular the model is used to demonstrate the stages in the maintenance process when impact analysis techniques can be applied and when they cannot. After a brief reprise of software process modelling and a short introduction to software maintenance modelling, the definition of the new model follows. It is necessary to introduce process modelling and existing software maintenance models in order to highlight the difference between the model developed in this thesis and other models. The model developed in this thesis is only a model of the information required to make decisions during the software maintenance process and not a process model.

Existing models of maintenance are used as a foundation for the new model. In recent years the software engineering research community has been focusing attention on the software processes used to develop and maintain software. Dowson and Wildeden [26] define a software process as :

*A set of related activities, seen as a coherent process subject to reasoning, involved in the production of a system.*

A process can be regarded as a systematic approach to the creation of a product or the accomplishment of some particular task. A set of instructions to accomplish a task is a process. This set of instructions is called a process description. A process description is a specification for how the task should be carried out. The process description purports to model the behaviour of an instantiation of the process. Therefore a software process description may be regarded as a software process model [33].

Hinley and Bennett [43] argue that process models need to have the following characteristics in order to provide real benefits for maintenance management :

1. The model must contain organisational, behavioural and functional aspects of the maintenance process.
2. Real world objects such as documentation, source code and change requests must be recognised.
3. The roles of the people involved in the maintenance process must be recognised.
4. The model must be represented in diagrammatic form so that measurement and control points can be identified.
5. The model must also recognise how communication between people involved in the maintenance process is achieved.

6. A framework for maintenance managers which provides guidance in the use of the model must also be included.

7. The model must be capable of being easily changed in order to reflect both changes in working practice or experimental changes to the model itself.

Imposing a structure of work within the maintenance process may reduce the difficulty of the whole task by decomposing it into a number of sub tasks of reduced complexity [15]. Kellner [52] identifies process understanding and analysis of processes as one of the objectives of process modelling. Examples of maintenance models are shown in table 2.1. The models developed between 1976 and 1983 are very simplistic and the order of the phases are indicated. However these models present no descriptive details of how tasks are to be conducted. The models developed between 1984 and 1987 provide further details of how to perform maintenance tasks. The different categories of maintenance tasks are also addressed. The models developed between 1988 and 1991 have emphasis on change control, measurement and different organisational perspectives on maintenance. Most of the previous models of the maintenance process which have led to publication are very general and do not include identification of decision points where impact analysis information is useful. The models above the double horizontal line shown in the table 2.1 focus on technical issues whereas the models below the line focus on management issues.

| Maintenance Model | Year Published | Bibliographic Entry |
|---|---|---|
| Boehm | 1976 | [11] |
| Liu | 1976 | [56] |
| Swanson | 1976 | [84] |
| Sharpley | 1977 | [77] |
| Yau and Collofello | 1978 | [95] |
| Parikh | 1982 | [68] |
| Martin and McClure | 1983 | [59] |
| Patkau | 1983 | [69] |
| Osborne | 1987 | [67] |
| Arthur | 1988 | [4] |
| Foster | 1989 | [32] |
| Pfleeger and Bohner | 1990 | [72] |
| Bennett | 1991 | [9] |

Table 2.1: Existing Maintenance Models

It is clear from the evidence presented in Chapter 1 concerning unexpected linkages that existing impact analysis techniques are not providing accurate information about the impacts of changes at an early stage in maintenance projects. One of the software maintenance models most relevant to this thesis was developed by Arthur [4]. This model is relevant to the model developed in this Chapter because it focuses on management issues to optimise maintenance process and use available resources in an effective way, rather than concentrating on purely technical issues. The model represents the software maintenance from the release planning perspective, and impact analysis is implicit within this model. The model is as follows :

1. Rank change requests into a priority order;
2. Select highest priority changes that can be made with available resources;
3. Secure agreement on the content and timing of system releases;
4. Obtain approval to implement the changes;
5. Schedule work into groups to maximise productivity;
6. Prepare release information.

The objectives of release planning are to establish a schedule of system releases and to determine the contents of each release. It is important to rank the changes in some order of priority so that, for example, those changes a yielding the greatest profits to an organisation are implemented first. The ranking of changes is an important activity which has an impact on the profitability of a system. However in practice subjective ranking methods are the norm [31]. The software maintenance process can be optimised by the use of release planning techniques. Examples of such optimisations are a reduction in both project costs and introduced defects, by scheduling and batching of releases, based on change impact or impact information. The cost of the software maintenance process can be reduced if precise and unambiguous information is available concerning the potential impacts of a change on an existing system. This information can be provided with impact analysis techniques. The earlier the information is produced in a maintenance project the greater will be the reduction in cost.

## 2.3 A Change Analysis View of Software Maintenance

In order to judge at which stage of a project a particular impact analysis technique can be used, it is necessary to identify where change information is required within the software maintenance process, also to what extent a particular impact analysis technique can provide this change information. Change analysis is the decomposition of a change in a software system into the different facets of the change such as costs benefit analysis, impact analysis, testing planning, project priority analysis. The maintenance model developed in this thesis is not a process model but a model to achieve the following :

1. To identify the decisions made during the change analysis process.
2. To identify the information needed to support the change analysis process.

The model does not include the roles of maintenance staff, nor the behavioural and organisational aspects of the maintenance process which a process model would. In order to understand the impact analysis process, it is necessary to understand the information upon which this process relies and in particular how the information is collected, derived and disseminated. The decision making process must be clearly understood. The making of a decision means the designing and committing to a strategy, to irrevocably allocate valuable maintenance resources.

The scope of the model includes the change proposal evaluation phase to the post implementation review activity. Some phases are identified but are not necessarily refined if they are not considered pertinent to the change analysis process. Prototyping and packages are not considered in this model as these approaches largely address the development of new software and not the evolution of old systems.

The model in this thesis is broken into six phases in order to investigate decision making processes and in particular the assessment of the impact of a change. The model is based on a consolidation of existing maintenance models and some of the concepts extracted from the TSB Bank plc development method.

Each phase is separated into states and tasks. Existing maintenance models are general and high level. A model whose granularity is fine will allow a more detailed analysis of the process being modelled. The model developed in this thesis consists of three levels.

1. The *phase level,* which represents the phases which a project will pass through before its completion.
2. The *state level,* which represents the states in which a project may be at any given time.
3. The *task level,* which represents the tasks which must be completed in order to reach a particular state. Included at this level are the project deliverables and the decisions made. Two types of decisions are recorded management decisions and technical decisions. The decisions are not impact analysis decisions but are change analysis decisions which often make use of impact analysis information.

The phases represented by this model are:

**Phase 1. CHANGE PROPOSAL EVALUATION.**
**Phase 2. CHANGE DESIGN.**
**Phase 3. CHANGE IMPLEMENTATION.**
**Phase 4. SYSTEM TESTING.**
**Phase 5. SYSTEM HANDOVER.**
**Phase 6. POST IMPLEMENTATION REVIEW.**

The definition of each phase of the model and the objectives of the phase are the following :

**Phase 1. CHANGE PROPOSAL EVALUATION.**

The objectives of change proposal evaluation are to provide information enabling the selection of change proposals for maintenance most beneficial to the business and also to provide information enabling development resources to be used efficiently and effectively.

**Phase 2. CHANGE DESIGN.**

The objectives of the change design phase are to carry out detailed investigations establishing a clear understanding of the existing software system. A second objective is to produce a change design in sufficient detail to confirm the technical approach and identify any critical areas of system performance. Other objectives of this phase are to refine the estimates for resources and equipment and to define the roles and responsibilities involved in the maintenance of the system.

**Phase 3. CHANGE IMPLEMENTATION.**

The objectives of this phase are firstly to produce the technical documentation describing what the application must do in order to satisfy the business proposal and system proposal produced in the earlier phases. The second objective is to develop and change the source code to reflect the technical documentation. A third objective is to test the program modules and connecting modules until they function to specification without error.

**Phase 4. SYSTEM TESTING.**

The objectives of this phase are firstly to ensure that current facilities are not degraded by new or amended facilities and secondly to ensure that the new or amended facility satisfies the business proposal and system proposal. A final objective is to ensure that the interfaces between sub-systems are correct and also that the interfaces between systems are correct.

**Phase 5. SYSTEM HANDOVER.**

The main objectives of this phase are to obtain user acceptance of a project or release and to produce user, operations and support documentation. Other important objectives are to identify and carry out the necessary training for the users of the system, the support staff and the operations staff. The system must then be operated in as near as possible to operational conditions.

**Phase 6. POST IMPLEMENTATION REVIEW.**

The objectives of this phase are to identify the good and bad features in a project development in order to identify and eliminate bad practices and ensure good ones are adopted. This phase of the model is not particularly relevant to this thesis so it is not developed.

The model is now further refined to include the various states within a project.

**Phase 1. CHANGE PROPOSAL EVALUATION:**

State 1: Initial Investigation of Change:

State 2: Authority to Proceed:

State 3: Determine Business Requirements:

State 4: Develop System Proposal:

State 5: Project Review and Scheduling:

State 6: QA Checkpoint

**Phase 2. CHANGE DESIGN:**

State 1: Perform Design:

State 2: QA Checkpoint

**Phase 3. CHANGE IMPLEMENTATION:**

State 1: Prepare Technical Documentation:

State 2: Prepare for Testing:

State 3: Create Test Data:

State 4: Produce Source Code:

State 5: Module Test:

State 6: QA Checkpoint:

**Phase 4. SYSTEM TESTING:**

State 1: Prepare for Testing:

State 2: Perform Testing:

State 3: Release for Acceptance Testing:

State 4: QA Checkpoint:

**Phase 5. SYSTEM HANDOVER:**

State 1: Prepare for implementation into organisation:

State 2: Perform user acceptance:

State 3: Perform confidence testing:

State 4: Run pilot:

State 5: Perform implementation:

State 6: QA Checkpoint:

**Phase 6. POST IMPLEMENTATION REVIEW:**

State 1: Prepare for maintenance review

State 2: Perform Maintenance Review

State 3: Project Analysis

State 4: Perform Actions

In the effective management of a software process the use of models which organise the process and make it visible, are most useful. This can be achieved by making documents, reports and reviews more visible [80]. In order to reach a particular state a number of tasks must be completed. These tasks are added to the model below. The following documents in table 2.2 are also included in this model.

| Documents Represented | Abbreviation used in the Model |
|---|---|
| Change Request | (CR) |
| Business Requirements | (BR) |
| System Proposal | (SP) |
| Project Schedule Offer to User | (PSO) |
| System Design | (SD) |
| Technical Documentation | (TDOC) |
| Test Plan | (TP) |
| Test Data | (TD) |
| Source Code | (SC) |
| Test Results | (TR) |
| System Test Plan | (STP) |
| System Test Data | (STD) |
| Review Plan | (RVP) |
| Review Document | (RD) |
| Analysis Report | (AR) |

Table 2.2: Deliverables Represented in the Model

Documents are represented in the model so that progress can be monitored at various stages during a project. Such a document-oriented model is used by many government organisations and software houses [80]. The documents also serve to identify the information which is produced as the by-product of the maintenance process. Decisions made in one phase of the model will be based on the information contained in documents produced from previous process model phases. Therefore modelling the document deliverables in this change analysis process model will help in deciding in which phases existing impact analysis techniques can be used effectively. Both the information available to maintenance staff and also the information which existing impact analysis techniques analyse in order to determine the change impact will be two factors in this study.

Any maintenance task which involves making a decision is underlined. For example, "Task 4: Identify Time Scales for Hand Over (*Decision No. 2*)" indicates that a decision must be made. These decisions are discussed in Section 2.4.

**PHASE 1 CHANGE PROPOSAL EVALUATION.**

State 1. Initial Investigation of Change:

Task 1. Receive Change Request from user (**CR**)

Task 2. Define Development or Amendment to Service (**BR**)

Task 3. Calculate Costs and Benefits (*Decision No. 1*)

Task 4. Define the business case for the change

State 2. Authority to Proceed:

Task 1. Assign Priority for Investigation

Task 2. Add Project to Project Register

State 3. Determine Business Requirements:

Task 1. Investigate Existing Service

Task 2. Identify Requirements for New Service

Task 3. Identify Service Levels Required

Task 4. Identify Time Scales for Hand Over (*Decision No. 2*)

Task 5. Re-calculate Costs and Benefits (*Decision No. 3* )

Task 6. Document Business Requirements

State 4. Develop System Proposal:

Task 1. Define Outputs, Inputs and Processing (*Decision No. 4*)

Task 2. Estimate the Project Risks

Task 3. Estimate Costs and Benefits of the Project (*Decision No. 5*)

Task 4. Document System Proposal (**SP**) (*Decision No. 6*)

State 5. Project Review and Scheduling: (*Decision No. 7*

Task 1. Categorise Projects (*Decision No. 8*)

(Adaptive, Corrective, Perfective, Preventive)

Task 2. Assign Priorities to Projects (**PSO**) (*Decision No. 9*)

State 6. QA Checkpoint

**PHASE 2. CHANGE DESIGN.**

State 1. Perform Design: (**SD**)

Task 1. Walk through System Proposal

Task 2. Refine Data Requirements

Task 3. Prepare Physical Data Design (*Decision No. 10*)

Task 4. Design Data Base Schema

Task 5. Separate Processing into Components (Architectural Level)

Task 6. Update Estimates (*Decision No. 11*)

State 2. QA Checkpoint

## PHASE 3. CHANGE IMPLEMENTATION.

State 1. Prepare Technical Documentation:

Task 1. Decompose Processing into Components (Detailed Level)

Task 2. Write Technical Documentation (*Decision No. 12*) **(TDOC)**

Task 3. Define or Redefine Internal File Design

Task 4. Produce Data Base Schemas

Task 5. Update Estimates

State 2. Prepare for Testing:

Task 1. Produce Test Strategy **(TP)**

Task 2. Identify Tests to Perform

Task 3. Determine Order of Testing

Task 4. Determine Test Resource Requirements

Task 5. Specify Expected Results

State 3. Create Test Data:

Task 1. Determine Requirements (*Decision No. 13*)

Task 2. Create Environment for Testing **(TD)**

State 4. Produce Source Code:

Task 1. Analyse Existing Code if necessary

Task 2. Design Programs and/or Program Modifications **(SC)**

Task 3. Perform Source Code Impact Analysis (*Decision No. 14*)

Task 4. Review and Update Estimates

Task 5. Extend Test Plan if necessary

Task 6. Produce and/or Amend Code

Task 7. Desk Check New Code

Task 8. Walk through Code

State 5. Module Test:

Task 1. Test According to Tests Plan

Task 2. Note Test Results

Task 3. Check Results Against Expectations (*Decision No. 15*)

Task 4. Review and Update Estimates

State 6. QA Checkpoint:

**PHASE 4. SYSTEM TESTING.**

State 1. Prepare for Testing.(**STP,STD**)

State 2. Perform Testing. (*Decision No. 16* )(**TR**)

State 3. Release for Acceptance Testing.

State 4. QA Checkpoint.

**PHASE 5. SYSTEM HANDOVER.**

State 1. Prepare for implementation into organisation.

State 2. Perform user acceptance.

State 3. Perform confidence testing.

State 4. Run pilot System.

State 5. Re-Insertion.

State 6. QA Checkpoint.

**Phase 6. POST IMPLEMENTATION REVIEW:**

State 1: Prepare for maintenance review. (RVP)

State 2: Perform Maintenance Review.(RD)

State 3: Project Analysis. (AR)

State 4: Perform Actions.

The model developed in this thesis implies a waterfall life cycle model where one phase flows into another. This type of model was derived from other engineering activities [76]. The waterfall life cycle style model is by far the most widely used software life cycle model and is well accepted [80]. However it is expected that the model in this thesis may be cyclic. The cycles represent the iteration of phases and tasks which may occur in the event of changes to the system proposal which may be initiated by users or maintenance staff.

In the model developed in this thesis there is no indication of parallelism. However there is no reason why various tasks cannot be performed in parallel. For example as soon as the content of the project is known in detail, such as, which source code modules will be affected, then preparations can be made for testing, system testing and system hand over. Therefore the amount of parallel processing would seem to be affected by the availability of information regarding the impact of a change. This also suggests that the inclusion of parallel processes in the maintenance model will reduce the time between request for change and system hand over. This is another important strategic reason for early detection of the impact of a change.

It is thought that the parallelism and cyclic nature of this model are organisation specific features and not generic features. The level of cycles and parallelism will depend on the technology used to support the tasks in the model. If poor requirements analysis technology is used then there will be many cycles. This is because the business proposal and system proposal will not be frozen if the requirements are initially incorrect. On the other hand if no impact analysis techniques are used, there may be few parallel maintenance processes except perhaps for any independent maintenance processes which can be identified by examining task dependencies [80].

## 2.4 Classifying Impact Analysis Techniques

The following questions, which are shown in table 2.3 must be answered in order to proceed from one state to another. All of the decisions made in Phase 1 are management decisions and all of the decisions in phases 2,3 and 4 are technical decisions. In order to answer the questions satisfactorily the following information shown in table 2.4 must be available.

This table 2.4 will facilitate a phased classification of impact analysis techniques according to when particular techniques may be used by maintenance staff. There are 16 possible classes within which impact analysis techniques can be placed, corresponding to the sixteen decision points in the table 2.3. From the table 2.4 it can be concluded that in knowing the direct impacts, then the indirect impacts and the relationship of new software to these components, can be regarded as central to the decision making process. This information is therefore very important in enabling maintenance and development resources to be used efficiently and effectively.

| DECISION | PHASE | STATE | DECISION |
|---|---|---|---|
| No. 1 | Phase 1 | State 1 | What are the cost benefits? |
| No. 2 | Phase 1 | State 3 | What are the time scales required? |
| No. 3 | Phase 1 | State 3 | What are the cost benefits? |
| No. 4 | Phase 1 | State 4 | What are the inputs, outputs, processing and data? |
| No. 5 | Phase 1 | State 4 | What is the implementation strategy? |
| No. 6 | Phase 1 | State 4 | What is the testing strategy? |
| No. 7 | Phase 1 | State 4 | What are the cost and benefits? |
| No. 8 | Phase 1 | State 5 | How can the project be categorised? |
| No. 9 | Phase 1 | State 5 | What is the priority for the project? |
| No. 10 | Phase 2 | State 1 | Which are the impacted system features? |
| No. 11 | Phase 2 | State 1 | What are the estimates now? |
| No. 12 | Phase 3 | State 1 | Which technical documentation needs to be written/amended? |
| No. 13 | Phase 3 | State 3 | Which test data is required? |
| No. 14 | Phase 3 | State 4 | What is the source code impacted? |
| No. 15 | Phase 3 | State 5 | What caused a particular defect? |
| No. 16 | Phase 4 | State 2 | What caused a particular defect? |

Table 2.3: Decisions Based on Impact Information

| DECISION | PHASE | STATE | INFORMATION REQUIRED TO MAKE DECISION |
|---|---|---|---|
| No. 1 | Phase 1 | State 1 | benefits to business in financial terms |
| No. 2 | Phase 1 | State 3 | volume of work in total |
| No. 3 | Phase 1 | State 3 | impacts and new inputs, outputs, processing and data |
| No. 4 | Phase 1 | State 4 | impacts and new inputs, outputs, processing and data |
| No. 5 | Phase 1 | State 4 | impacts and new inputs, outputs, processing and data |
| No. 6 | Phase 1 | State 4 | impacts and new inputs, outputs, processing and data |
| No. 7 | Phase 1 | State 4 | impacts and new inputs, outputs, processing and data |
| No. 8 | Phase 1 | State 5 | impacts and new inputs, outputs, processing and data |
| No. 9 | Phase 1 | State 5 | impacts and new inputs, outputs, processing and data |
| No. 10 | Phase 2 | State 1 | impacts and new inputs, outputs, processing and data |
| No. 11 | Phase 2 | State 1 | amount of work left |
| No. 12 | Phase 3 | State 1 | technical documentation needing maintenance |
| No. 13 | Phase 3 | State 3 | test data required |
| No. 14 | Phase 3 | State 4 | source code entities |
| No. 15 | Phase 3 | State 5 | source code entities |
| No. 16 | Phase 4 | State 2 | source code entities |

Table 2.4: Information Requirements for Decision Making

## 2.5 Summary

In this chapter the thesis is placed in the software engineering context. The reasons for the importance of impact analysis are presented. The maintenance process is examined and the results, namely a model of the decisions made in this process, are used to develop a classification scheme into which existing impact analysis techniques can be placed.

Unexpected linkages and errors in live systems suggest that maintenance staff are not provided with sufficient impact analysis information about a maintenance project. Many maintenance projects are delivered late or delivered on time but with a deterioration in system quality because the amount of work involved in a maintenance project was under-estimated. This is probably because the impact of a change to a system was not known at the beginning of the project. Therefore existing impact analysis techniques were perhaps difficult to apply at an early stage in the project.

The next chapter strengthens the argument that existing techniques are not useful for the early detection of the impact of a change. This is achieved by placing existing impact analysis techniques into the classification scheme developed in this Chapter.

The placement of each technique within the classification scheme is achieved by analysing the theory underpinning the technique, the characteristics of the technique, the information analysed by the technique and finally the results which may be produced by applying the technique in practice.

The use of this classification scheme provides an original method of evaluating and identifying the significance of existing impact analysis techniques with respect to how early these techniques can detect ripple effects. The same classification scheme is applied to the work developed in this thesis in order to facilitate a comparison between previous work and this new work.

# Chapter 3

# Impact Analysis Techniques: An Analysis and Comparison

## 3.1 Introduction

This chapter defines impact analysis and presents a review of the impact analysis literature in chronological order. An analysis and comparison of existing impact analysis techniques is discussed. The theoretical basis of each technique or approach is examined, aswell as its practical use, its value and significance, and also the problems associated with it. In particular the stage the techniques can be used in the maintenance process is identified. Lastly the characteristics of documentation are examined.

## 3.2 What is an Impact Analysis?

### 3.2.1 Definition

The term impact is used by many authors to denote the knock on effects propagated through software, because of the different types of dependencies in software systems. Some authors use the term impact to mean the same as **ripple effect**. Yau [95] defines the ripple effect as the following :

*Ripple effect is the phenomenon by which changes to one program area may be felt in other program areas.*

This view suggests that ripple effects only occur at the source code level. For example a modification in an assignment statement for variable A may have impacts in any statement in which A is used. A modification in an IF or WHILE type statement may affect each statement whose execution is conditioned by the IF or WHILE construct. Each ripple effect in turn can have other ripple effects when the new value of A is used to evaluate another variable B. Every statement where B appears may be affected. This type of effect is called a copy propagation [3].

Agusa et al, [2] define the ripple effect as the following :

*The ripple effect is the situation that some modification of requirements description results in a logical inconsistency and we are unable to read that description as intended.*

This view suggests that the ripple effect can also occur in documentation. Pfleeger and Bohner [72] define impact analysis as the following :

*Impact analysis is the assessment of the effect of a change and aids the maintenance team in identifying software work products affected by software changes.*

This view of the detection of the ripple effect, that is, **impact analysis** or **ripple effect analysis**, suggests that all work products, both documentation and source code are prone to the impact. This type of analysis allows the maintenance managers and programmers to assess the consequences of a particular change to the source code. It can be used as an estimate of the cost of a change. The more the change causes other changes to be made ('ripples') then, in general, the higher the cost. Carrying out this analysis before a change is made, allows an assessment of the cost of the change and allows management to make a trade-off between alternative changes.

In this thesis the phrase impact analysis will be used instead of ripple effect analysis and two types of impacts will be addressed :

1. **direct impacts** which are work products specifically mentioned in a change request;
2. **indirect impacts** which are additional work products requiring maintenance in order to remain consistent with the changes made by the direct impacts.

In order to maintain a software product, impacts must be considered in two types of work products:

1. system documentation which includes all of the products describing the source code of a product;
2. source code which includes all data files descriptions, modules and job control language.

System documentation consists of the following components :

1. requirements;
2. specifications;
3. architectural designs;
4. detailed designs.

Indirect impacts on documentation can be caused by the following :

1. description of assignment of data from one data entity to another data entity where a data entity can be any source code component capable of storing data;
2. deletion of a document component which another document component may explicitly or implicitly reference.

Indirect impacts in source code can be caused by:

1. data declaration change;
2. data flow value assignment;
3. control flow if assignments;
4. calling nested procedure or function;
5. functional function assignment.

There are two types of indirect impact identified by Yau, Collofello and McGregor [95] Firstly the **logical ripple effect** which is an inconsistency introduced into a program area by a change to another program area and secondly the **performance ripple effect** which is a change in a module's performance as a consequence of a software change in another module. In this thesis performance ripple effects are not considered because they are complex and certainly merit a separate investigation. The term **ripple propagation** means the spreading of the ripple effect. **Stability analysis** differs from impact analysis in that it considers the sum of the potential impacts rather than a particular impact caused by a change .

**Stability** is considered as one of the major attributes of maintainability because the understanding and modification of a program, and also the calculation of possible impacts, constitutes a major proportion of the software maintenance effort. Program stability has been defined by Yau and Collofello [97] as :

*The resistance of a program to the amplification of changes in the program*

**Documentation stability** is defined in this thesis as :

*The resistance of documentation to the amplification of changes in the documentation*

This thesis concentrates on impact analysis and is not directly concerned with evaluating the quality attributes of stability.

## 3.3 Historical Overview of Impact Analysis Techniques

The research into impact analysis has evolved over two decades. The origins of impact analysis research can be traced back to 1972 when Haney published the first paper addressing the subject. This paper presented an adjacency matrix which stored probabilities of one module propagating an impact to other modules. The probabilities were based on previous changes.

Since 1972 many people have investigated the problem of detecting the impact of a proposed change. Most approaches have analysed source code impacts and not documentation ripple effects. There have been several lines of approach taken. Surprisingly few people have explored the statistical approach introduced by Haney which offers a very practical approach to detecting indirect impacts. Several researchers have analysed source code dependencies and in particular data flow dependencies. Some methods are based on storing semantic information concerning dependencies in source code. Yau [95, 96, 97, 98] produced several noteworthy results in the form of algorithms to compute the ripple effect and a module's stability. In the early 1980's these techniques were studied in great detail. Other contributions in the mid 1980's looked into documentation traceability, for example SODOS [44] and DIFF [34, 36]. One other noteworthy result is impact analysis in algebraic specifications [64]. None of the previous approaches involved any kind of mathematical formal specification, except for the Designer/Verifier Assistant [62]. One of the most important recent contributions was made by Pfleeger and Bohner [72]. The approach adopted was a graph model of all the life cycle work products which could be measured. As changes were requested, measurements of the impact could be made and implementation decisions could be made [72].

The trend in impact analysis research, as well as in software engineering, is towards representing and using higher levels concepts rather than just source code listings [18]. The reasons for this trend are related to the developments in software engineering namely, the recognition of work products other than program code as important in particular for large systems. The future directions for impact analysis research will be firstly, in characterising interconnectivity in work products other than source code and secondly in the development of techniques for identifying projects, with possible system wide ramifications.

## 3.4 Analysis of Existing Impact Analysis Techniques

Recently a very simple scheme for classifying impact analysis techniques was proposed by Collofello and Vennergrund [18]. In this scheme, impact analysis techniques are divided into two types, as determined by information analysed by particular techniques. The two types identified were syntactic and semantic impact analysis techniques. The scheme is broad but not free from objections. The classification is very simple and clear however it is not useful when choosing particular impact techniques to be applied at various stages during a software maintenance project.

There are actually three distinct categories of impact analysis techniques. *Syntactic* based impact analysis techniques work on source code representations [97]. The impacts are determined by analysing the control and data flow of the source code. *Semantic* based impact analysis techniques work on higher level representations than the source code [18, 93]. The semantic information reflects the intention of the program and represents knowledge of the basic assumptions which must be true for the correct operation of the system. *Statistically* based impact analysis techniques work on source code representations such as probability interconnection matrices representing the probability of impacts to other source code components [41, 81].

In this thesis an alternative scheme of classification with finer divisions is proposed. The divisions are based on when the techniques can be applied in the software maintenance process. Specifically the states modelled in Chapter 2 will form the divisions of classification scheme .

Any research contributions which fall into any of these three categories syntactic, semantic and statistical are discussed in this thesis.

### 3.4.1 The 1970's Developments

In 1971 Grosch [40] identified that for some large systems the problem to be solved and the system designed to solve it are in such a constant flux that stability of the software is never achieved. The paper indicated that impact analysis was a problem but did not propose any solutions.

#### Module Connection Analysis (1972)

Haney [41] was not the first person to identify that the problem of contingency laden estimates for development projects are frequently exceeded. However Haney was one of the first people to identify the problem of underestimated maintenance project resources :

*Our most significant problem has been gross underestimation of the effort required to change (either for purpose of debugging or adding function) a large, complex system.*

Haney describes a technique which models the stability of a large system as a function of its internal structure. This technique includes a matrix formula for modelling the ripple effect of changes in a system. The matrix records the probability that a change in one module will necessitate a change in any other module in the system. For example a release information sheet can be used to revise the probabilities of module impact propagation by recording the changes made to modules and also to the other modules affected. The more this technique is used the greater the predictive value of the model.

The matrix formula based on these probabilities can be used for explaining why the process of changing a system is more involved than maintainers might believe. The technique can be used to estimate the number of changes required as a result of changing one particular module. No data is presented to show how the estimated total changes are related to cost of the actual changes required. A weakness in this technique, namely the assumption that all modifications to a module have the same magnitude of impact, could make the estimation of change resources inaccurate. The distinctive feature of this approach is that it analyses source code only. Hence it is only possible to deduce source code impacts using this technique. Therefore the technique can only support the decisions 13,14,15 and 16 [1].

### A Program Stability Measure (1977)

A program stability measure was also developed by Song [81]. A technique is presented which quantifies the quality of a program in terms of its information structure. The information structure is based on the sharing of information between components of a program. The technique is based on a connectivity matrix and random Markovian processes and makes a similar assumption to the technique described by Haney, that all modules changed share the same information content and

[1]These decisions are presented in Chapter2, table 2.3, page 22

the sharing of information between modules is equal. The technique presented is more complicated than that of Haney and has not been validated. The distinctive feature of this approach is that it analyses source code only. Hence it is only possible to deduce source code impacts using this technique since only source code information is analysed. Therefore the technique can only support the decisions 13,14,15 and 16.

### Ripple Effect Analysis (1978)

Yau et al [95] make an important contribution to impact analysis techniques. This was probably one of the first papers to coin the phrase ripple effect. Two techniques developed to analyse the impact of a change from the functional and performance perspectives are developed. The first technique tracks all statements within a module which may be affected by the definition of a variable. Then the inter-module boundaries are considered. The performance ripple effect, that is, the analysis of a program module whose performance may change as a consequence of a program modification, is a complex task. This is achieved by developing a performance dependency relationship between pairs of modules. This exists if one module can cause a change in performance in another module if it should be modified. Both techniques are discussed from a philosophical point of view and details of algorithms are not included. It is only possible to deduce source code impacts using this technique and therefore the technique can only support decisions 13,14,15 and 16.

### A Designer/Verifier Assistant (1979)

Moriconi [62] describes a theory which can answer questions about the effects of hypothesised changes and can make proposals on how to proceed in an orderly fashion when making changes to software. This was the first paper to give advice on how to make changes and it was the first paper to address the changing of designs for formally verified programs. The distinctive feature of this approach is that it analyses source code and documentation in the form of functional specifications therefore it is only possible to deduce inconsistencies in design specifications and direct impacts on the source code, from specifications. This is because the theory developed is for simulating changes to software designs. Therefore the technique can support the decisions 12 and 14.

### Program Slicing (1979)

Weiser [91] demonstrated that programmers view programs not necessarily from a textual or module structure point of view. Participants in experiments were asked to debug three programs and they were provided with program fragments some of which were based on slices and others were random. The participants were then asked to rate the fragments, based on how sure they were that it had been used in the programs. The results showed that programmers view programs in terms of independent strands of execution. A strand of execution is defined as all the program statements which contribute to the contents of a program variable. It is only possible to deduce source code impacts using this technique and therefore the technique can only support the decisions 14,15 and 16.

### 3.4.2 The Early 1980's Developments

#### Stability Measures (1980)

A measure of the logical stability of software systems was proposed by Yau and Collofello [97]. The logical stability of a module is a measure of the resistance to the impact of such a modification on other modules in the program in terms of logical considerations. There are two parts of logical ripple effect considered by Yau and Collofello. One part concerns intra-module change propagation and the other concerns inter-module change propagation. Therefore the information on which the metric is based can be used to identify other modules impacted. This is achieved by analysing the data flow within modules and the data flow between modules. It is only possible to deduce source code impacts using this technique since only source code information is analysed. Therefore the technique can only support the decisions 14,15 and 16.

#### System Level Ripple Effect Analysis (1983)

Agusa, Kishimoto and Ohno [2] convert system requirements into clausal form and analyse them using resolution and predicate calculus. The technique is one of the more theoretically motivated approaches to ripple effect analysis. This approach is an early ripple detection approach. The information analysed by the analyser is the requirements specifications. The approach adopted was significant. It advanced ripple effect techniques because it was the first approach to detecting the ripple effect in a top down manner though various levels of abstraction. Resolution is used to detect logical inconsistencies in the requirement description clauses. The distinctive feature of this approach is that it analyses documentation and therefore the technique can support the decisions

12 and 14.

### Requirements Impact Analysis (1983)

Chikofsky [16] discusses a Problem Statement Analyser (PSA) and its associated Problem Statement Language for software maintenance. This tool manipulates user requirements and system design at the beginning of the life cycle. Queries are given to the tool to produce the modules affected by a change to a system. This was a significant contribution to the field of impact analysis as it was the first investigation to address early detection of the ripple effect. The distinctive feature of this approach is that it analyses documentation and therefore the technique can support the decisions 12 and 14.

### An Efficient Technique for Impact Analysis (1984)

Yau and Chang [98] developed a new stability analysis technique which is similar to the technique described above but which is easier to apply to large programs. This technique is not a code based technique, it is instead intended to be applied to the design phase of software project. The technique is more efficient because the source code is not analysed. The complete information regarding variable definition and usage is not a requirement of this technique. Instead dependencies between global variables and modules are extracted from design documents. This approach uses whatever design information is available and therefore the technique can support decisions the 14, 15 and 16.

### Relational Query Languages (1985)

Glagowski [38] developed a software maintenance tool to support ripple effect analysis. The tool is based on a relational model to store system documentation and is designed to help the maintainer achieve a quick understanding of system structure. The documentation is very low level, for example flow charts, procedural abstractions and data abstractions. This relationship information can be produced from source code itself, automatically. However it is not abstracted from written documentation. The tool is very useful for recording graphical information because the lines on flow diagrams can be represented by relations. The distinctive feature of this approach is that it analyses documentation only. However it is only possible to deduce source code impacts. Therefore the technique can support the decisions 14, 15 and 16.

### Design Stability Measures (1985)

Yau and Collofello [99] also developed design stability measures that are intended to be used for assessing the quality of program designs based on design documentation in the design phase of software development. They are based on the counting of assumptions made about module interfaces and shared global data structures. This information is taken from program design documentation. The distinctive feature of this approach is that it identifies source code impacts by analysing design information. Therefore the technique can support the decisions 12 and 14.

### Implicit Information Detection (1985)

Stankovic [82] investigated the implicit information problem, that is, the detection of relationships that are not directly implemented in the code and data. For example two copies of a data structure which must remain consistent, are modified by two different procedures. The procedures must update both copies of the data structure otherwise the requirement will be violated. This work is an extension to the work by Chikofsky [16]. Control flow and data flow information are recorded and information is displayed in a flow chart form to the user. The PSL/PSA structures are connected to the source code information to allow the user to detect implicit relations which may be violated by proposed changes. The distinctive feature of this approach is that it analyses implicit dependencies stored in database and therefore the technique can support decisions 12 and 14.

## 3.4.3 The Late 1980's Developments

### SOFTLIB (1986)

Sommerville et al [79], developed a documentation library system based around the UNIX file store. In this paper all documentation for a source code component is grouped into a set. This makes it easy to trace the different representations of a particular component. The distinctive feature of this approach is that it analyses both documentation and source code. Therefore the technique can support the decision 12.

### Document Support Environment (1986)

This work by Horowitz and Williamson [44] describes a computerised environment, SODOS (Software Documentation Support), which supports the definition and manipulation of documents used in developing software. In SODOS software life cycle documents are linked by recording intra and inter document relationships. The underlying graph model is independent of any particular documentation or development method. The main contribution is the facility to detect inconsistencies before the later stages in the maintenance process. The model of documentation is based on a graph structure and is represented as a relational model. No impact analysis algorithms are suggested in SODOS. The distinctive feature of this approach is that it analyses documentation only and therefore only decision 12 is supported.

**Neptune (1986)**

Delisle and Schwartz [22] describe a documentation system called Neptune which is a hypertext database for documenting program structure. Neptune uses a directed graph of nodes to model the structures of a program and documentation text. Attributes attached to nodes describe the content of the nodes. Links between documentation can also have attributes assigned to them. Neptune concentrates on documenting the coarse structure of documentation at the level of modules and procedures. The distinctive feature of this approach is that it analyses both documentation and source code. The system, like the SODOS system, has no underlying impact model therefore only decision 12 is supported.

**Enhancing Software Reliability (1986)**

Thebaut and Wilde [85] argue that since the impact of a proposed change helps the maintainer understand the program before it is changed this understanding will lead to greater reliability in the new version of the source code. This approach includes three techniques for achieving this goal, namely impact analysis, dynamic analysis and symbolic evaluation. The impact analysis algorithm marks the program statements which are directly affected by the change. Then other statements which are affected by the first set of changes are marked and so on. This approach is very simplistic and readily applied. The distinctive feature of this approach is that it analyses source code only and therefore the technique can support the decisions 14, 15 and 16.

**FORTUNE Document Environment (1987)**

FORTUNE as described by Mullin and McGowan [63], is a collaborative project forming part of the Alvey Software Engineering programme. Its objective was to produce an integrated documentation tool that will support the creation and update of textual and graphical documentation throughout a project life cycle. FORTUNE is based on a traditional life cycle model. The facility provided supports links between different levels of documentation such as requirements and design. The distinctive feature of this approach is that it analyses documentation only and therefore the technique can support decision 12.

**Semantic Information Tool (1987)**

Ripple effect analysis is based on semantic information which is higher level than source code and not derivable from source code. The semantic information represents both the intent of the program (function) and also knowledge of the basic assumptions which must be true for correct operation of the system using semantic nets. A prototype impact analysis tool based on both syntactic and semantic information was constructed by Collofello and Vennergrund, and is described [18]. The prototype enables the maintenance information to be captured in the form of semantic conditions which can be linked to syntactic constructs. This allows the maintainer to determine the impact of a change based on semantic and syntactic information. The distinctive feature of this approach is that it analyses source code only. Therefore the technique can support decision 14.

**Data Flow Analysis (1988)**

Keables et al. [50] developed algorithms for regression testing. The algorithms are used for variable analysis, reaching definitions, definition chains and use chains. The algorithms are used to detect new undefined references and useless definitions. The reaching definitions and modified definition and reaching chains assist the maintainer in debugging modified code. The distinctive feature of this approach is that it analyses source code only and therefore the technique can support decision 14.

**Dependency Directed Reasoning (1988)**

Dhar and Jarke [23] present a maintenance tool which supports an approach to impact analysis using a knowledge based system. The system stores knowledge about a software system which can be used to deduce which impacts will result from a modification. This is achieved by using a formalism called REMAP (REpresentation and MAintenance of Process Knowledge). The contribution of this paper was the recording of design decisions, for example the reasons for the use of a particular design solution in a system. The distinctive feature of this approach is that it analyses documentation and source code. Therefore the technique can support the following decisions 12 and 14.

### Modifiability Metrics (1988)

Yau and Chang [100] present a model for measuring software modifiability . The central concept of this model is attributed to its localisation property. The localisation property of a module indicates whether or not changes made to the module would have only a localised effect. The metric is developed in two stages, firstly the intra-localisation property is developed and secondly an inter-localisation property is developed. A number of experiments were conducted to validate the metric. 180 maintenance changes were made to five programs. The results show that the metric is a good measure for the estimation of the effort required to modify the modules of the programs. The distinctive feature of this approach is that it analyses source code only and therefore the technique can support decision 14.

### Hypertext and CASE (1988)

This work by Bigelow [10] describes the basis for a programming environment. Hypertext is used as a CASE environment in which documents and code can be related to each other. Whilst this tool provides traceability it is not a good facility for impact analysis since there is no underlying model of interconnectivity. Instead the links between objects are created freely. The distinctive feature of this approach is that it analyses both documentation and source code. Therefore the technique can support the decisions 12, 13 and 14.

### Module Interconnection Graphs (1989)

Calliss [14] provides a model of interconnectivity between **modules** which is well founded in graph theory. The model of inter-module dependencies can be used for impact analysis between modules. Previous models of interconnectivity had concentrated on intra-module dependencies. Techniques are provided which analyse the entities shared between groups of modules. The model and model analysis techniques were implemented using a relational database. The distinctive feature of this approach is that it analyses source code and therefore the technique can support decision 14.

### 3.4.4 Recent Advances

#### Document Integration Facility (1989 and 1990)

The System Factory project [34, 36] contains a document integration facility which provides a mechanisms for developing and documenting software development objects and their relationships. DIF, Document Integration Facility is a hypertext system for integrating and managing documents produced during the development of a project. The hypertext features provide traceability between life cycle documents including source code and test cases. Each document has a hierarchical structure and a template which is filled in as a form. This ensures that all documents of a particular type are standardised. The distinctive feature of this approach is that it analyses documentation and source code. Therefore the technique can support decision 12,13 and 14.

#### Deductive Databases for Code Analysis (1989)

Dietrich and Calliss [24] developed a deductive database for analysing the interconnections between source code modules. This work is based on the theoretical work on interconnection graphs developed by Calliss [14]. The distinctive feature of this approach is that it analyses source code only and therefore the technique can support decision 14.

#### Dynamic Slicing (1990)

The conventional notion of a program slice is the set of all statements which might affect the value of a given variable occurrence. The notion of dynamic slice is introduced by Agrawal and Horgan [1] and is the set of all statements which actually affect the value of a variable occurrence of a given program input. Four approaches are considered by Agrawal and Horgan for dynamic program slicing.

The first approach considers the nodes of a dependence graph which appear in the execution history for any given program. Then the dynamic slice is derived by slicing the projected dependence graph with respect to a particular variable. The second approach is a more accurate approach than the first as only the edges of the dependence graph which are actually executed are traversed. This solves the problem of multiple out-going data dependencies. The problem with the first two approaches is that if a node occurs multiple times in an execution history such as in an iterative control construct, large slices will be computed. The third approach creates a node for each occurrence of a statement in an execution history, with outgoing dependency edges to only those statements on which the statement occurrence is dependent. The fourth approach only creates a new node if another node with the same transitive dependencies does not already exist. The distinctive feature of this approach is that it analyses source code only and therefore the technique can support decision 14.

**Traceability Graphs (1990)**

A framework was postulated by Pfleeger and Bohner [72] which defines horizontal traceability as the relationship between particular work products and vertical traceability as the traceability within a particular object. The authors claim to have introduced vertical traceability. However this appears to be a synonym of intra-document relationships which is included in the SODOS approach. Whilst this paper provides a good framework for maintenance impact metrics, nevertheless the documentation aspects of the graph model appear imprecise, for example the question of what constitutes a node in the graph. Many requirements, specifications and designs are written in natural language and therefore there must be strict transformations performed on the document to produce a traceability graph which is a good model of the system interconnectivity. Pfleeger and Bohner recognise impact analysis as a primary activity in software maintenance and present a framework for software metrics which could be used as a basis for measuring stability of the whole system including documentation. The framework is based on a graph, called the traceability graph, which shows the interconnections among source code, test cases, design documents and requirements. This framework provides a good example of including software work products as part of the system, although it is anticipated that the level of detail on a diagram is insufficient to make detailed stability measurements. The distinctive feature of this approach is that it analyses documentation and source code and therefore the technique can support the decisions 12 13 and 14.

**Slicing Using Dependence Graphs (1990)**

Horwitz et al, [45] advanced program slicing techniques by developing a program slicing technique which generates a slice of an entire program, where slices cross boundaries of procedure calls . This is achieved by a slicing a dependence graph called a system dependence graph. The graph models the main program and a collection of procedures, and any parameters which passed between the procedures. The distinctive feature of this approach is that it analyses source code only and therefore the technique can support decision 14.

### Teleological Models (1990)

Karakostas [48, 49] addresses the problem of mapping change requirements to operational software and contributes a new model which maps requirements to operational software. The model is a teleological model which records the purpose of source code objects with respect to business system processes. The model is different from an architectural model as an architectural model only concentrates on source code. The teleological model is also different from a conceptual model as it does not model how the system is implemented. This approach is probably the best early detection method as it links domain concepts with source code objects. Therefore the technique can support decisions 4, 12 and 14. The reason a teleological model can help a maintainer or maintenance manager to make decision 4 is because the model links application domain concepts with source code objects and change proposals are written at the domain level.

### Graph Model for Software Evolution(1990)

Luqi [57] presents a graph model of software evolution. The objects and activities in software evolution are modelled with a graph to enable automatic assistance in maintaining consistency in an evolving software system. The model consists of two components namely system components and evolution steps. Two dependencies of interest are "uses" and "derives". The derives dependency provides relationships between different levels of abstraction and the uses dependency provides a structural relationship at the source code level. The distinctive feature of this approach is that it analyses both documentation and source code. Therefore the technique can support decision 12 and 14.

### The Evolution Support Environment System (1990)

Ramamoorthy et al [75] present a system providing a framework for capturing and making available semantic information about software components. The system provides traceability among specifications, design, code, and test cases during development. However the main aim of this work is to provide version control for evolving systems. The interconnection model underlying the traceability facilities is based on an entity relationship model. The main navigation facilities are forward and backwards through different levels of abstraction. The distinctive feature of this approach is that it analyses both documentation and source code. Therefore the technique can support decisions 12 and 14.

### An Environment for Documenting Software Features (1990)

Hart [42] developed an environment which addresses the problems in modifying software by explicitly linking software designs to implementations. The implementation is partitioned according to the features it supports. This paper argues that existing approaches do not make explicit links between design and source code. The approach adopted in this paper is to document the design document elements with feature contexts. Features can span a range of program structures and Hart claims that they provide the maintainer with a functional slice of a system. The distinctive feature of this approach is that it analyses documentation and therefore the technique can support decision 12.

### Software Evolution Expert System (1990)

Pau and Kristinsson [70] describe a software maintenance system for understanding programs, generating and updating software correction documentation and helping interpret errors and is called SOFTM. The system relies on a transition network based code description and diagnostic inference procedures. The system is aimed at error-prone corrective maintenance tasks by non-designer staff. The distinctive feature of this approach is that it analyses source code and therefore the technique can support decision 14.

### Propagating Changes in Algebraic Specifications (1991)

Nakagawa and Futatsugi [64] explore the use of algebraic specification methodologies in dealing with the impacts of changes during the software development processes. Algebraic specifications are used because algebraic specifications arose out of abstract data type research. Algebraic specifications offer mechanisms for modularisation which in turn offer a neat way of defining products and their relations. This paper is a significant contribution because it offers the first precise analysis of the impact of a change unlike other approaches all of which offer approximations of changes. The distinctive feature of this approach is that it analyses documentation and therefore the technique can support decision 12.

### Decision Based Software Development (1991)

Wild et al., [93] proposed an interesting new paradigm for impact analysis on the products involved in the maintenance activity. This paradigm views the design process as a series of interelated decisions which provide a new abstraction of the documentation base. The relationships between decisions in life cycle products can help assess the impact of changing a system. Some evidence is provided that this approach is useful for assessing the impact of change. A comparison between a functional based impact and decision based impact was conducted. The decision based approach was a lower level of granularity than the functions. The evidence presented showed that the impact on decisions was more precise than the functions based approach. Whilst this approach appears very useful for making new systems which are highly maintainable, it may be difficult to take an existing release of documentation and code and then establish a decision dependency graph. The distinctive feature of this approach is that it analyses documentation and therefore the technique can support decision 12.

**A Code Analysis Knowledge Base (1992)**

Dietrich and Calliss [25] produced a knowledge base for inter-module code analysis. The knowledge base can be used to analyse programs written in languages that contain a clustering concept called module. The work includes a general design for a code analysis knowledge base using an entity relationship modelling approach. The contribution of this work is analysis techniques for new languages which have scoping rules for importing and exporting data to and from modules. The distinctive feature of this approach is that it analyses source code and therefore the technique can support decision 14.

## 3.5 Comparison of Existing Impact Analysis Techniques

By considering the information analysed by existing impact analysis techniques they can be placed in the table 3.1. The double horizontal line in table 3.1 indicates the critical region in the maintenance process where projects are allocated budgets based on the contents of the project. It is clear from the evidence in this chapter that most techniques for impact analysis are more useful in the later stages of software maintenance but not before the allocation of resources to projects are made. Therefore software maintenance managers will have to estimate the size of a maintenance project rather than using an impact analysis technique to understand the components of a proposed maintenance project.

| NO. | DECISION | NUMBER OF TECHNIQUES |
|---|---|---|
| No. 1 | What are the cost benefits? | |
| No. 2 | What are the time scales required? | |
| No. 3 | What are the cost benefits? | |
| No. 4 | What are the inputs, outputs, processing and data? | * |
| No. 5 | What is the implementation strategy? | |
| No. 6 | What is the testing strategy? | |
| No. 7 | What are the cost and benefits? | |
| No. 8 | How can the project be categorised? | |
| No. 9 | What is the priority for the project? | |
| No. 10 | Which are the impacted system features? | |
| No. 11 | What are the estimates now? | |
| No. 12 | Which technical documentation needs to be written/amended? | ***************** |
| No. 13 | Which test data is required? | ****** |
| No. 14 | What is the source code impacted? | ************************* |
| No. 15 | What caused a particular defect? | ****** |
| No. 16 | What caused a particular defect? | ****** |

Table 3.1: Histogram of the Frequency of Techniques which can be used in various states of the Maintenance Process

The author's view of existing impact analysis techniques is that they are difficult to apply at an early stage in the software maintenance process. There are a number of problems which are associated with existing approaches to impact analysis :

1. Existing source code syntactic techniques are difficult to apply at the change proposal phase of a project because the relation of the change proposal to the source code is often not very well understood at this stage, and change proposals are written at a much higher level of abstraction than the code.

2. Existing syntactic techniques do not provide sufficient detail to give an accurate understanding of the effect of a change because most of the emphasis concerns the impact on the source code.

3. Semantic annotations of source code syntactic relationships are also difficult to reason with at the change proposal evaluation phase, because change proposals are written at a much higher level of abstraction than these semantic annotations.

4. Statistical techniques have only addressed the connections between modules and have ignored intra-module connections and also documentation inter and intra-document connections.

5. Documentation impact analysis techniques are often just an analysis of keywords common to two or more documents. This approach is practical, however it does not capture the essence of the indirect impact.

Gallagher [33] argues that the maintainer can locate the documentation to be maintained once the source impacts have been analysed. However system documentation should reflect changes in business requirements before the source code is considered. Often during the development of a new software system the deliverables produced at particular phases are validated against deliverables produced in previous phases. For example checking a design satisfies a specification. Therefore in the maintenance process the same approach should be adopted. It is therefore not possible to start examining source code impacts at a requirements stage of a maintenance project.

In making a change, it is desirable to minimise the defects, effort and cost whilst maximising the customer satisfaction [33]. To minimise defects the impact of software changes must be detected before they manifest themselves during the various testing phases of the maintenance process. To minimise the effort and cost afforded to a project, the proposal formulated must be produced using accurate information in order to provide a sound basis for the development process in subsequent phases. Customer satisfaction can only be achieved if the service provided by the delivered system conforms with the requirements set out in the business requirements. The required functionality will only be delivered if the correct budget is allocated to a project. This is heavily dependent on the assessment of the size of a project. Customer satisfaction can also only be achieved if the service provided by the system operates free from error. All of these requirements are heavily dependent on the production impact analysis information early in the maintenance process.

In order to achieve an insight into and an understanding of the impact at an early stage it is necessary to analyse the information which is available at such a stage in the maintenance process. Since documentation is maintained before source code, it is necessary to analyse documentation in order to achieve an understanding of the impact [28]. A model of documentation is easier to manipulate than the actual documentation as it would contain less details. A sample of publications concerning documentation are analysed to abstract some general principles and characteristics of structure and content of documentation. The results of this analysis with added new ideas will form the basis of a new model developed in Chapter 4.

## 3.6 What is Documentation?

### 3.6.1 Definition

Documentation is defined by [46] as:

*A collection of documents on a given subject.*

A document is defined by [46] as:

*A medium, and the information recorded on it, that generally has permanence and can be read by a person or a machine. Examples in software engineering include project plans, specifications, test plans, user manuals.*

The survey conducted by Lientz and Swanson [55] identifies the quality of software documentation as the most significant technical problem in program comprehension. High level documentation explaining the overall purpose of the software system and the relationships between components can be useful in obtaining an overview picture of a system [59]. Low level documentation explains what a program does, and why. It is essential for maintaining all but the simplest of programs.

Both types of documentation can be stored in a documentation base. A documentation base is a set of many different kinds of document components. Document components are defined in this to be :

*One of the parts of which a document base is composed such as textual components, graphical components and components of mathematical text.*

In commercial data processing departments informal specifications seems to be favoured. However formal specifications tend to be favoured where there is a necessity for precision and where a strong mathematical background exists. Although mathematically based specifications are used in some organisations it is argued that these will have evolved from natural language specifications.

It is difficult to detect impacts in systems produced by traditional development methods because of the informality and implicitness of the interconnection relationships between the entities of the system. The problem is also compounded by the fact that these systems have often been revised many times and the quality of the source code and documentation will have been degraded with an increase in structural complexity [53]. If keyword descriptions are used to characterise interconnections in documentation then many false predictions may be made about the impact of a change. A keyword has no meaning, for example the word "file" could be a noun in one document and a verb in another document. The fact that both documents contain this word does not necessarily imply any dependency between these documents. Keywords do not provide an accurate description of what may be contained in a document or document component. It may also be difficult to control the amounts of document components impacted. For example little or no information may be produced because of a bad choice of keyword, or large amounts of document components may be impacted which are not relevant to a change because of the context of the keywords.

Hence in order to increase the predictive value of an impact analysis technique, an attempt at understanding the semantics of a documentation must be made. In addition the documents must be decomposed into a unit of analysis which makes the results produced by applying an impact analysis technique meaningful. For example to say that two chapters are impacted is not meaningful because the amount of information they convey could be very different. Secondly these chapters may be very different in length. If the chapters are converted into standard units of analysis then it can be more appropriate to make a comparison between them.

### 3.6.2 Classes of System Documentation Components

Software Documentation as defined in [46] consists of:

- User Documentation:
  - User Manual

- Program Documentation:
  - Requirements Specification
  - Design Specification
- Data Documentation:
  - Data Model
  - Data Dictionary

Data documentation can be useful for determining where data is used in software systems. However data documentation does not describe any indirect impacts. For example an entity relationship model can describe indirect ripple effects but there may no link between the entities in the model and the entities within software constructs. A requirements specification describes the requirements for a system. Typically included are functional requirements, performance requirements, interface requirements, design requirements and development standards. This document may therefore include descriptions of the processing of data. A design specification is a document which describes the design of a system or component. Typical contents include system or component architectures, control logic, data structures, input/output formats, interface descriptions and algorithms. There may be much impact information detectable at this level of abstraction. User documentation describes the way in which a system or system component is to be used in order to obtain the desired result. It therefore does not describe any impact information. The requirements specification and design specification seem to be most useful in detecting the impact information of a change. In software documentation a requirement component in a requirement specification will be linked to one or more design components in a design specification. This type of dependency can be called a life cycle link.

The components of documentation must be capable of having their contents described sufficiently well in order to form interconnections between components. This is because interconnections are based on the content of document components.

Arthur in [4] describes several types of document content types, which are the following :

1. description of input;
2. description of output;

3. description of processing;
4. description of an interface;
5. description of a data flow;
6. description of human factors;
7. description of hardware;
8. description other systems;
9. description of other documents.

This list of content types could easily be extended. For example the document type description of input could be decomposed into two sub-types such as description of keyboard input and the sub-type description of file input.

The concept of a documentation segment can also be introduced, where a segment is defined to be :

> *a defined size of unit of documentation which can be characterised in terms of document types.*

### 3.6.3 Characteristics of System Documentation

The sheer diversity of the constructs and facilities which form documentation poses a challenge to formal models of the structure of documentation. It is possible however to analyse documents if the following observations on documentation are taken into consideration :

- documentation has some sort of topology which can be subjected to analysis;
- documentation consists of one or more documents;
- documentation can be hierarchical in structure;
- documents can contain hierarchically structured sections;

- documents are logically connected through the mutual information they share, for example two documents could both describe the processing of a data file;
- documents can contain inter and intra document dependencies;
- documents can contain text and/or graphics;
- sections can be decomposed into segments which provide analysis with standard size document component;
- these irreducible segments can represent the smallest component of documentation;
- some documents may be missing or incomplete;
- some sections of documents may be missing or incomplete;
- document sections may contain subjects or topics that we shall call themes.

From these observations the hierarchical structure of documentation can be modelled for subsequent analysis. This hierarchical structure can be mapped to a documentation tree. A documentation tree is a hierarchy and is defined by [46] as:

> *A diagram that depicts all of the documents for a given system and shows their relationships to one another.*

Documentation components can be regarded as **entities** where entities are parts of documentation which can be named or denoted. For example a document section, document summary, data file description and a process description are all entities.

