An approach to impact analysis in software maintenance

Fillon, Pierrick

An Approach To Impact Analysis in Software Maintenance

Pierrick Fillon

Thesis submitted for the requirements of the degree of Master of Science

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

30th June, 1994

Kaizen
改善

Kaizen is more akin to a philosophy, it is rather an amalgamation of interrelated principles ..... of initiating improvements. The starting point for *Kaizen* is the recognition and an admission that a problem exists, the ultimate goal is to make a gradual change.

A mes Parents, A Elena

Keywords: static impact analysis, ripple effects, traceability, dependencies.

Abstract

Impact analysis is a software maintenance activity, which consists of determining the scope of a requested change, as a basis for planning and implementing it. After a change request has been specified (change understanding) and the initial part of the system to be changed has been identified (change localization), impact analysis helps to understand consequences of the change on other parts of the system. Induced changes, also named ripple effects, among software components are detected. Most existing approaches perform impact analysis for changes occurring at the code level.

In this thesis, concepts developed to perform impact analysis at the code level are applied to trace changes occurring at the design level. The method consists of proposing an activity model addressing the different steps of impact analysis and a data model on which propagations of changes can be traced. The method is validated with a case study applied to a system from the aerospace field. The tools we developed on PCTE help for consistency checks in HOOD based designs during editing. Our data-model based on an Entity Relationship notation describes a way to model HOOD diagrams in PCTE and further on to propagate changes on the repository.

Examples chosen address the design phase of a simple engine system. We show that addressing modifications at a higher level of abstraction than the code eases understanding and localization of changes. It also limits the propagation of ripple effects (i.e., unexpected behaviour of the system) by detecting secondary changes at an earlier stage.

Acknowledgements

I am most grateful to my supervisor Prof. Keith H. Bennett. His guidance, encouragement and enthusiasm have been much appreciated throughout the different stages of this project.

I would like to thank in particular Dr. Albert Bokma and Dr. Gerardo Canfora, who have given me support and advice during my stay at Durham. Beyond having been colleagues, there are first of all very good friends. Thanks are also given to the staff of the Department of Computer Science at Durham University for their assistance.

The financial support provided by Matra Marconi Space France, and the guidance, in particular from Dr. Jean-Pierre Queille, Technical leader of the EPSOM research team working within the Eureka-ESF project is gratefully acknowledged.

Declaration

The work contained in this thesis has not been submitted elsewhere for any other degree or qualification, and unless otherwise referenced it is the author's own work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Purpose of the Research

This thesis presents an approach to impact analysis at the *design* level for the purpose of software maintenance. The aerospace industry is confronted with large scale software systems, sometimes of several millions of lines of code and with a lifetime typically of around 20 years. Due to the high cost of for developing new systems, this lifetime tends to be extended. Thus a greater emphasis is put on evolution, reuse and maintenance activities. Software maintenance has been estimated as the most costly phase of the life-cycle. It accounts for more than 50% of the total life-cycle costs and according to classic studies by Lientz [53, 54, 55] and Nosek [65], it shows no sign of declining. As software systems evolve, maintenance operations become more complex and as a result maintenance projects often fail to meet deadlines and cost targets. One reason for the high cost of maintenance is that there is a lack of methods to estimate the scope of changes. Two important factors for this activity are analysis and validation of the modified software.

Impact analysis [1] is a maintenance activity, which consists of analyzing the software and determining the scope of a requested change, as a basis for planning and implementing it. A system is constructed of components that are connected. Given an initial change the aim of impact analysis is to detect all components that are affected and must be changed as a result. This approach helps to control unexpected behaviour of the system after the maintenance operation. Moreover it is important to detect those affected parts before deciding whether to implement the changes, for example, in order to correctly evaluate the total cost of the change.

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

1.2 Motivations and Objectives

Maintenance involves different phases of the software life-cycle. Change analysis must consider where the change originally occurs and according to the type of the modification the level it propagates to. There are a number of motivations to perform impact analysis at the design level, and which can be summarized as follows:

- Most current approaches perform impact analysis for changes occurring only at the code level. This may be a wrong approach since most change requests address other **artifacts**, and in particular design documents.
- Addressing changes that appear at a higher level of abstraction than the code, such as in documentation describing the design, eases understanding and localization of changes. It also limits the propagation of unexpected results by detecting effects of changes at an earlier stage.
- Current techniques are difficult to apply at earlier stages of the maintenance life-cycle, where little or no documentation is available about the system. It is then at that point critical to assess the impact of change requests, which are stated at a higher level of abstraction. Thus, looking at software components, such as design documents provides a valid background.

The main objective in the present thesis is *to propose an approach performing impact analysis at a higher level in the software maintenance process than the code level.* The method that is presented consists of an **activity model** addressing different steps of impact analysis and a **data-model** on which propagation of changes can be traced. Finally, the method is validated with a case study on documents describing the design of a simple aircraft Engine Monitoring System.

1.3 Requirements

This thesis is based on the following assumptions. It is assumed that only functional attributes are relevant and not attributes such as safety, security, reliability or performance. Furthermore, approaches related to knowledge base systems or natural language processing are not being addressed.

The test case in the thesis refers to systems in which code, design components, and related documentation, are available for investigation. The method addresses technical issues of impact analysis. Although, the research work is applied to the field of aerospace systems, it has a wider applicability.

1.4 Thesis Structure

This thesis is divided in two parts.

Chapter 2, 3 and 4 give an introduction to the research field. Chapter 2 outlines different types of operations conducted during software maintenance activities and presents several activity models related to the field of impact analysis. Chapter 3 introduces to the state-of-the art in impact analysis. It presents approaches and techniques at the code level. This study provides a background for drawing analogies at the design level and the state-of-the-art of design analysis is presented. Chapter 4 describes the HOOD method investigated in this thesis.

Main contributions are presented in chapters 5 and 6 defining an interconnection model for design changes. Chapter 5 presents an interconnection model and chapter 6 illustrates it with a *case study*. Conclusions and suggestions for further research are outlined in chapter 7.

Chapter 2

Maintenance and Change Analysis

Software maintenance has been identified as a crucial activity in the process of software evolution. Changes occur at different levels in the life-cycle and correspondingly there exist different types of changes or modifications, as well be addressed in section 2.1. Therefore a general process model to conduct change analysis is required, and which supports impact analysis, as described in section 2.2. The place of impact analysis in the frame of the maintenance process is explained in section 2.3. Since a system is made from components, which are connected, it is necessary to investigate the traceability between those components to determine the scope of the change, as described in section 2.4.

2.1 Types of Software Maintenance

IEEE [43] defines **software maintenance** as: *the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment.* Consequently, as Lientz states it [53], maintenance activities can be classified according to their type as follows:

- **Corrective maintenance**
  Even with the best quality assurance it is likely that defects will occur in the software. Corrective maintenance identifies repeatable errors, corrects them and generates test cases.
- **Adaptive maintenance**
  Over time the original environment for which the software was developed may change. For example, the software has to be adapted to a new processing

environment such as new processors, operating system or changes of the data environment such as peripheral devices. These changes are classified as adaptive maintenance operations.

- **Perfective maintenance**
  This type of maintenance describes functionality enhancements of a software system. When a software is used the user wants additional functions and further enhancements to the existing system. The maintainer then has to introduce new functions to the system without adversely affecting the current behaviour and functionality of the system.

- **Preventive maintenance**
  Unlike other types of maintenance, preventive maintenance is undertaken independently of any change request anticipating changes to be relevant such as modifications or enhancements. Another aim is to improve the software with regard to performance, quality, standard conformance or maintainability. Since this maintenance operation is undertaken when the system is under investigation, previous experiences give a greater confidence to maintainers in introducing additional changes. The main tasks are to review the software system to make it more maintainable and to improve its structure. Different techniques such as redesigning, recoding or testing are used to achieve this goal.

Lowell [57] uses a slightly different terminology for these three types of maintenance (corrective, adaptive and perfective maintenance). In particular, the definition of perfective maintenance that Lowell uses cover also activities relevant to the above definition of preventive maintenance. The diagram emphasizes for each type of maintenance particular program's attributes, namely its specification, design, implementation or quality. Records on NASA projects show for example, that 90% of the corrective maintenance classifies changes occurring at the code level.

For each type of change to be 'impacted' a different process-model may be defined. The aim of impact analysis is then to consider the level where changes occur ('starting point') to detect induced changes ('ending points'). Moreover, for error corrections (corrective maintenance), possible origins may be found at different levels. For example, errors occurring in the code may be due to errors at a higher level, such as requirements mis-understanding. The correction then will induce changes in other artifacts (e.g., design changes to keep the system in conformance with the code). Table 2.1 illustrates the relation between maintenance types and the level where the change request arises.

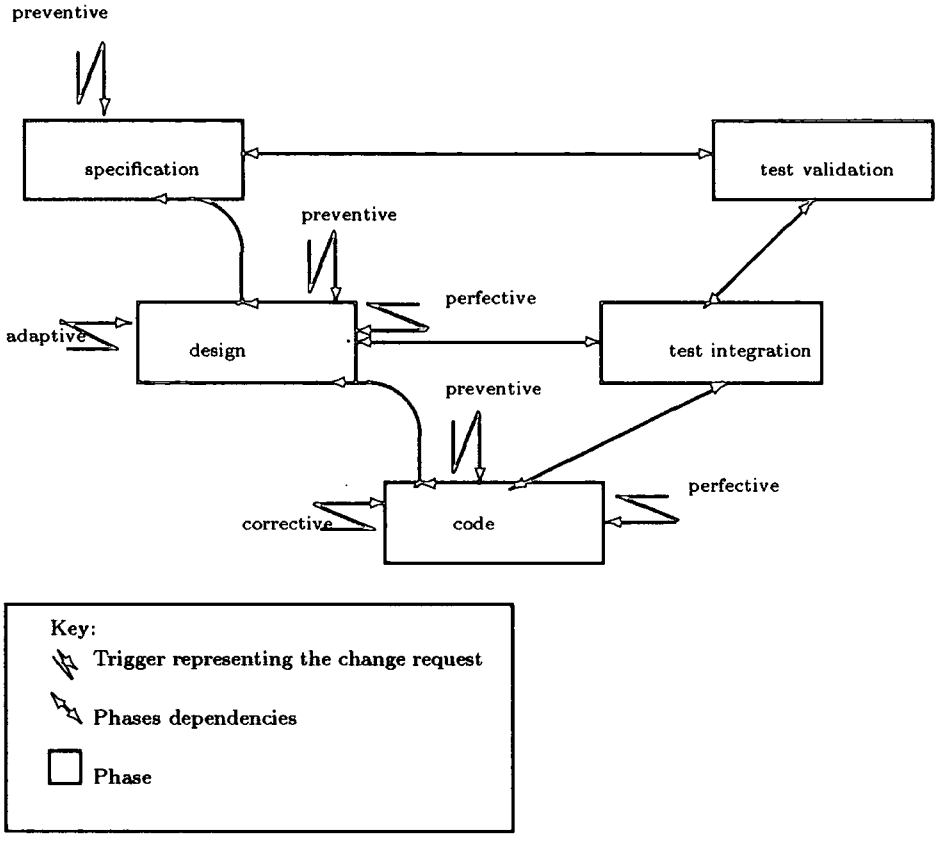

Figure 2.1: Maintenance types and level of the change request

2.2 Modelling Maintenance Activities

As Collofello [24, 23] explains it actors [1] involved in maintenance expressed needs to improve the maintenance process, both at managerial and technical levels. **Process modelling** is a step in this direction, Lowell [57]. It defines detailed analysis and modelling of maintenance activities in order to understand the process (descriptive point of view), to control it (prescriptive point of view) and to guide it (indicative point of view). The chosen model in this thesis is the generic process model developed in the Eureka project called EPSOM [36]. It has the advantage of covering a large group of technical activities used for maintenance operations. The model has been initially derived from past and on-going projects (descriptive aspect). The aim is to observe actual practice, and to characterize and improve the maintenance process. Terminology and activities describing process models are in accordance with ESA standards, as is the EPSOM model.

[1] An actor is a human agent carrying out a role during the process.

2.2.1 A Generic Process Model for Maintenance

This subsection describes a generic process model developed in the EPSOM project. It is based on a classical software engineering model referenced as the *'V-like'* model. Note that **process model** and **activity model** are notions, which are often confused. An activity model is a particular view of the process model concentrating on activities. In this case, notions of actors and documents are not relevant.

The EPSOM model identifies two main phases delimited in time by the activity, which consists of implementing the change. Thus, the descendant part of the *'V-cycle* is related to understanding and development steps including impact analysis, whereas the ascendant part concerns mainly testing activities. The model is presented below. It includes six steps and describes corrective maintenance operations.

The entry point of the maintenance process corresponds usually to the detection of a trigger called a 'software problem report' (SPR). This SPR [2] indicates an anomaly in the use of the software, a change request or a difficulty in understanding how to use the software.

- **Step 1: Problem Understanding**
  The aim is to gain a sufficient understanding of the problem to decide if the problem has to be pursued or not.

- **Step 2: Localization**
  This step determines the origin of the anomaly. It identifies parts of the system, which are concerned to the new or changed requirement.

- **Step 3: Solution Analysis**
  After the localization step, several (or just one) solution(s) may be considered. The next step is to perform an impact analysis for each of them to decide which solution is preferable.

- **Step 4: Impact Analysis**
  The aim is to estimate all changes which have to be carried-out in addition to the initial change. The impact is first considered at the technical level and then at a managerial level. The *technical impact analysis* consists of determining the propagation on all system components. It is applied to code, design, requirements and tests items. The *managerial impact analysis* investigates changes generated in the whole system but also points out other consequences such as costs or organizational issues. It is applied to the schedule of work, costs of the maintenance operation, and costs of the non-availability of the

[2] In this process model the SPRs are collected by the actor called the SRB - Software Review Board.

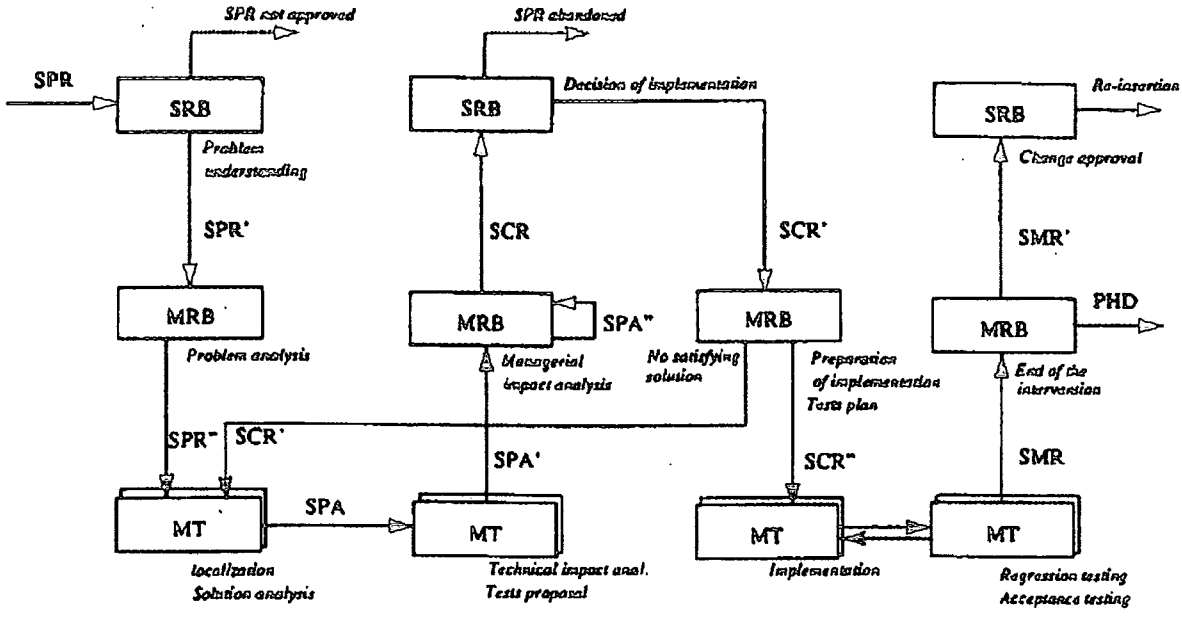

Figure 2.2: EPSOM generic process model for maintenance

system. After that step, a *decision* has to be taken, which is either *to implement the change issued by one of the proposed solutions*, or to leave the system *as it is without any intervention.*

- **Step 5: Implementation of Changes**
  The selected solution is implemented and validated following the usual V-like life-cycle.

- **Step 6: Tests Generation and Validation of the System**
  Regression and acceptance testing are part of the maintenance operation. Regression testing checks that *'what has not been intentionally changed is not changed'*. It implies that if a failure is detected the impact analysis step has to be re-processed. IEEE [43] definition of acceptance testing is that it checks that *'the implemented change is in conformance with the problem report'*.

The end of the maintenance operation is marked by a trigger closing the intervention. This activity model provides a complete framework for the whole maintenance process.

2.2.2 Activity Models for Impact Analysis

The impact analysis phase has been described within the maintenance process. The EPSOM model proposes that when the change is localized and a solution is chosen, maintainers have to conduct an impact analysis. Subsequently, a decision is taken to implement the change, or to leave the system as it is without any intervention. As we are concerned with technical aspects not only restricted to corrective maintenance operations, it is necessary to study other models. Moreover, it helps to understand the evolution in the field of process modelling during the last twenty years. These models specify activities connected to impact analysis such as organizational, or managerial activities.

First, we present a general model introduced by Yau, and secondly a more detailed model from Mac Clure that provides guidelines on how to perform the various tasks of maintenance. Finally, a model presented by Pfleeger and based on a metrics approach is described.

Yau's Model: A Ripple Effect Assessment Based on Stability Metrics. This model [84, 89]represents information about development and maintenance of software systems and emphasizes the relationships between phases of the software life cycle. It also gives the basis for automated tools to assist maintenance personnel in making changes to existing software systems. It is the first model presenting an activity related to the measurement of ripple effects through stability analysis. Each phase of the process is associated with software quality factors and metrics. After having as described determined the maintenance objectives, maintainers have to go through phases the following steps:

1. *Understanding the Program.* This consists of analysing the program in order to understand it. The phase is associated with complexity, documentation and self-descriptiveness attributes of the program.

2. *Generate a Maintenance Proposal.* A maintenance proposal is generated to perform the implementation. This requires a clear understanding of both maintenance objectives and program to be modified. According to Yau, the ease of generating a maintenance proposal is affected by the extensibility attribute.

3. *Accounting for Ripple Effect.* This phase is crucial for impact analysis. Ripple effects are annotated as a consequence of program modifications. Yau states that "*if the stability of a program is poor, the impact of any modification on the program is large*".

4. *Testing.* The modified program is tested to ensure that it has at least the

same reliability level as before. The relevant attribute for this phase is the testability of the program.

Yau's view is focussed on *'accounting for ripple effect'*. This measure is based on the stability of the program defined as *'the resistance to the amplification of changes in the program'*. Yau's model like those of Boehm [14], Liu [56], or Sharpley [77] are the earliest software maintenance models. Although Yau is the first author to focus on impact analysis, the proposed activity model is very simplistic. The 'order' of phases to be followed is explained, but details of how they should be performed are not addressed. Recent models like this of Mac Clure [21] provide further details on how to perform maintenance operations. This model is particularly interesting since it illustrates the notion of *'a-posteriori' impact analysis.*

Mac Clure: A-posteriori Impact Analysis
This model [64] is a refinement of the general Boehm model for maintenance. This model is also based on three main steps: *Understanding the change request*, *Performing the modification* and *Validation of the modification.* Mac Clure [21, 22] distinguishes three sub-steps for the second step - *Performing the modification-*, which are *Design the change*, *Alter the code* and *Minimize side effects.* In this model, when the change has been understood, it is implemented and side effects measured according to new modifications that occur. This *'a-posteriori'* strategy is inadequate because by looking at the symptom rather than at the origin of the change only few ripple effects are detected.

Most recent models dedicated to software maintenance, like those of Foster [35] or Pfleeger are for a number of reasons more elaborate. For example, Pfleeger's model is interesting to investigate, although it addresses the management rather than the technical viewpoint of maintenance. The author proposes to improve the maintenance process by managing it through metrics.

Pfleeger's Model: A Framework for Software Maintenance Metrics.
This model [70, 71] emphasizes impact analysis and proposes a framework for software maintenance metrics support. A set of metrics is proposed to help management to take decisions with regard to the modification to perform. The major activities are:

1. *Manage software maintenance.* This controls the sequence of activities by receiving feedback with metrics and determining the next appropriate action.
2. *Analyse software change impact.* It evaluates the effects of a proposed change regarding the scope of the impact and the traceability of the system after

performing the change.

3. *Understand software under change.* Source code and related product analysis are needed to understand the software system and the proposed change. The likely degradation of system characteristics (e.g. system complexity or quality of the documentation) help to decide if the change has to be implemented or not.

4. *Implement maintenance change.* The proposed change is performed. The adaptability of the system is analysed to perceive the difficulty of implementing the change.

5. *Account for ripple effect.* This phase consists of analysing the propagation of changes to other modules as a result of the change just implemented. Stability, coupling and cohesion of affected modules helps to check the effectiveness of the impact analysis.

6. *Retest affected software.* Modifications are tested to meet new requirements, and the overall system is subject to regression testing to meet existing ones. Testability, completeness and verifiability are evaluated in this activity.

Pfleeger's model shows an enhancement of Yau's model, as the software quality factors and metrics are associated to development phases to monitor product and process quality. Through the metrics proposed, this model is restricted regarding types of maintenance requests it may cover. Evolutive [3] and adaptive maintenance [4] are supported by the framework. Other types however, such as preventive, anticipative or perfective maintenances cannot be performed with this model.

It should be noted that existing models are mainly based on a qualitative approach and therefore predominantly handle management rather than technical issues. Maintenance operation may affect unexpected parts of the system. Thus it is necessary to conduct an impact analysis in order to estimate the propagation of proposed changes. Developing a process model helps maintainers in this task. This model has to be applicable to modifications occurring in the code, but also in design or documentation artifacts.

[3] e.g., a new functional requirement.

[4] e.g., requirements concerning environment modifications.

2.3 Software Change Analysis

Change analysis is one of the steps in the maintenance process and is characterized by steps 3 and 4 of the EPSOM model described previously . A task performed by change analysis consists of identifying the impact of the proposed modifications. Change analysis is directly affected by the quality of the program components and therefore different analysis (code analysis, design analysis) have to be conducted. Most of existing techniques tend to consider direct effects of a modification on the source code. Very few techniques are used to identify indirect effects. This is the goal of impact analysis.

2.3.1 Definitions for Impact Analysis

Weiss [80] proposed to classify changes according to the development phase they are related to. His classification is the result of an investigation carried out on NASA projects. For each phase, statistics are given in percentage of the total number of changes.

- Requirements phase. *19%* of the changes involve modifications of requirements or functional specifications.
- Design phase. *52%* of changes expressed by maintainers correspond to design modifications. Thus, it is the phase where the largest number of modifications occur.
- Code implementation phase. Only *7%* of the modifications concern interventions on the code such as insert, delete or debug components.
- Miscellaneous changes. Weiss classified in this category 'environment changes' (*3%*) caused by changes in hardware or software environments and 'planned enhancements' (*19%*).

