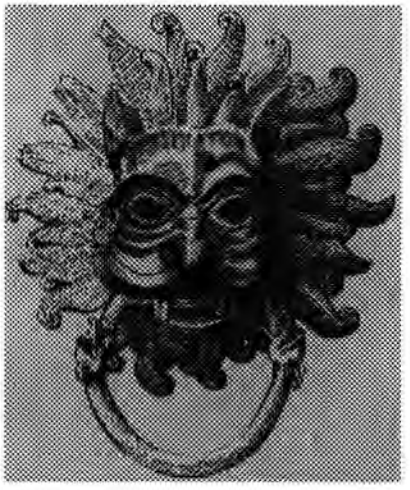
An interaction paradigm for impact analysis

Thierry Bodhuin

University of Durham

Department of Computer Science
Centre for Software Maintenance



An Interaction Paradigm for Impact Analysis

M.Sc. by thesis

Thierry BODHUIN

Abstract

The Aerospace industry is concerned with huge software projects. Software development is an evolving process resulting in larger and larger software systems. As systems grow in size, they become more complex and hence harder to maintain. Thus it appears that the maintenance of software systems is the most expensive part of the software life-cycle, often consuming 50-90% of a project total budget.Yet while there has been much research carried out on the problems of program and system development very little work has been done on the problem of maintaining developed programs. Thus it will be essential to improve the software maintenance process and the environment for maintenance.

Historically, the term *Software Maintenance* has been applied to the process of modifying a software program after it has been delivered and during its life time. The high cost of software during its life cycle can be attributed largely to software maintenance activities, and a major part of these activities is to deal with the modifications of the software. These modifications may involve changes at any level of abstraction of a software system (i.e design, specification, code,...). *Software Maintenance* has to deal with modifications which can have severe *Ripple Effects* at other points in the software system. Impact Analysis addresses the problem and attempts to localize these *Ripple Effects*.

In this thesis the *Software Maintenance* process and more specifically the *Impact Analysis* process is examined. The different parts of the implementation for the Impact Analysis System are explained. The main results of the thesis are the dependencies generation and the graph tool used to visualize these dependencies as well as the impacts on general dependency graph for impact analysis purpose.

To Mariella and my parents

Acknowledgements

This work was supported by the Department of Advanced Software Engineering of *Matra Marconi Space France* (MMS-F).

I am grateful to my supervisors Mr. Jean Pierre QUEILLE (MMS-F) and Mr. Malcolm MUNRO for their guidance throughout this study. I would like to thank the Software Maintenance team in the Department in which I have worked in MMS-F for their support, advice and collaboration during this work.

I would also like to thank Professor Keith H. BENNETT for the facilities provided, and all the members of the *Centre of Software Maintenance* for invaluable discussions during this research.

Contents

List of Figures

1.Introduction

1.1. Purpose of the Research

The Aerospace industry is concerned with huge software projects, Software development is an evolving field resulting in larger and larger software system. As systems grow in size, they become more complex and hence harder to maintain. Thus it appears that the maintenance of software systems is the most expensive part of the software life-cycle *[LIENTZ_80]*, often consuming 50-90% of a project's total budget.Yet, while there has been much research carried out on the problems of program and system development, very little work has been done on the problem of maintaining developed programs.

It is impossible to produce systems of any size which do not need to be maintained. Over the lifetime of a system, its original requirements may be modified to reflect changing needs; the working environment may change and errors may appear. Because maintenance is unavoidable, systems should be designed and implemented so that maintenance problems are minimized.*Software Maintenance* has to deal with modifications which can have severe *Ripple Effects* at other points in the software system from those of these modifications. Impact Analysis addresses the problem and attempts to localize these *Ripple Effects*. The study of the *Ripple Effects* is a major factor in the *Software Maintenance* process because of their effects of the utilisation of the system.

The purpose of this research is to create an environment for *Software Engineering [BOEHM_76]* and more particularly for *Software Maintenance* in order to visualize the usually unexpected *Ripple Effects* from a set of modified object (Documentation component, function, test cases, design objects,...).

1.2. Objectives of the Research

The main objective of the research addressed is:

- *how can the impacts of a change be detected and visualized at the earliest possible stage of the maintenance process ?*

Different aspects of this problem have been considered to perform this task:

- *how to model the system on which the impact analysis has to be performed ?*

- *how to propagate the information that an object has been modified through the system and to determine the possible impacts of this change ?*

- *how to visualize the system and the impacts of a set of changes on it ?*

1.3. Criteria for Success

1. Description of a model for *Software Maintenance* process focused on *Impact Analysis.*

2. An *Impact Analysis* visualization system that will visually:

   a. represent the connectivity between objects,

   b. show the different impacts and their importances.

3. Evaluation of the *Impact Analysis* system.

1.4. Organisation of Thesis

The second chapter describes software maintenance in terms of its different activities, the different types of maintenance activities and the different kind of tools available to handle with the maintenance problem.

The third chapter focuses on the impact analysis in general.

The fourth chapter discusses graph theory and the need for displaying the dependencies with the representation of graphs and it also describes algorithms for the automatic placement of nodes and arcs in a graph.

The fifth chapter shows the results of impact analysis with the implementation of an impact analysis system.

Finally the sixth chapter includes the conclusion and further research.

2. Software Maintenance

2.1. Introduction

In this chapter the subject of software maintenance is placed in the *Software Engineering* context.

The notion of software process and software maintenance process are discussed. The main maintenance problems and the cost of *Software Maintenance* itself are identified.

2.2. Software Engineering Context

2.2.1. Software Engineering

The term *Software Engineering* was first introduced in the late 1960s at a NATO conference held to discuss what was then called the "software crisis" (*[SOMMERVILLE_92]*). This crisis arose with the introduction of third generation computer hardware because of their capacity and power meant that the applications which could not be built before were now feasible.

But the existing methods for building large software were not well enough defined and techniques applicable for small systems could not be scaled up. Many software projects were being delivered far behind the planned schedule, cost much more than originally expected, were unreliable, expensive and difficult to maintain and performed poorly.Software development was in crisis.

Now more than 20 years later, the software crisis still exists and has not been solved. Although many improvements have been made in Software management, engineering methods and techniques in tools for system developments and in the skills of development staff, the demand for software is increasing faster than improvements in software productivity.

The techniques of software engineering have been introduced in an attempt to reduce the cost of software system in the computer industry.

Software Engineering [BOEHM_76] is defined as:

> *"Software engineering involves the practical application of scientific knowledge to the design and construction of computers programs and*

the associated documentation required to develop, operate and maintain them."

or

Software Engineering [IEEE_90] is defined as:

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

Both of these definitions suggest that methods, procedures rules and principles are used in software engineering.

2.2.2. Software Engineering Processes

The identification of the "software crisis" in the late 1960s and the notion that software development is an engineering discipline led to the view that the process of software development is like other engineering processes *[SOMMERVILLE_92]*. Thus, a model of the software development process was derived from other engineering activities *[ROYCE_70]*. Because of the cascade from one phase to another, this model is known as the "waterfall" model (*Figure 1*).

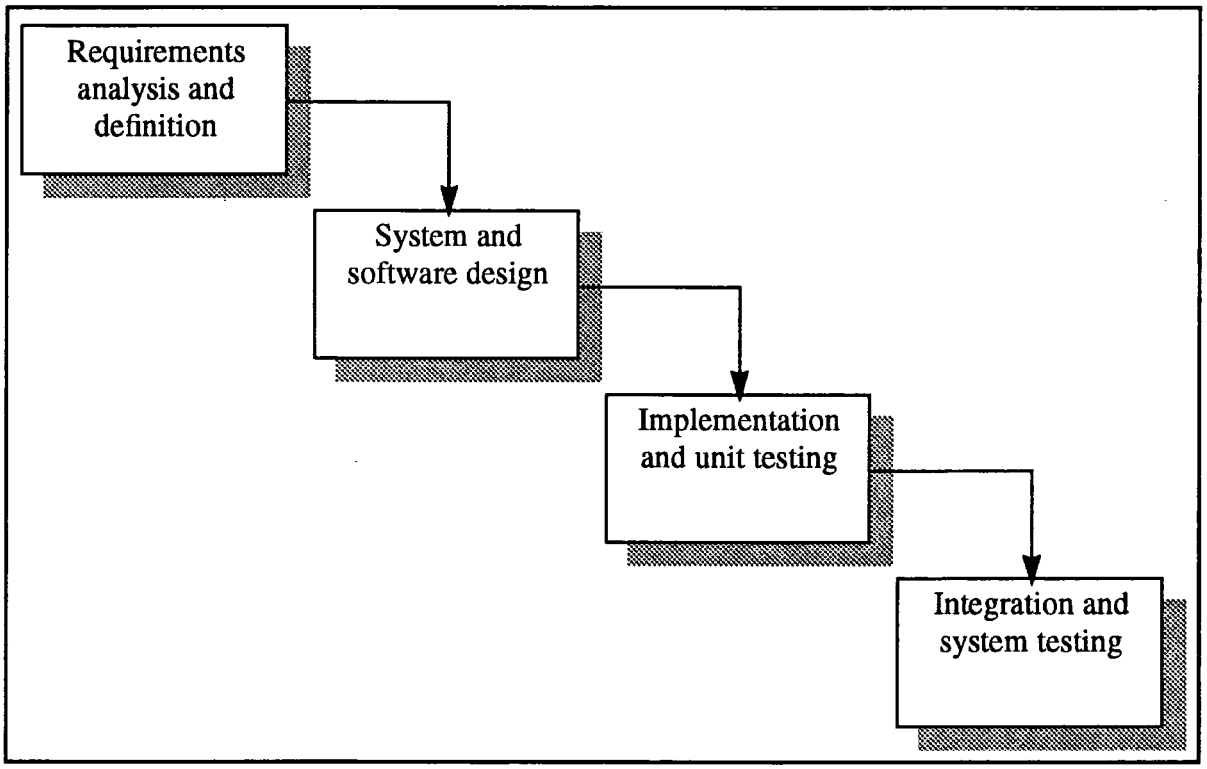


Figure 1 : The waterfall model of software development

However this development model soon appear only appropriate for some classes of software system and other development models have been created for satisfying other kinds of development:

- *Exploratory programming*: This approach involves developing a working system, as quickly as possible, and then modifying that system until it performs in an adequate way. This approach is usually used

in artificial intelligence (AI) systems development where users cannot formulate a detailed requirements specification and where adequacy rather than correctness is the aim of the system designers.

- *Prototyping:* This approach is similar to Exploratory programming in that the first phase of development involves developing a program for user experiment. However, the objective of the development is to establish the system requirements. This is followed by a re-implementation of the software to produce a production-quality system.

- *Formal transformation:* This approach involves developing a formal specification of the software system and transforming this specification using correctness-preserving transformations, to a program.

- *System assembly from reusable components:* This technique assumes that systems are mostly made up of components which already exist. The system development process becomes one of assembly rather than creation.

There are numerous variation of the simple process model of *Figure 1*, *Figure 2* shows an iterative and more complete one.

Development life-cycle [IEEE_83]: It is the period of time that begins with the decision to develop a software product and ends when the product is delivered. The development cycle typically includes a requirement phase, design phase, implementation/testing phase and integration/testing phase.

The *Software life-cycle [IEEE_83]* It is the period of time that starts when a software product is conceived and ends when the product is no longer available for use. The software life-cycle typically includes the development life-cycle and the operation and maintenance phase.