## 3.7 Summary

After making a comparative analysis of existing impact analysis techniques it can be seen that most impact analysis techniques can only be used in the later stages of a maintenance project. However the impact information is required in the early stages of a maintenance project [27]. This early detection of the impact is a very important area that has been little explored. Few models or model analysis techniques are available for documentation impact analysis or source code impact analysis by analysing the content of related documentation.

The remainder of this thesis develops and evaluates both a new model of documentation interconnectivity and also techniques with which to analyse this model. These techniques are used to consider the possible impacts of proposed changes to software systems. The next chapter presents the underlying principles of the new model developed by the author, an informal description and a formal description of the model.

# Chapter 4

# Ripple Propagation Graph Definition

## 4.1 Introduction

In this chapter the general principles underlying the new model of ripple propagation developed are discussed. All of the necessary mathematical concepts and terminology used within this thesis are introduced before providing a formal definition of the model. The formal definition of the model is split into four model components. Each component of the model is described informally with respect to its purpose and characteristics before a formal description is presented.

## 4.2 Impact Analysis at the Documentation Level

In this thesis a new approach to impact analysis based on the analysis of documentation is presented. Glagowski in [38] argues:

> *Maintenance personnel usually begin their task by acquiring an understanding of relevant system parts by browsing the development documents. When a proposed change is formulated, the system must once again be searched for possible impacts. These information processing tasks can be extremely laborious and time consuming especially when*

*personnel changes are frequent or when the original designers no longer maintain the system leaving the job to those not yet familiar with the system. Efficient maintenance of software demands better access to software documentation than manually sifting through piles of listings, manuals and other documents.*

The model developed in this Chapter will address the above problem area, specifically to enable the detection of ripple effects in documentation. The model will apply certain principles of existing impact analysis techniques. These principles are :

1. The idea of using program dependence graphs to model data flow, except that a model description of data flow (in the documentation) will be modelled. These principles of modelling data flow are described in [14, 50].
2. The concept of program slicing will be applied to documentation. These principles are described in [92].
3. The idea of using previous source code impact information to indicate where previous impacts have occurred will be applied to documentation. These principles are described in [41].

None of these principles has been applied to documentation. The model developed will be akin to some existing work addressing impact analysis at the documentation level such as that of the following :

| | | | |
|---|---|---|---|
| Agusa, Kishimoto and Ohno | (1983) | System Level Ripple Effect Analyser | [2] |
| Chikofsky | (1983) | Requirements Impact Analysis | [16] |
| Horowitz and Williamson | (1986) | SODOS | [44] |
| Sommerville et al | (1986) | DIF | [79] |
| Delisle and Schwartz | (1986) | NEPTUNE | [22] |
| Mullin and McGowan | (1988) | FORTUNE | [63] |
| Pfleeger | (1990) | Traceability Graphs | [72] |
| Karakostas | (1990) | Teleological Models | [48] |

with the notable difference that none of the above research provided a decompositional characterisation of documentation, or any document abstraction algorithms which are useful for tracing the indirect impact of a change. In particular, few techniques provide any assistance in how to navigate through highly interconnected documentation.

## 4.3 Philosophy of the New Model

A new model called a Ripple Propagation Graph ( $RPG(\mathcal{V},\mathcal{E})$ ), is described in the remainder of this Chapter. The $RPG(\mathcal{V},\mathcal{E})$ models interconnections at the documentation level. This thesis describes a model of the data flow at the documentation level. The model concentrates on describing the processing of data at the segment level and the dependencies between these segments. There are a number of underlying features of this model, namely:

1. Segment content representation.
2. Thematic structure for characterising segments.
3. Modelling thematic properties of document segments.
4. Modelling the likelihood of ripple effects.
5. Modelling expert judgement.
6. Modelling release information.

*Segment Content Representation:* The modelling of dependencies in documentation necessitates the consideration of how a segment will be represented in the $RPG(\mathcal{V},\mathcal{E})$ model. Indirect impacts in source code can be caused by:

1. data flow value assignment;
2. control flow if assignments;
3. calling nested procedure or function; and
4. functional function assignment

as described in Chapter 3. In order to model the description of these features in documentation, a representational system that encodes the meaning of the content of a document segment by decomposition into a set of primitive features is required. The core of this approach must include some fundamental concepts which are sufficient to capture the description of indirect impacts. Two concepts are introduced:

1. A conceptual action which can be thought of as being a verb.

2. A conceptual object which can be thought of as being a noun.

In terms of data flow analysis the conceptual action would be a definition or use of a data item. The conceptual object would be a data item which can be named or denoted in a computer program. Considering this copy propagation scenario,

> *If a document describes a process where module A creates a file, which module B uses to create a second file, which module C uses then there is a dependency between module A and module C.*

here the conceptual actions are "create" and "uses" and the conceptual objects are the files created by "module A" and "module B". There is a direct relationship between the verbs in software descriptions and functions of software components. If the files have different identifiers then this dependency will not be detected in the documentation with a keyword approach. Therefore in order to detect indirect impacts in documentation it is necessary to understand the semantics of document segments.

*Thematic Structure for Characterising Segments:* An efficient scheme for classifying document segments into categories would greatly facilitate impact analysis at the documentation level. Document segment classification is a special instance of the general problem of pattern recognition and determination of clusters of related items. A documentation segment could be classified according to the conceptual object or objects described or referenced in a document segment.

The recognition of a particular conceptual object and action could be thought of as the recognition of a particular **theme**. A theme in linguistical terms is a subject written about [30]. Each theme could be thought of as being a category. Therefore by discovering the particular objects and actions it would be possible to map a document segment to one or more theme categories.

Each theme can be given a context. For example it would be useful to record the particular context of the processing of data in terms of definition or use. It would be possible to identify where themes co-occur in the system documentation in a more precise manner than using keyword indexes. Descriptions of copy propagations in the documentation can also be recorded which would facilitate a thematic data flow analysis, that is, a data flow analysis view of source code at the documentation level. As change proposals describe changes to information requirements and changes to data then it is possible to provide a mapping from a change proposal to the operational software by linking the themes in change proposals to themes recorded in segments.

*Modelling Thematic Properties of Document Segments:* The content of a document segment entity can be modelled by recording both the theme or themes in the segment and whether they co-occur in other segments. The description of copy propagations could be modelled by recording a relationship between two different segments. In particular the segment describing the source code which is the recipient of the data in the assignment can be recorded.

*Modelling the likelihood of Ripple Effects:* By assigning a number, a probability, between zero and unity to the occurrence of an event, one has a measure of the likelihood of the occurrence. For example unity indicates that the event will occur with certainty.

*Modelling Expert Judgement:* Judgement is extensively applied in searching for the solution to any significant technical problem [51]. Judgements are needed in all stages of dealing with impact analysis problems at an early stage in the process. It is better that these judgements are made by experts rather than non experts, because they have the knowledge and experience to make these judgements.

Expert judgements typically break the thought process of a maintainer into smaller parts and apply logic to integrate these parts. For example a maintainer might judge that module A could connect with module B. The maintainer might also judge that module B is connected to module C, therefore the maintainer may conclude module A may propagate an indirect impact to module C. Data and calculations may provide numerical estimates for some of the problem parts of the thought process. In addition the steps and the judgements used in an explicit thought process can be documented. If knowledge of several areas of a software system is required on a specific impact analysis problem, then it is possible that no maintainer has the expertise to make overall judgements. The problem must be decomposed and expertise used from various maintainers.

The quantification of maintainer's expert judgement concerning software interconnectivity has many advantages over words. First words are ambiguous and imprecise. For example "small chance of a ripple effect" has a large range of interpretation. On the other hand, a "10 percent chance of a ripple effect" has an unambiguous meaning, so quantification facilitates communication. Furthermore numerical expression of judgements both allow and force maintainers to be precise about what they know about software interconnectivity within a particular system and in addition to acknowledge what they do not know about it.

If multiple lines of reasoning and judgements lead to the same result then it will provide comfort in the use of this judgement as representing the current state of knowledge about the interconnectivity within a system. An extremely important part of any elicitation of expert judgement is the accompanying documentation. It is desirable to make the reasoning on which expert judgements are based as clear as possible. Any assumptions and reasoning must be recorded [51].

*Modelling Release Information:* A software release can be defined as the set of components which are affected by a group of related maintenance projects. Release information is information concerning the contents of a software release. For example which documents were changes and which modules were changed. In particular which documents affected other documents and which modules affected other modules can also be included [4].

Cooper and Munro in [21] recognise the utility of collecting change information. Cooper and Munro argue that a record of changes to a software system forms a store of experience gained by programmers while maintaining a system. It is also argued that given a method of extracting this change information it can be of use in future maintenance projects which impact similar areas of a software system.

In quantitative terms it may provide a method of ranking indirectly impacted document entities and source modules according to the probability of them being affected, based on past experience. It was observed in Chapter 3 that most existing source code based ripple effect techniques detect the worst case indirect impact. It may be useful to see what the most likely indirect impact is.

To summarise, the principles underlying the model are :

1. The source code potential indirect impacts are abstracted from the documentation to form a model of the source code interconnectivity.
2. The model attempts to re-use the information produced as a by-product of the maintenance process in order to determine the likelihood of indirect impacts. This is achieved by feeding back release information and maintainer's expert judgement concerning interconnectivity of particular systems.
3. The components of the model can be linked directly to the change proposals to provide early detection of the indirect impact of a change.
4. The model integrates concepts of deterministic models (the modelling of structure and content of documentation) with non deterministic models (the use of probablistic information).

## 4.4 Theoretical Basis of the New Model

The translation of problems into abstractions based on the use of mathematical models, provides the manipulation and reasoning about model properties in a rigorous way. Therefore the model developed in this thesis has rigorous foundation and theoretical basis. This section is intended to be a brief introduction to the basic mathematics used as the theoretical basis within this thesis. The mathematics required is as follows: set theory, bag theory, relations, functions and graph theory. Most of the mathematical definitions have been taken from [73, 94, 61].

### 4.4.1 Set Theory

A *set* is a collection of unique objects. Any *object* $x$ in a set $X$ is called an *element* or *member* of $X$; $x$ being an element of $X$ will be denoted by $x \in X$, while the opposite will be $x \notin X$. A set $X$ is a *proper subset* (proper inclusion) of set $Y$ denoted by $X \subset Y$, if and only if set $X$ is included in but not equal to set $Y$:

$$(X \subset Y) \leftrightarrow (X \subseteq Y)\&(X \neq Y)$$

A set $X$ is a *subset* (inclusion) of a set $Y$, denoted by $X \subseteq Y$, if and only if whatever is a member of $X$, is also a member of $Y$:

$$X \subseteq Y \leftrightarrow (\forall x)(x \in X \Rightarrow x \in Y)$$

$Y = X$ will denote $X$ is equal to $Y$; that is all elements of $X$ are also elements of $Y$ and all elements of $Y$ are elements of $X$. $X \cup Y$ denotes the union of sets X and Y; that is the set consisting of those elements belonging either to $X$ or to $Y$. $X \cap Y$ denotes the intersection of the sets X and Y; that is the set of elements belonging to both $X$ and $Y$. $\emptyset$ denotes an *empty set*; that is a set with no elements. The *cardinality* of set X is the number of elements in the set and is denoted as $\mid X \mid$. Often there is a large set containing all the elements of interest called the *space, universe or universal set* and is denoted by $S$.

Operations on sets provide methods for creating subsets of particular sets. The following notation $\{y \mid y \in X\}$ defines a set indicating that all $y$ in the set are members of a set $X$. The $\mid$ indicates $y \in X$ is a property of that set. This notation is used for defining properties on sets which are produced as the results of set operations.

Let $X$ and $Y$ be subsets of some universal set $S$. The set operations *intersection*, *union* and *difference* can be defined as follows:

1. The set *intersection* of $X$ and $Y$ is the set $X \cap Y$ defined by $X \cap Y = \{x \mid x \in X \wedge x \in Y\}$
2. The set *union* of $X$ and $Y$ is the set $X \cup Y$ defined by $X \cup Y = \{x \mid x \in X \vee x \in Y\}$
3. The set *difference*, of $X$ and $Y$ is the set $X - Y$ defined by $X - Y = \{x \mid x \in X \wedge x \notin Y\}$.

The term *disjoint* describes sets that have no common members. Two sets are disjoint if the intersection of the two sets is empty. A *partition* of a set X is a collection of mutually disjoint nonempty subsets of X (that is, the intersection of any pair of subsets is an empty set) whose union equals X.

Sets can be explicitly enumerated; the elements are written between the braces { and }. For example $\{4, 2, 5, 56\}$ denotes a set with elements 4,2,5 and 56.

### 4.4.2 Bag Theory

In this thesis the notion of *bag* is also used. A *bag*, is similar to the notion of set, in that it is a collection of elements. Unlike a set, however, an element can belong to a bag more than once.

$B[e]$ is used to denote the cardinality of element $e$ in bag $B$, i.e., $e \in B$ if $B[e] > 0$. The set operations intersection, union and difference carry across to bags:

1. The *bag union* of *b1* and *b2* $(b1 \cup b2)[e] = b1[e] \sqcup b2[e]$
2. The *bag intersection* of *b1* and *b2* $(b1 \cap b2)[e] = b1[e] \sqcap b2[e]$
3. The *bag difference* of *b1* and *b2* $(b1 - b2)[e] = (b1[e] - b2[e]) \sqcup 0$
4. The *bag addition* of *b1* and *b2* $(b1 + b2)[e] \hat{=} b1[e] + b2[e]$

Like sets, bags can be explicitly enumerated; the elements are written between bag brackets $\prec$ and $\succ$. For example $\prec$1,1,2,2,2$\succ$ denotes a bag with elements 1 and 2 whose frequency of occurrence are equal to 2 and 3 respectively.

The *function set* converts a bag to a set by forgetting multiplicity:

$$set\text{: } bag\ \ A \rightarrow \text{set } B.$$

For example *set* $\prec$ 1,2,2,3 $\succ$ = { 1,2,3 }.

The function $bag_n$, goes the other way converting a set into a bag:

$$bag : N \rightarrow set\ T \rightarrow bag\ T.$$

For example $bag_2\ \{1,2\} = \prec 1,1,2,2 \succ$.

In this thesis sets are converted to bags to indicate elements which may be duplicated after a bag operation is performed on a set. This conversion is denoted by changing from brackets representing a set { and } to brackets representing a bag, $\prec$ and $\succ$.

### 4.4.3 Relations

A *relation* is an association between, or property of, one or more objects. A *term* is interpreted as an object. For example $\{x \mid x \in Y\}$ defines a set with predicate indicating that all $x$ in the set are members of a set $Y$. A *sentence* describes a relation. A *predicate* is an expression that, when combined with one or more terms, forms a sentence. The notation $\{x : T \mid P(x)\}$ represents the set containing exactly those elements $x$ of type T such that, the predicate P holds (the | is pronounced 'such that'). The :T can be removed if it is obvious what the type of elements $x$ are.

### 4.4.4 Functions

A *function* $f$ from set X to set Y, written $f : X \rightarrow Y$, is a rule which associates with each element $x \in X$, one and only one element in Y. This element of Y is usually denoted by $f(x)$. Set X is called the *domain* of the function and set Y the *image set* of the function. It is not necessary for all the elements of Y to be the image of some $x \in X$ under $f$.

The *range* (or codomain) of the function is that subset of the image set $Y$ which consists of all the possible images under $f$ of all the elements of the domain X. It is denoted by $f(X)$. A function with domain X and range Y is also called a mapping or map from $X$ to $Y$.

The function $f$ is called one to one if the images of distinct elements of $X$, under $f$, are distinct elements of $Y$. Functions can be *many to one* or *one to one* relations between two sets. A *partial function* $f$ from set $X$ to set $Y$ is a function which maps from a subset of $X$ to $Y$.

The set of all such partial functions from a set X to a set Y is denoted as $X \nrightarrow Y$. These functions are partial as there is no necessity for all the members of set X to be mapped to members of set Y, the function may be defined only on a subset of set X. Sometimes it is important that the domain of a function $f : X \nrightarrow Y$ is the whole of X. This type of function is called a *total function* from $X$ to $Y$ and it is denoted as $f : X \rightarrow Y$.

For any function, it is useful to record its domain and range. The *signature* of a function is written with the domain and range sets separated by an arrow. For example *slice:* $A \rightarrow B$. The domain of a function of more than one argument is given as a list of all of the argument sets separated by crosses. For example *slice:* $A \times B \rightarrow C$.

### 4.4.5 Graph Theory

A *graph* consists of a set of elements called *vertices*, and a set of arcs connecting the vertices called *edges*. Formally a graph is represented by $G(\mathcal{V},\mathcal{E})$, where $V$ is a set of vertices $\{n_1, ..., n_\xi\}$ and $\mathcal{E}$ is the set of ordered pairs called edges, $\{(x, y) \mid (x, y) \in \mathcal{V} \times \mathcal{V}\}$. The number of vertices is represented by $|\mathcal{V}|$ and the number of edges by $|\mathcal{E}|$. Given any graph edge $(v_1, v_2)$, then $v_1$ vertex is called the **start vertex** of the edge and the $v_2$ is called the **stop vertex** of the edge.

The graph $G_s(\mathcal{V}_s,\mathcal{E}_s)$ is a *subgraph* of $G(\mathcal{V},\mathcal{E})$ if the vertices in $\mathcal{V}_s$ are also $\mathcal{V}$ and the edges in $\mathcal{E}_s$ are also in $\mathcal{E}$. This subgraph relationship is denoted as $G_s(\mathcal{V}_s,\mathcal{E}_s) \sqsubseteq G(\mathcal{V},\mathcal{E})$.

A *strict subgraph* of $G(\mathcal{V},\mathcal{E})$ is a subgraph which does not contain all the vertices and edges of $G(\mathcal{V},\mathcal{E})$. This subgraph relationship is denoted as $G_{ss}(\mathcal{V}_{ss},\mathcal{E}_{ss}) \sqsubset G(\mathcal{V},\mathcal{E})$. The graph $G_{ps}(\mathcal{V}_{ps},\mathcal{E}_{ps})$ is a *proper subgraph* of $G(\mathcal{V},\mathcal{E})$ if $G_s(\mathcal{V}_s,\mathcal{E}_s) \sqsubset G(\mathcal{V},\mathcal{E})$ and if in the graph $G_{ps}(\mathcal{V}_{ps},\mathcal{E}_{ps})$ there is no edges in $\mathcal{E}$ which has only the start vertex or stop vertex in $\mathcal{V}_{ps}$, but not both the start vertex or stop vertex in $\mathcal{V}_{ps}$.

If two vertices are indirectly connected through one or more intermediate vertices, then there is said to be a path between the two vertices. A *path* is a sequence of vertices such that successive pairs of vertices are *adjacent* (connected by an edge). If in a given path each vertex is visited only once the path is called a *simple path.* The *neighbours* of a vertex n is the set of vertices adjacent to n. A *directed path* is a sequence of vertices such that successive pairs of vertices are adjacent and the order of the pairs indicates the direction of the path.

Graphs have two kinds of edges: *directed* and *undirected.* A directed edge indicates that information can flow only in the direction of the edge. An undirected edge indicates that information can flow both ways. An *undirected edge* can be defined as the pair of directed edges $(v_1, v_2)$ and $(v_2, v_1)$ A graph containing only directed edges is called a *directed graph.* Similarly a graph containing only undirected edges is called an *undirected graph.* A *multigraph* is a set of vertices and a bag of edges so there may be two or more edges joining two given vertices in the same direction. A *simple graph* has at most one edge joining the vertices. A graph $G$ is termed *connected* if for every pair of vertices $u$ and $v$ of $\mathcal{V}(G)$, $u$ is connected to $v$, otherwise $G$ is said to be *disconnected.* A *circuit* is a path whose first and last vertices are the same. A *simple circuit* is a circuit where except for the first and last vertices, no vertex appears more than once. A *cycle* in a graph is a simple circuit. A graph that has no cycles is said to be an *acyclic graph.*

The *valency* of a vertex $v$ of $G$ is the number of edges incident at $v$, i.e., the number of edges $(v', v)$ in $\mathcal{E}$ for each $v'$ in V. This is written as $p(v)$. Any vertex of valency 0 is called an *isolated vertex* and any vertex with valency 1 is called a *terminal vertex.*

In many applications of graph theory it is useful to assign weights to graph edges. The weight can represent the carrying capacity of an edge or the strength of the relationship between two vertices. Such a graph is called a *parameterised graph.* An edge belonging to a parameterised graph can be defined as a triple $(n_i, n_j, w)$ where $w$ is the weight. A graph may also have information associated with each member of the vertex set. Such a graph is called a *labelled graph*, which is a pair $(G, \theta)$ where $G$ is the graph and $\theta$ is the label information.

Given a directed graph $G(\mathcal{V}, \mathcal{E})$, the *transitive closure* $C(\mathcal{V}, \mathcal{E})$ of $G$ is a directed graph such that there is an edge $(v, w)$ in $C$ if and only if there is a directed path from $v$ to $w$ in $G$.

In this thesis the model analysis techniques introduced in Chapter 6 make extensive use of the analysis of graph theory models. The following describes the graph operations used for this analysis. *Graph Union* is the creation of a graph that contains all the vertices and edges from two given graphs. The graph union operation is denoted by :

$$\textit{Graph Union } G_1(\mathcal{V}_1,\mathcal{E}_1) \sqcup G_2(\mathcal{V}_2,\mathcal{E}_2)$$

and can be specified as follows :

$$\textit{Graph Union: Graph} \times \textit{Graph} \rightarrow \textit{Created Graph}$$
$$\text{where the created graph is } (\mathcal{V}_1 \cup \mathcal{V}_2, \mathcal{E}_1 \cup \mathcal{E}_2)$$

The notation *Graph : Graph* $\times$ *Graph* $\rightarrow$ *Created Graph* represents a total function. The graph union operation is based on the set union property and ensures duplicate vertices and edges are removed.

$$\textit{Distributed Graph Union } G_1(\mathcal{V}_1,\mathcal{E}_1) \sqcup \ldots \sqcup G_n(\mathcal{V}_n,\mathcal{E}_n)$$

and can be specified as follows :

*Distributed Graph Union: Graphs* $\rightarrow$ *Created Graph*
where the created graph is $(\mathcal{V}_1, \cup ... \cup \mathcal{V}_n, \mathcal{E}_1 \cup ... \cup \mathcal{E}_n)$

*Graph Intersection* is the process of creating a graph from two graphs in terms of characteristics which are common to both graphs. The graph intersection operation is denoted by the following :

*Graph Intersection* $G_1(\mathcal{V}_1,\mathcal{E}_1) \sqcap G_2(\mathcal{V}_2,\mathcal{E}_2)$

and can be specified as :

*Graph Intersection: Graph* $\times$ Graph $\rightarrow$ *Created Graph*
where the created graph is $(\mathcal{V}_1, \cap \mathcal{V}_2, \mathcal{E}_1 \cap \mathcal{E}_2)$

The *Distributed Graph Intersection* is the process of creating a graph from two or more graphs in terms of characteristics which are common to these graphs. The graph intersection operation is denoted by the following :

*Distributed Graph Intersection* $G_1(\mathcal{V}_1,\mathcal{E}_1) \sqcap ... \sqcap G_n(\mathcal{V}_n,\mathcal{E}_n)$

and can be specified as :

*Distributed Graph Intersection:* Graphs $\rightarrow$ *Created Graph*
where the created graph is $(\mathcal{V}_1, \cap ... \mathcal{V}_n, \mathcal{E}_1 \cap ... \mathcal{E}_n)$

Several techniques of abstracting subgraphs from a graph model are developed in Chapter 6. Each of these techniques are based on combinations of these graph intersection and graph union operations.

### 4.4.6 Summary of the Theoretical Basis

In this thesis graphs are used to model and reason about components of software systems where the vertices are the components and the edges are the relationships between the components. The edges are weighted to give an indication of the probability of one component being affected with an inconsistency by a change to another component. This weight is derived from quantitative expert judgement concerning potential indirect impacts and also from release information. Sets are used to describe the vertices and edges which make up the graph models. The graph models will be analysed by extracting subgraphs using combinations of graph union and graph intersection operations. Relations are used to define properties on the edge sets and vertex sets during various steps of the graphs analysis. The bag theory is used to represent vertices which are impacted more than once by a change.

## 4.5 Model Definition

In this thesis a graph called a **Ripple Propagation Graph**[1] $RPG(\mathcal{V},\mathcal{E})$ is used to record the interconnection between document entities and the interconnection between document entities and source code. The $RPG(\mathcal{V},\mathcal{E})$ is an **interconnection graph** of the form :

DEFINITION 1. Let *the Ripple Propagation Graph* = (*Entities, Dependencies*) where,

1. a set of *Entities* that are the components of interconnection, i.e., a set of graph vertices $\{n_1, ..., n_\xi\}$ called a vertex set;
2. a set of *Dependencies* that define the interconnection that exists among entities, i.e., a set of ordered pairs representing the graph edges $\{(n_i, n_j) \mid n_i, n_j \in \textit{Entities}\}$ called an edge set.

Some graph edges are parameterised therefore a parameter is associated with a graph edge of the form :

$$\{(n_i, n_j, \textit{edge parameter}) \mid n_i, n_j \in \textit{Entities}\}$$

The $RPG(\mathcal{V},\mathcal{E})$ consists of four independent and specialised forms of interconnection graph which comply with the above specification, they are the following :

[1]Any of the author's notations or abbreviations are also defined in the glossary of notation given in Appendix A

- The *Hierarchical Interconnection Graph ($HIG(\mathcal{V},\mathcal{E})$)* which models the decompositional structure of documentation;
- The *Thematic Interconnection Graph ($TIG(\mathcal{V},\mathcal{E})$)* which models the content of documentation;
- The *Source Attributes Graph ($SAG(\mathcal{V},\mathcal{E})$)* which models the source code entities described within document entities;
- The *Weighted Interconnection Graph ($WIG(\mathcal{V},\mathcal{E})$)* which models the probability of ripple propagation between document entities and between source code entities.

**DEFINITION 2.** Let the $RPG(\mathcal{V},\mathcal{E}) = HIG(\mathcal{V},\mathcal{E}) \sqcup TIG(\mathcal{V},\mathcal{E}) \sqcup SAG(\mathcal{V},\mathcal{E}) \sqcup WIG(\mathcal{V},\mathcal{E})$, a connected graph. Each of these four *interconnection graphs* is a subgraph of the $RPG(\mathcal{V},\mathcal{E})$ denoted as:

$$\textit{Interconnection Graph} \sqsubset RPG(\mathcal{V},\mathcal{E}).$$

The distributed intersection of the four specialised interconnection graphs, $HIG(\mathcal{V},\mathcal{E}) \sqcap TIG(\mathcal{V},\mathcal{E}) \sqcap SAG(\mathcal{V},\mathcal{E}) \sqcap WIG(\mathcal{V},\mathcal{E})$ will produce a graph $G' = (\mathcal{V}',\mathcal{E}')$ where $\mathcal{V}'$is a set of segment entities with valency equal to 0, that is, they are all terminal vertices. The set $\mathcal{V}' \subset RPG(\mathcal{V},\mathcal{E})$, in other words the segment entity is the entity which is common to all four interconnection graphs. The set $RPG(\mathcal{E}\prime)$ is an empty set after the distributed intersection operation has been performed.

Before formally defining each interconnection graph of the $RPG(\mathcal{V},\mathcal{E})$ , the distinctive features of each subgraph are discussed. The $RPG(\mathcal{V},\mathcal{E})$ is defined set theoretically in the same fashion as in DEFINITION 1. This will allow properties to be defined on the sets and will be of use in later chapters when defining operations on the $RPG(\mathcal{V},\mathcal{E})$ . The properties will show the conditions which will be true after a particular graph operation has been performed on the $RPG(\mathcal{V},\mathcal{E})$ .

### 4.5.1 Hierarchical Interconnection Graph

The **Hierarchical Interconnection Graph** ($HIG(\mathcal{V},\mathcal{E})$) is a specialised form of interconnection graph which records dependencies between any entities within a documentation hierarchy. The graph provides a compositional characterisation of the hierarchy in terms of the different document entities it contains. The $HIG(\mathcal{V},\mathcal{E})$ also records each document entity type from the large granularity to the smallest granularity. The $HIG(\mathcal{V},\mathcal{E})$ contains segment entities which are the smallest document entities. Any entity in the hierarchy above a segment entity is called a **composite entity**. The *decompositional* dependencies exist between $HIG(\mathcal{V},\mathcal{E})$ entities and are the following:

- `consists of` *dependency*
  this dependency is associated with an edge of the form
  *(composite entity,composite entity).*

- `has segment` *dependency*
  this dependency is associated with an edge of the form
  *(composite entity,segment entity).*

- `composite entity type` *dependency*
  this dependency is associated with an edge of the form
  *(composite entity,composite type entity).*

- `segment entity type` *dependency*
  this dependency is associated with an edge of the form
  *(segment entity,segment type entity).*

- `composite entity type description` *dependency*
  this dependency is associated with an edge of the form
  *(composite type entity,composite type description entity).*

- `segment entity type description` *dependency*
  this dependency is associated with an edge of the form
  *(segment type entity,segment type description entity).*

Each of the dependencies above shows what the document hierarchy consists of and how it is structured. For example the `consists of`[2] dependency shows that the start vertex of this edge represents a document entity which contains another document entity represented by the stop vertex. The `has segment` dependency records that a particular document entity represented by the start vertex of this edge contains a segment entity represented by the stop vertex. These dependencies provide information regarding the role of each document entity in the hierarchy. For example the `composite entity type` dependency records the type of the composite entity. The `segment entity type` dependency records the type of the segment entity. The `composite entity type description property` and `segment entity description property` dependencies record a description of the type of entities within the hierarchy. The $HIG(\mathcal{V}, \mathcal{E})$ can be specified set theoretically as follows :

**DEFINITION 3.** Let the *Hierarchical Interconnection Graph = (Documentation Entities, Decompositional Dependencies)* be a directed and labelled graph where,

*Documentation Entities =*

*composite entities* ∪ *segment entities* ∪ *composite type entities* ∪
*segment type entities* ∪ *segment type description entities* ∪
*composite type description entities*

*Decompositional Dependencies =*

*consists of dependencies* ∪ *has segment dependencies* ∪
*composite entity type dependencies* ∪ *segment entity type dependencies* ∪
*composite entity type description dependencies* ∪
*segment entity type description dependencies*

*and,*

*consists of dependencies=*

{*(a,b)* | *a* ∈ *composite entities, b* ∈ *composite entities* }

[2]Words written in typeface highlight graph dependencies

*has-segment dependencies=*

$\{(a,b) \mid a \in$ *composite entities,* $b \in$ *segment entities* $\}$

*composite entity type dependencies=*

$\{ (a,b) \mid a \in$ *composite entities,* $b \in$ *composite type entities*$\}$

*segment entity type dependencies=*

$\{(a,b) \mid a \in$ *segment entities,* $b \in$ *segment type entities*$\}$

*composite entity type description dependencies=*

$\{(a,b) \mid a \in$ *composite type entities,* $b \in$ *composite type description entities*$\}$

*segment entity type description dependencies=*

$\{(a,b) \mid a \in$ *segment type entities,* $b \in$ *segment type description entities*$\}$

*and the Hierarchical Interconnection Graph* $\sqsubset RPG(\mathcal{V},\mathcal{E})$.

### 4.5.2 Thematic Interconnection Graph

The **thematic structure** of documentation, models the organisation of themes in the documentation. The Thematic Interconnection Graph ( $TIG(\mathcal{V},\mathcal{E})$) is a specialised form of interconnection graph which records dependencies between segment entities based on the content of each segment entity. The $TIG(\mathcal{V},\mathcal{E})$ records the thematic structure and the semantic role of each document entity within a documentation system. This is achieved by modelling the description of data flow within a software system. The *thematic* dependencies are:

- `has theme vertex` *dependency*
  this dependency is associated with an edge of the form
  *(segment entity,theme vertex entity).*

- `has theme` *dependency*
  this dependency is associated with an edge of the form
  *(theme vertex entity,theme code entity).*

- `co-occurs` *dependency*
  this dependency is associated with an edge of the form
  *(theme vertex entity,theme vertex entity).*

- `copy propagation description` *dependency*
  this dependency is associated with an edge of the form
  *(theme vertex entity,theme vertex entity).*

- `definition use description chain` *dependency*
  this dependency is associated with an edge of the form
  *(theme vertex entity,theme vertex entity).*

- `thematic context property` *dependency*
  this dependency is associated with an edge of the form
  *(theme vertex entity,context entity).*

- `theme category` *dependency*
  this dependency is associated with an edge of the form
  *(theme code entity,category description entity).*

Each of these dependencies shows the content of the segment entities. This set of dependencies provides information to logically deduce the indirect impacts rather than just the direct impacts of a change. The `has theme vertex` dependency records that a segment entity represented by the start vertex of the edge contains a theme represented by the stop vertex of the edge. The `has theme` dependency records that theme vertex represented by the start vertex of the edge has a particular theme associated with it which is represented by the stop vertex of the edge. The `co-occurs` dependency records that a theme vertex represented by the start vertex of the edge is associated with another theme vertex represented by the stop vertex of the edge because both theme vertices have the same theme. The `copy propagation description` dependency records that a theme vertex represented by the start vertex of the edge is connected to a segment which describes data transmitted to another part of a source code system. The segment entity describing the receipt of data has a theme vertex entity represented by the stop vertex of the edge. The `definition use description chain` dependency records that there is a description of a definition use chain represented by the start vertex of the edge with the represented by the stop vertex of the edge. The `thematic context property` dependency records that the theme vertex represented by the start vertex of the edge has a particular definition or use property with the represented by the stop vertex of the edge. The `theme category` dependency records that a theme code represented by the start vertex of the edge is associated with a thematic category with the represented by the stop vertex of the edge. The $TIG(\mathcal{V}, \mathcal{E})$ can be specified set theoretically as follows :

**DEFINITION 4.** Let the *Thematic Interconnection Graph = (Thematic Interconnection Entities, Thematic Dependencies)* be a directed and labelled graph where,

*Thematic Interconnection Entities =*

*segment entities* ∪ *theme vertex entities* ∪ *theme code entities* ∪
*thematic context entities* ∪ *category description entities*

*Thematic Dependencies =*

*has theme vertex dependencies* ∪ *has theme dependencies* ∪
*co-occurs dependencies* ∪ *copy propagation description dependencies* ∪
*definition use description chain dependencies* ∪
*thematic context property dependencies* ∪ *theme category dependencies*

*and,*

*has theme vertex dependencies =*

{*(a,b)* | *a* ∈ *segment entities*, *b* ∈ *theme vertex entities*}

*has theme dependencies=*

{*(a,b)* | *a* ∈ *theme vertex entities*, *b* ∈ *theme code entities*}

*co-occurs dependencies =*

{*(a,b)* | *a,b* ∈ *theme vertex entities*} *copy propagation description dependencies=*

$\{(a,b) \mid a,b \in$ *theme vertex entities*$\}$

*definition use description chain dependencies=*

$\{(a,b) \mid a,b \in$ *theme vertex entities*$\}$

*thematic context property dependencies=*

$\{(a,b) \mid a \in$ *theme vertex entities*, $b \in$ *context entities*$\}$

*theme category dependencies=*

$\{(a,b) \mid a \in$ *theme code entity*, $b \in$ *category description entities*$\}$

*and the Thematic Interconnection Graph* $\sqsubset RPG(\mathcal{V},\mathcal{E})$ .

### 4.5.3 Source Code Association Graph

The **Source code Association Graph** $SAG(\mathcal{V},\mathcal{E})$ is a specialised form of interconnection graph which records dependencies between segment entities and source code entities. The dependencies are based on the source code entities which are explicitly or implicitly described in a segment entity. This graph provides information regarding the source code components of a system which are associated with particular document entities within a system.

The *document entity and module entity association* dependencies are:

- `segment describes part of module` *dependency*
  this dependency is associated with an edge of the form
  *(segment entity,module entity).*

- `module type` *dependency*
  this dependency is associated with an edge of the form
  *(module entity,language type entity).*

- `module belongs to system` *dependency* this dependency is associated with an edge of the form *(module entity,system entity).*

- `associated system` *dependency*
  this dependency is associated with an edge of the form *(system entity,system entity).*

- `module test required` *dependency*
  this dependency is associated with an edge of the form *(module entity,module test entity).*

- `system test required` *dependency*
  this dependency is associated with an edge of the form *(module entity,system test entity).*

- `module uses data file` *dependency*
  this dependency is associated with an edge of the form *(module entity,data file entity).*

- `module uses job control language` *dependency*
  this dependency is associated with an edge of the form *(module entity,jcl entity).*

- `module uses map base` *dependency*
  this dependency is associated with an edge of the form *(module entity,map base entity).*

- `system supplies data to` *dependency*
  this dependency is associated with an edge of the form

*(system entity,system entity).*

- `system data received from` *dependency*
  this dependency is associated with an edge of the form
  *(system entity,system entity).*

- `module uses data dictionary region` *dependency*
  this dependency is associated with an edge of the form
  *(module entity,data dictionary region area entity).*

- `segment describes part of system` *dependency*
  this dependency is associated with an edge of the form
  *(segment entity,system entity).*

Each of these dependencies shows the relationship between segment entities and source code modules. The `segment describes part of module` dependency records that a segment entity represented by the start vertex of the edge is associated with a source code **module** represented by the stop vertex of the edge. This association exists because the segment describes part or all of the source code module. The `module type` dependency records that a source code module is represented by the start vertex of the edge which is associated with module type entity.

The module type entity is represented by the stop vertex of the edge which indicates the module type i.e, the programming language in which the module is written. The `module belongs to system` dependency records that a source code module represented by a start vertex of the edge is associated with a system represented by the stop vertex of the edge. The `associated system` dependency records that a system is represented by the start vertex of the edge and is associated to another system through a system interface. The other system is represented by the stop vertex of the edge.

The remaining dependencies show relationships at the source level which could be detected from the documentation. The `module test required` dependency records that a source code module is represented by the start vertex of the edge and is associated with a unit test represented by the stop vertex of the edge. The `system test required` dependency records that a module is represented by the start vertex of the edge and is associated with a system test represented by the stop vertex of the edge. The `module uses data file` dependency records that a module is represented by the start vertex of the edge and is associated with a data file represented by the stop vertex of the edge. The `module uses job control language` dependency records that a module is represented by the start vertex of the edge and is associated with a job control language file represented by the stop vertex of the edge. The `module uses map base` dependency records that a module is represented by the start vertex of the edge and is associated with a module map base mapping between data file and physical storage) represented by the stop vertex of the edge.

The `system supplies data to` dependency records that a system comprising of modules is represented by the start vertex of the edge and is associated with another system comprised of modules represented by the stop vertex of the edge because data is transmitted through an interface from one system to another. The `system data received from` dependency records that a system comprising of modules is represented by the start vertex of the edge and is associated with another system comprising of modules represented by the stop vertex of the edge. This is because the system represented by the start vertex receives data through an interface from the system represented by the stop vertex of the edge. The `segment uses data dictionary region` dependency records that a segment entity is represented by the start vertex of the edge and is associated with a component of a data dictionary represented by the stop vertex of the edge. This association exists because the segment entity refers to the data dictionary. The `segment describes part of system` dependency records that a segment entity is represented by the start vertex of the edge and is associated with a group of modules whose system name is represented by the stop vertex of the edge. The $SAG(\mathcal{V}, \mathcal{E})$ can be specified set theoretically as follows :

DEFINITION 5. Let the *Source Association Graph = (Source Code Entities, Document Entity and Module Association Dependencies)* be a directed and labelled graph where,

*Source Code Entities =*

*segment entities* $\cup$ *module entities* $\cup$ *system entities* $\cup$ *module test entities* $\cup$
*data file entities* $\cup$ *jcl entities* $\cup$ *map base entities* $\cup$
*system test entities* $\cup$ *data dictionary region area entities* $\cup$ *language type entities*

*Document Entity and Module Association Dependencies* =

*segment describes part of module dependencies* ∪ *module type dependencies* ∪
*module belongs to system dependencies* ∪ *associated system dependencies* ∪
*module test required dependencies* ∪ *system test required dependencies* ∪
*module uses data file dependencies* ∪
*module uses job control language dependencies* ∪
*module uses map base dependencies* ∪ *system supplies data to dependencies* ∪
*system data received from dependencies* ∪
*segment uses data dictionary region dependencies* ∪
*segment describes part of system*

*and,*

*segment describes part of module dependencies*=

$\{(a,b) \mid a \in \textit{segment entities},\ b \in \textit{module entities}\}$

*module type dependencies*=

$\{(a,b) \mid a \in \textit{module entities},\ b \in \textit{module type entities}\}$

*module belongs to system dependencies*=

$\{(a,b) \mid a \in \textit{module entities},\ b \in \textit{system entities}\}$

*associated system dependencies* =

$\{(a,b) \mid a\ ,b \in \textit{system entities}\}$

*module test required dependencies*=

$\{ (a,b) \mid a \in \textit{module entities},\ b \in \textit{module test entities}\}$

*system test required dependencies*=

$\{(a,b) \mid a \in \textit{module entities},\ b \in \textit{system test entities}\}$

*module uses data file dependencies*=

$\{(a,b) \mid a \in \textit{module entities},\ b \in \textit{data file entities}\}$

*module uses job control language dependencies*=

$\{(a,b) \mid a \in \textit{module entities},\ b \in \textit{jcl entities}\}$

*module uses map base dependencies=*

$\{(a,b) \mid a \in \textit{module entities},\ b \in \textit{map base entities}\}$

*system supplies data to dependencies=*

$\{(a,b) \mid a,b \in \textit{system entities}\}$

*system data received from dependencies=*

$\{(a,b) \mid a,b \in \textit{system entities}\}$

*segment uses data dictionary region area dependencies=*

$\{ (a,b) \mid a \in \textit{segment entities},\ b \in \textit{data dictionary region area entity}\}$

*segment describes part of system dependencies=*

$\{(a,b) \mid a \in \textit{segment entities},\ b \in \textit{system entities}\}$

*and the Source Code Association Graph* $\sqsubset RPG(\mathcal{V},\mathcal{E})$ .

### 4.5.4 Weighted Interconnection Graph

The **Weighted Interconnection Graph** ($WIG(\mathcal{V},\mathcal{E})$) is a specialised form of interconnection graph which provides information regrading the likelihood of a indirect impact between two entities on the graph model. This is achieved by maintaining a probability of indirect impact as a parameter on the graph edge between two entities forming a weighted edge. The *weighted* dependencies are:

- `document potential impact` *dependency*
  this dependency is associated with an edge of the form
  *(segment entity,segment entity,release information parameter).*

- `module potential impact` *dependency*
  this dependency is associated with an edge of the form
  *(module entity,module entity,release information parameter).*

- `document expert judgement` *dependency*
  this dependency is associated with an edge of the form
  *(segment entity,segment entity,expert judgement parameter).*

- `module expert judgement` *dependency*
  this dependency is associated with an edge of the form
  *(module entity,module entity,expert judgement parameter).*

The `document potential impact` dependency represents a dependency between a segment entity represented by the start vertex of the edge and a segment entity represented by the stop vertex of the edge. The parameter associated with the edge represents the probability of a indirect impact being propagated from the start vertex to the stop vertex if the start vertex is modified, based on previous indirect impacts recorded as release information.

The `module potential impact` dependency represents a dependency between a module entity represented by the start vertex of the edge and a module entity represented by the stop vertex of the edge. The parameter associated with the edge represents the probability of a indirect impact being propagated from the start vertex to the stop vertex, if the start vertex is modified based on previous indirect impacts recorded as release information.

In the case of the `document potential impact` dependency and the `module potential impact` dependency it is necessary to associate information with edge between the stop and start vertices.

- `Release Information` *Parameter*
  this parameter is associated with the following piece of information

  percentage chance of propagation :

The `expert judgement` dependency represents a historical dependency between a module entity represented by the start vertex of the edge and a module entity represented by the stop vertex of the edge. The parameter associated with the edge represents the probability of a ripple effect being propagated from the start vertex to the stop vertex if the start vertex is modified.

- `Expert Judgement` *Parameter*

  this parameter is associated with the following information :

  judgement number,
  release number,
  project number,
  quantitative judgement,
  qualitative judgement,
  judgement reason,
  judgement assumption,
  judgement made by,
  date

The `expert judgement` parameters record the judgement number, the software release number and project number which the judgement was based on. A quantitative judgement is recorded in the form of a percentage. The corresponding qualitative judgement, the reason behind the judgement and any assumptions which are made are also recorded. One judgement is recorded for each pair of entities in the model. Several maintainers will estimate the potential indirect impact and a single judgement will be arrived by consensus and then recorded. The person who controls this process will have his or her name associated with the judgement. The $WIG(\mathcal{V}, \mathcal{E})$ can be specified set theoretically as follows :

**DEFINITION 6.** Let the *Weighted Interconnection Graph = (Weighted Interconnected Entities, Weighted Dependencies)* be a directed, parameterised and labelled graph where,

*Weighted Interconnected Entities =*
  *segment entities* ∪ *module entities*

*Weighted Dependencies =*
  *document potential impact dependencies* ∪ *module potential impact dependencies*

*document expert judgement dependencies* $\cup$ *module expert judgement dependencies*

*Edge Parameters =*

*release information parameters* $\cup$ *expert judgement parameters*

*and an*

*expert judgement parameter=*

$a \in N$,

$b \in$ *real numbers,*

$c \in$ *real numbers,*

$d \in$ *real numbers* $\bullet$ $a => 0$ *and* $a <= 100$,

$e \in$ *character strings,* $f \in$ *character strings,*

$g \in$ *character strings,* $h \in$ *character strings,*

$i \in$ *character strings.*

*and a*

*release information parameter=*

$a \in$ *real numbers* $\bullet$ $a >= 0$ *and* $a <= 100$,

*These parameters are used to label the following edges,*

*document potential impact dependencies=*

$\{(a,b,p) \mid$ $a,b \in$ *segment entities,*
$p \in$ *release information parameters* $\}$

*module potential impact dependencies=*

$\{(a,b,p) \mid$ $a,b \in$ *module entities,*
$p \in$ *release information parameters* $\}$

*module expert judgement dependencies=*

$\{(a,b,p) \mid$ $a,b \in$ *module entities,*
$p \in$ *expert judgement parameters* $\}$

*document expert judgement dependencies*=

$\{(a,b,p) \mid$ $a,b \in$ *segment entities*

$p \in$ *expert judgement parameters* $\}$

*and the Weighted Interconnection Graph* $\sqsubset$ $RPG(\mathcal{V},\mathcal{E})$ .

## 4.6 Summary

In this chapter the principles underlying the new model developed have been presented. The distinctive features of the approach to modelling ripple propagation have been discussed before providing a formal definition of the model with a mathematically sound basis.

The model is based on graph theory and records the inter- and intra- connections between documentation entities by considering the description of source code data flow within these document entities. In particular the model records the probability of potential indirect impacts between document entities. The probabilities are recorded on the edges in the graph model and are derived from expert judgements concerning system interconnectivity and previous system release information. The next chapter shows how the $RPG(\mathcal{V},\mathcal{E})$ model can be constructed from the documentation system, the understanding of information produced during the maintenance process, the expert judgements on potential indirect impacts and also the previous system release information.

Chapter 5

# Ripple Propagation Graph Construction

## 5.1 Introduction

In this chapter techniques for constructing ripple propagation graphs are presented. Each technique is based on a mapping from the data extracted from the documentation and from the maintainer's understanding of system structure, to the $RPG(\mathcal{V},\mathcal{E})$. The $RPG(\mathcal{V},\mathcal{E})$ construction technique is decomposed into four main graph construction techniques in order to make the building of the model easier, by considering each subgraph of the model independently. Each technique is described by an algorithm after the principles of the technique have been discussed.

Some of the graph construction techniques also make use of certain guidelines. These guidelines are not merely checklists but relate to the structure of the graph model and the source of data. The guidelines take the form of situations which can occur when collecting data to construct the model and contain the consequential reaction to be made.

**Notation for Describing the Algorithms:**

In addition to describing the graph construction techniques, simple pseudocodes are included for the algorithms. The pseudocodes are included to enhance the description of the graph construction techniques. For the algorithms a Pascal like language has been used with **Begin** ... **End** constructs.

## 5.2 Philosophy of the Graph Construction Techniques

The data for the model comes from different sources and it would be complicated to provide a single graph construction technique. Therefore four different graph construction techniques are developed. Two additional techniques are also developed for maintaining existing graphs (addition and deletion of edges and vertices). The six techniques are :

1. The **HIG Crystallisation** technique which is used for constructing the $HIG(\mathcal{V}, \mathcal{E})$.
2. The **TIG Crystallisation** technique which is used for constructing the $TIG(\mathcal{V}, \mathcal{E})$.
3. The **Graph Annotation** technique which is used for constructing the $SAG(\mathcal{V}, \mathcal{E})$.
4. The **Graph Parameterisation** technique which is used for constructing the $WIG(\mathcal{V}, \mathcal{E})$.
5. The **Graph Splicing** technique which is used for constructing the $RPG(\mathcal{V}, \mathcal{E})$ by combining subgraphs.
6. The **Graph Clipping** technique removes $RPG(\mathcal{V}, \mathcal{E})$ components.

The $RPG(\mathcal{V}, \mathcal{E})$ can be constructed incrementally by building a subgraph and adding it to the main $RPG(\mathcal{V}, \mathcal{E})$. The table 5.1 shows the system documentation, the maintainer's understanding of possible ripple effects, the data extracted and the subgraph which is constructed from this data.

The first column in the table shows the sources of data which are used to build the $RPG(\mathcal{V}, \mathcal{E})$ model. The second column shows exactly the data which is extracted from the data sources and the third column shows which type of subgraph of the $RPG(\mathcal{V}, \mathcal{E})$ the data is used to construct.

| Source of Data | Data for the Model | $RPG(\mathcal{V},\mathcal{E})$ Model |
|---|---|---|
| System Documentation | structure of documentation | $HIG(\mathcal{V},\mathcal{E})$ |
| | content of documentation | $TIG(\mathcal{V},\mathcal{E})$ and $SAG(\mathcal{V},\mathcal{E})$ |
| Release Information | modules changed | $WIG(\mathcal{V},\mathcal{E})$ |
| | document segments changed | $WIG(\mathcal{V},\mathcal{E})$ |
| Expert Judgement | modules that will change | $WIG(\mathcal{V},\mathcal{E})$ |
| | documents that will change | $WIG(\mathcal{V},\mathcal{E})$ |

Table 5.1: Mappings between Data and the $RPG(\mathcal{V},\mathcal{E})$

The construction techniques presented in this Chapter are augmented by a set of guidelines which are to be carried out manually. There are no provisions for automating these guidelines.

## 5.3 HIG Crystallisation

**The Problem:** *To extract the $HIG(\mathcal{V},\mathcal{E})$ from a documentation system. The $HIG(\mathcal{V},\mathcal{E})$ can be partially constructed or fully constructed.*

### 5.3.1 Technique Description

The aims of this technique are firstly, to identify the composite entities and to record the hierarchical structure of the composite entities that form a documentation system. The second aim is to factor the document entities into standard sized units of analysis called segment entities.

The technique builds the $HIG(\mathcal{V},\mathcal{E})$ by adding the $HIG(\mathcal{V},\mathcal{E})$ edges and vertices which are defined in Chapter 4, definition 3. The document entities are decomposed from a document into segment entities. This is called **composite entity factoring**. Each entity in the hierarchy is allocated an entity type in order to build up a picture of the composition of the documentation.

The technique can be described in the following way,

*HIG Crystallisation: Documentation $\rightsquigarrow$ HIG$(\mathcal{V},\mathcal{E})$*

This operation can be denoted as :

*HIG Crystallisation* $(Documentation, HIG(\mathcal{V}, \mathcal{E})) \underline{\triangle}$ :

```
Algorithm HIG Crystallisation
Input: Documentation
Output: $HIG(\mathcal{V}, \mathcal{E})$

Begin (Crystallisation of the $HIG(\mathcal{V}, \mathcal{E})$ )
    Let $\mathcal{V}$ be a set of vertices, $\{n_1, ..., n_\xi\}$, and
    Let $\mathcal{E}$ is the set of edges, $\{(x, y) \mid (x, y) \in \mathcal{V} \times \mathcal{V}\}$
    {composite entity factoring - this models the hierarchy and segments}
    parse-documentation-structure and:
    create the set Documentation Entities {Chapter 4, Definition 3}
    create the set Decompositional Dependencies {Chapter 4, Definition 3}
    Let $\mathcal{V}$ = Documentation Entities
    Let $\mathcal{E}$ = Decompositional Dependencies
    construct-graph $HIG(\mathcal{V}, \mathcal{E})$
End (Crystallisation of the $HIG(\mathcal{V}, \mathcal{E})$ )
```

**Remarks.**
*Parse-documentation-structure* is a function which extracts the structural features of the documentation and maps it to the sets $\mathcal{V}$ and $\mathcal{E}$. This function will have to be instantiated for a particular documentation method. *Construct-graph* is a function which creates a graph from the two sets $(\mathcal{V}', \mathcal{E}')$. The two sets $\mathcal{V}$ and $\mathcal{E}$ separately do not form a graph.

The subgraph produced contains all the vertices and edges which make up the $HIG(\mathcal{V}, \mathcal{E})$.

### 5.3.2 HIG Construction Guidelines

The HIG graph crystallisation construction technique uses guidelines to support the function *Parse-documentation-structure* and are the following :

1. Identifying Documentation Entities:
    (a) Identify the range of documentation to be modelled, that is identify which documents are to be modelled.
    (b) Identify the set of unique document entity types to be recorded in the $HIG(\mathcal{V}, \mathcal{E})$.
    (c) Identify the sequences of document entity types which may occur within a document hierarchy. For example a sequence could be Book, Chapter, Section and Segment.
    (d) Identify which types of entity are to be classed as segment entities.
    (e) Identify the document entity types which can be connected to the segment entities.
2. Detecting Documentation Entities:
    (a) Detect all document entities within the given range of documentation.
    (b) As each document entity is detected, assign a document entity type to it.
    (c) If an entity is a segment entity, then add it to the set of segment entities.
    (d) If an entity is not a segment entity, then add that document entity to the set of composite entities.
3. Creating Decompositional Dependencies:
    (a) Connect the composite entities together with the consists of dependency ensuring that the composite entities appear in the right order in the hierarchy.
    (b) Connect the document entities to the segment entities with the *has segment dependency*. Only a segment entity which is contained within a composite entity can be connected to the composite entity.

## 5.4 TIG Crystallisation

**The Problem:** *To extract the* $TIG(\mathcal{V}, \mathcal{E})$*, from a documentation system.*

### 5.4.1 Technique Description

The aims of this technique are to record the meaning of the segment and its role with the document hierarchy. The technique records the **thematic structure** (the organisation of the themes) within the document hierarchy being modelled. This is achieved by recording the vertices and edges defined in Chapter 4, definition 4. For each theme vertex in the $TIG(\mathcal{V},\mathcal{E})$ a theme is recorded. The segments where these themes also co-occur are recorded. Theme vertices may be linked together with copy propagation dependencies. The choice of the direction a dependency is an important concept of this particular part of the technique. This is achieved by recording the thematic context of a theme. For example whether a theme describes a definition or a use in the source code. This second part of the technique is called **segment theme analysis**.

The technique can be described in the following way,

*TIG Crystallisation: Documentation* $\rightarrow$ *TIG*$(\mathcal{V},\mathcal{E})$

This operation can be denoted as :

*TIG Crystallisation (Documentation,*$TIG(\mathcal{V},\mathcal{E})$*)* $\underline{\triangle}$ :

---

```
Algorithm TIG Crystallisation
Input: Documentation
Output: $TIG(\mathcal{V},\mathcal{E})$

Begin (Crystallisation of the $TIG(\mathcal{V},\mathcal{E})$ and $TIG(\mathcal{V},\mathcal{E})$ )
    Let $\mathcal{V}$ be a set of vertices, $\{n_1, ..., n_\xi\}$, and
    Let $\mathcal{E}$ is the set of edges, $\{(x,y) \mid (x,y) \in \mathcal{V} \times \mathcal{V}\}$
    {segment theme analysis- this records the thematic structure }
    parse-documentation-for-themes and:
    create the set Thematic Interconnection Entities {Chapter 4, Definition 4}
    create the set Thematic Dependencies {Chapter 4, Definition 4}
    Let $\mathcal{V}$ = Thematic Interconnection Entities
    Let $\mathcal{E}$ = Thematic Dependencies
    construct-graph $TIG(\mathcal{V}, \mathcal{E})$
  End
```

**Remarks.**

*parse-documentation-for-themes* is a function which analyses documentation with respect to the Thematic Interconnection Entities and Thematic Dependencies contained within documentation. *Construct-graph* is a function which creates a graph from the two sets $(\mathcal{V}', \mathcal{E}')$. The two sets $\mathcal{V}$ and $\mathcal{E}$ separately do not form a graph.

### 5.4.2 Thematic Structure Detection Guidelines

The graph crystallisation construction technique uses guidelines to aid the detection of themes in segment entities, that is the *parse-documentation* function. The guidelines are the following :

1. A segment entity can be considered to contain descriptions of the processing of data called conceptual actions (CAs). The actual data can be considered to be conceptual objects (COs).
2. Read the entire segment entity.
3. Decompose the segment entity content, based on its language primitives into a set of tokens.
4. Identify all the tokens that implicitly or explicitly imply COs within the token set. One or more tokens can represent a CO.
5. Identify all the tokens that implicitly or explicitly imply CAs within the token set. One or more tokens can represent a CA.
6. Relate the CAs to the COs. For example identify which CA processes a particular CO. This will constitute a theme.
7. Determine whether a CA is a description of assignment of data or a use data.
8. Determine which CA will be the next CA to process a particular CO. This will be sufficient information to build a picture of the assignment and usage of data within a software system.