Weiss refines the study measuring *'the number of code components affected several times for a considered change'* [5]. He found that 34% of changes affected only one component and 26% of changes affected two components. These empirical measures show that most modifications are caused by changes in the design, and that those changes are propagated among artifacts developed in other phases, such as specification, or code. In our study we investigate changes occurring at the design level.

[5]For Weiss the granularity of a code component is a function.

Impact analysis. Impact analysis is concerned with *determining the scope of a requested change as a basis for planning and implementing it.* Lowell [57] defined impact analysis as follows:

> *"Impact analysis is the activity of determining parts of the system to be modified in order to accomplish a change. To accomplish a change means to determine the confidence that the change conforms to its specification or to what we intend it to do."*

Since Lowell's study, the definition and understanding of the problem have been improved. Recent models such as the EPSOM model propose a more detailed definition: **"Impact analysis consists in estimating all the changes, which are consequent to the initial change".** Accordingly, impact analysis is an activity that is performed after understanding and locating the change to be accomplished. Impact analysis also has a wide spectrum and is related to several phases of the maintenance life-cycle (figure 2.2). Thus, changes to the system may produce unexpected results on code, design or analysis artifacts. In order to avoid this, new test cases are generated [6] or a simpler solution may be proposed.

2.3.2 Definitions for Ripple Effects Propagation

This section introduces definitions related to the field of impact analysis. Although ripple effect analysis is a refinement of impact analysis, many authors use it synonymously. As for many terms used in maintenance there is no standard definitions even though a recent publication from IEEE [44]. [7]

Ripple effects propagation. When used in the context of software maintenance, *ripple effect propagation* implies errors or undesirable behaviours that occur as a result of a modification. Yau [88] provides the original definition:

> *"Ripple effect propagation is the phenomenon by which changes made to a software-component along the software life-cycle [specification, design, code, or test phase] have tendencies to be felt in other components."*

The application of ripple effect analysis is to identify components, which need additional maintenance effort to ensure their consistency to the original change. It

[6] Regression testing addresses the testing of a system or components to verify that modifications have not caused unintended effects and that the system or components still comply with the specified requirements.

[7] Note: From the author- Although it is only just been published, this standard model seems very much of the 1986 era of development. It is also heavily criticised in the maintenance community for being too little and too late.

suggests that changes affect the whole system at different level of representation (code, documentation, or design). Pfleeger [71] summarized this view and suggested the need for traceability to detect ripple effects (section 2.2.2). *"Impact analysis is the assessment of the effect of a change. It aids the maintenance team in identifying software work products affected by software changes".*

Side effects and ripple effects. There is a difference in usage of the terms ripple effects and side effects by authors. Thus, the two notions are sometimes confused, in particular because they are mainly both applied at the code level. **Side effects** are a facility used when the connection between two elements might produce by-product effects, which were intended. For example, side effects might suggest a design decision that must implement an algorithm in an iterative or recursive way.

Conversely, according to Yau [88] **ripple effects** correspond to the phenomenon that changes made to a program in one area have the tendency to be felt in other areas. In other words, it relates to the propagation of a change. The application of such analysis is to identify areas, which need additional maintenance efforts to ensure consistency with the original change. Ripple effects occurring at the code level concern changes to a statement, a variable item or a subprogram. They may be detected during regression testing. Ripple effects may also have their origin at the design level (i.e., due to a design change) and can induce major code changes.

Another view is given by authors who recognize that ripple effects occur also at the documentation level because maintenance focuses not only on source code modification, but also on maintaining documentation. Agusa [1] supports this view and states ripple effect as follows:

> *"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."*

Ripple effects in documentation exist when changes to code are not reflected in supporting technical documentation (code or design documentations) or user-oriented manuals. Some authors, for instance Pressman [72], consider that documentation, which does not accurately reflect the current state of the software is worse than no documentation at all. This leads often to an incorrect assessment of software characteristics. However, some maintenance requests only focus on documentation. It is the case when the documentation has to be clarified without intervening changes to the software such as design or code components. Most errors occurring in documentation are mostly reduced by reviewing techniques.

2.4 Traceability and Maintenance

A major distinction between development and maintenance is the set of constraints imposed on maintainers by the existing implementation of the system. Information about system dependencies may be missing, incorrect or complex as a result of continued changes. A model of the maintenance process must indicate a framework to evaluate and perform the change. Models previously presented evaluate effects of a proposed change. This activity of analysing the software change impact [8], determines if the modification affects the rest of the system. Main factors to investigate are scope of the impact and traceability of the system. The former corresponds to the number and size of artifacts affected by the initial change. The latter expresses an 'ability' of a system that suggests the connectivity of relevant work-products and whether the system is easy or hard to navigate once the proposed change is performed. Some research has been done in the field of traceability for maintenance purposes.

2.4.1 Definitions for Traceability

This subsection introduces to the terminology related to traceability. Traceability issues have been raised during development phases of the system. The term traceability is defined by IEEE [43] as follows:

> *"Traceability is the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another".*

Traceability covers the set of relations between input and output at each step of the development process and is distinguished in two types *horizontal and vertical traceability. Vertical traceability* relates to relationships between work-products. For example, each design component is traced to code components that implement that part of the design. Similarly, *horizontal traceability* addresses relationships among parts of the work-product such as requirements, design, code or tests items. Both types of traceability are necessary to understand the complete set of relationships to be assessed during impact analysis.

Studies on traceability. In the early 1980's, the need for traceability in maintenance is firstly reported by Lowell [57]. This author based his approach on those two types of traceability and distinguished different impacts of a change. Firstly,

[8] It is annotated as the step 'Accounting for ripple effect' in the Yau model and as the step 'Performing the Modification' in the Mac Clure model -subsection 2.2.2.

impacts that may be traced in items represented in the same formalism (e.g., modifying a specification may affect other specifications). Secondly, impacts may be traced among items represented differently (e.g., tracing changes in a design item to changes in related code items). Recent studies on traceability in maintenance in the EPSOM project [36] have refined this approach. Traceability makes it possible to trace from the requirement of a particular function in a system, through its specification, design and actual program code implementing it.

Traceability is not only restricted to the development process, as it can also be provided for testing purposes. This is illustrated by different types of propagation. A *'left to right'* propagation [9] is defined as the traceability of the output of a development phase against the input of the corresponding validation phase. For example, architectural design documents are connected to integration test cases. Similarly, a *'right to left'* propagation, consists of validating the output of one validation phase against the input of the corresponding development phase. For example, it may be used to detect an erroneous test against a requirement.

Traceability and consistency. Another view of traceability is expressed by IEEE [43] as *'the degree to which each element in a software development product establishes its reason for existing'*. This statement supposes that traceability may be used in a system to control completeness within a phase and the consistency between phases. For example, it expresses the degree to which an element at the code level refers to the design that it satisfies.

Traceability and quality. A method for assessing and controlling change consists in explicitly incorporating metrics to express the traceability of a system. The activity model presented by Yau [82] (section 2.2.2) is very useful in evaluating effects of change on the system to be maintained. However, this model does not explicitly refer to a metrics approach.

Pfleeger [71] proposed a framework of 'traceability and metrics' (section 2.2.2). The maintenance process is viewed regarding software work-products as a traceability graph of software life-cycle objects connected by horizontal and vertical traceability. The former addresses the process metrics (relationships across parts of the work-product). Different graphs are investigated (graphs related to requirement-design, design-code, code-test links) and the traceability of a graph is expressed as 'the number of paths in the minimal set of tracing paths'. Vertical traceability addresses product metrics (relationships among parts of the work-product) and complexity measures such as cyclomatic number V(g), number of nodes or in/out number of edges per nodes. The author considers that this type of relationship may be easily generated by compilers and other static analysis tools. The model depicts how metrics can be used to manage maintenance. The management of maintenance

[9] It is recommended to look at the development life-cycle in form of a 'V' to understand directions given for each propagation.

controls the sequence of activities by receiving feedbacks with metrics and determining the next appropriate activity.
Pfleeger's model offers a unified view of the impact of a change - along the software life-cycle - based on different types of traceability (horizontal and vertical). However, this model is restricted to management purposes and costs aspects of the change.

2.4.2 Definitions of Objects, Relationships and Closure

Traceability can be applied to different elements either on objects (e.g., on functions, interfaces, HOOD objects), on artifacts (documents, files) or between objects and supports. In this paragraph the need of traceability to perform impact analysis is focused on finding and determining impacts. Definitions are given for concepts of object, relationship and transitive closure.

In software systems, a general definition, of an **object** is that it corresponds to any 'concept' we choose to identify explicitly (e.g., a variable, a statement or a function).

A **relationship** between two objects A and B is a three tuple. Given objects a and b, a relation R is defined as $< a, R, b >$. A dependency is a *directed* relationship (e.g. calls, uses, read, write relationships) *A depends on B* means that a change to A, causes a change to B. Types of traceability described previously may be illustrated by several relationships.
Vertical traceability may correspond to links such as 'is-implemented-by (between design and code elements) or *semantical* links like 'is-described-by' (between elements of different nature such a a source-code and a user guide).
On the opposite, *horizontal traceability* may correspond to *structural* links such as 'calls', 'uses' (between functions) or to links between elements of same nature, such as 'is-composed-of' links. Traceability has an impact on maintenance activities. A modification may induces several changes and involve several relationships. Tracing those changes leads to a wide-spread impact.

Concerning **transitive closure**, Lowell [57] specifies that *'the basic goal of determining impacts is to find the transitive closure of a relationship (or set of relationships)'*. It can be defined formally (Aho [2]):

> *Definition: Transitive closure*
> *Let G be a graph. Define G* to be the graph that contains all nodes of G. The edges of G* are as follows: if there is a path of length 0 or more between node A and B in G, then the edge (A, B) is in G*. G* is called the transitive closure of G.*

In our study, for computing the transitive closure we choose a simple way.

Computing a transitive closure
Let us consider the computation of a transitive closure of a directed graph. If the graph is represented by a predicate *arc* such that *arc (X, Y)* is true *iff* there is an arc from node X to node Y, then we can express paths in the graph by the rules:
1) path (X, Y) :- arc (X,Y).
2) path (X, Y) :- path (X,Z) & path (Z,Y).

The first rule says that a path can be a single arc, and the second says that the concatenation of any paths, say one from X to Y and another from Y to Z, yields a path from X to Y. These rules are expressed by the following equation.

$$path(X,Y) = arc(X,Y) \cup \pi_{X,Y}(path(X,Z) \bowtie path(Z,Y))$$

where π and $\bowtie$ respectively represent projection and join of relational algebra.

2.5 Conclusion

This chapter outlines different types of operations conducted during software maintenance activities. Change analysis is one of the step of the maintenance process and concerns in particular impact analysis, which aims to detect all changes consequent to a modification. It has been explained that current activity models supporting impact analysis do mainly consider modifications at the code level, even though changes occur at different phases in the life-cycle.

The design of a modification requires an examination of ripple effects, unexpected behaviour of the system due to the initial modifications. If the impact is too large, or if the traceability is severely hampered by the change, management staff may choose at this point not to implement the change. Assessing the traceability to maintain a system helps then to find out complex, or highly coupled parts in the system.

Chapter 3

Background on Impact Analysis

Most existing techniques performing ripple effect analysis are applicable at the code level. They do not consider impact of modifications on program specification, analysis or design artifacts. This chapter presents techniques at the code level and gives an underlined background for the next chapter, applying a new approach at the design level.
In previous chapters, it has explained that early stages of the maintenance cycle are crucial for understanding the system. A careful examination of the documentation available is necessary. However, such information is not always available and other methods have to be used. by maintainers when performing impact analysis. It is necessary to understand the system looking at documents available such as design, source-code or code artifacts.

This chapter is concerned with techniques used at the code level. Section 3.1 introduces different approaches for code analysis and related techniques. Section 3.2 presents different types of dependencies analysis on which those techniques are applied. In particular, those techniques are compared and tools supporting them examined. Section 3.3 presents traceability models and motivations for propagating changes among software artifacts produced in the life-cycle. In section 3.4 conclusions and argumentation to perform impact analysis at earlier stages are outlined.

3.1 Types of Code Analyses

A software system may be analysed under different view-points either statically or dynamically, addressing both syntactical and semantical aspects.

Static analysis. IEEE [43] definition is "*Static analysis is the process of evaluating a system or a component based on its form, its structure, its content, or documentation.*"In software maintenance it may useful for different purposes to analyse a program without executing it. This thesis aims to apply static analysis to detect errors in design artifacts. There are four basic categories of static analysis.

The first category relates to **general analyses**, which aim to single out properties of the program. Inspection and walk-through techniques are traditional examples of such techniques.

IEEE [43] defines **Walk-through techniques** as techniques, which consist of *"A static analysis technique in which a designer or programmer leads members of the development team and other interested parties through a segment of documentation or code, and the participants ask questions and make comments about possible errors, violation of development standards, and other problems."*. Similarly, **inspection techniques** are defined in IEEE terminology [43] as: *"A static analysis technique that relies on visual examination of development products to detect errors, violations of development standards, and other problems. Types include code and design inspections"*. Although these techniques are slightly different to walk-through techniques, they are also difficult to automate.

A second category concerns **specific analyses**. It means for example the detection of specific classes of errors or anomalous constructs, such as inconsistencies between actual and formal parameters, or variables usages. In contrast to the previous category, such analyses can be easily automated, for example, to detect errors statically.

A third category corresponds to **symbolic execution**, which can be used for range-bound analysis (i.e. to confirm that variables, typically array index variables, will remain within bounds). Some control-structures such as 'loops' are manually investigated to test for exceptional conditions that might occur at run-time. In fact, such analysis is rather difficult to automate.

Finally, a last type of static analysis is based on **qualitative measurements** of program-code. Measurements help to estimate the effort required to understand the program and perform changes. Basic measurements concern size (in terms of lines of code) and complexity of the program structure. For example, Mac Cabe techniques correlate number of decisions (i.e. cyclomatic number called V(g)) with nesting structures. These metrics techniques have been largely investigated and much research has been undertaken in that field.

Dynamic analysis. IEEE [43] defines dynamic analysis as 'the process of evaluating a system or component based on its behaviour.' Fundamental aspects are introduced by Huang [42] who presents a detection of data flow anomaly through program instrumentation. Another approach proposed by Taylor [79] concerns algorithms for analysing concurrent programs. Such dynamic aspects of dependencies analysis are not under the scope of our study.

3.2 Study of Code Dependencies

Several types of dependencies between program entities exist and correspondingly, for maintenance purposes various techniques are available. The state-of-the-art is presented, in particular concerning data-flow dependencies analyses.

3.2.1 Types of Program Dependencies

Wilde [81] uses the concepts of *program entities* and *program dependencies* to define a dependency graph that helps in understanding relationships in a software system. Program entities are divided into *program modules* (such as procedures, functions and complete programs) and *data objects* (e.g. variables, data types, files and data structures). Wilde classifies program dependencies as follows:

- Definition dependencies where one program entity is used to define another.
- Type dependencies where one data type is used to define another type.
- Calling dependencies where one program module calls another.
- Functional dependencies between data objects and program modules that create or update them.
- Data Flow dependencies between data objects where the value held by one object may be used to calculate or set the value of another.

Both static and dynamic analysis can be used to investigate these dependencies. However our study focusses on data-flow, calling and functional dependencies since the two first categories are rather relevant to language issues. The aim is to understand how they affect the process of maintaining software.

3.2.2 Data-flow Dependencies Analysis

Concepts of data dependencies are introduced to model interactions between data items, such as variables and constants. Analysing data-flow dependencies consists in gathering information on use and definitions of those items in the program. There exist several types of data dependencies, but for the purpose of the thesis we shall concentrate on data-flow dependencies. Aho [2] defines data-flow analysis as *"Given a control flow structure, data flow analysis is the process of collecting information about the flow of data throughout the corresponding code segment."* Firstly, rules on variable usages are given and secondly different methods of analysis are presented.

Study on Variable Usages. Osterweil and L. Fosdick [67] carried out the first study in the field of data-flow analysis by examining definition/use pairs of data items as well as anomalous usages in Fortran programs. Osterweil distinguishes three possible states for a data, as:

- defined state - a value is stored in the variable (also named definition),
- referenced state - the value stored in the variable is used (also named use),
- undefined state- the value stored in the variable is unknown.

Osterweil [67] gives two rules on variable usages concerning sequence of actions that can be performed on them.

- Rule 1: A *reference* must be preceded by a *define* without an intervening *undefine*, also named read-value action.
- Rule 2: A *define* must be followed by a *reference* without an intervening *define* or *undefine*, also named write-value action

Traditional methods of analysis check if data items are correctly defined and referenced in the program. Violations can then be detected. Calliss [16] considered three resulting anomalous paths for a variable:

- undefined reference: the value stored of a variable is used before the variable is given a value. This violates rule 1.
- double define: the value stored in a variable is changed without intervening reference, the old value being not used. This violates rule 1.
- lost define: the value stored in a variable is undefined, the old value being used. This violates rule 2.

It should be noted that one category of anomalies has not been listed by Calliss, namely data items that are *defined* and *unreferenced.* It refers, for example, to the result returned by a function and which is never referenced. Detecting such anomalies requires an examination of each path in the flow graph, which is easily performed by a depth-first transversal algorithm.

Classifications of data-flow analysis methods.

Much work has been done on data-flow analysis. Initially, it has been performed statically with the help of tools such as cross-referencers [1], which present data in lists of definition/use pairs. More recent techniques tend to develop dynamic aspects to detect more precisely data-flow anomalies occurring at run-time. Three orthogonal views of data-flow analysis can be proposed.

- A control flow-graph view provided by **iterative and interval analyses.**
  Iterative analysis consists of traversing nodes in the control-flow graph of a program, propagating data-flow information as nodes are 'visited'. This procedure is iterated until the data flow information identified with each node does not change.

  Interval analysis is composed of two steps: the *elimination phase* and the *propagation phase.* It defines *intervals* as sub-graphs of the control-flow graph of a program. The elimination phase consists of combining these intervals and their data flow information. A succession of increasingly simpler flow-graphs results in the analysis of data-flow relations in the program. The propagation phase propagates the information back to the initial intervals. F. E. Allen [3] presents "A program data-flow analysis procedure" .

- Regeneration of data-flow information with **incremental and exhaustive analyses.**
  Following a change, all data-flow information for the whole program can be recalculated. The traditional approach of exhaustive analysis is nowadays carried out in tools (such as compilers, syntax-directed editors) by an incremental method for updating data flow information.
  The purpose of incremental analysis is different. When a segment of the program is changed, data flow information is updated with the information provided by the change. Not all types of change are supported by algorithms. In particular, structural changes, involving change of control-flow, are the most difficult to implement. A survey on incremental algorithms is provided by Burke and Ryder.

- Analysis of procedure usages and scopes of variables with **inter and intra-procedural techniques.**

[1] A cross-referencer recently proposed on the market is called Hindsight-C, for C code. It formats data in lists according to their usages in the program.

Analyses	iterative-interval	incremental-exhaustive	inter/intra-procedural	Interface
source of information	CF graph	DF graph	variables scope	interfaces
Criteria	intervals of CF	DF graph-segment	procedures calling point	data definitions and declarations, parameters

Table 3.1: Types of analyses for program dependencies.

Inter-procedural analysis calculates data-flow information only for one procedure or function at a time. When a call to a procedure appears, it is assumed that the procedure can modify or use any global variable. A classical inter-procedural algorithm is outlined by Barth [10] in his work called "A practical inter-procedural data-flow analysis algorithm".

By contrast, in intra-procedural analysis, for each procedure the *summary information* is calculated. This information usually consists of variables, which may be modified, used or preserved. The summary information is then used at the point of call of a procedure defined as the 'entry-point'.

These techniques are summarized in table 3.1. For each of them the source of information (i.e. structure from which items are extracted) and criteria of the analysis are outlined.

3.2.3 Alternative Techniques

Other techniques may be used to perform static analysis of a program. The two following examples investigate different views of the system, namely the controlling of its execution and the checking of interfaces it defines.

Control-flow dependency analysis. This technique is based on analysing the sequence of execution of statements in a program. It can be used for different purposes such as for structural analyses. Structural information detects potential errors in the code (and unreachable code or procedures having no-exit points) and assess the program complexity.

Interface analysis and type checking. This aims to check consistency of interfaces between modules for data definitions and declarations. For example, it checks errors due to a subprogram call with the wrong number of parameters, respectively with the wrong types.

3.2.4 Tools Support

Many features of analyses mentioned above are supported nowadays by compilers. These tools can be useful to conduct impact analysis at the code level. Most static analysis tools provide automatic or semi-automatic translators from the source programming language into their own target intermediate language. The purpose is to enable several types of analysis on the same 'platform' [2]. This approach aims to extract certain aspects of the code for specific analyses such a detection of global variables or access to external procedures.

Traditional references cite the DAVE tool developed by Osterweil [68, 66, 67] applied on Fortran programs to detect data-flow anomalies (i.e. errors) and inconsistencies in the system. Osterweil also presents basic algorithms supporting this tool. A second tool commonly referred in the literature is OMEGA [69], which analyses data-flow for C code. This tool has also been developed by Osterweil and Wilson.

3.3 Design Analysis and Vertical Traceability

This section presents the state-of-the-art in the field of design analysis, in particular for maintenance purposes. A traceability model between design and code is presented (section 3.3.1) that points-out benefits of investigating vertical traceability. To support the analysis of the process of changing the design, recording decisions may be useful (section 3.3.2). Then, motivations for propagation of changes between design and analysis artifacts are explained (section 3.3.3).

3.3.1 A Traceability Model between Design and Code

The traceability model presented in figure 3.3.1 proposes three areas of mapping between design and code.

- Area A, consists of design components or design decisions that are not traceable [3] or not implemented in the code. For example, it concerns design elements that represent a performance constraint on the software system. Adding such a constraint may not imply the creation of new code elements. This category also corresponds to design errors (e.g. forgotten or removed code element) or design discrepancies (redundant items or not used design elements).

[2] It should be noted that the translation of the source code representation into an intermediate form does not provide more semantics about the program.

[3] Not traceable refers to design elements having no related concept or representation in the code.

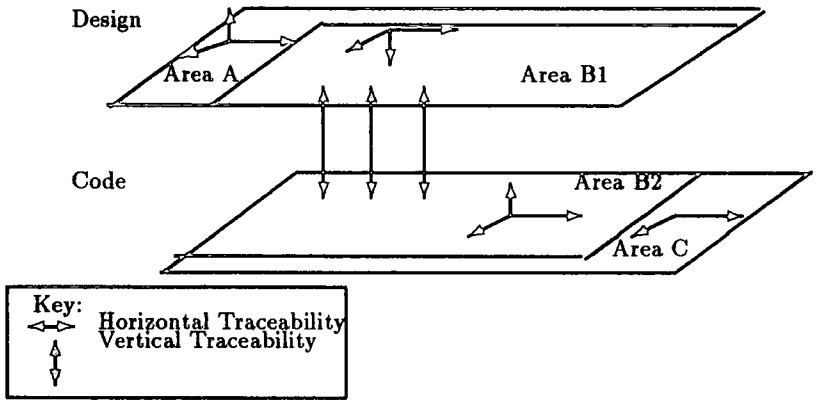

Figure 3.1: A Traceability Model Between Design and Code

- Area B, includes components of the design (area B1) traceable in the code (area B2). Dependencies in this area are probably the easiest to detect and trace with tools support. In this area, the traceability realizes a projection of the design into the code. It consists of identifying links between an object of the design and an object of the code. These links have the *n:m* cardinality that means a design element may be implemented by several code elements and vice-versa.