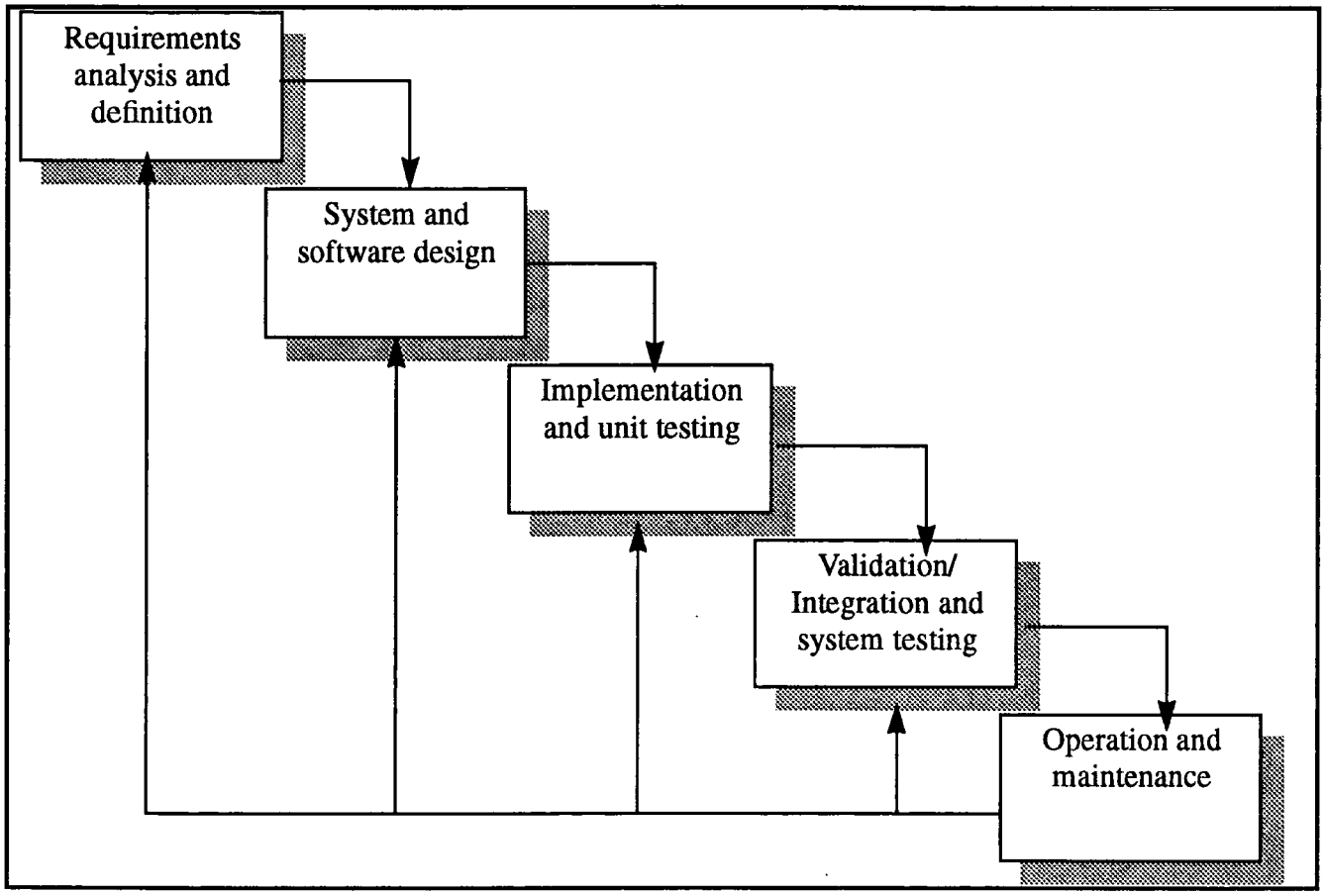


Figure 2 : The software life-cycle

- The **requirement phase** *[IEEE_83]* is the period during which the requirement for a software product, such as the functional and performance capabilities are defined and documented.

- The **design phase** *[IEEE_83]* is the period of time during which the designs for architecture, software components, interfaces, and data are created, documented, and verified to satisfy requirements.

- The **implementation/testing phase** *[IEEE_83]* is the period of time during which a software product is created from design documentation and debugged. Design must be translated into a machine executable form. The coding step accomplishes this translation through the use of

conventional programming languages (i.e., C, Ada, Fortran, Cobol, Pascal,...) or so-called Fourth generation languages.

- The **validation/integration/testing phase** *[IEEE_83]* is the period of time during which the components of a software product are validated and integrated. The software product is validated to determine whether or not requirements have been satisfied. Testing is a multi-step activity that serves to verify that each software component properly performs its required functionality with respect to the specifications and validates that the system as a whole meets overall customer requirements. The integration has the purpose to install the different components of the system in a same environment, the testing ensures that it performs as required.

- The **operation and maintenance phase** *[IEEE_83]* is the period of time during which a software product is employed in its operational environment, monitored for satisfactory performance, and modified as necessary to correct problems or to respond to changed requirements.

The Spiral model of the software process ([BOEHM_88]) *(Figure 3)* has been evolving for several years, based on experience with various refinements of the waterfall model as applied to large government software projects. The Spiral Model has as its major distinguishing feature the fact that it creates a risk-driven approach to the software process rather than a primarily document-driven or code-driven process.The Spiral Model can accommodate most previous models as special cases and further provide guidance as to which combination of previous models best fits a given software situation.It incorporates many of the strengths of other models and resolves many of their difficulties.

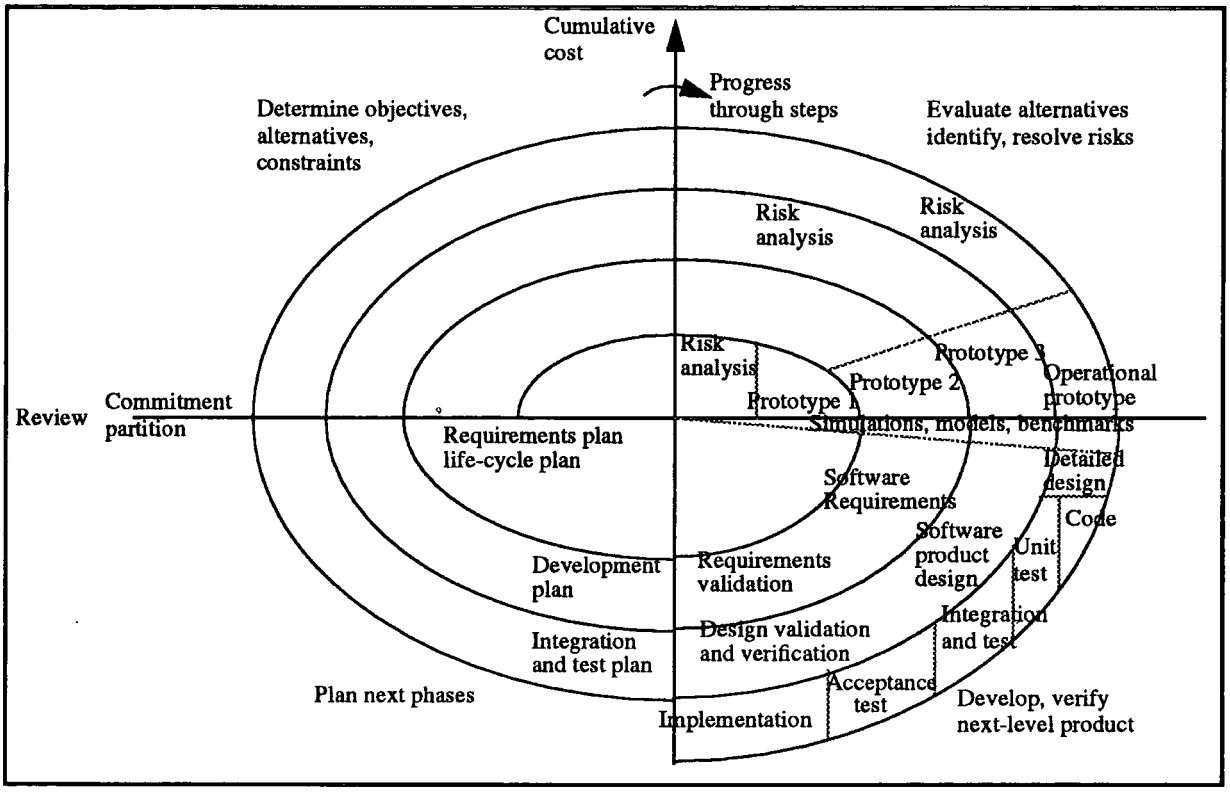


Figure 3 : Spiral model of the software process (*[BOEHM_88]*)

Once the system has been released the maintenance process begins.

2.3. Types of Software Maintenance

Software Maintenance [IEEE_90] has been defined as:

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

Software Maintenance has become established as a sub-discipline within the general field of *Software Engineering*. This has not always been the case, with software maintenance being given very low status by the software engineering community.*Software Maintenance* is a complex and serious problem, serious because of the costs, and complex because of the wide range of activities involved (i.e. requirement analysis, program comprehension, impact analysis, test,...).Over the life of software the *Software Maintenance* effort has been estimated to be frequently more than 50% of the life-cycle costs *[LIENTZ_80]*.

Software maintenance has been divided into four categories *[LIENTZ_80]*: Perfective maintenance, Adaptive maintenance, Corrective maintenance and Preventive maintenance. These terms have been widely adopted in industry and form a useful distinction in classifying types of software maintenance.

* *Perfective maintenance* It means changes which improve the system in some way without changing its functionality. It includes all changes, insertions, deletions, modifications, extensions, and enhancements which are made to the system to meet the evolving and/or expanding needs of the users. As a simple example, a tax program may need to be modified to reflect new tax laws but, usually, modifications are much more substantial.

* *Adaptive maintenance* It is the maintenance which is required because of changes in the environment of the software system. New versions of the operating system, new or different hardware are for instance modifications in the environment which necessitates *Adaptive maintenance*.

* *Corrective maintenance* It is the correction of previously undiscovered

system errors. It refers to changes necessitated by actual errors in a system. Under management pressure, emergency repairs may be undertaken ('patching') which often cause considerable problems later.

* *Preventive maintenance* It includes the activities designed to make the code, design and documentation easier to understand and to work with, such as restructuring or documentation up-dates. This type of maintenance usually improves the maintainability of the system. This type of maintenance is only referred by some authors (*[LIENTZ_80], [ARNOLD_82], [PRESSMAN_85]*).

The result of a survey *[LIENTZ_80]* discovered that about 50% of maintenance was perfective, 25% adaptive, 21% corrective and 4% preventive (*Figure 4*).

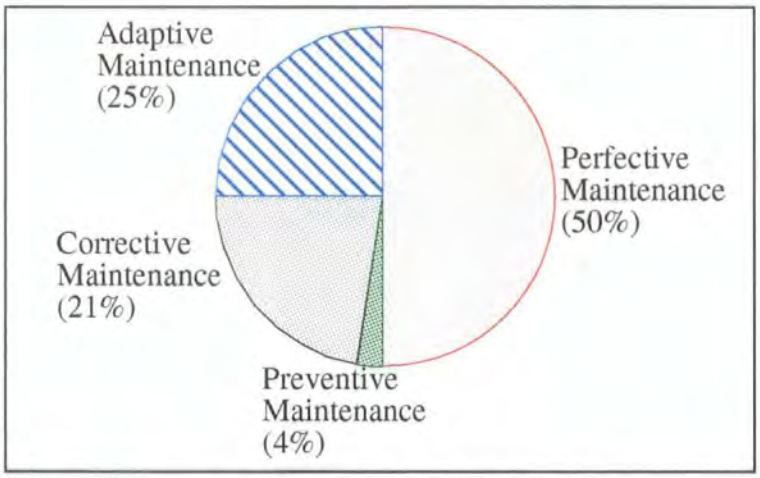


Figure 4 : Maintenance effort distribution

Coding modifications (Corrective) are usually relatively cheap to do; design modifications (Adaptive) are more expensive as they may involve the rewriting of several program components. Requirements modifications (Perfective) are the most expensive because of the redesign which is usually involved.

2.4. Software Maintenance Process

Sommerville *[SOMMERVILLE_92]* (with preventive maintenance added) describes a maintenance process as shown in *Figure 5.* The maintenance process is triggered by a set of change requests from system users or management. The costs and impact of these changes are assessed and, assuming it is decided to accept the proposed changes, a new release of the system is planned. This release will usually involve elements of perfective, adaptive, corrective and maybe preventive.