9. An example of a theme is “the summation of employee salaries” and its theme code might be “Th5”. The “summation” would be a CA and “employee salaries” would be CO.

10. Furthermore this could be “The salary total is set equal to the summation of employee salaries”. The “salary total” is also a CO and the word “set” also implies an assignment of data.

11. Locate to which theme category the data entities belong. This is achieved by consulting a catalogue of themes. The theme catalogue contains a list of themes and theme codes which can be recorded in a table with two columns, one column for the theme and one column for the theme code. (The rows should be ordered in alphabetical order). The theme codes are used to abbreviate the themes. Once the theme in a segment has been matched with a theme in the theme catalogue the corresponding theme code is extracted from the theme catalogue.

12. Finding the theme (CA and CO) within the theme catalogue is open to human interpretation. Guidelines for this process are the following :

    (a) The themes in the catalogue are separated classes(Which can be defined by a maintenance manager).

    (b) Select the classes in which a theme might belong.

    (c) Examine each class and identify a set of candidate themes which might map to the theme (CA and CO).

    (d) Give each theme in the candidate list a score (the higher the score the closer the match to the CA and CO).

    (e) Rank the candidate themes in descending order.

    (f) Select the theme with the highest score and extract the corresponding theme code from the theme catalogue.

    (g) The segment entity can now be coded with this code.

## 5.5 Graph Annotation

**The Problem:** *Extract the $SAG(\mathcal{V},\mathcal{E})$ from a documentation system. The $SAG(\mathcal{V},\mathcal{E})$ can be partially constructed or fully constructed.*

### 5.5.1 Technique Description

The aims of this technique are to record any source code objects implicitly or explicitly described by a particular segment entity. These relationships help improve the mapping between system documentation and operational software, and also improve the characterisation of document content. The main concept of this technique is segment scanning. The segment scanning part of the technique analyses a segment entity and detects the description of source code entities within segment entities. Conceptually this technique is very similar to that of the Crystallisation technique with the notable difference that the Annotation technique concentrates on recording any source code objects described within a segment whereas the Crystallisation technique concentrates only on thematic structure. The attributes recorded by the technique are defined in Chapter 4, definition 5.

The technique can be described in the following way,

*Graph Annotation :* $HIG(\mathcal{V},\mathcal{E}) \times$ *Documentation* $\rightarrow SAG(\mathcal{V}',\mathcal{E}')$

This operation can be denoted as :

Graph Annotation $(HIG(\mathcal{V},\mathcal{E})$,*Documentation*$)\underline{\triangle}$ :

```
Algorithm Graph Annotation
Input: $HIG(\mathcal{V},\mathcal{E})$, Documentation
Output: $SAG(\mathcal{V}',\mathcal{E}')$

Begin (Annotation of the $SAG(\mathcal{V}',\mathcal{E}')$ )
    Let $\mathcal{V}'$ be a set of vertices, $\{n_1, ..., n_\xi\}$, and
    Let $\mathcal{E}'$ is the set of edges, $\{(x,y) \mid (x,y) \in \mathcal{V} \times \mathcal{V}\}$
    {segment scanning}
    parse-documentation-for-source-code and:
    create the set Source Code Entities {Chapter 4, Definition 4}
    create the set Document Entity and Module Association Dependencies
        { Chapter 4, Definition 4}
    Let $\mathcal{V}'$ = Source Code Entities
    Let $\mathcal{E}'$ = Document Entity and Module Dependencies
    construct-graph $SAG(\mathcal{V}', \mathcal{E}')$
  End (Annotation of the $SAG(\mathcal{V}', \mathcal{E}')$)
```

**Remarks.**
This technique can be applied either incrementally or to the whole system. *parse-documentation-for-source-code* is a function to analyse the content of documentation with respect to the Source Code Entities and Document Entity and Module Dependencies.

*Construct-graph* is a function which creates a graph from the two sets $(\mathcal{V}', \mathcal{E}')$. The two sets $\mathcal{V}$ and $\mathcal{E}$ separately do not form a graph.

### 5.5.2 Source Code Entity Detection Guidelines

The graph annotation construction technique uses guidelines to aid the detection of source code entities described in segment entities. These guidelines are for the realisation of the *parse-documentation-for-source-code* function. The guidelines are the following :

1. Read the entire segment entity.
2. Decompose the segment content, based on the language primitives, into a set of tokens.
3. Identify all the tokens that implicitly or explicitly imply COs within the token set. One or more tokens can represent a CO.
4. Identify all the tokens that implicitly or explicitly imply CAs within the token set. One or more tokens can represent a CA.
5. Identify the module entities which relate to the CAs. In other words identify which source code module is described by each segment entity.
6. Identify all the COs which relate to a particular module which is described within a segment entity.

7. Add to this list of COs, a list other source code objects which are related to a particular module.

## 5.6 Graph Edge Parameterisation

**The Problem:** *To extract the $WIG(\mathcal{V},\mathcal{E})$ from a documentation system in particular the release information documentation. The $WIG(\mathcal{V},\mathcal{E})$ can be partially constructed or fully constructed. The second part of this problem is to add maintainer's expert judgements about potential ripple effects to the $RPG(\mathcal{V},\mathcal{E})$.*

### 5.6.1 Technique Description

The aims of this technique are to add weighted edges between segment entities in order to indicate the strength of the relationship between the entities. The parameterised edges are added as the information becomes available.

The name graph edge parameterisation was selected as it indicates the process of adding parameterised edges the $RPG(\mathcal{V},\mathcal{E})$ according to previous release information and expert judgements. This technique involves parameterising the edges between segments which is called **edge parameterisation**. There are two methods of achieving this, namely **direct edge parameterisation** and **indirect edge parameterisation**. Direct edge parameterisation data is recorded from release information which describes which segments had previously affected other segments. As an extension to this feature the previous ripple effects between the modules referenced or described in the segment entities are also recorded. The indirect data comes from expert judgements about the potential ripple effects between segment entities. The connections between module entities are more easy to visualise than connections between segment entities because of the implicitness of the segment entity interconnections. Expert judgements are recorded about possible inter-module ripple effects in addition to inter-segment ripple effects. The vertices and edges recorded by this technique are defined in Chapter 4, definition 6.

The technique can be described in the following way, *Graph Edge Parameterisation* : $RPG(\mathcal{V},\mathcal{E}) \times (v1, v2, w) \rightarrow RPG(\mathcal{V}',\mathcal{E}')$

(Note: The $WIG(\mathcal{V},\mathcal{E}) \sqsubseteq RPG(\mathcal{V}',\mathcal{E}')$)

This operation can be denoted as :

*Graph Edge Parameterisation* $(RPG(\mathcal{V},\mathcal{E}),(v1, v2, w)) \underline{\triangle}$ :

```
Algorithm Graph Edge Parameterisation
Input: $RPG(\mathcal{V},\mathcal{E})$, $(v1, v2, w)$
Output: $RPG(\mathcal{V}',\mathcal{E}')$
Begin (Edge Parameterisation )
    Let $\mathcal{V}'$ be a set of vertices, $\{n_1, ..., n_\xi\}$, and
    Let $\mathcal{E}'$ be the set of edges, $\{(x,y) \mid (x,y) \in \mathcal{V} \times \mathcal{V}\}$
    Let the triple $(v1, v2, w)$ be an edge from vertex $v1$ to $v2$ with parameters $w$
    { w (weight) are parameters recording the probability of a ripple effect}
    delete-edge $RPG((v1, v2))$
    Let $\mathcal{E}' = RPG(\mathcal{E}) \cup (v1, v2, w)$
    Let $\mathcal{V}' = \mathcal{V}$
   construct-graph $RPG(\mathcal{V}',\mathcal{E}')$
End (Edge Parameterisation)
```

**Remarks:**

The edge $(v1, v2)$ with a parameter can represent a possible ripple effect between two modules described in the documentation or between two segment entities belonging to the documentation. The parameter can be expert judgement (indirect parameterisation) or release information (direct parameterisation). *Construct-graph* is a function which creates a graph from the two sets $(\mathcal{V}',\mathcal{E}')$. The two sets $\mathcal{V}$ and $\mathcal{E}$ separately do not form a graph.

### 5.6.2 Expert Judgement Collection Guidelines

The graph edge parameterisation construction technique uses guidelines to aid the detection of expert judgement. These guidelines support the creation of the parameter $w$, that is, the weight for the graph edges in the $WIG(\mathcal{V}, \mathcal{E})$. The guidelines are the following :

1. Select two segment entities on the model (or two modules described in two different segment entities).
2. Record assumptions and reasons for a particular judgement.
3. If two conflicting judgements are made then the maintainer who is most reliable in making judgements and the most experienced at maintaining the system should have their judgement recorded.
4. The edge on the model $(u, v, w)$ should be created by mapping an entity to $u$ and entity to $v$ and the expert judgement should be mapped to $w$.

### 5.6.3 Release Information Collection Guidelines

The graph edge parameterisation construction technique uses guidelines to aid the collection of release information. These guidelines support the creation of the parameter $w$, that is, the weight for the graph edges in the $WIG(\mathcal{V}, \mathcal{E})$. The guidelines are the following :

1. When a segment entity propagates a ripple effect to another segment entity or where a module entity propagates a ripple effect to another module entity, then a new percentage probability of a ripple effect between these two entities must be recorded on the edge.
2. The percentage probability is calculated by:
   the number of times the entity has propagated a ripple effect to a particular entity \ the number of times the entity propagating the ripple effect has been modified * 100.
3. The edge on the model $(u, v, w)$ should be created by mapping an entity to $u$ and an entity to $v$. The probability of a ripple effect should be mapped to $w$.

## 5.7 Graph Splicing

**The Problem:** *Create an $RPG(\mathcal{V},\mathcal{E})$ from one or more of the subgraphs or create a model of multiple systems by combining $RPG(\mathcal{V},\mathcal{E})$ models.*

### 5.7.1 Technique Description

The aim of this technique is to combine two graphs together to form one graph. Duplicate vertices and edges do not appear in the resultant graph. The technique takes any named graph and adds it to the main $RPG(\mathcal{V},\mathcal{E})$ model.

The technique can be described in the following way,

*Graph Splice:* $RPG(\mathcal{V},\mathcal{E}) \times \textit{graph} \rightarrow RPG(\mathcal{V}',\mathcal{E}')$

This operation can be denoted as :

*Graph Splice* $RPG(\mathcal{V},\mathcal{E}) \sqcup \text{graph}(\mathcal{V},\mathcal{E}) \underline{\triangle}$

---

```
Algorithm Graph Splice
Input: $RPG(\mathcal{V}_1,\mathcal{E}_1)$,$graph(\mathcal{V}_2,\mathcal{E}_2)$
Output: $RPG(\mathcal{V}',\mathcal{E}')$

Begin (Splice graph with an $RPG(\mathcal{V},\mathcal{E})$ )
    Let $\mathcal{V}'$ be a set of vertices, $\{n_1,...,n_\xi\}$, and
    Let $\mathcal{E}'$ is the set of edges, $\{(x,y) \mid (x,y) \in \mathcal{V}\times\mathcal{V}\}$
    Let $\mathcal{V}' = \mathcal{V}_1 \cup \mathcal{V}_2$
    Let $\mathcal{E}' = \mathcal{E}_1 \cup \mathcal{E}_2$
    construct-graph$RPG(\mathcal{V}',\mathcal{E}')$
End (Splice graph with an $RPG(\mathcal{V},\mathcal{E})$ )
```

**Remarks.**

The graph union operation is based on the set union property and ensures that the duplicate vertices and edges are removed. *Construct-graph* is a function which creates a graph from the two sets $(\mathcal{V}', \mathcal{E}')$. The two sets $\mathcal{V}$ and $\mathcal{E}$ separately do not form a graph.

## 5.8 Graph Clipping

**The Problem:** *To remove vertices and edges from the $RPG(\mathcal{V}, \mathcal{E})$ given an ordered pair $(a, b)$ or a vertex $a$ or $b$ representing the item to be clipped from the $RPG(\mathcal{V}, \mathcal{E})$.*

### 5.8.1 Technique Description

This technique can be applied when any information in the model becomes outdated and needs to be removed. The aim of this technique is to remove vertices and edges from the $RPG(\mathcal{V}, \mathcal{E})$.

The technique can be described in the following way,

*Graph Clipping :* $RPG(\mathcal{V}, \mathcal{E}) \times$ *item to be clipped* $\rightarrow RPG(\mathcal{V}', \mathcal{E}')$

*This operation can be denoted as :*

*Graph Clipping* $(RPG(\mathcal{V}, \mathcal{E})$, *item to be clipped)* $\underline{\Delta}$ :

---

*Algorithm* **Graph Clipping**

**Input:** $RPG(\mathcal{V}, \mathcal{E})$, *item to be clipped (i.e., edge or vertex)*

**Output:** $RPG(\mathcal{V}', \mathcal{E}')$

*(The edge to be clipped is denoted as $(a, b)$, and the entity to be clipped is denoted as $a$ or $b$.)*

**Begin** *(Clipping $RPG(\mathcal{V}, \mathcal{E})$)*

1. *If the item to be clipped is a Composite Entity perform step 10 & 17* **else**
2. *If the item to be clipped is a Segment Entity perform step 11 & 17* **else**
3. *If the item to be clipped is a Thematic Entity perform step 12 & 17* **else**
4. *If the item to be clipped is a Consists of Dependency perform step 13 & 17* **else**
5. *If the item to be clipped is a Has Segment Dependency perform step 14 & 17* **else**
6. *If the item to be clipped is a Describes Part of Module Dependency perform step 15 & 17* **else**
7. *If the item to be clipped is a Thematic Dependency perform step 16 & 17* **else**
8. *If the item to be clipped is any other entity remove that entity from the set $\mathcal{V}$* **else**
9. *If the item to be clipped is any other dependency remove that dependency $(a, b)$ from the set $\mathcal{E}$ and perform step 17.*
10. *Clip Composite Entity $a$ from the $HIG(\mathcal{V}, \mathcal{E})$:*
    (a) *Find the edge $(a, b)$ connecting entity $a$ to the next hierarchical level in the $HIG(\mathcal{V}, \mathcal{E})$.*
    (b) *Delete this edge $(a, b)$.*
    (c) *Delete all decompositional entities below entity $a$ in the $HIG(\mathcal{V}, \mathcal{E})$.*
    (d) *Delete all decompositional dependencies below entity $a$ in the $HIG(\mathcal{V}, \mathcal{E})$.*
    (e) *Delete all thematic entities connecting to the decompositional entities which were deleted in step 9(c).*
    (f) *Delete all thematic dependencies associated with deleted thematic entities .*
11. *Clip Segment Entity $a$ from the $HIG(\mathcal{V}, \mathcal{E})$:*

(a) *Find the set of has segment* edges *connecting entity $a$ to the next hierarchical level in the $HIG(\mathcal{V},\mathcal{E})$.*

(b) *Delete this set of has segment* edges *from the $HIG(\mathcal{V},\mathcal{E})$.*

(c) *Delete* all *weighted dependencies* associated *with the segment entities deleted.*

(d) *Delete* all associated *Document Entity and Module Association Dependencies* from *the $SAG(\mathcal{V},\mathcal{E})$.*

(e) *Delete* all *thematic entities connecting to deleted segment entities* from *the $TIG(\mathcal{V},\mathcal{E})$.*

(f) *Delete* all *thematic dependencies* connecting *to deleted segment entities* from *the $TIG(\mathcal{V},\mathcal{E})$.*

12. *Clip Thematic Entity* from *the $TIG(\mathcal{V},\mathcal{E})$:*

    (a) *Delete Thematic Entity,* from *$TIG(\mathcal{V},\mathcal{E})$*

    (b) *Delete* all *thematic dependencies* connecting *to deleted Thematic Entity,* from *$TIG(\mathcal{V},\mathcal{E})$.*

13. *Clip Consists of Dependency* from *the $HIG(\mathcal{V},\mathcal{E})$:*

    (a) *Delete dependency $a, b$* from *$HIG(\mathcal{V},\mathcal{E})$.*

    (b) *Delete* all *decompositional entities* below *entity $a$.*

    (c) *Delete* all *decompositional dependencies* below *entity $a$.*

    (d) *Delete* all *thematic entities connecting to decompositional entities in the $TIG(\mathcal{V},\mathcal{E})$.*

    (e) *Delete* all *thematic dependencies* connecting *to decompositional entities in the $TIG(\mathcal{V},\mathcal{E})$.*

    (f) *Delete* all associated *Document Entity and Module Association Dependencies* from *the $SAG(\mathcal{V},\mathcal{E})$.*

    (g) *Delete* all *Source Code Entities* from *the $SAG(\mathcal{V},\mathcal{E})$.*

14. *Clip Has Segment Dependency* from *the $HIG(\mathcal{V},\mathcal{E})$:*

    (a) *Find the set of has segment* edge *connecting entity $a$ to the next hierarchical level in the $HIG(\mathcal{V},\mathcal{E})$.*

    (b) *Delete this has segment* edge *from the $HIG(\mathcal{V},\mathcal{E})$.*

    (c) *Delete* all associated *weighted dependencies* from *the $SAG(\mathcal{V},\mathcal{E})$.*

    (d) *Delete* all *thematic entities connecting to deleted segment entities* from *the $TIG(\mathcal{V},\mathcal{E})$.*

(e) *Delete all associated Document Entity and Module Association Dependencies from the $SAG(\mathcal{V}, \mathcal{E})$.*

(f) *Delete all Source Code Entities affected, from the $SAG(\mathcal{V}, \mathcal{E})$.*

(g) *Delete all thematic dependencies connecting to deleted segment entities from the $TIG(\mathcal{V}, \mathcal{E})$.*

15. *Clip Describes Part of Module Dependency from the $SAG(\mathcal{V}, \mathcal{E})$:*

   (a) *Delete all associated Document Entity and Module Association Dependencies from the $SAG(\mathcal{V}, \mathcal{E})$.*

   (b) *Delete all Source Code Entities from the $SAG(\mathcal{V}, \mathcal{E})$.*

16. *Clip Thematic Dependency from the $TIG(\mathcal{V}, \mathcal{E})$:*

   (a) *If the edge is a has theme vertex dependency then firstly delete the edge and the stop vertex and secondly delete all edges arriving at the stop vertex in the $TIG(\mathcal{V}, \mathcal{E})$.*

   (b) *If the edge is a has theme dependency then delete the edge and the stop vertex from the $TIG(\mathcal{V}, \mathcal{E})$.*

   (c) *If the edge is a co-occurs dependency then delete the edge from the $TIG(\mathcal{V}, \mathcal{E})$.*

   (d) *If the edge is a copy propagation dependency then delete the edge from the $TIG(\mathcal{V}, \mathcal{E})$.*

   (e) *If the edge is a definition use description chain dependency then delete the edge from the $TIG(\mathcal{V}, \mathcal{E})$.*

   (f) *If the edge is a thematic context property dependency then delete the edge and the stop vertex from the $TIG(\mathcal{V}, \mathcal{E})$.*

   (g) *If the edge is a theme category dependency then delete the edge and the start and stop vertex from the $TIG(\mathcal{V}, \mathcal{E})$.*

17. *Construct Graph $RPG(\mathcal{V}', \mathcal{E}')$*

**End** *(Clipping $RPG(\mathcal{V}, \mathcal{E})$)*

**Remarks.**

This simple technique can be applied to any $RPG(\mathcal{V}, \mathcal{E})$ or any subgraph $\sqsubseteq RPG(\mathcal{V}, \mathcal{E})$.

## 5.9 Summary

In this Chapter techniques for constructing the $RPG(\mathcal{V},\mathcal{E})$ model have been presented. Each technique builds or changes a different facet of the model. The graph crystallisation technique constructs the $HIG(\mathcal{V},\mathcal{E})$ and the $TIG(\mathcal{V},\mathcal{E})$. The graph annotation technique constructs the $SAG(\mathcal{V},\mathcal{E})$. The graph parameterisation constructs the $WIG(\mathcal{V},\mathcal{E})$. The graph splicing technique joins together any two subgraphs or any two $RPG(\mathcal{V},\mathcal{E})$ models and the graph clipping technique removes any unwanted graph edges and vertices. These techniques will be applied to examples of documentation structure and case study in Chapter 8 and will be evaluated in Chapter 9. The next chapter presents several techniques for analysing the $RPG(\mathcal{V},\mathcal{E})$ model, once it has been partially or fully constructed.

# Chapter 6

# Ripple Propagation Graph Analysis

## 6.1 Introduction

The proposed documentation analysis techniques make extensive use of manipulating the $RPG(\mathcal{V}, \mathcal{E})$ introduced in Chapter 4. This Chapter explains the graph manipulations and the notation that will be used. The graph manipulations are described formally. Collectively this set of manipulations for $RPG(\mathcal{V}, \mathcal{E})$ analysis is called thematic graph slicing.

**Notation for Describing the Techniques:**

Each $RPG(\mathcal{V}, \mathcal{E})$ analysis technique presented in this Chapter includes the problem to be solved by the technique, reasons for developing the technique, the technique description, the results produced by the technique, the use of the results and the limitations of the technique. The algorithms within this Chapter operate on the two $RPG$ sets $\mathcal{V}$ and $\mathcal{E}$. The impacted bag of entities $\mathcal{BV}$ and the impacted bag of edges $\mathcal{BE}$ are constructed from different subsets of the $RPG(\mathcal{V}, \mathcal{E})$. The impacted set of entities $\mathcal{V}'$ and the impacted set of edges $\mathcal{E}'$ are constructed from different subsets of the $RPG(\mathcal{V}, \mathcal{E})$. These subsets are identified by propagating the ripple effect from specified entities in set $\mathcal{V}$ along directed paths recorded in set $\mathcal{E}$. The following is an example of the notation used :

Find the *Theme Vertex Entities Impacted:*

1. **Let** *Theme Vertex Entities Impacted* =
   $\prec$ A$_i$ $\mid (A_i, t_n) \in$ *Has Theme Dependency* and $t_n \in \mathcal{TB} \succ$
2. **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, t_n) \mid (A_i, t_n) \in$ *Has Theme Dependency* and $t_n \in \mathcal{TB} \succ$

The above example maps entities in a bag called $\mathcal{TB}$, to entities in a bag called *Theme Vertex Entities Impacted.* The mapping is achieved using the *Has Theme Dependency*, where $A_i$ is a member of *Theme Vertex Entities Impacted* and $t_n$ is a member of the bag $\mathcal{TB}$. The bag $\mathcal{BE}$ contains the impacted edges between the two bags, *Theme Vertex Entities Impacted* and $\mathcal{TB}$. This approach will make the algorithms easier to comprehend. Another notation considered for describing the graph analysis techniques was to use the transitive closure of the change. This notation, whilst being elegant and concise, may be more difficult to understand than describing how the bags of impacted entities are built up.

## 6.2 Philosophy of the Graph Slicing Techniques

The aims of the graph analysis techniques are to manipulate the $RPG(\mathcal{V}, \mathcal{E})$ model and extract subgraphs from it. The subgraphs extracted represent impacted documentation entities and relationships between these entities. The impact analysis techniques are based on the ideas of program slicing. The idea of program slicing as a form of program decomposition was introduced by Weiser [91, 92]. A program slice is a source code to source code transformation where the resulting program statements conform to some criterion. Examples of slicing criterion are :

1. Program statements which do not affect the value of a particular variable in a given range of program statements.
2. All of the program statements which have contributed to the value within a chosen variable, within a range of program statements.

The following is an example of a program which is sliced with respect to the second slice criterion mentioned above.

```
The original program fragment:
1    BEGIN
2     READ(X,Y);
3     TOTAL : =  0.0;
4     SUM : =  0.0;
5     IF X< =  1
6     THEN SUM : =  Y
7     ELSE BEGIN
8             READ(Z);
9             TOTAL : =  X*Y;
10            END;
11    WRITE(TOTAL,SUM)
12    END.
```

One slice is for the variable Z and from line 12 and the other for the variable X from line 9.

```
   Slice on criterion <12,{Z}>.  Slice on criterion <9,{X}>

 1   BEGIN                         1    BEGIN
```

```
2   READ(X,Y);                    2    READ(X,Y);
5   IF X< =  1                    12   END
6   THEN  SUM : =  Y
7   ELSE READ(Z);
12  END.
```

---

There have been several refinements to the theory of program slicing [33, 45, 1, 35] including recently, the slicing of models of module interconnection. These models were based on graph theory. Graph slicing is defined by Calliss [14] to be :

*The process of creating a new graph by extracting a subgraph from a given graph according to some criterion.*

In the concluding Chapter of the Gallagher thesis [33] the following question is posed :

*Slicing is a data flow and control flow method, for analysis of source text. Can it be done on higher level objects to yield any insight?*

To date, there have been no publications which have addressed this problem. The solution to the early ripple effect problem presented in this thesis, uses the idea of slicing graph theory models of higher levels of abstraction than source code itself.

In this thesis a new type of slice called **thematic graph slicing** is proposed. Thematic graph slicing is defined as the analysis of the thematic structure of a ripple propagation graph. Each document segment entity on an $RPG(\mathcal{V}, \mathcal{E})$ is characterised with the thematic structure describing the semantic role of that document segment entity. A bag of themes is extracted from a change proposal and the resulting bag forms the **slice criterion**. These themes are then mapped to segments on the graph, for the identification of the set of directly affected document segment entities. The members of this set are called the **impact points**. The graph can then be traversed from these impact points to identify affected components. This aspect is described in more detail in the following pages.

## 6.3 Constructing a Graph Slicing Criterion

This section presents a technique for extracting information from change proposals and creating a criterion for the slicing techniques which have been developed.

### 6.3.1 A Model of a Change Proposal

Examples of a **change proposal** can be found in [4, 15]. These proposals are very simple and contain proposal identification information, the proposed change and the reasons for the change. The following is an example of the content a change proposal based on [4, 15] :

1. Proposal Identification Information.
2. Contents.
3. Business Summary.
4. Detailed Business Requirements.
5. Service Levels.
6. Timescales.
7. Assumptions and Constraints.

Change proposals for perfective maintenance focus on change to the services provided by a computer system. Change to services can imply changes in the way data already in the system and also new data are processed. A generalisation that can be stated about change proposals is that they discuss changes to the processing of data. Processing can be considered as conceptual actions (CAs), whilst data can be considered as conceptual objects (COs). The problem is how to map change proposal contents onto known objects and processes in an operational system.

A change proposal containing the above information needs to be analysed with respect to themes (description of the processing of data) which it contains. These themes may be orientated towards the user or customer view of an application (business system oriented themes). Hence themes which may be familiar to a maintenance programmer may not be explicitly mentioned in the proposal. There needs to be some corresponding business themes and technical themes.

### 6.3.2 Technique Description

**The Problem:** *To extract a bag of themes mentioned in a change proposal.*

The aim of the technique is to extract a bag of themes from a change proposal. This is achieved by analysing the content of the change proposal and identifying the themes. Some themes occur more than once in a change proposal therefore a bag is used to accommodate multiple occurrences of the same theme. For example a proposed project may have two different work packages affecting the processing of the same piece of data. The technique then converts these themes into themes which occur at the system level. The technique has the following definition :

*Create Theme Bag : Change Proposal* $\rightarrow$ *Theme Bag*

This operation can be denoted as :

*Create Theme Bag (Change Proposal, Theme Bag)*

### 6.3.3 Theme Bag Construction Guidelines

The strategy for analysing a change proposal and creating a theme bag is a multipass strategy, namely :

1. Divide the change proposal into units where each unit maps onto the model of a change proposal.
2. Identify the detailed business requirements.
3. Decompose detailed business requirements into a set of individual requirements.

4. For each requirement, study the content of the requirement with references to the proposed changes to the way data is processed.

5. Identify the CAs and COs in the change proposal to produce a candidate list of themes.

6. Identify any phrases or paragraphs which may imply themes.

7. Repeat this steps 4 to 6 until the whole of the business requirements have been analysed.

### 6.3.4 Results Produced by the Technique

This theme bag can be regarded as a model of the change proposal and will be used for analysing the $RPG(\mathcal{V},\mathcal{E})$ model. The theme bag will be used as the graph slicing criterion for the graph slicing techniques described in this thesis. Associated with this theme bag will be the following :

1. proposal identification information such as proposal number and project number,

2. source of change information indicating where the change proposal originated, such as the division, department and originators name.

This is so that the results of analysing the impact of a proposal can be linked with a proposal when considering the impact of many competing change proposals. The theme bag will be denoted as $\mathcal{TB}$ within the description of the $RPG(\mathcal{V},\mathcal{E})$ slicing techniques.

## 6.4 Overview of the Graph Slicing Techniques

In this thesis there are five techniques for slicing documentation. All of the techniques take as input a Theme Bag which is then used as a slice criterion. All the techniques manipulate the $RPG(\mathcal{V},\mathcal{E})$ model. The extracted $RPG(\mathcal{V}',\mathcal{E}')$ subgraph is called a **Change Implication Graph** ( $CIG(\mathcal{V}',\mathcal{E}')$). The five techniques are the following :

1. The **Thematic Graph Slicing** is a primitive slicing technique which only considers the co-occurrence of themes in the $RPG(\mathcal{V},\mathcal{E})$ model. A problem with this technique is that it

only extracts directly impacted segment entities and composite entities. The technique does not capture the essence of the secondary ripple effect, namely the description of source code assignments in documentation.

2. The **Complex Thematic Slicing** analyses firstly the co-occurrence of themes and then the indirect ripple effects using the copy propagation dependencies. This technique not only identifies segment entities impacted but also includes the composite entities impacted. A problem with this technique is that it produces a large amount of information. This is also a problem of source code based slicing techniques. Dynamic slicing [1] was introduced at the source code level to alleviate this problem. However, documentation cannot be executed. A second problem with this technique is that it only informs the maintainer of the entities impacted but does not indicate the types of entity impacted.

3. The **Weighted Graph Slicing** is a Complex Thematic Slice which includes probabilities on the edges of the resulting $CIG(\mathcal{V}', \mathcal{E}')$. The probabilities are based on previous release information. This is to help the maintainer reduce this slice further by removing entities and edges which in all probability will not be impacted. This is so that the maintainer can analyse all of the edges with low probabilities of ripple effects, with a view to removing them from the $CIG(\mathcal{V}', \mathcal{E}')$. This technique also identifies all of the types of entities impacted. A problem with this technique is that it is necessary to have a data base of release information with which to calculate the probabilities of ripple effects.

4. The **Augmented Graph Slicing** is also a Complex Thematic Slice which includes probabilities on the edges of the resulting $CIG(\mathcal{V}', \mathcal{E}')$. However the probabilities are based on quantitative expert judgement, therefore a database of release information is not required. The problem with this technique is that it does not indicate any source code objects which are mentioned in the documentation.

5. The **Annotated Graph Slicing** is also a Complex Thematic Slice which includes a compositional characterisation of the impacted segment entities. This slice analyses the $RPG(\mathcal{V}, \mathcal{E})$ with respect to the source code entities impacted. This technique does not analyse source code but extracts source code entities described by impacted segment entities.

When the above slicing algorithms terminate, the $(\mathcal{V}', \mathcal{E}')$ produced is a subgraph of the $RPG(\mathcal{V}, \mathcal{E})$ and is denoted as graph containment :

$$CIG(\mathcal{V}',\mathcal{E}') \sqsubseteq RPG(\mathcal{V},\mathcal{E}) \ .$$

The $CIG(\mathcal{V}',\mathcal{E}')$ produced by the slicing techniques has the following properties. It is a directed, acyclic, labelled and strict sub graph. The $\gamma$ and $\delta$ $CIG(\mathcal{V}',\mathcal{E}')$ s are also parameterised graphs. The $CIG(\mathcal{V}',\mathcal{E}')$ contains the transitive closure of the change characterised by $\mathcal{TB}$. The names $\alpha$ graph slice, $\beta$ graph slice, $\gamma$ graph slice, $\delta$ graph slice, and $\epsilon$ graph slice are used to abbreviate the five graph slicing techniques respectively.

## 6.5 $\alpha$ Graph Slice (Thematic)

**The Problem:** *To extract a $CIG(\mathcal{V}',\mathcal{E}')$ containing composite entities and segment entities which are directly impacted by tracing the co-occurs dependencies.*

### 6.5.1 Reasons for Developing the Technique

The reasons for developing this technique are firstly to find all of the segment entities which are impacted by the themes within the theme bag. The second reason for developing this technique is to trace vertically upwards from a segment following the has segment dependency and the consists of dependency to the top of the $HIG(\mathcal{V},\mathcal{E})$ identifying all of the composite entities impacted.

### 6.5.2 Technique Description

The technique can be described in the following way,

*$\alpha$ Graph Slice :* $RPG(\mathcal{V},\mathcal{E}) \rightarrow \alpha \ CIG(\mathcal{V}',\mathcal{E}')$

This operation can be denoted as :

*$\alpha$ Graph Slice* ( $RPG(\mathcal{V},\mathcal{E})$ , $\mathcal{TB}$ ) $\underline{\triangle}$ :

*Algorithm* $\alpha$ **Graph Slice**

**Input:** $RPG(\mathcal{V},\mathcal{E})$ , $\mathcal{TB}$

**Output:** $\alpha$ $CIG(\mathcal{V}',\mathcal{E}')$

**Begin** ( $\alpha$ Graph Slice)

```
1. Find the Theme Vertex Entities Impacted:
    (a) Let Theme Vertex Entities Impacted =
        $\prec A_i \mid (A_i, t_n) \in$ Has Theme Dependency and $t_n \in \mathcal{TB} \succ$
    (b) Let $\mathcal{BE} = \mathcal{BE} + \prec(A_i, t_n) \mid (A_i, t_n) \in$ Has Theme Dependency and $t_n \in \mathcal{TB}\succ$
2. Find the Co-occurs Dependencies Impacted:
    (a) Let $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ Co-occurs Dependency and $A_i \in$ Theme Vertex
        Entities Impacted, $B_n \in$ Theme Vertex Entities Impacted$\succ$
3. Find the Theme Entities Impacted (directly and indirectly impacted themes)
    (a) Let Theme Entities Impacted =
        $\prec B_n \mid (A_i, B_n) \in$ Has Theme Dependency and $A_i \in$ Theme Vertex Entities Impacted $\succ$
    (b) Let $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ Has Theme Dependency and $A_i \in$ Theme Vertex
        Entities Impacted $\succ$
4. Find the Segment Entities Impacted:
    (a) Let Segment Entities Impacted =
        $\prec A_i \mid (A_i, B_n) \in$ Has Theme Vertex Dependency and $B_n \in$ Theme Vertex Entities
        Impacted $\succ$
    (b) Let $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ Has Theme Vertex Dependency and $B_n \in$ Theme
        Vertex Entities Impacted $\succ$
5. Find the Composite Components Impacted[1]:
```

[1]This step in the analysis is called the composite entity slice

```
    (a) Let Composite Components Impacted =
        $\prec A_i \mid (A_i, B_n) \in$ Has Segment Dependency and $B_n \in$ Segment Entities Impacted $\succ$
        Let $\mathcal{BE} = \mathcal{BE} + \prec (A_i, B_n) \mid (A_i, B_n) \in$ Has Segment Dependency and $B_n \in$ Segment Entities Impacted $\succ$
    (b) Let Composite Components Impacted = Composite Components Impacted +
        $\prec A_i \mid (A_i, B_n) \in$ Consists of Dependency and $B_n \in$ Composite Components Impacted $\succ$
        Let $\mathcal{BE} = \mathcal{BE} + \prec (A_i, B_n) \mid (A_i, B_n) \in$ Consists of Dependency and $B_n \in$ Composite Components Impacted $\succ$
    (c) Extract the set of Composite Components from the bag of Composite Components Impacted.
    (d) Repeat step 5.(b) and 5.(c) for each hierarchical level in the $RPG(\mathcal{V}, \mathcal{E})$ until the newly calculated set of Composite Components does not change;

6. Find all impacted documentation entities :
    (a) Let $\mathcal{BV}$ =
        Composite Components Impacted + Segment Entities Impacted + Theme Vertex Entities Impacted + Theme Entities Impacted

7. Create the sets $\mathcal{V}'$ and $\mathcal{E}'$ from $\mathcal{BV}$ , $\mathcal{BE}$ :
    (a) Extract set $\mathcal{E}'$ from $\mathcal{BE}$
    (b) Extract set $\mathcal{V}'$ from $\mathcal{BV}$

8. Construct the $\beta$ $CIG(\mathcal{V}', \mathcal{E}')$ by associating the sets $\mathcal{V}'$ and $\mathcal{E}'$

End ( $\alpha$ Graph Slice)
```

### 6.5.3 Results Produced by the Technique

The $\alpha$ $CIG(\mathcal{V}', \mathcal{E}')$ contains the following types of impacted vertices and edges.

1. Graph Vertices: *Composite Entities* $\cup$ *Segment Entities* $\cup$ *Theme Vertex Entities* $\cup$ *Theme Entities*
2. Graph Edges: *Co-occurs Dependencies* $\cup$ *Consists of Dependencies* $\cup$ *Has Segment Dependencies* $\cup$ *Has Theme Vertex Dependencies* $\cup$ *Has Theme Dependencies*
3. Other Features: This technique identifies direct impacts.

The results produced by this graph slicing technique, $\alpha$ $CIG(\mathcal{V}', \mathcal{E}')$ can be used to help make the decisions identified in Chapter 3 (see table 3.1 ) :

1. Decision No. 12 Which technical documentation needs to be written/amended?

The bags $\mathcal{BV}$ , $\mathcal{BE}$ can be used to show multiple impacts in the sliced $\beta$ $CIG(\mathcal{V}', \mathcal{E}')$.

A problem with this technique is that it does not capture the essence of the secondary ripple effect, that is the description of source code assignments in documentation.

## 6.6 $\beta$ Graph Slice (Complex Thematic)

**The Problem:** *To extract a $\beta$ $CIG(\mathcal{V}', \mathcal{E}')$ containing composite entities and segment entities which are directly impacted by tracing the co-occurs dependencies and the copy propagation dependencies.*

### 6.6.1 Reasons for Developing the Technique

The reasons for developing this technique are that the $\alpha$ graph slice does not analyse the essence of the ripple effect, namely, descriptions of assignments also the $\alpha$ graph slice only considers the co-occurrence of themes. This technique also addresses the problem of identifying the different types of entities appearing in the impacted slice. For example some entities will describe input, output, possibly hardware and human factors etc. This part of the theory provides a more precise picture of what is impacted than simply identifying entities impacted. Another reason for developing this technique is to detect multiple impacts on the same entity.

### 6.6.2 Technique Description

The technique can be described in the following way,

$\beta$ *Graph Slice :* $RPG(\mathcal{V},\mathcal{E}) \rightarrow \beta\ CIG(\mathcal{V}',\mathcal{E}')$

This operation can be denoted as :

$\beta$ *Graph Slice* ( $RPG(\mathcal{V},\mathcal{E})$ , $\mathcal{TB}$ ) $\underline{\Delta}$ :

---

*Algorithm* $\beta$ **Graph Slice**

**Input:** $RPG(\mathcal{V},\mathcal{E})$ , $\mathcal{TB}$

**Output:** $\beta\ CIG(\mathcal{V}',\mathcal{E}')$ , bag $\mathcal{BV}$ , bag $\mathcal{BE}$

**Begin** ( $\beta$ Graph Slice)

```
1. Find the Theme Vertex Entities Impacted:
   (a) Let Theme Vertex Entities Impacted =
       $\prec A_i \mid (A_i, t_n) \in$ Has Theme Dependency and $t_n \in \mathcal{TB} \succ$
   (b) Let $\mathcal{BE} = \mathcal{BE} + \prec (A_i, t_n) \mid (A_i, t_n) \in$ Has Theme Dependency and $t_n \in \mathcal{TB} \succ$
2. Find the Co-occurs Dependencies Impacted:
   (a) Let $\mathcal{BE} = \mathcal{BE} + \prec (A_i, B_n) \mid (A_i, B_n) \in$ Co-occurs Dependency and $A_i \in$ Theme Vertex
       Entities Impacted, $B_n \in$ Theme Vertex Entities Impacted$\succ$
```

3. Find the *Theme Vertex Entities Impacted* using *Copy Propagation Dependency*:

   (a) **Let** *Theme Vertex Entities Impacted = Theme Vertex Entities Impacted* + $\prec B_n \mid (A_i, B_n) \in$ *Copy Propagation Dependency and* $A_i \in$ *Theme Vertex Entities Impacted* $\succ$

   (b) **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Copy Propagation Dependency* and $A_i \in$ *Theme Vertex Entities Impacted* $\succ$

   (c) **Extract** the set of *Theme Vertex Entities Impacted* from the bag of *Theme Vertex Entities Impacted*

   (d) Repeat steps 3.(a), 3.(b) and 3.(c) until the newly calculated set of *Theme Vertex Entities Impacted* does not change;

4. Find the *Thematic Context Entities:*

   (a) **Let** *Thematic Context Entities Impacted* = $\prec B_n \mid (A_i, B_n) \in$ *Thematic Context Dependency and* $A_i \in$ *Theme Vertices Impacted* $\succ$

   *(b)* **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Thematic Context Dependency and* $A_i \in$ *Theme Vertices Impacted* $\succ$

5. Find the *Theme Entities Impacted* (directly and indirectly impacted themes)

   (a) **Let** *Theme Entities Impacted* = $\prec B_n \mid (A_i, B_n) \in$ *Has Theme Dependency and* $A_i \in$ *Theme Vertex Entities Impacted* $\succ$

   *(b)* **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Has Theme Dependency and* $A_i \in$ *Theme Vertex Entities Impacted* $\succ$

6. Find the *Segment Entities Impacted:*

   (a) **Let** *Segment Entities Impacted* = $\prec A_i \mid (A_i, B_n) \in$ *Has Theme Vertex Dependency and* $B_n \in$ *Theme Vertex Entities Impacted* $\succ$

*(b)* **Let** $\mathcal{BE} = \mathcal{BE} + \prec (A_i, B_n) \mid (A_i, B_n) \in$ *Has Theme Vertex Dependency and* $B_n \in$ *Theme Vertex Entities Impacted* $\succ$

7. Find the *Composite Components Impacted:*

   (a) **Let** *Composite Components Impacted* =

   $\prec$ A$_i$ | $(A_i, B_n) \in$ *Has Segment Dependency* and $B_n \in$ *Segment Entities Impacted* $\succ$

   **Let** $\mathcal{BE} = \mathcal{BE} + \prec (A_i, B_n) \mid (A_i, B_n) \in$ *Has Segment Dependency* and $B_n \in$ *Segment Entities Impacted* $\succ$

   (b) **Let** *Composite Components Impacted* = *Composite Components Impacted* +

   $\prec A_i \mid (A_i, B_n) \in$ *Consists of Dependency and* $B_n \in$ *Composite Components Impacted* $\succ$

   **Let** $\mathcal{BE} = \mathcal{BE} + \prec (A_i, B_n) \mid (A_i, B_n) \in$ *Consists of Dependency and* $B_n \in$ *Composite Components Impacted* $\succ$

   (c) **Extract** the set of *Composite Components* from the bag of *Composite Components Impacted.*

   (d) Repeat steps 7.(b) and 7.(c) for each hierarchical level in the *RPG*$(\mathcal{V}, \mathcal{E})$ until the newly calculated set of *Composite Components* does not change;

8. Find *Segment Entity Types Impacted:*

   (a) **Let** *Segment Entity Types Impacted* =

   $\prec B_n \mid (A_i, B_n) \in$ *Segment Entity Type Dependency and* $A_i \in$ *Segment Entities Impacted* $\succ$

   *(b)* **Let** $\mathcal{BE} = \mathcal{BE} + \prec (A_i, B_n) \mid (A_i, B_n) \in$ *Segment Entity Type Dependency and* $A_i \in$ *Segment Entities Impacted* $\succ$

9. Find *Segment Type Description Entities Impacted:*

   (a) **Let** *Segment Type Description Entities Impacted* =

   $\prec B_n \mid (A_i, B_n) \in$ *Segment Entity Type Description Dependency and* $A_i \in$ *Segment Entity Types Impacted* $\succ$

   *(b)* **Let** $\mathcal{BE} = \mathcal{BE} + \prec (A_i, B_n) \mid (A_i, B_n) \in$ *Segment Entity Type Description Dependency and* $A_i \in$ *Segment Entity Types Impacted* $\succ$

10. Find *Composite Entity Types:*

    (a) **Let** *Composite Entity Types Impacted* =
    $\prec B_n \mid (A_i, B_n) \in$ *Composite Entity Type Dependency and* $A_i \in$ *Composite Components Impacted* $\succ$

    *(b)* **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Composite Entity Type Dependency and* $A_i \in$ *Composite Components Impacted* $\succ$

11. Find *Composite Type Description Entities Impacted:*

    (a) **Let** *Composite Type Description Entities Impacted* =
    $\prec B_n \mid (A_i, B_n) \in$ *Composite Entity Type Description Dependency and* $A_i \in$ *Composite Entity Types Impacted* $\succ$

    *(b)* **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Composite Entity Type Description and* $A_i \in$ *Composite Entity Types Impacted* $\succ$

12. Find all impacted documentation entities :

    (a) **Let** $\mathcal{BV}$ =
    *Composite Components Impacted* + *Composite Entity Types Impacted* + *Composite Type Description Entities Impacted* + *Segment Entities Impacted* + *Segment Entity Types Impacted* + *Segment Type Description Entities Impacted* + *Theme Vertex Entities Impacted* + *Theme Entities Impacted* + *Thematic Context Entities Impacted*

13. Create the sets $\mathcal{V}'$ and $\mathcal{E}'$ from $\mathcal{BV}$ , $\mathcal{BE}$ :

    (a) **Extract** set $\mathcal{E}'$ from $\mathcal{BE}$

    (b) **Extract** set $\mathcal{V}'$ from $\mathcal{BV}$

14. Construct the $\beta$ $CIG(\mathcal{V}', \mathcal{E}')$ by associating the sets $\mathcal{V}'$ and $\mathcal{E}'$

**End** ( $\beta$ Graph Slice)

### 6.6.3 Results Produced by the Technique

The $\beta$ $CIG(\mathcal{V}', \mathcal{E}')$ contains the following types of impacted vertices and edges.

1. Graph Vertices: *Composite Entities* $\cup$ *Composite Type Entities* $\cup$ *Composite Type Description Entities* $\cup$ *Segment Entities* $\cup$ *Segment Type Entities* $\cup$ *Segment Type Description Entities* $\cup$ *Theme Vertex Entities* $\cup$ *Theme Entities* $\cup$ *Thematic Context Entities* $\cup$ *Composite Type Description Entities* $\cup$ *Segment Type Description Entities*
2. Graph Edges: *Co-occurs Dependencies* $\cup$ *Consists of Dependencies* $\cup$ *Has Segment Dependencies* $\cup$ *Copy Propagation Dependencies* $\cup$ *Has Theme Dependencies* $\cup$ *Has Theme Dependencies* $\cup$ *Thematic Context Dependencies* $\cup$ *Segment Entity Type Dependencies* $\cup$ *Segment Entity Type Description Dependencies* $\cup$ *Composite Entity Type Dependencies* $\cup$ *Composite Entity Type Description Dependencies*
3. Other Features : This technique identifies the direct impacts and also the indirect impacts of a theme bag.

The bags $\mathcal{BV}$ , $\mathcal{BE}$ contain the frequency of occurrence of impacted vertices and edges within the $CIG(\mathcal{V}', \mathcal{E}')$. The frequency of multiple occurrences of a particular entity can be examined with the following bag operation : $\mathcal{BV}$ [ $x$ ].

The results produced by this graph slicing technique namely, the $\beta$ $CIG(\mathcal{V}', \mathcal{E}')$ can be used to help make the following decisions identified in Chapter 3 (see table 3.1).

1. Decision No. 4 What are the inputs, outputs, processing and data?
2. Decision No. 12 Which technical documentation needs to be written/amended?

The bags $\mathcal{BV}$ , $\mathcal{BE}$ can be used to show multiple impacts in the sliced $\beta$ $CIG(\mathcal{V}', \mathcal{E}')$.

A problem with this technique is that it produces a large amount of information within the extracted graph slice. Dynamic slicing was introduced at the source code level approach to slicing [1] to alleviate this problem. However documentation cannot be executed. A second problem with this technique is that it only informs the maintainer of the entities impacted but, does not indicate the types of entity impacted. It is important to know the types of document entities impacted as it provides an insight into the impacted source code.

## 6.7 $\gamma$ Graph Slice (Weighted)

**The Problem:** *To extract the $\gamma$ $CIG(\mathcal{V}',\mathcal{E}')$ containing the following : impacted documentation based on co-occurrence of themes, impacted documentation based on descriptions of source code assignments in the documentation. An analysis of the types of document entities impacted. Impacted modules and probabilities of ripple effects between modules based on change history.*

### 6.7.1 Reasons for Developing the Technique

This graph slicing technique is an extension to the previous work on the $\alpha$ and $\beta$ graph slicing techniques. This technique attempts to improve the characterisation of the slice by considering the weights on the edges. The weighted edges which appear in the resultant $\gamma$ $CIG(\mathcal{V}',\mathcal{E}')$ are within specified range of probabilities. This probability range is specified with two parameters supplied by the person using this analysis technique. This facility enables the size of the $\gamma$ $CIG(\mathcal{V}',\mathcal{E}')$ to be reduced.

### 6.7.2 Technique Description

The technique can be described in the following way,

$\gamma$ *Graph Slice :* $RPG(\mathcal{V},\mathcal{E}) \rightarrow \gamma\ CIG(\mathcal{V}',\mathcal{E}')$

This operation can be denoted as :

$\gamma$ *Graph Slice* ( $RPG(\mathcal{V},\mathcal{E})$ , $\mathcal{TB}$,$\mathcal{LP}$, $\mathcal{HP}$ ) $\underline{\Delta}$ :

*Algorithm* $\gamma$ **Graph Slice**

**Input:** $RPG(\mathcal{V},\mathcal{E})$ , $\mathcal{TB}$,

$\mathcal{LP}$ (lowest probability to appear in the $\gamma CIG(\mathcal{V}',\mathcal{E}')$)

$\mathcal{HP}$ (highest probability to appear in the $\gamma CIG(\mathcal{V}',\mathcal{E}')$)

**Output:** $\gamma$ $CIG(\mathcal{V}',\mathcal{E}')$ , bag $\mathcal{BV}$ , bag $\mathcal{BE}$

**Begin** ( $\gamma$ Graph Slice)

1. Find the *Theme Vertex Entities Impacted:*
   - (a) **Let** *Theme Vertex Entities Impacted* =
     $\prec A_i \mid (A_i, t_n) \in$ *Has Theme Dependency and* $t_n \in \mathcal{TB} \succ$
   - *(b)* **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, t_n) \mid (A_i, t_n) \in$ *Has Theme Dependency and* $t_n \in \mathcal{TB}\succ$
2. Find the *Co-occurs Dependencies Impacted:*
   - (a) **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Co-occurs Dependency* and $A_i \in$ *Theme Vertex Entities Impacted,* $B_n \in$ *Theme Vertex Entities Impacted*$\succ$
3. Find the *Theme Vertex Entities Impacted* using *Copy Propagation Dependency*:
   - (a) **Let** *Theme Vertex Entities Impacted* = *Theme Vertex Entities Impacted* +
     $\prec B_n \mid (A_i, B_n) \in$ *Copy Propagation Dependency and* $A_i \in$ *Theme Vertex Entities Impacted* $\succ$
   - (b) **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Copy Propagation Dependency* and $A_i \in$ *Theme Vertex Entities Impacted* $\succ$
   - (c) **Extract** the set of *Theme Vertex Entities Impacted* from the bag of *Theme Vertex Entities Impacted*
   - (d) Repeat steps 3.(a), 3.(b) and 3.(c) until the newly calculated set of *Theme Vertex Entities Impacted* does not change;
4. Find the *Thematic Context Entities:*

(a) **Let** *Thematic Context Entities Impacted* =

$\prec B_n \mid (A_i, B_n) \in$ *Thematic Context Dependency and* $A_i \in$ *Theme Vertices Impacted* $\succ$

*(b)* **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Thematic Context Dependency and* $A_i \in$ *Theme Vertices Impacted* $\succ$

5. Find the *Theme Entities Impacted* (directly and indirectly impacted themes)

(a) **Let** *Theme Entities Impacted* =

$\prec B_n \mid (A_i, B_n) \in$ *Has Theme Dependency and* $A_i \in$ *Theme Vertex Entities Impacted* $\succ$

*(b)* **Let** $\mathcal{BE} = \mathcal{BE} + \prec(B_n, A_i) \mid (B_n, A_i) \in$ *Has Theme Dependency and* $A_i \in$ *Theme Vertex Entities Impacted* $\succ$

6. Find the *Segment Entities Impacted:*

(a) **Let** *Segment Entities Impacted* =

$\prec A_i \mid (A_i, B_n) \in$ *Has Theme Vertex Dependency and* $B_n \in$ *Theme Vertex Entities Impacted* $\succ$

(b) **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Has Theme Vertex Dependency* and $B_n \in$ *Theme Vertex Entities Impacted* $\succ$

7. Find the *Composite Components Impacted:*

(a) **Let** *Composite Components Impacted* =

$\prec$ A$_i$ $\mid (A_i, B_n) \in$ *Has Segment Dependency* and $B_n \in$ *Segment Entities Impacted* $\succ$

**Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Has Segment Dependency* and $B_n \in$ *Segment Entities Impacted* $\succ$

(b) **Let** *Composite Components Impacted* = *Composite Components Impacted* +

$\prec A_i \mid (A_i, B_n) \in$ *Consists of Dependency and* $B_n \in$ *Composite Components Impacted* $\succ$

**Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Consists of Dependency and* $B_n \in$ *Composite Components Impacted* $\succ$

(c) **Extract** the set of *Composite Components* from the bag of *Composite Components Impacted.*

(d) Repeat steps 7.(b) and 7.(c) for each hierarchical level in the $RPG(\mathcal{V}, \mathcal{E})$ until the newly calculated set of *Composite Components* does not change;

8. Find *Segment Entity Types Impacted:*

(a) **Let** *Segment Entity Types Impacted =*
$\prec B_n \mid (A_i, B_n) \in$ *Segment Entity Type Dependency and* $A_i \in$ *Segment Entities Impacted* $\succ$

*(b)* **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Segment Entity Type Dependency and* $A_i \in$ *Segment Entities Impacted* $\succ$

9. Find *Composite Entity Types:*

(a) **Let** *Composite Entity Types Impacted =*
$\prec B_n \mid (A_i, B_n) \in$ *Composite Entity Type Dependency and* $A_i \in$ *Composite Components Impacted* $\succ$

*(b)* **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Composite Entity Type Dependency and* $A_i \in$ *Composite Components Impacted* $\succ$

10. Find *Segment Type Description Entities Impacted:*

(a) **Let** *Segment Type Description Entities Impacted =*
$\prec B_n \mid (A_i, B_n) \in$ *Segment Entity Type Description Dependency and* $A_i \in$ *Segment Entities Types Impacted* $\succ$

*(b)* **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Segment Entity Type Description Dependency and* $B_n \in$ *Segment Entity Types Impacted* $\succ$

11. Find *Composite Type Description Entities Impacted:*

(a) **Let** *Composite Type Description Entities Impacted =*
$\prec B_n \mid (A_i, B_n) \in$ *Composite Entity Type Description Dependency and* $A_i \in$ *Composite Entity Types Impacted* $\succ$

```
   (b) Let $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ Composite Entity Type Description and $A_i \in$ Composite Entity Types Impacted $\succ$

12. Find all Document Potential Impact dependencies :

   (a) Let $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ Document Potential Impact Dependency, percentage chance of propagation $>= \mathcal{LP} \bigwedge$ percentage chance of propagation $<= \mathcal{HP}$ and $B_n \in$ Segment Entities Impacted $\succ$

13. Find Module Entities described by impacted Segment Entities:

   (a) Let Module Entities Impacted =
       $\prec$ $B_n \mid (A_i, B_n) \in$ Segment Describes Part of Module Dependency and $A_i \in$ Segment Entities Impacted $\succ$

   (b) Let $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ Segment Describes Part of Module Dependency and $A_i \in$ Segment Entities Impacted $\succ$

14. Find all Module Potential Impact dependencies impacted:

   (a) Let $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ Module Potential Impact Dependency, percentage chance of propagation $>= \mathcal{LP} \bigwedge$ percentage chance of propagation $<= \mathcal{HP}$ and $B_n \in$ Module Entities Impacted $\succ$

15. Find all impacted documentation entities :

   (a) Let $\mathcal{BV}$ =
       Composite Components Impacted + Composite Entity Types Impacted + Segment Entities Impacted + Segment Entity Types Impacted + Theme Vertex Entities Impacted + Theme Entities Impacted + Thematic Context Entities Impacted + Composite Type Description Entities Impacted + Segment Type Description Entities Impacted + Module Entities Impacted ;

16. Create the sets $\mathcal{V}'$ and $\mathcal{E}'$ from $\mathcal{BV}$ , $\mathcal{BE}$ :

   (a) Extract set $\mathcal{E}'$ from $\mathcal{BE}$
```

```
    (b) Extract set $\mathcal{V}'$ from $\mathcal{BV}$
17. Construct the $\gamma$ $CIG(\mathcal{V}', \mathcal{E}')$ by associating the sets $\mathcal{V}_\prime$ and $\mathcal{E}_\prime$.
End ( $\gamma$ Graph Slice)
```

## 6.7.3 Results Produced by the Technique

The $\gamma$ $CIG(\mathcal{V}, \mathcal{E})$ contains the following types of impacted vertices and edges.