  Links used in the direct mapping between design and code are easy to trace for a system designed in HOOD and implemented in Ada. As explained in section 4.3, HOOD design principles are based on Ada mechanisms. Therefore, many tools exist, which automatically transform a HOOD design into Ada code skeletons, and conversely tools that abstract a HOOD design tree from Ada code [4].

- Area C, identifies code elements that have no representation in the design. It refers to low-level mechanisms (e.g. type of structure used in the code: arrays, pointers) not specified, but necessary for the implementation. It corresponds also to discrepancies in case the code represents functions which are never used by the software system, or remaining code elements from a previous implementation while the design has changed.

Definition of a process to investigate traceability.

The process of tracing aims to propagate changes within design items or between items represented at different levels of abstraction. The former case is called horizontal traceability (section 2.4.1), and restricts propagation of changes at the

[4]The HOOD design tree (HDT) is the tree of the system being designed, and consists of the root object and its successive decompositions into child- objects until terminal objects are reached.

design level. The later case is called vertical traceability and enables the detection of effects in the design induced by making changes in analysis or code artifacts.

For example, between design and code the process of tracing consists first of considering changes located in area C and to investigate effects on areas B. The second step is to use traceability links between design and code artifacts to propagate effects in the design (from area B2 to area B1). Finally, the last step concerns the propagation of those links within items not mapped in the code (area A).

The main benefits of this approach are to detect effects of changes investigating the system at several levels of abstraction.

3.3.2 Alternative Views of Design Analysis

This paragraph presents recent work in the field of recording design decisions. Arrango [7, 8] specifies that for maintenance operations, the designer should not only describe components that compose the system. It should also record decisions taken in selecting and modifying components. This author introduces this approach to Software Development & Maintenance and implements it with examples describing constructions of editors.

Lanubile [50] proposes a traceability support system based on design decisions. Different representation models of a same software system are possible according to the design method or the environment chosen (operating system or hardware platform). The model of Lanubile investigates systems, which are initially represented using a design method called the *Essential Model* and transformed later into a target model called the *Language-oriented model.* The system is based on traceability relations existing between objects, but also on tracking decisions that have a role in the transformation process. Therefore design decisions are recorded as entities in the graph description. Using a traceability model, which connects different views of a system structure with the design decisions made helps to evaluate effects of changes, and to choose between alternatives. Dependencies are evaluated through the named *'dependency descriptor'.* The value of this "descriptor" is modified by adding, removing or checking the existence of input, output, cause and derivation relations between components.

Arrango [6] refines and extends this approach by proposing a tool to track design decisions, which record four types of objects representing different aspects of the design process.

1. *Problem element objects* represent information on problems and solutions. It concerns the specification of the design called by Arrango the 'What-question'.
2. *Design decision objects* represent information about possible actions and choices.

It concerns the implementation of the design, called the 'How' part of the design process.

3. *Assertions objects* capture the justification of the design decision. Those elements answer the rationale of the design process i.e. the 'Why' part.

4. *agenda objects* refer to the physical number of the decision.

The first three types of objects capture information about design decisions, when agenda objects records only the index of the decision.

3.3.3 Motivations for Changes Propagation

The need to propose a traceability model to support impact analysis and the role of traceability to conduct maintenance activities has been discussed previously (section 2.4). We now have to focus on tracing between design and analysis artifacts, particularly between DCFDs (Data Flow Control Flow Diagrams) and HOOD diagrams.

Supposing a modification on a set of requirements objects is undertaken. To implement those modifications, the maintenance team modifies a set of objects in the analysis and design artifacts. Our interest is to trace such dependencies between analysis and design and to compare the set of design objects really modified to the set of design objects that is the projection of modified analysis objects. There are three possible cases, which may occur:

1. the two sets are rigorously equivalent. Modified design and analysis objects have a one-to-one relationship.

2. the two sets are different and some design objects, which were previously implementing the modified analysis element are not changed. This case arises when the element is implemented by several design objects. Then the modification has an effect limited to particular design objects. However it leads also to errors if the modification is not applicable (i.e. no related semantics in the design) or not feasible in the design.

3. the two sets are different because some design objects were modified when they do not implement any modified analysis elements. This is the case when the requirement modification implies the creation of new design objects only. It is also the case when several analysis elements are implemented by a unique design object.

These three cases show that it is important to estimate the traceability between analysis and design. Our approach proposes to record in a data-base relations between entities issued from analysis and design phases. Then a 'tracing tool' analyses the completeness backwards and forwards of those elements.

Backwards traceability requires that each output of a phase shall be traceable to an input to that phase. Outputs that cannot be traced to inputs are unnecessary. Backward tracing is normally done by including with each item a statement of why it exists (e.g. the description of the function of a component may be the list of functional requirements). For the purpose of our study, with a *backward* propagation -i.e., projection of the design into the requirements- **'nucleus'** objects that do no implement requirements objects are going to be detected (case 3).

Conversely, forwards traceability requires that each input to a phase shall be traceable to an output of that phase. Forwards traceability demonstrates completeness. Forward tracing is normally done by constructing cross-reference matrices and therefore holes in the matrix demonstrate easily incompleteness. In our study, a *forward* propagation identifies **'over-specification'** (also called fossils) that mean analysis elements, which are not implemented in the design or design lacks (case 2).

Applying such approach with two different types of propagation may be used to verify the consistency of objects and interfaces in HOOD versus the consistency of DCFD diagrams.

3.4 Conclusion

In this chapter, several techniques to perform impact analysis at the code level have been presented. However, changes occurring in the code represent a category of minor importance compared with the modifications happening in the whole software life-cycle. Different types of traceability and propagation mechanisms have been outlined. In particular it has been stressed that a modification in one phase has significant impacts on elements of the same phase, but also on other phases. Several types of dependencies have been outlined.

Implementation activities correspond to the translation of design elements in instructions executable by the computer. Conversely, design activities consist of translating software requirements into a set of representations describing data, structural, architectural and algorithmic aspects of the program. The purpose of the thesis is to propose ways of extending techniques investigated at the code level to perform change analysis at earlier stages in the maintenance of a project. In particular, maintainers would benefit from using a traceability support to analyse design artifacts.

Chapter 4

Design Analysis for HOOD

HOOD was developed by the ESA (European Space Agency) in conformance to software standards for aerospace projects [31]. It takes its starting point from Grady Booch [13] by adopting the object-oriented paradigm, but aiming to be more precise and 'well-described' in the definition of concepts and design process (e.g., definition of a *BNF*). Although its name HOOD suggests an *object-oriented approach*, it is only *object-based* because it does not support concepts defining an object-oriented model (e.g., inheritance).

HOOD is used by several ESA projects and is becoming a standard method in Europe for Ada projects. It has been used by large projects like Columbus (European space station program) and EFA (European Fighter Aircraft). The method is standardized, and defined in manuals [40, 39] owned and maintained by the HOOD Users Group. A wide range and number of tools and environments support the method.

This chapter presents an approach to the analysis of dependencies existing in design documents. The HOOD method and its design process are presented in section 4.1. Features and design principles used for our study are summarized in sections 4.2 and 4.3 respectively. Finally, a classification of design dependencies in HOOD is proposed in section 4.4. Concluding remarks are outlined in section 4.5.

4.1 The Design Process

The scope of the design phase extends from the decomposition of the functional requirements of the system to the implementation phase. Objectives are to identify data and functional components and static structures. Much of the creativity in this process is due to the possible decompositions of requirements. Some of those decompositions are solutions to the given problem, but others, which appear to be solutions are not. Designing is a combination of bottom-up, top down and middle out activities that is divided into several phases: logical design, architectural and detailed design. The HOOD method defines a design process from **architectural to detailed design** [1]. HOOD consists of four steps, each of them producing an artifact.

1. **Problem Definition:**
   This first step consists of understanding the problem to analyse and structure informal requirements. Documents produced are the Statement Of the Problem (SOP) and Analysis & Structuring data Requirements (ASR).

2. **Elaboration of an informal solution strategy:**
   Requirements are refined into a design solution. The Informal Solution Strategy (ISS) document is produced.

3. **Formalisation of the strategy:**
   In this step objects and operations are identified. Nouns are selected to form a list of objects, respectively verbs to form a list of operations [2]. Objects and operations lists are then combined into an *object-operation table*, and grouped to form HOOD-objects. In this step design decisions, especially regarding object type and exceptions mechanisms are justified.

4. **Formalisation of the solution:**
   This phase consists of refining the design, in particular *Object interfaces* and to describe formally with the BNF Objects and Operation Control Structures. *Object Description Skeletons* (ODS) are produced and later used as the basis for detailed design and coding phases. This last step is performed iteratively since each HOOD object is decomposed into smaller components until *terminal objects* are identified.

The scope of the thesis focuses on the final step and related design documents (ODS).

[1] However HOOD is oriented towards system development with Ada as implementation language. Therefore it is possible to produce easily Ada pseudo-code skeletons from HOOD diagrams.

[2] Note that Data Flow Control Flow Diagrams -DCFD(s)- and State Transition Diagrams -STD(s)- are another means of identifying objects and operations.

4.2 HOOD Features

In comparison to traditional design methods, since HOOD is an object-based method and therefore concepts of data and operations are the main strength of this approach. Notions of HOOD-ODS, object, operation and relationship are introduced in this section.

Object Description Skeleton - ODS. An ODS specifies the architecture of a system defining data-flows and functions of the program. It is described by a Program Definition Language (PDL) syntactically defined by a grammar (BNF form) and complies to some design rules defined in the HOOD reference manual [40]. An ODS is represented in a textual form from which a graphical form (HOOD diagram) may be derived automatically. It contains six main *sections* depicted in figure 4.1.

```
Object level description
Provided interface
Required  interface
Object  Behaviour Control  Structure -OBCS-
Internals
Operation Control Structure(s) -OPCS(s)-
```

Figure 4.1: Sections described in the ODS

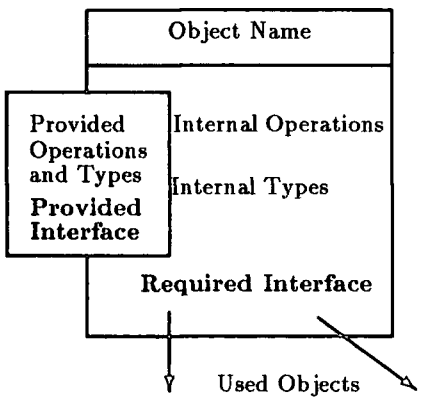

Figure 4.2: A Basic HOOD Object

Operations. Two types of operations are distinguished: *constrained* and *non-constrained.* A *constrained operation* [3] is triggered by an external event (e.g., interrupt or task call). Thus the execution of the operation depends either on the internal state of the object (e.g., guards on accept statements) or on the execution request.

Generally, in asynchronous communication, a process A (i.e., in HOOD it corresponds to the "execution of the operation") can send data to a process B and continue their execution without waiting for B to be ready to receive such data. On the opposite, in the synchronous case, two processes must reach pre-determined communication points in their flow of control to exchange information with each other. Several types of execution request are supported by the method [4] represented by the symbols: HSER, LSER, ASER and TOER triggers, which indicate that the client requesting process execution is suspended after its request:

- until full completion of the requested service (HSER- Highly Synchronous Execution Request),
- until completion of the requested processing by a server process (LSER- Loosely Synchronous Execution Request),
- not blocked at all (ASER- A Synchronous Execution Request). This kind of execution request corresponds to message passing communication protocols,
- or until run-out of a time delay (TOER- Timed Out Execution Request).

Objects. Objects are the basic units of modularity. An object is a collection of operations and types. There exists different kind of objects depending on the operations it contains and on the structure of the object. Concerning control-flow it is necessary to distinguish in HOOD control-flow and control of processing. Control-flow signifies that a flow carries out control information rather than just data (i.e., data-flow information) such as depending upon a state or event. Below, it is explained how active and passive objects deal with control-flow. On the other hand, in HOOD, the control of processing between active objects is processed by a specific structure called OBCS. There are passive objects, active objects, environment objects and classes.

- *Passive objects* are objects in which the control-flow is transferred from the *using* to the *used* object. It means that whenever a provided operation is executed the control-flow is transferred immediately to that operation. This corresponds to sequential processing.

[3] A constrained operation is annotated in a HOOD diagram by a zigzag arrow attached to it.
[4] A complete description is proposed by Robinson [74].

OBJECT ..is <Object_Type[3]>
PARAMETER[1] TYPES[2] OPERATIONS[4] CONSTANTS[4] --only for Class instances
DESCRIPTION
Text in natural language giving all information for understanding and maintaining the object.
This text may be structured according to documentation sections (H1,H2,H3) defined by HOOD
IMPLEMENTATION CONSTRAINTS
Natural language text giving hardware constraints (memory limits,cpu) for the object.
PROVIDED INTERFACE
TYPES --signature[4] in Ada, with associated informal description in natural language
CONSTANTS --signature in Ada, with associated informal description in natural language
OPERATION_SETS --list of set names
OPERATIONS[5] --signature in Ada, with associated informal description in natural language
EXCEPTIONS --signature in Ada, with associated informal description in natural language
REQUIRED INTERFACE
For each Required Object TYPES[6] CONSTANTS[6] OPERATIONS[6] EXCEPTIONS[6]
OBCS
DESCRIPTION--in natural language
CONSTRAINED OPERATIONS --Execution Requests on provided operations
DATA FLOWS --Textual Description of Data labels and direction along the graphical use relationship
EXCEPTION_FLOWS --Textual Description of Exception labels along the graphical use relationship
End of USER'S Manual of the Object

INTERNALS
OBJECTS[7]
TYPES[8] CONSTANTS[8] DATA[8] EXCEPTIONS[8]
OPERATIONS[10]
OBCS (for active terminal objects only)
PSEUDO_CODE[11] --suitables notation to express control
CODE --in target language
OPERATION CONTROL STRUCTURES (for terminal objects only)
For each OPCS[12]
Description
Used_Operations
Exceptions_Propagated[13]
Exceptions_Handled[14]
PSEUDO_CODE--in ADA PDL or PDL
CODE--in target language
END<Object_Name>

[3]Object types are CLASS,ENVIRONMENT,OP_CONTROL (operation), and VIRTUAL_NODE
[4]This fields exist and are only edited for CLASSES and their INSTANCES. They describe the formal parameters for classes and effective parameters for instances.
[4]Signature=syntactic definition. By default it is expressed in Ada syntax, whatever target language
[5]If such an item is itself member of a set, then its declaration is followed by the declaration "member of <set-Name>, thus it supports textually the definition of the set.
[6]The provider object is specified using the dotted notation, types and constants required for the definition of a signature of an operation, another type, a declaration of a data and/or for an instantiation. Operations used from server. Exceptions associated to these operations and provided by the server.
[7]This field is empty for terminal objects.
[8]These fields provide for definition of implementation of associated structures in terminal objects, and for defintion of implemented_by' relationship for non terminal ones.
[9]This field only exists for terminal objects.
[10]This field declares textually the "implemented-by" links of parent operations down to child ones. Additional internal operations (not shown on the graphical description) may be declared here in the process of step-wise refinement of operation control structures.
[11]The notation can be a graphical one or a textual one, allowing to specify formally control (Ada, Petri nets, Finite state Automata, Esterel, Temporal Logic) in order to allow for system dynamic verification, as defined by composition of OBCS along the use relationship. These notations should also provide for automatic generation of code. (see (Heitz, 92) for more details).
[12]These subfields are filled for each operation internal or provided.
[13]Exceptions which may be raised and propagated during execution of OPCS.
[14]Exceptions which are treated locally in the OPCS, and they can be reraised or not.

Figure 4.3: Ods Outline

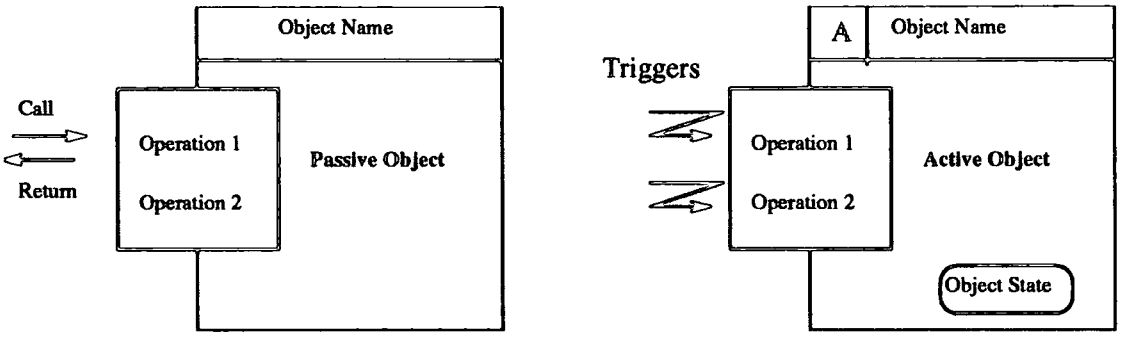

Figure 4.4: Passive and Active Objects

- *Active objects* [5] are objects in which the control-flow is not transferred. Such objects respond to the stimulus they receive according to their internal state defined in the OBCS. More specifically, an active object is an object that define at least one constrained operation.

- *Environment objects* are objects belonging to other systems, it means to another HOOD design tree and that can be referred through *use* relationships.

- Finally, *classes* are generic objects, amenable of instantiation.

The different types of objects are illustrated in chapter 6 presenting a case study.

Relationships. Different types of relationships exist between operations and objects. There are *use*, *include* and *implemented-by* relationships. *Use* relationships represent control-flow between objects [6] and refer to operations that are provided by the used objects. *Use* relationships refer to operations that are provided by the used object, but does not refer to other entities like types.
Include relationships concern the successive decomposition of objects into child-objects within the HOOD design process. *Implemented-by* relationships exist between operations. Operations of a parent object have to be *implemented-by* an operation of a child object. It means that whenever an object calls a parent operation, it actually calls the child operation.

[5] Active objects are distinguished in a HOOD diagram by an A in the top left hand corner of the related HOOD object.

[6] *Use relationships* are shown as a directed thick arrow between objects.

```
procedure OPCS_Start is
../..
-- Code
begin
 Timers-Driver.Init (Monitoring-Timer,Monitoring-Frequency, IT-1Hz-Address );
 Bargraphs.Init;
 Analog_display.Init_analog;
 Motor_sensors.Init;
 Timers-Driver.Start (Monitoring-Timer);
end OPCS_Start;

-- END_OPERATION OPCS_Start
```

Figure 4.5: OPCS for Operation controller Start: Code part only

4.3 Design Principles

HOOD enforces structuring of objects according to three principles, as follows : information hiding, control structuring and hierarchical decomposition.

Information Hiding. An object is defined by its external properties and internal structure that is hidden from other objects using it. HOOD encourages low coupling and high cohesion within objects by following Kafura's [47] design rules. One object has to *"see the minimum of the object it calls (fan-in)"* and has to *"show the minimum of its internal structure to its calling objects' (fan-out)"*. An object has a visible part (interface) and a hidden part (internals) that cannot be accessed directly by external objects. The interface defines services (types, constants, operations and exceptions) provided by the object (provided interface), as well as services required from other objects (required interface). This is depicted in figure 4.2. On HOOD diagrams only provided interfaces can be represented, while required interfaces are indicated through relationships between used objects and internals can be found in ODS documents.

Control Structuring. Generally, control-flow describes [43] for constrained operations:

- Sequential and parallel execution of operations
- Synchronous and asynchronous dynamical behaviour

HOOD supports those concepts and since it is used in real time systems, it isolates the expression of the reactive part of a system from its transformational

```
-- OBJECT_CONTROL_STRUCTURE
task OBCS_Ctrl_EMS is
 entry Start;   entry Stop;   entry Monitor;
 --for Monitor use at IT_1Hz_Address;
end OBCS_Ctrl_EMS;

task body OBCS_Ctrl_EMS is
begin
  loop
   loop
   select
      accept Start; OPCS_Start; exit;
   or accept Stop; -- empties Stop queue
   or accept Monitor; -- empties Monitor queue
   end select;
   end loop;      ../..
  end loop;
end OBCS_Ctrl_EMS;
```

Figure 4.6: OBCS of controller object: Code part only -abstract

part. The *OPCS* of an object exclusively describes the transformational semantics of an operation. Dynamical behavioural aspects [7] are described in the *OBCS*. In the *OBCS*, constructs are similar to Ada-language since it refers to control statements such as loops. Examples of OBCS and OPCS are given in figures 4.5, 4.5. More details are given in Appendix 3, Figures A 3.5 and A 3.14. The two types of objects defined previously handle control structuring differently:

- *Passive objects* do not have any semantics related to dynamical behaviour (i.e., to control flow). The control is transferred from the calling to the called object and the operation is carried out immediately. Passive objects contain operations, which can only be executed sequentially in a synchronous mode.

- In contrast, for *active objects* the execution of *provided constrained* operations is controlled by the *OBCS*. The control is not transferred and reaction to the stimulus must be serviced at a time determined by the internal state of the called object. Such operations are constrained in their execution according to either the internal state of the called object, or may be triggered by an

[7] Behavioural or Behaviour are mis-leading words since such concepts may describe various observable aspects/behaviors of the system. Moreover each behaviour requires a specific formalization. In this thesis, those concepts will refer to the HOOD-object dynamical behaviour, i.e. to the execution model of operations within one object.

external event (execution request). *Active objects* may operate simultaneously for several client-objects (i.e., calling objects).

Hierarchical Decomposition. This construction principle is supported by *include* and *use* relationships. HOOD defines terminal and non-terminal objects. *Terminal* objects define objects that can not be decomposed any further. Alternatively, *non-terminal objects* can be decomposed in several *child-objects*, which collectively provide the same functionality as the parent object. Moreover, an object using the parent object must also use at least one child object. In a HOOD diagram, this object is called *uncle object* and it connected to at least one child-objects through a *use* relationship. Associated arrows may have attached to it data- or exceptions-flows. An other principle for a correct hierarchical decomposition is that usually active objects should be place at the top of the hierarchy and passive objects at the bottom, in order to comply with Kafura's laws [47].

4.4 Types of Design Dependencies

Dependencies expressed in design artifacts may be classified into functional, data or control dependencies. The following classification corresponds to an analogy of the study at the code level (subsection 3.2.1). In footnotes, examples relevant to the code level are given.

1. **Functional dependencies:** [8]
   In HOOD, functional dependencies correspond to the *Include* relationship. An object can be decomposed into a set of *child* objects in an iterative process until all objects are primitive, it corresponds in HOOD terminology to *terminal* objects. Thus, a design is complete when all *parent* operations are carried out by child operations. This is defined by the *implemented-by* relationship. *Provides/requires* links belong to functional dependencies, because they describe services available in the system.

2. **Data dependencies:** [9]
   In HOOD, objects provide and require data types and operations. There are different types of data-flow, either in the same (*in*) or opposite direction (resp. *out*) of the *use* relationship between two objects or two operations [10].

[8] At the code level, functional dependencies correspond to call-tree or modules dependencies and also to data-flow dependencies between procedures.