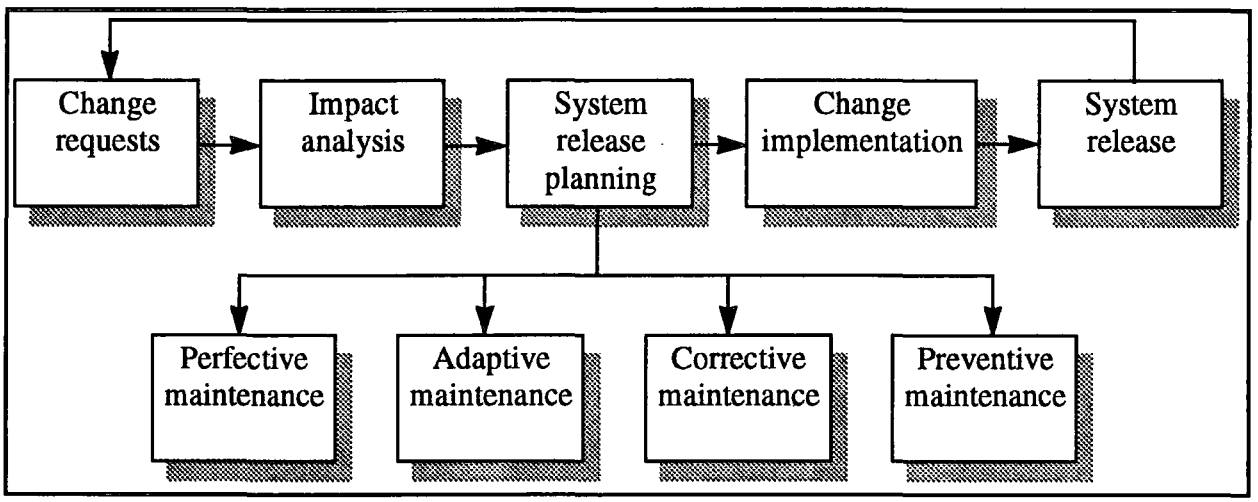


Figure 5 : A maintenance process model

The process model of *Figure 5* emphasizes the different types of Software Maintenance (i.e. perfective, adaptive, corrective and preventive maintenance).

First a Change request arrived, a study of the *Ripple Effects* by *Impact Analysis* has to be done to evaluate the cost of this change. If this change is accepted, a planning of the implementation is done and the change is made according to the type of maintenance, the system is then released to the users after the implementation.

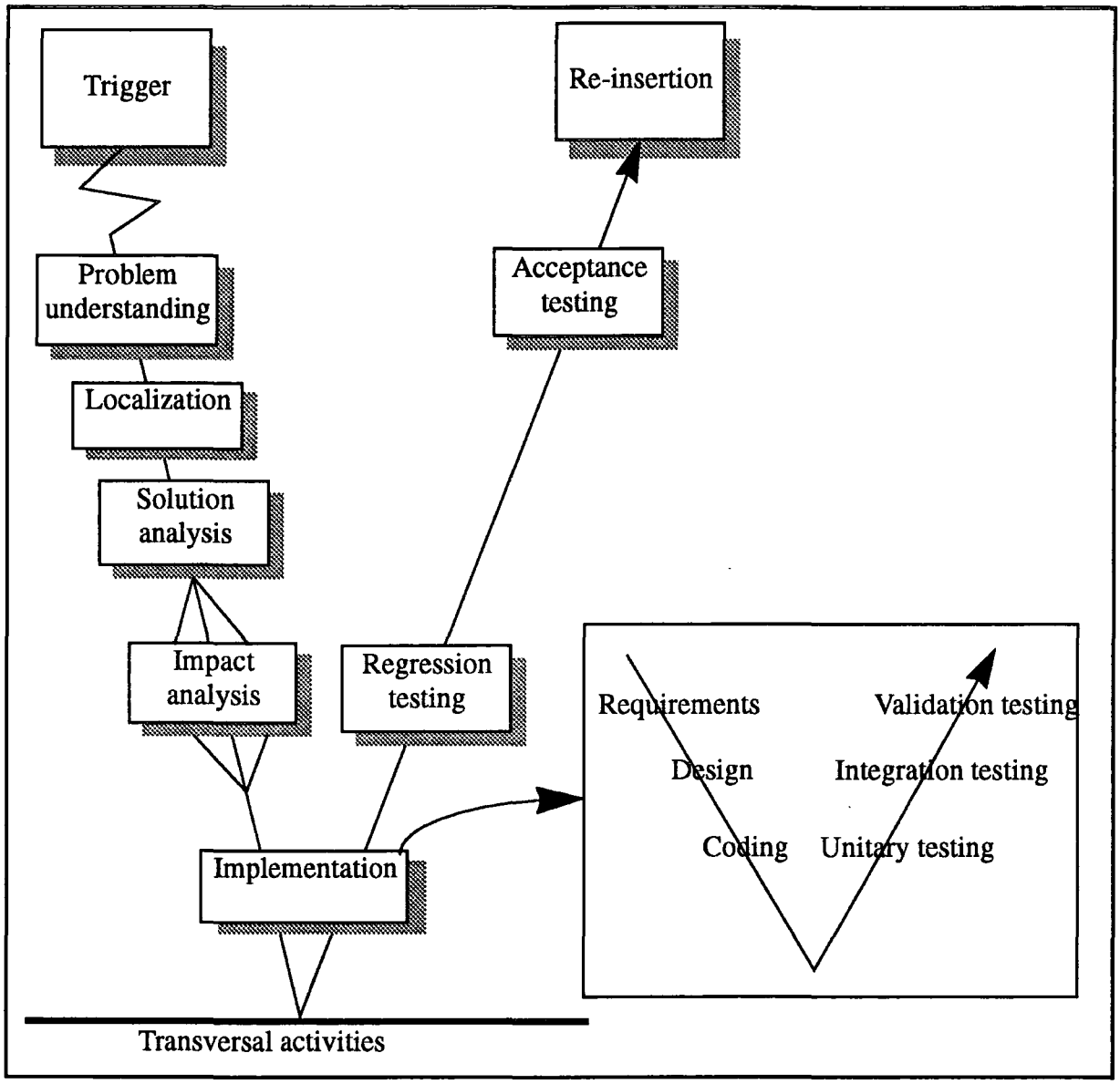


Figure 6 : The ESF/Epsom project maintenance process model

Another software maintenance process model has been defined by Harjani *[HARJANI_92]* shown in *Figure 6.* which emphasizes the understanding of the problem and the different solutions together with an estimation of the cost of the change (Solution Analysis and Impact Analysis).The process model of the *Figure 6* focuses particularly on the first phase of the change: Understanding the change that has to be made (What and Where the change has to be done), and choosing the solution which requires the lowest cost in terms of time, money, etc. by using *Impact Analysis* techniques.

2.5. Software Maintenance Problems and Costs

Traditionally, *Software Maintenance* has not been part of the software life-cycle in the same sense as the earlier stages of the life-cycle, but rather occupies a detached position and is considered as a post-delivery activity. The word 'maintenance' carries connotations of less intellectual activity than 'design' because:

* A large number of people consider maintenance as just correction of errors resident in the software after a release of the system,

* Many people consider money spent in *Software Maintenance* as wasted because they do not think that the system will be changed/enhanced later and do not consider spending money for a proper understanding/documentation/ design of the system,

* Maintenance is always under budgetary pressures as this activity usually comes at the end of the project and with the end of the budget.

There is no process model of the software maintenance which is completely accepted by the software engineering community, this implies *[SIMON_91]*:

* A lack of management of software maintenance,

* A lack of understanding of how to maintain a software system,

* A lack of historical data on maintenance and error histories,

* Difficulties in estimating the cost of modifications.

The implementation and documentation of the software system can cause problems because:

- several programming languages in a software system means problems of communication between them,
- poor software design can mean inadequacy of the implementation,
- poorly coded software (few comments, poorly structured programs, use of non-standard language features of the compiler,...) is difficult to understand.
- no documentation or inadequacy of the documentation with the software code.

All these problems (and many others) are the cause of the high cost of the *Software Maintenance* and therefore there is a need to use more methods and tools to reduce these increasing costs, the trend has been for the cost of *Software Maintenance* to increase *(Figure 7)*.

% of the software budget	1970s	1980s	1990s
Development	60-65 %	40-60 %	20-30 %
Maintenance	35-40 %	40-60 %	70-80%

Figure 7 : Escalating maintenance costs *([PFLEEGER_87], [PRESSMAN_85])*.

2.6. Software Maintenance Tools

One way to overcome some of the costs of *Software Maintenance* is to provide tools to help the software maintainers. A classification of *Software Maintenance* tools has been detailed by Simon *[SIMON_91]*:

- Tools for Program Comprehension: static code analysers, code visualisation, cross referencers, source code comparisons, debuggers.
- Tools for Reverse Engineering: restructurers, reformatters, re-engineering, reverse engineering.
- Tools for Testing: regression testing, test coverage monitor.
- Tools for Software Management: software configuration management, product management.

This classification corresponds in fact to the different phases of the software maintenance process. Because a system exists before any maintenance is performed it has to be understood and analysed, the tools for this purpose are Program Comprehension and Reverse-Engineering tools. As a change on the system is done, tools for testing are required to know about the possible errors which are the cause of this change. The tools of Software Management are the final stage of the maintenance activity, they are used to keep the system under control after delivery to the users.

2.7. Ripple Effects and Impact Analysis

Ripple Effects are the phenomena by which changes to one program have tendencies to be felt in other program areas.

Impact Analysis is

> *The task of assessing the effects of making a set of changes to a software system.*

The study of the *Ripple Effects* is a very important phase in the Software Maintenance process as it is used to determine several attributes of the change requested: areas affected (or potentially affected) by the initial change and by using these the cost of the change can be computed in term of number of people to work and money.

The next chapter is focused on the *Impact Analysis* process and explains all the activities involved.

2.8. Summary

Software Maintenance has been defined in terms of categorisation of tasks and different categories of maintenance (i.e. perfective, corrective, adaptive and preventive) have been explained. At the time, the traditional software life-cycle model was established, software maintenance had not assumed the great importance it has today, and so the model was oriented almost exclusively to the development of software. Consequently software maintenance has found its niche within the model by default. The Software life-cycle is product-based and the process that has created the product is not mentioned with all management activities. Therefore, there is an important area of research on the modelling of all activities involved in the software development and maintenance process.

3. Impact Analysis

3.1. Introduction

Once software systems have been installed they are often changed to reflect changes in other sub-systems with which they are connected. One of the reasons why software maintenance is so difficult is that changes made at one point in a software system may have severe *Ripple Effects [YAU_78]* at other points. It is generally accepted that maintainers need to analyse the impacts of any proposed change to establish its correctness. However the term *Impact Analysis* seems to be used in many different ways so there is no clear consensus as to when and how *Impact Analysis* should be carried out, when it is complete, or even exactly what the objectives of impact analysis should be *[WILDE_94]*.

Impact Analysis [WILDE_94] is:

> *"The task of assessing the effects of making a set of changes to a software system."*.

Because these effects are not limited to the code, *Impact Analysis* must consider impacts on design and specification as well as on code. As a consequence of the wide area of information (components) and primitive automatic production of relationships (dependencies) between the components, *Impact Analysis* is a difficult task and requires inputs from the user concerning the model of the system.

Impact Analysis is viewed as a necessarily approximate technique which must focus on the cost-effective minimization of unwanted side-effects.

3.2. Impact Analysis Concepts

The scope of *Impact Analysis* is the software and all related documentation, including graphics, of a system. It thus encompasses the following:

- Source code,
- Generation and installation procedures,
- Data files which may be required to execute properly the software,
- Usage documentation and operational procedures,
- Development documentation or maintenance documentation,
- Test cases,
- ...

The goals of the impact analysis are *[ARNOLD_93]*:

- To understand rapidly the consequences of changes and avoid errors,
- Develop more effective test cases,
- Give change impact information to managers,
- Warn that a modification may be dangerous, so that
  - a simpler solution should be found
  - extensive testing undertaken
- Provide a quick check on the impacts of a change ("scope out" a problem)

Software change is the biggest part of *Software Maintenance*, and *Impact Analysis* is usually required for making software changes, so *Impact Analysis* is extremely important.*Impact Analysis* requires there to be a model of the system on which this analysis can be done.

This model is constructed from the life-cycle documents requirements, design, code, test cases, etc. and can be represented as dependencies between the different components of the system. The construction of this model is usually done from the static description of the system and requires static documentation/code analysers.

Static impact analysis analyse the impacts on the static structure of a program and Dynamic impact analysis analyse the impacts on the objects accessed during program execution.

The model of the system has to be created before the task of impact analysis, then a set of changes on the object has to be defined and modelled. The maintainers use their knowledge of the system to assess the change set, attempting to demonstrate that the change has been correctly bounded and that all the components that need to be changed have been identified. Once the change set and the model of the system have been defined, *Impact Analysis* (usually the maintainer apply some propagation rules (even if unconscious of it) for the impacts) has to perform the propagation of these initial modifications through the dependencies of the system and produces the resulting impacts.