1. Graph Vertices: *Composite Entities* $\cup$ *Composite Type Entities* $\cup$ *Segment Entities* $\cup$ *Segment Type Entities* $\cup$ *Theme Vertex Entities* $\cup$ *Theme Entities* $\cup$ *Thematic Context Entities* $\cup$ *Composite Type Description Entities* $\cup$ *Segment Type Description Entities* $\cup$ *Module Entities Impacted*

2. Graph Edges: *Co-occurs Dependencies* $\cup$ *Consists of Dependencies* $\cup$ *Has Segment Dependencies* $\cup$ *Copy Propagation Dependencies* $\cup$ *Document Potential Impact Dependencies* $\cup$ *Module Potential Impact Dependencies* $\cup$ *Has Theme Dependencies* $\cup$ *Thematic Context Dependencies* $\cup$ *Segment Entity Type Dependencies* $\cup$ *Segment Entity Type Description Dependencies* $\cup$ *Composite Entity Type Dependencies* $\cup$ *Composite Entity Type Description Dependencies* $\cup$ *Segment Describes Part of Module*

3. Other Features : This technique indicates the direct impacts, indirect impacts and the chance of propagation between segment entities.

The bags $\mathcal{BV}$ , $\mathcal{BE}$ contain the frequency of occurrence of impacted vertices and edges within the $CIG(\mathcal{V}', \mathcal{E}')$. The frequency of multiple occurrences of a particular entity can be examined with the following bag operation : $\mathcal{BV}$ [ $x$ ].

The results produced by this graph slicing technique, $\gamma$ $CIG(\mathcal{V}', \mathcal{E}')$ can be used to help make the following decisions identified in Chapter 3 (see table 3.1).

1. Decision No. 4 What are the inputs, outputs, processing and data?

2. Decision No. 12 Which technical documentation needs to be written/amended?

The bags $\mathcal{BV}$ , $\mathcal{BE}$ can be used to show multiple impacts in the sliced $\gamma$ $CIG(\mathcal{V}', \mathcal{E}')$.

A problem with this technique is that it is necessary to have a data base of release information with which to calculate the probabilities of ripple effects.

## 6.8 $\delta$ Graph Slice (Augmented)

**The Problem:** *To extract the $\delta$ $CIG(\mathcal{V}', \mathcal{E}')$ containing the following : impacted documentation based on co-occurrence of themes and impacted documentation based on descriptions of source code assignments in the documentation. An analysis of the types of document entities impacted. Impacted modules and probabilities of ripple effects between modules based on recoded expert judgement in the form of probabilities.*

### 6.8.1 Reasons for Developing the Technique

The reasons for developing this technique are to solve the problem of organisations which are without data describing previous impacts of projects. This is achieved by considering probabilities based on expert judgements. The weighted edges which appear in the resultant $\delta$ $CIG(\mathcal{V}', \mathcal{E}')$ are within specified range of probabilities. This probability range is specified with two parameters supplied by the person using this analysis technique. This facility enables the size of the $\delta$ $CIG(\mathcal{V}', \mathcal{E}')$ to be reduced.

### 6.8.2 Technique Description

The technique can be described in the following way,

$\delta$ *Graph Slice* : $RPG(\mathcal{V},\mathcal{E}) \rightarrow \delta CIG(\mathcal{V}',\mathcal{E}')$

This operation can be denoted as :

$\delta$ *Graph Slice* ( $RPG(\mathcal{V},\mathcal{E})$ , $\mathcal{TB}$,$\mathcal{LP}$, $\mathcal{HP}$) $\underline{\triangle}$ :

```
Algorithm δ Graph Slice
Input: $RPG(\mathcal{V},\mathcal{E})$ , $\mathcal{TB}$
        $\mathcal{LP}$ (lowest probability to appear in the $\delta CIG(\mathcal{V}',\mathcal{E}')$)
        $\mathcal{HP}$ (highest probability to appear in the $\delta CIG(\mathcal{V}',\mathcal{E}')$)
Output: $\delta$ $CIG(\mathcal{V}',\mathcal{E}')$ , bag $\mathcal{BV}$ , bag $\mathcal{BE}$

Begin ( δ Graph Slice)

  1. Find the Theme Vertex Entities Impacted:
     (a) Let Theme Vertex Entities Impacted =
         $\prec A_i \mid (A_i,t_n) \in$ Has Theme Dependency and $t_n \in \mathcal{TB} \succ$
     (b) Let $\mathcal{BE} = \mathcal{BE} + \prec(A_i,t_n) \mid (A_i,t_n) \in$ Has Theme Dependency and $t_n \in \mathcal{TB}\succ$
  2. Find the Co-occurs Dependencies Impacted:
     (a) Let $\mathcal{BE} = \mathcal{BE} + \prec(A_i,B_n) \mid (A_i,B_n) \in$ Co-occurs Dependency and $A_i \in$ Theme Vertex
         Entities Impacted, $B_n \in$ Theme Vertex Entities Impacted$\succ$
  3. Find the Theme Vertex Entities Impacted using Copy Propagation Dependency:
     (a) Let Theme Vertex Entities Impacted = Theme Vertex Entities Impacted +
         $\prec B_n \mid (A_i,B_n) \in$ Copy Propagation Dependency and $A_i \in$ Theme Vertex Entities
         Impacted $\succ$
     (b) Let $\mathcal{BE} = \mathcal{BE} + \prec(A_i,B_n) \mid (A_i,B_n) \in$ Copy Propagation Dependency and $A_i \in$ Theme
         Vertex Entities Impacted $\succ$
     (c) Extract the set of Theme Vertex Entities Impacted from the bag of Theme Vertex
         Entities Impacted
```

(d) Repeat steps 3.(a), 3.(b) and 3.(c) until the newly calculated set of *Theme Vertex Entities Impacted* does not change;

4. Find the *Thematic Context Entities:*

   (a) **Let** *Thematic Context Entities Impacted* =
   $\prec B_n \mid (A_i, B_n) \in$ *Thematic Context Dependency and* $A_i \in$ *Theme Vertices Impacted* $\succ$

   *(b)* **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Thematic Context Dependency and* $A_i \in$ *Theme Vertices Impacted* $\succ$

5. Find the *Theme Entities Impacted* (directly and indirectly impacted themes)

   (a) **Let** *Theme Entities Impacted* =
   $\prec B_n \mid (A_i, B_n) \in$ *Has Theme Dependency and* $A_i \in$ *Theme Vertex Entities Impacted* $\succ$

   *(b)* **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Has Theme Dependency and* $A_i \in$ *Theme Vertex Entities Impacted* $\succ$

6. Find the *Segment Entities Impacted:*

   (a) **Let** *Segment Entities Impacted* =
   $\prec A_i \mid (A_i, B_n) \in$ *Has Theme Vertex Dependency and* $B_n \in$ *Theme Vertex Entities Impacted* $\succ$

   (b) **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Has Theme Vertex Dependency* and $B_n \in$ *Theme Vertex Entities Impacted* $\succ$

7. Find the *Composite Components Impacted:*

   (a) **Let** *Composite Components Impacted* =
   $\prec A_i \mid (A_i, B_n) \in$ *Has Segment Dependency* and $B_n \in$ *Segment Entities Impacted* $\succ$
   **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Has Segment Dependency* and $B_n \in$ *Segment Entities Impacted* $\succ$

(b) **Let** *Composite Components Impacted* = *Composite Components Impacted* + $\prec A_i \mid (A_i, B_n) \in$ *Consists of Dependency and* $B_n \in$ *Composite Components Impacted* $\succ$

**Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Consists of Dependency and* $B_n \in$ *Composite Components Impacted* $\succ$

(c) **Extract** the set of *Composite Components* from the bag of *Composite Components Impacted.*

(d) Repeat steps 7.(b) and 7.(c) for each hierarchical level in the $RPG(\mathcal{V}, \mathcal{E})$ until the newly calculated set of *Composite Components* does not change;

8. Find *Composite Entity Types:*

(a) **Let** *Composite Entity Types Impacted* = $\prec B_n \mid (A_i, B_n) \in$ *Composite Entity Type Dependency and* $A_i \in$ *Composite Components Impacted* $\succ$

*(b)* **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Composite Entity Type Dependency and* $A_i \in$ *Composite Components Impacted* $\succ$

9. Find *Segment Entity Types Impacted:*

(a) **Let** *Segment Entity Types Impacted* = $\prec B_n \mid (A_i, B_n) \in$ *Segment Entity Type Dependency and* $A_i \in$ *Segment Entities Impacted* $\succ$

*(b)* **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Segment Entity Type Dependency and* $A_i \in$ *Segment Entities Impacted* $\succ$

10. Find *Composite Entity Types:*

(a) **Let** *Composite Entity Types Impacted* = $\prec B_n \mid (A_i, B_n) \in$ *Composite Entity Type Dependency and* $A_i \in$ *Composite Components Impacted* $\succ$

*(b)* **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Composite Entity Type Dependency and* $A_i \in$ *Composite Components Impacted* $\succ$

11. Find *Segment Type Description Entities Impacted:*

    (a) **Let** *Segment Type Description Entities Impacted =*

    $\prec$ $B_n$ | $(A_i, B_n)$ $\in$ *Segment Entity Type Description Dependency and* $A_i$ $\in$ *Segment Entity Types Impacted* $\succ$

    *(b)* **Let** $\mathcal{BE}$ = $\mathcal{BE}$ + $\prec(A_i, B_n)$ | $(A_i, B_n)$ $\in$ *Segment Entity Type Description Dependency and* $A_i$ $\in$ *Segment Entity Types Impacted* $\succ$

12. Find *Composite Type Description Entities Impacted:*

    (a) **Let** *Composite Type Description Entities Impacted =*

    $\prec$ $B_n$ | $(A_i, B_n)$ $\in$ *Composite Entity Type Description Dependency and* $A_i$ $\in$ *Composite Entity Types Impacted* $\succ$

    *(b)* **Let** $\mathcal{BE}$ = $\mathcal{BE}$ + $\prec(A_i, B_n)$ | $(A_i, B_n)$ $\in$ *Composite Entity Type Description and* $A_i$ $\in$ *Composite Entity Types Impacted* $\succ$

13. Find *Module Entities* described by impacted *Segment Entities:*

    (a) **Let** *Module Entities Impacted =*

    $\prec$ $\mathrm{B}_n$ | $(A_i, B_n)$ $\in$ *Segment Describes Part of Module Dependency and* $A_i$ $\in$ *Segment Entities Impacted* $\succ$

    (b) **Let** $\mathcal{BE}$ = $\mathcal{BE}$ + $\prec(A_i, B_n)$ | $(A_i, B_n)$ $\in$ *Segment Describes Part of Module Dependency* and $A_i$ $\in$ *Segment Entities Impacted* $\succ$

14. Find all *Expert Judgement Concerning Modules* for *Impacted Segment Entities:*

    (a) **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n)$ | $(A_i, B_n)$ $\in$ *Module Expert Judgement Dependency,percentage chance of propagation* $>=$ $\mathcal{LP}$ $\bigwedge$ *percentage chance of propagation* $<=$ $\mathcal{HP}$ and, $B_n$ $\in$ *Module Entities Impacted* $\succ$

15. Find all *Expert Judgement Concerning Segment Entities* for *Impacted Segment Entities:*

    (a) **Let** $\mathcal{BE}$ = $\mathcal{BE}$ + $\prec(A_i, B_n)$ | $(A_i, B_n)$ $\in$ *Document Expert Judgement Dependency, percentage chance of propagation* $>=$ $\mathcal{LP}$ $\bigwedge$ *percentage chance of propagation* $<=$ $\mathcal{HP}$, *and* $B_n$ $\in$ *Segment Entities Impacted* $\succ$

16. Find all impacted documentation entities :

    (a) **Let** $\mathcal{BV}$ =
    *Composite Components Impacted* + *Composite Entity Types Impacted* + *Segment Entities Impacted* + *Segment Entity Types Impacted* + *Theme Vertex Entities Impacted* + *Theme Entities Impacted* + *Thematic Context Entities Impacted* + *Module Entities Impacted* + *Segment Type Description Entities Impacted* + *Composite Type Description Entities Impacted*

17. Create the sets $\mathcal{V}'$ and $\mathcal{E}'$ from the bags $\mathcal{BV}$ , $\mathcal{BE}$ :

    (a) **Extract** set $\mathcal{V}'$ from $\mathcal{BV}$

    (b) **Extract** set $\mathcal{E}'$ from $\mathcal{BE}$

18. Construct the $\delta$ $CIG(\mathcal{V}', \mathcal{E}')$ by associating the sets $\mathcal{V}_{\prime}$ and $\mathcal{E}_{\prime}$

**End** ( $\delta$ Graph Slice)

---

### 6.8.3 Results Produced by the Technique

The $\delta$ $CIG(\mathcal{V}', \mathcal{E}')$ contains the following types of impacted vertices and edges:

1. Graph Vertices: *Composite Components Impacted* ∪ *Composite Type Entities* ∪ *Segment Entities* ∪ *Segment Type Entities* ∪ *Composite Type Description Entities* ∪ *Segment Type Description Entities* ∪ *Theme Vertex Entities* ∪ *Theme Entities* ∪ *Thematic Context Entities* ∪ *Module Entities Impacted*

2. Graph Edges: *Co-occurs Dependencies* ∪ *Consists of Dependencies* ∪ *Has Segment Dependencies* ∪ *Segment Entity Type Dependencies* ∪ *Segment Entity Type Description Dependencies*

$\cup$ *Composite Entity Type Dependencies* $\cup$ *Composite Entity Type Description Dependencies* $\cup$ *Has Theme Dependencies* $\cup$ *Has Theme Vertex Dependencies* $\cup$ *Thematic Context Dependencies* $\cup$ *Copy Propagation Dependencies* $\cup$ *Module Expert Judgement Dependencies* $\cup$ *Document Expert Judgement Dependencies*

3. Other Features : This technique indicates the direct impacts, indirect impacts and the chance of propagation between segment entities. The chance of propagation is based maintainers expert judgement.

The bags $\mathcal{BV}$ , $\mathcal{BE}$ contain the frequency of occurrence of impacted vertices and edges within the $CIG(\mathcal{V}', \mathcal{E}')$. The frequency of multiple occurrences of a particular entity can be examined with the following bag operation : $\mathcal{BV}$ [ $x$ ]. The results produced by this graph slicing technique namely the, $\delta$ $CIG(\mathcal{V}', \mathcal{E}')$ can be used to help make the following decisions identified in Chapter 3 (see table 3.1).

1. Decision No. 4 What are the inputs, outputs, processing and data?
2. Decision No. 12 Which technical documentation needs to be written/amended?

The bags $\mathcal{BV}$ , $\mathcal{BE}$ can be used to show multiple impacts in the sliced $CIG(\mathcal{V}', \mathcal{E}')$.

The main problem in this technique and its predecessors, the $\alpha$ , $\beta$ , and $\gamma$ techniques is that they do not give a good characterisation of any of the source code constructs mentioned within the impacted documentation. This is important since the documentation describes the source code constructs.

## 6.9 $\epsilon$ Graph Slice (Attributed)

**The Problem:** *To extract the $\epsilon$ $CIG(\mathcal{V}', \mathcal{E}')$ impacted documentation based on co-occurrence of themes and impacted documentation based on descriptions of source code assignments in the documentation. A list of source code entities mentioned in impacted document entities.*

### 6.9.1 Reasons for Developing this Technique

This technique has been developed to improve the characterisation of the composition of the slice produced by the $\delta$ Graph Slice. The slice aims to link the change proposal with the operational software constructs which are described in the documentation.

### 6.9.2 Technique Description

The technique can be described in the following way,

$\epsilon$ *Graph Slice : RPG*$(\mathcal{V}, \mathcal{E}) \rightarrow \epsilon$ *CIG*$(\mathcal{V}', \mathcal{E}')$

This operation can be denoted as :

$\epsilon$ *Graph Slice* ( *RPG*$(\mathcal{V}, \mathcal{E})$ , $\mathcal{TB}$ ) $\underline{\triangle}$ :

---

*Algorithm* $\epsilon$ **Graph Slice**

**Input:** *RPG*$(\mathcal{V}, \mathcal{E})$ , $\mathcal{TB}$

**Output:** $\epsilon$ *CIG*$(\mathcal{V}', \mathcal{E}')$ , $\mathcal{BV}$ , $\mathcal{BE}$

**Begin** ( $\epsilon$ Graph Slice)

1. Find the *Theme Vertex Entities Impacted:*
   (a) **Let** *Theme Vertex Entities Impacted* =
   $\prec A_i \mid (A_i, t_n) \in$ *Has Theme Dependency and* $t_n \in \mathcal{TB} \succ$
   (b) **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, t_n) \mid (A_i, t_n) \in$ *Has Theme Dependency and* $t_n \in \mathcal{TB}\succ$
2. Find the *Co-occurs Dependencies Impacted:*
   (a) **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Co-occurs Dependency* and $A_i \in$ *Theme Vertex Entities Impacted,* $B_n \in$ *Theme Vertex Entities Impacted*$\succ$

3. Find the *Theme Vertex Entities Impacted* using *Copy Propagation Dependency*:

   (a) **Let** *Theme Vertex Entities Impacted* = *Theme Vertex Entities Impacted* + $\prec B_n \mid (A_i, B_n) \in$ *Copy Propagation Dependency and* $A_i \in$ *Theme Vertex Entities Impacted* $\succ$

   (b) **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Copy Propagation Dependency* and $A_i \in$ *Theme Vertex Entities Impacted* $\succ$

   (c) **Extract** the set of *Theme Vertex Entities Impacted* from the bag of *Theme Vertex Entities Impacted*

   (d) Repeat steps 3.(a), 3.(b) and 3.(c) until the newly calculated set of *Theme Vertex Entities Impacted* does not change;

4. Find the *Thematic Context Entities:*

   (a) **Let** *Thematic Context Entities Impacted* = $\prec B_n \mid (A_i, B_n) \in$ *Thematic Context Dependency and* $A_i \in$ *Theme Vertices Impacted* $\succ$

   *(b)* **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Thematic Context Dependency and* $A_i \in$ *Theme Vertices Impacted* $\succ$

5. Find the *Theme Entities Impacted* (directly and indirectly impacted themes)

   (a) **Let** *Theme Entities Impacted* = $\prec B_n \mid (A_i, B_n) \in$ *Has Theme Dependency and* $A_i \in$ *Theme Vertex Entities Impacted* $\succ$

   *(b)* **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Has Theme Dependency and* $A_i \in$ *Theme Vertex Entities Impacted* $\succ$

6. Find the *Segment Entities Impacted:*

   (a) **Let** *Segment Entities Impacted* = $\prec A_i \mid (A_i, B_n) \in$ *Has Theme Vertex Dependency and* $B_n \in$ *Theme Vertex Entities Impacted* $\succ$

10. Find *Segment Type Description Entities Impacted:*

    (a) **Let** *Segment Type Description Entities Impacted =*
    $\prec B_n \mid (A_i, B_n) \in$ *Segment Entity Type Description Dependency and* $A_i \in$ *Segment Entity Types Impacted* $\succ$

    *(b)* **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Segment Entity Type Description Dependency and* $A_i \in$ *Segment Entity Types Impacted* $\succ$

11. Find *Composite Type Description Entities Impacted:*

    (a) **Let** *Composite Type Description Entities Impacted =*
    $\prec B_n \mid (A_i, B_n) \in$ *Composite Entity Type Description Dependency and* $A_i \in$ *Composite Entity Types Impacted* $\succ$

    *(b)* **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Composite Entity Type Description and* $A_i \in$ *Composite Entity Types Impacted* $\succ$

12. Find *Module Entities* described by impacted *Segment Entities:*

    (a) **Let** *Module Entities Impacted =*
    $\prec \mathrm{B}_n \mid (A_i, B_n) \in$ *Segment Describes Part of Module Dependency and* $A_i \in$ *Segment Entities Impacted* $\succ$

    (b) **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Segment Describes Part of Module Dependency* and $A_i \in$ *Segment Entities Impacted* $\succ$

13. Find *System Entities* Impacted:

    (a) **Let** *System Entities Impacted =*
    $\prec B_n \mid (A_i, B_n) \in$ *Module belongs to System Dependency* $A_i \in$ *Module Entities Impacted* $\succ$

    (b) **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Module belongs to System Dependency* and $A_i \in$ *Module Entities Impacted* $\succ$

14. Find *Associated System Entities* Impacted:

(a) **Let** *Associated System Entities Impacted* =
$\prec A_i \mid (A_i, B_n) \in$ *Associated System Dependency* $B_n \in$ *System Entities Impacted* $\succ$

(b) **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Associated System Dependency* and $B_n \in$ *System Entities Impacted* $\succ$

15. Find *Module Type Entities* Impacted:

(a) **Let** *Module Type Entities Impacted* =
$\prec B_n \mid (A_i, B_n) \in$ *Module Type Dependency* $A_i \in$ *Module Entities Impacted* $\succ$

(b) **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Module Type Dependency and* $A_i \in$ *Module Entities Impacted* $\succ$

16. Find *Module Test Entities* Impacted:

(a) **Let** *Module Test Entities Impacted* =
$\prec B_n \mid (A_i, B_n) \in$ *Module Test Required Dependency* $A_i \in$ *Module Entities Impacted* $\succ$

(b) **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Module Test Required Dependency and* $A_i \in$ *Module Entities Impacted* $\succ$

17. Find *System Test entities* Impacted:

(a) **Let** *System Test Entities Impacted* =
$\prec B_n \mid (A_i, B_n) \in$ *System Test Required Dependency* $A_i \in$ *Module Entities Impacted*$\succ$

(b) **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *System Test Required Dependency and* $A_i \in$ *Module Entities Impacted* $\succ$

18. Find *Data File Entities* Impacted:

(a) **Let** *Data File Entities Impacted* =
$\prec B_n \mid (A_i, B_n) \in$ *Module Uses Data File Dependency* $A_i \in$ *Module Entities Impacted*$\succ$

(b) **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Module Uses Data File Dependency and* $A_i \in$ *Module Entities Impacted* $\succ$

19. Find *Job control language Entities Impacted*:

(a) **Let** *Job Control Language Entities Impacted* =

$\prec B_n \mid (A_i, B_n) \in$ *Module uses Job Control Language Dependency* $A_i \in$ *Module Entities Impacted*$\succ$

(b) **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Module uses Job Control Language Dependency and* $A_i \in$ *Module Entities Impacted* $\succ$

20. Find *Map Base Entities* Impacted:

(a) **Let** *Map uses Base Entities Impacted* =

$\prec B_n \mid (A_i, B_n) \in$ *Module uses Map Base Dependency* $A_i \in$ *Module Entities Impacted*$\succ$

(b) **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Module uses Map Base Dependency and* $A_i \in$ *Module Entities Impacted* $\succ$

21. Find *System Entities Supplying Data* Impacted:

(a) **Let** *System Entities Supplying Data* =

$\prec A_i \mid (A_i, B_n) \in$ *System Supplies Data to Dependency* $B_n \in$ *System Entities Impacted*$\succ$

(b) **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *System Supplies Data to Dependency and* $B_n \in$ *System Entities Impacted* $\succ$

22. Find *System Entities Receiving Data* Impacted:

(a) **Let** *System Entities Receiving Data* =

$\prec A_i \mid (A_i, B_n) \in$ *System Entity Receives data from Dependency* $B_n \in$ *System Entities Impacted*$\succ$

(b) **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *System Entity Receives data from Dependency and* $B_n \in$ *System Entities Impacted* $\succ$

23. Find *System Entities Described by Segments* Impacted:

(a) **Let** *System Entities Described by Segments* =

$\prec B_n \mid (A_i, B_n) \in$ *Segment Describes Part of System Dependency* $A_i \in$ *Segment Entities Impacted*$\succ$

(b) **Let** $\mathcal{BE} = \mathcal{BE} + \prec(A_i, B_n) \mid (A_i, B_n) \in$ *Segment Describes Part of System Dependency and* $A_i \in$ *Segment Entities Impacted* $\succ$

24. Create bag $\mathcal{E}$ by performing a bag addition on all bags of impacted Entities

    (a) **Let** $\mathcal{BV}$ = *Theme Vertex Entities Impacted* + *Thematic Context Entities* + *Theme Entities Impacted* + *Composite Entities Impacted* + *Composite Type Entities Impacted* + *Segment Entities Impacted* + *Segment Entity Types Impacted* + *Segment Type Description Entities Impacted* + *Composite Type Description Entities Impacted* + *Module Entities Impacted* + *System Entities Impacted* + *Associated System Entities Impacted* + *Module Type Entities Impacted* + *Module Test Entities Impacted* + *System Test Entities Impacted* + *Data File Entities Impacted* + *Job Control Language Entities Impacted* + *Map Base Entities Impacted* + *System Entities Supplying Data to* + *System Entities Receiving Data* + *System Entities Described by Segments Impacted*

25. Create the sets $\mathcal{V}'$ and $\mathcal{E}'$ from $\mathcal{BV}$ , $\mathcal{BE}$ :

    (a) **Extract** set $\mathcal{V}'$ from $\mathcal{BV}$

    (b) **Extract** set $\mathcal{E}'$ from $\mathcal{BE}$

26. Construct the $\epsilon$ $CIG(\mathcal{V}', \mathcal{E}')$ by associating the sets $\mathcal{V}_\prime$ and $\mathcal{E}_\prime$

**End** ( $\epsilon$ Graph Slice)

### 6.9.3 Results Produced by the Technique

The $\epsilon$ $CIG(\mathcal{V}', \mathcal{E}')$ produced contains the following types of impacted vertices and edge:

1. Graph Vertices: *Composite Entities* ∪ *Composite Type Entities* ∪ *Segment Entities* ∪ *Segment Type Entities* ∪ *Composite Type Description Entities* ∪ *Segment Type Description Entities* ∪ *Module Entities* ∪ *Module Type Entities* ∪ *Module Test File Entities* ∪ *System Test File Entities* ∪ *Module Data File Entities* ∪ *Dictionary Region Entities* ∪ *Job Control Language*

*Entities* ∪ *Map Base Entities* ∪ *System Entities* ∪ *Theme Vertex Entities* ∪ *Theme Entities* ∪ *Thematic Context Entities*

2. Graph Edges: *Co-occurs Dependencies* ∪ *Consists of Dependencies* ∪ *Has Segment Dependencies* ∪ *Segment Entity Type* ∪ *Segment Entity Type Description* ∪ *Composite Entity Type* ∪ *Composite Entity Type Description* ∪ *Has Theme Dependencies* ∪ *Has Theme Vertex Dependencies* ∪ *Has Context Dependencies* ∪ *Copy Propagation Dependencies* ∪ *Segment Describes Part of Module Type Dependencies* ∪ *Module Test Required Dependencies* ∪ *Module System Test Required Dependencies* ∪ *Module uses Data File Dependencies* ∪ *Module uses Data Dictionary Dependencies* ∪ *Module uses Job Control Language Dependencies* ∪ *Module uses Map Base Dependencies* ∪ *Module belongs to System Dependencies* ∪ *System Supplies Data to Dependencies* ∪ *System Data Received from Dependencies*

3. Other Features : This technique identifies the source code objects described within segment entities.

The bags $\mathcal{BV}$ , $\mathcal{BE}$ contain the frequency of occurrence of impacted vertices and edges within the $\epsilon\ CIG(\mathcal{V}', \mathcal{E}')$. The frequency of multiple occurrences of a particular entity can be examined with the following bag operation : $\mathcal{BV}$ [ $x$ ].

The results produced by this graph slicing technique namely, the $\epsilon\ CIG(\mathcal{V}', \mathcal{E}')$ can be used to help make the following decisions identified in Chapter 3 (see table 3.1).

1. Decision No. 4 What are the inputs, outputs, processing and data?
2. Decision No. 5 What is the implementation strategy?
3. Decision No. 6 What is the testing strategy?
4. Decision No. 10 Which are the impacted system features?
5. Decision No. 12 Which technical documentation needs to be written/amended?
6. Decision No. 13 Which test data is required?

The bags $\mathcal{BV}$ , $\mathcal{BE}$ can be used to show multiple impacts in the sliced $\epsilon\ CIG(\mathcal{V}', \mathcal{E}')$.

## 6.10 Summary

In this Chapter an model of a change proposal has been presented. A technique for constructing slicing criterion by extracting the themes from this change proposal has been developed. The resultant slice criterion is called a theme bag. Several techniques for slicing the $RPG(\mathcal{V}, \mathcal{E})$ are presented. The theme bag is used as a slicing criterion for all of the $RPG(\mathcal{V}, \mathcal{E})$ slicing techniques.

The slicing techniques developed are based on source code slicing techniques. It has been shown how the principles of the work of Weiser [91] can be applied to higher levels of abstraction than source code, such as documentation in order to give an insight into the direct impacts and indirect impacts of a proposed change to a software system. At the source code level, slicing as been extended to dynamic slicing to consider the statements in a source code slice which were only executed [1]. This reduced the size of the resultant slice and helps the maintainer focus on important contents of a slice.

Theoretically this has been achieved at the documentation level by producing probabilities of ripple effects based on release information and expert judgements about probable ripple effects. These probabilities help reduce the size of the graph slice. This is achieved by deleting weighted edges and associated start and stop vertices whose weights are not within a specific range probabilities. The range be selected by a maintenance manager or a maintainer.

The $RPG(\mathcal{V}, \mathcal{E})$ analysis techniques can be combined by adding the resulting $CIG(\mathcal{V}', \mathcal{E}')$ s together. For example the splicing operation can be used for this purpose.

$$\textit{Graph Splice}\ CIG(\mathcal{V}_1', \mathcal{E}_1') \sqcup CIG(\mathcal{V}_2', \mathcal{E}_2')$$

The resultant spliced graph will only have one occurrence of any duplicate vertices or edges.

By associating attributes with segment entities the value of the model can be intuitively increased. Attributes can describe entities and provide information about them. For example if it is deduced that a particular segment entity is impacted then all of the attributes for that segment could be implicated. By varying the themes in the $\mathcal{TB}$ it is possible to study the differing impacts on the system. This allows a maintainer or maintenance manager to conduct an analysis of the impact of differing hypothesised changes.

To summarise the $RPG(\mathcal{V},\mathcal{E})$ slicing techniques presented in this Chapter can be placed within the impact analysis classification scheme presented in Chapter 3, (see table 3.1).

| NO. | DECISION | TECHNIQUES |
|---|---|---|
| No. 1 | What are the cost benefits? | |
| No. 2 | What are the time scales required? | |
| No. 3 | What are the cost benefits? | |
| No. 4 | What are the inputs, outputs, processing processing and data? | $\alpha$ , $\beta$ , $\gamma$ , $\delta$ , $\epsilon$ |
| No. 5 | What is the implementation strategy? | $\delta$ $\epsilon$ |
| No. 6 | What is the testing strategy? | $\epsilon$ |
| No. 7 | What are the cost and benefits? | |
| No. 8 | How can the project be categorised? | |
| No. 9 | What is the priority for the project? | |
| No. 10 | Which are the impacted system features? | $\epsilon$ |
| No. 11 | What are the estimates now? | |
| No. 12 | Which technical documentation needs to be written/amended? | $\alpha$ , $\beta$ , $\gamma$ , $\delta$ , $\epsilon$ |
| No. 13 | Which test data is required? | $\epsilon$ |
| No. 14 | What is the source code impacted? | $\delta$ |
| No. 15 | What caused a particular defect? | |
| No. 16 | What caused a particular defect? | |

Table 6.1: Theoretical Evaluation of the Graph Slices

It can be seen in table 6.1 that the $RPG(\mathcal{V},\mathcal{E})$ slicing techniques can be used to :

1. The identify the inputs, outputs, processing affected by the proposed change.

2. The identify the documentation needed to be maintained.

3. The identify the testing required for the proposed change.

4. The identify the software constructs which are impacted by the proposed change.

The next Chapter presents a prototype implementation of the $RPG(\mathcal{V},\mathcal{E})$ model construction techniques, and the thematic graph slicing techniques.

# Chapter 7

# Prototype Implementation

## 7.1 Introduction

This chapter describes a prototype tool, MAGENTA (MAnaGemENt Technical impact Analyser), to support two aspects of ripple effect analysis. These are the recording of the $RPG(\mathcal{V}, \mathcal{E})$ and the thematic slicing. A definition of a prototype is given and its usefulness in evaluating the ideas in this thesis are discussed. An overview of the prototype implementation for the model and model analysis techniques described in Chapter 4, 5, and 6 is presented before introducing a graph description language which facilitates the entry of the documentation graphs. Finally, details are given to show how the software tools can be used to automate the construction and analysis of the ripple propagation graph ($RPG(\mathcal{V}, \mathcal{E})$).

## 7.2 Rationale and Overview

A prototype implementation is defined in [46] as :

*A preliminary type, form or instance of a system that serves as a model for the later stages, or for the final complete version of the system.*

A prototype incorporates components of the actual product. Typically, a prototype exhibits functional capabilities, low reliability and inefficient performance. A prototype is useful for :

1. providing feedback on the technique;
2. determining the feasibility of the technique;
3. aiding the investigation of other issues such a s technical issues in the proposed tool.

Another reason for developing a prototype implementation of MAGENTA is that it was not possible to write a reasonably complete set of specifications due to an incomplete understanding of the problem. Iterative expansion and refinement of the system were therefore required. This was because prior to the development of the prototype, the method had not been completely refined. This meant it was not possible to define the prototype implementation of MAGENTA without some exploratory development. Often it was not clear how to proceed with the next enhancement of the method until the current version of the method was implemented and evaluated.

A model and method can be examined from a theoretical point of view but can only be evaluated with small examples. Software tools provide automation which enables more realistically sized models to be produced. The use of the prototype implementation of MAGENTA can facilitate the evaluation of the model and provide pointers as to how it can be exploited in a real world setting.

The prototype implementation of MAGENTA was developed using iterative refinements based on its operation within several testbeds, see figure 7.1 . Each of the testbeds are examples of the structure of software and its associated documentation described using the $RPG(\mathcal{V}, \mathcal{E})$. Each version of the prototype was evaluated with respect to the validity of the tool, that is, the correctness of the results produced. The strengths and weaknesses were also identified to provide feedback for further development of the method.

The requirements of the prototype implementation of MAGENTA are :

1. Easy entry of the graph model.
2. Efficient representation of the graph model.
3. Efficient retrieval of all or part of the graph model.

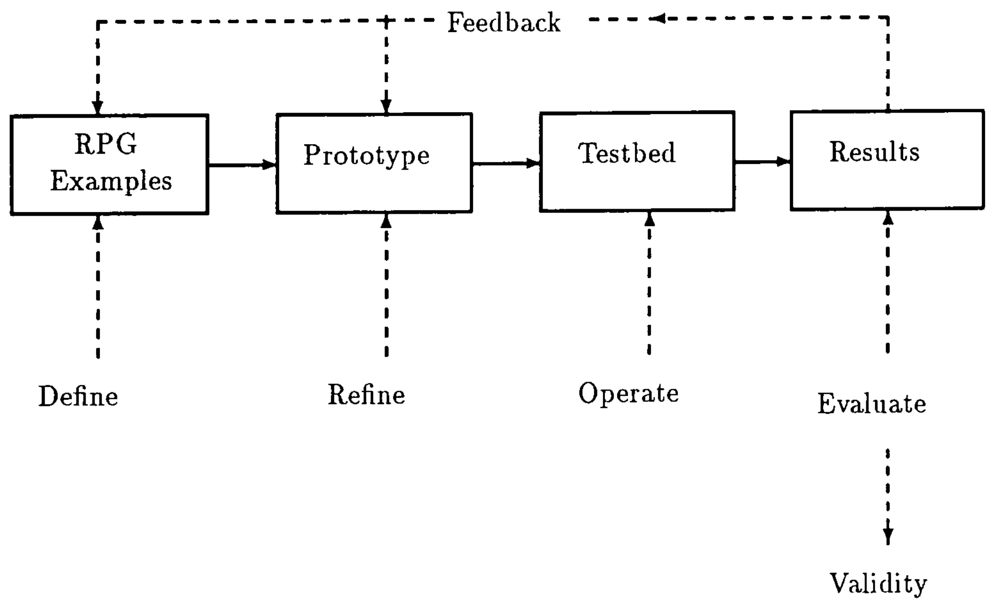


Figure 7.1: A Diagram of the Prototype Development Cycle

In order to satisfy these requirements in the prototype, it is necessary to make use of data base technology. Conventional languages are capable of implementing a data base using a file based approach. However there are some limitations in adopting this approach. For example, the data base will store the graph model. As the method develops, so the data base will also change, to reflect revisions in the graph model. Changes in the data base will also propagate changes through the code providing the graph operations. In conventional programming languages an application is data-dependent. It is impossible to change the storage or access strategy without affecting the application. Large amounts of source code would also have to be written to facilitate retrieval of information from the graph data base. The use of a dedicated data base system will address these problems. Three well known conventional data models are relational, hierarchical, and network models [86]. Hierarchical data base models have a tree structure, network data base models have a graph structure and relational data base models consist of rectangular tables. When providing a data base for the storage of a graph model which is likely to develop in unforeseen ways it is necessary to choose a data model which can easily be changed. In the hierarchical and network approach the links between the data are built into the data base hence making changes to its structure difficult. In the relational model, links are established in the data itself making unforeseen changes easier to implement. Deductive data bases and programs can be viewed as a powerful extension to the relational data base model, the extra power coming from the ability to specify rules for drawing conclusions about the stored data.

A deductive data base provides a general purpose question-answering system based on the facts it contains. The set of facts necessary for question answering can be viewed as a statement, derived from premises rather than from assumptions. Therefore, the advantage of such a data base system is its deductive power. The name of the relationship between stored objects is called a predicate, the various objects connected by the predicate are called arguments and each instance of a predicate is called a clause. There are two different types of clauses, namely facts and rules. A fact is a predicate with a number of arguments and a rule provides a description of how conclusions can be drawn from facts. Further information concerning the link between logic and data bases can be found in [39].

The example in figure 7.2 shows two facts written in the programming language Prolog (*Pro*gramming in *Log*ic). The `has-theme(VERTEX,THEME)` fact indicates a relationship between the two arguments [1] `VERTEX` and `THEME` and the, `has-theme-vertex(SEGMENT,VERTEX)` fact indicates a relationship between the arguments `SEGMENT` and `VERTEX`.

```
has-theme(VERTEX,THEME),
has-theme-vertex(SEGMENT,VERTEX)
```

Figure 7.2: An Example of a Prolog Fact

A rule will have a predicate with arguments enclosed in parentheses; but it will also have information following. Facts can be thought of as storing information and rules can be thought of as ways of drawing conclusions about the stored information.

Consider the example of a Prolog rule in figure 7.3. This rule indicates that a particular `SEGMENT` is coded with a particular `THEME` if (denoted by the :- symbol) the `VERTEX` has that particular `THEME` and the `SEGMENT` has this particular `VERTEX`. Therefore given a `SEGMENT` the `THEME` can be deduced or alternatively given a `THEME`, the `SEGMENT` with that particular `THEME` can be deduced.

It is very easy to draw conclusions by combining different clauses.

[1] the arguments in a Prolog fact are often referred to as objects

```
segment-theme(THEME,SEGMENT):-
        has-theme(VERTEX,THEME),
        has-theme-vertex(SEGMENT,VERTEX).
```

Figure 7.3: An Example of a Prolog Rule

One of the most attractive features of deductive data bases is that the nature of a problem to be solved can be described, rather than listing the steps the computer should take to solve this problem.

Logic notation provides a precise language for the explicit expression of goals knowledge and assumptions. It provides the foundation for deducing goals from premises, for studying the truth or falsity of statements given the truth of falsity of other statements, for establishing the consistency and for verifying arguments.

In this work the prototype implementation provides the user with information that allows an assessment of where the ripple effect of a change may be. The prototype implementation of MAGENTA consists of two tools:

1. the graph construction tool which automates the $RPG(\mathcal{V},\mathcal{E})$ crystallisation, edge parameterisation and annotation operations;
2. the graph analysis tool which implements the $RPG(\mathcal{V},\mathcal{E})$ thematic slicing operations.

The diagram in figure 7.4 shows the process for constructing the graph and analysing as change proposals are formulated. The diagram illustrates a simple maintenance model starting with a change proposal from which a bag of themes is extracted to deduce the impact of the proposed change. As the system is maintained the information produced as a by product of the maintenance process such as programmer understanding, is recorded in the $RPG(\mathcal{V},\mathcal{E})$. Release information is also recorded into the $RPG(\mathcal{V},\mathcal{E})$ model. Alternatively a reverse engineering process could also be used as a source of information to build up the $RPG(\mathcal{V},\mathcal{E})$ model for a particular system. The more the system is maintained the greater the graph will be trained to approximate the interconnectivity of the system being modelled by the $RPG(\mathcal{V},\mathcal{E})$.

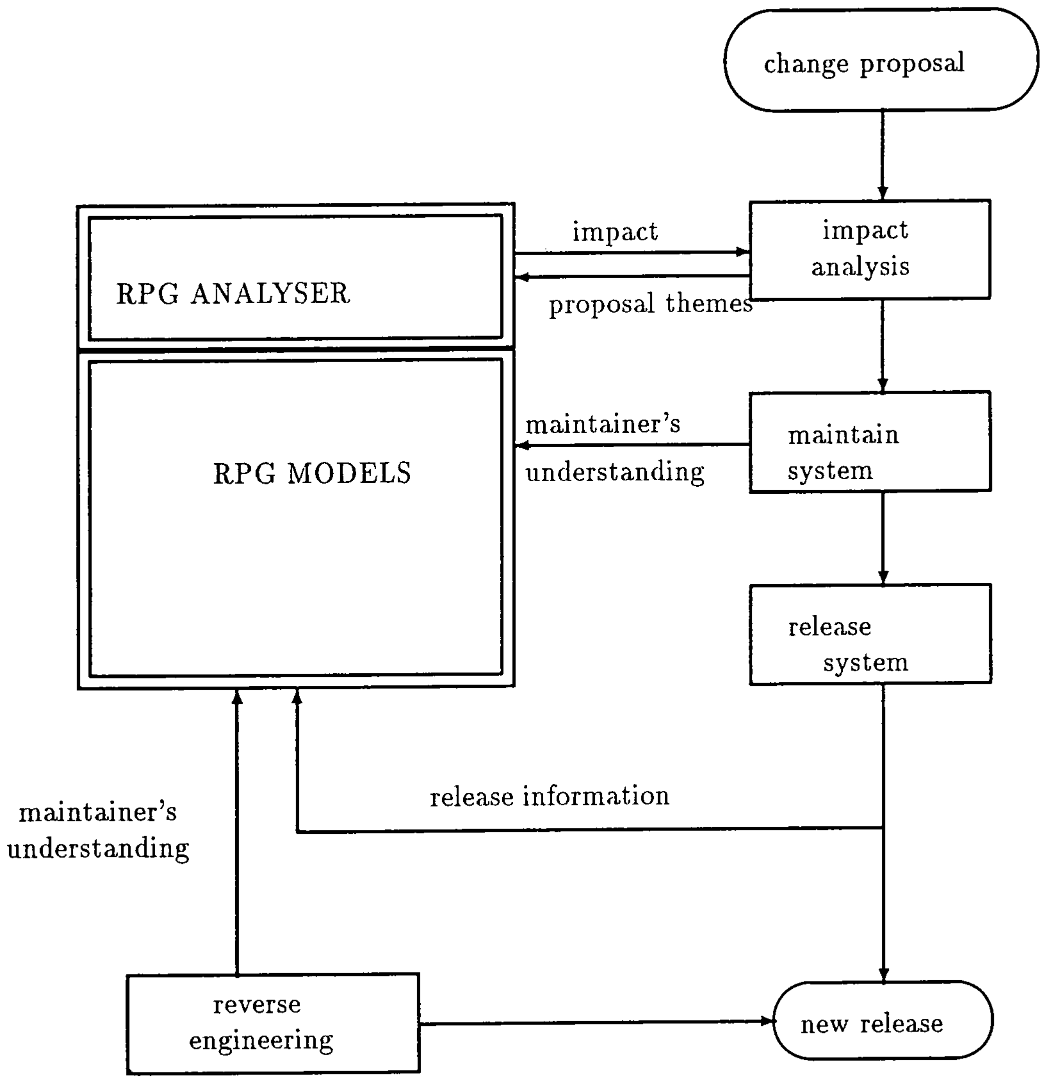


Figure 7.4: RPG Construction and Analysis

## 7.3 RPG Description Language

A graph model of the structure of documentation can be regarded as an abstraction of the material which describes the semantics of a software system. Diagraming tools can be useful for entering graph theory pictures but they can be expensive to customise to the requirements of a particular model. A more reasonable solution to the entering of information is to use a graph description language written in some notation which can be easily entered using a text editor. A graph description written in such a language also provides a convenient method of storage, and the graph can easily be analysed in this machine readable form. The requirements of such an $RPG(\mathcal{V},\mathcal{E})$ description language are the following :

1. It will record the interconnectivity within a system.
2. It will represent the structure of the documentation, including the content and interconnections between these components.
3. It will be possible to query systems expressed in the language in order to derive impact analysis information such as which other documents and source code modules are affected by a change to a particular component.

In considering the example of a simple hierarchical graph in figure 7.5, a representation of the graph can be recorded using relations between graph vertices as in figure 7.6. These relations are a description of the decompositional structure described in Section 4.3 and can easily be described in the programming language Prolog, as in figure 7.7.

In order to show how graph operations can be implemented in the form of queries consider the example of the following :

> *given a set of* **segment entities** *containing one member, determine the composite entities connected to that segment.*

Theoretically the slicing criterion of the $RPG(\mathcal{V},\mathcal{E})$ is a tuple $(g,T)$, where g denotes a specific segment in $RPG(\mathcal{V},\mathcal{E})$ and T is a subset of themes in $RPG(\mathcal{V},\mathcal{E})$ . The slicing criterion determines a projection function from a documentation segment trajectory in which only the value of themes

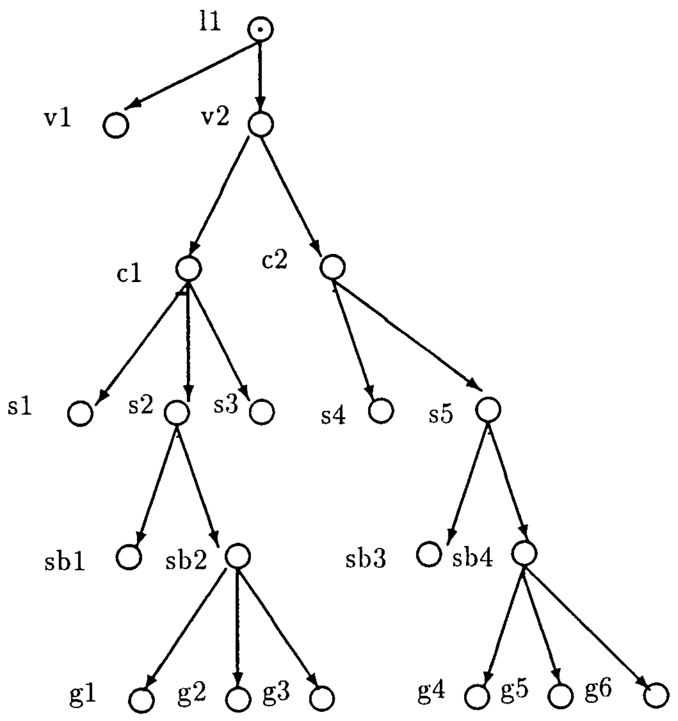


Figure 7.5: An Example of a Hierarchical Graph

| | |
|---|---|
| `l1 is composed of v1` | `s2 is composed of sb1` |
| `l1 is composed of v2` | `s2 is composed of sb2` |
| `v2 is composed of b1` | `s5 is composed of sb3` |
| `b1 is composed of c1` | `s5 is composed of sb4` |
| `b1 is composed of c2` | `sb2 has segment g1` |
| `c1 is composed of s1` | `sb2 has segment g2` |
| `c1 is composed of s2` | `sb2 has segment g3` |
| `c1 is composed of s3` | `sb4 has segment g4` |
| `c2 is composed of s4` | `sb4 has segment g5` |
| `c2 is composed of s5` | `sb4 has segment g6` |

Figure 7.6: An Example of $RPG(\mathcal{V}, \mathcal{E})$ Relations

```
consists-of(l1,v1).   consists-of(s2,sb1).
consists-of(l1,v2).   consists-of(s2,sb2).
consists-of(v2,b1).   consists-of(s5,sb3).
consists-of(b1,c1).   consists-of(s5,sb4).
consists-of(b1,c2).   has-segment(sb2,g1).
consists-of(c1,s1).   has-segment(sb2,g2).
consists-of(c1,s2).   has-segment(sb2,g3).
consists-of(c1,s3).   has-segment(sb4,g4).
consists-of(c2,s4).   has-segment(sb4,g5).
consists-of(c2,s5).   has-segment(sb4,g6).
```

Figure 7.7: An Example of $RPG(\mathcal{V}, \mathcal{E})$ Relations in Prolog

in T are preserved. This $RPG(\mathcal{V}, \mathcal{E})$ slicing operation provides a restriction transformation which gives a mapping from one set to another and filters out any vertices in the $RPG(\mathcal{V}, \mathcal{E})$ not satisfying a particular slicing criterion. Formally this consists of two steps, the second of which iterates until the top of the document hierarchy is reached. The algorithm is:

**Algorithm**, (*Find composite entities affected* [2].)

1. Find the *Composite Components Impacted:*

   (a) **Let** *Composite Components Impacted* =

   $\prec$ $A_i$ | $(A_i, B_n) \in$ *Has Segment Dependency* and $B_n \in$ *Segment Entities Impacted* $\succ$

   **Let** $\mathcal{BE}$ = $\mathcal{BE}$ + $\prec(A_i, B_n)$ | $(A_i, B_n) \in$ *Has Segment Dependency* and $B_n \in$ *Segment Entities Impacted* $\succ$

   (b) **Let** *Composite Components Impacted* = *Composite Components Impacted* +

   $\prec A_i$ | $(A_i, B_n) \in$ *Consists of Dependency and* $B_n \in$ *Composite Components Impacted* $\succ$

   **Let** $\mathcal{BE}$ = $\mathcal{BE}$ + $\prec(A_i, B_n)$ | $(A_i, B_n) \in$ *Consists of Dependency and* $B_n \in$ *Composite Components Impacted* $\succ$

   (c) **Extract** the set of *Composite Components* from the bag of *Composite Components Impacted.*

[2]This algorithm is taken from Section 6.5.2 of Chapter 6

(d) Repeat step 1.(b) & 1.(c) for each hierarchical level in the $RPG(\mathcal{V},\mathcal{E})$ until the newly calculated set of *Composite Components* does not change;

This can be implemented in Prolog as given in figure 7.8. This is a particularly illustrative example of how a small amount of Prolog source code can implement some queries.

```
doc-affected(G):-
          has-segment(F,G),
          consists-of2(R,F),
          fail.
consists-of2(R,F):-
          consists-of(R,F).
consists-of2(R,F):-
          consists-of(F1,F),
          consists-of2(R,F1).
```

Figure 7.8: An $RPG(\mathcal{V},\mathcal{E})$ Query Written in Prolog

The segment is supplied in the argument G of `doc-affected` and the first composite entity F is determined with the has-segment(F,G) clause. The object F is then used to find the next composite entity F1 in the hierarchy using the recursive consists-of2 rule. There are two consists-of2 rules, the first one is for the detection of the top of the hierarchy in the graph and the second rule is for the detection of other composite entities which are not located at the bottom or top of the hierarchy. The second consists-of2 rule implements the recursion i.e. the repetition of the second step of the algorithm. So for example the query `doc-affected(g1)` applied to the graph in figure 7.5 determines the composite entities which are impacted by a change to the segment entity g1 and would produce the following output :

```
sb2
s2
c1
b1
v2
l1
```

It is therefore possible to conclude, for example, that composite entity c1 is connected to segment entity g1.

This indicates the feasibility and power of the Prolog representation of the relations and the analysis of them. The problem in inventing new notations is that tools must be built for interpreting these notations or for transforming these notations into programming language representations understood by compilers or interpreters. The use of Prolog as a documentation description language ensures that a powerful amount of automatic processing is available without implementing additional source code routines.

Each graph dependency mentioned in Chapter 4 is described using a **Ripple Propagation Graph Description Language (RPDGL)** sentence. Each RPDGL sentence has the same syntax and semantics as a Prolog fact. The name of each fact has exactly the same name as that dependency in the graph that it represents. The objects between which there is a dependency are enclosed in parenthesis and also have the same name as the graph vertices described in Chapter 4. Therefore a mapping can be clearly seen between the method and its realisation in a prototype implementation of MAGENTA.

The following four tables give example definitions of each sentence in the language. The sentences in table 7.1 describe the Hierarchical Interconnection Graph ($HIG(\mathcal{V},\mathcal{E})$), in table 7.2 describe the Thematic Interconnection Graph ($TIG(\mathcal{V},\mathcal{E})$), in table 7.3 describe the Source Attributes Graph ($SAG(\mathcal{V},\mathcal{E})$) and in table 7.4 describe the Weighted Interconnection Graph ($WIG(\mathcal{V},\mathcal{E})$).

| |
|---|
| rpg(system). |
| consists-of(composite-entity,composite-entity). |
| has-segment(composite-entity,segment-entity). |
| has-theme-vertex(segment-entity,theme-vertex-entity). |
| composite-entity-type(composite-entity,composite-entity-type). |
| segment-entity-type(segment-entity,segment-entity-type). |
| composite-entity-type-description(composite-entity-type,type-description). |
| segment-entity-type-description(segment-entity-type,type-description). |

Table 7.1: Hierarchical Interconnection Graph Description

| |
|---|
| has-theme(theme-vertex-entity,theme-entity). |
| co-occurs(theme-vertex-entity,theme-vertex-entity). |
| copy-propagation-description(theme-vertex-entity,theme-vertex-entity). |
| definition-use-description-chain(theme-vertex-entity,theme-vertex-entity). |
| thematic-context(theme-vertex-entity,context-entity). |
| theme(theme-code-entity,category-description-entity). |

Table 7.2: Thematic Interconnection Graph Description

| |
|---|
| segment-uses-data-dictionary-region(segment-entity,region-area-entity). |
| segment-describes-part-of-module(segment-entity,module-entity). |
| segment-describes-part-of-system(segment-entity,system-entity). |
| module-belongs-to-system(module-entity,system-entity). |
| system-supplies-data-to(system-entity,system-entity). |
| system-data-received-from(system-entity,system-entity). |
| module-type(module-entity,module-type-entity). |
| module-test-required(module-entity,module-test-entity). |
| system-test-required(module-entity,system-test-entity). |
| module-data-file(module-entity,data-file-entity). |
| module-uses-job-control-language(module-entity,jcl-entity). |
| module-map-base(module-entity,map-base-entity). |
| associated-system(system-entity,system-entity). |

Table 7.3: Source Attributes Graph Description

| |
|---|
| document-potential-impact-dependency(segment-entity,segment-entity, percentage-chance-of-propagation). |
| module-potential-impact-dependency(segment-entity,segment-entity, percentage-chance-of-propagation). |
| module-expert-judgement(judgement-number,module-entity,release-number,module-entity, quantitative,qualitative). |
| document-expert-judgement(judgement-number,module-entity,release-number,module-entity, quantitative,qualitative). |
| judgement-reason(judgement-number,judgement-reason,judgement-assumption). |
| judgement-person(judgement-number,staff-name,date). |
| staff-responsible-for-system(system-entity,staff-name). |
| staff-who-have-maintained(system-entity,staff-name,system-knowledge). |

Table 7.4: Weighted Interconnection Graph Description

## 7.4 RPG Construction

There are four requirements for the graph data base construction process which the prototype implementation of MAGENTA must support:

1. recording the $HIG(\mathcal{V},\mathcal{E})$;
2. recording the $TIG(\mathcal{V},\mathcal{E})$;
3. recording the $SAG(\mathcal{V},\mathcal{E})$; and
4. recording the $WIG(\mathcal{V},\mathcal{E})$.

In figure 7.9 a structure chart shows the design of the prototype implementation of MAGENTA [3]. The labelled boxes represent modules and the edges labelled with capital letters represent parameters. The functional components are :

**Control Module.** [4] This invocates all other modules. There is no user interface within this prototype therefore the control module deals with output to the maintainer.

1. $RPG(\mathcal{V},\mathcal{E})$ **Loader.** This loads into the construction tool an existing $RPG(\mathcal{V},\mathcal{E})$ model. The parameter A denotes that the $RPG(\mathcal{V},\mathcal{E})$ is passed as an output parameter to the control module.

2. **Command Interpreter.** This allows the maintainer to select particular construction facilities. The parameter B denotes that an output parameter containing the graph operation required is returned to the control module.

3. **Structure Crystalliser.** This implements the graph crystallisation operation described in Chapter 5 to construct the $HIG(\mathcal{V},\mathcal{E})$. The parameter C denotes that the $RPG(\mathcal{V},\mathcal{E})$ is passed as input parameters and the Delta $CIG(\mathcal{V},\mathcal{E})$ is returned as an output parameter to the control module.