[9] Code analysis of data dependencies consists of producing data-flow graphs or cross-references for temporary or persistent (files or variables declared statically) data.

[10] Data-flow corresponds to parameters of operations and is mapped into Ada with *in, out or in/out* parameters.

Dependencies / *HOOD Concepts*		Class 1 functional	Class 2 data	Class 3 control
Elements	Object	X		
	Operation		X	
	Interface	X		X
	Dataflow		X	
	OBCS			X
	OPCS	X		
Relations	Use			X
	Include	X		
	Implemented_by	X		

Table 4.1: Types of dependencies and HOOD concepts

3. **Control dependencies:** [11]
   In HOOD, control flow dependencies are expressed between objects linked by a **use relationship**. As previously explained in section 4.2, control-flow signifies that a flow carries out control information rather than just data. The control between active objects is processed by a specific structure called OBCS. The control flow interaction may be decomposed in two different ways. Firstly, when the OBCS is handled in one dedicated child object, then that must be an active object. Secondly, the OBCS may be handled in several child objects. Selecting between those two cases determines the resolution of **exception flow**, which occur if an abnormal return of control flow during execution of a provided operation. An exception propagates along the *use relationship* [12] from the operation where it raises to the *exception_handler* of the user object, executing the associated recovering code. Therefore exceptions propagate from the child to the parent operation, and then to objects *using* this parent-object [13].

Categories presented above are ordered in table 4.1. For each category, HOOD concepts (elements and relations) are listed. It shows that views of a design may be investigated independently since each view correspond to different concepts.

[11] At the code level, control dependencies correspond to the analysis of control flow structures (if-then-else or goto statements).

[12] In a HOOD diagram an exception flow is shown by a line crossing the *use relationship*.

[13] For the link of type *implemented_by* between HOOD objects exceptions are not shown in the graphical representation.

4.5 Summary

HOOD is a complete design method. The HOOD notation and language support designers in their task providing consistency checks of interfaces and defining artifacts to be produced. In this chapter, we have presented the structuring principles of HOOD and how they can be used for the purpose of traceability within software life-cycle artifacts, such as requirements, analysis, design or code documents. This facility should ease maintenance interventions, in particular an interconnection model supporting impact analysis would be beneficial in providing maintainers with tools to automate and perform changes.

The classification of design dependencies in HOOD, which is proposed helps in building the data-model to investigate artifacts. We classified dependencies into three categories, which are functional, data or control. For each category, HOOD concepts have been listed in order to show how the correspondance between HOOD features and the proposed classification. Indeed, the HOOD design method can describe both architectural, structural and dynamical aspects of a system. However, we deliberately restricted our study to architectural and structural aspects in order to master the problem. Investigating issues like related to real-time system would have possibly required a more complex data-model and approach which are presented in the next chapter.

Chapter 5

An Interconnection Model for HOOD

In this chapter an interconnection model to analyse dependencies in HOOD documents is proposed. It consists of a data-model, which maps concepts existing in HOOD and possible transformations of the design. The adequacy of the interconnection model depends on its ability to access fine-grained data and to trace propagation of changes. This approach is the contribution of the thesis to the field of impact analysis.

The considered approach and criteria to build a model are outlined in section 5.1. Product and process views of the modelling activity are presented in section 5.2 for the data-model and in section 5.3 for the activity model. The aim of the interconnection model is to support impact analysis checking of design consistency and propagation of changes resulting from design transformations. Heuristics for those transformations and formal descriptions are presented for horizontal propagation within HOOD artifacts in section 5.4.

5.1 Modelling HOOD

A software engineering approach to a problem entails proposing a model to systematize development of activities. This section defines criteria to validate our approach. A conceptual model is presented describing the rationale for modelling HOOD with an ERM representation.

5.1.1 Criteria for Modelling and Validation

The main objectives of the model are to perform *design and change analyses.* A system may be investigated for different purposes, but we focus our study on *checking design consistency* and validation of transformations. Thus, the rationale for Modelling HOOD is outlined. For any kind of software engineering activity, criteria for modelling and validation are concerned with the adequacy of the proposed solution to the given problem [1] and the complexity and level of the notation of the proposed solution-model. Specifically for our approach such criteria are stressed below.

Rationale for Modelling HOOD. HOOD is used to specify a system from architectural to detailed design phases. Emphasis is put on architectural (hierarchical decomposition, information hiding) and behavioural (control structuring) aspects [2]. Thus, a software system is analysed at a high level of abstraction (early phases of the life-cycle) and HOOD provides information at a coarse grain level (object interfaces mainly). Detailed analyses cannot be conducted by investigating only HOOD artifacts, which consist of graphical (HOOD-diagrams) or textual descriptions (ODS documents). Therefore, it is necessary to apply other techniques, which are able to consider finer grained elements, in particular data aspects.

Mapping Concepts. One of the criteria to build our interconnection model concerns its ability to *map concepts* of the real-world. Thus, this model must support validation of design rules and principles. HOOD determines rules, such as for operation usages or organization of objects. Design principles refer, for example, to laws presented by Kafura, which determine the correctness of design artifacts. The rational of the model refers also to types of transformations supported. In our study, functional changes (e.g., modification of the organization of objects) are considered.

Complexity of the Solution and Notation. Similarly, criteria concerning the solution proposed are crucial factors. It concerns the *complexity* of the solution model, in terms of types of relationships and entities existing, or heuristics for propagations. Another aspect relates to the granularity of the solution and the

[1] In other words, it corresponds to the ability of the solution-domain to map concepts of the real-world.

[2] HOOD design principles are presented in section 4.3.

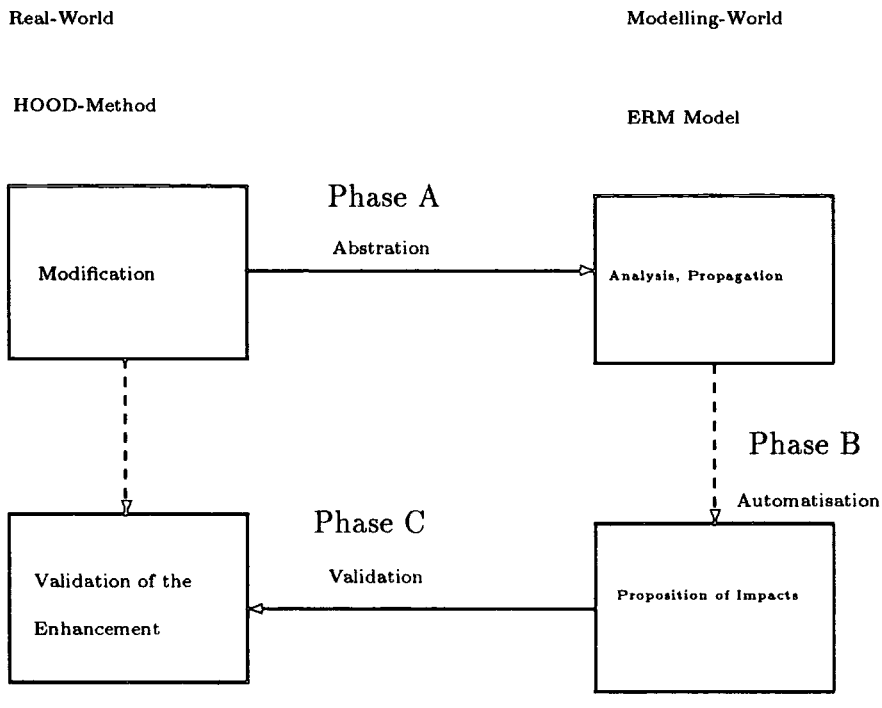

Figure 5.1: Modelling HOOD

support of accessing fine-grained data, such as a single attribute or a small piece of contents of an artifact. It also refers to the choice of the *notation*. It would have been possible to use different notations in the present interconnection-model. Since the Entity-Relationship-Model (ERM) described by Chen [18] in the early seventies seems to be the most appropriate model for data-oriented aspects of a system, we will adopt this notation [3]

5.1.2 A Conceptual Model

The purpose of impact analysis is to predict direct and indirect effects of a change. Our approach (figure 5.1) consists of transforming a problem expressed in the *real-world* (HOOD model) into the *modelling-world* (ERM model). Once this has been achieved, the impact propagation can be automated and finally results obtained can be "re-transformed" in the *real-world* to assess the modification.

- Phase A: Abstraction
  Modifications to be undertaken on the system, modelled with a HOOD design, are transformed into an ERM representation. The accuracy of such

[3]The ERM notation is ease to use and widespread among software engineers, and can be demonstrated to users without necessarily requiring prior software knowledge.

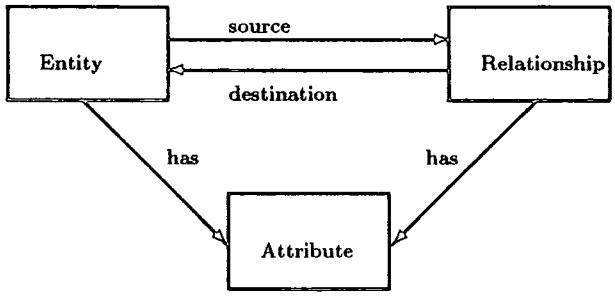

Figure 5.2: Semantics of an ERM representation

transformations depends on the completeness of mapping concepts from the HOOD-model to the ERM-model (syntax and semantics aspects) . Therefore a classification of design dependencies in HOOD is proposed (section 4.4). It points out three categories of dependencies: data, functional and control dependencies. In an ERM representation those three views can be supported.

- Phase B: Impact Propagation
  Since our model is used to perform impact analysis it must comply with syntax and semantics defined by the HOOD method. *Entities* and *relationships* have been built on the analysis of the Object Description Skeleton (ODS) document. Moreover, as presented in the next section, HOOD-rules to propagate changes have also been mapped into the model.

- Phase C: Validation
  Results of the impact analysis provide system designers and maintainers with useful information. In the *modelling-world* discrepancies or impacted elements are found and have to be expressed in the *real-world*. The validation of the model then consists in accurately reflecting the changed requirements.

5.2 A Data Model Expressed in ERM

This subsection explains how the HOOD data model has been extracted from the study of the design method, its construction rules and also its syntax which is presented in a *BNF form* [40]. Key points of our interconnection model are to map HOOD concepts and rules to control changes.

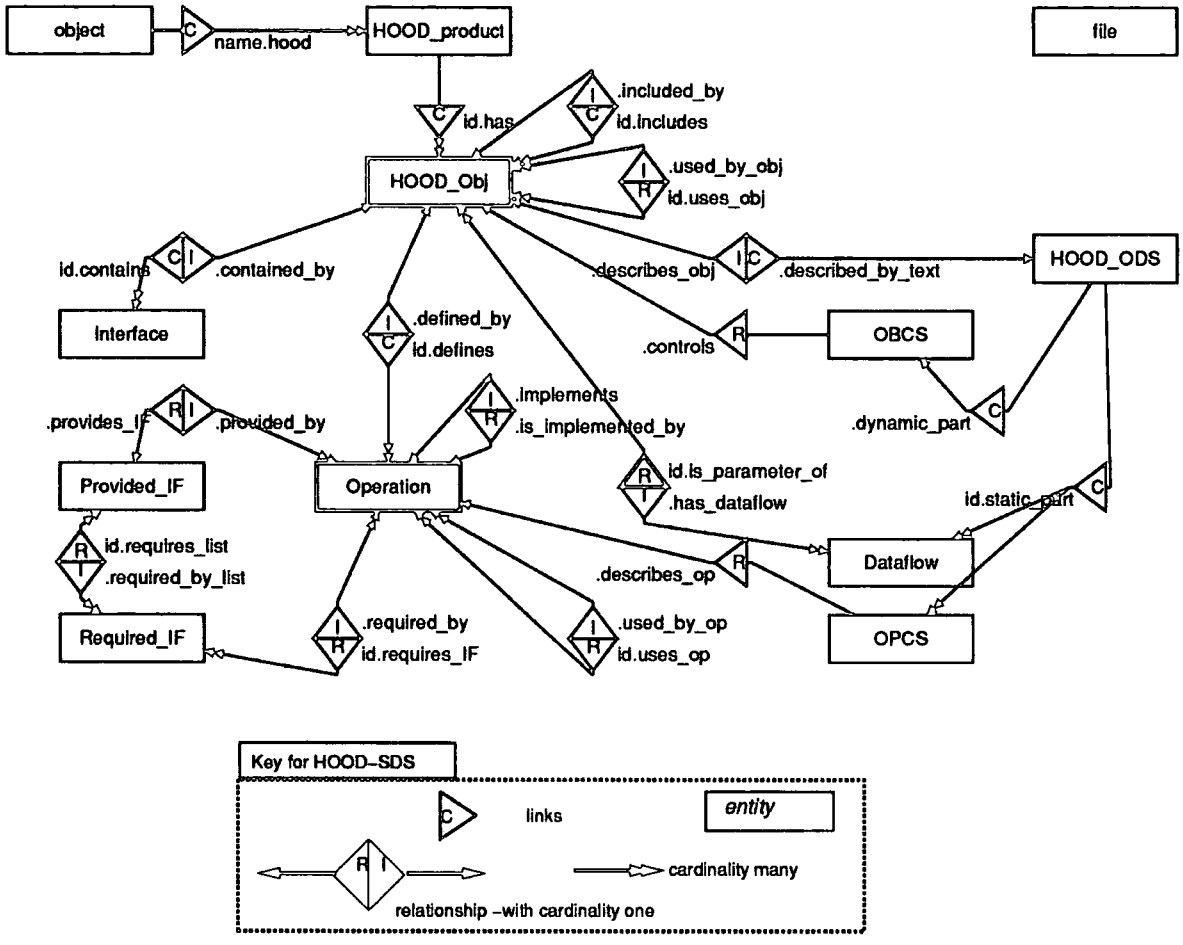

Figure 5.3: HOOD data-model

5.2.1 HOOD Concepts in ERM

The analysis of *sections* described in a design document (ODS) has formed the basis of building the data-model [33]. Therefore entities like ODS, Object, Operations, Interface (and its subtypes Provided_IF, Required_IF) OBCS, OPCS, Dataflow can be found. The data-model consists of entities, but also of relationships (i.e, links), which should support design principles described by the HOOD method. Therefore *use* and *include* for objects and *implemented-by* relationships for operations have been preserved and named accordingly. The ERM notation is illustrated on figure 5.2. Base on this notation, we propose a data-model [4] depicted in figure 5.3.

[4]In PCTE, ERM constructions are supported using SDS -Schema Definition Set.

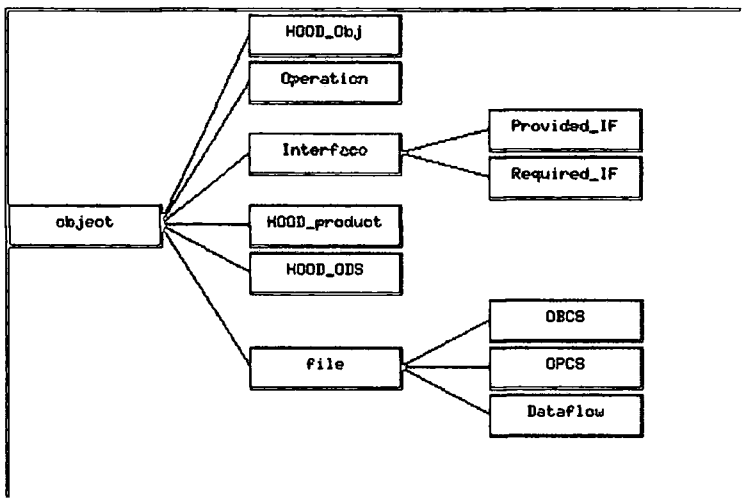

Figure 5.4: Subtree of HOOD data-model

To achieve information hiding, links to interface and the operation attribute *operation_type* have been designed. Control structuring principles are supported by the *OBCS* entity (for control-flow) and attributes *operation_status* for operations, respectively the attribute *object_status* for objects. Hierarchical decomposition of objects is supported by the relationship *include* and the attribute *object_type*.

In order to support propagations on the data-model several links have been added such as a relation between *provided* and *required* interfaces capturing the idea that there are complementary pairs of provide and require operations. Moreover the relationship *used_by/uses_op* between operations [5] supports fine grained propagation. In this interconnection model, *dataflow* has been expressed as a separate entity to ease for transformations on objects, 'redirection' of flows for each object. Finally, cardinality issues have been easily supported by the data-model by using the ERM notation (SDS tools in PCTE). For example, a single cardinality shows than an ODS may have several OPCS(s), but only one OBCS. Similarly, one interface is *contained_by* only one object when one object can *contain* several interfaces (one for each operation) according to our design.

5.2.2 Description of the HOOD Data Model

The model has been specified using the ERM notation proposed by Chen [18]. It consists of an ERM diagram represented in a graphical or textual form collecting

[5] Semantics of *used_by/uses_op* relationships for operations and *used_by_obj/uses_obj* relationships for objects are equivalent.

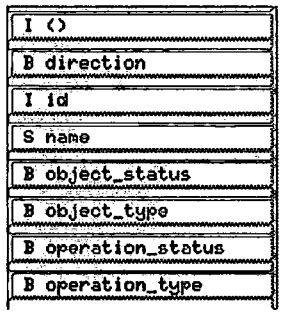

Figure 5.5: Attributes of HOOD data-model

entities and relationships. Entities are ordered into a hierarchy defined in a subtree (figure 5.4) and are 'qualified' by attributes (figure 5.5). The model contains ten types of entities and approximately twenty types of relationships. Therefore types of links and operation *attributes* have been carefully chosen with a set of *attributes* to 'qualify' an entity.

Following is a complete description of entities, relationships and attributes [6] contained in the data-model. *Object* and *Operation* entities, which are the central part of the proposed model are further detailed, and a textual description is given.

All diagrams, figures 5.3, 5.4 and 5.5 have been produced by PCTE-Emeraude. Therefore the syntax and semantics they convey are on some points different from the general ERM model. It is advised to ease the understanding of those figures to refer to the next subsection 5.2.3.

1. **ODS -Object Description Skeleton- and HOOD_product Entities**
   A system is composed of several objects, in particular a *root-object*. By definition, this object (entity HOOD_product) is decomposed further in objects so building the HOOD design tree. Each defined HOOD-object is described in a textual form contained in the ODS entity. The case study (chapter 6) consists of several ODSs, one for each design object described in the system -see appendix 3.

2. **Object Entity**
   This corresponds to a HOOD-object that is described in a textual form (ODS) and in a graphical form (HOOD-diagram). As it can be shown on figure 5.3, the current implementation of our data-model does not record HOOD-diagrams (objects of type graphics) in such in the database. Those elements are considered to be simply rebuilt from the ODSs (objects of type text) if necessary.

[6] For attributes possible instances are given in parenthesis.

```
HOOD_Obj : subtype of object ;
 Attributes
 name : string := "" ;
 object_type : boolean := false ; [terminal =1,  non-terminal =0]
 object_status : boolean := false ; [passive =1, active =0]

 Relationships
 includes      : composition link ( id ) to Many HOOD_Obj ;
 included_by   : implicit link to HOOD_Obj ; /*recursive relationship*/

 uses_obj      : reference link ( id ) to Many HOOD_Obj ;
 used_by_obj   : implicit link () to HOOD_Obj ; /*recursive relationship*/

 defines       : composition link ( id ) to Many Operation ;

 contains      : composition link ( id ) to Many Interface ;

 described_by_text : composition link to HOOD_ODS ;

end HOOD_Obj

Key for figure:
 [ ]           : instances of attributes
 ( ) to Many   : cardinality of links
```

Figure 5.6: HOOD Object Entity in Textual Form

An object is characterized by the operations it defines, its state and interface. The interface describes the visibility towards other objects according to construction rules. An object is denoted (i.e., de-referenced) by a name and is qualified by the attributes *object_status* and *object_type*. Finally, an object may *use* or *include* other objects, section 4.3 - Information Hiding.

3. **Operation Entity**
   Operations can be provided, required or internal to an object. This is supported by the attribute *operation_type*, which may have two instances, *internal or external.* Internal operations have been designed as operations *defined* by an object, but without any *provides_IF* links. The attribute *operation_status* implements the fact that operations are *constrained* or *non-constrained.* The static part of an operation is defined in the OPCS and its dynamic part in the OBCS (for constrained operation only). An operation may require for its

```
Operation : subtype of object ;
 Attributes
 name string := "" ;;
 operation_type : boolean := false ; [internal =1, external =0]
 operation_status boolean := false ; [constrained =1, non-constrained =0]

 Relationships
 uses_op     : reference link ( id ) to Many Operation ;
 used_by_op  : implicit link () to Operation   ; /*recursive relationship*/

 is_implemented_by : reference link to Operation ;
 implements        : implicit link to Operation ; /*recursive relationship*/

 provides_IF : reference link to Provided_IF ;

 requires_IF : reference link ( id ) to Many Required_IF ;

end Operation

Key for figure:
 [ ]          : instances of attributes
 ( ) to Many  : cardinality of links
```

Figure 5.7: Operation Entity in Textual Form

implementation operations provided by other used objects (*uses_by/uses_op* relationship). It may also be renamed if it is implemented by an other object (*is_implemented_by/implements* relationship [7]).

4. **OPCS -Operation Control Structure - Entity**
   Each operation of the object has an *operation control structure* (OPCS) defining in detail parameters and logic of the operation.

5. **OBCS -Object Behaviour Control Structure- Entity**
   This entity is defined for active objects only. An object may be controlled by one and only one *OBCS* (relationship *controls*). It is related to the internal state of the object, in particular for synchronization constraints and control sequencing (i.e., control-flow).

6. **Interfaces**

[7]The transformation *rename* in HOOD is mapped in our model by the relationship *is_implemented_by/implements*.

The data-model is designed so that to each provided operation of an object corresponds an interface. An interface defines the signature of an operation and an operation can provide an interface or require several interfaces. As shown on the diagram, figure 5.3 in the PCTE-ERM notation it corresponds to the Reference links, noted R. The provided interface of an object is the union of the provided interfaces of its operations, resp. the required interface of its required operations. On figure 5.3, this is depicted by the Composition links, noted C. To distinguish between provided and required interfaces for objects and operations, we designed two subtypes Provided_IF and Required_IF of the type *Interface.*

7. **Dataflow**
   An object is composed of in/out dataflows. A dataflow may be the parameter of several objects, reciprocally one object may have several dataflows. A dataflow must be linked to an object and not to an operation because by definition in HOOD it is accessed by dereferencing objects.

5.2.3 Benefits and Limitations of PCTE for the implementation

PCTE Notation for Entity Relationship Diagrams.

Figures 5.3, 5.4 and 5.5 are produced by the PCTE-tools plateform we used. PCTE supports the ERM notation. Thus, boxes correspond to entities (objects types) and connections to relationships. Concerning, *cardinality* different types can be represented in the graphical form of the data-model provided by PCTE-tools:

- *cardinality 1:M* (one-to-many relation), if the source entity is connected to many entities of the target object type,
- and *cardinality 1:1* (one-to-one) , if the source entity is connected to only one entity of the target object type.

Under PCTE-Emeraude, tools indicate the cardinality only on the graphical documents and not in the textual descriptions produced by the database. Thus, it has been added in order to improve the readability of the data-model. Similarly, for the instances of attributes which are is reported in brackets on figures 5.6 and 5.7.