Note that it is probably not feasible to foresee all the impacts of a change *[WILDE_94]*.

First, because the complexities of real programming language structures such as pointers, virtual functions and so on make the collection of a complete representation of the system at the code level very difficult.

Second, some kinds of relationships such as timing interactions, may be data dependent, while others may depend on the intricacies of particular compilers, operating systems or subroutine libraries. It is probably impractical to expect tools to have complete knowledge of all the possible problems.

Impact Analysis may be performed "a posteriori" (i.e. after the change has been implemented) or predictively (before the implementation of the change). The first case is often called "regression analysis", and that kind of analysis is generally performed through "regression testing", the objective of which being to check that parts of the system that have not been intentionally changed are still performing. The use of "*Impact Analysis*" will be reserved for the case where the analysis is performed before the actual implementation of the change, with the objective of gathering information on the impacts of the planned change, precisely in order to take the decision whether to implement the change or to choose between several proposed implementations of the change. Another difference between "a priori" *Impact Analysis* and "a posteriori" regression analysis is the degree of definition of the change under study, this one still being in process of elaboration in the first case, while it is already

implemented (and thus presumably perfectly defined and known) in the second case *[QUEILLE_93]*.

Impact Analysis is an analysis which begins before the real implementation of the set of changes and will only give a sub-part of the possible impacts. Its principal goal is to reduce the cost of post-implementation discovered side effects of a change.

3.3. Impact Analysis Process

3.3.1. Decomposition and Representation of the System

Impact Analysis requires the constitution of a representation of the system on which the analysis has to be carried out. This activity is called Decomposition [ARNOLD_93]. The inputs to this decomposition process can be source code, documentation, as well as the knowledge of the maintainer/developer on the system. *Figure 8* shows how the representation of a system/part of a system can be produced from different sources of information. A single formalism to represent all types of *Components* and all types of dependencies between them is important in order to be able to do *Impact Analysis* on all the parts of the system at the same time (code, documentation,...) and through the dependencies between these different parts.

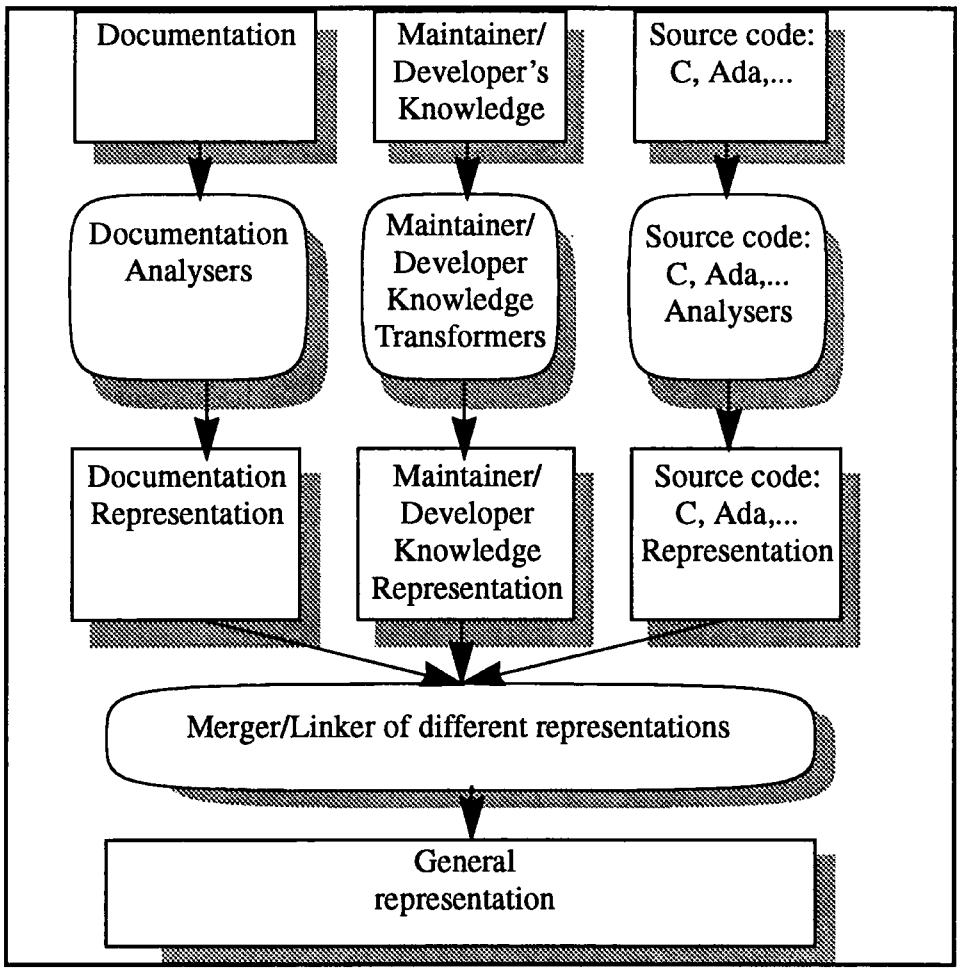


Figure 8 : Decomposition and General representation.

However *Impact Analysis* can be applied on some parts of the system and does not require that all the system is modelled during the task, but in this case the impacts will be characterized only on the parts of the system on which the task is performed.

The system can be modelled as a network of objects and links. In this network, objects represent the different *Components* of the system, at various granularity levels and they are typed according to the nature of the components they represent. Links represent dependency relationships between the components corresponding to the objects they connect; they typed according to the nature of the dependency relationship they represent. For instance, the dependency relationship between a source code module and the unit tests which test it may be represented by "test/is-tested-by" links between the object which represents this module and the objects which represent the corresponding unit tests.

Various approaches have been taken by different researches in primarily modelling source code (*[GOPAL_89]* and *[MOSER_90]* for Ada, *[WILDE_89],[WILDE_87]* for C, *[NARAYANASWAMY_88]* for Common Lisp, *[COLBROOK_89]*, *[GALLAGHER_91]*, *[CALLISS_90],[CALLISS_89],[CALLISS_88],[CANFORA_93],[JIANG_91],[LYLE_89]* using Program Slicing for Control or Data flow or Abstract Data types from C programs).

One of the few works at the documentation level *[TURVER_93c]* models the interconnections between documentation entities for the purpose of "Early Ripple Propagation".

Here are some examples of objects and links generally exhibited at the code level (Wilde *[WILDE_87]*):

- Data flow dependencies, between two objects, occur when the value of one object is used to compute the value of another object.
- Definition dependencies occur when one program entity is used to define another one.
- Calling dependencies occur when one function calls another one.
- Functional dependencies occur when a global data object is created or updated by a module.

Many other kinds of dependencies at the source level have been defined by Wilde *[WILDE_87]*, only a few are presented here.

Dependencies are often represented by graphs. For instance *Figure 9* shows a dependency graph for a small C program.

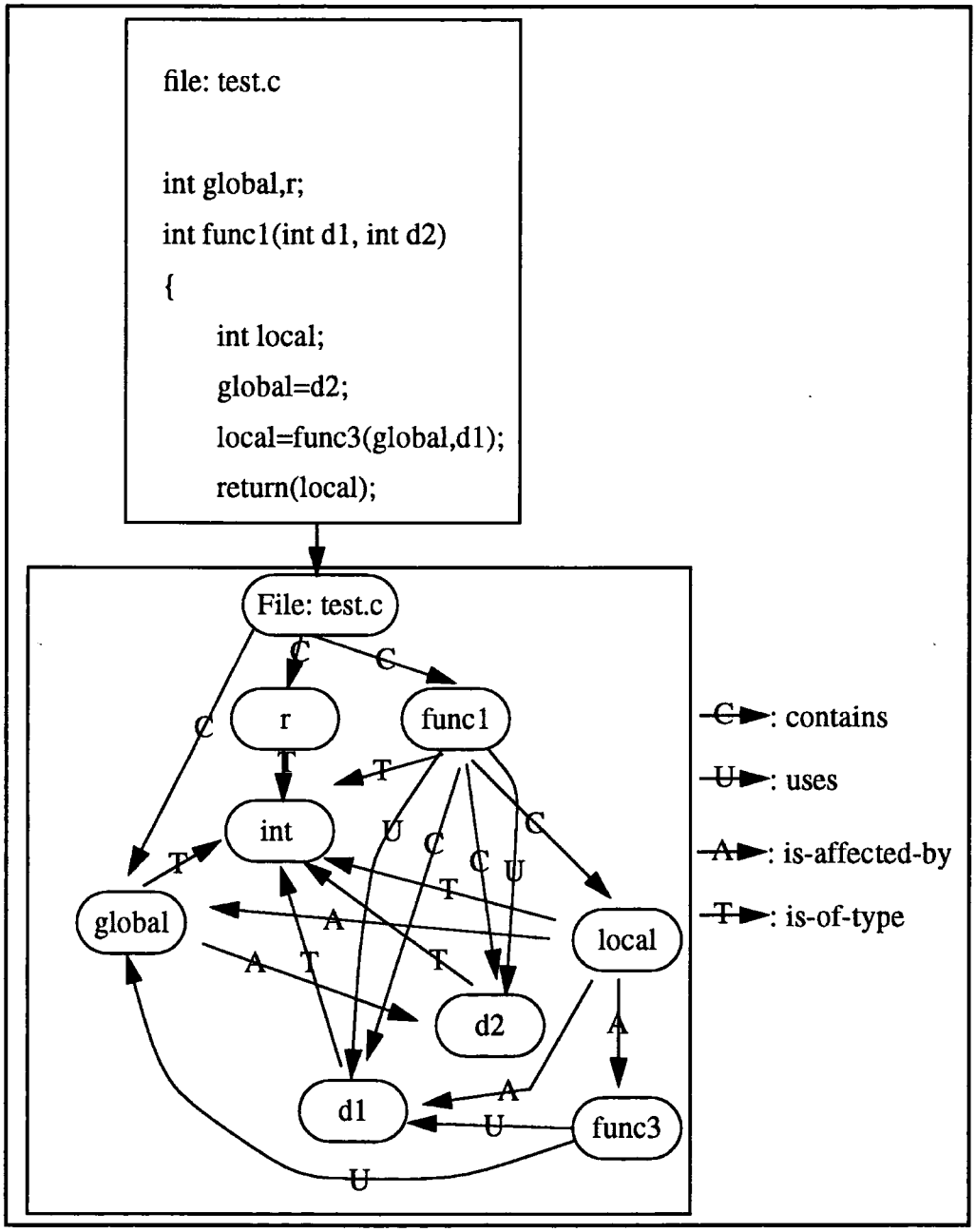


Figure 9 : Possible dependencies for C program.

Figure 9 shows the dependencies between the different objects of the small C program. For instance: the file (test.c) contains (-C->: is linked with the type

of link "contains") three objects (variables "global" and "r" and function "func1") and these objects are linked also to the object "int" (which symbolises the type integer) because of their type or their return value type. The variable "local" is-affected-by (-A->) the result of the function "func3" which uses (-U->) the variables "global" and "d1".

3.3.2. Specifying Changes and Defining a Change Set

Changes are generally proposed by a variety of sources *[ARNOLD_93]*: users, managers, programmers, analysts, contractors, customers, market conditions, computer conferences, legal changes. All the proposed changes are stored and are reviewed before being permitted to continue. In the next step the importance of changes is considered in terms of cost, usefulness, etc. The last step will be the certification of the change.

Related changes have to be grouped together and dependent changes can be ordered so that more independent changes are performed first.

Change proposals are usually described in natural language description and a process of transfer of this description to change set (modifications on the dependency graph) has to be done. This task could be performed semi-automatically if the language used in the change proposals is close enough to the impact analysis' change model. However this task is usually completed only by the maintainer.

The change set is a set of couples of the form (object, modification type), specifying the type modification to be applied to an object.

3.3.3. Propagating Change Set, Obtaining Impact Set

From a change set and the assessment of these changes, it is necessary to find out what other objects are affected. This task is performed automatically by the impact analysis tool from a model of the system, a model of propagation (The model of propagation is composed of *Propagation rules*) and a change set.

Impact Analysis gives the maintainer an idea which parts of the system are affected by the proposed changes and allows him to evaluate the consequence, in terms of objects to modify, of the requested changes.