4. **Theme Crystalliser.** This implements the graph crystallisation operation described in Chapter 5 to construct the $TIG(\mathcal{V},\mathcal{E})$. The parameter D denotes that the $RPG(\mathcal{V},\mathcal{E})$ is passed as

[3]This figure relates to the final version of the prototype after iterative refinement

[4]The functional components written in boldface are not entries in the bibliography.

input parameters and the transformed $RPG(\mathcal{V},\mathcal{E})$ is returned as an output parameter to the control module.

5. **Annotator.** This implements the graph annotation operation described in Chapter 5 to construct the $SAG(\mathcal{V},\mathcal{E})$. The parameter E denotes that the $RPG(\mathcal{V},\mathcal{E})$ is passed as input parameters and the transformed $RPG(\mathcal{V},\mathcal{E})$ is returned as an output parameter to the control module.

6. **Parameteriser.** This implements the graph edge parameterisation operation described in Chapter 5 to construct the $WIG(\mathcal{V},\mathcal{E})$. The parameter F denotes that the $RPG(\mathcal{V},\mathcal{E})$ is passed as input parameters and the transformed $RPG(\mathcal{V},\mathcal{E})$ is returned as an output parameter to the control module.

7. **$RPG(\mathcal{V},\mathcal{E})$ Saver.** This stores the graph for subsequent analysis the $RPG(\mathcal{V},\mathcal{E})$ Analyser. The parameter G denotes that the $RPG(\mathcal{V},\mathcal{E})$ is passed as input parameters.

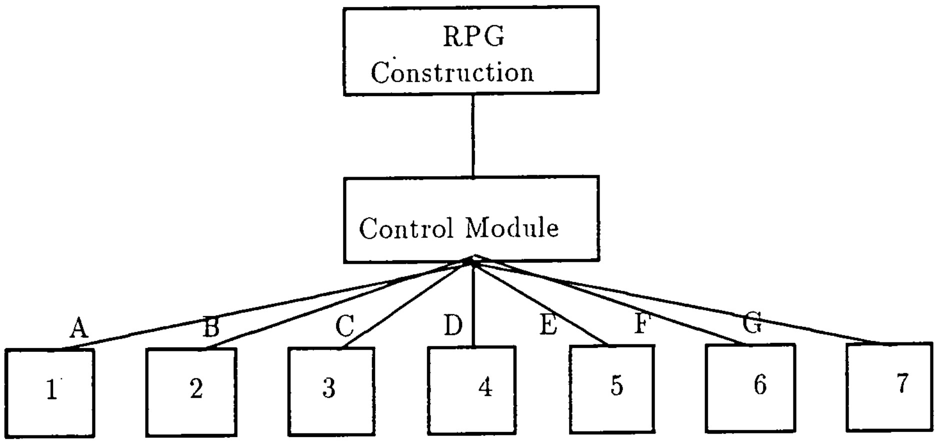


Figure 7.9: Structure of the Prototype Implementation($RPG(\mathcal{V},\mathcal{E})$ Construction)

In addition to constructing a graph a theme catalogue must also be constructed to hold a record of all the theme codes and themes associated with a particular $RPG(\mathcal{V},\mathcal{E})$. The theme catalogue is simply a text file of Prolog facts, which are entered with a text editor as they are detected in documentation. Each Prolog fact in the catalogue records a theme. For example :

```
theme(th52,describes-processing-of-report-lines-files).
```

indicates that theme code "th52" represents the theme "processing of report lines files". The following figure 7.10 shows a fragment of a theme catalogue.

```
theme(th50,describes-processing-of-recycled-reconciliation-movements).
theme(th51,describes-processing-of-rcn-movements-drip-ffed-file).
theme(th52,describes-processing-of-report-lines-files).
theme(th53,describes-processing-of-audit-tapes).
```

Figure 7.10: An Example of Themes Recorded in a Prolog

## 7.5 RPG Analysis

In order to derive impact information such as which other documents are affected by a change to a particular document segment or bag of segments, the graph slice operations described in Chapter 6 must be applied to the model. The requirements (informal) of the $RPG(\mathcal{V},\mathcal{E})$ analyser tool are the following :

1. **Tool overview and summary**

   The prototype implementation of MAGENTA automates the graph storage and analysis.

2. **Operating environment**

   The prototype implementation of MAGENTA operates in a Unix [5]operating system environment and is written in the programming language Edinburgh Prolog.

3. **User displays and report formats**

   The six techniques for analysing an $RPG(\mathcal{V},\mathcal{E})$ described in Chapter 6 each produce a subgraph of the $RPG(\mathcal{V},\mathcal{E})$ called a **Change Implication Graph** (CIG). The prototype implementation of MAGENTA produces change implications reports to describe each of the Change Implication Graphs. The **Change Implication Reports** (CIR) produced simply identify all the graph vertices and edges present in the extracted $CIG(\mathcal{V},\mathcal{E})$ .

4. **Modes of operation**

   There are two modes of operation for the $RPG(\mathcal{V},\mathcal{E})$ analyser tool :

[5]Unix is registered trademark of AT&T

(a) Interactive, that is, subgraphs of the $RPG(\mathcal{V}, \mathcal{E})$ can be produced by the maintainer by constructing queries in Prolog. For example if a change implication report indicates that a particular segment entity or module is impacted, then it is possible to go through the graph dictionary step by step to examine the graph.

(b) Selective, that is, the $RPG(\mathcal{V}, \mathcal{E})$ can be sliced on particular themes to produce change implication graphs using the five analysis techniques developed in Chapter 6.

5. **User command summary**

Each command is the name of the graph slice operation and the argument represents the change proposal number. Each change proposal number represents a bag of themes extracted from a change proposal and entered into the data base. The argument P enclosed in parentheses contains the change proposal number.

(a) view-change-proposal(P).

(b) view-thematic-graph-slice(P).

(c) view-complex-thematic-graph-slice(P).

(d) view-weighted-graph-slice(P).

(e) view-annotated-graph-slice(P).

(f) view-augmented-graph-slice(P).

(g) view-all-slices(P).

Alternatively Prolog's query interface can be used as a graph manipulation language, that is the interactive mode of operation.

6. **Command syntax and system options**

Each command has the name for the query and one or more arguments enclosed in round brackets followed by a full stop.

7. **Tool usage**

The tool is operated by entering Prolog on the Unix command line. A question mark followed by a minus sign will then appear "?-". This is a prompt indicating either a graph slice command or graph dictionary query is required.

The hierarchical structure of the prototype implementation of the method described in Chapters 4, 5, and 6 is shown in figure 7.11 [6]. The labelled boxes represent modules and the edges labelled with capital letters represent parameters. The functional components are the following :

**Control Module.** This is responsible for invocating all other modules.

8. **$RPG(\mathcal{V},\mathcal{E})$ Loader.** This loads in a particular $RPG(\mathcal{V},\mathcal{E})$ model. The parameter H denotes that the $RPG(\mathcal{V},\mathcal{E})$ is an output parameter from module 8 to the control module.

9. **Command Interpreter.** This allows the maintainer to select particular $RPG(\mathcal{V},\mathcal{E})$ transformations. The parameter I denotes that the transformation selected by the maintainer is an output parameter to the control module.

10. **Alpha Graph Slicer.** This extracts an Alpha $CIG(\mathcal{V},\mathcal{E})$ from the $RPG(\mathcal{V},\mathcal{E})$, The parameter J denotes that the $RPG(\mathcal{V},\mathcal{E})$ and Alpha criterion are passed as input parameters and the Alpha $CIG(\mathcal{V},\mathcal{E})$ is returned as an output parameter to the control module.

11. **Beta Graph Slicer.** This extracts a Beta $CIG(\mathcal{V},\mathcal{E})$ from the $RPG(\mathcal{V},\mathcal{E})$, The parameter K denotes that the $RPG(\mathcal{V},\mathcal{E})$ and Beta criterion are passed as input parameters and the Beta $CIG(\mathcal{V},\mathcal{E})$ is returned as an output parameter to the control module.

12. **Gamma Graph Slicer.** This extracts a Gamma $CIG(\mathcal{V},\mathcal{E})$ from the $RPG(\mathcal{V},\mathcal{E})$, The parameter L denotes that the $RPG(\mathcal{V},\mathcal{E})$ and Gamma criterion are passed as input parameters and the Gamma $CIG(\mathcal{V},\mathcal{E})$ is returned as an output parameter to the control module.

13. **Delta Graph Slicer.** This extracts a Delta $CIG(\mathcal{V},\mathcal{E})$ from the $RPG(\mathcal{V},\mathcal{E})$, The parameter M denotes that the $RPG(\mathcal{V},\mathcal{E})$ and Delta criterion are passed as input parameters and the Delta $CIG(\mathcal{V},\mathcal{E})$ is returned as an output parameter to the control module.

14. **Epsilon Graph Slicer.** This extracts an Epsilon $CIG(\mathcal{V},\mathcal{E})$ from the $RPG(\mathcal{V},\mathcal{E})$, The parameter N denotes that the $RPG(\mathcal{V},\mathcal{E})$ and Epsilon criterion are passed as input parameters and the Epsilon $CIG(\mathcal{V},\mathcal{E})$ is returned as an output parameter to the control module.

15. **Interactive Graph Slicer.** This allows the maintainer to construct his or her own queries as an alternative to executing the 5 graph slicing algorithms above. The parameter O denotes that the $RPG(\mathcal{V},\mathcal{E})$ and a maintainer defined criterion are passed as input parameters and the maintainer defined $CIG(\mathcal{V},\mathcal{E})$ is returned as an output parameter to the control module.

[6]This figure relates to the final version of the prototype after iterative refinement

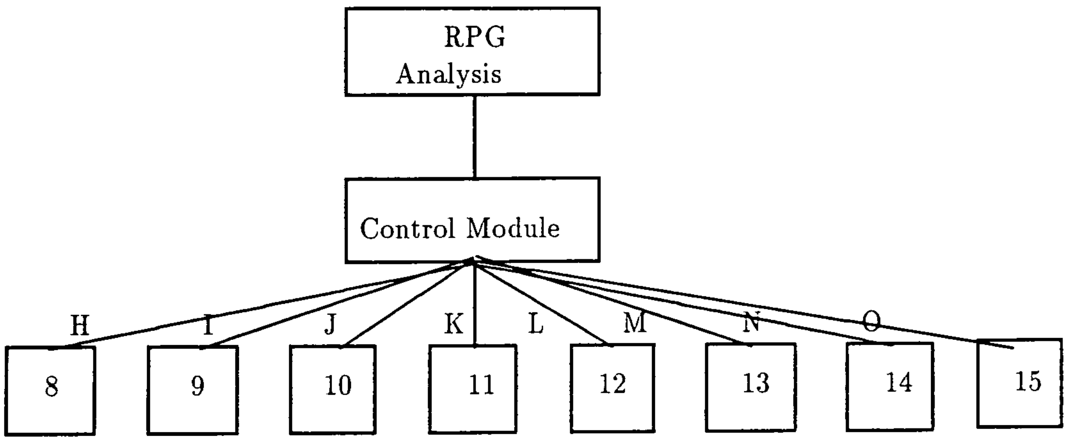


Figure 7.11: Structure of the Prototype Implementation($RPG(\mathcal{V},\mathcal{E})$ Analysis)

To analyse the impact of a **change proposal** a bag of themes must be created and recorded. Each bag of themes is associated with a change proposal by using the change proposal number, project number and the release number. Examples of other change proposal attributes that can be recorded are division, department, originators-name, date etc. A Change Proposal Description (CPD) is recorded using the graph description language and is loaded into the $RPG(\mathcal{V},\mathcal{E})$ analyser. An example of this is given in table 7.5.

| |
|---|
| proposal-identification(proposal-number,release-number, project-number). |
| source-of-change(proposal-number,division,department,originators-name,date). |
| proposal(proposal-number,theme). |

Table 7.5: Change Proposal Description

For example if a description of a system was stored in the file "rpg-system-name" and the change proposals were stored in the file "proposals" then they would be loaded into the analyser as in figure 7.12.

```
:prolog
| ?- [rpg-system-name].
| ?- [proposals].
| ?- [rpg-graph-translib].
| ?-
```

Figure 7.12: The $RPG(\mathcal{V},\mathcal{E})$ Analyser Invocation Interface

The "?-" is the Prolog prompt to select a graph transformation from the analysis tool. These transformations are stored in the file "rpg-graph-translib". Consider the example in figure 7.13 The command "view-change-proposal(P)" would display the details of the proposal whose number

```
| ?- view-change-proposal(P).
.
.
.
| ?- view-complex-thematic-slice(P).
.
.
.
```

Figure 7.13: An Example of the $RPG(\mathcal{V}, \mathcal{E})$ Analyser User Interface

is specified with the argument "P". The command "view-complex-thematic-slice(P)" would extract the subgraph according to this particular slice criterion and the themes associated with the proposal whose number is specified with the argument "P". All of the graph operations are specified in this way.

The output from the prototype implementation of MAGENTA is called a Change Implication Graph and is described by Change Implication Reports. The change implication reports are simply lists of vertices and edges remaining in the $CIG(\mathcal{V}, \mathcal{E})$ after applying particular graph operations. There are five types of change implication reports produced, one for each of the $CIG(\mathcal{V}, \mathcal{E})$ s defined in Chapter 6. The CIR consists of a header describing the graph manipulation applied to the specified $RPG(\mathcal{V}, \mathcal{E})$ model followed by the change details. The CIR consists of documentation and source code components impacted and is annotated with the probabilities of ripple effects, if these have been recorded. Actual examples of these reports are discussed in the next Chapter in which the method is applied.

## 7.6 Conclusions

A prototype of MAGENTA has been implemented using the language Prolog for both the slicing techniques and the graph description language. Using Prolog as a graph description language means that the graph description is machine readable and can therefore be converted into an alternative form for use by another type of tool. For example another tool may require the use of the release information. In order to access this information little work would have to be done in integrating the two tools. It is possible that both tools may use the Prolog facts as the internal structure for information. In this instance the tools would be well integrated with each other with respect to inter-operability, that is they would have a common view of the data. This inter-operability aspect is beyond the scope of this work, however it is an important consideration if the tool is to be incorporated into an integrated project support environment. The use of Prolog as a graph description language makes entry of the graph model easy since a text editor can be used. The use of Prolog facts as a representation requires one line of a text file to store each graph edge, start vertex and stop vertex. Using Prolog's inference engine makes retrieval of all or part of the graph model efficient, in terms of the amount of source code which must be written to achieve this.

The prototype implementation of MAGENTA is not integrated with a static analysis tool such as an inter-module code analyser. This could be a useful feature when, for example, a dependency is discovered between two source code modules in analysing the $RPG(\mathcal{V}, \mathcal{E})$. Then the actual dependency in the source code could be examined. It would also be useful to have a master index of all the components which comprise a software system. Such a master index could include the components which are live, also components which not contained in a live system but are potentially re-usable and finally the components which are being developed or maintained. Thus any source code entities impacted by a change proposal could quickly be located. The current prototype implementation does not provide information for the maintainer to interpret transformed $RPG(\mathcal{V}, \mathcal{E})$s. This would be a useful extension to the prototype implementation as it would provide guidance to the analysis of a large $CIG(\mathcal{V}, \mathcal{E})$.

It is argued that the user interface is an important part of a graph construction and analysis tool which will be used by information systems professionals. A poorly designed interface will increase the chance of errors and can significantly increase the time taken by a user to complete a graph construction or transformation task. There are many factors to consider when designing an interface for a tool. However it is thought that the treatment of this subject is beyond the scope of this work. The evaluation of the $RPG(\mathcal{V}, \mathcal{E})$ will not be affected by the lack of a user interface.

## 7.7 Summary

This Chapter described the prototype implementation of software tools to support the model and model analysis. It has also introduced the graph description language.

# Chapter 8

# Application of the Method

## 8.1 Introduction

In this chapter the concepts and ideas developed in Chapters 4, 5 and 6 are demonstrated with three prototyping experiments. The prototyping experiments use two examples of documentation structure and a major case study. Example 1 and 2 are small examples to demonstrate what can be achieved with the model and model analysis techniques. The third example, the application of the model to a real documentation system, is designed to determine the practical utility and feasibility of the model and model analysis techniques. Examples 1 and 2 and the case study collectively address the issues raised in the criteria for success in Chapter 1.

## 8.2 Method of Application

The method of application of the $RPG(\mathcal{V}, \mathcal{E})$ and its associated analysis techniques is a case study. A case study is a worked example of the application of technique. The reasons for using a case study are to provide a practical evaluation of the model and analysis techniques and also to provide pointers on how the analysis techniques can be used. The problem with using a case study and in particular one case study is that it is difficult to interpret the worth of the technique and in particular its comparison with other competing techniques.

Before the case study is presented the theoretical application of the technique is demonstrated using two examples of documentation structure. The examples presented in this Chapter are examples of the $RPG(\mathcal{V},\mathcal{E})$ at different stages in the development. The first two examples of application are both small and theoretical, so that the $RPG(\mathcal{V},\mathcal{E})$ can be visualised and the results of the analysis can be easily understood. The third example, namely the case study is considerably larger than the examples. This is used to provide insights into the practical utility of the $RPG(\mathcal{V},\mathcal{E})$ in industrial/commercial size systems. The first example presented is the example containing the least amount of detail. The second example presented includes all of the features of the model and all of the analysis techniques. The third example applies the $RPG(\mathcal{V},\mathcal{E})$ and all of the analysis techniques to a case study.

The examples and case study presented in this Chapter include a description of the scenario, a description of the techniques which were actually applied and a description of the results achieved. The examples presented in this Chapter relate to the characteristics of documentation described in Chapter 3. For example, each of the examples and the case study include the following features :

1. one or more documents;
2. hierarchical structure;
3. logically connected documents through the mutual information they share, for example two documents could both describe the processing of the same data file;
4. inter and intra-document dependencies;
5. the documents contain text;
6. the documents can be factored into segments;
7. some documents may be missing or incomplete;
8. some sections of documents may be missing or incomplete;
9. the segment entities contain themes.

The following steps form the thematic slicing method :

$RPG(\mathcal{V},\mathcal{E})$ Construction Steps:

**Step 1:** $HIG(\mathcal{V},\mathcal{E})$ Crystallisation

**Step 2:** $TIG(\mathcal{V},\mathcal{E})$ Crystallisation

**Step 3:** $WIG(\mathcal{V},\mathcal{E})$ Parameterisation

**Step 4:** $SAG(\mathcal{V},\mathcal{E})$ Annotation

$RPG(\mathcal{V},\mathcal{E})$ Analysis Steps:

**Step 5:** $\alpha$ Slicing

**Step 6:** $\beta$ Slicing

**Step 7:** $\gamma$ Slicing

**Step 8:** $\delta$ Slicing

**Step 9:** $\epsilon$ Slicing

Examples 1 and 2 are theoretical examples of documentation. Example 1 demonstrates the alpha slice on a very simple $RPG(\mathcal{V},\mathcal{E})$ and example 2 demonstrates all of the slicing techniques on a more complicated $RPG(\mathcal{V},\mathcal{E})$. However the major case study demonstrates all of the construction and analysis steps as the data for the model was extracted from a real documentation system.

## 8.3 Example 1

### 8.3.1 Scenario Description

The purpose of this example is to show a very simple application of the $RPG(\mathcal{V},\mathcal{E})$ to model interconnectivity within a document hierarchy. This section describes an example rather than a real documentation system. This is because the complexity of a real documentation system is too great to clearly demonstrate the analysis of the model.

This example of a documentation system contains six levels of hierarchy which are the following document types Libraries, Volumes, Chapters, Sections, Sub-sections and Segments.

The library is decomposed into two volumes $V_1$ and $V_2$. In order to make the example concise, only $V_2$ is decomposed. $V_2$ is decomposed into 2 chapters, 5 different sections, 4 subsections and 6 segments. Segments are factored into 9 theme vertices and 7 themes. In this example the two entities $C_1$ and $C_2$ represent two chapters describing two different modules. Co-occurs dependencies are included this example to show the description of the sharing of data between the two modules.

The graph can be described set theoretically in the following way :

*Composite Entities* $= \{ L_1, V_1, V_2, C_1, C_2, S_1, S_2, S_3, S_4, S_5, SB_1, SB_2, SB_3, SB_4 \}$

*Segment Entities* $= \{ G_1, G_2, G_3, G_4, G_5, G_6 \}$

*Thematic Entities* $= \{ T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9 \}$

*Themes* $= \{ t_1, t_4, t_5, t_8, t_{18}, t_{20}, t_{32} \}$

*Hierarchical Interconnection Graph:*

*Consists of Dependencies* =
$\{(L_1,V_1),(L_1,V_2),(V_2,C_2),(C_1,S_1),(C_1,S_2),(C_1,S_3),(C_2,S_4),(C_2,S_5),(S_2,SB_1),$
$(S_5,SB_3),(S_5,SB_4),(SB_2,G_1),(SB_2,G_2),(SB_2,G_3),(SB_4,G_4),(SB_4,G_5),(SB_4,G_6)\}$

*Thematic Interconnection Graph:*

*Has Theme Vertex Dependencies* =
$\{(G_1,T_1),(G_2,T_2),(G_2,T_3),(G_3,T_4),(G_3,T_5),(G_4,T_6),(G_5,T_7),(G_6,T_8),(G_6,T_9)\}$

*Has Theme Dependencies* =
$\{(T_1,t_1),(T_2,t_1),(T_3,t_4),(T_4,t_{18}),(T_5,t_{20}),(T_6,t_4),(T_7,t_{32}),(T_8,t_5),(T_8,t_9)\}$

*Co-occurs Dependencies* =
$\{(T_1,T_2),(T_3,T_6)\}$

### 8.3.2 Slicing and Results

Given the $RPG(\mathcal{V},\mathcal{E})$ in figure 8.1 and the bag of themes $\mathcal{TB}$, where $\mathcal{TB} = \prec t_1, t_4, t_{20} \succ$ the following sets can be derived by applying the $\alpha$ slice over the $RPG(\mathcal{V},\mathcal{E})$.

1. *Thematic Components Impacted* = $\{T_1, T_3, T_6\}$;
2. *Segment Components Impacted* = $\{G_1, G_2, G_4\}$;
3. *Composite Components Impacted* =
   (a) *Composite Components Impacted'* = $\{SB_2, SB_4\}$;
   (b) *Composite Components Impacted'* = $\{SB_2, SB_4, S_2, S_5\}$;
   (c) *Composite Components Impacted'* = $\{SB_2, SB_4, S_2, S_5, C_1, C_2\}$;
   (d) *Composite Components Impacted'* = $\{SB_2, SB_4, S_2, S_5, C_1, C_2, V_2\}$;
   (e) *Composite Components Impacted'* = $\{SB_2, SB_4, S_2, S_5, C_1, C_2, V_2, L_1\}$;
4. *Documentation Entities Impacted* = $\{L_1, V_2, C_1, C_2, S_2, S_5, SB_2, SB_4, G_1, G_2, G_4\}$;

Although the change proposal could be directed towards the application documented in entity $C_1$, due to the thematic interconnectivity recorded in the $RPG(\mathcal{V},\mathcal{E})$ the ripple effect propagated to $C_2$ is detected. The figure 8.1 shows the example described by the scenario. The figure 8.26 shows the $\alpha$ slice of the $RPG(\mathcal{V},\mathcal{E})$ described by the scenario.

#### Step 5: $\alpha$ Slicing

Figure 8.2 represents the resultant $RPG(\mathcal{V},\mathcal{E})$ slice if the $\alpha$ slice is applied with the theme bag $\prec t_1, t_4, t_{20} \succ$. The slice shows the segment entities impacted and composite components impacted.

The $\mathcal{BV}$ =
$\prec L_1, V_2, C_2, C_1, S_2, S_5, SB_2, SB_4, G_1, G_2, G_3, G_4 \succ$

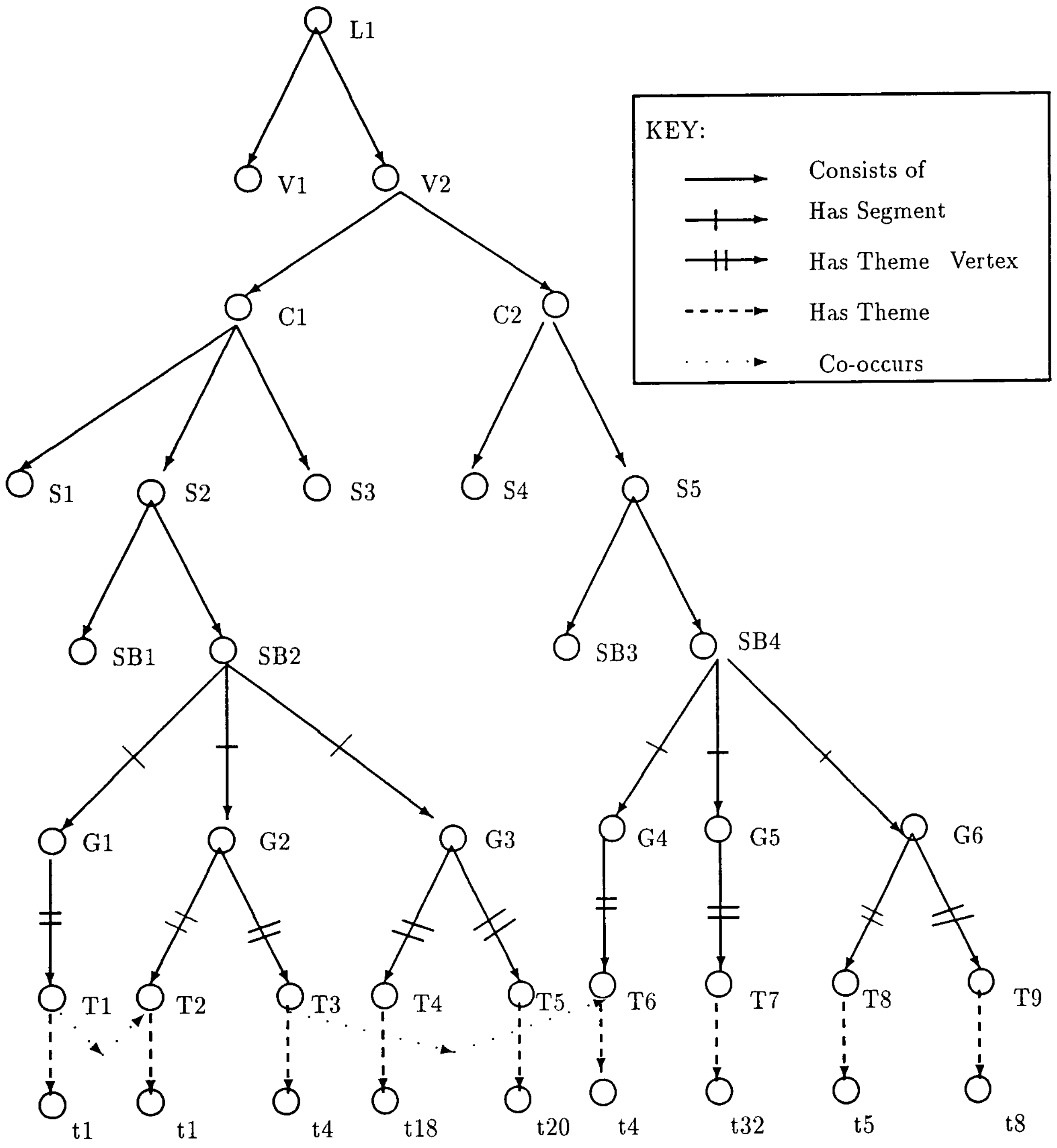


Figure 8.1: $RPG(\mathcal{V}, \mathcal{E})$ Example 1

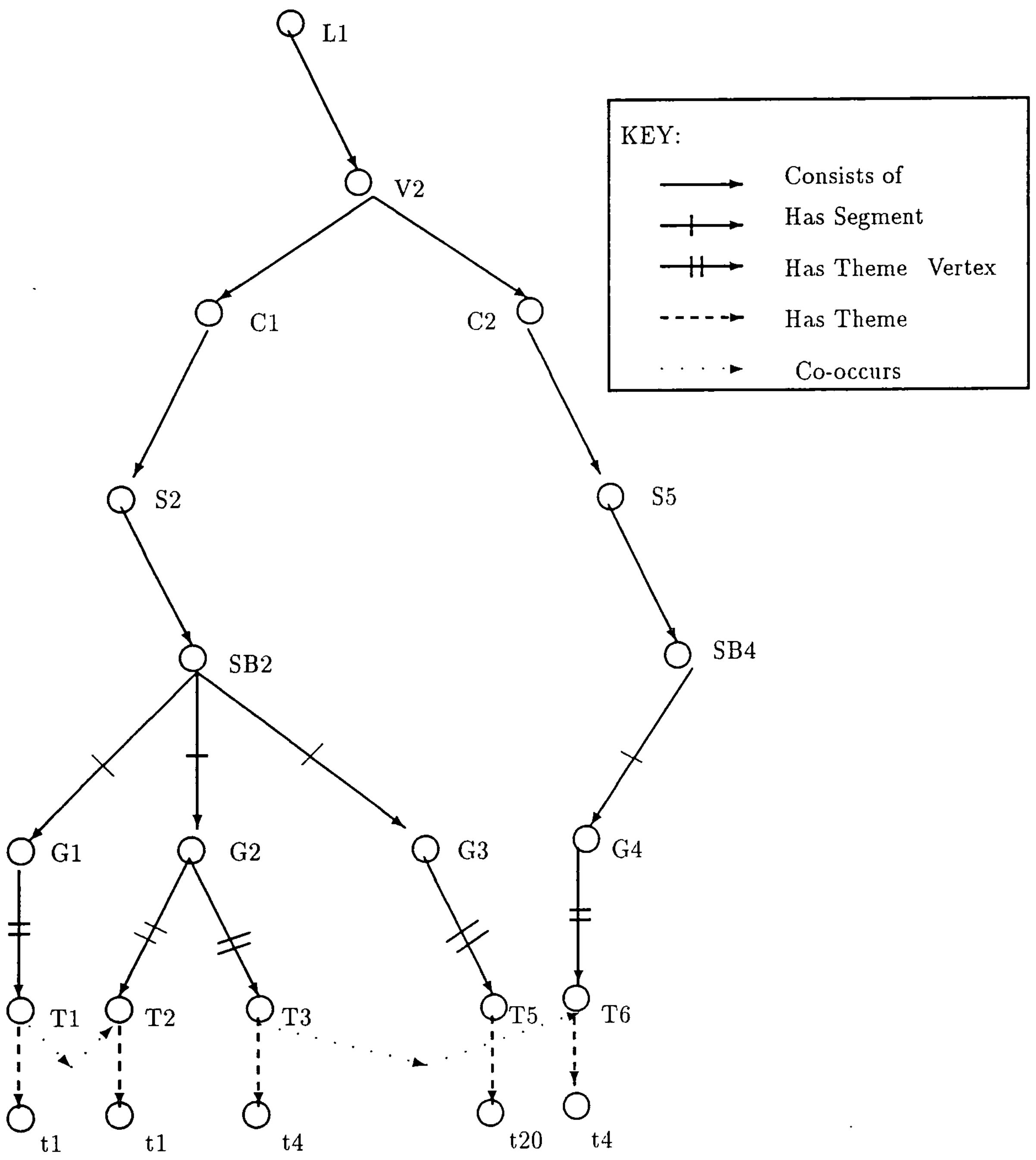


Figure 8.2: Example 1 $\alpha CIG(\mathcal{V}, \mathcal{E})$

The $\mathcal{BE}$ =
$\prec (L_1, V_2), (V_2, C_2), (V_2, C_1), (C_1, S_2), (C_2, S_5), (S_2, SB_2), (S_5, SB_4),$
$(SB_2, G_1), (SB_2, G_2), (SB_2, G_3), (SB_4, G_4), \succ$

### 8.3.3 Discussion

In this example the $\alpha$ slicing technique is demonstrated using a very simple example of a document hierarchy where each of segments modelled are given one or more themes. The co-occurs dependencies are used to describe the interconnectivity within the document hierarchy. In this particular example a change is made to $C_1$ (representing Chapter 1 of the document). As $C_2$ also contains themes which are common to $C_1$ it is possible that both Chapters $C_1$ and $C_2$ will be affected by the proposed change. The $\alpha$ slice provides information for a maintenance manager or maintainer to understand both the sharing of data used within an application and how it is described in documentation.

## 8.4 Example 2

### 8.4.1 Scenario Description

The purpose of this example is to show a very simple application of the complete $RPG(\mathcal{V}, \mathcal{E})$ to model interconnectivity within a document hierarchy. The objective is to show all of the features of the model and all of the thematic graph slicing techniques developed in this thesis. This particular example, like example 1, is an example rather than a real documentation system. This is because the complexity of a real documentation system is too great to clearly demonstrate the analysis of the model.

This example is supported with Appendix C. The appendix demonstrates the use of the RPGDL for describing the $RPG(\mathcal{V}, \mathcal{E})$ example 2, the associated theme catalogue, examples of change proposals and a consultation with the prototype implementation MAGENTA.

This example of a documentation system is an extension of the previous example and contains six levels of hierarchy namely Libraries, Volumes, Chapters, Sections, Sub-sections and Segments, as before. This example also contains the addition of the $WIG(\mathcal{V},\mathcal{E})$ and $SAG(\mathcal{V},\mathcal{E})$ to show the advantage of recording such information on the $RPG(\mathcal{V},\mathcal{E})$.

Since this second example is also theoretical it is unnecessary to describe the system described by the documentation, its features and functionality. However the links between the main documents and source code entities contained within this example is shown in figure 8.3.

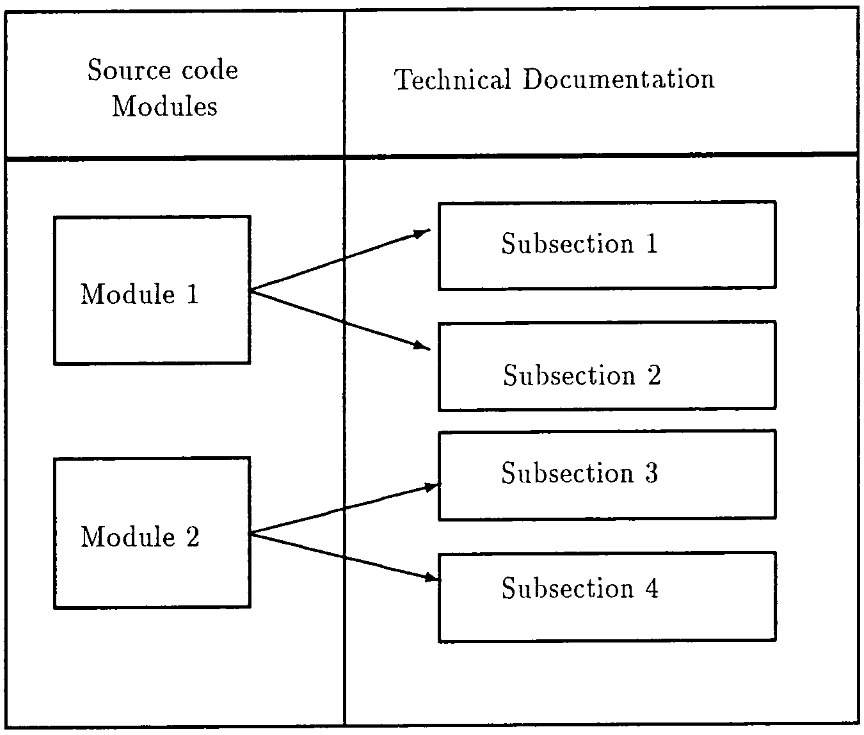


Figure 8.3: Example 2 Traceability between code and documentation

### 8.4.2 A Change Proposal

Although this example is theoretical, the extraction of themes from a change proposal is demonstrated. The 'Business Summary' and the 'Detailed Business Requirements' are the most important part of the change proposal as these are the only sections of the change proposal which are used for constructing a theme bag.

1. **Proposal Identification Information.**

   *Feasibility No: 1, 'Improve Data Placement' September 91*

2. **Business Summary.**

   *The proposal is to incorporate the Last Charges Date into personal cheque accounts an investment accounts.*

3. **Detailed Business Requirements.**

   *The system needs to establish the Last Charges date for an account record for personal cheque accounts an investment accounts*

4. **Service Levels.**

   *This proposal will help to ensure that existing service level agreements are met on a more regular basis.*

5. **Time scales.**

   *The project can be implemented with very little effort if the implementation coincides with other similar database changes.*

6. **Assumptions and Constraints.**

   *The last charges date is the only data item to be considered.*

### 8.4.3 Proposal Analysis

The proposal can be decomposed into the following CAs and COs :

**CAs:** incorporate, establish

**COs:** Last Charges Date, personal cheque accounts, investment accounts, account record

These CAs and COs can be related in the following way :

1. establish Last Charges Date.
2. account record.
3. Incorporate Last Charges Date onto personal cheques.
4. Incorporate Last Charges Date onto investment accounts.

The following table 8.1 represents a catalogue of themes. In this theme description the conceptual objects have a two descriptors. For example line one of the catalogue contains "a1" and "(account)". The first descriptor is the system orientated name of the conceptual object, whilst the second descriptor is the domain or business oriented descriptor.

| theme code | theme description |
|---|---|
| t1 | describes-processing-of-a1 (account record) |
| t2 | describes-processing-of-b1 |
| t3 | describes-processing-of-c1 |
| t4 | describes-processing-of-d1 (personal cheque accounts) |
| t5 | describes-processing-of-e1 |
| t31 | describes-processing-of-e2 |
| t32 | describes-processing-of-f2 (investment accounts) |

Table 8.1: Example 2 A Theme Catalogue

The CAs one in this case i.e., 'incorporate' is mapped onto the 'describes-processing' part of the theme and the COs are mapped onto the '-of-a1' part of the theme. By identifying the themes within the system a theme bag can be constructed.

$$\mathcal{TB} = \prec \text{t1,t4,t32} \succ$$

In this particular case there are no multiple changes to any themes in the proposal.

### 8.4.4 Slicing and Results

Figure 8.4 shows the example described by the scenario.

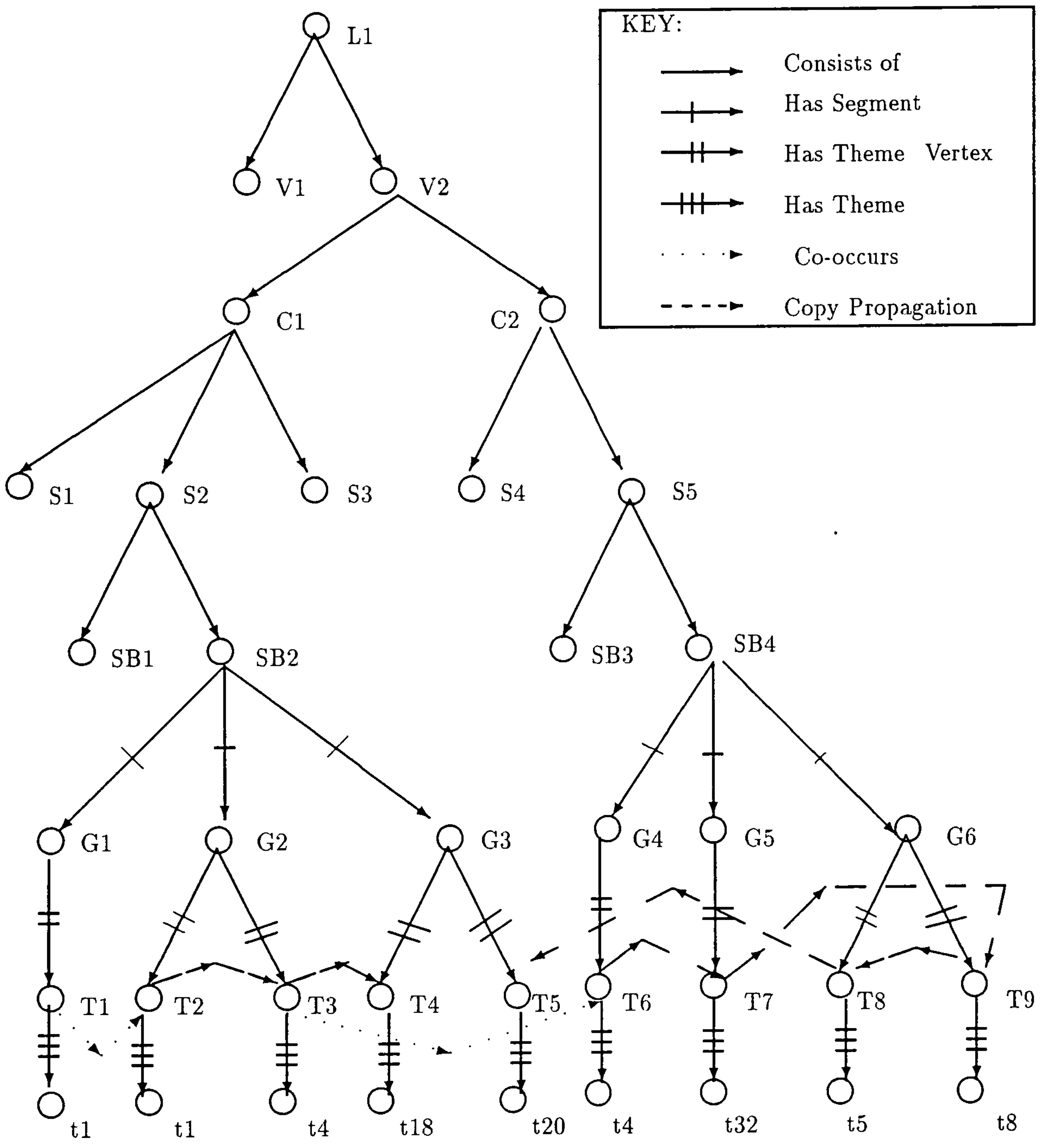


Figure 8.4: $RPG(\mathcal{V}, \mathcal{E})$ Example 2

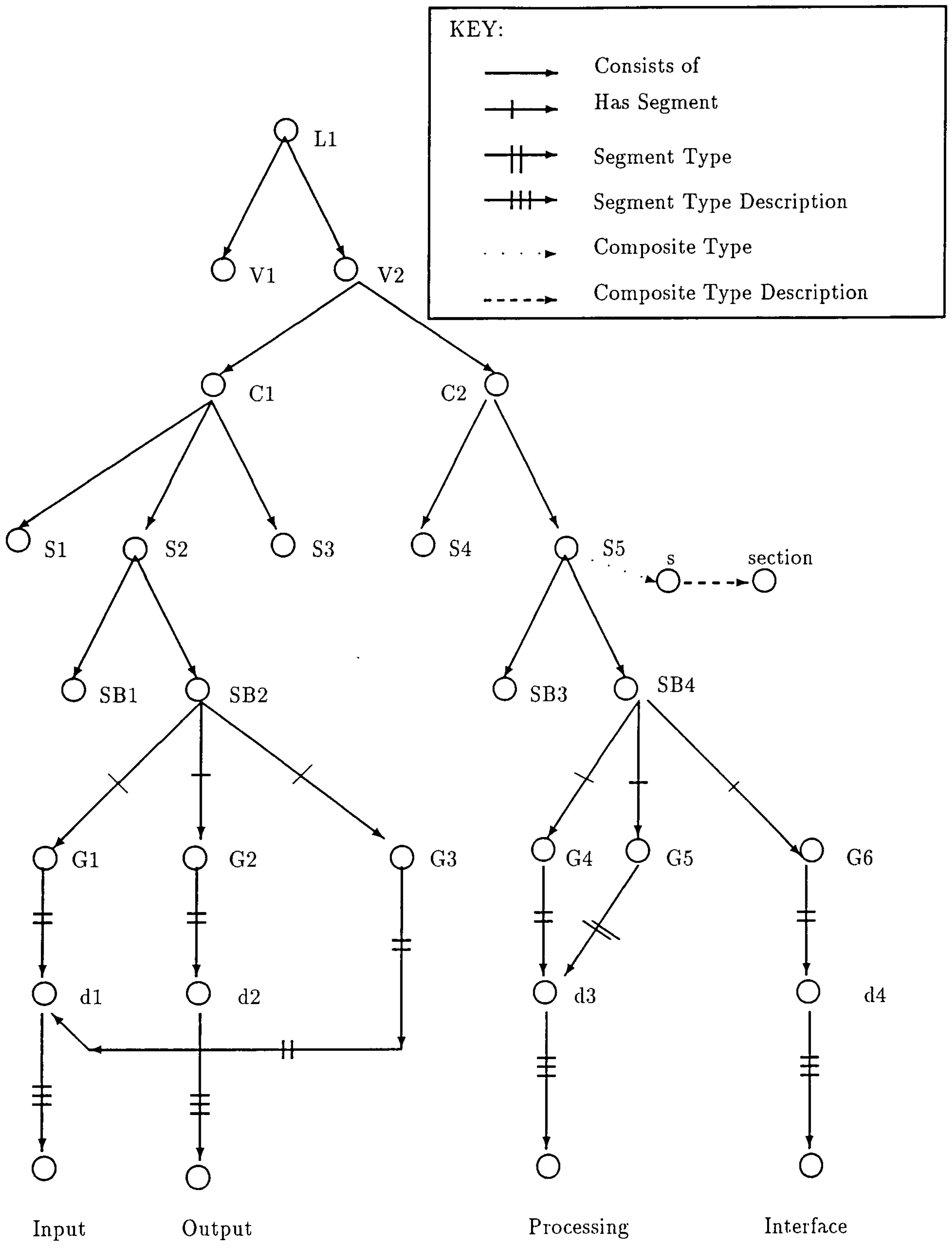


Figure 8.5: Example 2 with Composite and Segment Types

## Step 1 and 2: Crystallisation

The $HIG(\mathcal{V},\mathcal{E})$ and $TIG(\mathcal{V},\mathcal{E})$ Crystallisation techniques were not applied in example two as the example is not based on an actual real example of documentation. The figure 8.4 shows the $RPG(\mathcal{V},\mathcal{E})$ used as example 2. The figure 8.5 shows the example 2 with the composite and segment types.

## Step 3: $WIG(\mathcal{V},\mathcal{E})$ Parameterisation

In this example release information and expert judgements on interconnectivity are simulated to demonstrate in a concise way the usefulness of this information when slicing a model of documentation. Figure 8.6 shows an example of a $WIG(\mathcal{V},\mathcal{E})$ after direct edge parameterisation. Figure 8.7 shows an example of a $WIG(\mathcal{V},\mathcal{E})$ after indirect edge parameterisation.

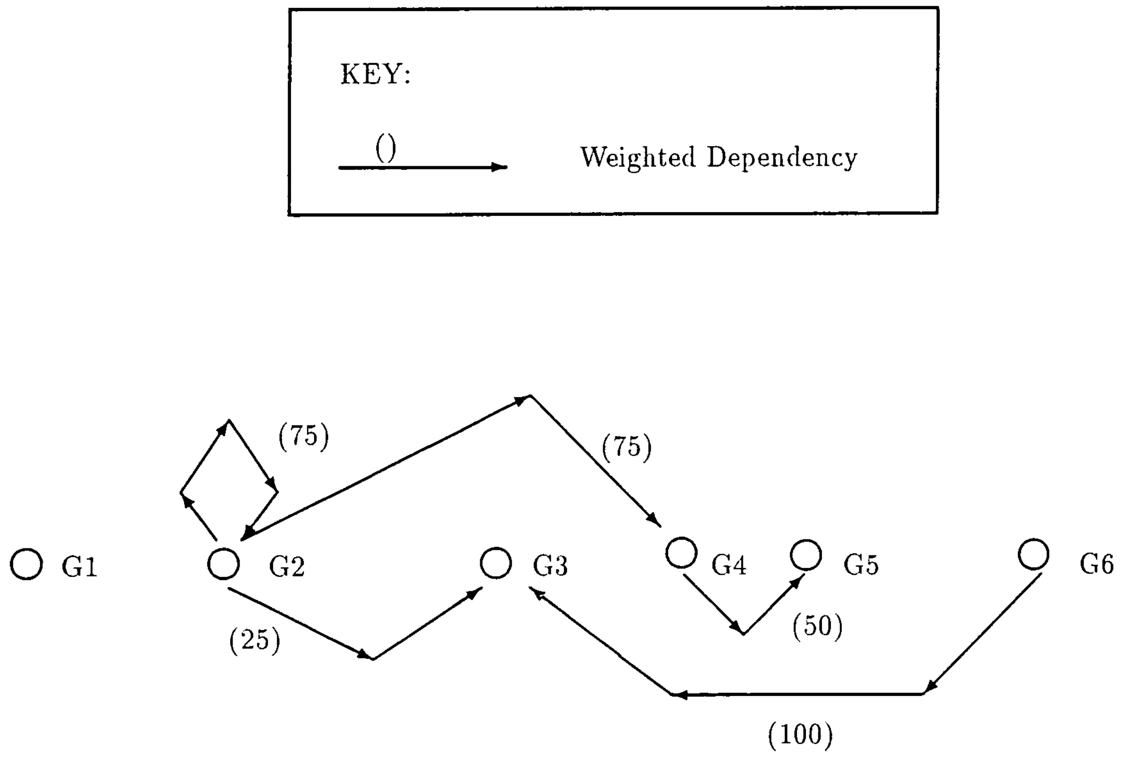


Figure 8.6: Example 2 containing $WIG(\mathcal{V},\mathcal{E})$

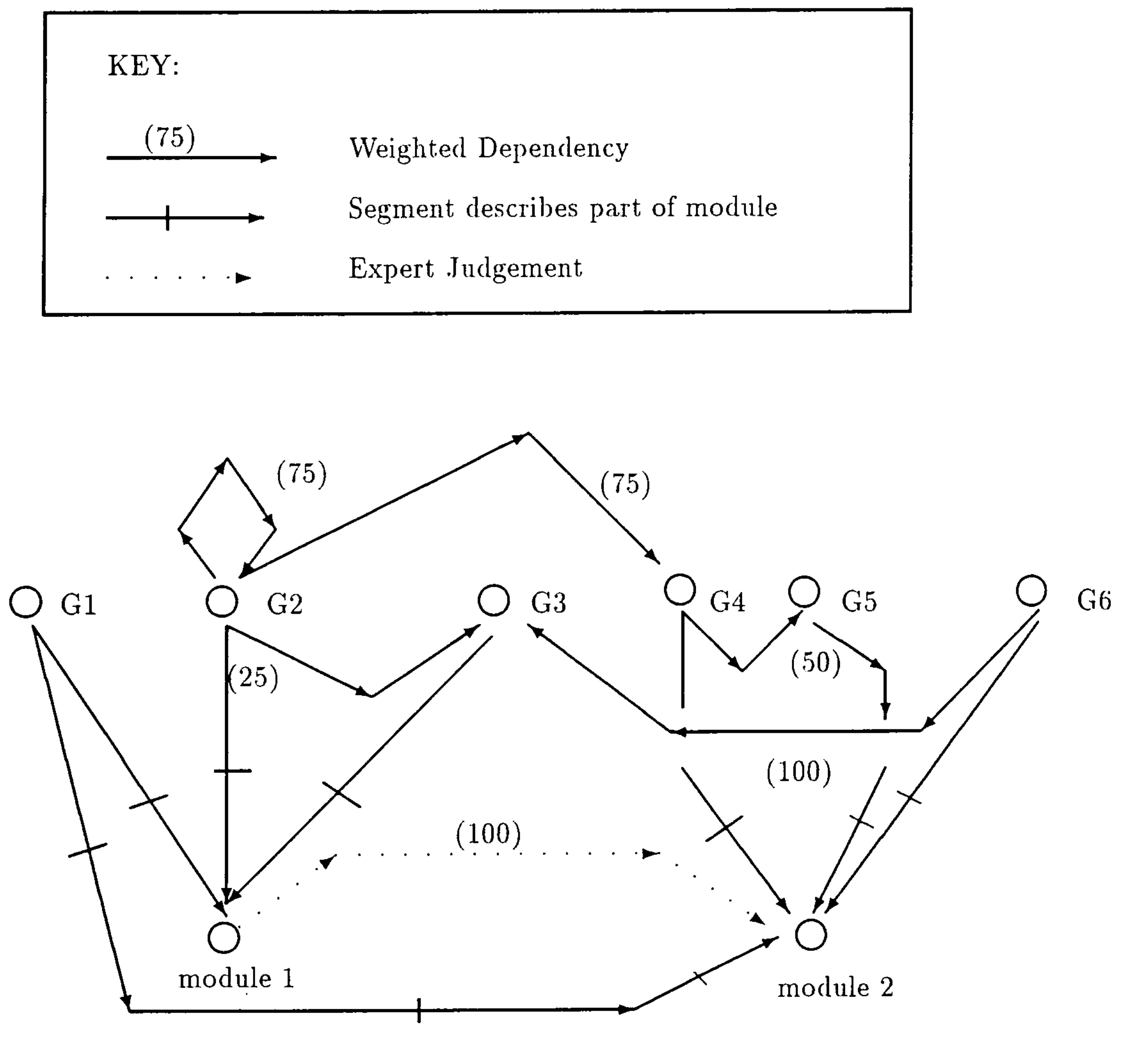


Figure 8.7: Example 2 containing $WIG(\mathcal{V}, \mathcal{E})$ based on expert judgement

### Step 4: $SAG(\mathcal{V},\mathcal{E})$ Annotation

Each of the segment entities have their content further characterised by this graph operation. See figure 8.8.

### Step 5: $\alpha$ Slicing

Figure 8.9 represents the resultant $RPG(\mathcal{V},\mathcal{E})$ slice if the $\alpha$ slice is applied to example 2 with the $\mathcal{TB} = \prec$ t1,t4,t32 $\succ$. The slice shows the segment entities impacted and composite components impacted.

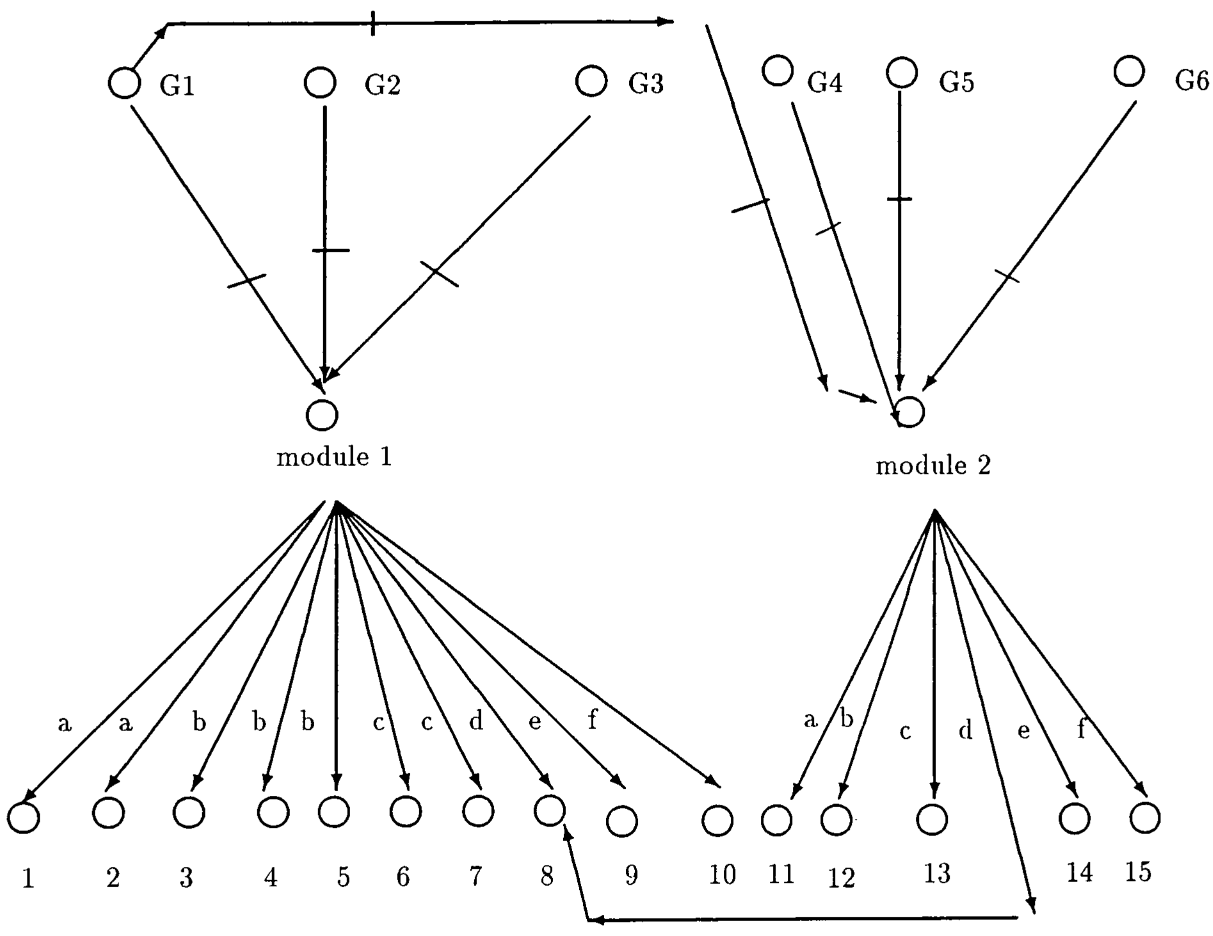
### Step 6: $\beta$ Slicing

Figure 8.10 represents the resultant $RPG(\mathcal{V},\mathcal{E})$ slice if the $\beta$ slice is applied to example 2 with the particular $\mathcal{TB} = \prec$ t1,t4,t32 $\succ$. The slice shows the segments impacted and composite components impacted based on tracing the copy propagations.