Concerning the type of links available under PCTE, it exists three types, which are: **C** for composition, **R** for Reference and **I** for Implicit. Under PCTE relationships are bi-directional links. Several types of relationships can be used according to the semantics the designer puts into the diagram: **(R, I)** or **(C,I)**.

In PCTE entity may be ordered in a type hierarchy represented in a subtree, figure 5.4. As described below the types HOOD-object, Operation, Interface, HOOD-Product and HOOD-ODS inherit directly from the built-in PCTE type *object*. In the design of the PCTE database the PCTE type *file* is a subtype of *object*. The type *file* may record a content. This is the reason why the defined entities OBCS, OPCS and Dataflow are designed to inherit from the built-in type *file*. Inheritance links are only indicated by looking at the subtree and not in the HOOD data-model. However the type *file* is indicated in this last figure 5.3.

How to avoid implementation problems in PCTE improving the SDS

Savoia [75] presents a different SDS modelling HOOD. *Use* relations are designed as separated entities which connect either objects (*uses_by_obj/used* relation) or operations (*uses_by_op/used_op* relation). Such a model has the disadvantage firstly to convey a too fine granularity (which is not required), secondly to assume that semantically use relations between objects and operations are similar [8] and finally to input inconsistencies on the repository. Indeed ordered links cannot be supported by PCTE. It means that it is not possible to impose an order to the links of a given type that enumerate from an object. Hence this order information has to be maintained by tools the author proposed to use string attributes to store the key of the links in the proper order. This duplication of information is of course somehow a weakness of the implementation. Therefore our design proposes to record those *use* relations as links to the corresponding entities. The three weak points previously enumerated are then avoided.

5.2.4 Classification of HOOD Rules and Constraints

Different rules exist for objects, operations or on link types (e.g., *use* or *include* relationships between objects). Those rules cover syntactical and semantical aspects of the design method. Moreover, a set of rules can be expressed to support checking of design consistency. Our approach consists of classifying those rules into three categories (table 5.1 and Appendix 1 - HOOD rules) and of proposing for each category a tool supporting the checking mechanism on PCTE.

Class 1: This class describes syntactical constraints. For example, *rule o1* (Appendix 1) expresses that 'an operation is either *external* (i.e., is in the provided interface) or *internal*'.

Class 2: This class groups abstract and dynamic constraints such as *rule o16* 'an object cannot have both internal objects and *internal* operations'. This rule checks that if the object is *non-terminal* it does not define *internal* operations and reciprocally that if the object is *terminal* it does not *include child-objects.*

[8] This should be proved mathematically using for example commutative diagrams [46].

Category View	Class 1 Syntax	Class 2 Semantics	Class 3 Design consistency	Transformations
Conceptual View	ERM notation	semantics	semantics	heuristics, constraints
Environment View	PCTE repository	Hoodchecker	Hoodchecker	Hoodmodifier, Hoodchecker

Table 5.1: Types of constraints/transformations and related support

Class 3: This class corresponds to checking design consistency. For example, if *rule c5* is not verified an error message is given to designers indicating that the design must be changed. On the opposite, *rule c6* produces a warning since a design providing operations that are not required is a source of discrepancies.

Classifying rules into categories is useful in particular to undertake the development of several modular tools. These tools which have been developed in the context of this thesis are outlined in section 6.1.

5.3 An Activity Model for Impact Analysis

Activity models presented previously (section 2.2.2) have shown weaknesses in supporting activities for performing impact analysis. Therefore, we propose a new model described in this section and illustrate it with a case study (chapter 6). The proposed model helps us to understand and formalize different technical activities involved in impact analysis. It consists of four steps, namely: decomposition of the initial change, modelling the change, tracing the impact and assessing the impact. This activity model, depicted in figure 5.8, is an extension of the conceptual model since step 1 represents phase A, steps 2 and 3 corresponds to phase B and step 4 to phase C.

- Step 1: **Specifying the Change**
  The change to perform is explained and specified in a free text form called 'the change proposal'. It describes the dynamic aspects of the change, but does not specify it in terms of system's specification or code elements. This step verifies the accuracy and completeness of the information in the change request. When looking at the change request it should be determined whether the description is clear and concise. The change is translated into the terminology of the system (i.e., using terms defined by maintainers and understandable from the users) to minimize mis-understanding. Moreover expected effects of

the change, which may occur are also described. According to its granularity and complexity, the initial change may be decomposed into sub-changes, thus providing a better understanding of the problem. Each sub-change may be analysed separately.

- Step 2: **Modelling the Change**
  The change is matched to the data model (expressed in ERM), which describes the system. At the end of this step the initial set of items directly affected by the change are determined in terms of entities and links.

- Step 3: **Tracing the Impact**
  As a consequence of the initial change, unexpected ripple effects as changes in the system occur. The goal of this step is to detect them according to the type of propagation investigated. Different algorithms and heuristics for propagations may be proposed. Our study present for example one heuristic for propagating changes among entities of type 'HOOD-operation' in section 5.4.3.

- Step 4: **Assessing the Impact**
  Results on objects and relationships are translated back into the context of the original change. Moreover, the scope and complexity of the requested change are documented. This includes a description of affected software components (modules/units, configuration items, databases) and documentation. The complexity of the change is also reported, regarding the relationships between impacted components.

This activity model mainly addresses technical issues. It does only deal with one change at a time. Therefore, an extension of our method could be to group several changes together and to schedule a maintenance intervention once each change has been specified and modelled (step 2). To support steps of the activity model, the following section presents different types of design modifications in HOOD and related heuristics (steps 1-2-3). Finally, horizontal propagations and benefits of the model for assessing impacts are explained (step 4).

5.4 Horizontal Propagation

Objects and operations can be modified in many ways. Such types of functional changes are investigated in this section. Finally, the thesis focuses on one type of transformation, presenting in detail its heuristic and assessment.

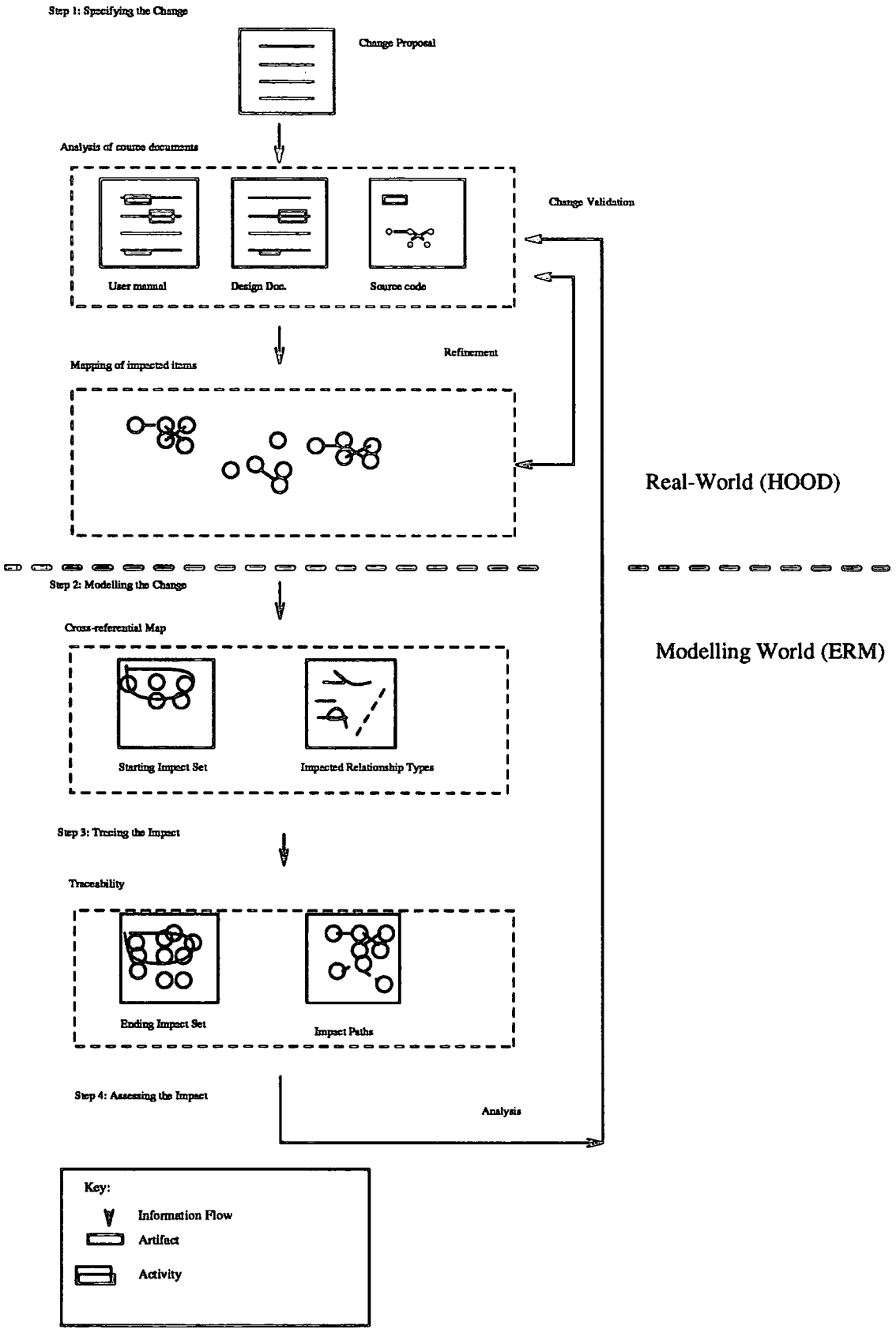

Figure 5.8: Impact analysis activity model

5.4.1 Design Checking and Types of Transformations

Our approach consists both within HOOD artifacts of checking consistency of the design to detect anomalies and also consecutively to a transformation to evaluate the possible impacts.

Design Checking. It corresponds for example, to determine the usage of operations. An operation may have different states, such as: defined, provided, required or used. Thus, the following anomalies or discrepancies may occur [9]:

1. Anomalies occur if an operation is required, but not provided, or not defined (un-defined states).
2. Discrepancies are detected if an operation is provided, but not required (un-referenced state), or if an operation is required but not used (un-used state).

Types of Transformations. Several types of transformations may be required for objects and operations.

- **Object Modification.** The level of objects in the hierarchy may be modified, such as by promoting or delegating objects. An object may be split into two objects, or two objects may be 'merged' or 'imported' in a single one. This last type of modification must preserve services defined in the interface and provided by initial objects. Two cases may appear. The two objects are 'merged' in a single one such that the interface of the resulting object is a folding of the two initial interfaces. We call this transformation *'merging objects'*. The signature (defined in the interface) and implementation parts (defined in the body) of operations are unchanged. In the second case, operations may be rewritten, which means that the signature of provided operations is not changed, but the body part of those operations is changed and encapsulated in the new object. Since for this transformation, operations defined in initial objects are changed, we call it *'importing objects'*.
- **Operation Modification.** Operations may be modified, added or removed. Similarly to objects, an operation may be split into two operations, or two operations provided by an object may be 'merged' in a single one.

Heuristics have been designed such as for merging objects or operations. This last type of transformation and the support provided by our tool *Hoodmodifier* developed on the PCTE platform are outlined in subsection 5.4.3. Those heuristics for describing transformations on HOOD artifacts are outlined below:

[9] It is advised to refer to subsection 3.2.2, Study on Variable Usages, for an analogy at the code level.

- **Merging Operations.** Two operations can be merged in a new resulting operation if they belong to the same terminal object. Those operations cannot be implemented_by other operations. Their attributes (*operation_status* and *operation_type*) must be of the same value otherwise an error or warning message arises.

- **Merging Objects.** Two terminal objects can be merged if they belong to the same parent object. Their attributes (*object_status* and *object_type*) must be of the same value otherwise an error or warning message arises. For those objects the former interfaces are folded into the new interface of the resulting object.

It should be noted that transformations where more semantics is required, such as split of operations or objects, promoting or delegating objects, can only be partially automated (i.e., close interaction with the user is necessary). Such modifications are beyond the scope of this thesis.

5.4.2 Assessing the Impact

This sub-section presents impact assessments illustrated by an example . In the following, a modification to an operation and consecutive direct and indirect impacts are described.

In HOOD, changes may be analysed at different levels. At a coarse level, the textual description, as defined in the ODS of the object to change, provides information concerning provided/required interfaces. At a finer grained level, looking at the internal implementation of objects -described in OPCS and OBCS structures- helps to trace changes at a detailed level. Different guidelines and constructs may be used to investigate the propagation of modifications, and is described below.

- Analysing **direct** impacts consists of checking operation attributes and modification of the object behaviour[10]. If an operation provided by an object is modified, it affects the object containing it. For example, adding an operation of type *constrained* to an object changes its dynamic behaviour (i.e, defined in the OBCS). If this object was previously *passive*, it becomes an *active* object. HOOD rules also check *use* relationships between objects to avoid cycles and also to respect dynamical aspects of the design (i.e, an active object shall not be used by passive objects).

[10]According to HOOD it defines the execution model of the operations defined in an object, i.e. the control-flow.

- Analysing **indirect** impacts consists of analysing operations by using the modified operation. Different types of propagation may be used to investigate modifications. In this thesis, we use an *object-operation* cross-table, which is issued from the analysis of provided/required interfaces and of OPCSs structures, indicating *used/using* operations (Appendix, Figures A 3.6, 3.7, 3.13). The analysis of OPCSs determines provided/required pairs. Firstly, through the analysis of this table it is possible to trace objects affected by the change and to find out connections between operations and therefore between objects. This reproduces the 'call-graph' of operations. The table matches the calling operations (i.e., callees) displayed on the rows and the operations called on in the columns. Anomalies presented in previous sub-sections are detected looking at the usage of operations.

5.4.3 Heuristic for the Transformation 'Merging operations'

A heuristic has been designed, which consists of verifying pre-conditions to perform the transformation 'Merging operations'. These conditions are mathematically described, using **set theory** and are depicted in figure 5.9. Once the transformation has been done, post-conditions should be satisfied.

Pre-conditions.
-Let op_i and op_j be two operations defined by the same object (obj).
-Let S_i and S_j (resp. S'_i and S'_j) be the sets of operations having *incoming* [11] (resp. *outgoing* [12]) links with op_i and op_j respectively.
Then, for the transformation 'merging operations' (symbol: $\vee$) between op_i and op_j pre-conditions are defined as follows: op_i **and** op_j can be merged into a new operation $op_i \vee op_j$ **IFF**
- op_i and op_j are *defined* by the same *terminal* object [constraint on *defines* link and rule 016],
- op_i and op_j have no *is_implemented_by* link [13] [rule 010],
- op_i and op_j have identical values for attributes *operation_status* and *operation_type*.

Steps for the Transformation 'merging operation'. This consists of the three following steps:

Step 1 : **Update of entity attributes.** Attributes values of $op_i \vee op_j$ are defined

[11] For operations, incoming links refers to *used_by_op* and *required_by* links.
[12] For operations, outgoing links concerns *uses_op* and *requires_IF* links.
[13] For this transformation *is_implemented_by/implements* links do not apply to operations since the object must be *terminal*.

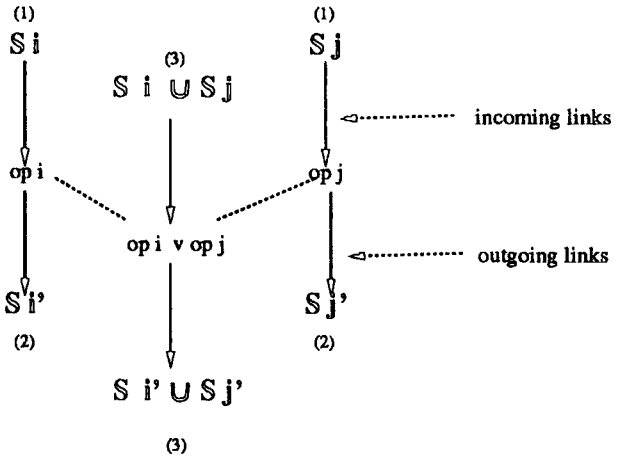

Figure 5.9: Definition of constraints on incoming/outgoing links

as follows: for *operation_status* if op_i and op_j are both *non-constrained* [14] then the corresponding attribute value for $op_i \vee op_j$ is *non-constrained* else it is *constrained.* Similarly, for *operation_type* attribute if op_i and op_j have the same *operation_type* value then for $op_i \vee op_j$ this attribute also has the same value.

Update of links. For the new entity $op_i \vee op_j$ of type operation corresponding relationships are created: between *operation* and *object* types (*defined_by/defines* relationships) and between *operation* and *interface* types (*provides_IF/provided_by* relationships).

Step 2 : **'Redirecting' incoming/outgoing links.** Links connected to the former entities op_i and op_j will be replaced by links to the new entity $op_i \vee op_j$. Beyond pre-conditions, at this stage **intermediate conditions** must be satisfied during the transformation. It consists of constraints to avoid duplication of *incoming* and *outgoing* links and constraints to avoid cycles between operations [15].

Definition of intermediate conditions on incoming/outgoing links

$op_i \vee op_j$ has *incoming* links in $S_i \cup S_j$ and has *outgoing* links in $S'_i \cup S'_j$
iff $\nexists op_k$ so that op_k has *incoming* links in $S_i \cup S_j$
and op_k has *outgoing* links in $S'_i \cup S'_j$ *constraint* (3)

In particular, $op_k \vee op_k = op_k$ if op_k has *incoming* links in $S_i \cap S_j$ (1) (resp. if

[14] Notions of *constrained/non-constrained* operations are defined in section 4.2.

[15] Duplications of *incoming* (resp., *outgoing*) links refer to constraint (1) (resp., constraint (2)) and avoiding cycles refers to constraint (3).

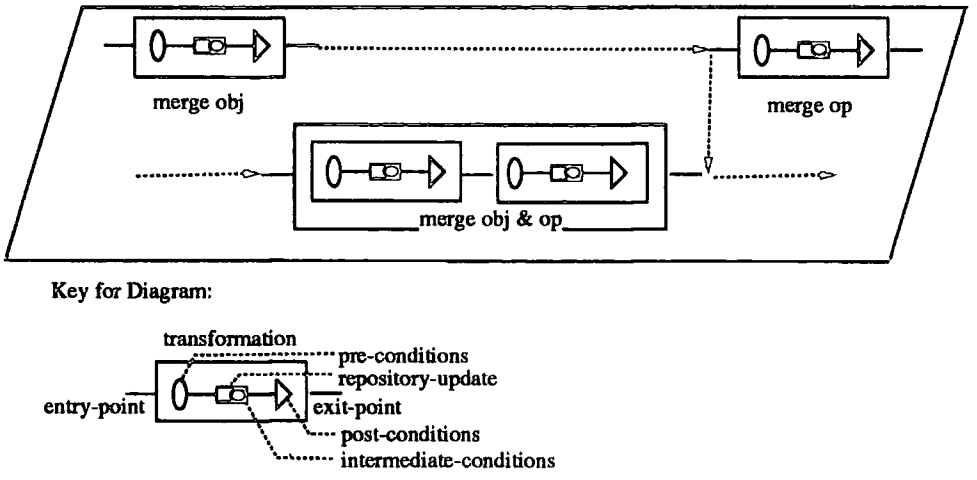

Figure 5.10: Network of transformations

op_k has *outgoing* links in $S'_i \cap S'_j$ constraint (2)). Note that for constraint *3* if $S_j = S'_j = \emptyset$, $S_i \cap S'_i = \emptyset$ holds. □.

Step 3: **Remove former entities.** Entities of type operation (op_i and op_j) and corresponding relationships (same as for step 1) are deleted from the object-base.

Post-conditions. Those conditions must be satisfied on completion of the transformation. Respectively, the transformation is **valid IFF** HOOD rules (Appendix 1) are still verified in particular for objects (rules C1, C7), interfaces (rules C4, C8) and operations (rules C5, C6, C9). *Intermediate conditions* are reported in the *post-conditions* (it concerns C11 for operations).

In conclusion, it is necessary to recall that pre-conditions must be fulfilled to perform a transformation. Moreover, our heuristic consists of intermediate conditions that controls possible cycles between operations due to the merging of links. Post-conditions which should be satisfied have been listed. The heuristic for 'Merging objects' is similar to the heuristic for 'Merging operations' and figure 5.10 illustrates how such transformations can be combined in a network.

5.5 Summary

In this chapter, a conceptual framework is presented to support checks in HOOD based designs and describes a way to keep HOOD diagrams in agreement during edit-

ing. Several heuristics describing possible transformation have been designed such a for merging HOOD objects or HOOD operations. However, other transformations possible in a HOOD document cannot be supported yet by our tool development since they would require a much complex underlined model, in particular concerning the data-model.

For the purpose of tracing relationships among design elements, we proposed an interconnection model defined using the Entity Relationship notation. This model extracted from the study of conventional HOOD documents (HOOD ODS and HOOD diagrams) has been designed so that its complexity was easy to master and that it really could map concepts of the real-world, i.e. concepts present in HOOD designs. The aim of our study is to control and predict ripple effects during the change process. Therefore an activity model, which supports the process and product view of impact analysis activities is outlined.

Thus, a user who has to develop and maintain HOOD documents could pick-up the data-model and the activity model we proposed to support in his/her work. In the next chapter, we present indeed a case study to illustrate how this can be performed. HOOD diagrams can be recorded in the PCTE object base thanks to the ERM notation we choose and that is supported by PCTE. This repository is then helpful supporting the tracing among elements of the object-base consecutively to a transformation.

Chapter 6

Case study for HOOD

This chapter illustrates the interconnection model presented previously. Section 6.1 outlines the tool development conducted to implement and validate our approach using a repository called PCTE (Portable Common Tool Environment). A series of tools is proposed to support the users (i.e., the maintainers) in this task to conduct design analysis and design transformations. Section 6.2 describes the Aircraft Engine Monitoring System issued from a large industrial project. Finally, in section 6.3 several types of changes are performed and results provided by the tools are explained.

6.1 Tools Support for Horizontal Propagation

Our approach consists of expressing a set of constraints at the design level (subsection 5.2) and of detecting errors/warnings corresponding to violations of those constraints. Rules between design elements -Appendix 1- have been expressed that can be used either to check design consistency or for a given modification to detect direct and indirect changes consequently induced. Our approach has been validated under a PCTE environment. The following paragraph is a brief introduction to PCTE, while a complete description can be found in Bancilhon [9]. Benefits of PCTE and comparisons with other environments are also outlined.

Introduction to PCTE

The objectives of PCTE [26] are to implement basic utilities and working prototypes of a Portable Common Tool Environment (PCTE) to support tool development. PCTE has been developed since 1983 in the context of Esprit projects by large software companies such as Bull, Siemens-Nixdorf, GEC, ICL and Olivetti. Various prototypes of PCTE functionalities are available on the market. For our

study, we used PCTE-Emeraude on a Sun Sparc workstation. The Object Management System (OMS) of PCTE implements the Entity Relationship Model (ERM) so that: entities are represented by typed objects ordered in a hierarchy and that relationships are represented by bi-directional links. Objects may have a content (if they inherit from the object type *file*). PCTE is a repository, which is different from a relational data-base in the sense that the contents of objects have no semantics. Another difference concerns the support of a version management system and the possibility of defining several types of objects and relationships. PCTE is composed of two layers, namely the *meta-base* and the *object-base*. The *meta-base* defines types of objects and relationships in a Schema Definition Set (SDS). Each tool has a specific view of the *object-base* through the *working schema*, which is a set of SDS. Objects in PCTE are identified through their access path (i.e., Unix pathnames). SDS are interpreted and not compiled. Thus modifying a SDS limits the impact on the actual repository. Moreover, it provides a better integration between tools accessing the same data-base.