3.3.4. Understanding and Analysing the Impact Set

The *Impact Analysis* tool should allow the user to understand why an object needs to be modify, that is to say from which previous change and from which *Propagation rule* this object has been affected. This feature is required because of the propagation system may respond "maybe this object has to be modified" and in this case the maintainer has to validate or invalidate the "potential" impact proposed by the *Impact Analysis* tool.

3.4. Summary

The *Impact Analysis* in this chapter has been described considering all the activities involved in it. That is to say, the modelling of the system and the generation of a object/link model of the system, the modelling of the change request, the propagation of the change and finally the visualization of the result. In the next chapters, the visualization aspects of the *Impact Analysis* tool will be explained in details: graph representation and interaction with the model of the system.

4. Graph Theory and Representation

4.1. Introduction

Many areas of Computer Science involve the drawing of a graph on a 2-dimensional surface. These include design, diagrams for information systems, algorithm animation, circuit schematics and network presentation, etc..... In this chapter the principal notions of the graph theory and some algorithms for automatic placement of nodes and links in a network for the purpose of visualization will be presented.

4.2. Graph Theory

4.2.1. The Notion of Graph

A graph G=(N, A) consists of [CARRE_91]:

- A finite set $N=\{n_1,n_2,...,n_n\}$, the elements of which are called nodes.
- A subset A of the Cartesian product N x N, the elements of which are called arcs.

A graph can be depicted as a diagram in which nodes are represented by points in the plane, and each arc (n_i,n_j) is indicated by the arrow drawn from the point representing n_i to the point representing n_j.

Example 1 : Flowcharts and control-flow Graphs. A hardware integer division procedure is shown in *Figure 10* and a familiar flow-chart representation of this function is shown in *Figure 11*, the control-flow graph of the function is shown in *Figure 12*.

```
void division( int x, int y, int *q, int *r)
/* pre: x>0 ; y>0 ; post: x=(*q)*y+(*r) ; 0<=(*r)<y */
{
     int w;
     *r=x; *q=0; w=y;
     while (w<=x) { w*=2; }
     while (w!=y) {
             *q=2*(*q);
             w=w>>1; /* logical shift right , division by 2 */
             if (w<=(*r)) {
                  *r=*r-w; *q=*q+1;
             }
     }
}
```

Figure 10 : Hardware integer division algorithm

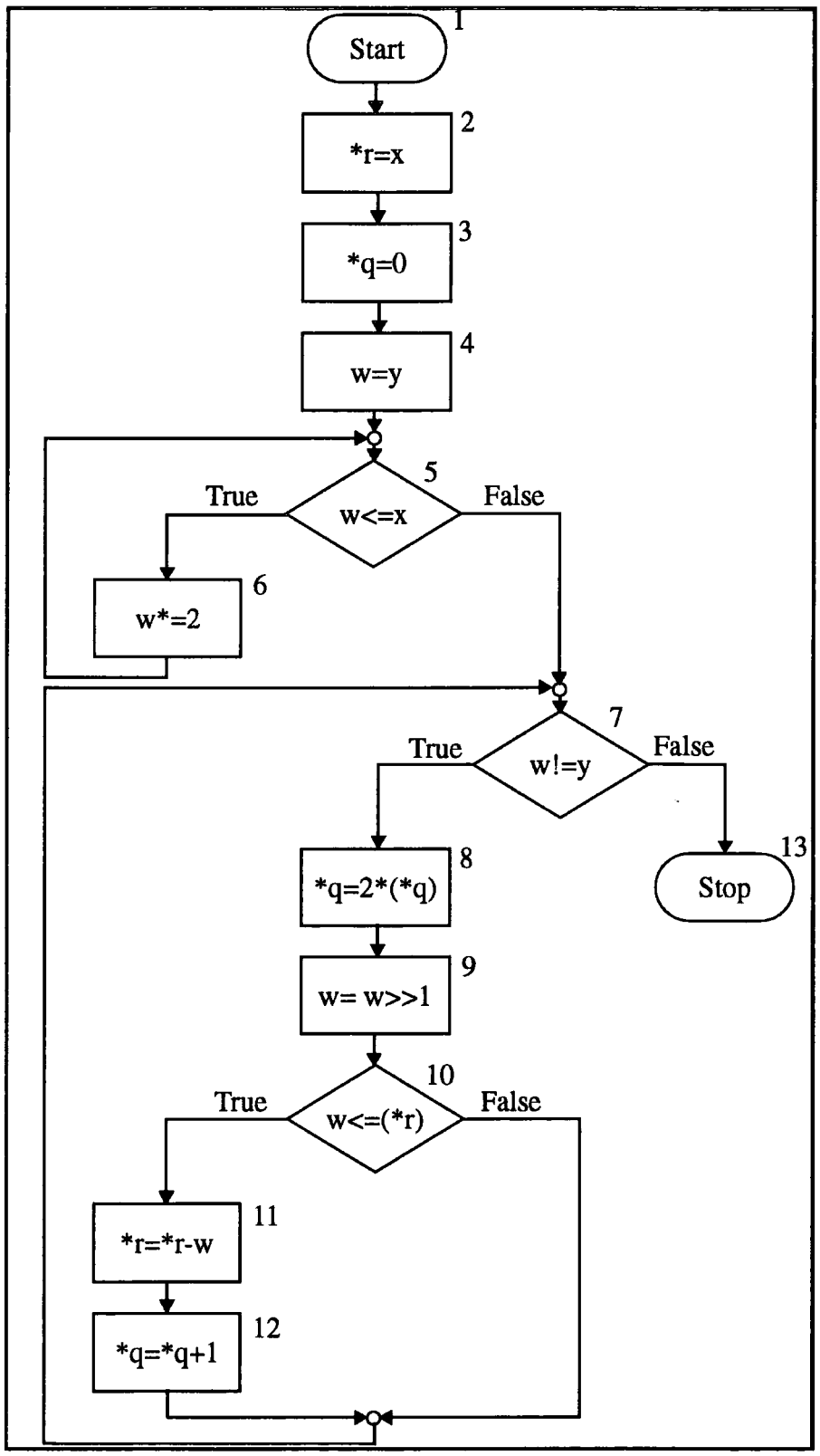


Figure 11 : Flow-chart of the function shown in *Figure 10.*

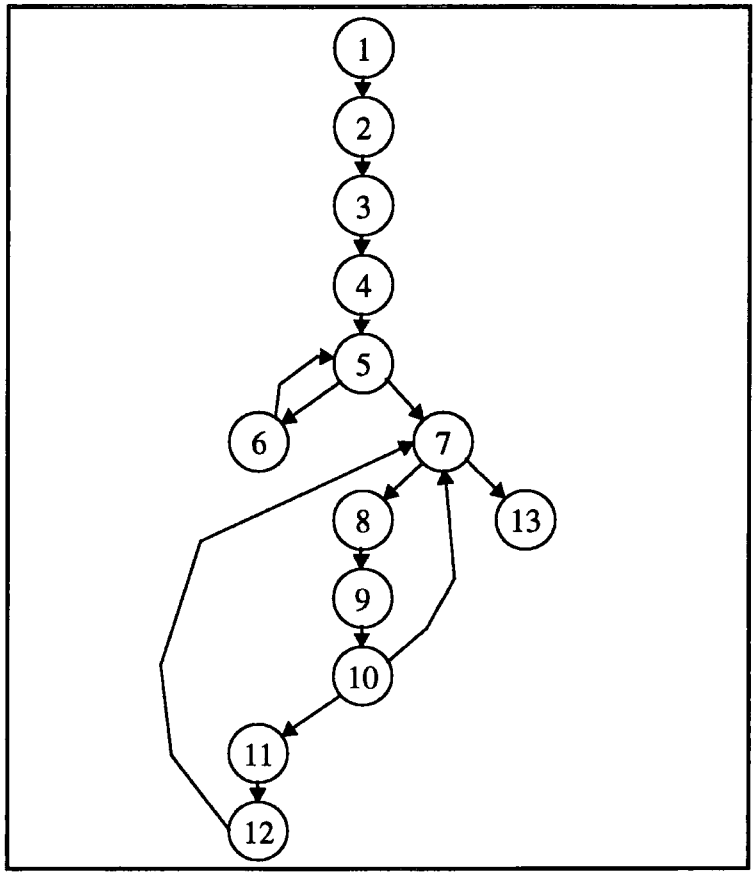


Figure 12 : Control-Flow of the function shown in *Figure 10.*

4.2.2. Basic Terminology

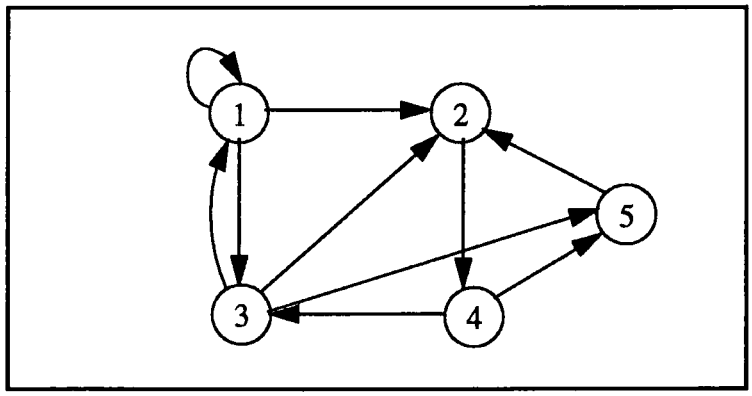


Figure 13 : Example of graph

- Head and tail:

  In text, it is customary to represent an arc (u, v) as u->v. We call v

the head of the arc and u the tail with the notion that v is at the head of the arrow and u at its tail.

For example, 1->2 is an arc of *Figure 13*; its head is node 2 and its tail is node 1. Another arc is 1->1; such an arc from a node to itself is called a loop. For this arc, both the head and the tail are node 1.

- Predecessors and Successors:

  When u->v is an arc, we can also say that u is a predecessor of v, and that v is a successor of u. Thus, the arc 1->2 tells us that 1 is a predecessor of 2 and that 2 is a successor of 1. The arc 1->1 tells us that node 1 is both a predecessor and a successor of itself.

- Labels:

  It is permissible to attach a label to each node. Labels will be drawn near their node. Similarly, we can label arcs by placing the label near the middle of the arc. Any type can be used as a node label or an arc label.

  For instance, *Figure 14* shows a node named 1, with a label "dog", a node named 2, labelled "cat", and an arc 1->2 labelled "bites".

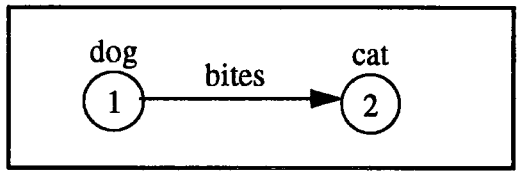


Figure 14 : A labelled graph with two nodes.

We should not confuse the name of a node with its label. Node names must be unique in a graph, but two or more nodes can have the same label.

- Paths and Length of a path

  A path in a directed graph is a list of nodes (v_1, v_2,... v_k) such that there is an arc from each node to the next, that is, v_i->v_i+1 for i=1, k-1. The length of the path is k-1, the number of arcs along the path.

  For example (1,2,4) is a path of length two in *Figure 13*. The trivial case k=1 is permitted. That is, any node v by itself is a path of

length zero from v to v. This path has no arcs.

- Cyclic and Acyclic Graphs

A cycle in a directed graph is a path of length 1 or more that begins and ends at the same node. The length of the cycle is the length of the path. Note that a trivial path of length 0 is not a cycle, even though it "begins and ends at the same node". However, a path consisting of a single arc v->v is a cycle of length 1.

Example: Consider the graph of *Figure 13*. There is a cycle (1,1) of length 1 because of the loop 1->1. There is a cycle 2 because of the arcs 1->3 and 3->1. Similarly, (2,4,3,2) is a cycle of length 3, and (2,4,3,5,2) is a cycle of length 4. Note that a cycle can be written to start and end at any of its nodes. That is, the cycle (v_1,v_2,... v_k, v_1) could also be written as (v_2,... v_k, v_1, v_2) or as (v_2,... v_k, v_1,v_2,v_3) and so on. For example, the cycle (2,4,3,5,2) could also be written as (3,5,2,4,3).

On every cycle, the first and last nodes are the same. We say that a cycle (v_1,... v_k,v_1) is SIMPLE if no node appears more than once among v_1,...v_k; that is, the only repetition in a simple cycle occurs at the final node.