### Step 7: $\gamma$ Slicing

Figure 8.11 represents the resultant $RPG(\mathcal{V},\mathcal{E})$ slice if the $\gamma$ slice is applied to example 2 with the particular $\mathcal{TB} = \prec$ t1,t4,t32 $\succ$. The slice shows the segments impacted and the composite components impacted based on tracing the copy propagations. In addition the probabilities of ripple effects between segment entities are also impacted. These probabilities are based on previous release information. For this particular example the bags $\mathcal{BE}$ and $\mathcal{BV}$ are given to show their utility see figure 8.12.

If the $\mathcal{TB}$ is expanded to $\prec th_1, th_4, th_{32}, th_1, th_4, th_{32} \succ$ the actual impact of the change doubles and would be shown in the resultant bags $\mathcal{BV}, \mathcal{BE}$. However the $CIG(\mathcal{V},\mathcal{E})$ does not change. This demonstrates the importance of constructing impacted bags containing multiple impacts on the same entity or edge and shows the weakness of sets which only contain one occurrence of each entity.



KEY:

| SAG Entities | | | | SAG Dependencies | |
|---|---|---|---|---|---|
| 1 | COBOL | 9 | jcl 1 | a | module type |
| 2 | assembler | 10 | map 1 | b | module test file |
| 3 | test mod 1 | 11 | COBOL | c | system test required |
| 4 | test mod 2 | 12 | test mod 2 | d | module data file |
| 5 | test mod 4 | 13 | system test 1 | e | module job control language |
| 6 | system test 1 | 14 | jcl 2 | f | module map base |
| 7 | system test 16 | 15 | map 2 | | |
| 8 | batch file | | | | |

Figure 8.8: Example 2 containing $SAG(\mathcal{V}, \mathcal{E})$

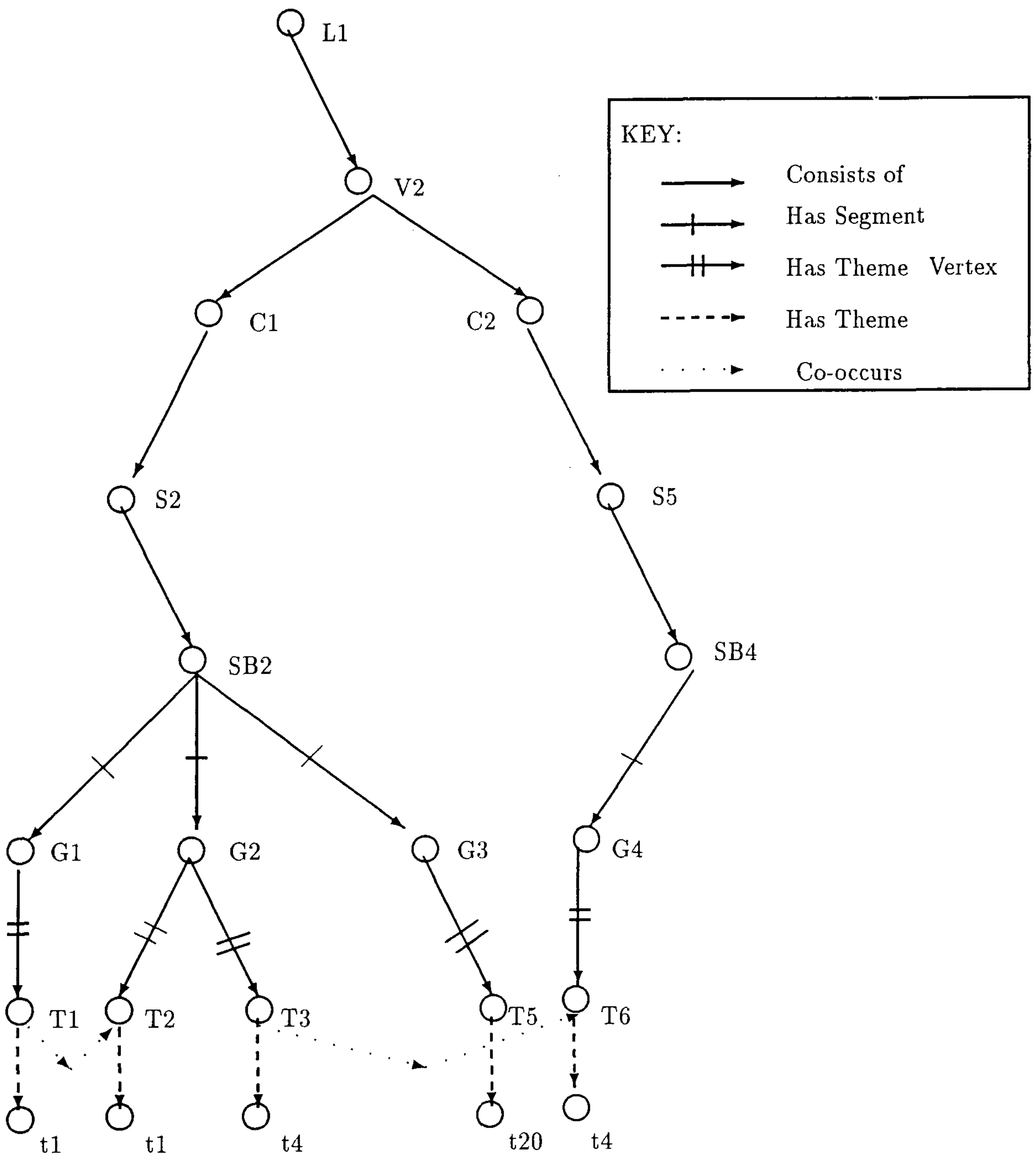


Figure 8.9: Example 1 $\alpha CIG(\mathcal{V}, \mathcal{E})$

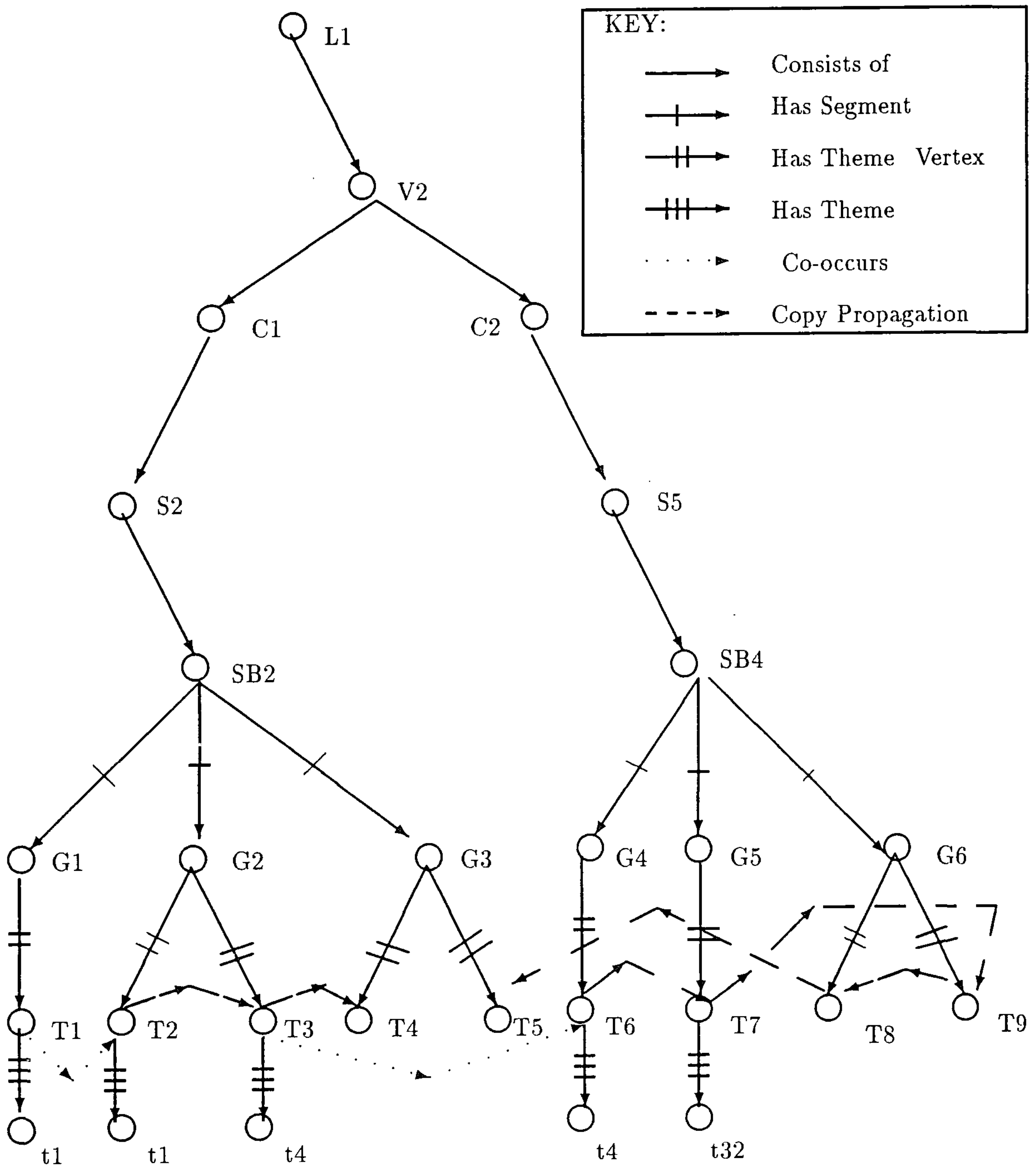


Figure 8.10: Example 2 $\beta CIG(\mathcal{V},\mathcal{E})$

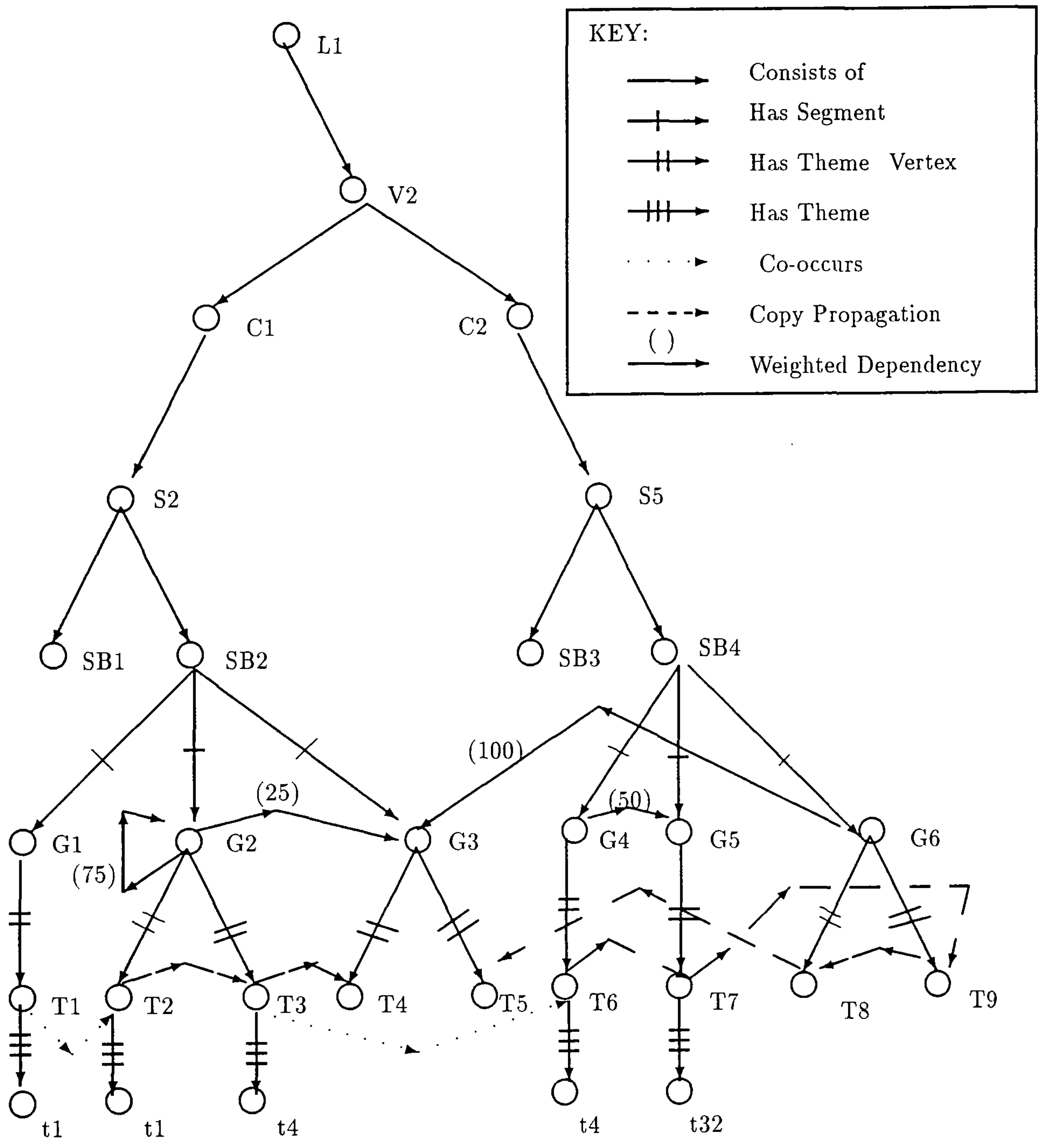


Figure 8.11: Example 2 $\gamma CIG(\mathcal{V}, \mathcal{E})$

$\mathcal{BE} =$

$\prec (t_1,T_1),(t_1,T_2),(t4,T_3),(t_4,T_6),(t32,T_7),(T_1,T_2),(T_2,T_3),(T_3,T_4),(T_2,T_3),$
$(T_3,T_4),(T_3,T_4),(T_3,T_6),(T_6,T_7),(T_7,T_9),(T_9,T_8),(T_8,T_5),(T_7,T_9),(T_9,T_8),$
$(T_8,T_5),(T_1,G_1),(T_2,G_2),(T_3,G_2),(T_6,G_4),(T_7,G_5),(T_3,G_2),(T_4,G_3),(T_4,G_3),$
$(T_7,G_5),(T_9,G_6),(T_8,G_6),(T_5,G_3),(T_9,G_6),(T_8,G_6),(T_5,G_3),(G_1,SB_2),(SB_2,S_2),$
$(S_2,C_1),(C_1,V_2),(V_2,L_1),(G_2,SB_2),(SB_2,S_2),(S_2,C_1),(C_1,V_2),(V_2,L_1),(G_2,SB_2),$
$(SB_2,S_2),(S_2,C_1),(C_1,V_2),(V_2,L_1),(G_3,SB_2),(SB_2,S_2),(S_2,C_1),(C_1,V_2),(V_2,L_1),$
$(G_5,SB_2),(SB_2,S_2),(S_2,C_1),(C_1,V_2),(V_2,L_1),(G_2,SB_2),(SB_2,S_2),(S_2,C_1),(C_1,V_2),$
$(V_2,L_1),(G_3,SB_2),(SB_2,S_2),(S_2,C_1),(C_1,V_2),(V_2,L_1),(G_3,SB_2),(SB_2,S_2),(S_2,C_1),$
$(C_1,V_2),(V_2,L_1),(G_5,SB_2),(SB_2,S_2),(S_2,C_1),(C_1,V_2),(V_2,L_1),(G_6,SB_4),(SB_4,S_5),$
$(S_5,C_2),(C_2,V_2),(V_2,L_1),(G_6,SB_4),(SB_4,S_5),(S_5,C_2),(C_2,V_2),(V_2,L_1),(G_3,SB_4),$
$(SB_4,S_5),(S_5,C_2),(C_2,V_2),(V_2,L_1),(G_6,SB_4),(SB_4,S_5),(S_5,C_2),(C_2,V_2),(V_2,L_1),$
$(G_6,SB_4),(SB_4,S_5),(S_5,C_2),(C_2,V_2),(V_2,L_1),(G_3,SB_4),(SB_4,S_5),(S_5,C_2),(C_2,V_2),$
$(V_2,L_1) \succ$

$\mathcal{BV} =$

$\prec t_1,t_1,t_4,t_4,T_1,T_2,T_3,T_6,T_1,T_2,T_3,T_6,T_7,T_3,T_4,T_4,T_7,T_9,T_8,T_5,T_9,T_8,T_5,$
$G_1,G_2,G_2,G_3,G_5,G_2,G_3,G_3,G_5,G_6,G_6,G_3,G_6,G_6,G_4,SB_2,S_2,C_1,V_2,L_1,SB_2,S_2,C_1,$
$V_2,L_1,SB_2,S_2,C_1,V_2,L_1,SB_4,S_5,C_2,V_2,L_1,SB_4,S_5,C_2,V_2,L_1,SB_2,S_2,C_1,V_2,L_1,SB_2,$
$S_2,C_1,V_2,L_1,SB_2,S_2,C_1,V_2,L_1,SB_2,S_2,C_1,V_2,L_1,SB_4,S_5,C_2,V_2,L_1,SB_4,S_5,C_2,$
$V_2,L_1,SB_2,S_2,C_1,V_2,L_1,SB_4,S_5,C_2,V_2,L_1,SB_4,S_5,C_2,V_2,L_1,SB_2,S_2,C_1,V_2,L_1 \succ$

Figure 8.12: Example 2 $\gamma$ bags containing multiple impacts

### Step 8: $\delta$ Slicing

Figure 8.13 represents the resultant $RPG(\mathcal{V},\mathcal{E})$ slice if the $\delta$ slice is applied to example 2 with the $\mathcal{TB} = \prec$ t1,t4,t32 $\succ$. The slice shows the segments impacted and the composite components impacted based on tracing the copy propagations. In addition the probabilities of ripple effects between segment entities are also impacted. These probabilities are based on expert judgement.

### Step 9: $\epsilon$ Slicing

Figure 8.14 represents the resultant $RPG(\mathcal{V},\mathcal{E})$ slice if the $\epsilon$ slice is applied to example 2 with the particular $\mathcal{TB} = \prec$ t1,t4,t32 $\succ$. The slice shows the segments impacted and the composite components impacted, based on tracing the copy propagations. In addition, the probabilities of ripple effects between segment entities are also impacted. These probabilities are based on expert judgement.

## 8.4.5 Discussion

In this example all of the analysis techniques have been demonstrated on a small example in order to demonstrate the usefulness of graph slicing. It has been shown how the themes within the change proposal formally traced to operational software by analysing documentation interconnectivity and content.

The analysis techniques have been shown to provide a method of understanding the documentation and also a method of manipulating the documentation without getting overwhelmed by the volume of information. The maintenance manager is able to focus on the impacted documents, the impacted source code constructs and the reasons why they are impacted.

The extraction of a subgraph representing the impact of a change allows the maintenance manager or maintainer to investigate the edges on the subgraph representing the transitive closure. This helps the maintenance manager understand the reasons why a particular component is impacted.

The use of bag theory for storing multiple impacts of the same entity has also been demonstrated.

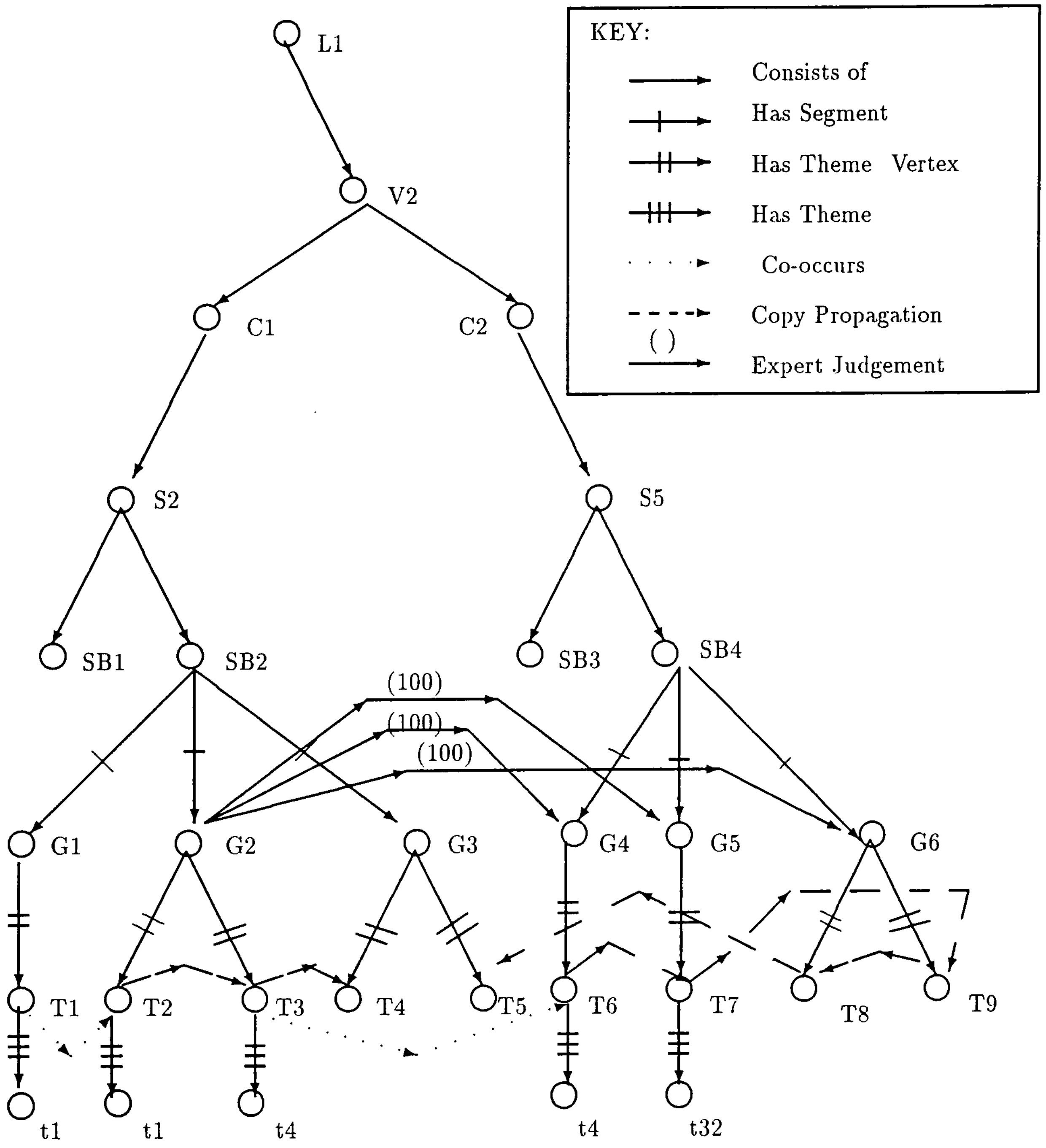


Figure 8.13: Example 2 $\delta CIG(\mathcal{V}, \mathcal{E})$

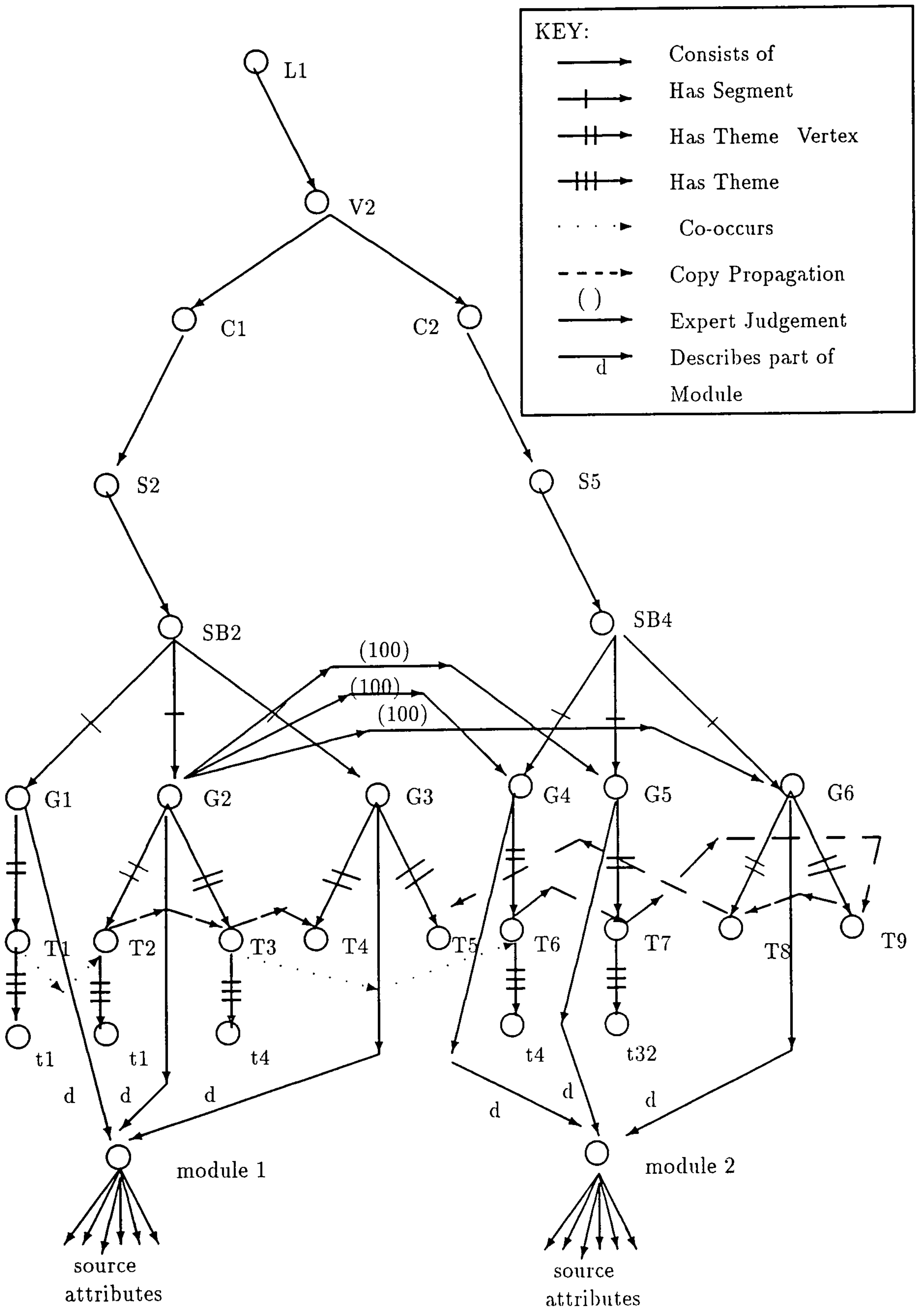


Figure 8.14: Example 2 $\epsilon CIG(\mathcal{V}, \mathcal{E})$

## 8.5 Case Study

The purpose of the case study is to provide insight into the practical feasibility of the $RPG(\mathcal{V},\mathcal{E})$ and thematic graph slicing techniques. The example used for this case study is a real example of system documentation.

### 8.5.1 Scenario Description

The financial system is a system supporting the banking business. The banking system has an onion shaped architecture with all of the banking data files at the centre of the system and layers of transaction processing, networking and application software around the data files. The whole system consists of 900 source code modules. The core banking system (the application software) consists of many different applications supporting the retail banking business and contains 2 million lines of COBOL source code.

A subsystem from the batch processing component was chosen as it is the simplest part of the banking system. A simple system for the case study called the End of Day system (EOD) was selected from an area of the banking system called Batch Applications. Most of these applications read data from files, process the data and then write it back to the same or new data files. The EOD system is designed to fulfill the following functions :

1. Extract all relevant information from the Audit tapes relating to the banking data.
2. Maintain a historical record of transactions processed on a daily, weekly, monthly and quarterly basis, and produce relevant reports for the banks.
3. Control the production of all printing requirements in terms of reports, statements and microfiche.

There are several major suites of programs within the area of Batch Applications. These are the following :

1. Audit Tape Extract (AUT)

2. Transaction History (HST)

3. Reports Production System (RPS)

The main functions of AUT are to extract all relevant data from the day's audit tapes and to sort the data into the required sequence. It also produces files for external transmission at the earliest point after the bank closes down.

The main function of HST is to produce daily and periodic reports for the branches showing complete historical records of all transactions which have been input to the system. This also involves the creation and maintenance of various master files for subsequent reporting and interrogation.

All report lines produced by On-line, Drip Feeds and Sequential Passes, as well as those created by EOD programs, are input to the RPS for subsequent production of statements, microfiche, or paper reports for the branches. A special file is used, namely the Reports Control File to control the various different formats and layouts of these reports.

This case study is based on the AUT program suite. The documentation is not in formal mathematical notation. Instead it is written in natural language. However, the documentation does have a clear hierarchical structure. The case study is taken from the financial services sector. The style of the document maps onto the languages used within this sector. For example the programming language COBOL has procedures which map onto the process descriptions within the documentation example used. This style of documentation was chosen because the interconnections between the documentation entities are more implicit than the document interconnections within mathematically based specifications and designs.

The reason why the financial services were chosen is because the type of processing is very data intensive. There is also much sharing of data through a central data base within the financial services, which is one of the main factors causing the indirect ripple effect.

The actual case study document provided contained 2849 lines in 67 pages and 22908 words. The document was the smallest piece of documentation which could be located within the project sponsoring organisation. The documentation had a hierarchy containing 7 levels and at the bottom level is 437 entities wide. These levels were :

1. Library

2. Volume

3. Book

4. Chapter

5. Process Description

6. Process Elements

7. Segments

The documentation for the entire financial system is stored within a library containing a Chapter for each sub-system. The case study presented in this thesis contains one Chapter. A Chapter describes a complete application. Within this Chapter there are five process descriptions describing the five functions of the system used as the case study. Each process description contains a number of process elements which describe discrete steps within the process. There are 44 different process elements. Process description 1 contains 4 process elements, process description 2 contains 26 process elements, process description 3 contains 1 process element, process description 4 contains 11 process elements and process description 5 contains 2 process elements. These process elements were decomposed into 437 different segments.

The document describes 3 source code modules which form a sub-system. Process descriptions 1 and 2 describe 1 module, process description 3 describes another module and process descriptions 4 and 5 describes the third module. The modules communicate through shared data. The documentation describes the processing and data within each module and therefore the process descriptions are implicitly connected because of this.

The system is written in the programming language COBOL and the data files are stored in a CODASYL data base. The source code is 9000 lines in total. The COBOL source code is 5000 lines long and the data file descriptions are 4000 lines long. The supporting documentation consists of system presentation and system technical documentation. The system presentation gives an overview of the application whilst the system technical documentation describes the source code of the application.

The audit tape extract program suite is the first EOD suite to be run operationally each day. It comprises 9 live programs. Collectively these programs use many of the database areas of the CORE banking system

The major program in the suite is AUT01. It is the most important program in the whole EOD system since it extracts all relevant transactions from the day's audit tapes, reformats them where necessary and passes them on to the appropriate areas. The other AUT programs process files created by AUT01, and produce output files for immediate transmission to external agencies or for examination at the bank's computer centre.

The AUT01 program was selected for this case study. Figure 8.15 shows the relationship between the modules and the documentation of the AUT01 sub-system.

This program consists of 3 modules, namely AUT01a, AUT01b and AUT01c. These three modules provide input, sorting and output functions respectively. All of these modules are described by the document used in this case study. Process descriptions 1 and 2 describe module AUT01a, process description 3 describes AUT01b and process descriptions 4 and 5 describe module AUT01c.

The AUT01 program takes as input, the Audit Tapes and Reconciliation Movements. It sorts these records and outputs them to the following files : Reconciliation Movements Drip Feed, Reconciliation Movements Drip Feed Control File, Interest Capital Transactions File, Interest Capital Control File, Sorted Transaction File, Report Lines File, External Transactions, AUT control File, and Computer Centre Report Lines File.

The following data flow diagram presented in fig 8.16 describes the main stores and processes which comprise the AUT01 sub-system of Batch Application System. The documentation used as the case study describes this process. The rest of this Chapter describes the employment of the $RPG(\mathcal{V}, \mathcal{E})$ and its analysis techniques to the EOD system.

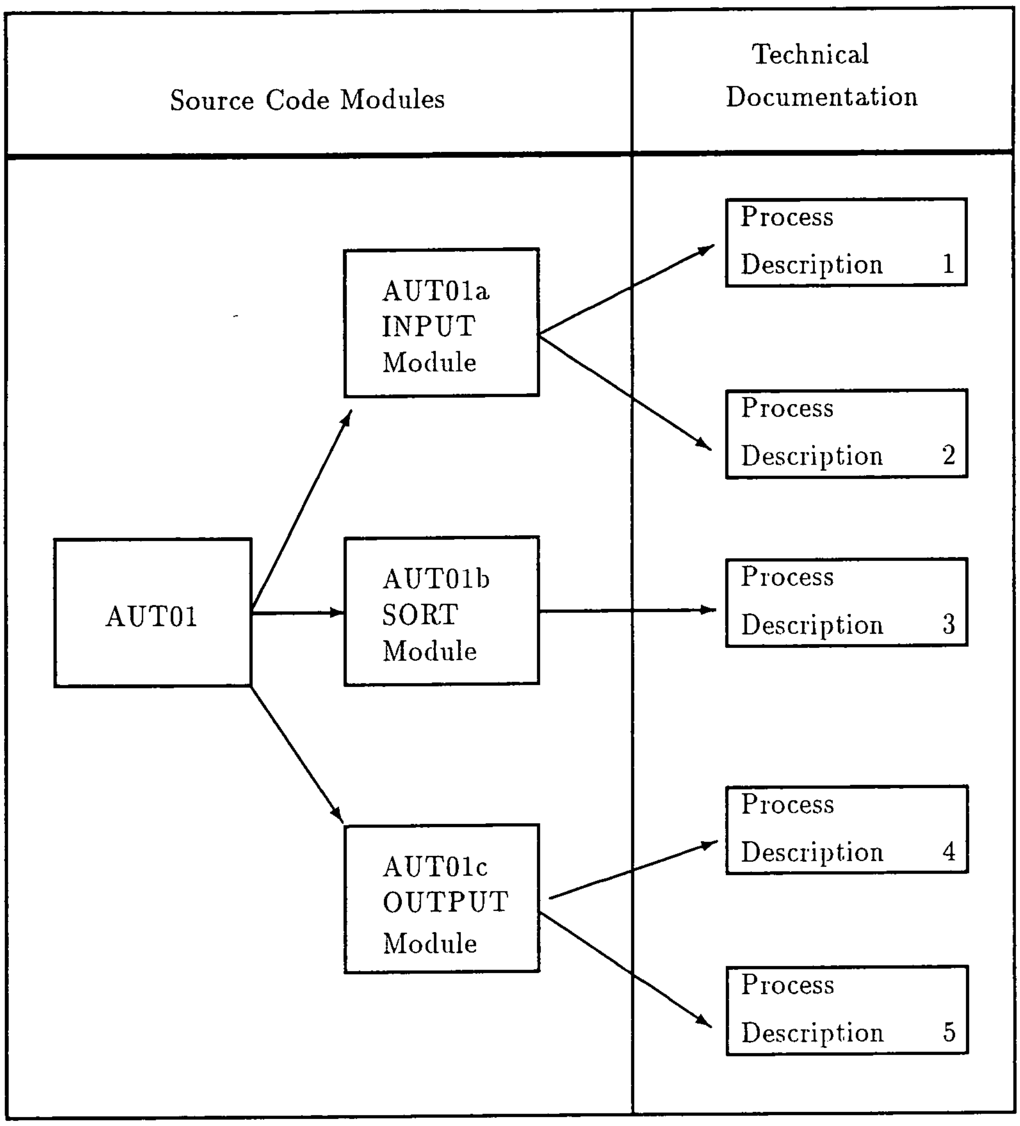


Figure 8.15: Case Study Traceability between the code and documentation

### 8.5.2 Graph Construction

#### Step 1: $HIG(\mathcal{V},\mathcal{E})$ Crystallisation

Analysis of the documentation gives a simple context diagram, see figure 8.17. This context diagram does not reflect the actual structure of the document analysed in this case study. Instead

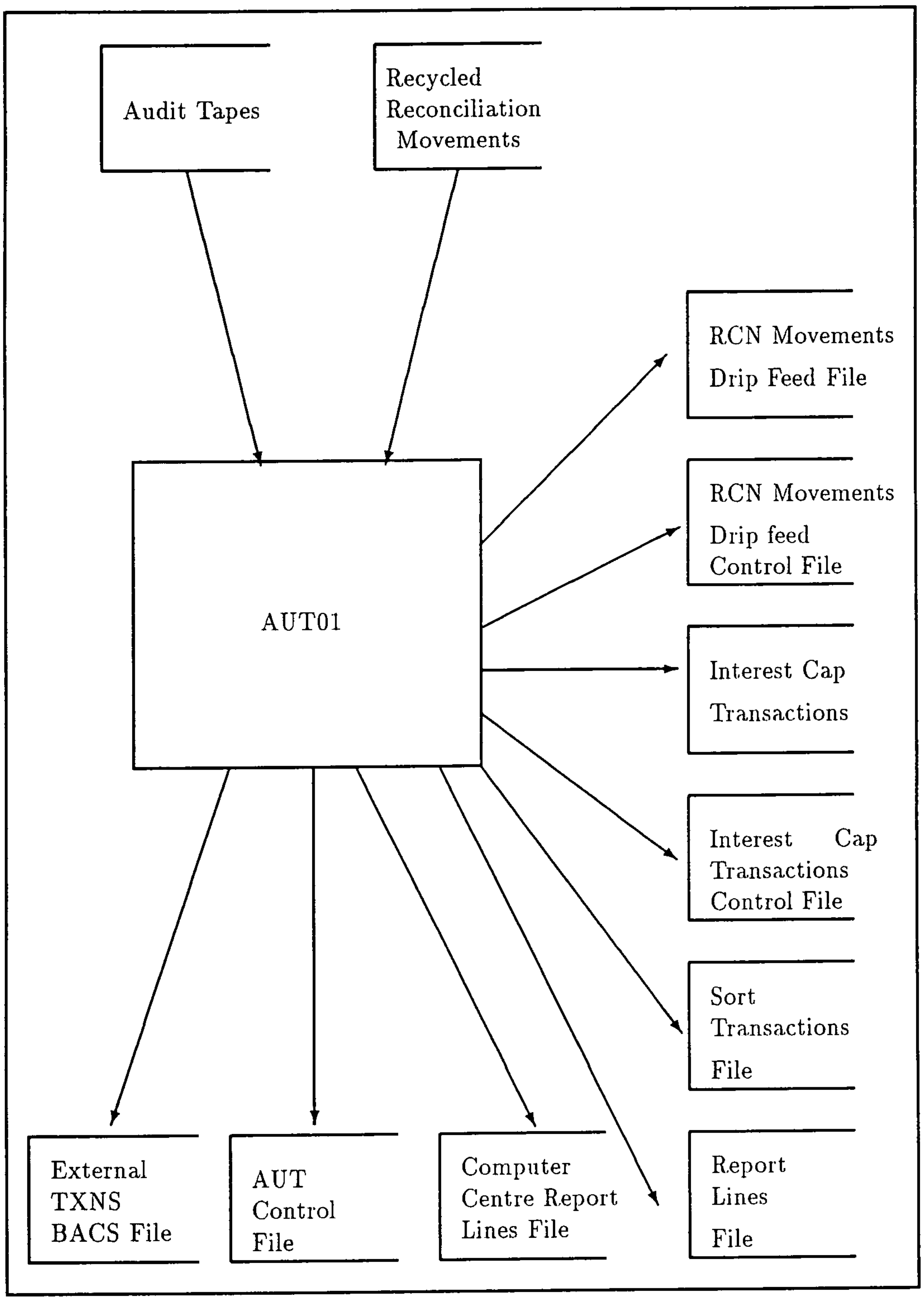

Audit Tapes
Recycled Reconciliation Movements
RCN Movements Drip Feed File
RCN Movements Drip feed Control File
AUT01
Interest Cap Transactions
Interest Cap Transactions Control File
Sort Transactions File
External TXNS BACS File
AUT Control File
Computer Centre Report Lines File
Report Lines File

Figure 8.16: An Overview of the System to be Analysed

the context diagram reflects the **entity types** which may occur in the hierarchy and at which levels in the hierarchy they are permitted to occur.

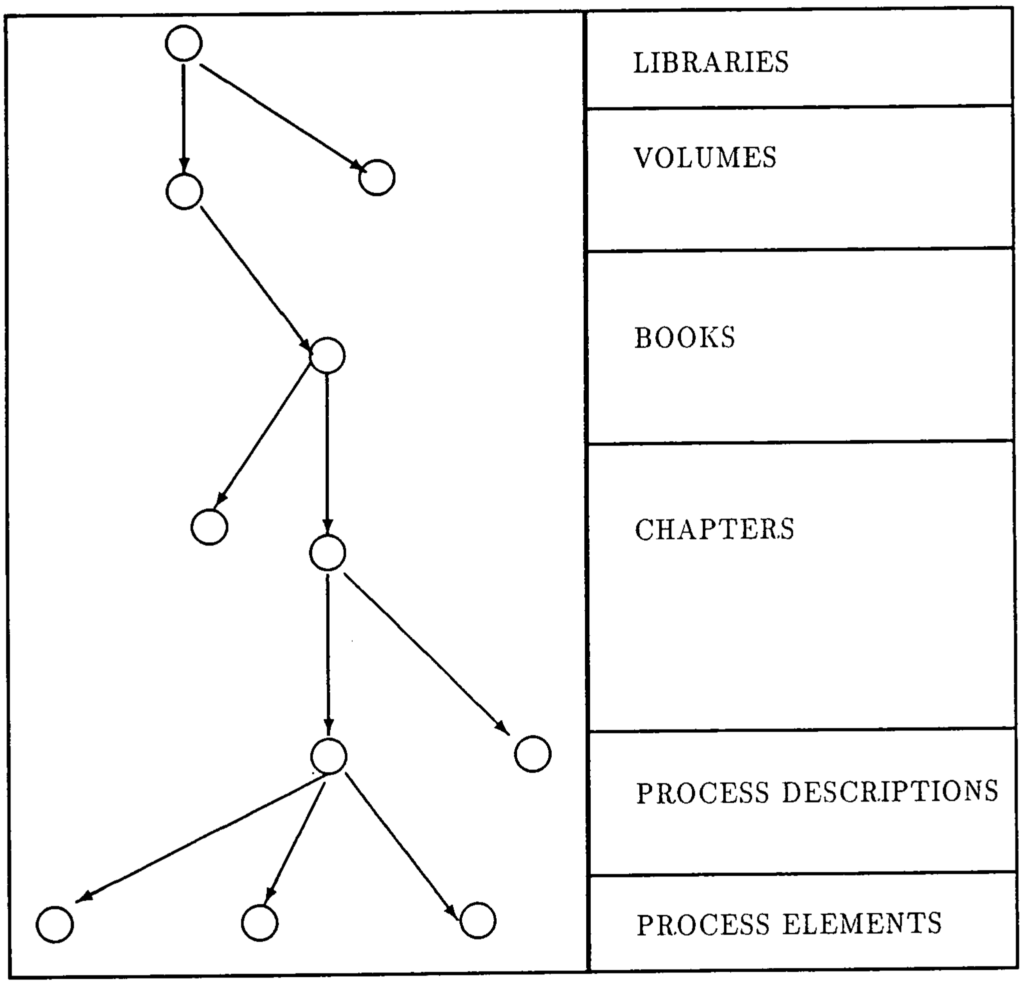


Figure 8.17: A Simple Context Sketch of the $RPG(\mathcal{V}, \mathcal{E})$ Entity Types

Figure 8.18 shows an example of a fragment of text from the documentation case study. The fragment of text shown in figure 8.18 is decomposed into four segment entities. This factoring process was performed by dividing the document fragment into rectangles as shown in figure 8.19. Figure 8.20 shows an example of the graph extracted from the fragment of text in figure 8.18. Each rectangle represents a document entity type. The case study was also mapped to graph fragments like the graph presented in figure 8.20

Once the set of segment entities were created they were analysed with respect to their thematic structure.

ELEMENT    INITIAL ENTRY TO AUDIT TAPE DEBLOCK
ELEMENT SUMMARY

The standard routine is informed which types of audit tape records are required by the extract and will validate the processing range determined by Operator Communication against the audit tapes to be read. If the range is not consistent with information on each audit tape header the routine will inform the Control procedure which will take the appropriate action.

1. The Application Date is requested from the Operator using Operator Communication Standard Routine (OCSR).

2. The date is validated by the Date Validation Standard Routine, if it is invalid, a message 'Invalid Date' is output and the operator is requested to re-input the date.

3. If this is the first run of the day the Application Date input in step 1 is stored for updating the AUT Control File at the end of run.

Figure 8.18: An Example of a Documentation Fragment Containing Hierarchical Structure

```
-----------------------------------------------------------------------
|  ELEMENT    INITIAL ENTRY TO AUDIT TAPE DEBLOCK                     |
| ------------------------------------------------------------------  |
| | ELEMENT SUMMARY                                                 | |
| |                                                                 | |
| | The standard routine is informed which types of audit tape      | |
| | records are required by the extract and will validate the       | |
| | processing range determined by Operator Communication against the | |
| | audit tapes to be read. If the range is not consistent with     | |
| | information on each audit tape header the routine will inform the | |
| | Control procedure which will take the appropriate action.       | |
| |-----------------------------------------------------------------| |
| |  1.      The Application Date is requested from the Operator using | |
| |          Operator Communication Standard Routine (OCSR).        | |
| |-----------------------------------------------------------------| |
| |  2.      The date is validated by the Date Validation Standard  | |
| |          Routine, if it is invalid, a message 'Invalid Date' is | |
| |          output and the operator is requested to re-input the   | |
| |          date.                                                  | |
| |-----------------------------------------------------------------| |
| |  3.      If this is the first run of the day the Application Date | |
| |          input in step 1 is stored for updating the AUT Control | |
| |          File at the end of run.                                | |
| ------------------------------------------------------------------  |
-----------------------------------------------------------------------
```

Figure 8.19: An Example of a Factored Documentation Fragment

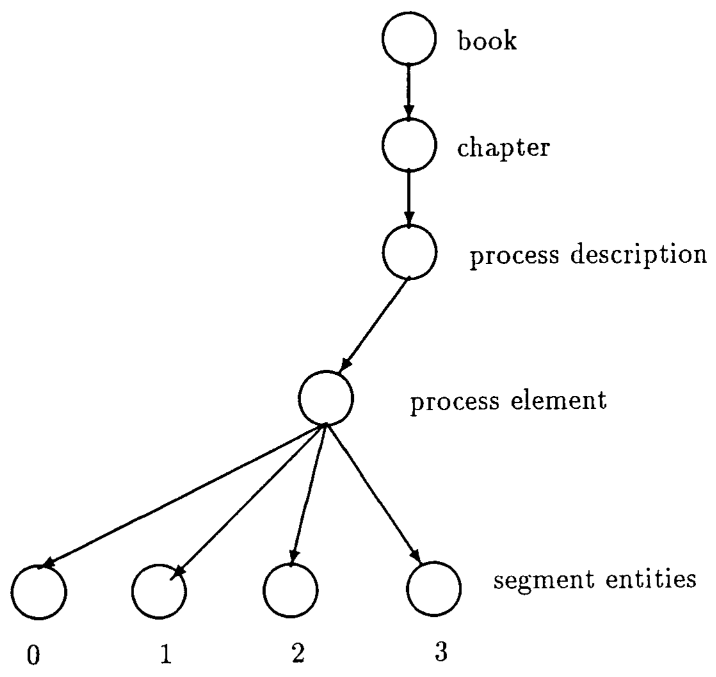


Figure 8.20: An Example of an $HIG(\mathcal{V},\mathcal{E})$

**Step 2: $TIG(\mathcal{V},\mathcal{E})$ Crystallisation**

Before each segment entity could be coded with one or more themes, a theme catalogue had to be created. The appendix contained within the documentation described the data files used by the modules AUT01a, AUT01b and AUT01c. Therefore a simple list of data items which may be described within the documentation was created. Each of the segments in the $RPG(\mathcal{V},\mathcal{E})$ was analysed for its content. This was conducted manually. In terms of size, all of the segments were between two lines and ten lines. In terms of information content they were all about the same. For example most segment entities contained one, two or three themes.

Where copy propagations were detected, the thematic context of themes were recorded, for example, whether or not the theme is a definition or use of a data item. Figure 8.21 is an example of a segment isolated from the process description 1 in the case study.

The "11" is the identification number of this text fragment and the "C134E012" is the project code which produced this piece of documentation. The "account number", "low values" and "parent account number" are the Conceptual Objects and the "move" is the Conceptual Action. The word "If" implies a conditional logic construct and the word "move" implies an assignment. Each theme

```
11  If  the  account  number is not present and not available, C134E012
    move low values to parent account number.                   C134E012
```

Figure 8.21: An Example of a Document Fragment Containing a Segment Entity

is given a theme code. The themes are added to the set of known themes if they are not already members of the set of themes. "Low values" is assigned "th1", "account number" is assigned "th2" and "parent account number" is given "th3". The resulting graph fragment is shown in figure 8.22. The dotted line in figure 8.22 records the "move", i.e., the copy propagation.

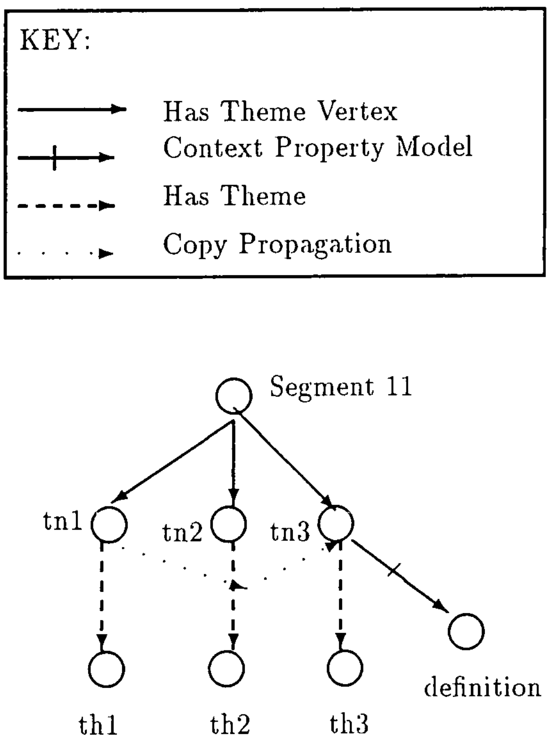


Figure 8.22: A graph recording the content of a segment entity

The whole text fragment is mapped to a segment on the graph and each CO is modelled with a theme vertex. The "parent account number" is the recipient of the data in the assignment described, therefore the thematic context is the description of a data definition. This is used to ensure that the edge representing the copy propagation points in the right direction. Each segment content was modelled in this way.

## Step 3: $WIG(\mathcal{V}, \mathcal{E})$ Parameterisation

No directed fine grained release information was available for this particular case study because the bank does not collect this type of information. Therefore simulated release information was used to put weights on the edges joining the segment entities (direct edge parameterisation). Some of the weights were also based on simulated quantitative expert judgements (indirect edge parameterisation) concerning potential ripple propagation. Figure 8.23 shows an example of the addition of a weighted edge to the $WIG(\mathcal{V}, \mathcal{E})$ between segment 4 and segment 1. The weight on this particular graph has been extracted from previous release information using the guidelines in Section 5.6.3.

Figure 8.23 also shows an example of the addition of a weighted edge between segment 4 and segment 1, containing a probability of a ripple effect between two segment entities based on expert judgement using guidelines presented in Section 5.6.2. This particular probability has been arrived at by the consensus of maintainers associated with the maintenance of the system which is used as the case study.

There is also another weighted edge added between segment entity 13 and segment entity 4. This is an example of an implicit link between segment entity 13 and segment entity 4. It is implicit because there is no link between this pair of segment entities based on thematic dependencies. However from past experience the weight shows that if segment entity 13 is changed then segment entity 4 must also be changed.

## Step 4: $SAG(\mathcal{V}, \mathcal{E})$ Annotation

The segment entities were also scanned for any source code entities which were implicitly or explicitly mentioned. For example the source modules, and test files which are described by the documentation were recorded. Figure 8.24 shows an example of a segment which describes a module called AUT01a. Within this segment several source code entities are described. Figure 8.25 shows the model extracted from the case study described by the scenario.