Supporting Design Consistency Using PCTE Tools

To support such activities, tools can be used. The *SDS* tool easily supports rules defined in the first category of constraints (table 5.1) since it corresponds to the syntax of links [1]. However concerning constraints defined in categories (2) and (3), it has been necessary to develop a tool under PCTE. This environment supports several languages including Ada [30], C [29] and C++ [27]. We have implemented a prototype tool in C++ checking those rules, that we therefore called **Hoodchecker**. A second tool called **Hoodmodifier** and described below has also been implemented for this thesis.

Benefits of Using PCTE to Support Design Transformations

The current version of the Hoodmodifier prototype supports under PCTE several In table 5.1, the conceptual and environment views of methods and of the developed tools are depicted. For each category of constraints/transformations related tool support has been indicated. The following paragraph details how tools may support the checking of conditions and valid the transformation.

A design transformation is valid *iff* pre- and post-conditions fixed by the heuristic of the transformation are verified. Our tool development on PCTE has shown that these conditions can be checked automatically on the PCTE repository. Similarly, the heuristic presented in section 5.4.3 and composed of three steps can be automated. For example, the steps for the transformation 'merging operation' consists of sub-steps: sub-step 1 -Update of entity attributes, sub-step 2 -'Redirecting' incoming/outgoing links, and sub-step 3 -Remove former entities.

[1] For example, cardinality aspects are ensured via the category of links and existence of an *implicit* reverse links.

If the transformation involves the user in step 1 for the definition of entities and links for new entities [2], the rest of the activity has been designed to be supported by tools. Thus in step 1 update of attributes is automatically performed. In step 2 'redirecting' links is performed on the repository by *Hoodmodifier* verifying intermediate conditions (figure 5.9). Finally, step 3 (removal of former entities and links) also updates the repository without requiring any user interaction.

Limitations of PCTE to Support our Data-Model and Transformations

For our tool development and implementation of the data-model, we used the plateform called PCTE-Emeraude, which is based on the PCTE version 1.5. One of the limitation concerns the modifications of the key for links without removing the object. Once instances of links are removed from the database the key is not re-calculated . This requirement is not supported by the ECMA-149 standard (2nd edition-June 1993), neither implemented in PCTE Emeraude. Such a lack could possibly lead to inconsistencies on the repository.

Moreover, since we did only have PCTE-Emeraude available and no other plateform implementing fully ECMA-PCTE requirements, we had to limit the flexibility and extensibility of our data model. Indeed, conceptually ECMA-PCTE allows more types of links than those available on PCTE-Emeraude. It is possible to define other categories than Composition (C), Reference (R), or Implicit (I), namely Existence (E) and Designation (D). It also allows multiple inheritance of parent types.

6.2 The Aircraft Engine Monitoring System

This section presents a case study conducted to validate our approach experimentally. Criteria for selecting an adequate example are explained and the case study is described.

6.2.1 Criteria for a Case Study

Different reasons have lead to the choice of the engine case study. Firstly, HOOD strengths are in modelling functional, hierarchical and behavioural [3] aspects of a software system. Our method addresses structural changes. Therefore the chosen case study has to handle a system on the functional point of view and different changes have to be performed such as functional changes (operations modifications)

[2] In the current implementation, the user inserts those elements in the repository via the tool *oms_browser*.

[3] Behavioural aspects refer to real-time issues such as task handling or exception resolution. It is defined in the OBCS structure.

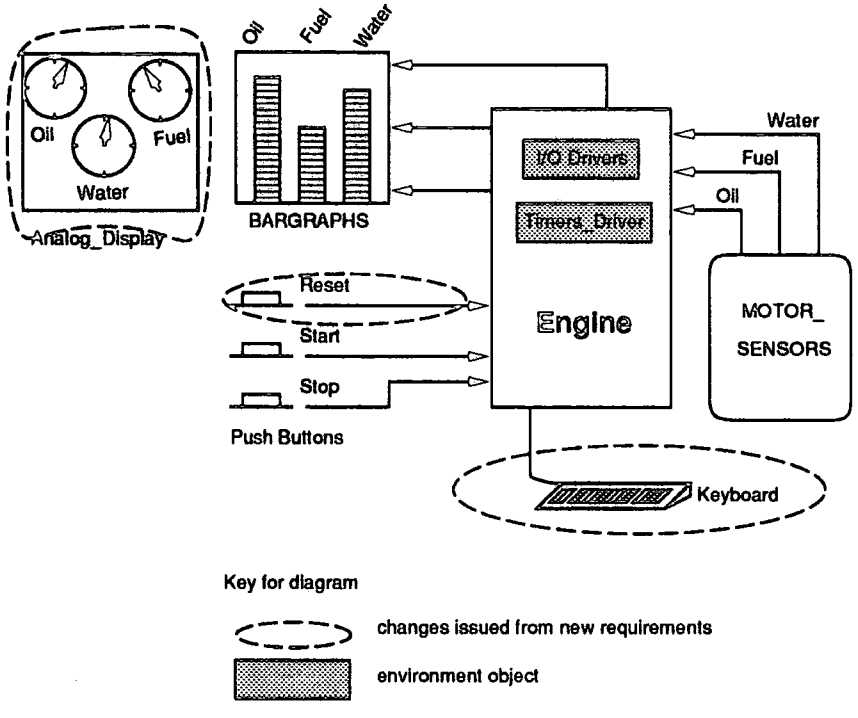

Figure 6.1: Aircraft Engine Monitoring System: Description

or architectural changes (modifications of object hierarchy). Secondly, to validate the approach the example has to be simple to master and complete. The engine system handles only few functionalities and represents an existing complete system.

In 1981, the DTI [4] issued a report [25] the purpose of which was to give guidance to practitioners of system design in Ada. The study examined in depth four system development methodologies applied to the same problem, that of designing an aircraft monitoring system. This DTI report has been made public through some research publications [73]. For this thesis, we based our case study on this example, adding our perspectives to a (simplified) system dealing with a generic engine system.

[4]The DTI is the British Department of Trade and Industry.

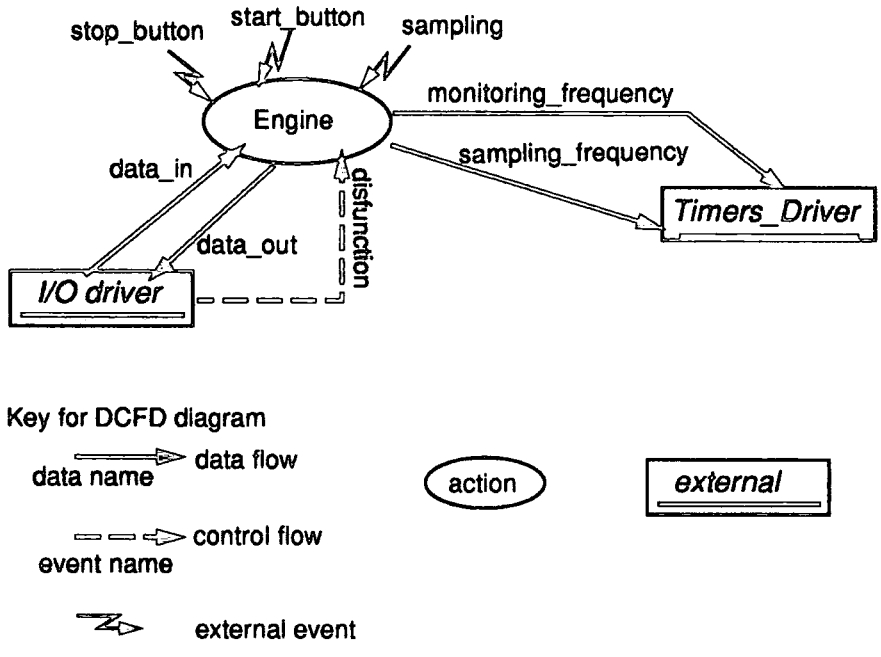

Figure 6.2: Aircraft Engine Monitoring System: Context Diagram (bef. change)

6.2.2 Case Study Description

System Description

The *engine* system monitors an aircraft and has different inputs [5] and outputs [6]. The *engine* uses also environment drivers Input_Output and Timers_Drivers. For example, sensors are sampled by a signal sent from the timers at a precise frequency. Pushbuttons are used to start and stop the engine. The system is depicted on figure 6.1.

System Before Changes

The engine system is analysed before modifications. Analysis and design documents have been produced by the development team Data Flow Control Flow Diagrams -DCFDs- and HOOD diagrams (figures 6.2, 6.3 and 6.4). It shows the objects composing the engine system represented at three levels of hierarchy:

- a parent object *engine*,

- decomposed in children objects *Controller*, *Bargraphs* and *Motor_Sensors*,

[5] Input hardware interfaces correspond to *start, stop and reset pushbuttons.*

[6] Output hardware interfaces are represented by *bargraphs* and *analog_Display* objects.

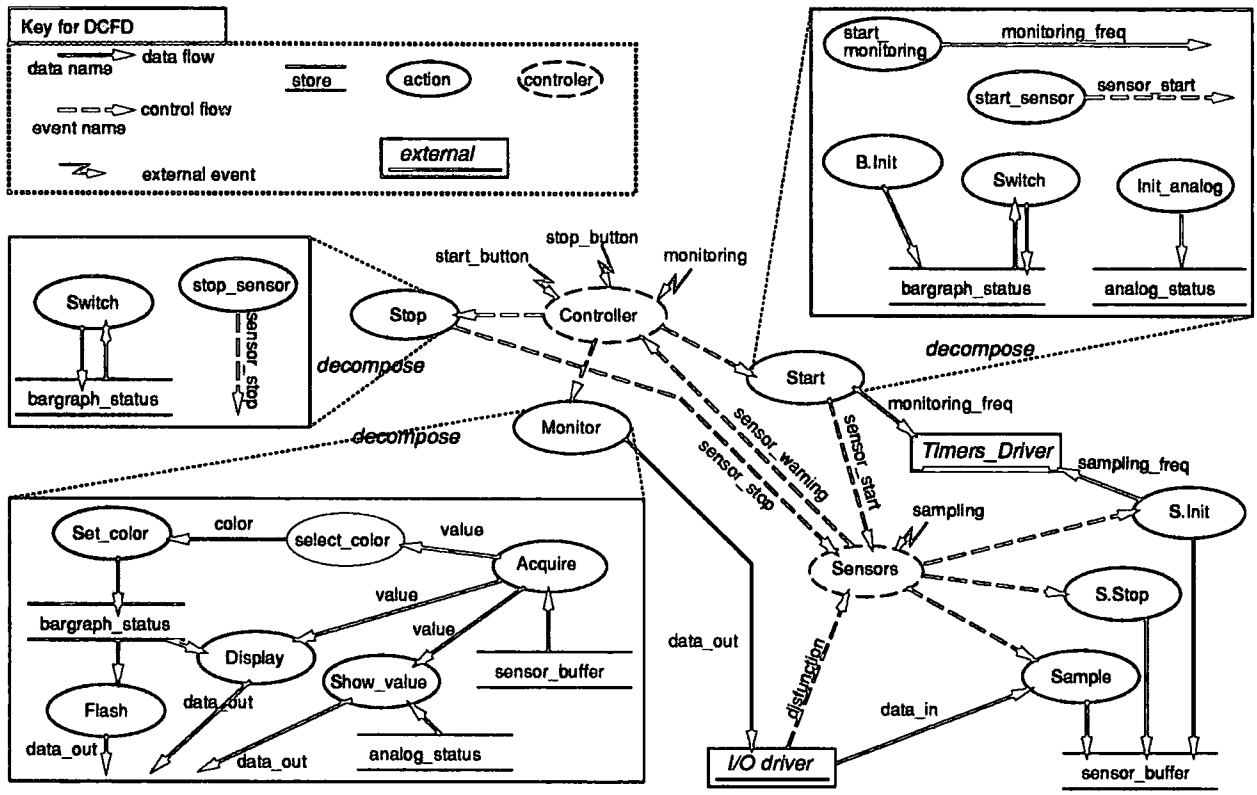

Figure 6.3: Aircraft Engine Monitoring System: DCFD (bef. change)

- using services provided by *environment* objects *Timers_Driver* and *IO_Driver*.

The *Controller* object is an *active* object, which has been designed to provide a number of operations *start, stop, monitor*. It requires several operations (*init, display, ...*) and has internal operations. The object controls the acquisition and display of values. To this end it uses the following objects: *Motor_Sensors* object, which samples [7] the engine values (fuel, oil, water); *Bargraphs* [8], which displays values on a Bargraph in normal mode, except if an error occurs (e.g., sensors disfunction) switching then the displayed values to a red flashing mode. Since environment objects refer to hardware components (i.e., handling interruptions) they are designed as *active* objects.
Although the proposed model is simple, in total the ERM representation of the system is composed of over 150 links and 50 ERM-entities collected in the data-base (i.e., in the implementation system under PCTE an entity refers to a PCTE object).

[7] The *samples* operation is *constrained* and executed when the signal *sampling_frequency* arises.
[8] This object is *passive* providing only *non-constrained* operations.

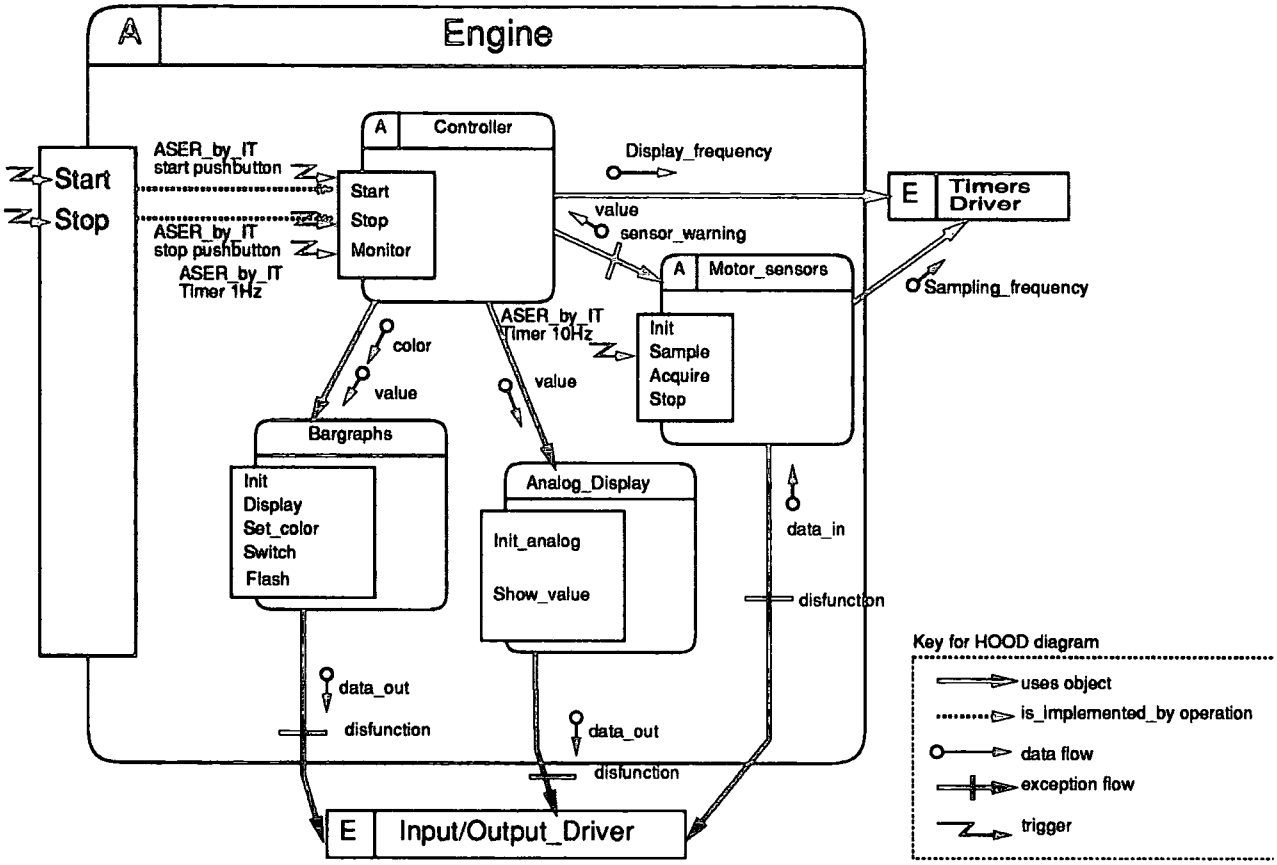

Figure 6.4: Aircraft Engine Monitoring System HOOD Design (bef. change)

6.3 Investigation of Changes

This section describes different changes investigated with the aircraft engine monitoring system (subsection 6.3.1). At the design level, impact analysis can be used for two purposes, namely : checking the design consistency (i.e., before any changes a system is statically verified) or validating the transformations. This is outlined in subsection 6.3.2, respectively subsection 6.3.3.

6.3.1 Case Study and User Change Requests

The initial *engine* system is transformed and several changes are performed on objects and operations. Initially, the Bargraphs object defines the operations *Init, Switch, Set_color, Flash* and *Display.* Followings changes are performed on objects or on operations.

Nb	Type of transformation	set of operations before	set of operations after
1	merge_op, merge_obj	Init + Init_analog	Init
2	split_op	Switch	Switch_On, Switch_Off
3	merge_op	Set_color + Flash	Set_color_Flash
4	preserve_op	Display	Display -
5	adding_new_op	—	clearscreen
6	merge_obj	Show_value	show_value

Table 6.1: Transformations applied on operations defined by Bargraphs object

- A new constrained operation *reset* [9] is added to the parent object *Engine*. It results by adding the operation *clearscreen* to the Bargraphs object [10].

- Bargraphs and Analog_display are merged into a single object called *Analog_Display*.

- For the Bargraphs object, operations are merged or splited. For example, *Set_color* and *Flash* operations are merged in a single operation called *Set_color_Flash*. On the other hand, *Switch* operation is split into *Switch_On* and *Switch_Off* operations. Display operation is preserved, in the sense that it is not changed. Those transformations are summarized for the Bargraphs object in table 6.1. Operations that result of the 'merging' of *Bargraphs* and *Analog_display* objects are underlined in table.

Note that by definition of our heuristics for the 'merging' of operations defined by different objects is not allowed. However, following the merging of two objects corresponding operations can thus be merged. This is the case for the operations *Init_analog* and *Show_value* defined by *Analog_display*. As a result, to the merging transformation of *Bargraphs* and *Analog_display*, operations *Init* and *Init_analog* can be merged into a single operation (transformation #1, table 6.1).

System After the Changes

After the changes have been carried out Bargraphs defines the following set of operations: *Init, Switch_On, Switch_Off, Set_color_Flash, Display, Clearscreen and Show_value*. Updated versions of analysis and design artifacts describing the system are produced. It consists of DCFD and HOOD diagrams (figures 6.5 and 6.6).

[9] Execution mode is ASER - A Synchronous Execution Request.

[10] Clearscreen implements the functionality of the parent operation *reset*.

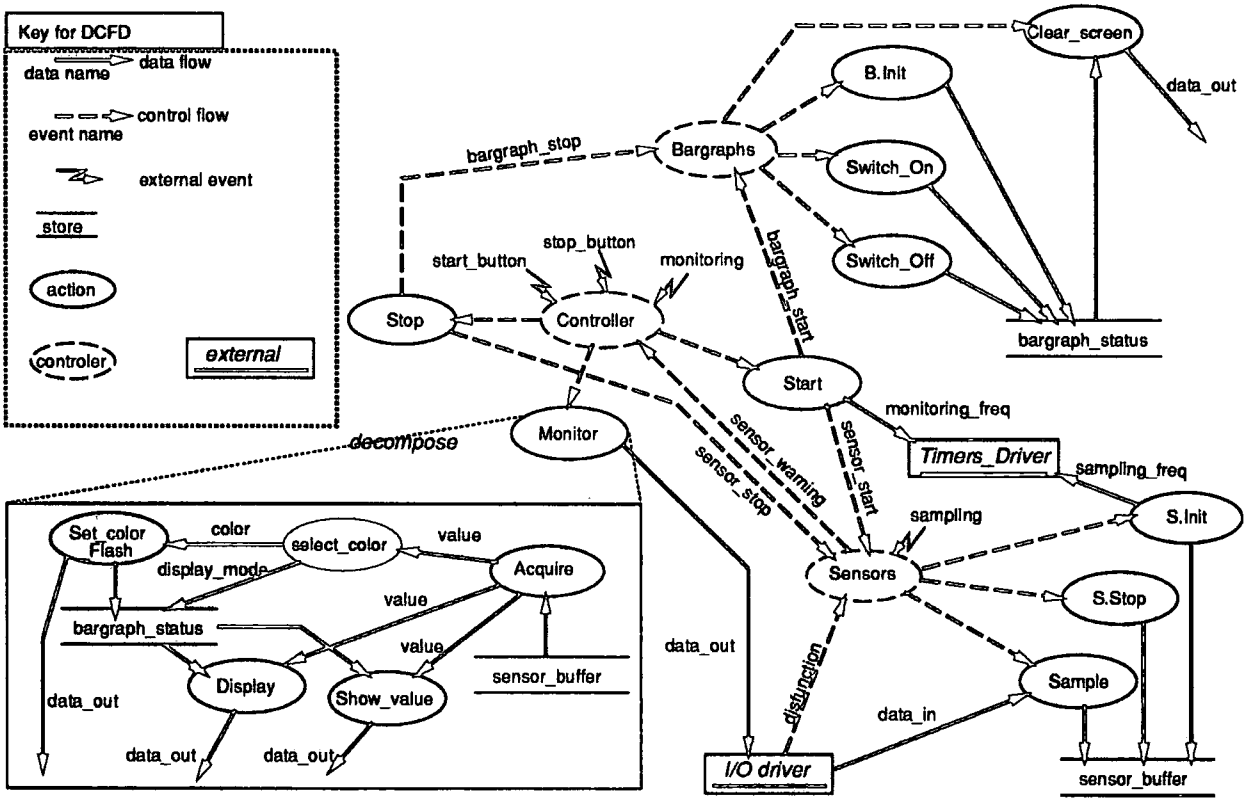

Figure 6.5: Aircraft Engine Monitoring System: DCFD (after change)

6.3.2 Validation of Design Consistency

Initial design artifacts of the engine system, which are produced in the development phase are useful to maintainers. For reverse engineering or maintenance purposes, maintainers can undertake an analysis of design consistency. As Bennett [12] argues, such cases of *preventive* maintenance improve the quality of the system, as well as easing future maintenance actions, whether corrective, adaptive or perfective.
The original engine system included inconsistencies that were detected by the *Hood-checker* tool. In particular, the analysis of interfaces (provides/required pairs) has shown that some operations had been required but not provided (this corresponds to an error) and that operations had been provided but not required (this corresponds to a warning). With respects to concerning parent operations, errors have been pointed out, such operations having no link to implementation (i.e., to any child-object). By construction principles, a *parent* object should be decomposed into a set of *child-objects*, which collectively should provide the same functionality as the parent. This anomaly induced an error in the dynamic behaviour of the design.