- Cyclic Graph

If a graph has one or more cycles, we say that the graph is cyclic. If there are no cycles, the graph is said to be acyclic. By the arguments used above about a simple cycle, a graph is cyclic if and only if it has a simple cycle, because if it has any cycles at all, it will have a simple cycle.

- Undirected graphs

Sometimes it makes sense to connect nodes by lines that have no direction, called edges. Formally, an edge is a set of two nodes. The edge {u, v} says that nodes u and v are connected in both directions. If {u, v} is an edge, then nodes u and v are said to be adjacent or to be neighbours.A graph with a symmetric arc relation, is called an

undirected graph.

- Paths and cycles in Undirected Graphs

  A path in an undirected graph is a list of nodes $(v_1,...v_k)$ such that each node and the next are connected by an edge. That is, $\{v_i,v_{i+1}\}$ is an edge for i=1,...k-1. Note that edges being sets, do not have their elements in any particular order. Thus, the edge $\{v_i,v_{i+1}\}$ could just as well appear as $\{v_{i+1},v_i\}$. The length of the path $(v_1,...,v_k)$ is k-1. As with directed graphs, a node by itself is a path of length 0. Defining cycles in undirected graphs is a little tricky. The problem is that we do not want to consider a path such as (u, v, u) which exists whenever there is an edge {u, v}, to be a cycle. Similarly, if $(v_1,...,v_k)$ is a path, we can traverse it forward and backward, but we do not want to call the path $(v_1,... v_{k-1},v_k,v_{k-1},...,v_1)$ a cycle.

- Simple cycle

  Perhaps the easiest approach is to define a simple cycle in an undirected graph to be a path of length three or more that begins and ends at the same node, and with the exception of the last node does not repeat any node. The notion of a non-simple cycle in an undirected graph is not generally useful, and we shall not pursue this concept.

- Initial and terminal end-points of an arc:

  For an arc (n_i,n_j), the node n_i is the initial end-point and the node n_j is the terminal end-point.

- Arcs incident to and from a node:

  If an arc A has a node n_i as its initial end-point, we say that the arc is incident from n_i; whereas if an arc A has node n_j as its terminal end-point we say that arc A is incident to n_j. The number of arcs incident from a node n_i is called the out-degree of n_i and it is denoted by $\rho^+(n_i)$; while the number of arcs incident to n_j is called the in-degree of n_j and is denoted $\rho^-(n_j)$.

- Partial graphs:

  If we remove from a graph G=(N,A) a subset of its arcs, we are left with a graph of the form: $(H = (N, A'))$, $where\ (A' \subseteq A)$ which is called a partial graph of N.

- Sub-graphs:

  If we remove from a graph G=(N,A) a subset of its nodes, together with all the arcs incident to or from those nodes, we are left with a graph of the form:

  $$(H = (N', A')),\ where\ (N' \subseteq N)\ ,A' = A \cap (N' \times N')$$

  which is called a sub-graph of N. We may describe H more precisely, as the sub-graph of G generated by N'.

See also [CARRE_91], [DEO_74].

4.3. Graph Representation

4.3.1. Introduction

Various algorithms have been proposed for producing graph drawings that are aesthetically pleasing (depending on the structure of the graph: Tree, Directed Graph, etc.).

In this section we present some of these algorithms and their results are presented. These algorithms are designed to produce aesthetically pleasing drawings of graphs. A graph drawing algorithm reads as input a combinatorial description of a graph G, and produces as output a drawing of G according to a given graphic standard.

4.3.2. Graph Drawing

A graph is composed of nodes and arcs between these nodes. Various graphic standards have been proposed for the representation of graphs in the plane. Usually nodes are represented by symbols such as circles or boxes, and each arc (n_i,n_j) is represented by a simple open curve between the symbols associated with the nodes n_i and n_j *[BATTISTA_93]*.

A drawing such that each edge is represented by a polygonal chain is a polyline drawing (*Figure 15*). There are two common special cases of this standard. A straight-line drawing maps each arc into a straight-line segment (*Figure 16*). This standard is commonly adopted in graph theory text.

An orthogonal drawing maps each arc into a chain of horizontal and vertical segments (*Figure 17*). Entity relationship graphs in database design are usually drawn according to this standard. The polyline drawing can be modified to use a curved representation of the arcs.

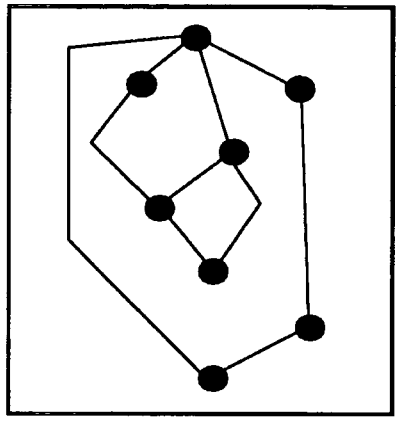


Figure 15 : Polyline drawing

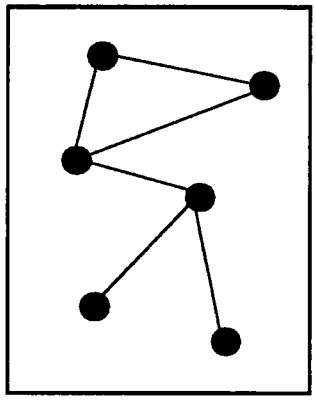


Figure 16 : Straight-line drawing

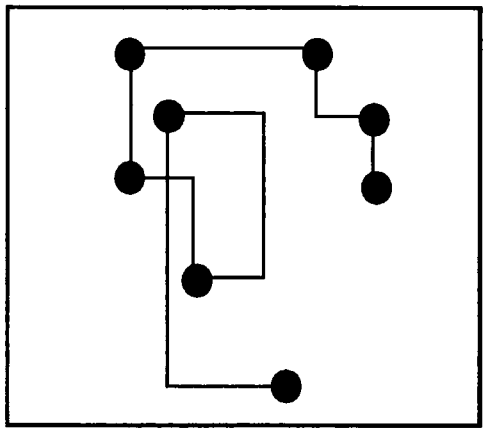


Figure 17 : Orthogonal drawing

Each node and arc have attributes which may have a graphical representation depending on the value of this attribute and of its type.

For instance the type of a node may be represented by the form of the graphical object (Square, Circle, Drawing,...).

Some of the possible graphical attributes are:

* Forms (Square, Circle, Drawing,...),
* Size,
* Colours,
* Type of drawing of lines (Solid, Dashed, Arrows,...),
* Text and Position of this text.

A matching between internal attributes of node/arc and graphical object/link has to be done in order to display part of the information contained in the graph with graphical aspects. *Figure 18* shows a possible representation of graph with colours, style of line, text to display attributes of nodes/arcs of a graph.

Figure 18 represents the modelling of a software system with several function ("main", "F", "G", "H", "F1", "F.c f11" (C "static" function), "F2", "F21", "F22", "G1", "G11", "G111", "G112", "G.c g13" (C "static" function) "H1", "H2", "H21", "H22", "H221") and with several variable (""V1", "V2", "V3", "V4", "V5", "G.c v" (C "static" (local) variable), "H.c v" (C "static" (local) variable)). In this figure the triggers on the objects ("main", "F", "F2", "F.c f11", "F11", "H", "V1", "V2",...) represent modification applied on these objects and the colours of the objects represent the level of impact of the modifications applied on these objects. So the graph of this figure shows several types of information: structure of the system (structural dependencies), call-graph, modifications, impact and impact level of these modifications on the other part of the system.

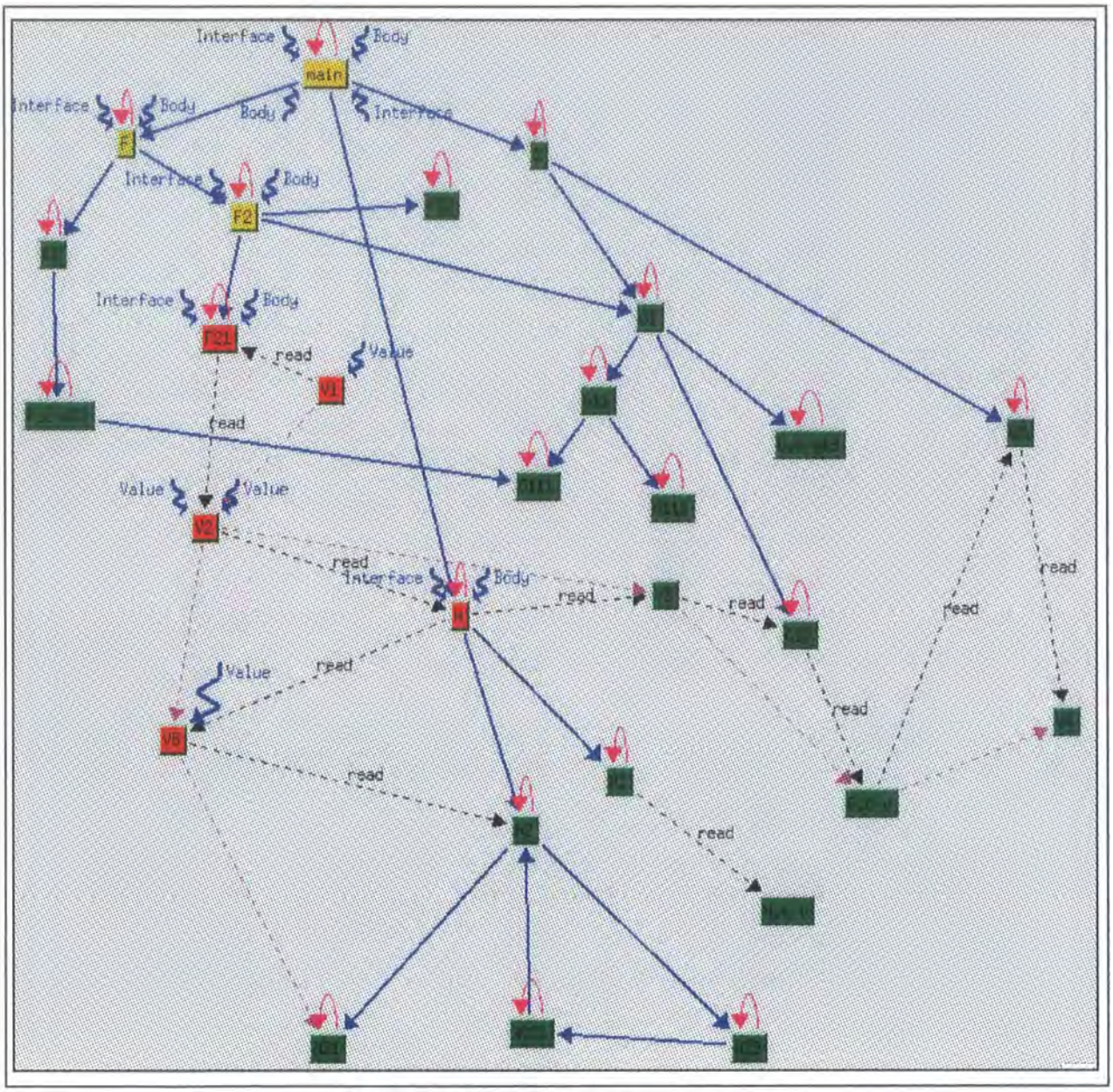


Figure 18 : Example of graph representation

4.3.3. Categorization of Graph Drawing Algorithm

Graph drawing algorithms can be categorized with respect to the following parameters *[TAMASSIA_88]*:

- Class of graphs:

  Each algorithm is usually targeted to a specific class of graphs,i.e., trees, planar graphs, directed or undirected graphs.

- Graphic standard:

  The simplest way to draw a graph in the plane consists of placing first the nodes, and then drawing the arcs as straight-line segments. This is called the straight-line standard (*Figure 16*). Another widely used graphic standard, called grid standard, consists of embedding the graph in a rectangular grid so that the nodes are placed at grid crossing, and the arcs follow the horizontal and vertical tracks of the grid (*Figure 17*).

- Computational Complexity:

  With regard to complexity issues, the running time of the drawing algorithm is critical in interactive applications. However, it can be noted that many natural aesthetic correspond to NP-hard optimization problems (for example CROSS and AREA). This explains why most implemented algorithms are heuristic.