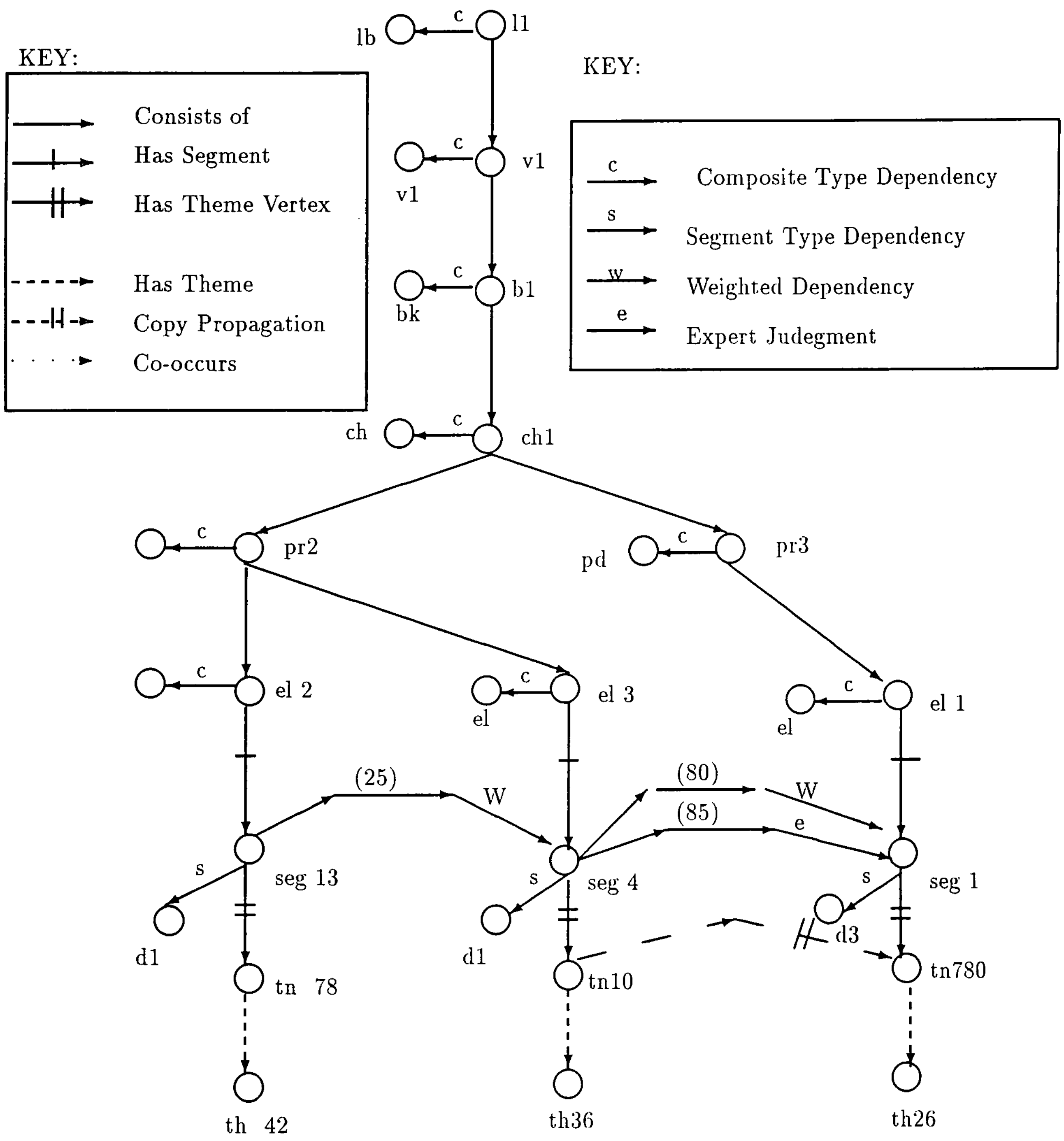


Figure 8.23: An Example of Edge Parameterisation

KEY:

| SAG Dependencies |
|---|
| $\cdots\rightarrow$ Describes part of module |
| $\xrightarrow{a}$ module type |
| $\xrightarrow{b}$ module test file |
| $\xrightarrow{c}$ system test required |
| $\xrightarrow{d}$ module data file |
| $\xrightarrow{e}$ module map base |

| SAG Entities | |
|---|---|
| 1 | Cobol |
| 2 | Module test 1 |
| 3 | Module test 3 |
| 4 | Module test 4 |
| 5 | End of Day 1 |
| 6 | End of Day 2 |
| 7 | Reconciliation File |
| 8 | Audit Tapes File |
| 9 | Reconciliation Map Base |
| 10 | Audit Tapes Map Base |

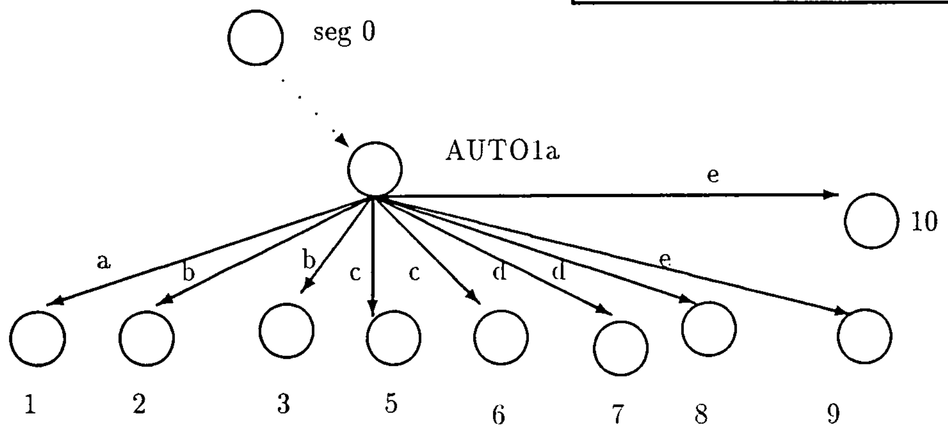


Figure 8.24: An Example of Annotation

### 8.5.3 Examples of Simple Slices

**Step 5: $\alpha$ Slicing**

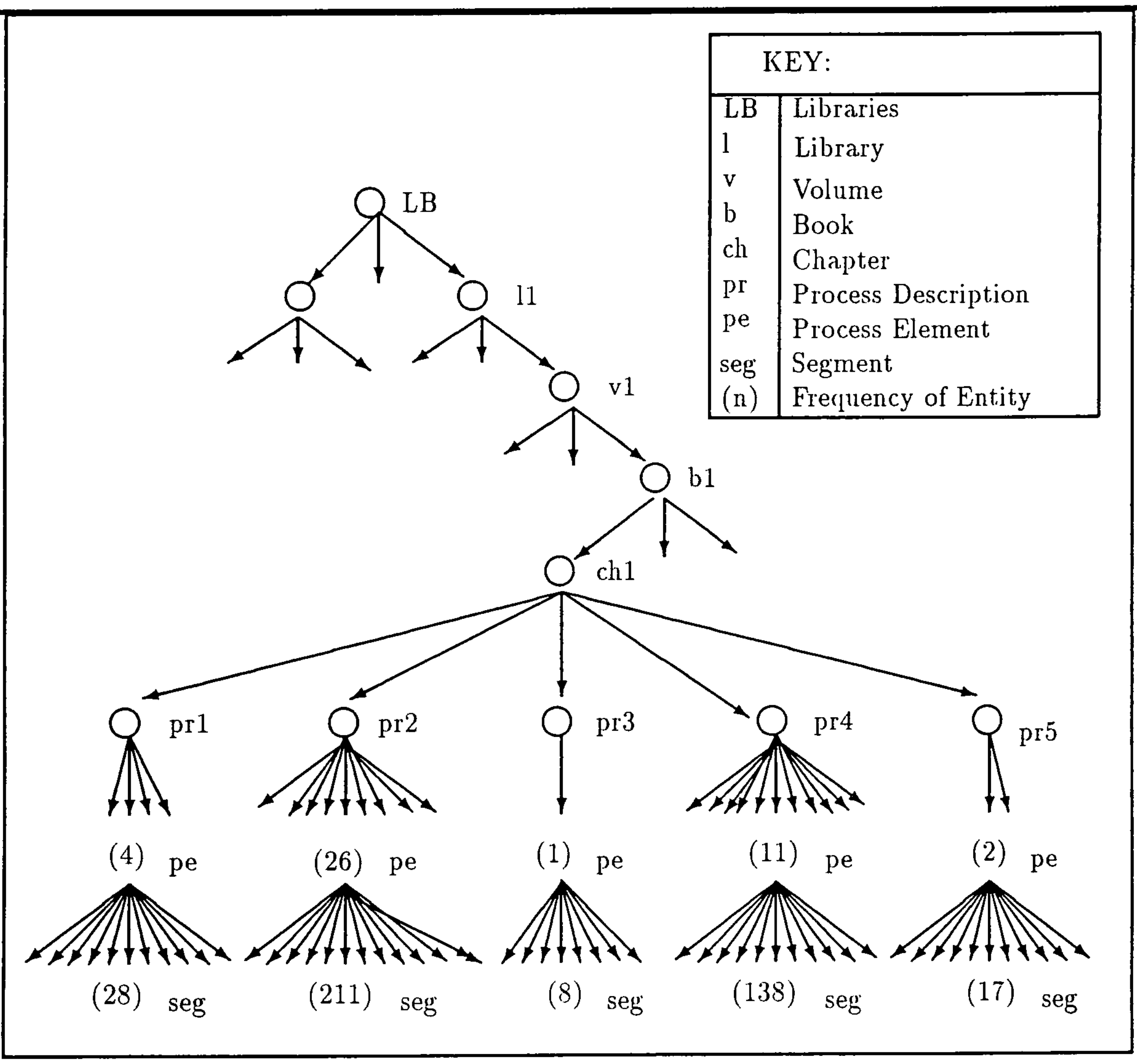


Figure 8.25: $RPG(\mathcal{V}, \mathcal{E})$ Case Study

Assuming a $\mathcal{TB}$ containing the following theme $\prec$ th5 $\succ$ has been created from a change proposal, the resulting $\alpha CIG(\mathcal{V}, \mathcal{E})$ is shown in figure 8.26. The theme code "th5" represents a change to any document entities describing the processing of aut control file. Figure 8.26 is an abstraction of the graph shown in figure 8.25. The graph indicates that all the segment entities have the theme code $\prec$ th5 $\succ$. Therefore it can be concluded that the entities pr1, pr2 and pr5 are all potentially impacted by the change proposal containing "th5".

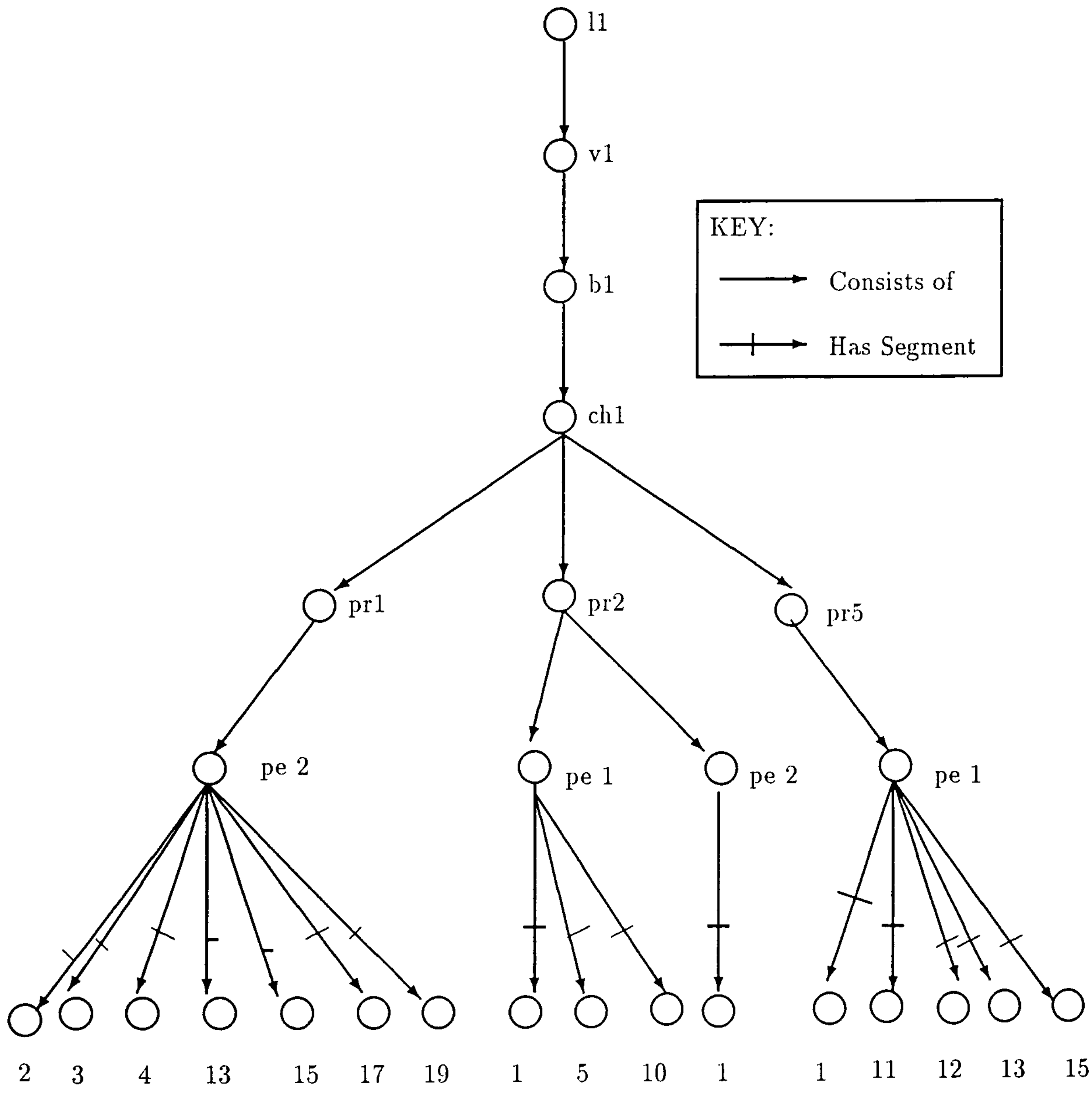


Figure 8.26: Case Study $\alpha CIG(\mathcal{V}, \mathcal{E})$

## Step 6: $\beta$ Slicing

The theme th34 is chosen to simulate a change proposal. The theme th34 represents a change affecting the "processing of the recycled reconciliations movements". This theme was used as a slicing criterion for the beta slice. The use this particular slice and slice criterion on the case study resulted in a small subgraph of the $RPG(\mathcal{V}, \mathcal{E})$ which actually contained no copy propagations. However the resulting $\beta CIG(\mathcal{V}, \mathcal{E})$ was an better characterisation of the impact than that produced by the $\alpha$ slice. The $\beta CIG(\mathcal{V}, \mathcal{E})$ produced contains all the types of entities impacted by the $\mathcal{TB}$. This helps a maintenance manager deduce what may be impacted at the source code level without consulting any source code. It can be seen from consulting the graph that segment entities of types d1 and d2 are impacted. Types d1 and d2 represent the description of input and output respectively. The $\beta CIG(\mathcal{V}, \mathcal{E})$ is shown in figure 8.27.

By examining the segment entities and segment entity types in this extracted subgraph it can be seen that the impacted segment entity types are d1 and d2. In this particular case study d1 and d2 represent the description of input and output respectively. This particular subgraph informs a maintenance manager or maintainer that the proposed change impacts the input module and output module of the program AUT01 because entities pr1 and pr5 describe these modules and are both in extracted the subgraph.

The next example of a $\beta$ slice shown in figure 8.28 demonstrates how the copy propagation dependency can be used to consider indirect impacts at the documentation level.

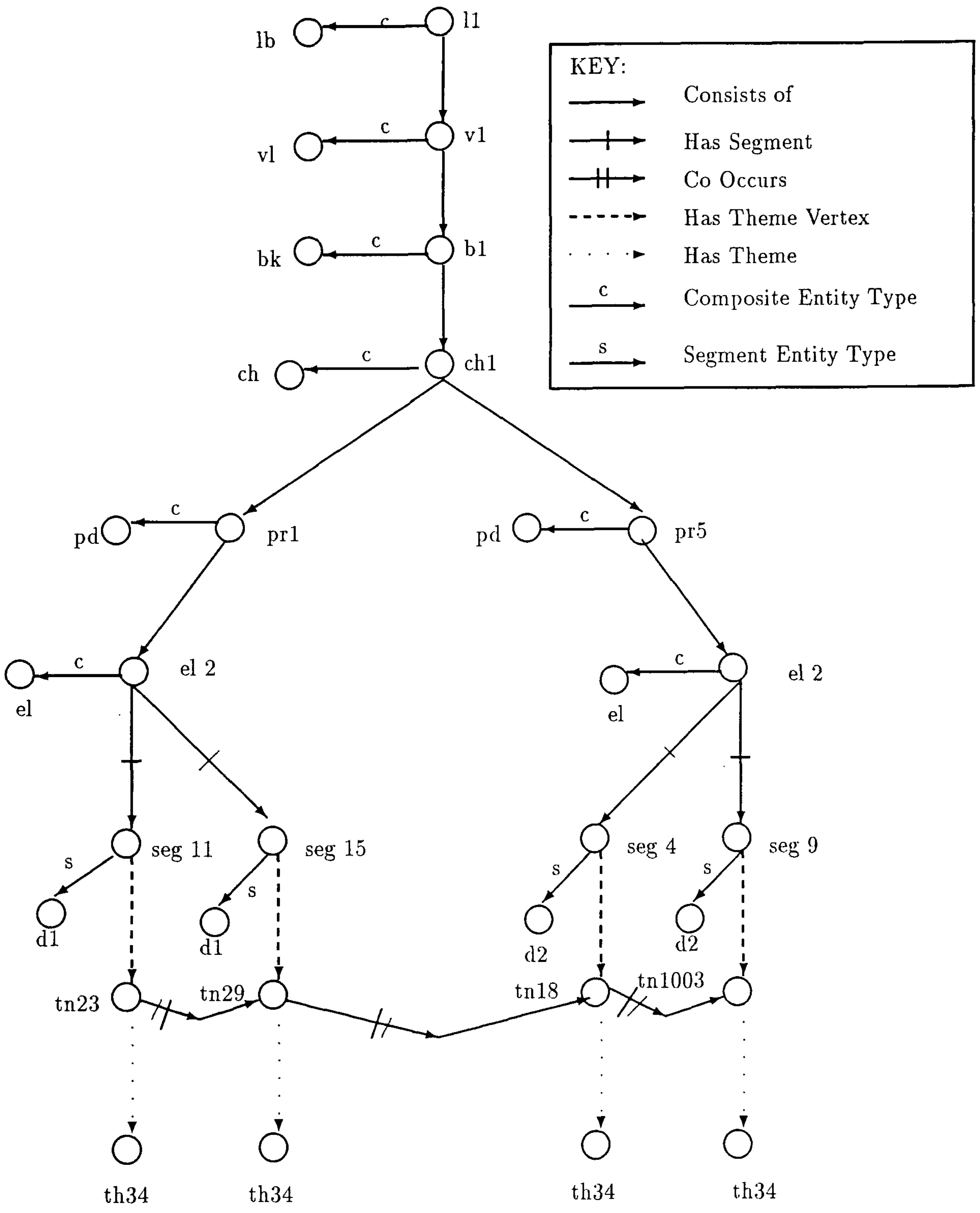


Figure 8.27: Case Study $\beta CIG(\mathcal{V}, \mathcal{E})$

### 8.5.4 A Change Proposal

This example of a proposed change only contains the 'Business Summary' and the 'Detailed Business Requirements' as these are the only sections of the change proposal which are used for constructing a theme bag.

---

1. **Proposal Identification Information.**

   *Feasibility No: 1, 'Revise Region Code' October 91*

2. **Business Summary.**

   *The proposal is to change the style of the region code data item.*

3. **Detailed Business Requirements.**

   *The proposal is to <u>incorporate</u> the revised <u>region code</u> onto the system.* The code will changed from numeric to alphanumeric.

4. **Service Levels.**

   *This proposal will help to ensure that existing service level agreements are met on a more regular basis.*

5. **Time scales.**

   *The project can be implemented with very little effort if the implementation coincides with other similar database changes.*

6. **Assumptions and Constraints.**

   *The region code is the only data item to be considered.*

---

### 8.5.5 Proposal Analysis

The proposal can be decomposed into the following CAs and COs :

**CAs:** incorporate

**COs:** region code

These CAs and COs can be simply related in the following way :

1. incorporate region code

The following table 8.2 represents a fragment of a catalogue of themes. In this particular case study only one theme descriptor is used since the business system data names are the same as the software system names of data.

| theme code | theme description |
|---|---|
| th34 | describes-the-processing-of-recyc-rec-movements-file |
| th35 | describes-the-processing-of-region-closure-indicators |
| th36 | describes-the-processing-of-region-code |
| th37 | describes-the-processing-of-region-of-origin-branch |
| th38 | describes-the-processing-of-reports-data-block |
| th39 | describes-the-processing-of-result-indicator |
| th40 | describes-the-processing-of-scottish-pre-cutover |

Table 8.2: Case Study A Theme Catalogue

The CAs, one in this case i.e., 'incorporate' is mapped onto the 'describes-the-processing' part of the theme and the COs are mapped onto the '-of-region-code' part of the theme. By identifying the themes within the system a theme bag can be constructed.

$$\mathcal{TB} = \prec \text{th36} \succ$$

### 8.5.6 Slicing and Results

The theme th36 represents a change affecting the "processing of region code". The theme th36 co-occurs in several segment entities and has a ripple effect on another process description through a copy propagation dependency. The other theme affected is th26 which "describes the processing of branch code". By traversing up the graph it is possible to conclude that another process description is affected by the change i.e., process description pr3.

This particular subgraph shown in figure 8.28 informs a maintenance mananager or maintainer that process description 2 and process description 3 will be impacted by the proposed change. The copy propagation dependency between theme vertex tn10 and tn780 indicates why the process description 3 is impacted.

Considering the $\beta$ slice shown in figure 8.28, if the theme bag was increased from $\prec$ th36 $\succ$ to $\prec$ th36,th26,th26 $\succ$ the resultant graph remains exactly the same. However the theme bags $\mathcal{BE}$ and $\mathcal{BV}$ actually increase in size to reflect the multiple impacts on the entities in the $\beta$ slice. For example the bags will be the same as the sets representing the graph except that the directed path from l1 to th26 will be in the bag $\mathcal{BV}$ three times and the entities on this path will occur in the bag $\mathcal{BV}$ three times. This is because the theme bag directly impacts theme vertex tn780 twice and it is also impacted indirectly from theme vertex tn10. Therefore to estimate the volume of work involved in processing the change $\prec$ th36,th26,th26 $\succ$ it would incorrect to base the estimation on the graph. Instead the bags $\mathcal{BE}$ and $\mathcal{BV}$ should be used.

## Step 7: $\gamma$ Slicing

The $\gamma CIG(\mathcal{V},\mathcal{E})$ is shown in figure 8.29. The same theme bag is used, that is th36. The $\gamma CIG(\mathcal{V},\mathcal{E})$ produced contains the same features as the previous graph except that the chance of ripple propagation is included. This probability is based on simulated previous ripple effects. The graph shown in figure 8.23 shows two weighted dependencies 25% and 80%. When the $\gamma$ slice is applied to the $RPG(\mathcal{V},\mathcal{E})$ with $\mathcal{LP}$ equal to 80% and $\mathcal{HP}$ equal to 100%, then only weighted edges within this range of probabilities appear in the resultant $\gamma$ $CIG(\mathcal{V},\mathcal{E})$ presented in figure 8.29.

The weighted edge shows there is an 80% chance that if segment 4 is changed then segment 1 will be affected. In this case it shows that it is very likely that the proposed change to segment 4 will propagate a change to segment 1. In terms of the case study this extracted subgraph suggests that both modules AUT01a and AUT01b will be impacted.

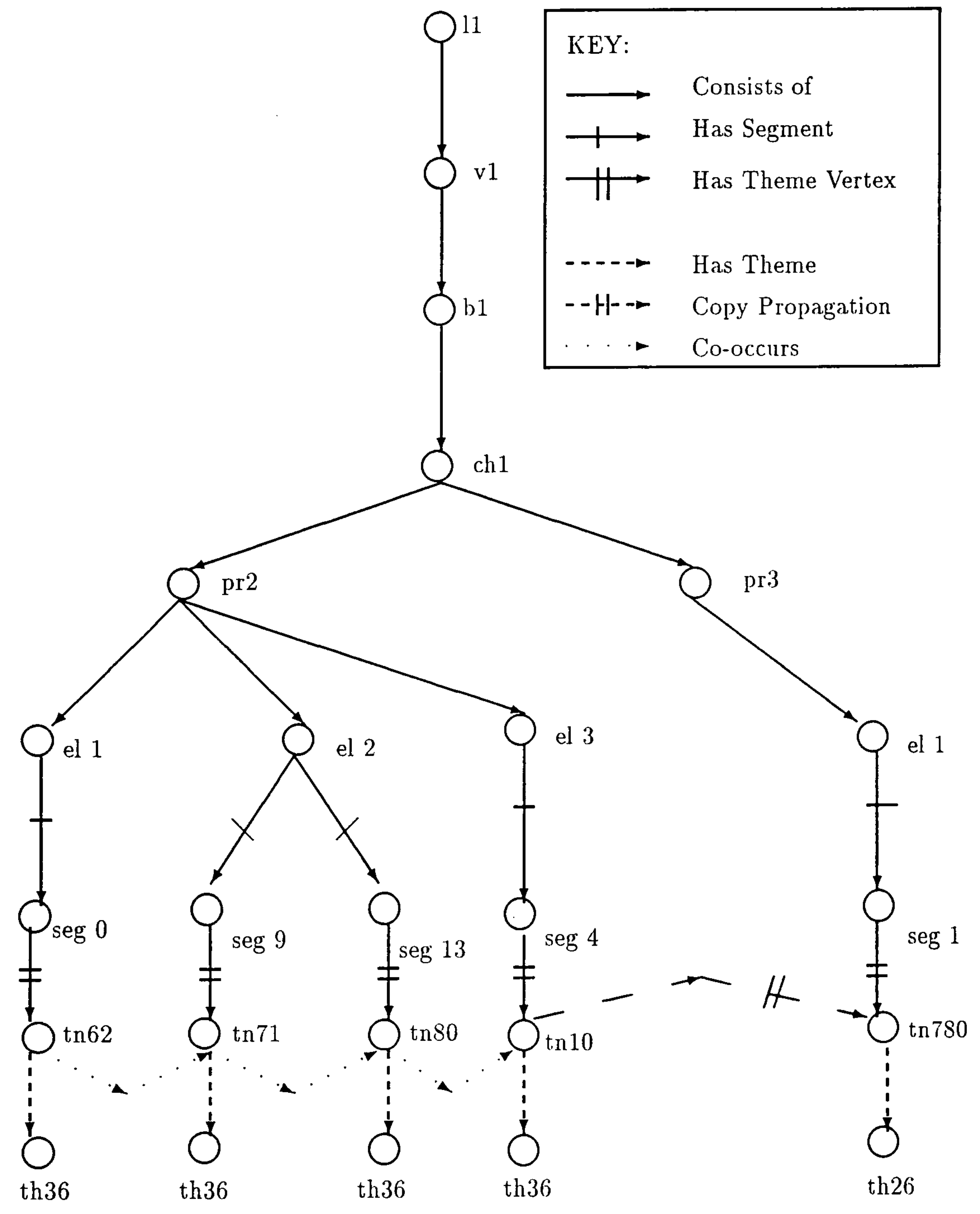


Figure 8.28: Case Study A further $\beta CIG(\mathcal{V}, \mathcal{E})$

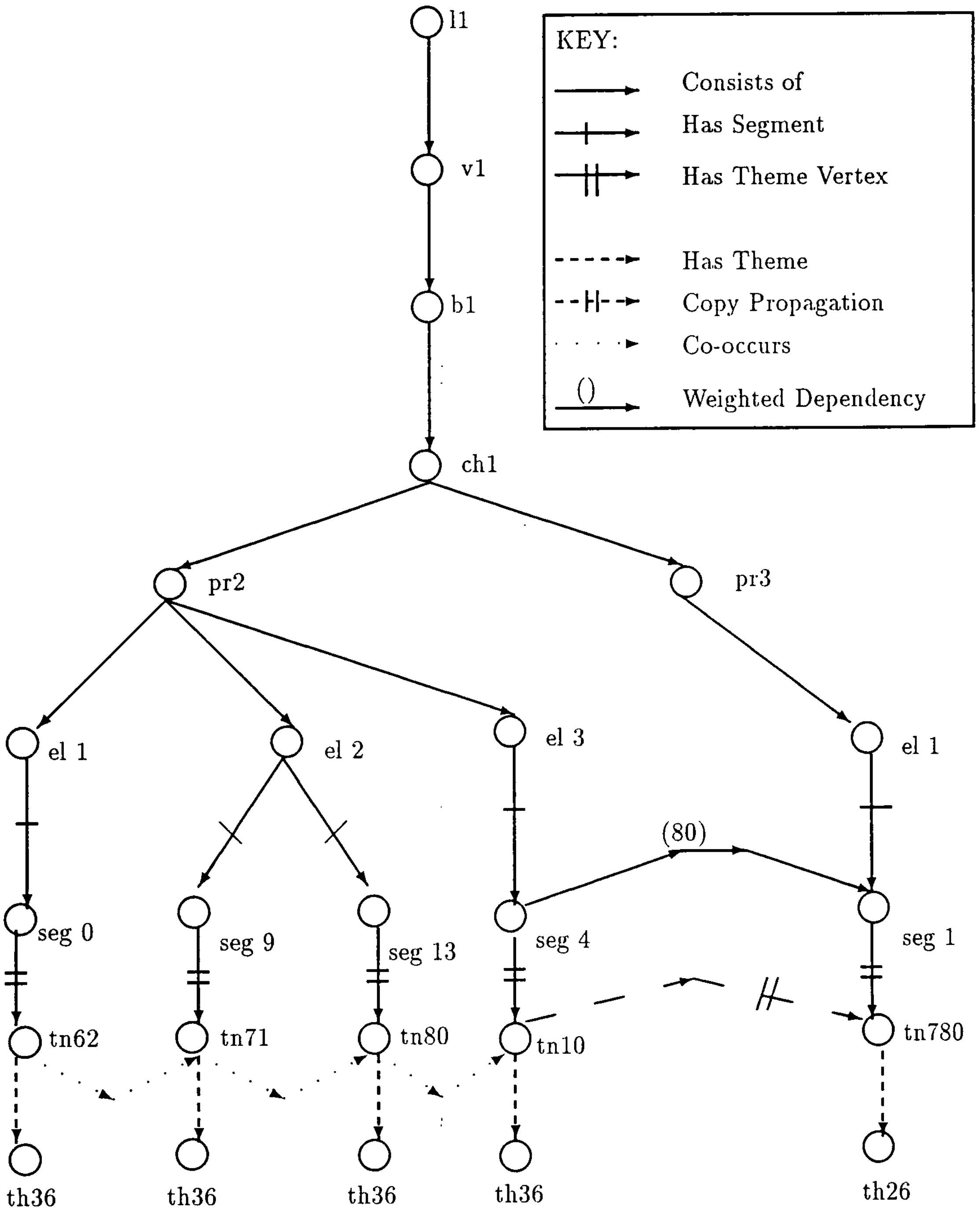


Figure 8.29: Case Study $\gamma CIG(\mathcal{V}, \mathcal{E})$

## Step 8: $\delta$ Slicing

The $\delta CIG(\mathcal{V},\mathcal{E})$ is shown in figure 8.30. The $CIG(\mathcal{V},\mathcal{E})$ produced contains the same features as the previous graph except that the chance of ripple propagation is included based on simulated expert judgement about ripple effects. When the $\delta$ slice is applied to the $RPG(\mathcal{V},\mathcal{E})$ with $\mathcal{LP}$ equal to 85% and $\mathcal{HP}$ equal to 100%, then only weighted edges within this range of probabilities appear in the resultant $\delta$ $CIG(\mathcal{V},\mathcal{E})$, which is presented in figure 8.30. The graph indicates that there is 85% chance of the ripple effect being propagated from segment entity 4 to segment 1. This probability has been estimated by maintainers who have previously maintained the system.

In terms of the case study the $\delta$ $CIG(\mathcal{V},\mathcal{E})$ suggests that module AUT01a and AUT01b will be impacted and if module AUT01a is changed then there is 85% chance that module AUT01b will also be impacted as a result of this change.

## Step 9: $\epsilon$ Slicing

The $\epsilon CIG(\mathcal{V},\mathcal{E})$ is shown in figure 8.31. The extra key for the $\epsilon CIG(\mathcal{V},\mathcal{E})$ is shown in figure 8.32. This graph shows the actual source code entities which are impacted. This particular subgraph shown in figure 8.31 informs a maintenance manager or maintainer that process description 2 and process description 3 will be impacted by the proposed change. The copy propagation dependency between theme vertex tn10 and tn780 indicates the description of an assignment of data to a data item. This is the reason why the process description 3 is impacted.

One of the most important features of this particular type of $CIG(\mathcal{V},\mathcal{E})$ is that it shows the type of source code entities which are described in the impacted segment entities.

For example the process description 2 (pr2) shown in 8.31 is impacted because segment entities 0,9,13 and 4 are all impacted by theme th36 (the processing of region code). The segment entities 0,9,13 and 4 all describe part of the module AUT01a. The $CIG(\mathcal{V},\mathcal{E})$ also informs the maintenance manager or maintainer that the documentation describing this module also contains descriptions of source code entities related the module AUT01a. The dependencies shown in 8.31 indicate these related source code entities. The meaning of the dependencies a to e and the entities 1,2,3,5,6,7,8,9,10 etc are shown in key associated with this graph in figure 8.31. The $SAG(\mathcal{V},\mathcal{E})$

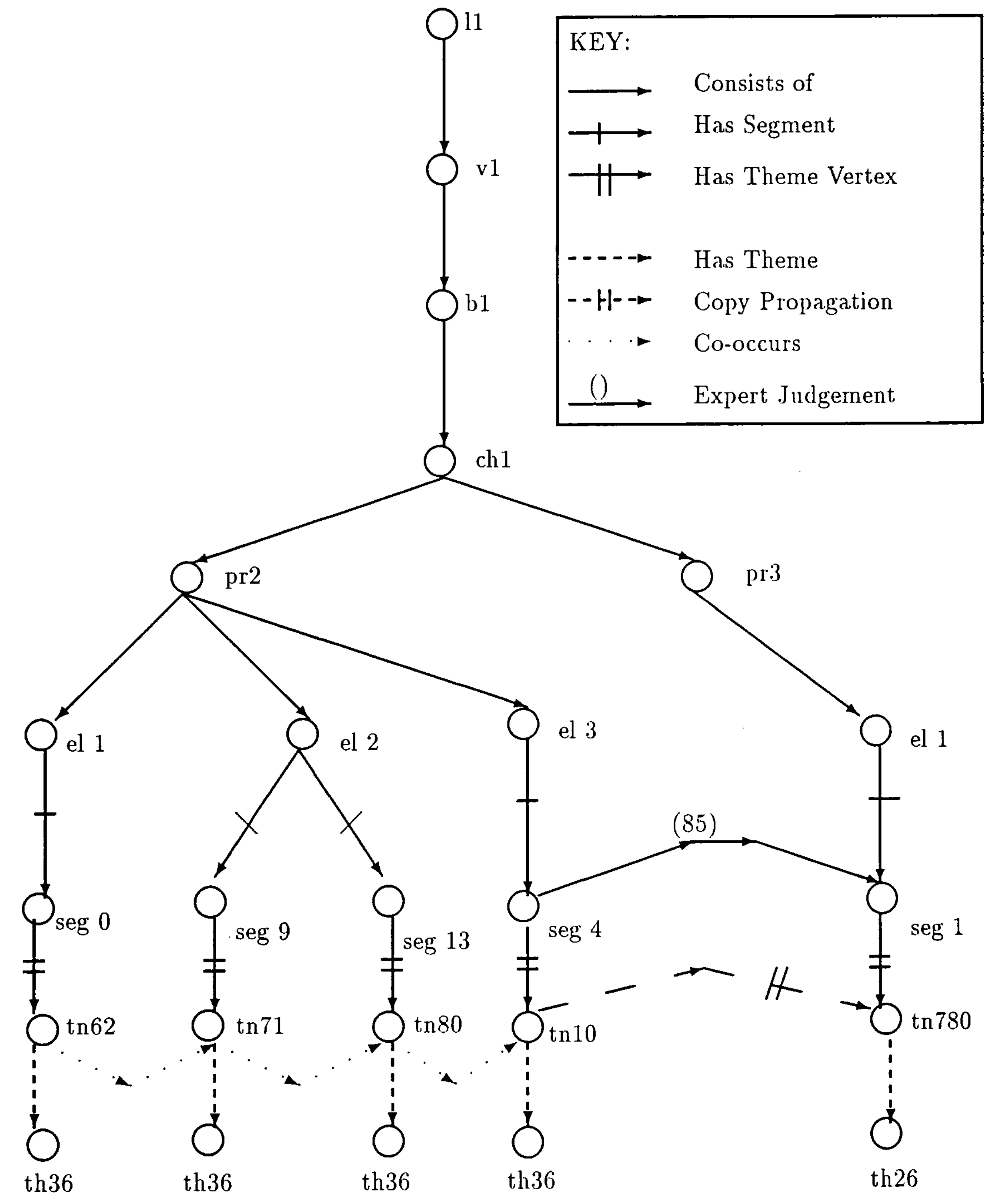


Figure 8.30: Case Study $\delta CIG(\mathcal{V}, \mathcal{E})$

entities associated with AUT01a inform a maintenance manager that if process description 2 of Chapter 1 is changed then the COBOL module AUT01a will need to be changed and tested with module tests 1,3,4 and the system tests End of Day 1 and 2. The dependencies marked d and e also indicate the data files and map bases which may be implicated by the change. As segment entity 1 of process description three is impacted therefore module AUT01b is also impacted with its associated source code entities 1,4,5 and 6.

The combination of all of the results of the slicing techniques together provides many viewpoints of the interconnectivity of the documentation in one graph slice. This is shown in figure 8.33.

### 8.5.7 Discussion

In this case study each of the nine steps of the $RPG(\mathcal{V},\mathcal{E})$ construction and analysis have been demonstrated on a real documentation subsystem from the financial services sector.

It has been shown how the themes can be detected within a change proposal and then formally traced to operational software by analysing documentation interconnectivity and content. The use of probabilities of ripple effects based on both expert judgement and previous ripple effects, can help a manager focus on a particular document entity or module which has a high probability of a ripple effect.

The $HIG(\mathcal{V},\mathcal{E})$ Crystallisation technique was very simple to apply to this case study once the context sketch had been drawn. The $TIG(\mathcal{V},\mathcal{E})$ Crystallisation technique relies on human interpretation of the meaning of a document entity. This enables the precise extraction of themes. The $WIG(\mathcal{V},\mathcal{E})$ Parameterisation technique was applied using simulated data.

The $SAG(\mathcal{V},\mathcal{E})$ Annotation technique relies on human interpretation of the meaning of a document entity. Therefore, like the $TIG(\mathcal{V},\mathcal{E})$ Crystallisation technique, it enables the precise extraction of the description of source entities in document entities. The recording of all of the features mentioned in the $RPG(\mathcal{V},\mathcal{E})$ seems an intuitively attractive thing to do. However in practice the content analysis processes must be supported by software tools, if $RPG(\mathcal{V},\mathcal{E})$ models are to be constructed for large systems.

One notable feature of the $CIG(\mathcal{V},\mathcal{E})$ is the size of the graphs produced even for a small case study.

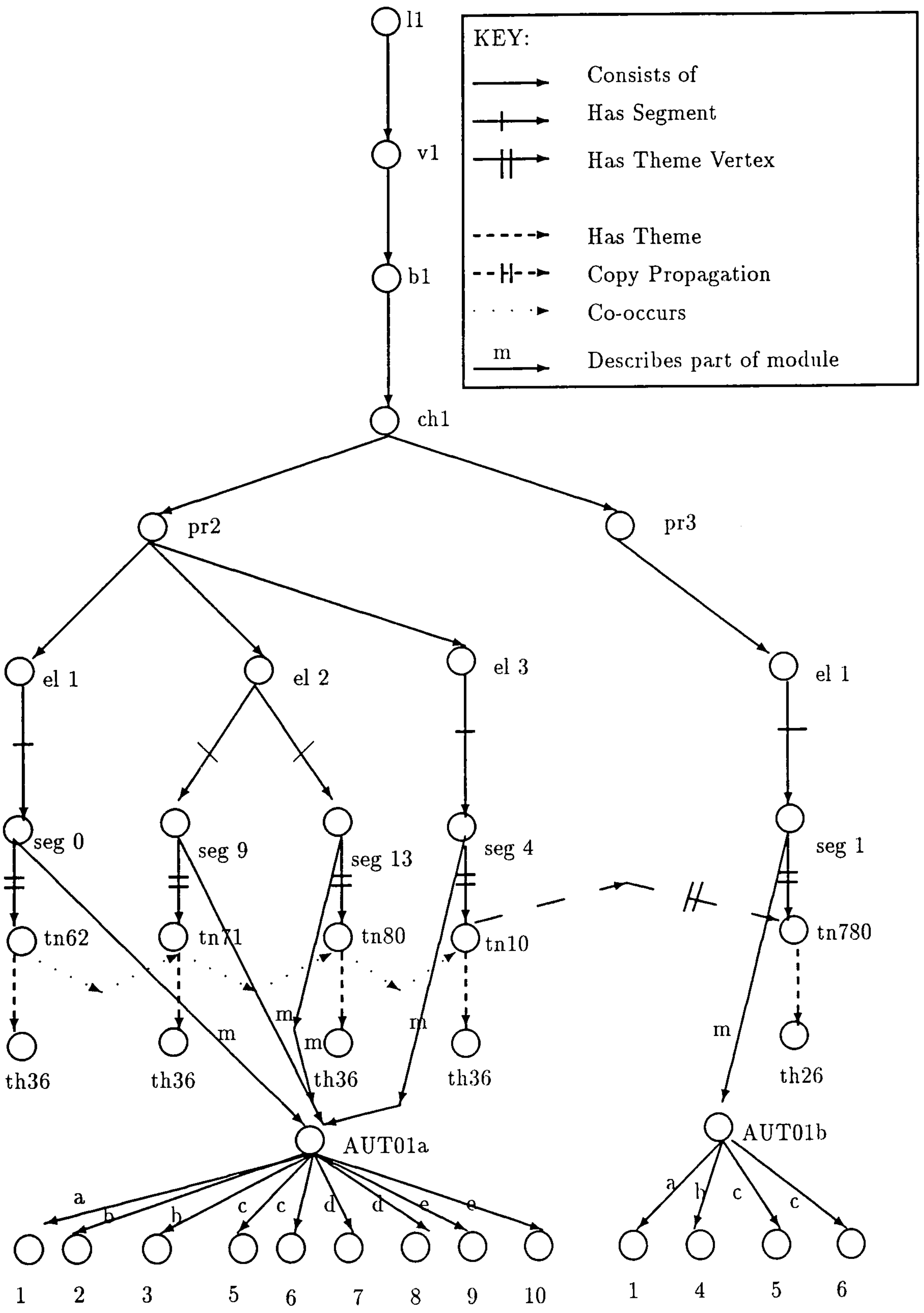


Figure 8.31: Case Study $\epsilon CIG(\mathcal{V}, \mathcal{E})$

KEY:

SAG Dependencies :

a module type

b module test file

c system test required

d module data file

e module map base

SAG Entities :

1 Cobol

2 Module test 1

3 Module test 3

4 Module test 4

5 End of Day 1

6 End of Day 2

7 Recycled Reconciliation Movements File

8 Audit Tapes File

9 Recycled Reconciliation Movements File Map Base

10 Audit Tapes File Map Base

Figure 8.32: Case Study $\epsilon CIG(\mathcal{V}, \mathcal{E})$ Key

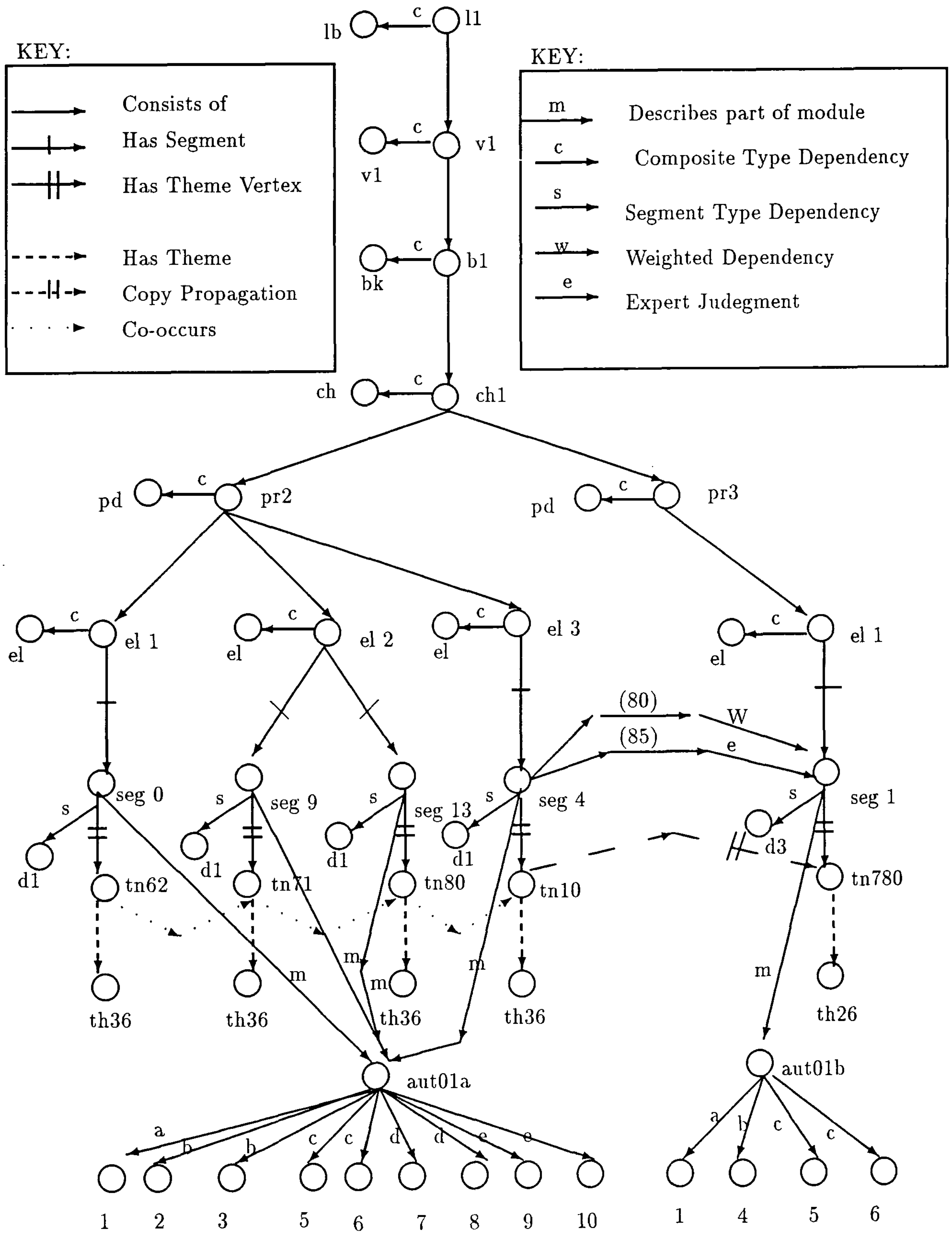


Figure 8.33: $CIG(\mathcal{V}, \mathcal{E})$ Splicing

This particular case study is actually linked to the rest of the core banking system through its input and output files, therefore a slice through the whole system could be very large indeed.

### 8.5.8 Interesting Additional Features

During this case study a number of other labels and dependencies were recorded in the model. These were the following :

- `Segment Describes Part of System` *Dependency*
  This dependency is associated with an edge of the form
  *(Segment Entity,System Entity)*
- `System Entity Supplies Data To` *Dependency*
  This dependency is associated with an edge of the form
  *(System Entity,System Entity)*
- `System Entity Receives Data from` *Dependency*
  This dependency is associated with an edge of the form
  *(System Entity,System Entity)*
- `Skill Base Entity Label`
  - Who is responsible for the system.
  - Who has maintained the system.
  - How much experience does a particular person have.

An example of this graph is shown figure 8.34. By recording the `Segment Describes Part of System` dependency it is possible to identify all system names which may be implicated by the change proposal. By recording the `System Supplies Data To` and `System Receives Data from` dependencies it is possible to investigate why particular system interfaces are impacted. The facility to add a label to any vertex within a particular $RPG(\mathcal{V},\mathcal{E})$, describing who is responsible for the particular work product is useful for the following reasons. It allows any key staff who are responsible for any system to be identified at the beginning of a project. It allows the forward planning of project meetings between key personnel.

Finally by recording who has maintained the system(s) impacted and by recording the experience these personnel have, it is possible to allocate maintainers with experience of particular systems to projects which affect these systems. This should reduce the cost of understanding existing systems.

Another interesting feature of the $RPG(\mathcal{V},\mathcal{E})$ is the facility to detect implicit dependencies in documentation, that is dependencies which are not based on the thematic structure of the documentation. For example, once a $CIG(\mathcal{V},\mathcal{E})$ has been extracted from an $RPG(\mathcal{V},\mathcal{E})$, the $CIG(\mathcal{V},\mathcal{E})$ can be further analysed in the following way :

1. Examine each segment entity in the $CIG(\mathcal{V},\mathcal{E})$.
2. Check if any segment entity has a weighted edge to another segment entity. The stop vertex of this weighted edge may be either in the $CIG(\mathcal{V},\mathcal{E})$ or in part of the $RPG(\mathcal{V},\mathcal{E})$ which is not impacted using the slicing techniques. Add each weighted edge to a set of impacted implicit edges.
3. Delete all edges from the impacted implicit edges set which have thematic dependencies joining the theme vertices associated with each start and stop vertex of edges in the impacted implicit edges set.

The remaining members of the impacted implicit edges set represent the implicit dependencies based on previous changes.

## 8.6 Summary

In this chapter the $RPG(\mathcal{V},\mathcal{E})$ construction and analysis techniques have been demonstrated using two examples of documentation structure and a major case study. It has been demonstrated how the $RPG(\mathcal{V},\mathcal{E})$ can be used at the change proposal stage of the maintenance process by extracting theme bags from change proposals. It has also be shown how the themes within the change proposal can be formally traced to operational software constructs. The techniques for constructing the

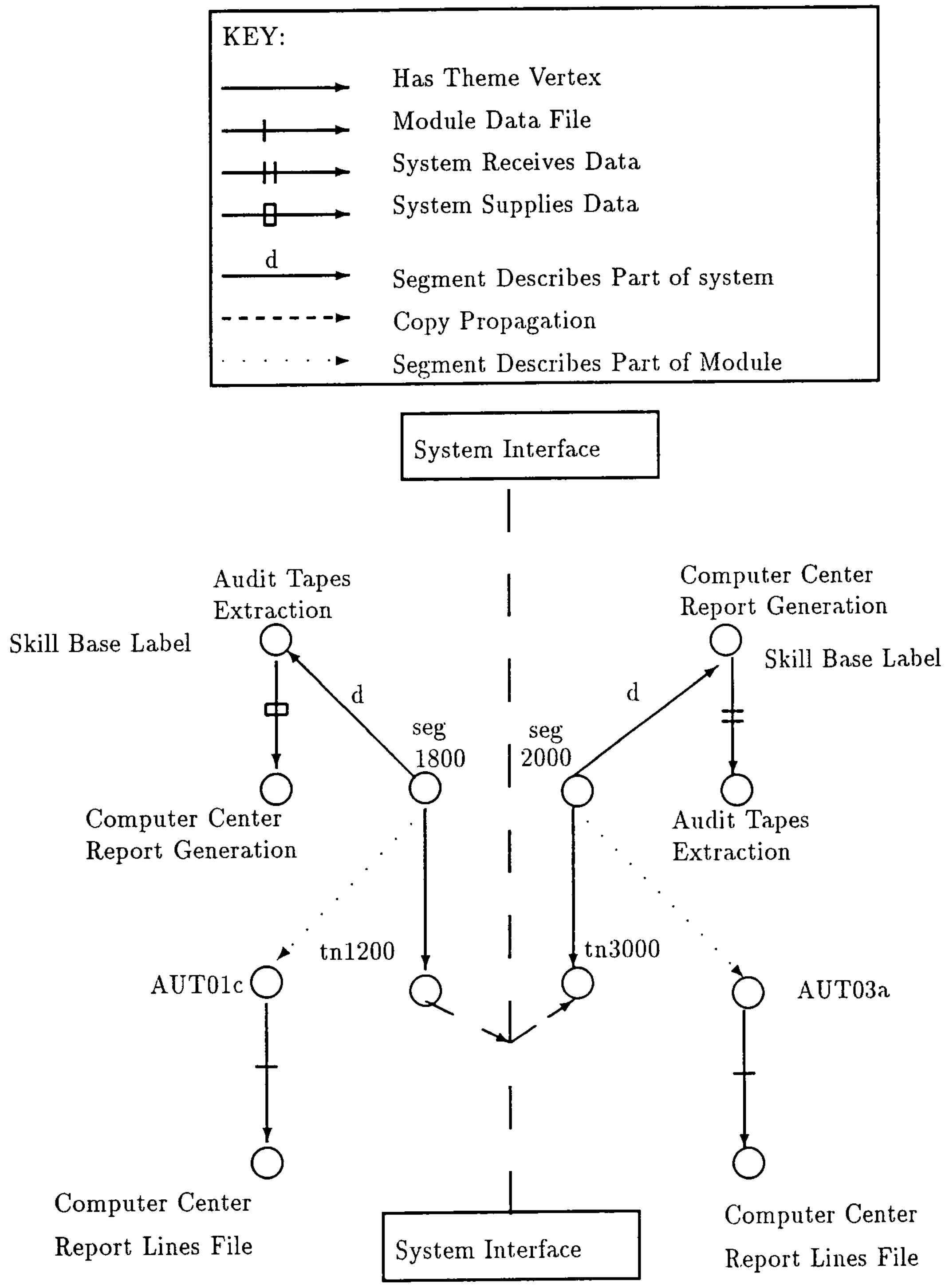


Figure 8.34: A System Interconnection Graph (SIG($\mathcal{V}, \mathcal{E}$))

$RPG(\mathcal{V},\mathcal{E})$ have been shown before demonstrating how the $RPG(\mathcal{V},\mathcal{E})$ can be sliced to provide impact information. The table 8.3 shows which particular thematic slices can be used to provide impact information to help a maintenance manager or maintainer make the decisions identified in Chapter 2.

| NO. | DECISION | TECHNIQUES |
|---|---|---|
| No. 1 | What are the cost benefits? | |
| No. 2 | What are the time scales required? | |
| No. 3 | What are the cost benefits? | |
| No. 4 | What are the inputs, outputs, processing processing and data? | $\alpha$ , $\beta$ , $\gamma$ , $\delta$ , $\epsilon$ |
| No. 5 | What is the implementation strategy? | $\delta$ $\epsilon$ |
| No. 6 | What is the testing strategy? | $\epsilon$ |
| No. 7 | What are the cost and benefits? | |
| No. 8 | How can the project be categorised? | |
| No. 9 | What is the priority for the project? | |
| No. 10 | Which are the impacted system features? | $\epsilon$ |
| No. 11 | What are the estimates now? | |
| No. 12 | Which technical documentation needs to be written/amended? | $\alpha$ , $\beta$ , $\gamma$ , $\delta$ , $\epsilon$ |
| No. 13 | Which test data is required? | $\epsilon$ |
| No. 14 | What is the source code impacted? | $\delta$ |
| No. 15 | What caused a particular defect? | |
| No. 16 | What caused a particular defect? | |

Table 8.3: Practical Application of the Graph Slices

A summary of the techniques demonstrated is presented is shown in table 8.4.

| Steps | Technique | Example 1 | Example 2 | Case Study |
|---|---|---|---|---|
| Step 1 | $HIG(\mathcal{V},\mathcal{E})$ Crystallisation | - | - | $\surd$ |
| Step 2 | $TIG(\mathcal{V},\mathcal{E})$ Crystallisation | - | - | $\surd$ |
| Step 2 | Theme Catalogue Construction | - | - | $\surd$ |
| Step 3 | $WIG(\mathcal{V},\mathcal{E})$ Parameterisation | - | $\surd$ | $\surd$ |
| Step 4 | $SAG(\mathcal{V},\mathcal{E})$ Annotation | - | $\surd$ | $\surd$ |
| Step 5 | $\mathcal{TB}$ Construction | - | $\surd$ | $\surd$ |
| Step 5 | $\alpha$ Slice | $\surd$ | $\surd$ | $\surd$ |
| Step 6 | $\beta$ Slice | - | $\surd$ | $\surd$ |
| Step 7 | $\gamma$ Slice | - | $\surd$ | $\surd$ |
| Step 8 | $\delta$ Slice | - | $\surd$ | $\surd$ |
| Step 9 | $\epsilon$ Slice | - | $\surd$ | $\surd$ |

Table 8.4: Techniques Demonstrated

The table represents the different steps which were applied to the examples and case study. The tests were designed to show the following :

1. The use of the four specialized subgraphs of the $RPG(\mathcal{V}, \mathcal{E})$.
2. The techniques for slicing documentation to produce impact information.
3. The use of impact information to help make decisions within the maintenance process.

In the next Chapter the results and evidence presented in this Chapter is used to evaluate the $RPG(\mathcal{V}, \mathcal{E})$, its associated analysis techniques, and in particular the value of the information produced by the analysis techniques in comparison with other impact analysis techniques.

# Chapter 9

# Evaluation of the Method

## 9.1 Introduction

In this chapter the method of this thesis is compared with other methods for the detection of the ripple effect of a change. The method is evaluated from the point of view of its feasibility, the extent to which the impact information can be trusted and in particular, how early in a project this impact information can be produced.

## 9.2 Evaluation Against the Criteria for Success

The evaluation of impact analysis is very problematic because it involves conducting controlled experiments to determine the precision of the method. Precision measures the size of the estimated impact of change. This can be used to determine the value of a method compared with another competing method, provided that they are used under the same experimental conditions. The conduction of an experiment would involve collecting data from many projects of a similar profile.

There is also another problem, which consists of identifying that which is relevant to a change, in addition to that which is actually impacted. For example a component of a system might need to be understood when making a change, yet it may not require actually changing. Depending on whether the maintenance programmer is familiar with a system this notion of relevance becomes subjective.

Ideally a large number of users should try to map change proposals into change proposal bags. Proposals affecting many parts of the system should be produced. This would mean producing a model of a very large system and would be impractical as an experiment within the time allocated for a Ph.D project. However the use of small subsystems as sources of data would inadvertently suggest specific themes to be used. This would probably bias any metrics for measuring the precision and relevance of the prediction.

As a consequence of the difficulty of finding realistic test conditions it was felt that any attempt to deduce experimental results would be too subjective to be of value in assessing the validity of this particular impact analysis model and analysis techniques.

The criteria for the success of the investigation have been achieved and are as follows :

1. The production of an interconnection graph usable by management.
2. The ability to trace user requirements to operational software.
3. The production of a method for building and maintaining the graph.
4. The early provision of impact information using the graph slicing methods.
5. The evaluation of the information produced by the model with respect to other impact analysis methods.

In examples of the method application it was shown how a bag of themes could be extracted from a change proposal in order to analyse the graph. If an $RPG(\mathcal{V}, \mathcal{E})$ has been constructed from information drawn from a number of projects and also if the change proposal affects this part of the system, then it will be possible to trace the change proposal to the operational software. Simple methods have been developed for constructing the graph namely the graph crystallisation, parameterisation and annotation. As it is possible to trace the change proposal to the operational software and also the graph contains an abstraction of the interconnectivity of the source code, then the early provision of impact information using the graph is possible, rather than having to wait until the coding phase for the detection of the ripple effect.

Instead of an experiment, an openly subjective argument is offered for the use of thematic structure, the use of release information and expert judgement. This can be discussed under the following headings :

- *Theory of structure:* The model provides an explanation of the organisation of the documentation of a system. It is not just a simple hierarchy but rather it explains the role of each document component using the segment types and composite entity types. These features provide a disposition of the document components which provide descriptions of the software. This model of documentation structure provides a greater understanding of the arrangement and purpose of documentation than does the existing book, chapter and section paradigm. This is a practical paradigm which is often used in industry [27].

- *Use of thematic structure:* The thematic structure provides a mapping between change proposals and operational software. The thematic structure is a more precise method of recoding the meaning of document segment entities than using keywords. Many document components can be impacted, some of which will need understanding but not maintaining where as other components will need both.

- *Use of weighted graph edges:* The use of the weighted edges on the graph provides information with which the maintainer may wish to classify impacted document entities and source code modules according to the chance of being impacted. For example if the maintainer discovers that one document entity will propagate a ripple effect to another and this would imply the maintenance and testing of another source code module. The chance of this happening can be examined. If a module has a 1% chance of being impacted, then this does not imply that the module is not relevant to a proposed maintenance project. It means that based on past experience this particular module probably does not require maintaining.

- *Use of expert judgement:* The recording of expert judgement is useful for the purpose of formally carrying forward expertise concerning interconnectivity and ripple propagation, from one project to another. It is useful to analyse the parametrised edges containing expert judgement probabilities and the reasons and assumptions behind these probabilities. This is because it tells the maintainer the actual chance of the ripple effect occurring rather than just providing the worst case estimate of the ripple effect, which is often much larger than the actual ripple effect. This may mean in some cases that certain maintenance projects would be rejected on technical grounds. Hence the value of using probability is that it allows maintainers to see that a strategic project with a large ripple effect which would normally be rejected, could be actually cost effective. Expert judgement data is not a substitute for release information but it is useful to use when there is a only small amount release information available or none at all.

• *Intuitive Evidence:* Most software processes data and most software shares data. It is this sharing and processing of data which causes the ripple effect. Documentation describes software in terms of what the software does and how the software achieves this. Therefore if a model of documentation is to be used to reason about the ripple effect, then such a model must capture the description of the sharing and processing of data.