6.3.3 Validation of Transformations on PCTE

Several changes have been listed in section 6.3.1, in particular for the Bargraphs object in table 6.1. This section investigates two changes, which corresponds to the transformations #3 and #5. The other changes are not explained since they are either similar or being not supported yet by our tools.

Example 1: **Merging Operations**

'Merging' operations consists of 'transforming' several operations provided by an object in a single operation. Results and support given by our prototypes Hood-modifier and Hoodchecker (subsection 6.1) are presented below.

- Step 1: **Specifying the Change**. In the engine system two operations *Set_color* and *Flash* provided by Bargraphs object are 'merged' in a single operation *Set_color_Flash* (transformation #3, table 6.1).
- Step 2: **Modelling the Change**. This consists of identifying entities and relationships involved in the modification. This transformation involve entities of types operations, but also indirectly attributes of objects entities defining the operations to be merged. Entities concerned by this change are depicted in figures A 3.1 for the entity Bargraph and figure A 3.2 for the entity Controller.
- Step 3: **Tracing the Impact**. Conditions are analysed in detail for this step.

  - *Pre-conditions* are full-filled. The HOOD diagram before the transformation (figure 6.4). mainly shows impacts at a coarse grain level on objects, but

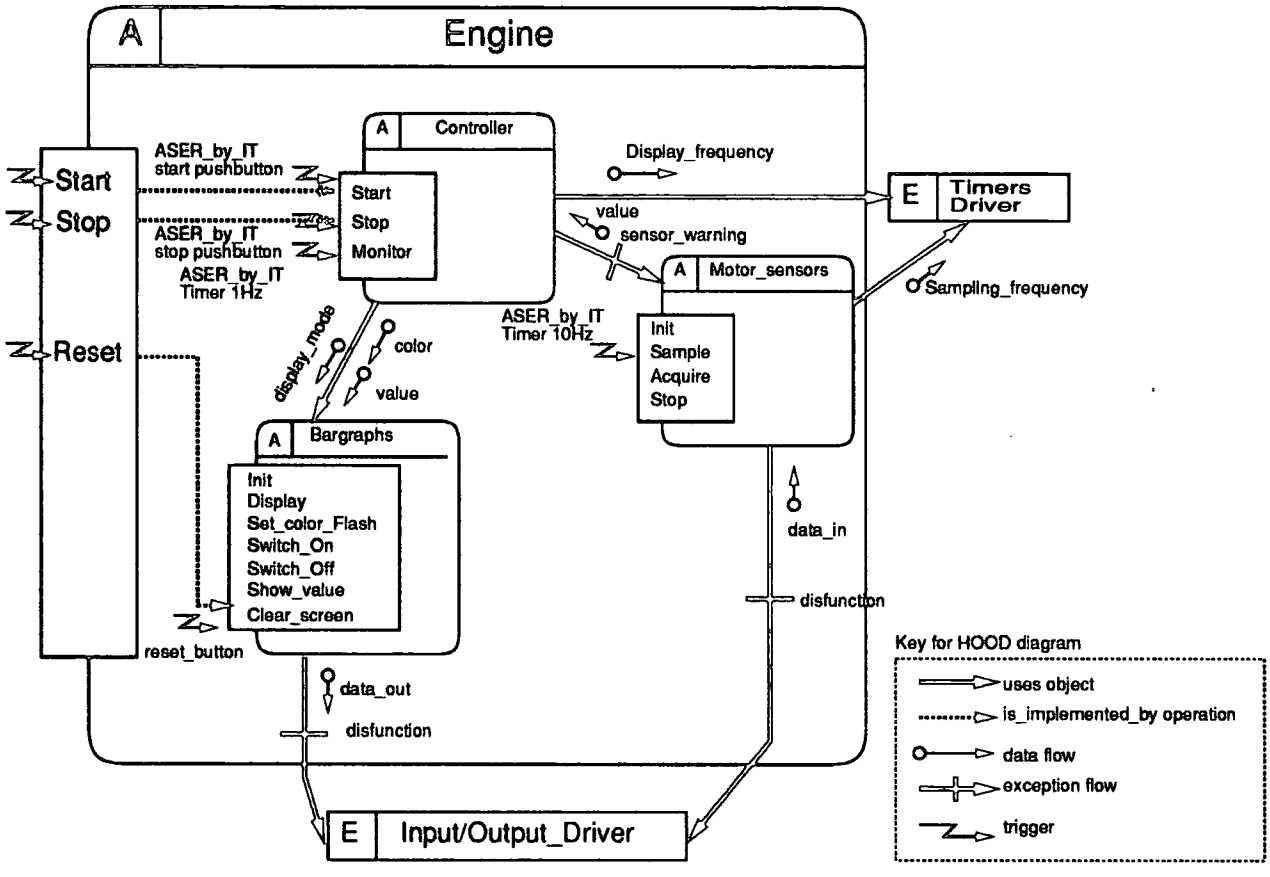

Figure 6.6: Aircraft Engine Monitoring System: HOOD Design (after change)

not at finer grained level such as required interfaces or operations [11]. Such information is contained in the ODS [12]. Therefore it is recommended to follow-up the propagation to look at the HOOD data-model (figure 5.3). Set_color and Flash are defined by the same object that is a *terminal object* (rule 016, o10). The two operations have the same type (*non-constrained*), same status (*external*) and are both *provided*. Since no constraint violation is detected, we can conclude that the pre-conditions are full-filled.

- The *transformation* is correctly performed. For step 1, attributes value (checked in the pre-conditions) are automatically calculated. *Operation_status* is *non-constrained* and *operation_type* is *external*. In step 2, for incoming/outgoing links intermediate-conditions to avoid duplication of links and cycles are verified. Those constraints are reported in the post-conditions. Indeed for *Set_color* and *Flash* operations it does not exist operations common to sets defining incoming and outgoing operations. Finally, in step 3 former entities and links on the repository are removed.

- *Post-conditions.* For this transformation, rules [13] are checked again for entities such as objects (rules C1, C7), interfaces (rules C4, C8) and operations (rules C5, C6, C11). No constraints violation of those post-conditions have been detected, which means that the *merging* is valid according to the heuristic defined. The HOOD diagram after the transformation (figure 6.5) partially illustrates new links and interfaces (only provided interfaces) between objects.

- Step 4: **Assessing the Impact**. This last step concludes that the transformation is valid because pre- and post-conditions are full-filled. Moreover intermediate conditions (rule C11) are also satisfied. In the task of merging operations maintainers have been supported by PCTE-tools and consistency of the repository has been proved -formal description, figure 5.9.

Example 2: **Adding a New Operation**

In our study, this type of modification consists of adding a new operation **ClearScreen** to the list of operations provided by Bargraphs object.

- Step 1: **Specifying the Change**. The screen may be refreshed by pressing a push-button *reset* located on the display, transformation #5 in table 6.1.

  It corresponds to the activation of the operation ClearScreen. This operation is *provided* (attribute operation_type), and *constrained* (attribute opera-

[11] On HOOD diagrams only *provided* operations are shown.

[12] Not visible propagations of rules on the diagram are underlined.

[13] For the 'merging operations' transformation, some rules are not applicable such as rules o10, I17, I16 because Bargraphs object is *terminal*. Similarly, rule C9 does not apply because operations are non-constrained.

tion_status) [14]. *Bargraphs* is an object of type *passive* (attribute object_status) providing the operations *Init, Display, Set_Color, Flash and Switch.*

- Step 2: **Modelling the Change.** The change is Modelled in terms of new entities (OBCS, OPCS) and new relationships.
- Step 3: **Tracing the Impact.** Adding the *Clear_Screen* operation changes the type of Bargraph to active, but also establishes new connections between operations. It is designed that Clear_Screen is implemented using the *Switch_Off* operation provided by Bargraphs object. Entities Bargraphs Object and Clear_Screen Operation are modelled on figures A 3.1 and A 3.10. Two new entities are created an OPCS for the operation and an OBCS for the object, figure A 3.11. Consequently, new links are established. Tracing the impact shows that the object containing the new operation changed its type. Adding a new operation, which only uses an operation defined by the same object has a limited propagation.
- Step 4: **Assessing the Impact.** This transformation is valid because conditions are satisfied. Similarly to example 1, rules are verified and propagation of changes has been described.

6.4 Conclusion

The purpose of the thesis is to support impact analysis in design artifacts specified with the HOOD method. It has been deliberately decided first to conduct a simple and complete study, the aircraft engine system, in order to master details of the different analyses to be performed. The current study focuses on functional aspects (merging operations, adding an operation) and on particular views of HOOD (information hiding and hierarchical decomposition principles). Thus, as explained previously transformations involving either more semantics or dynamic aspects of the system (e.g., control structuring principles) are not within the scope of our study. Assessing effects of changes in the real-world depends mainly on the user interpretation.

This task can be partially supported by a software engineering approach. Implementation and tests under a PCTE environment validate our solution and show that design dependencies are accurately investigated with our interconnection model. The data-model could be supported by the PCTE repository and transformations performed by the tool we developed. Tools supported then the user during the change process indicating violations (errors or warnings) to HOOD construction rules. Using PCTE as a repository has shown benefits for tracing dependencies

[14]The execution request is of type ASER_by_IT.

since it enables the user to access fine-grained data, such as a single attribute or a small piece of the contents of an artifact. This has been a major criteria in our choice to base the tool development and heuristics on PCTE.

Our approach could possibly be applied to a larger or more complex systems. The problem could be scaled-up by improving the performance and enhancing the functionalities of our tools. The complexity of the case study would require further conceptual work to propose an extended data-model and more complex transformations with a possibly mathematically underlined model.

Further work concerning the tool development and the area of impact analysis are outlined in next chapter.

Chapter 7

Summary and Further Research

7.1 Summary

This thesis proposes an approach to impact analysis in software maintenance. Software maintenance consists of several activities, in particular the understanding of impact analysis, which aims to detect all changes consequent to a modification. The design of a modification requires an examination of those unexpected behaviour of the system and assessing the traceability of a system is a crucial factor. The purpose of the thesis is to propose ways of extending techniques investigated at the code level to perform change analysis at earlier stages in the maintenance of a project. In particular, maintainers would benefit from using traceability support to analyse design artifacts specified with the HOOD method.

The process view of impact analysis has been described through an activity model. The product view is supported through an interconnection model providing maintainers with tools to represent transformations, automate and perform changes. Thus, horizontal propagation of changes at design level and design verification analyses can be undertaken. Heuristics have been implemented and tested on a case study, the aircraft engine monitoring system.

7.2 Discussion of the results of the case study and tools development

This section presents a discussion based on our study and explains what worked well, what failed and further work concerning the tool development.

Benefits of the study: "A data-model expressing HOOD concepts and a case study to validate it"

The rules we proposed to check HOOD design were correct and efficient. We could check both rules violation expressing the design correctness before and also after any modification. The data-model had been designed carefully to represent HOOD concepts and eventually to be simple to master without prior knowledge of HOOD. Our interconnection model contains ten types of entities and approximately twenty types of relationships.

The choice of the case study was also determinant since with the aircraft engine monitoring system we could illustrate most of the HOOD concepts. We had deliberately restricted our study to structural aspects and not investigating real-time issues, for example. Although the case study is rather small, in total the ERM representation is composed of over 150 links and 50 entities [1] collected in the data-base. With such a small system it worked well, but possibly to scale to bigger systems it would be necessary to modify the data-model as well as to improve the performance of our tools.

Support of PCTE. The suitability of PCTE-OMS to support successfully a *traceability platform* for development and maintenance activities arises from our experience. Our tools development including the prototypes *Hoodchecker* and *Hoodmodifier* has shown benefits of using PCTE. At the meta-level shared data-models offer the possibility of expressing constraints between tools. At the object-base level consistency has been proved formally and the case study has illustrated notification mechanisms between tools. Moreover PCTE proposes a tool called *OMS_Browser*, that enables the user to access directly the repository. We used this feature when testing the data-model. It has been very helpful since it was possible to access to a fine granularity such as entity or links attributes. However, any change of the data-model required the tester to load again all entities and links in the repository. After entering the elements in the object base, the rest of the activity has been easily supported by the tools we developed.

Another limitation exists with the current version of PCTE and could possibly represent an handicap for large size applications. Indeed, PCTE does not support modifications of the key for links without removing the object. In other words, the key for links is not recalculated after deleting instances of links. This requirement is not supported by the ECMA-149 standard and therefore not implemented in PCTE Emeraude (v12.4). Such a lack could possibly lead to inconsistencies on the repository.

Tools development. Possibly our tools could be improved on two aspects, which are the parsing of HOOD diagrams and the user interface. As explained above, at present the user enters the links directly in the repository through the oms_browser

[1] In our implementation in PCTE, an entity refers to a PCTE object.

tool. In particular for large size system, we should develop a parser for HOOD design to avoid this manual interaction with the repository.

Another aspect concerns the user interface that is very primitive presenting results in an on line textual mode. We would like to develop a graphical user interface pointing-out to the user on the HOOD diagrams possible warning or errors. Similar editors have been developed on PCTE to edit DCFD diagrams or state-transition diagrams.

Improving the Activity Model. Our approach aims to support impact analysis during the change process and in particular in understanding, which change can be made without impacting the structure and semantics of the systems. One important aspect is to avoid ripple effects without reducing the possibly of transformations on HOOD documents. Checking that the system described with the HOOD documents has not been changed is restricted at present to the possible propagations of the changes we can find out with our tools. Another aspect we did not investigate concerns concurrent changes which happen frequently in real systems during maintenance. By definition it corresponds to a set of changes performed at the same time. Techniques exist to try to cluster those changes since their inter-correlation is determinant to predict the impact analysis.

7.3 Further Research

Current methods for tracing dependencies among software artifacts proposed by Pfleeger [71] or Cimitile [4] put emphasis on propagation of coarse grained elements to avoid 'domain dependence'. If the impact assessment of a software system change is too coarse, it must be decomposed to understand complex relationships. On the other hand, if it is too granular, it is difficult to reconstruct impacts into recognizable understood software work-products. It seems then that opposing *granularity versus domain dependence* may not be the proper approach since those views could be independent. Our approach and interconnection model are suited for accessing the fine-grained data, such as a single attribute or a small piece of the contents of an artifact.
A summary of practical studies to assess impact analysis processes, given by Arnold [5], shows that frameworks have been proposed for tracing dependencies along the software-life cycle. Among them are earlier work of Yau [88, 82], of the SODOS project from Horowitz [41], or more recently of Wilde [81]. Several researchers constructed meta-models to investigate software artifacts. Brinkkemper [15] proposed a technique to evaluate constraints written in predicate logic. Although the usage of predicate logic increases the expressive power of the meta-model, it has the disadvantage integrating other paradigms (i.e. functional or procedural paradigms). Other approaches such as a model proposed by Sawada [76] express constraints via

a constraint description language, external to the repository. Possibly, the efficiency on the repository could be increased, but limitations to express either complex constructs or dynamic constraints show that such models cannot fully support *propagation of changes.*

Our study focused on structural aspect of HOOD. However, *behavioral aspects* of Hood (i.e., related to control-flow for example) could also be investigated and therefore constraints modelling should support *dynamic constraints* (e.g., dependencies between operations and objects dynamically created). Our study focuses on a few types of transformation, since higher abstraction modifications requires more semantics of the application. It would also justify more fundamental work mathematically to describe them.
We investigated horizontal traceability and proposed a framework to support propagation of changes. This could be enhanced to support other aspects, such as *vertical traceability* applying it, for example, to trace changes forwards and backwards between analysis and design documents. Tracing dependencies between analysis and design artifacts would then help to perform a complete impact analysis, representing the system at different levels. However, it must be noted that at a higher level of abstraction (e.g., analysis), syntactical constructs are in smaller number than those at a lower level (e.g., design) closer to implementation details. Therefore, mapping concepts is complex since one construct may be mapped to several low-level concepts. Thus propagation of changes cannot be selectively directed, without knowing semantics of the application or a close interaction between tools and users.

Appendix 1 : HOOD Rules (Abstract)

- Class 1: Syntactical Constraints - supported by the ERM notation.
  - rule o1: An operation may be in the external interface of an object or internal to an object.
  - rule o2: Each operation shall be provided by one and only one object.
  - rule o4: Each parent operation shall be implemented by an operation of a child object.
- Class 2: Semantical Constraints - supported by the hoodchecker tool.
  - rule o10: An operation of a terminal object shall not be implemented_by an operation of another object.
  - rule o12: Each operation of a non terminal object shall be implemented_by an operation of a child object.
  - rule o16: An object shall not have both internal objects and internal operations.
  - rule I7: An object shall not decompose or be decomposed from itself.
- Class 3: Design Consistency Rules - supported by the hoodchecker tool.
  - rule C1: If an object A has a required operation of an object B then object A must use object B.
  - rule C4: The provided Interface of the used object shall correspond to the required interface of the using objects.
  - rule C5: An operation which is required must be provided.
  - **rule C6**: An operation which is provided must be required.
  - rule C7: (reverse to rule C1) An using object must have an operation which requires an op. of the used object.
  - **rule C8**: The provided IF of an object cannot be empty.
  - **rule C9**: A constrained operations must be provided.
  - *rule C10*: An object shall not use itself directly or indirectly (cycle).
  - *rule C11*: An operation shall not use itself directly or indirectly (cycle).
  - *rule C11-bis*: Between two operations having a use relationship for the same data-name it can only be one dataflow (in, out) - i.e. either flows cannot exists in both directions.
  - rule I16: A constrained operation provided by a non terminal object must be implemented by a constrained operation.

Note: If there are any violations to rules it produces errors or *warnings* in **bold.**

Appendix 2 : Transformations on HOOD artifacts (Abstract)

- **check_merge_op**. Two operations can be merged in a new resulting operation if they belong to the same terminal object. Those operations cannot be implemented_by other operations. Their attributes (*operation_status* and *operation_type*) must be of the same value otherwise an error or warning message arises.

- **check_merge_obj**. Two terminal objects can be merged if they belong to the same parent object. Their attributes (*object_status* and *object_type*) must be of the same value otherwise an error or warning message arises. For those objects the former interfaces are folded into the new interface of the resulting object.

Notes: The above two rules check constraints on links and attributes to support merging transformations. If there are any violations to rules it produces errors or *warnings*. Those constraints are supported by the hoodchecker and hoodmodifier tools.

Appendix 3 : Case Study - HOOD Objects

This appendix gives an abstract of entities stored in the data-base. The system is described before change, in particular, for the objects Bargraphs, Controller, Motor-sensors and Timers-Driver (figures 3.1, 3.2, 3.3, 3.4) and for operations controller.Start (figures 3.5). After change the system is described for two transformations (section 6.3).

- Example 1: It refers to the 'merging' of two objects, respectively of operations (transformation #3,table 6.1). Controller and Bargraphs objects are depicted on figures 3.6, 3.7 and related object-operation cross-tables on figures 3.8, 3.9.
- Example 2: This modification consists of adding a new operation **ClearScreen** to the list of operations provided by Bargraphs object. It corresponds to the transformation #5 in table 6.1. New entities Clear-Screen, Clear-Screen.OPCS and Bargraphs.OBCS are described on figures 3.10, 3.11, 3.12.

* Note 1: We indicate constrained operations with a start.

* Note 2: New elements (attributes, operations, interfaces) are indicated by the symbol $.

* Note 3: Internal operations are indicated by the symbol #. They are also reported in the object-operation cross-tables. Internal operations are operations listed in the *defines* list of operations, but not in the *contains* Interface list (e.g., value_out_of_range for controller object).

System Description before change

```
HOOD Object       Bargraphs.obj
Attributes
 object_type  : [terminal =1]
 object_status: [passive =1]
Description --|The bargraphs allow to display values, in red or green
with or without flashing, on appropriate display devices.|--
Links:
included_by      engine.obj
uses_obj         Input_Output_Driver.obj
used_by_obj      Controller.obj
defines          Init.op; Display.op; Show_value.op; Set_Color.op;
                 Flash.op; Switch.op
contains         Init.IF; Display.IF;  Show_value.IF; Set_Color.IF;
                 Flash.IF; Switch.IF
described_by_text  Bargraphs.ods
```

Figure A 3.1: Bargraphs Object (Entity Bargraphs.obj)

```
HOOD Object       Controller.obj
Attributes
 object_type   :  [terminal =1]
 object_status :  [active =1]
Description --|This object is the controller of the Engine. It starts
and stops the Engine (starts and stops pushbuttons) and monitors the
display. The monitoring is triggered every second (by interruption).|--
Links:
included_by        engine.obj
uses_obj           Bargraphs.obj; Analog_Display; Motor_sensors.obj;
                   Timers_Driver.obj; I_O_Driver.obj
defines            Start.op*; Stop.op*; Monitor.op*;value_out_of_range#
contains           Start.IF*; Stop.IF*; Monitor.IF*
described_by_text  Controller.ods
```

Figure A 3.2: Controller Object (Entity Controller.obj)

```
HOOD Object       Motor-sensors.obj
Attributes
 object_type  : [terminal =1]
 object_status: [passive =1]

Description --|This object samples oil pressure, water  temperature  and
fuel  level at a given frequency. It stores the read values of the three
sensors at any time. It may provide the mean of stored values  for  each
sensor.|--

Links:
is_used_by       Controller.obj
uses             Timers_Driver.obj;Input_Output_Driver.obj
defines          Init.op; Sample.op*; Acquire.op; Stop.op
contains         Init.op; Sample.op*; Acquire.op; Stop.op
described_by_text  Motor-sensors.ods
```

Figure A 3.3: Motor-sensors Object (Entity Motor-sensors.obj)

```
HOOD Object       Timers_Driver.obj
Attributes
 object_type  : [terminal =1] /*environment object*/
 object_status: [active =1]

Description --|The Timers_Driver manages a  set  of  timers  which  send
cyclic  interruptions  at  a  specified address.  A timer is initialised
with a given frequency and may be started (re- started)  or  stopped  at
any  moment.   A  timer  then  may be deleted from the list of available
timers.|--

Links:
is_used_by      Controller.obj; Motor-sensors.obj
defines         Init.op; Start.op*; Stop.op*; Delete.op
contains        Init.IF; Start.IF*; Stop.IF*; Delete.IF
described_by_text  Timers_Driver.ods
```

Figure A 3.4: Timers-Driver Object (Entity Timers-Driver.obj)

```
procedure OPCS_Start is

-- Description --This operation initialises the system (the bargraphs,
the analog_display, the sensors) and then starts a timer to trigger the
monitoring at the frequency of 1 Hz. The monitoring timer is started
only when all hardware devices are initialised.|--

-- Used_operations
Timers_Driver.Init
Timers_Driver.Start
Motor_sensors.Init
Bargraphs.Init

-- Code
begin
 Timers-Driver.Init (Monitoring-Timer,Monitoring-Frequency, IT-1Hz-Address );
 Bargraphs.Init;
 Analog_display.Init_analog;
 Motor_sensors.Init;
 Timers-Driver.Start (Monitoring-Timer);
end OPCS_Start;

-- END_OPERATION OPCS_Start
```

Figure A 3.5: OPCS for Operation controller.Start (Abstract of the ODS)