- Aesthetics:

  The term aesthetics, is used to denote the criteria that concern graphic aspects of readability. A well-admitted aesthetic, valid independently from the graphic standard, is the minimization of crossings between arcs. Also, to avoid unnecessary waste of space, it is usual to keep the area occupied by the drawing reasonably small. Drawings that are optimal with respect to a specific aesthetic are generally not optimal with respect to another one. An ideal algorithm should be able to take into account variable weights for the different aesthetics.

- Constraints:

Aesthetics characterize a tidy drawing from the graphical point of view. However, they cannot deal with features that require knowledge about the meaning of the drawing. Semantic features can be expressed by means of constraints on the drawing, which must be explicitly provided to the algorithm as additional input.Examples of constraints are, positioning a group of nodes close to one another, and placing specific nodes on the external boundary of the drawing.

The following taxonomy can be applied to both aesthetics and constraints and is useful to understand their interaction. An aesthetic or constraint may be:

Local/Global: when it refers only to a part of the drawing, global otherwise.

Hierarchic/Flat: hierarchic when it concerns the relative position of a set of symbols, flat otherwise.

Batini [BATINI_84] gives an analysis of the literature in order to determine the aesthetics most commonly adopted in several graph systems. *Figure 19* presents a synthesis of this work where the aesthetics are classified according to the previous paragraph (L: Local, G: Global, H: Hierarchic, F: Flat), *Figure 20* give a similar classification for several types of constraints.

In many documentation applications a sequence of drawings is produced by means of successive updates. An example can be found in top-down methods for software development, where new graphs are created by expanding symbols into more complex structures. It could be expected that two successively generated graph representations to differ only locally, dynamic aesthetics and constraints can be considered that minimize the sum of the "distances" between all consecutive drawings of the sequence, where the distance between two drawings is suitably defined.

Acronym	Aesthetic	Category
AREA	minimization of the area occupied by the drawing	G/F
BALAN	balance of the graph with the respect to the vertical axis or horizontal axis	G/H
BENDS	minimization of the number of bends along the arcs	G/F
CONVEX	maximization of the number of faces drawn as convex polygon	G/F
CROSS	minimization of crossings between arcs	G/F
DEGREE	nodes with high degree in the centre of the drawing	L/F
DIM	minimization of differences among nodes dimensions	G/F
LENGTH	minimization of the global length of arcs	G/F
MAXCON	minimization of the length of the longest arc	G/F
SYMM	symmetry of sons in hierarchies	L/H
UNIDEN	uniform density of nodes in the drawing	G/F
VERT	verticality of hierarchic structures	L/H

Figure 19 : A taxonomy of aesthetics

Acronym	Constraint	Category
CENTER	place a set of given nodes in the centre of the drawing	L/F
DIMENS	assign the dimension of the symbols representing specified nodes	L/F
EXTERN	place specified nodes on the external boundary of the drawing	L/F
NEIGH	place close together a group of nodes	L/H
SHAPE	draw a sub-graph with the specified shape	L/H
STREAM	place a sequence of nodes along a straight line	L/H

Figure 20 : A taxonomy of constraints

In the next sections, the different kind of algorithms will be referred to, according to the taxonomies defined in *Figure 19* and *Figure 20.*

4.3.4. Trees

Trees are extremely common data structures and various algorithms have been proposed for producing automatic drawings of trees *[REINGOLD_81],[MOEN_90],[WETHERELL_79],[BLOESCH_93].*

These algorithms adopt the straight-line standard and the aesthetics CROSS, VERT, SYMM and AREA. Reingold *[REINGOLD_81]* observed a drawback common to most of the algorithms: the drawing of a subtree is influenced by the positioning of nodes outside that subtree, so that a symmetric tree may be drawn asymmetrically. The authors introduced the aesthetic ISO to guarantee that a symmetric tree is drawn symmetrically.

Moen *[MOEN_90]* has considered the drawing of trees with the dynamic aspect and various size of nodes: a tree is not completely redrawn and repositioned if a subtree is inserted or deleted.

4.3.5. General Graphs

The main aesthetics that are usually adopted for these kind of graphs are: SYMM, CROSS, BENDS, LENGTH/MAXCON and UNIDEN. In general the optimization problems associated with these aesthetics are NP-hard *([GAREY_79], [MORET_91]*). Besides time complexity limitations, these aesthetics are also "competitive" in that the optimality of one often prevents the optimality of others. Because of such difficulties, general approaches to graph drawing are usually heuristic and because of the wealth of techniques available for drawing *Planar* graphs the usual strategy is to *planarize* the graph and then apply a planar graph drawing algorithm.

4.3.6. Planar Graphs

The term planarization is used for several related problems. In general, planarization seeks to transform a non-planar graph into a planar graph with a small number of well defined operations (*[OZAWA_80]*).

Clearly, planar drawings are aesthetically highly desirable because they improve the readability of the edges by avoiding the crossings/overlapping of them.

The most common planarization operation is edge deletion: one must find a small number of edges whose deletion yields a planar graph. This is equivalent to find a planar sub-graph with a large number of edges.Finding a planar sub-graph with a maximum number of edges is NP-hard (*[OZAWA_80]*).

Another planarization technique is to find a drawing with the minimum number of crossings. Again, this problem is NP-hard (*[GAREY_79]*).

Most planar graph drawing methods proceed as follows:

Step 1: Test planarity (*[BATTISTA_88]*).

Step 2: (If the graph is planar), Construct a *Planar representation.*

Step 3: Use the *Planar representation* to draw the graph according to some graphic standards.

4.3.7. Directed Graphs

Directed graphs (*Digraph*) are a very important class of graphs in which all the edges between nodes are directed (i.e. a source and a destination of the edge). A *Digraph* G consists of a set of vertices $V = \{v_1, v_2, ...\}$, a set of edges $E = \{e_1, e_2, ...\}$ and a mapping Ψ that maps every edge onto some ordered pair of vertices (v_i, v_j). As in the case of undirected graphs, a vertex is represented by a point and an edge by a line segment between v_i and v_j with an arrow directed from v_i to v_j. For instance *Figure 21* shows a digraph with five vertices and ten edges. A digraph is also referred to as an oriented graph.

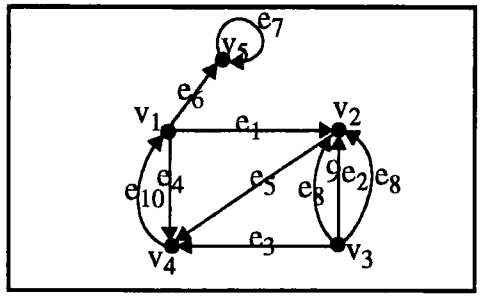


Figure 21 : Directed graph with 5 vertices and 10 edges

The acyclic *Digraph* is widely used to display hierarchical structures. Examples include PERT diagrams and various dependency graphs. It is customary to represent these graphs so that the edges all flow in the same direction, i.e., from top to bottom, or from left to right.

A great deal of work has been done in drawing algorithms to produce drawing of directed graphs because of the needs to represent hierarchies or dependencies.

In *[SUGIYAMA_81],[CARPANO_80],[GANSNER_88], [REGGIANI_88]* and *[BATTISTA_88]* several algorithms for directed graph drawings are presented.

4.4. Graph Algorithm Implementation and Visualization

In this research, a tool has been developed to display/edit a graph (and interact with it) with an automatic placement of the graph (nodes) using the environment system X/Windows and Motif on Unix platform.

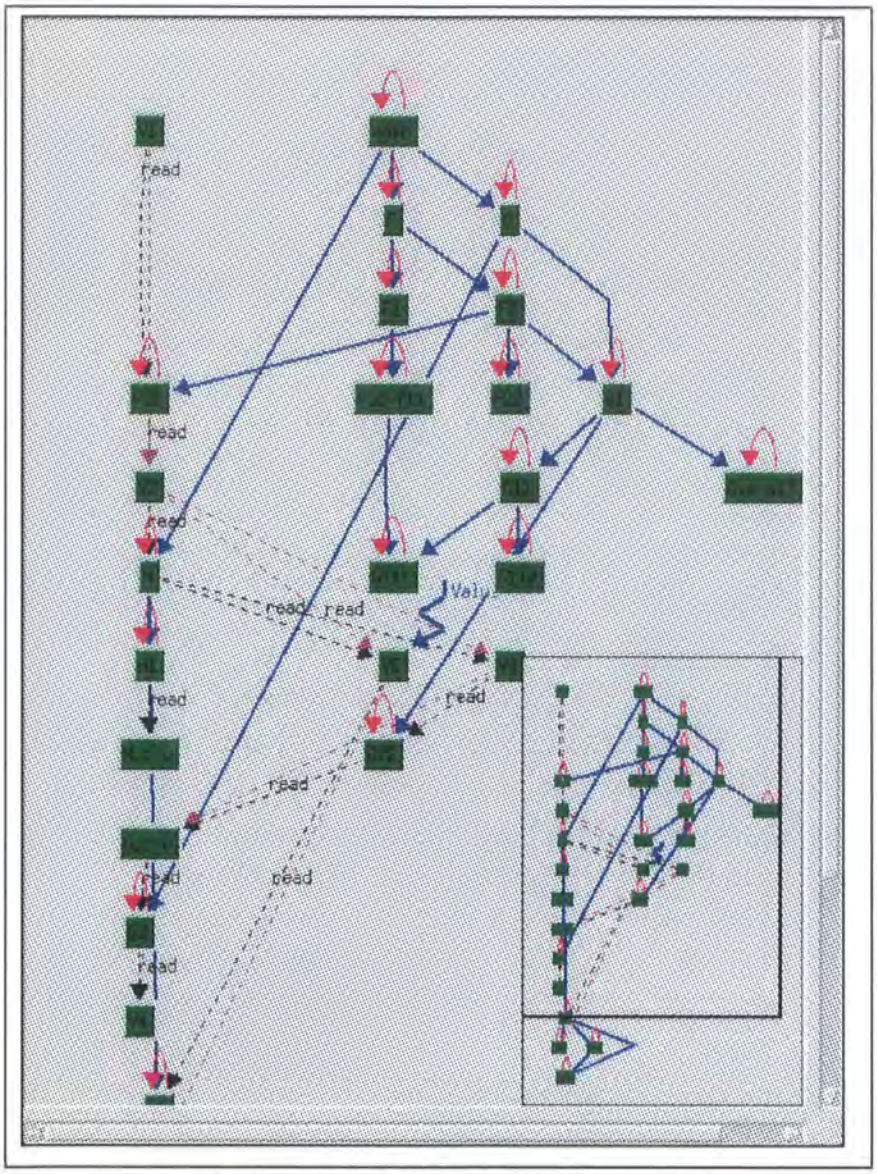


Figure 22 : A graph displayed with the implemented widget

The graph tool implemented is based on the "Motif -- OSF X User Environment Widget Set" (Copyright 1989 by Hewlett-Packard Company). It was debugged and enhanced in many ways (Automatic positioning for nodes and links, triggers on nodes, zoom/reduction,...).

The result of this work is a set of functionalities (X/Windows/Motif widget) which allows a user to display/edit a graph and interact with it in term of actions on the graph that the user can do. The *Figure 22* shows a graph displayed with the graph widget implemented and is planned to be used in the Impact Analysis System.

The basic requirements for the graph display tool are:

- To visualize vertices and edges as nodes and arcs on a two dimensional area with a representation in accordance with the connections between the vertices and with some pre-defined aesthetics (like reducing edge crossing, etc.). The nodes and arcs must have some graphical attributes (colour, text, shape, line style, etc.) which can be set to visualize an internal state of the vertices/edges,
- To be able to edit the graph: moving a node or an arc, adding or deleting a node/arc,
- To allow the visualization of triggers on nodes (in order to visualize for impact analysis the modifications applied to a component (node)),
- To have immediate feed-back on the characteristics of nodes or arcs on which the mouse pointer is: to display for instance the state of the object we are on.
- To offer a zoomed/reduced view of the graph to facilitate the understanding of a big graph which can be displayed on a single screen,
- To allow an automatic positioning of nodes and arcs in the graph and possibly manually move the nodes/arcs,
- To be able to interact with the graph: clicking on a node or arc to visualize the semantic of it (opening a document for instance by clicking on the representation of the document (icon)),

4.4.1. Visualization and Editing of a graph

The graph tool has been implemented using the C programming language and with the X11, Xt and Xm (Motif) libraries. It can be used in any Unix environment with X-Windows server. The nodes in the graph can be any widget or gadget of any X-Windows ToolKit (Xt, Motif, Athena,...).