## 9.3 Strengths and Weaknesses

The following strengths of the $RPG(\mathcal{V},\mathcal{E})$ approach are recognised.

The method is well founded in graph theory, bag theory and set theory. Also unlike other impact analysis methods, it is sensitive to previous ripple effects. This is achieved by recording the release information and expert judgement.

It is easy to extract themes incrementally when performing maintenance because the documentation must be understood in order to perform maintenance tasks anyway.

The model can be used at an earlier stage than most other impact analysis methods because the concept of a theme provides a good interface or protocol between change proposal and the $RPG(\mathcal{V},\mathcal{E})$ model.

The precision of the graph slicing methods, also the authenticity of the answers produced, are dependent on how well the themes are extracted from the documentation and then added to the $RPG(\mathcal{V},\mathcal{E})$.

This approach to impact analysis at the documentation level is based on source code principles, unlike existing methods for documentation analysis.

The $RPG(\mathcal{V},\mathcal{E})$ can provide the maintainer with a wealth of valuable information. For example it can indicate the inputs, outputs processing and interfaces impacted. The model also indicates the modules which support these features. The model indicates direct impacts and indirect impacts at the documentation level and the probabilities of these impacts. The model can also indicate the impacted skill base. No other document model provides this information. Other document models can identify impacted document based on keywords common to all documents.

The model provides the maintainer with explanations and justifications for the advice given about ripple effects. No other document model provides this to date.

By analysing the impacted weighted edges it is possible to detect implicit dependencies in the documentation.

The entity factoring provides a sound basis for interpreting results because change proposals can be compared in terms of the number of segment entities impacted. This is because the same unit of analysis is used to describe the change for different change proposals.

By using this $RPG(\mathcal{V},\mathcal{E})$ model it is possible to identify the software system structure affected. If a conceptual model is examined at an early stage, it will only indicate features of a system at the domain level and not the architectural structure of a system.

The method is a feasible approach since it feeds back information produced as a by-product of software maintenance. It does not need a new development paradigm to support the model.

The following weaknesses of the $RPG(\mathcal{V},\mathcal{E})$ approach are recognised.

A maintainer presented with a large amount of documentation may find it difficult to obtain an overview of the architecture of documentation even when it is structured like a program. Therefore an abstract $HIG(\mathcal{V},\mathcal{E})$ needs to be drawn first. This is required for orientation.

The modelling of documentation from the perspective of description of data flow within documentation, can be problematic since algorithms are only contained in low levels of abstraction in documentation. The extraction of thematic properties from documentation can lead to property distortion, which is the misunderstanding of the actual meaning of the documentation. Theme detection is currently human oriented and open to interpretation. Themes are not conducive to precise definition. This makes it difficult for an automated tool to extract the meaning of a document segment. Another problem associated with understanding documentation is the requirement of domain knowledge which may not be available to a maintainer constructing the graph model.

Theme extraction can be very tedious to do manually. If the graph is built up incrementally during maintenance then the model will only describe the parts of the system which have previously been maintained.

Documentation does not always reflect the semantics of the source code. Therefore basing a model of source code interconnectivity on the information derived from documentation could lead to false conclusions about the effects of hypothesized changes.

The use of previous ripple effect information has been shown to be useful for understanding potential ripple propagation. However different types of maintenance work can make different types of impact. For example during the life time of a system it may undergo corrective maintenance when it is first released but may later undergo perfective maintenance. The model does not take this into account and therefore false conclusions could be reached concerning the likelihood of a ripple effect. Perhaps the reasons for ripple effects should be recorded as part of the documentation.

There are problems in using expert judgement to understand the potential ripple effect of a project. For example, expert judgements are not equivalent to technical calculations based on laws or the availability of extensive data, therefore they may be incorrect. Judgements in the form of probabilities represent a snapshot at a point in time of a given state of knowledge of a particular expert. Therefore they may not necessarily be applicable to future maintenance projects.

The probabilities of ripple effects must be interpreted with care. For example each time a module propagates a ripple effect to another module it is possible that the structure and content of the module may change. A drastic change in structure and content of one module may alter the probability of ripple propagation to other modules and document entities. This is why it is very important to update the expert judgement weightings in order to ensure that misinterpretations of release information are minimized.

The limitation of using graph theory to model the content and interconnectivity of documentation is that the resultant models can be very large indeed.

There are a number of issues which have arisen during this research work. For example the interpretation of the results of graph analysis can be particularly difficult when using the slicing methods based on probabilities of ripple effects. There is the question of at which level of probability maintenance managers should focus.

A balance has to be struck between the need for meaningful representation, partial detection and ease of use whilst at the same time considering the feasibility of the method. A method of coding segments which is based on formal semantics provides a meaningful representation but suffers in other respects. The use of keywords to characterise segments provides ease of use but lacks meaningful representation. It is argued that the use of the theme as a method of capturing the role of a document segment entity provides sufficient information to achieve an understanding of the ripple effect and strikes the balance required. A ripple effect analysis method which is based on describing segment entity contents using themes and describing change proposals as a bag of themes, has a better capability of matching the change proposal to the documentation than one which is based on keywords.

## 9.4 Comparison with Other Methods

The extraction of the themes from change proposals and the recording of themes in documentation provide a mapping between change proposals and operational software. This is not possible with existing ripple effect methods. The existing syntactic, semantic and statistical methods are aimed at source code and do not provide ripple effect analysis of documentation. The method developed in this work is the only method which captures expert judgement explicitly and then formally carries it over to future maintenance projects. The method developed is the only method which is based on a model of documentation.

It is argued that the thematic slicing of $RPG(\mathcal{V}, \mathcal{E})$ models can be conducted at phase 1 state 4 of the maintenance model presented in Chapter 2, when decision 4 needs to be made. The teleological approach described in Section 3.4.4, can also be used at this same phase however it does not indicate the documentation to be maintained and neither does it indicate the probable impact of a change but only the worst case. It is thought that both $RPG(\mathcal{V}, \mathcal{E})$ and the teleological approach can be complementary.

If ripple effect methods are not used at all, then large amounts of contingency money will be tied up with projects. This may imply that certain projects which have good business cases are not ever committed. On the other hand maintenance projects often fail to meet deadlines and cost targets, because of the under-estimated resources required for dealing with the ripple effects of a change.

## 9.5 Interesting Observations

An interesting feature of the graph model developed is that it can be analysed with respect to segment entities. Some of these are dependent upon each other because themes co-occur in these segments or, the segments describe copy propagations. It is possible to observe segment entities which are independent of other segment entities. These segments could be called solitary segment entities. It may be that such entities are actually connected, however the model does not reflect this. Such segment entities could cause unexpected linkages.

One observation made about the size of slice trajectories is that they can be very large for any software systems bigger than toy examples. As each segment has a particular type then it may be possible to analyse the slice in terms of the type of segments which it contains. This is useful as some segments will describe data which is being output. Such segments can be removed from the slice as these segments do not describe ripple effects. Only segments describing assignments contribute to the ripple effect. For example the description of a subroutine, procedure or module containing output statements such as COBOL WRITE statements does not cause ripple effects. This idea of analysing slices originates from the doctoral thesis of Gallagher [33] describing decompositional slicing where source code slices are reduced to contain only assignment statements, that is, they are output restricted. This concept could be applied to the $RPG(\mathcal{V},\mathcal{E})$ slices to make the slices smaller and more manageable.

Another feature of the $RPG(\mathcal{V},\mathcal{E})$ which was observed, is that after the graph has been built up it is possible to analyse the interconnectivity of the entire system rather than just following a particular chain of graph edges.

Another interesting feature of the $RPG(\mathcal{V},\mathcal{E})$ not directly related to ripple effect analysis is that it may give rise to the production of a much better inventory of technical documentation than documentation systems which are simply based on keyword interconnections.

Using the $RPG(\mathcal{V},\mathcal{E})$ with large amounts of release information fed back into the $RPG(\mathcal{V},\mathcal{E})$ edges would enable the maintenance manager to determine which are the fundamental themes in the documentation. A fundamental theme is a theme which plays a critical role in a documentation system. For example it would be possible to observe which themes in a change proposal cause excessive amounts of ripple effects.

## 9.6 Summary

In this chapter the information which can be derived from the analysis of the ripple propagation graph is discussed with respect to its value and utility. The strengths and weaknesses of the ripple propagation and the graph analysis methods presented are also discussed. Finally the method presented is compared with other methods for the detection of the ripple effect of a change.

# Chapter 10

# Conclusions

## 10.1 The Main Achievements of the Research

The main achievement and result of this research is a ripple effect detection method. The ripple effect method developed in this thesis consists of analysing a new type of interconnection graph, called a **Ripple Propagation Graph**. The analysis techniques for reasoning and understanding interconnections in documentation are collectively called **Thematic Slicing** and are based on abstracting subgraphs from the Ripple Propagation Graph. These subgraphs are called **Change Implication Graphs** and represent the impact of a proposed change.

## 10.2 General Conclusions of the Research

There are a number of conclusions which can be inferred from the results of this investigation presented in this thesis :

- Impact analysis techniques provide information which is necessary to make decisions regarding the scope, priority, costs and testing involved in maintenance projects.

- Impact analysis techniques can be classified according to the stage in which the techniques can be used in the maintenance process and existing impact analysis techniques are most useful at a later stage in the maintenance process after important decisions regarding the scope, priority, costs and testing involved in maintenance projects have been made.
- The subject of impact analysis has been advanced because the graph model developed in this thesis can guide maintenance managers and programmers in tracing the ripple effects of a change at an earlier stage in the maintenance process than existing methods allow.
- A thematic graph slicing method for analysing documentation interconnections has been produced. The thematic graph slicing method is able to trace the contents of change proposals to operational software by analysing the thematic structure of documentation. This method of slicing has demonstrated that higher level objects than source code can be sliced.
- The hierarchical interconnection graph allows maintenance managers or maintainers to analyse the decompositional structure of documentation without navigating through large amounts of interconnected documentation.
- The thematic interconnection graph allows the maintenance programmer to determine the role a particular document entity plays in a documentation system and enables the searching for possible impacts of proposed changes.
- The source attributes graph allows the maintainer to determine source code entities described by particular documentation entity without having to read the document entity itself.
- The weighted interconnection graph models the previous dynamic behaviour of the system at the documentation level. This allows the maintainer to analyse the probable source code and documentation ripple effects of a proposed change based on change history and expert judgement concerning potential ripple effects. The weighted interconnection graph also allows the resultant thematic slices to be reduced in size. This can be achieved by eliminating weighted edges which are not with a specified range of probabilities.
- The weighted interconnection graph also allows the detection of implicit and unforseen dependencies.
- The use of bags for storing impacted graph edges and graph vertices enables the detection of multiple impacts. The provides a more precise analysis of the impact of a proposed change than simply examining the change implication graphs.

## 10.3 Relationship with the Wider Field

The work presented in this thesis has links with other software engineering topics.

The $RPG(\mathcal{V}, \mathcal{E})$ may help maintenance managers in improving the management of projects, since more impact information will be known at the beginning of a project. This information will also have some bearing on setting project milestones, cost estimation and scheduling.

It may be possible to compare the thematic structure with the actual structure of the source code data flow to determine whether the documentation is consistent with the source code. It may also be possible to determine which parts of the documentation may need restructuring, by considering the thematic interconnections in those areas of documentation which change frequently.

The $RPG(\mathcal{V}, \mathcal{E})$ will facilitate the production of new kinds of hypotheses concerning structural features of documentation and may provide clarification of concepts such as 'documentation stability' , 'documentation maintainability' and 'documentation change complexity'. The analysis of the graph model will also allow the production of impact analysis metrics.

The information produced by the $RPG(\mathcal{V}, \mathcal{E})$ analysis methods will help a configuration manager to ensure that changes are incorporated in a controlled way, providing an improved understanding of the work involved in processing a system proposal.

## 10.4 Suggestions for Future Research

### 10.4.1 Improved RPG Impact Analysis

$RPG(\mathcal{V}, \mathcal{E})$ slices may contain much information for analysis by the maintainer. Hence it would seem worthwhile investigating how a slice could be viewed in a more abstract way. It would be a very useful aid to the maintainer, in program understanding, to present the graph model developed in this work. However graph models of real world systems can be very large, so a practical method of presenting the abstract structure of the model would be useful. A more efficient method of coding the segments of the $HIG(\mathcal{V}, \mathcal{E})$ with themes, needs to be investigated. In particular a method which lends itself to automation would be beneficial.

### 10.4.2 Clustering Proposed Changes using Impact Information

It would be useful to develop a method for clustering together groups of change proposals which impact similar areas of a system to form software releases. It would also be most useful to extend such a clustering method to plan software releases which are of particular sizes.

### 10.4.3 A Unified Impact Analysis Model

Several models for documentation have been developed, for example [93, 17, 48]. Each of these models argues a strong case for the value of the model. However nobody has yet investigated the value of a unified model containing all of these models in one single model. The results of the investigation could also be added to the $RPG(\mathcal{V}, \mathcal{E})$ model.

### 10.4.4 Impact Analysis on Real Time Software Systems

Real time software systems design has not been considered from the perspective of ripple effect analysis. Real time software systems are systems where the correct functioning of the system depends on both the results produced by the system and the time at which the results are produced. Real time software systems are frequently very large. Hence coping with change and complexity during software maintenance is problematic for two main reasons :

1. Interfaces to environments are complex, asynchronous, highly parallel and distributed, all of which makes detection of the ripple effect difficult.
2. Performance of other related software sub-systems can be affected by minor ripple effects in the source code. Often these ripple effects are unforeseen at an early stage of a maintenance project.

### 10.4.5 Impact Analysis on Graphical Notations

Another possible direction for future research is the maintenance of graphical notations, which have been produced by software engineering tools. There are no methods for impact analysis on graphical notations.

### 10.4.6 A Process Model for Change Analysis

The model of the change analysis process developed in this thesis served as a means to rank existing impact analysis methods according to when they may be applied in the process. However it would very useful to develop the model into a process model to facilitate the improvement of the change analysis process.

For example one important use of the process model would be the identification of independent tasks and parallel maintenance processes. This would reduce the lead time between a request for change and the delivery of the new system.

## 10.5 Summary

The contribution of this work is a method for the detection of the ripple effect of a change using a model of documentation interconnectivity. This chapter has presented the main achievements of the research, the general conclusions, the relationship with the wider field and some suggestions for future research.

# Appendix A

# Glossary of Notation

**Author's Notation and Abbreviations.**

Words in boldface may be found in Appendix B, the Glossary of Terminology.

$\underline{\triangle}$ Denotes is defined to be.

$\mathcal{BE}$ Denotes a bag of impacted entities.

$\mathcal{BV}$ Denotes a bag of impacted vertices.

CA Denotes a **Conceptual Action.**

$CIG(\mathcal{V}, \mathcal{E})$ Denotes a **Change Implication Graph.**

CIR Denotes a **Change Implication Report.**

CO Denotes a **Conceptual Object.**

$(\mathcal{V}', \mathcal{E}')$ Denotes an extracted subgraph from an $RPG(\mathcal{V}, \mathcal{E})$

$HIG(\mathcal{V}, \mathcal{E})$ Denotes a **Hierarchical Interconnection Graph.**

$SAG(\mathcal{V}, \mathcal{E})$ Denotes a **Source Attributes Graph.**

$TIG(\mathcal{V}, \mathcal{E})$ Denotes a **Thematic Interconnection Graph.**

$\mathcal{TB}$ Denotes a **Theme Bag.**

$WIG(\mathcal{V}, \mathcal{E})$ Denotes a **Weighted Interconnection Graph.**

$RPG(\mathcal{V}, \mathcal{E})$ Denotes a **Ripple Propagation Graph.**

# Appendix B

# Glossary of Terminology

This glossary defines many technical terms as they are used in the text. Whenever understanding can benefit from referring to other terms in the glossary, the corresponding glossary entries are noted in boldface. For simplicity, formal definitions are avoided in favour of informal explanations.

**Annotated Graph Slicing**

A method of slicing graph theory models of documentation based on analysing the thematic structure within a document hierarchy. In particular with respect to **indirect impacts**. In addition the source code **entities** described by impacted segment entities is also included in the resultant slice.

**Augmented Graph Slicing**

A method of slicing graph theory models of documentation based on analysing the thematic structure within a document hierarchy. In particular with respect to **indirect impacts**. In addition the probability of ripple effects between segment entities, based on previous ripple effects, and expert judgements is included in the resultant slice.

**Change Implication Graph**

A graph describing the impact of a change. The graph describing the impact of a proposal change is a subgraph of a **Ripple Propagation Graph**.

**Change Implication Report**

A report describing a **change implication graph**.

**Change Proposal**

A form which triggers the process of maintenance. The form contains a request for software maintenance.

**Complete Thematic Slicing**

A method of slicing graph theory models of documentation based on analysing the thematic structure within a document hierarchy. In particular with respect to **indirect impacts**.

**Composite Entity**

Any entity in a document hierarchy which is not a **segment entity**. A composite entity is composed of other entities.

**Composite Entity Factoring**

The decomposition of composite entities into **segment entities**.

**Composite Entity Slice**

A subgraph containing all the vertices and edges between a **segment entity** and the top of a **Hierarchical Interconnection Graph**.

**Conceptual Action**

The description of a process acting on a data item.

**Conceptual Object**

The description of a data item.

**Direct Edge Parameterisation**

The addition of weights to edges based on previous ripple effects.

**Direct Impact**

An entity affected by a change proposal. The entity being explicitly mentioned in change proposal.

**Documentation Stability**

The analysis potential **ripple effect** propagated to all material that serves primarily to describe a computer program.

**Edge**

An ordered pair of vertices denoting a dependency between the **start vertex** and **stop vertex**.

**Edge Parameterisation**

The addition of weights to edges.

**Entity**

An entity is anything which can be named or denoted within a document.

**Entity Attributes**

Details about the characteristics of an entity.

**Entity Type**

The class or group of entities in which an **entity** belongs.

**Graph Annotation**

Firstly, the process of analysing documentation with respect to the source code entities described with the documentation and the construction of the corresponding Source Attributes Graph.

**Graph Clipping**

The process of deleting vertices and edges from a graph.

**Graph Parameterisation**

The process of adding weighted edges between segment entities. The weighted edges represent probabilities of ripple effects between segments. (See the other glossary entries **edge parameterisation**, **direct edge parameterisation** and **indirect edge parameterisation.**)

**Graph Splicing**

The process of combining two graphs together.

**HIG Crystallisation**

The process of analysing documentation with respect to its structural decomposition and the construction of the HIG to model this structural decomposition.

**Hierarchical Interconnection Graph**

A subgraph of the Ripple Propagation Graph. The Hierarchical Interconnection Graph records composite entities, segment entities and their structural dependencies.

**Impact Analysis**

The study of the consequential effects of changing entities on other entities.

**Impact Points**

The set of directly affected document segment entities. (See also **Direct Impact**)

**Indirect Edge Parameterisation**

The addition of weights to edges based on expert judgements concerning potential ripple effects.

**Indirect Impact**

The consequential effect on an entity caused by a change to another entity.

**Interconnection Graph**

An interconnection graph is a graph that is used to represent the dependencies between entities in a program, and a piece of the program documentations.

**Logical Ripple Effect**

An inconsistency introduced into a program area by a change to another program area [95].

## Module

A module is a named collection of source code entities, where the programmer has precise control over the entities that are imported from and exported to the surrounding environment. A module can contain **routines**.

## Performance Ripple Effect

The **performance ripple effect** which is a change in a module's performance as a consequence of a software change in another module.

## Release Information

Information which describes which modules and documents were changed in a project or group of projects. This information also describes any **ripple effects** involved with a particular project.

## Ripple Effect

Ripple effect is the phenomenon by which changes to one program area have tendencies to be felt in other program areas [95].

## Ripple Effect Analysis

To break down the complete **ripple effect** of a proposed change into its component parts.

## Ripple Propagation

The spreading of the **ripple effect**.

**Ripple Propagation Graph**

A graph theory model for recording the entities within a document hierarchy and their dependencies. The dependencies help model possible **ripple propagation** and previous ripple propagation in a document hierarchy.

**Ripple Propagation Graph Description Language**

A language for describing the Ripple Propagation Graph.

**Routine**

A sub-program unit that could be either a procedure or a function.

**Segment Entity**

The smallest type of entity within a document hierarchy

**Segment Scanning**

The process of examining a segment entity to determine the source code entities described within it.

**Segment Theme Analysis**

The process of examining a segment entity to determine the contained within it.

**Slice Criterion**

The criteria with which to judge whether or not a particular entity appears in a slice.

**Source Attributes Graph**

The Source Attributes Graph is a subgraph of the **Ripple Propagation Graph** and records all source code entities described implicitly or explicitly in segment entities.

**Stability**

The sum of the potential ripple effects that may be propagated within source code.

**Stability Analysis**

The analysis of sum of the potential ripple effects that may be propagated in a software system.

**Start Vertex**

The **vertex** that marks the starting point for an **edge**.

**Stop Vertex**

The **vertex** that marks the terminating point for an **edge**.

**Thematic Graph Slicing**

A method of slicing **Ripple Propagation Graph**s models of documentation based on analysing the thematic structure within a document hierarchy.

**Thematic Interconnection Graph**

A subgraph of the **Ripple Propagation Graph** which records a model of the themes contained within a document hierarchy.

**Thematic Structure**

The organisation of the content of documentation, that is the organisation of the themes. This is modelled with a **Thematic Interconnection Graph**.

**Theme**

A theme in linguistical terms is a subject written about [30]. In this thesis a theme consists of a **Conceptual Object** and **Conceptual Action**.

**Theme Bag**

A theme bag contains the themes within a change proposal. Some themes may occur more than once.

**TIG Crystallisation**

The process of analysing documentation with respect to its **thematic structure** and the construction of the TIG to model the thematic structure.

**Vertex**

The objects that comprise a tree or a graph. Objects can be a start vertex or a stop vertex

**Weighted Graph Slicing**

A method of slicing graph theory models of documentation based on analysing the thematic structure within a document hierarchy. In particular with respect to **indirect impacts**. In addition the probability of ripple effects between segment entities, based on previous ripple effects is included in the resultant slice.

**Weighted Interconnection Graph**

The Weighted Interconnection Graph is a subgraph of the **Ripple Propagation Graph**. This subgraph records the probable ripple effects between segment entities. These probabilities are based on previous release information and expert judgements about the probability of ripple effects.

# Appendix C

# Prototype Demonstration

This appendix demonstrates the use of the RPGDL, a theme catalogue, the description of change proposals and an actual example of the consultation with the prototype implementation. The example given in this appendix shows only one of the reports produced by the implementation. This is because the reports are long and mostly the same format. From the report the impacted bags $\mathcal{BV}$ and $\mathcal{BE}$ and the $\beta$ $CIG$ for example 2 (presented in Chapter 8) can be constructed. The purpose of this appendix is to support Chapter 7 and Chapter 8 by demonstrating how the prototype implementation of MAGENTA was used to help evaluate the $RPG(\mathcal{V}, \mathcal{E})$ and its associated thematic graph slicing techniques.

## C.1 RPG Description Language: An Example

This section presents an example of the use of Ripple Propagation Graph Description Language in describing the $RPG(\mathcal{V}, \mathcal{E})$. Example 2 was chosen because it is concise and it exercises all of the descriptive power of the graph description language. This graph description example is the verbatim RPGDL script for example 2.

```
rpg(banking-sub-system-1).
consists-of(l1,v1).
consists-of(l1,v2).
consists-of(v2,b1).
consists-of(b1,c1).
consists-of(b1,c2).
consists-of(c1,s1).
consists-of(c1,s2).
consists-of(c1,s3).
consists-of(c2,s4).
consists-of(c2,s5).
consists-of(s2,sb1).
consists-of(s2,sb2).
consists-of(s5,sb3).
consists-of(s5,sb4).
has-segment(sb2,g1).
has-segment(sb2,g2).
has-segment(sb2,g3).
has-segment(sb4,g4).
has-segment(sb4,g5).
has-segment(sb4,g6).
has-theme-vertex(g1,t1).
has-theme-vertex(g2,t2).
has-theme-vertex(g2,t3).
has-theme-vertex(g3,t4).
has-theme-vertex(g3,t5).
has-theme-vertex(g4,t6).
has-theme-vertex(g5,t7).
has-theme-vertex(g6,t8).
has-theme-vertex(g6,t9).
has-theme(t1,th1).
```

```
has-theme(t2,th1).
has-theme(t3,th4).
has-theme(t4,th18).
has-theme(t5,th20).
has-theme(t6,th4).
has-theme(t7,th32).
has-theme(t8,th5).
has-theme(t8,th9).
co-occurs(t1,t2).
co-occurs(t3,t6).
copy-propagation-description(t2,t3).
copy-propagation-description(t3,t4).
copy-propagation-description(t6,t7).
copy-propagation-description(t7,t9).
copy-propagation-description(t9,t8).
copy-propagation-description(t8,t5).
document-potential-impact-dependency(g2,g2,75).
document-potential-impact-dependency(g2,g3,25).
document-potential-impact-dependency(g2,g4,75).
document-potential-impact-dependency(g4,g5,50).
document-potential-impact-dependency(g6,g3,100).
definition-use-description-chain(t1,t2).
definition-use-description-chain(t3,t6).
thematic-context(t1,definition).
thematic-context(t2,use).
thematic-context(t3,definition).
thematic-context(t4,null).
thematic-context(t5,null).
thematic-context(t6,use).
thematic-context(t7,use).
thematic-context(t8,null).
thematic-context(t9,null).
composite-entity-type(l1,l).
```

```
composite-entity-type(v1,v).
composite-entity-type(v2,v).
composite-entity-type(b1,b).
composite-entity-type(c1,c).
composite-entity-type(c2,c).
composite-entity-type(s1,s).
composite-entity-type(s2,s).
composite-entity-type(s3,s).
composite-entity-type(s5,s).
composite-entity-type(sb1,ss).
composite-entity-type(sb2,ss).
composite-entity-type(sb3,ss).
composite-entity-type(sb4,ss).
segment-entity-type(g1,d1).
segment-entity-type(g2,d2).
segment-entity-type(g3,d1).
segment-entity-type(g4,d3).
segment-entity-type(g5,d3 ).
segment-uses-data-dictionary-region(g1,region-1).
segment-uses-data-dictionary-region(g2,region-1).
segment-uses-data-dictionary-region(g3,region-1).
segment-uses-data-dictionary-region(g4,region-1).
segment-uses-data-dictionary-region(g5,region-1).
segment-uses-data-dictionary-region(g6,region-1).
segment-describes-part-of-module(g1,module1).
segment-describes-part-of-module(g1,module2).
segment-describes-part-of-module(g2,module1).
segment-describes-part-of-module(g3,module1).
segment-describes-part-of-module(g4,module2).
segment-describes-part-of-module(g5,module2).
segment-describes-part-of-module(g6,module2).
module-belongs-to-system(module1,end-of-day-run).
module-belongs-to-system(module2,end-of-day-run).
```

```
module-type(module1,cobol).

module-type(module1,assembler).

module-type(module2,cobol).

module-test-required(module1,test-mod-1).

module-test-required(module1,test-mod-3).

module-test-required(module1,test-mod-4).

module-test-required(module2,test-mod-2).

system-test-required(module1,system-test-1).

system-test-required(module3,system-test-16).

system-test-required(module2,system-test-1).

module-uses-data-file(module1,batch-file).

module-uses-data-file(module2,batch-file).

module-uses-job-control-language(module1,jcl-1).

module-uses-job-control-language(module2,jcl-1).

module-uses-map-base(module1,map-1).

module-uses-map-base(module2,map-1).

associated-system(end-of-day-run,accounts-a).

associated-system(end-of-day-run,accounts-b).

associated-system(end-of-day-run,creditors-run).

module-expert-judgement(1,module1,module2,1.1,10,not-likely).

module-expert-judgement(2,module1,module2,1.1,15,probably-not).

module-expert-judgement(3,module1,module2,1.2,20,fairly-unlikely).

judgement-reason(1,share-little-data,null).

judgement-reason(2,only-one-file-used,null).

judgement-reason(3,not-much-interconnection,null).

judgement-person(1,turver,april111990).

judgement-person(2,turver,december121990).

judgement-person(3,munro,march211991).

composite-entity-description-property(l,library).

composite-entity-description-property(v,volume).

composite-entity-description-property(b,book).

composite-entity-description-property(c,chapter).

composite-entity-description-property(s,section).
```

```
composite-entity-description-property(ss,sub-section).
segment-entity-type-description(d1,description-of-input).
segment-entity-type-description(d2,description-of-output).
segment-entity-type-description(d3,description-of-processing).
segment-entity-type-description(d4,description-of-an-interface).
segment-entity-type-description(d5,description-of-data-flow).
segment-entity-type-description(d6,description-of-data-store).
segment-entity-type-description(d7,description-of-human-factors).
segment-entity-type-description(d8,description-of-hardware).
segment-entity-type-description(d9,description-of-other-system).
segment-entity-type-description(d10,description-of-other-document).
```

## C.2 Theme Catalogue Description: An Example

The following shows the theme catalogue developed for example 2 in Chapter 8. Each line in the catalogue records a theme. For example :

```
theme(th1,describes-processing-of-a1).
```

indicates that theme code "th1" represents the theme "-processing-of-a1". This catalogue of themes enables change proposals to be mapped to the "describes-processing-of" part of a line in the theme catalogue. This allows a change proposal to be modelled with a set or bag of theme codes. The theme codes in each table are also recorded on an $RPG(\mathcal{V}, \mathcal{E})$.

```
theme(th1,describes-processing-of-a1).
theme(th2,describes-processing-of-b1).
theme(th3,describes-processing-of-c1).
theme(th4,describes-processing-of-d1).
theme(th5,describes-processing-of-e1).
theme(th6,describes-processing-of-f1).
theme(th7,describes-processing-of-g1).
theme(th8,describes-processing-of-h1).
theme(th9,describes-processing-of-i1).
theme(th10,describes-processing-of-j1).
theme(th11,describes-processing-of-k1).
theme(th12,describes-processing-of-l1).
theme(th13,describes-processing-of-m1).
theme(th14,describes-processing-of-n1).
theme(th15,describes-processing-of-o1).
theme(th16,describes-processing-of-p1).
theme(th17,describes-processing-of-q1).
theme(th18,describes-processing-of-r1).
theme(th19,describes-processing-of-s1).
theme(th20,describes-processing-of-t1).
theme(th21,describes-processing-of-u1).
theme(th22,describes-processing-of-v1).
theme(th23,describes-processing-of-w1).
theme(th24,describes-processing-of-x1).
theme(th25,describes-processing-of-y1).
theme(th26,describes-processing-of-z1).
theme(th27,describes-processing-of-a2).
theme(th28,describes-processing-of-b2).
theme(th29,describes-processing-of-c2).
theme(th30,describes-processing-of-d2).
theme(th31,describes-processing-of-e2).
theme(th32,describes-processing-of-f2).
theme(th33,describes-processing-of-g2).
theme(th34,describes-processing-of-h2).
theme(th35,describes-processing-of-i2).
theme(th36,describes-processing-of-j2).
theme(th37,describes-processing-of-k2).
theme(th38,describes-processing-of-l2).
theme(th39,describes-processing-of-m2).
theme(th40,describes-processing-of-n2).
theme(th41,describes-processing-of-O2).
theme(th42,describes-processing-of-p2).
theme(th43,describes-processing-of-q2).
theme(th44,describes-processing-of-r2).
theme(th45,describes-processing-of-s2).
theme(th46,describes-processing-of-t2).
theme(th47,describes-processing-of-u2).
theme(th48,describes-processing-of-v2).
theme(th49,describes-processing-of-w2).
theme(th50,describes-processing-of-x2).
```

## C.3 Change Proposal Description: An Example

The following shows two change proposals described in the RPGDL. The first two lines "proposal-identification" identify the proposal number, project number and release number respectively. Lines three and four indicate the origin of the particular proposal. For example each "source-of-change" line describes the proposal number, division, department, name of originator and date of proposal. Each change proposal has one or more lines forming a list of the themes contained within each proposal.

```
proposal-identification(1,2,2.1).
proposal-identification(2,2,2.2).
source-of-change(1,secs,cs,s-carter,jan041992).
source-of-change(2,secs,cs,s-carter,jan071992).
change-proposal(1,th1).
change-proposal(1,th4).
change-proposal(1,th32).
change-proposal(2,th5).
change-proposal(2,th6).
```

## C.4 Prototype Consultation: An Example

This section of the appendix presents an annotated version of a verbatim report produced by the prototype implementation for example 2. This particular example of system structure and content under consideration is theoretical, although an attempt has been made to make it realistic.

This consultation describes the prototype implementation's $\beta$ slice analysis of the example 2 discussed in Chapter 8. For further clarity, explanatory notes have been added throughout the consultation report. Computer messages of no consequence have been deleted and blank space has been added or deleted to improve readability. Otherwise the report is a facsimile of the prototype implementation's operation as seen on a computer terminal.

---

|| **NOTE:** Explanatory notes (such as this one) appear at appropriate places throughout the consultation report. For clarity they are always marked by double bar "|| **NOTE:**" on the left margin.

User input is highlighted in underscore *italic characters*.

```
Prototype implementation output is denoted by simulated typed text.

Edinburgh Prolog version 1.5.04 (12 September 1988)
AI Applications Institute, University of Edinburgh
```

|| **NOTE:** This command loads into MAGENTA a file called rpg2 which contains the RPGDL for example 2 and the graph slicing algorithms.

```
| ?-
```

*[rpg2].*

```
yes
```

```
| ?-
```

*b(1).*

|| **NOTE:** This command "b(1)" executes the beta slicing algorithm on proposal number 1.

```
                         MAGENTA

Durham RPG Analyser Prototype Version 3.0 (March 31st 1993)
Centre for Software  Maintenance,  University of Durham
```

|| **NOTE:** This section of the report describes the characteristics of the report.

```
RPG Beta Slice (without thematic structure): Change Implication Report (Beta)

Transformation Description      : document types and
                                : indirect ripple effects

Slice  Criterion                : Theme  Bag


State Trajectory (Domain )      : Ripple Propagation Graph

Projection Property             : Bijective Mapping

Codomain Properties             : Directed,
                                  Labelled,
                                  Strict Sub-Graph,
                                  Disjoint
```

```
The CIG contains the following types of entities and  dependencies:


   Graph Vertices:
      Composite Entities
      Composite Type  Entities
      Segment Entities
      Segment Type Entities
   Graph Edges:
      Co-occurs  Dependency
      Consists-of  Dependency
      Has-Segment Dependency
      Copy Propagation Dependency
   Features:
      Direct Impacts Indicated
      Indirect Impacts Indicated



The CIG contains the following impacted entities and dependencies:
```

|| **NOTE:** The first section of the report shows all of the directly impacted segment entities, the segment entity types and the composite entity slices. The composite entity slice contains all of the composite components and their types which are connected to each impacted segment entity.

```
DIRECT IMPACT on segment entity:g1

segment  g1 is a :  description-of-input

composite entity slice on  g1 is :
     sb2     document entity type sub-section
     s2      document entity type section
     c1      document entity type chapter
     b1      document entity type book
     v2      document entity type volume
     l1      document entity type library

DIRECT IMPACT on segment entity:g2

segment  g2 is a :  description-of-output

composite entity slice on  g2 is :

     sb2     document entity type sub-section
     s2      document entity type section
     c1      document entity type chapter
     b1      document entity type book
     v2      document entity type volume
     l1      document entity type library
```

|| **NOTE:** As in the $\beta$ $CIG(\mathcal{V}, \mathcal{E})$ for example 2 which is described in Chapter 8, the segment entity 2 is impacted twice by the theme bag. This is also reflected in this $CIR(\mathcal{V}, \mathcal{E})$.

```
DIRECT IMPACT on segment entity:g2

segment  g2 is a :  description-of-output

composite entity slice on  g2 is :

     sb2     document entity type sub-section
     s2      document entity type section
     c1      document entity type chapter
     b1      document entity type book
     v2      document entity type volume
     l1      document entity type library

DIRECT IMPACT on segment entity:g4

segment  g4 is a :  description-of-processing

composite entity slice on  g4 is :

     sb4     document entity type sub-section
     s5      document entity type section
     c2      document entity type chapter
     b1      document entity type book
     v2      document entity type volume
     l1      document entity type library

DIRECT IMPACT on segment entity:g5

segment  g5 is a :  description-of-processing

composite entity slice on  g5 is :

     sb4     document entity type sub-section
     s5      document entity type section
     c2      document entity type chapter
     b1      document entity type book
     v2      document entity type volume
     l1      document entity type library
```

|| **NOTE:** This section of the report shows all the indirect impacts in the document hierarchy

through descriptions of copy propagations.

```
INDIRECT IMPACT on segment entity: g2
propagated from segment entity   : g2

segment  g2 is a :  description-of-output

composite entity slice on  g2 is :

     sb2     document entity type sub-section
     s2      document entity type section
     c1      document entity type chapter
     b1      document entity type book
     v2      document entity type volume
     l1      document entity type library

INDIRECT IMPACT on segment entity: g3
propagated from segment entity   : g2

segment  g3 is a :  description-of-input

composite entity slice on  g3 is :

     sb2     document entity type sub-section
     s2      document entity type section
     c1      document entity type chapter
     b1      document entity type book
     v2      document entity type volume
     l1      document entity type library

INDIRECT IMPACT on segment entity: g3
propagated from segment entity   : g2
```

```
segment  g3 is a :  description-of-input


composite entity slice on  g3 is :


     sb2     document entity type sub-section

     s2      document entity type section

     c1      document entity type chapter

     b1      document entity type book

     v2      document entity type volume

     l1      document entity type library


INDIRECT IMPACT on segment entity: g5

propagated from segment entity   : g4


segment  g5 is a :  description-of-processing


composite entity slice on  g5 is :


     sb4     document entity type sub-section

     s5      document entity type section

     c2      document entity type chapter

     b1      document entity type book

     v2      document entity type volume

     l1      document entity type library


INDIRECT IMPACT on segment entity: g6

propagated from segment entity   : g4


segment  g6 is a :  description-of-an-interface


composite entity slice on  g6 is :
     sb4     document entity type sub-section
     s5      document entity type section
     c2      document entity type chapter
     b1      document entity type book
     v2      document entity type volume
     l1      document entity type library

INDIRECT IMPACT on segment entity: g6
propagated from segment entity   : g4

segment  g6 is a :  description-of-an-interface

composite entity slice on  g6 is :

     sb4     document entity type sub-section
     s5      document entity type section
     c2      document entity type chapter
     b1      document entity type book
     v2      document entity type volume
     l1      document entity type library

INDIRECT IMPACT on segment entity: g3
propagated from segment entity   : g4

segment  g3 is a :  description-of-input

composite entity slice on  g3 is :

     sb2     document entity type sub-section
     s2      document entity type section
     c1      document entity type chapter
     b1      document entity type book
     v2      document entity type volume
     l1      document entity type library

INDIRECT IMPACT on segment entity: g6
propagated from segment entity   : g5

segment  g6 is a :  description-of-an-interface

composite entity slice on  g6 is :

     sb4     document entity type sub-section
     s5      document entity type section
     c2      document entity type chapter
     b1      document entity type book
     v2      document entity type volume
     l1      document entity type library

INDIRECT IMPACT on segment entity: g6
propagated from segment entity   : g5

segment  g6 is a :  description-of-an-interface

composite entity slice on  g6 is :

     sb4     document entity type sub-section
     s5      document entity type section
     c2      document entity type chapter
     b1      document entity type book
     v2      document entity type volume
     l1      document entity type library

INDIRECT IMPACT on segment entity: g3
propagated from segment entity   : g5
```

```
segment  g3 is a :  description-of-input

composite entity slice on  g3 is :

     sb2      document entity type sub-section
     s2       document entity type section
     c1       document entity type chapter
     b1       document entity type book
     v2       document entity type volume
     l1       document entity type library
```

|| **NOTE:** This section of the report contains the information associated with a theme bag such as the themes and proposal identification information.

```
 CHANGE PROPOSAL DETAILS:

     Ripple Propagation Graph Analysed     : banking-sub-system-1

     Source of Proposal Number    : 1

        proposal number     :  1
        release number      :  2
        project number      :  2.1
        Division            :  secs
        Department          :  cs
        Name                :  s.carter
        Date                :  jan041992

     Content for Proposal Number    : 1

        Proposal 1 contains  the following :
```

```
    Contains Thematic Category    : th1
    Thematic Category Description : describes-processing-of-a1
    Contains Thematic Category    : th4
    Thematic Category Description : describes-processing-of-d1
    Contains Thematic Category    : th32
    Thematic Category Description : describes-processing-of-f2

no
```

|| **NOTE:** The "no" indicates that no further impact information can be located. However when Prolog is used to prove that a goal such as "Chapter 1 is connected to Chapter 2", then if "no" was returned it would indicate that the goal is false and if "yes" was returned it would indicate the goal is true.

```
| ?-
```

*halt.*

```
Prolog terminated
```

# Bibliography

[1] Agrawal, H. and J.R. Horgan, 1990, **Dynamic Program Slicing,** *In the Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, White Plains, New York, June 20-22* pp 246-256

[2] Agusa, K., Y. Kishomoto, and Y. Ohno, 1983, **A Supporting System for Software Maintenance,** *In the Proceedings of the IFIP TC2* pp 481-501

[3] Aho, A.V., R. Sethi, J.D. Ullman, 1986, **Compilers Principles, Techniques and Tools,** *Addison-Wesley*

[4] Arthur, J., 1988, **Software Evolution,** *A Wiley-Interscience Publication, John Wiley & sons* pp 74-75

[5] Avellis, G., A. Iacobbe, D. Palmisano, G. Semeraro, C. Tinelli, 1991, **An Analysis of Incremental Assistant Capabilities of a Software Evolution Expert System,** *In the Proceedings of the IEEE Conference on Software Maintenance, Sorrento, Italy* pp 220-227

[6] Bennett, K.H., B.J. Cornelius, M. Munro and D.J. Robson, 1988, **Software Maintenance: A Key Area For Research,** *University Computing, 10(4)* pp 184-188

[7] Bennett, K.H., 1990, **The Software Maintenance of Large Software Systems: Management, Methods and Tools,** *Software Engineering for Large Software Systems, edited by B.A. Kitchenham, Elsevier Science Publishers Ltd* pp 1-26

[8] Bennett, K.H., 1992, **The Post Graduate Handbook,** *University of Durham, SECS, Computer Science*

[9] Bennett, K.H., B.J., Cornelius, M. Munro and D.J. Robson, 1991, **Software Maintenance,** *Software Engineer's Reference Book, edited by J.A. McDermid, Butterworth-Heinemann, Section 20*

[10] Bigelow, J., 1988, **Hypertex and CASE,** *IEEE Software, 5(2)* pp 23-27

[11] Boehm, B., 1976, **Software Engineering,** *IEEE Transactions on Computers, 25(12)* pp 1226-1241

[12] Broughton, R., 1991, **Private Communication, May 2nd,** *TSB Bank plc, Retail Banking, Implementation and Testing*

[13] Bundy, A., 1983, **A Computer Model of Mathematical Reasoning,** *Academic Press*

[14] Calliss, F.W., 1989, **Inter-Module Code Analysis Techniques for Software Maintenance,** *Ph.D thesis, University of Durham, Computer Science*

[15] Capretz, M.A.M., 1992, **A Software Maintenance Method Based on the Software Configuration Management Discipline,** *Ph.D thesis, University of Durham, Computer Science*

[16] Chikofsky, E., 1983, **Application of an Information Systems Analysis and Development Tool to Software Maintenance,** *In the Proceedings of the IFIP TC2* pp 503-516

[17] Cimitle, A., F. Lanubile, and G. Visaggio, 1992, **Traceability Based on Design Decisions,** *In the Proceedings of the IEEE Conference on Software Maintenance, Orlando, Florida* pp 309-317

[18] Collofello, J. and D.A. Vennergrund, 1987, **Ripple Effect Based on Semantic Information,** *Proceedings AFIPS Joint Computer Conference, 56* pp 675-682

[19] Collofello, J.S. and J.J. Buck, 1987, **Software Quality Assurance for Maintenance,** *IEEE Software 1987* pp 46-51

[20] Conte, S.D., H.E. Dunsmore and V.Y. Shen, 1986, **Software Engineering Metrics and Models,** *The Benjamin/Cummings Publishing Company, Inc.*

[21] Cooper, S.D. and M. Munro, 1989, **Software Change Information for Maintenance Management,** *Technical Report 89/4, University of Durham, School of Engineering and Computer Science*

[22] Delisle, N. and M. Schwartz, 1986, **NEPTUNE: A Hypertex System for CAD Applications**, *In the Proceedings of the ACM SIGMOD International Conference on Management of Data* pp 132-143

[23] Dhar, V. and M. Jarke, 1988, **Dependency Directed Reasoning and Learning in Systems Maintenance Support**, *IEEE Transactions on Software Engineering, 14(2)* pp 211-277

[24] Dietrich, S.W. and F.W. Calliss, 1991, **The Application of Deductive Databases to Inter-Module Code Analysis**, *In the Proceedings of the IEEE Conference on Software Maintenance, Sorrento, Italy* pp 120-128

[25] Dietrich, S.W. and F.W. Calliss, 1992, **A Conceptual Design for a Code Analysis Knowledge Base**, *Journal of Software Maintenance: Research and Practice, 4(1)* pp 19-36

[26] Dowson, M. and J.C. Wileden, 1985, **A Brief Report on the International Workshop on the Software Process and Software Environment**, *ACM Software Engineering Notes, 10* pp 19-23

[27] Edisbury, B., 1990, **Private Communication, December 5th**, *TSB Bank plc, Technology Research, Retail Banking*

[28] Evans, C., 1991, **Private Communication, May 2nd**, *TSB Bank plc, Retail Banking, Implementation and Testing*

[29] Fairley, R., 1985, **Software Engineering Concepts**, *McGraw-Hill International Editions*

[30] Farbey, B.A., 1984, **The Use of Graph Theory for Modelling Thematic Structure in the Context Documents**, *Ph.D Thesis City University, Department of System Science*

[31] Foster, J.R., 1989, **Priority Control in Software Maintenance**, *In the Proceedings of 7th International Conference Software Engineering for Telecommunications Switching Systems, Bournemouth U.K.* pp 163-167

[32] Foster, J.R., 1989, **A Process Model for Software Maintenance**, *In the notes of the Third Software Maintenance Workshop, Centre for Software Maintenance, University of Durham*

[33] Gallagher, K.B., 1989, **Using Program Slicing in Software Maintenance**, *Ph.D. Thesis, University of Maryland, Baltimore*

[34] Gallagher, K.B., 1990, **Surgeon's Assistant Limits Side Effects,** *IEEE Software 7(64)* pp 90-98

[35] Gallagher, K.B. and J.R. Lyle, 1991, **Using Program Slicing in Software Maintenance,** *IEEE Transactions on Software Engineering, 17(8)* pp 751-761

[36] Garg, P.K. and W. Scacchi, 1989, **Ishys: Designing an Intelligent Software Hypertext System,** *IEEE Expert, 4(3)* pp 52-63

[37] Georges, M., 1992, **MACS: Maintenance Assistance Capability for Software,** *Journal of Software Maintenance and Practice, 4(1)* pp 199-213

[38] Glagowski, T., 1985, **Using a Relational Query Language as a Software Maintenance Tool,** *In the Proceedings of the IEEE Conference on Software Maintenance, Washington, DC* pp 211-220

[39] Gray, P., 1984, **Logic Algebra and Databases,** *Ellis Horwood Limited*

[40] Grosch, H.R.J., 1971, **Why MAC, MIS and ABM won't fly,** *Datamation 17* pp 71-72

[41] Haney, F.M., 1972, **Module Connection Analysis,** *In the Proceedings of the AFIPS Joint Computer Conference* pp 173-179

[42] Hart, C.F. and J.J. Shilling, 1990, **An Environment for Documenting Software Features,** *In the Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments* pp 120-121

[43] Hinley, D.S. and K.H. Bennett, 1992, **Using a Model to Manage the Software Maintenance Process,** *In the Proceedings of the IEEE Conference on Software Maintenance, Orlando, Florida* pp 174-182

[44] Horowitz, E. and R. Williamson, 1986, **SODOS: A Software Document Support Environment- Its Definition,** *IEEE Transactions on Software Engineering, 12(8)* pp 849-859

[45] Horwitz, S., T. Reps, and D. Binkley, 1990, **Interprocedural Slicing using Dependence Graphs,** *ACM Transactions on Programming Languages and Systems, 12(1)* pp 35-46

[46] IEEE Standards Board and ANSII Standards Institute, 1990, **An American National Standard and IEEE Standard Glossary of Software Engineering Terminology,** *ANSI/IEEE Std610.12-1990*

[47] Kaiser, G.E. and D.E. Perry, 1987, **Workspaces and Experimental Databases: Automated Support for Software Maintenance and Evolution**, *In the Proceedings of the IEEE Conference on Software Maintenance, Austin, Texas* pp 108-114

[48] Karakostas, V., 1990, **A Teleological Perspective of Software and its Application to Large Scale Reuse and Software Maintenance**, *Ph.D Thesis, Manchester, UMIST*

[49] Karakostas, V., 1990, **Modelling and Maintenance of Software Systems at the Teleological Level**, *Journal of Software Maintenance and Practice, 2(1)* pp 47-59

[50] Keables, J., K. Robertson, and A. von Mayrhauser, 1988, **Data Flow Analysis and its Application to Software Maintenance**, *In the Proceedings of IEEE Conference on Software Maintenance, Phoenix, Arizona* pp 335-347

[51] Keeney, R.L. and D. Winterfeldt, 1989, **On the Uses of Expert Judgement on Complex Technical Problems**, *IEEE Transactions on Enginering Management, 36(2)* pp 83-86

[52] Kellner, M.I., 1991, **Multiple-Paradigm Approaches for Software Process Modelling**, *In the Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences* pp. 175-188

[53] Lehman, M.M., 1980, **On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle**, *The Journal of Systems and Software 1* pp 213-221

[54] Li, D., 1984, **A Prolog Database System**, *John Wiley & Sons Inc, Research Studies Press*

[55] Lientz, B.P. and E.B. Swanson, 1980, **Software Maintenance Management**, *Addison Wesley*

[56] Liu, C., 1976, **A Look at Software Maintenance**, *Datamation, 22(11)* pp 51-5

[57] Luqi , 1990, **A Graph Model for Software Evolution**, *IEEE Transactions on Software Engineering, 16(8)* pp 912-927

[58] Mandrioli, D. and C. Ghezzi, 1987, **Theoretical Foundations of Computer Science**, *John Wiley & Sons*

[59] Martin, J. and C. McClure, 1983, **Software Maintenance - The Problem and its Solution**, *Prentice Hall*

[60] McCabe, T.J., 1976, **A Complexity Measure**, *IEEE Transaction on Software Engineering, 2(4)* pp 308-320

[61] Morgan, C., 1990, **Programming From Specifications**, *Prentice Hall, International Series in Computer Science*

[62] Moriconi, M. and C. McClure, 1979, **A Designer/Verfier's Assistant**, *IEEE Transactions on Software Engineering, 5(4)* pp 387-400

[63] Mullin, D. and S. McGowan, 1988, **Fortune's Functional Definition**, *Technical Report 2017/twp/128, CAP(UK) Limited, January, Alvey Project ALV/PRJ/SE/050*

[64] Nakagawa, A.T. and K. Futatsugi, 1991, **Propagating Changes in Algebraic Specifications**, *Software Engineering Journal, 6(6)* pp 476-486

[65] Nilson, N.J., 1971, **Problem Solving Methods in Artificial Intelligence**, *McGraw-Hill Publications, New York*

[66] Nosek, J.T. and P. Prashant, 1990, **Software Maintenance Management: Change in the Last Decade**, *Journal of Software Maintenance Research and Practice, 2(3)* pp 157-174

[67] Osborne, W.M., 1987, **Building and Sustaining Software Maintainability**, *In the Proceedings of the Conference on Software Maintenance, Austin Texas* pp 13-23

[68] Parikh, G., 1982, **Some Tips, Techniques, and Guidelines for Program and System Maintenance**, *Winthrop Publishers, Cambridge, MA* pp 65-70

[69] Patkau, B.H., 1983, **A Foundation For Software Maintenance**, *PhD Thesis, Department of Computer Scinence, University of Toronto*

[70] Pau, L.F. and J.B. Kristinsson, 1990, **SOFTM: A Software Maintenance Expert System in Prolog**, *Journal of Software Maintenance: Research and Practice, 2(2)* pp 87-111

[71] Penedo, M.H. and E.D. Stuckle, 1984, **Integrated Project Master Database (PMDB)**, *IR & D Final Report, TRW Technical Report, TRW-84-SS-22, December*

[72] Pfleeger, S.L. and S.A. Bohner, 1990, **A Framework for Software Maintenance Metrics**, *In the Proceedings of the Conference on Software Maintenance, San Diego California* pp 320-321

[73] Polimeni, A.D. and H.J. Straight, 1985, **Foundations of Discrete Mathematics**, *Brooks/Cole*

[74] Ramamoorthy, C.V., V. Carg and A. Prakash, 1986, **Programming in the Large**, *IEEE Transactions on Software Engineering 12(7)* pp 769-783

[75] Ramamoorthy, C.V., Y.Usuda, A. Prakash and W.T.Tsai, 1990, **The Evolution Support Environment System**, *IEEE Transactions on Software Engineering, 16(11)* pp 1225-1234

[76] Royce, W.W., 1970, **Managing the Development of Large Software Systems**, *In the Proceedings of WESTCON, San Francisco*

[77] Sharpley, W.K., 1977, **Software Maintenance Planning for Embedded Computer Systems**, *In the Proceedings of IEEE COMPSAC 77* pp 520-6

[78] Smith, J.Q., 1988, **Decision Analysis A Bayesian Approach**, *Chapman and Hall*

[79] Sommerville, I., R. Welland, I. Bennett, and R. Thomson, 1986, **SOFTLIB- a Documentation Management System**, *Software- Practice and Experience, 16(2)* pp 131-143

[80] Sommerville, I., 1992, **Software Engineering- Fourth Edition**, *Addison-Wesley Publishing Company*

[81] Song, N.L., 1977, **A Program Stability Measure**, *In the Proceedings of the 1977 Annual ACM Conference* pp 163-173

[82] Stankovic, J.A., 1985, **A Technique for Identifying Implicit Information Associated with Modified Code**, *In the Proceedings of the IFIP TC2* pp 457-478

[83] Sterling, L. and E. Shapiro, 1986, **The Art of Prolog Advanced Programming Techniques**, *MIT Press, Massachusetts Institute of Technology*

[84] Swanson, E.B., 1976, **The Dimensions of Maintenance**, *In the Proceedings of the 2nd IEEE International Conference on Software Engineering* pp 492-497

[85] Thebaut, S. and N. Wilde, 1986, **Program Change Analysis: Improving the Reliability of Modified Programs**, *University of Florida, Technical Report SERC-TR-1-F*

[86] Tsichritzis, D.C. and F.H. Lochovsky, 1982, **Data Models**, *Prentice Hall*

[87] Turski, W.M., 1981, **Software Stability,** *In the Proceedings of the ACM EUR Conference on Systems Architecture* pp 107-116

[88] Turver, R.J., 1989, **Software Maintenance: Generating Front Ends for Cross Referencer Tools,** *M.Sc. Thesis, University of Durham, Computer Science*

[89] Turver, R.J. and M. Munro, 1993, **An Early Impact Analysis Technique for Software Maintenance,** *Journal of Software Maintenance: Research and Practice (To Appear)*

[90] Weinberg, G., 1983, **Kill that code!,** *Infosystems, August* pp 48-49

[91] Weiser, M.D., 1979, **Program Slices: Formal, Psychological and Practical Investigations of An Automatic Program Abstraction Method,** *Ph.D. Thesis, University of Michigan*

[92] Weiser, M.D., 1984, **Program Slicing,** *IEEE Transactions Software Engineering, 10(4)* pp 352-357

[93] Wild, C., K. Mally and L. Liu, 1991, **Decision Based Software Development,** *Journal of Software Maintenance Research and Practice, 3,(2)* pp 17-43

[94] Woodcock, J. and M. Loomes, 1989, **Software Engineering Mathematics,** *Pitman*

[95] Yau, S.S., J.S. Collofello and T. MacGregor, 1978, **Ripple Effect Analysis of Software Maintenance,** *In the Proceedings of the IEEE COMPSAC November 1978* pp 60-65

[96] Yau, S.S. and J.S. Collofello, 1979, **Some Stability Measures for Software Maintenance,** *in the Proceedings of the COMPSAC '79, New York* pp 674-679

[97] Yau, S.S. and J.S. Collofello, 1980, **Some Stability Measures for Software Maintenance,** *IEEE Transactions on Software Engineering, 6(6)* pp 545-552

[98] Yau, S.S. and S.C. Chang, 1984, **Estimating Logical Stability in Software Maintenance,** *In the Proceedings of the IEEE Computer Society Computer Software and Applications Conference, Chicago, Illinois* pp 109-119

[99] Yau, S.S. and J.S. Collofello, 1985, **Design Stability Measures for Software Maintenance,** *IEEE Transactions on Software Engineering, 11(9)* pp 849-856

[100] Yau, S.S. and P. Chang, 1988, **A Metric of Modifiability for Software Maintenance**, *In the Proceedings of the IEEE Conference on Software Maintenance, Phoenix, Arizona* pp 374-381

[101] Yeh, R.T. and F.G. Sobrinho, 1984, **Complexity Measures for Software Evolution**, *Technical Report TR-1422, University of Maryland, Computer Science*