Controller op.	Start	Stop	Monitor	Comments
Internals			X	
Outofrange				
Bargraphs				
Init	X			
Display			X	
Set_Color			X	
Flash			X	
Switch		X		
Sensors				
Init	X			
Sample	-	-	-	not used
Acquire			X	
Stop		X		
Timers_Driver				
Init	X			
Start	X			
Stop	-	-	-	not required
Delete		X		

Figure A 3.6: Object-operation Table for Controller

Bargraphs op.	Init	Display	Set_Color	Flash	Switch	Comments
I_O_Driver						
Put	X	X	X	X		
Get	-	-	-	-	-	not used

Figure A 3.7: Object-operation Table for Bargraphs

System Description after change

```
HOOD Object       Controller.obj
Attributes
 object_type  : [terminal =1]
 object_status: [active =1]
Description --|This object is the controller of the Engine. According to
the  different transformations performed on the system, the structure of
the object has been changed. Since Analog_display has been  merged  with
Bargraphs, this object is not any more in the list of used objects. This
object requires also new or changed operations, but this is not shown at
the  level  of this entity.  Otherwise the functionalities of the object
have been globally preserved.
Links:
included_by        Engine.obj
uses_obj           Bargraphs.obj;  Motor_sensors.obj;
                   Timers_Driver.obj; I_O_Driver.obj
defines            Start.op*; Stop.op*; Monitor.op*;value_out_of_range#
contains           Start.IF*; Stop.IF*; Monitor.IF*
described_by_text  Controller.ods
```

Figure A 3.8: Controller Object (Entity Controller.obj)

```
HOOD Object       Bargraphs.obj
Attributes
 object_type  : [terminal =1]
 object_status: [active =1]
Description --|This object samples oil pressure, water  temperature  and
fuel  level  at  10Hz and stores the read values of the three sensors at
any moment. It may provide the mean of stored values of a sensor.|--
Links:
included_by      engine.obj
is_used_by       Controller.obj
uses            Input_Output_Driver.obj
defines          Init.op; Display.op; Clear_Screen.op$; Show_value.op$;
                 Set_color_Flash.op$; Switch_On.op$; Switch_Off.op$
contains         Init.IF; Display.IF; Clear_Screen.IF; Show_value.IF;
                 Set_color_Flash.IF; Switch_On.IF, Switch_Off.IF
described_by_text  Bargraphs.ods
```

Figure A 3.9: Bargraphs Object (Entity Bargraphs.obj)

```
HOOD Operation   Clear_Screen.op
Attributes:
 operation_type  : [ external =0]
 operation_status: [constrained =1]  /*reset pushbutton*/

Description --|This operation refreshes the  displays|--

Links:
uses_op          Init.op; Display.op; put.op /*I_O.obj*/
implements       Reset /*defined in Engine.obj*/
provides_IF      Clear_Screen.IF
requires_IF      put.IF  /*I_O.obj*/
```

Figure A 3.10: New Operation Clear-Screen (Entity Clear-Screen.op)

```
HOOD_OBCS       Clear_Screen.opcs
Attributes

Description --| An activation of the button to clear the  screen
initialises the Bargraphs hardware.|--

Links:
component_of    Bargraph.obj
describes_op    Clear_Screen.op
```

Figure A 3.11: New Entity Clear-Screen OPCS (Entity Clear-Screen.opcs)

```
HOOD_OPCS       Bargraphs.OBCS
Attributes

Description --|The Bargraphs accepts start, stop  and  monitor  commands
from  the  controller  object  at  any  time.  As  justified  by  the
controller.OBCS a start command is not significant after a start command,
reciprocally  a  stop  or  a monitor are not significant after a stop
command.  The entity  Bargraphs.OBCS  controls  the  processing  of  the
following  operations  declared  in  Bargraphs: Init.op; Switch._On.op$;
Switch_Off.op$; Clear_Screen.op$. The Bargraph  accepts  a  Clear_Screen
operation  at  any  time.   This operation has no effect if the hardware
Bargraph is not previously initialised.|--

Links:
dynamic_part    Bargraph.ods;
controls        Bargraph.obj;
controls_op     Init.op; Switch._On.op$; Switch_Off.op$; Clear_Screen.op$;
controls_by_op  Start.op*; Stop.op*; /*from Controller Object */
```

Figure A 3.12: New Entity Bargraphs OBCS (Entity Bargraphs.OBCS)

Controller op.	Start	Stop	Monitor	Comments
Internals			X	
Outofrange				
Bargraphs				
Init	X			
Display			X	
Clear_Screen		X		
Set_Color_Flash			X	
Show_value			X	
Switch_On		X		
Switch_Off		X		
Sensors				
Init	X			
Sample	-	-	-	not used
Acquire			X	
Stop		X		
Timers_Driver				
Init	X			
Start	X			
Stop	-	-	-	not required
Delete		X		

Figure A 3.13: Object-operation Table for Controller

```
-- OBJECT_CONTROL_STRUCTURE /*OBCS of the controller object*/

--DESCRIPTION  --|TBD|--

-- PSEUDO_CODE --|TBD|--

-- CODE
task OBCS_Ctrl_EMS is
   entry Start;
   entry Stop;
   entry Monitor;
   for Monitor use at IT_1Hz_Address;
end OBCS_Ctrl_EMS;

task body OBCS_Ctrl_EMS is
begin
  loop
   loop
    select
    accept Start; OPCS_Start; exit;
    or
    accept Stop; -- empties Stop queue
    or
    accept Monitor; -- empties Monitor queue
    end select;
   end loop;

   loop
    select
    accept Start; -- empties Start queue
    or
    accept Stop; OPCS_Stop; exit;
    -- only when monitoring is completely finished
    or
    accept Monitor; OPCS_Monitor;
    end select;
   end loop;

  end loop;
end OBCS_Ctrl_EMS;
```

Figure A 3.14: OBCS of the Controller Object
(Abstract of the ODS: code part only)

Glossary

Note: Several definitions for the same concept may be found, depending on the context of its usage. This glossary refers only to definitions presented in the thesis. Only key concepts of the thesis are listed.

Software maintenance: Software maintenance is the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment.

Corrective maintenance: Maintenance activities performed to correct faults in a software system.

Adaptive maintenance: Maintenance activities performed to make a software system usable in a changed operating environment.

Perfective maintenance: Maintenance activities performed to enhance the functionality of a software system.

Preventive maintenance: Maintenance activities performed to improve performance, or maintainability of a software system.

Process modelling: Process modelling is the detailed analysis and modelling of maintenance activities to understand the process (descriptive point of view), to control it (prescriptive point of view) and to guide it (indicative point of view).

Activity model: An activity model is a view of the process model focused on activities.

Artifact: An artifact is a document produced through an activity model. It might be formulated in different formalisms such as a text, a diagram, a graphic or a set of mathematical descriptions. Syn: artefact.

Data-model: A data-model is a model, which emphasizes on the importance of aspects related to data in a software system.

Impact analysis: Impact analysis is the activity of determining parts of the system, which are to modify in order to accomplish a change. Accomplish a change means to determine the confidence that the change conforms to its specification or to what we intend it to do.

Ripple effect propagation: Ripple effect propagation is the phenomenon by which changes made to a software-component along the software life-cycle [specification, design, code, or test phases] have tendencies to be felt in other components.

Traceability: Traceability is the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another.

Relationship: A relationship between objects a and b is a three tuple.
Given objects a and b, a relation R is defined as $< a, R, b >$

[another usage is (relation-name, attributes) symbolized $R(a, b)$)].

data-flow analysis: Aho [2] defines it as *"Given a control flow structure, data flow analysis is the process of collecting information about the flow of data throughout the corresponding code segment."*

Control-flow analysis: A control-flow analysis describes the sequence of execution of statements in a program. This depends in particular of sequential and parallel operations and of synchronous/asynchronous behaviour of the program.

Dependency: From the maintainers point of view, there is a dependency between two components, if a change to one component has an impact that will require changes to the other.
A dependency is a *directed* relationship (e.g. calls, uses, read, write relations).
A depends on B means that a change to A, causes a change to B.

Dependency graph: A dependency graph is a directed graph.
A depends on B *iff there is a path from* A *to* B.

Transitive closure: Let G be a graph. Define G* to be the graph that contains all the nodes of G. The edges in G* are as follows: if there is a path of length 0 or more between node A and B in G, then the edge (A, B) is in G*. G* is called the transitive closure of G [2].

Computing a transitive closure: Note that the following example is not necessarily the best way to implement a transitive closure, but probably the most natural way. Let consider the computing of a transitive closure of a directed graph. If the graph is represented by a predicate *arc* such that *arc (X, Y)* is true *iff* there is an arc from node X to node Y, then we can express paths in the graph by the rules:

1) path (X, Y) :- arc (X,Y).
2) path (X, Y) :- path (X,Z) & path (Z,Y).

The first rule says that a path can be a single arc, and the second says that the concatenation of any paths, say one from X to Y and another from Y to Z, yields a path from X to Y. These rules are expressed by the following equation.

$$path(X, Y) = arc(X, Y) \cup \pi_{X,Y}(path(X, Z) \bowtie path(Z, Y))$$

where $\pi$and $\bowtie$ respectively represent projection and join of relational algebra.

Dynamic analysis: Dynamic analysis is the process of evaluating a system or component based on its behaviour.

Static analysis: Static analysis is the process of evaluating a system or a component based on its structure, or content.

Error: This word has different meanings. The first view expresses differences between a computed, or measured value and the specified, or theoretically correct value. It occurs, for example, if computed and expected results are different. A second view expresses faults in case of an incorrect step, process, or data definition, for example, an incorrect instruction in a program.

- Semantic error: An error resulting from a mis-understanding of the relationship of symbols, or groups of symbols to their meaning in a given language.

- Syntactic error: (Syn: syntax error) A violation of structural or grammatical rules defined for a language. For example, in FORTRAN using the statement B + C = A , instead of the correct statement A = B + C produces a syntax error.

References

[1] K. Agusa, Y. Kishimoto and Y. Ohno, 1983, **A Supporting System for Software Maintenance**, *In G. Teichroew and G. David, editors, System Description Methodologies, North Holland, Amsterdam, Proceeding of IFIP TC2*

[2] A. V. Aho, R. Sethi and J. D. Ullmann, 1986, **Compilers principles techniques and tools**, *Addison-Wesley Pub.*

[3] F. E. Allen and J Cocke , 1977, **A program data flow analysis procedure**, *Comm. of the ACM, Vol 19* pp 137-147

[4] P. Antonimi, P. Benedusi, G. Cantone, and A. Cimitile, 1987, **Maintenance and Reverse Engineering: Low Level Design Documents Production and Improvement**, *in Proc. Conference on Software Maintenance*

[5] R.S. Arnold and S.A. Bohmer, 1993, **Impact Analysis - Towards A Framework for Comparison**, *in Proc. Conference on Software Maintenance.*

[6] G. Arrango et al., 1991, **A Tool Shell for Tracking Design Decisions**, *in IEEE Software* pp 75-83

[7] G. Arrango et al., 1993, **The Graft-Host Method for Design Change**, *in Proc. 15th Int. Conf. Software Engineering* pp 243-255

[8] G. Arrango et al., 1993, **A process for consolidating and reusing Design Knowledge**, *in Proc. 15th Int. Conf. Software Engineering* pp 233-243

[9] F. Bancilhon, C. Delobel, P. Kanellakis, 1992, **Building an Object-Oriented Database System/The story of O2**, *The Morgan Kaufmann Series in Data Management Systems*

[10] J. M. Barth, 1978, **A practical inter-procedural data flow analysis algorithm**, *Comm of the ACM, Vol 21* pp724-736

[11] V.R. Basili and D.M. Weiss, 1981, **Evaluation of a Software Requirements Document by Analysis of Change Data**, *Proc. 5th Int. Conf. on software Engineering* pp 314-323

[12] K.H. Bennett, B.J. Cornelius, M. Munro and D.J. Robson, 1988, **Software Maintenance: A Key Area For Research**, *University computing, 10(4)* pp 184-188

[13] Grady Booch, 1983, **Software Engineering with Ada**, *Benjamin/Cummings Publishing*

[14] Boehm, 1976, **Software Engineering**, *IEEE transactions on Computing, 25* pp1226-1242

[15] S. Brinkkemper, 1990, **Formalisation of Information Systems Modelling**, *Thesis publisher.*

[16] F.W. Calliss, 1989, **Inter-Module Code Analysis Techniques for Software Maintenance**, *Ph.D thesis, University of Durham, Computer Science, 1989*

[17] G. Canfora and A. Cimitile, July 1992, **Reverse Engineering and intermodular data flow analysis: a theoretical approach**, *Journal of Software Maintenance, Vol 4* pp 37-59

[18] P. P. Chen, 1976, **The entity relationship model -towards a structured view of data**, *Trans. Database Systems, 1(1)* pp 9-30

[19] A. Cimitile, 1989, **A reverse engineering methodology to reconstruct hierarchical data flow diagrams for software maintenance**, *in Proc. Conference on Software Maintenance*

[20] A. Cimitile, 1989, **Maintenance and intermodular dependencies in Pascal environment**, *in Proc. Conference on Software Maintenance*

[21] Mac Clure C., Martin J., 1983, **Software Maintenance: The problems and its solutions**, *Prentice Hall*

[22] Mc Clure C., Carma L., 1981, **Managing Software Development and Maintenance**, *Publisher Van Nostrand Reinhold Co., New York, NY*

[23] L. D. Cousin and J. S. Collofello, 1992, **A Task-Based Approach to Improving the Software Maintenance Process**, *in Proc. 8th IEEE Conf. Software Maintenance* , pp 118–126

[24] James S. Collofello and Stephen Bortman, March 1986, **An Analysis of the Technical Information Necessary to Perform Effective Software Maintenance**, *in Proc. 5th Annual Phoenix Conference on Computers and Communications* pp 420–424

[25] DTI - British Department of Trade and Industry, (1981), **Report on the study of an Ada based system development Methodology**, *Technical Report.*

[26] ECMA - European Computer Manufacturers Association, Dec. 1990, **A reference model for Computer Assisted Software Engineering Environments**, *ECMA-155 Technical Report*

[27] ECMA - European Computer Manufacturers Association, 1993, **Portable Common Tool Environment (PCTE) C++ Programming Language Binding**, *Draft version 2*

[28] ECMA - European Computer Manufacturers Association, June 1993, **Portable Common Tool Environment (PCTE) abstract specification**, *ECMA-149 Standard version 2*

[29] ECMA - European Computer Manufacturers Association, June 1993, **Portable Common Tool Environment (PCTE) C Programming Language Binding**, *ECMA-158 Standard version 2*

[30] ECMA - European Computer Manufacturers Association, June 1993, **Portable Common Tool Environment (PCTE) Ada Programming Language Binding**, *ECMA-162 Standard version 2*

[31] ESA Board for Software Standardization and Control, Feb. 1991, **ESA Software Engineering Standard**, *ESA PSS-05*

[32] Esteban J.A. and Alvarez Carmen, 1992, **Reverse engineering from ADA to HOOD**, *ESF Publication*

[33] P. Fillon, C. Floyd and H. Biskup, (1991), **Objekt-orientierte Software Entwicklung und Werkzeuge**, *Technische Universität Berlin, Technical Report.*

[34] P. Fillon, T. Ajisaka, Y. Matsumoto, IEEE Conf., to appear, Nov. 1994, **A facility to trace dependencies for software maintenance on PCTE**, *Proc. Int. Conference on PCTE, San Fransisco*

[35] J. R. Foster and M. Munro, pp181-185, **A documentation method based on cross-referencing**, *1987*

[36] Del-Raj Harjani and Jean-Pierre Queille, Nov. 1992, **A Process Model for the Maintenance of Large Space System Software**, *in Proc. 8th IEEE Conf. Software Maintenance* 127–136

[37] J. Hartmann and D.J. Robson, 1988, **Approaches to Regression Testing**, *in Proc. IEEE Conference on Software Maintenance*

[38] J. Hartmann and D.J. Robson, 1990, **Techniques for Selective Revalidation**, *IEEE Software 7(1)* pp 31-36

[39] HOOD User Group, July 1992, **HOOD User Manual V3.1.1**, *Edition Masson*

[40] HOOD User Group, July 1992, **HOOD Reference Manual V3.1.1**, *Edition Masson*

[41] SODOS: A software document support environment, (1986), **E. Horowitz and R. Williamson**, *IEEE Transactions on Software Engineering, Vol SE-12 No8.*

[42] J. C. Huang, May 1979, **Detection of data flow anomaly trough program instrumentation**, *IEEE Trans. on soft Eng, Vol SE5* pp 226-236

[43] IEEE Standards Board and ANSI Standards Institute, 1990, **IEEE Standard Glossary of Software Engineering Terminology**, *ANSI/IEEE Std610.12-1990*

[44] IEEE Computer Society, 1993, **Standard for Software Maintenance**, *IEEE Std 1219-1993*

[45] J. Jachner and V. K. Agarwal , September 1984, **Data Flow anomaly detection**, *IEEE Trans. on soft Eng, Vol SE10* pp432-437

[46] M. Johnson, 1993, **On the value of commutative diagrams in Information modelling**, *Technical Report, University of Sydney.*

[47] D. Kafura and S. Henry, 1981, **Software Quality Metrics based on interconnectivity**, *Journal of Systems and Software* pp 121-131

[48] J. B. Kam and J. D. Ullmann, 1976, **Global data flow analysis and iterative algorithms**, *Journal of the ACM Vol 23* pp158-171

[49] B. L. Kell, 1966, **Impact and Change**, *The century Psychology Series*

[50] F. Lanubile & al., 1992, **Traceability and Design decisions**, *in Proc. Conference on Software Maintenance*

[51] B. Lientz, E. Swanson and E. Tompkins, 1978, **Characteristics of Application Software Metrics**, *CACM 21 (6)*

[52] B. Lientz, E. Swanson, October 1978, **Discovering issues in Software Maintenance**, *Data Management* pp15-18

[53] B. Lientz, E. Swanson , 1979, **Software Maintenance a user management tug of war**, *Data Management, April 79* pp 26-30

[54] B. Lientz, E. Swanson , 1980, **Software Maintenance Management**, *Addison Wesley*

[55] B. Lientz and E. Swanson, 1983, **Problems in Application Software Maintenance**, *in tutorial on Software Maintenance, editors G. Prikha and N. Zvegintzov, IEEE Computer Society Press*

[56] C.C. Liu, 1976, **A look at Software Maintenance**, *Datamation, 22* pp51-55

[57] J. A. Lowell, 1988, **Software Evolution: The Software Maintenance Challenge**, *edition Wiley* pp 39-71

[58] J. Mac Dermid and K. Ripken, 1983, **Life Cycle Support for Ada Environment**, *ESF Deliverable*

[59] TA Consultancy Ltd., issue 3, **Malpas: Management guide**, *2/1992*

[60] TA Consultancy Ltd., issue 1, **Malpas: ADA translator example guide**, *6/1992*

[61] Massimo d'Alessandro et al., 1993, **Modelling reusable HOOD designs on the PCTE Object Management System**, *Proc. PCTE conference.*

[62] N. Mitsuda, A. Sawada, T. Ajisaka, Y. Matsumoto, 1993, **A semantic-Directed graph editor on PCTE**, *Proc. JCSE conference.*

[63] Lawrence B. Mohr, 1988, **Impact analysis for program evaluation**, *Ed. The Dorsey Press*

[64] M. Moriconi and C. Mac Clure, 1979, **A Designer/Verifier's Assistant**, *IEEE Trans. on Software Engineering*

[65] C. Nosek, 1990, **Software Maintenance Management: Change in the Last Decade**, *Journal of Software Maintenance Research and Practice* Vol. 2(3)

[66] L. D. Fosdick and L. J. Osterweil, September 1976, **Data Flow Analysis in software reliability**, *Computing Survey, Vol 8* pp 305-310

[67] L. Osterweil and L. Fosdick, September 1976, **DAVE -A validation error detection and documentation system for FORTRAN programs**, *Software Practice and Experience*

[68] L. Osterweil and L. Fosdick, October 1976, **The detection of anomalous inter-procedural data flow**, *Proc of 2th int. conf. on soft. Eng., IEEE comp. soc. press* pp 624-628

[69] L. Osterweil and C. Wilson, September 1985, **OMEGA - a data flow analysis tool for the C programming language**, *IEEE Trans. on soft Eng, Vol SE11,* pp 832-838

[70] S.L. Pfleeger and A.B. Shawn, 1990, **A Framework for Software Maintenance Metrics**, *IEEE Conference on Software Maintenance* pp 320-331

[71] S.L. Pfleeger, 1991, **Software Engineering**, *Macmilan, second edition*

[72] R. Pressman, 1991, **Software Engineering: A Practitioner approach**, *Mac Graw-Hill, second edition* pp538-541

[73] J. P. Privitera, 1982, **ADA design language for the structured design methodology**, *Proc. of the AdaTEC Conference,* pp 76-90.

[74] P. Robinson, 1992, **Object-oriented Design**, *Unicom, published by Chapman and Hall*

[75] G. Savoia, 1993, **Building a case toolset on PCTE**, *Proc. PCTE conference.*

[76] A. Sawada, N. Mitsuda, T. Ajisaka, Y. Matsumoto, 1993, **Utilities for avoiding Constraints violation**, *Proc. PCTE conference.*

[77] W. K. Sharpley, 1977, **Software Maintenance planning for embedded computer systems**, *IEEE Compsac 77* pp520-526

[78] B. Schneiderman, R. Mayer, 1979, **Syntactic/Semantic Interactions in Programmer Behaviour: A Model and Experimental results**, *International Journal of Computer and Information Sciences* Vol. 8, nb 3, pp219-238

[79] R. N. Taylor , 1983, **A general purpose algorithm for analyzing concurrent programs**, *Comm. of the ACM Vol 26 no 5* pp362-376

[80] D.M. Weiss, 1981, **Evaluating software development by analysis of change**, *PhD Dissertation, University of Maryland*

[81] N. Wilde, 1989, **The Maintenance Assistant Work in Progress**, *Journal of Systems and Software, 9(1)* pp3-18

[82] S.S. Yau and J. S. Collofello, 1978, **Ripple Effect Analysis of Software Maintenance**, *in Proc. COMPSAC 78*

[83] S. S. Yau and J. S. Collofello, 1979, **Some stability Measures for Software Maintenance**, *in Proc. of the Computer Software and Applications conference, IEEE* pp 674-679

[84] S. S. Yau and J. S. Collofello, 1980, **Some Stability Measures for software Maintenance**, *IEEE Trans. Software Eng, 6(6)* pp 545-552

[85] S. S. Yau, 1984, **Methodology for Software Maintenance**, *RADC Report*

[86] S. S. Yau and J. S. Collofello, 1985, **Design Stability Measures for Software Maintenance**, *IEEE Transactions on Software Engineering (11)* pp 849-856

[87] S. S. Yau and S. Liu, 1984, **A Knowledge Based Software Maintenance Environment**, *in Procs. of the 10th COMPSAC Conf.* pp 72-78

[88] S.S. Yau and S. Liu, 1987, **Some approach to logical ripple effect analysis,** *Technical report, SERC*

[89] S. S. Yau and P. S. Chang, 1988, **A Metric of Modifiability for Software Maintenance,** *in proc. Conference on Software Maintenance* pp 374-381