The graph tool can be controlled directly with the mouse to move or edit nodes and arcs. Every attributes of the graph, nodes and arcs (text, colour, position,...) can be controlled with Application Programmer Interface (API) delivered with the graph tool (Set of functions).

The triggers on nodes have been implemented to allow representation of modifications applied on objects (nodes). A trigger is represented as broken lines ending on the node and it can have a text attached to it.

The functionalities for providing support about the mouse position (on nodes, arcs, zoom or graph) in the graph have been implemented and used for testing purposes. So in the "gtest" program developed, if a user moves the mouse pointer on to a node, it will change to represent a "O" (Object) or on to an arc it will represent an arrow. This can be used to define the mouse pointer representation according to the type of nodes or arcs on which the mouse pointer is located (function, variable, document,..., call links, inheritance link,...).

A zoom or reduction view of the graph has been implemented in order to facilitate the visualization of graph and to help the user to know where in the graph the visible portion of it is. This zoom/reduction view is useful in order to have a synthesised view of the complete graph and of its shape.

4.4.2. Algorithm for automatic positioning for nodes and arcs.

The algorithm used to position the nodes and the arcs in the graph is the one described by Sugiyama [SUGIYAMA_81]. It allows the drawing of any kind of graph because all the graphs are described as directed but all the arcs can be "directed", "not directed", "reverse directed" or "bidirected" according to the description of the directed graph.

Basically the algorithm used in the graph widget for the automatic placement of nodes is:

- Step 1:

  A "proper" hierarchy is formed from a graph. If the digraph generated has cycles, it is transformed (by changing the direction of some arcs) and we obtain a multi-level digraph or a hierarchy (Each node has now a level in the graph). Then, if the hierarchy has long span edges (difference of level between two connected nodes superior to 1), it is converted into a proper hierarchy by adding dummy vertices and edges.

- Step 2:

  The number of crossings of edges in the proper hierarchy is reduced by permuting orders of vertices in each level ().

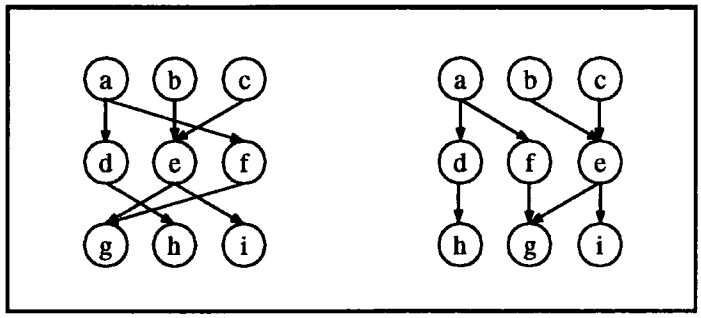


Figure 23 : Reducing crossings of edges.

- Step 3:

  Horizontal positions of vertices are determined by considering:
  Close" layout of vertices connected to each other (weighted by the number of connections between them). It is desirable that paths are short.

- Step 4 (Possible but not implemented):

  A two-dimensional picture of the hierarchy is automatically drawn where the dummy vertices and edges are deleted and the corresponding long span edges are regenerated.

4.5. Interacting with the graph

The graph tool developed is not to be just used to represent graphs, it must also be used interactively. Each node and arc in the graph has to be reached with a mouse click. For instance, a mouse click on a node allows a user to open a visualizer/editor for the type of node on which the notification has been done. Callbacks on each node and arc can be attached to actions carried out on them.

4.6. Summary

In this chapter, basic terminology and concepts required for the graph representation and visualization have been treated. A tool (Widget) has been elaborated in order to visualize the information which has to be represented for *Impact Analysis* purpose. The next chapter will present the system which have been developed to do the whole process of *Impact Analysis*.

5. Impact Analysis System

5.1. Introduction

The Impact Analysis System constructed is an environment to study the effects of a change on a system, that is impact analysis. For this purpose, a *Dependency model* and *Propagation model* is required together with the dependencies corresponding to a model of a software system. These dependencies will be used to understand the system by browsing through the dependencies, to design the modifications we want to apply on the system and to apply the Impact Analysis on them. Several steps will have to be done (Models generation, Dependencies generation, Modifications modelling, Propagation) to perform the task of Impact Analysis.

The Impact Analysis System described in this chapter was developed as part of the ESF/EPSOM project and has been continued in the ESPRIT/AMES project. The work for this thesis is mainly concerned with viewing dependency graphs.

5.2. Architecture

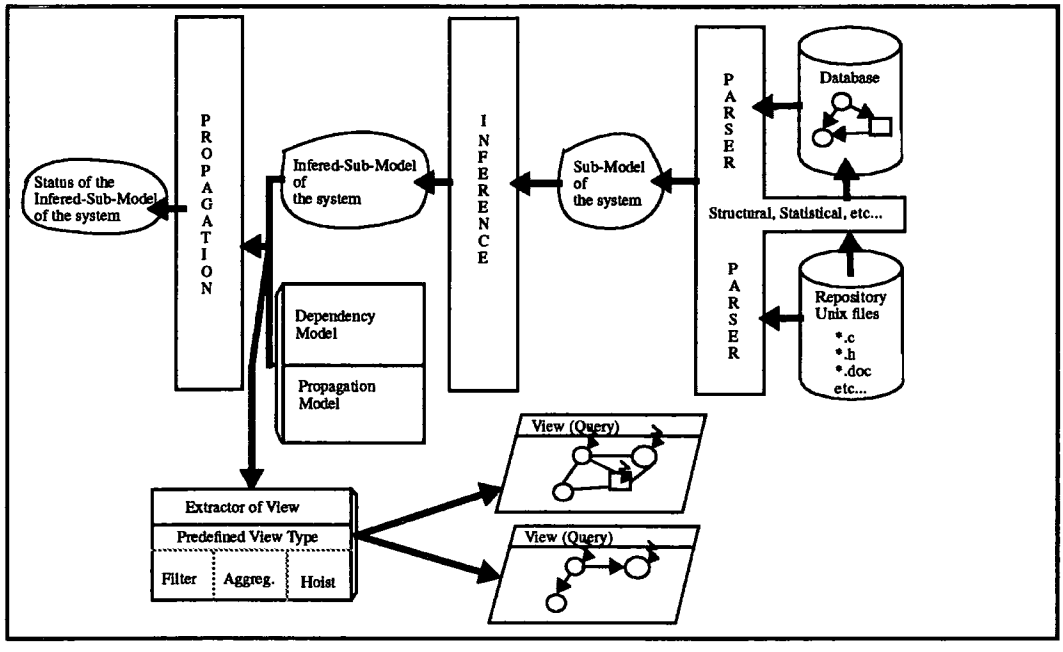


Figure 24 : Architecture impact Analysis System

This architecture of the system corresponds in fact to the process involved to create the model of the system, to visualize the system and its status by views and to apply modifications and *Impact Analysis* in order to understand the effects of these modifications on the other parts of the system.

The first step of *Impact Analysis* (PARSER) is to extract the information contained in the system in order to work on it in a more abstract way. For this purpose several tools have been developed, for example: parser of source code, parser of documentation. The purpose of these tools is to extract all the information of a software system. A C-code parser has been reused (*[BUNTER_93]*) to construct the C-code representation in Prolog facts. A tool (*[HTML_ANALYZER_94]*) to extract links in HTML (HyperText Mark-up Language) has been reused and modified for generating graph representation.

The second step of *Impact Analysis* (INFERENCE) is to infer a model of the system at a certain level of granularity of information required for the visualization/propagation. This step will for example create a model of source code in terms of function and variable and the dependencies between them instead of files. I have written several Translators in Prolog to extract dependencies from the output of the C-code parser.

The third step of *Impact Analysis* is (PROPAGATION) is (after having created a model of propagation for the model of dependencies generated by the second step) to apply the modifications on the model of the system and to propagate them to obtain all the impacts provoked by the initial set of modifications that the maintainers wanted to apply on the system. The propagation engine is currently in development in Matra Marconi Space (ESF/EPSOM and ESPRIT/AMES projects). I have according to the propagation model outlined in *Description of the Dependency Model page 66,* developed, with the graph visualization tool, for demonstration purpose a hard-coded propagation with visual effects on the "*Hardware integer division algorithm page 36*".

The visualization in the last step (VISUALIZATION) of *Impact Analysis* is an important part because it is required to understand the system on which the maintainers have to work, to model the modifications that they want to apply on the system and to obtain/visualize the effects on these modifications on the rest of the system. The graph tool that I have partly reused (The base of this tool comes from "Motif -- OSF X User Environment Widget Set, Copyright

1989 by Hewlett-Packard Company") and largely debugged and enhanced in many ways (Automatic positioning for nodes and links, triggers on nodes, zoom/reduction,...).

5.3. Models

A *Dependency model* is composed of types of objects and links that can allow a representation of the system to be created. Dependencies of a software system are an instantiation of a dependency model. Dependencies are used to represent a particular system at a certain level of granularity. For instance a *Dependency model* represented with a graph could be the one in *Figure 25*, the same with a text representation is shown in *Figure 26*.

There are three object types and eleven link types between these object types. For instance an object of type "module" may contain (link type "contains" between the object type "module" and the object type "function") another object of type "function", an object of type "function" may call another object of type "function" (link type "calls" between object type "function" and itself).

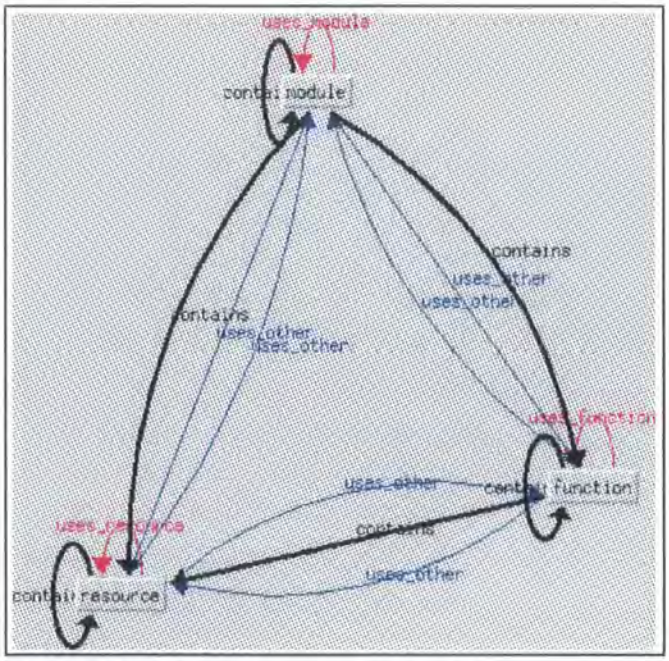


Figure 25 : A Dependency model (Graph Representation)

```
object_type(module).
object_type(function).
object_type(resource).
link_type(uses_module).
link_type(uses_function).
link_type(uses_resource).
link_type(uses_other).
link_type(contains).
valid_link(module, contains, module).
valid_link(module, contains, function).
valid_link(module, contains, resource).
valid_link(module, uses_module, module).
valid_link(module, uses_other, function).
valid_link(module, uses_other, resource).
valid_link(function, contains, function).
valid_link(function, contains, resource).
valid_link(function, uses_function, function).
valid_link(function, uses_other, module).
valid_link(function, uses_other, resource).
valid_link(resource, contains, resource).
valid_link(resource, uses_resource, resource).
valid_link(resource, uses_other, module).
valid_link(resource, uses_other, function).
```

Figure 26 : A Dependency model (Textual Representation)

A *Propagation model* is applied to a *Dependency model* by consideration of the types of modifications to be made and the Propagation rules that are applicable to the types of objects and links of the *Dependency model*. For instance, *Figure 27* shows a *Propagation model* based on the *Dependency model* of the *Figure 25* (*Figure 28* is the textual representation). The modification types applicable on the different object type are shown with the impact triggers in the graph (*Figure 27*). However the type of propagation of these modification type (*Propagation rule*) is not shown on this graph and requires another graph to visualize it. An example of a propagation rule is a modification type "public_change" on the object type "function" which is propagated through the link type "calls" (between two objects of type "function") to all the objects of type "function" with the modification type "private_change" ("automatic" which means that the modification is sure and not "potential") applied on them.

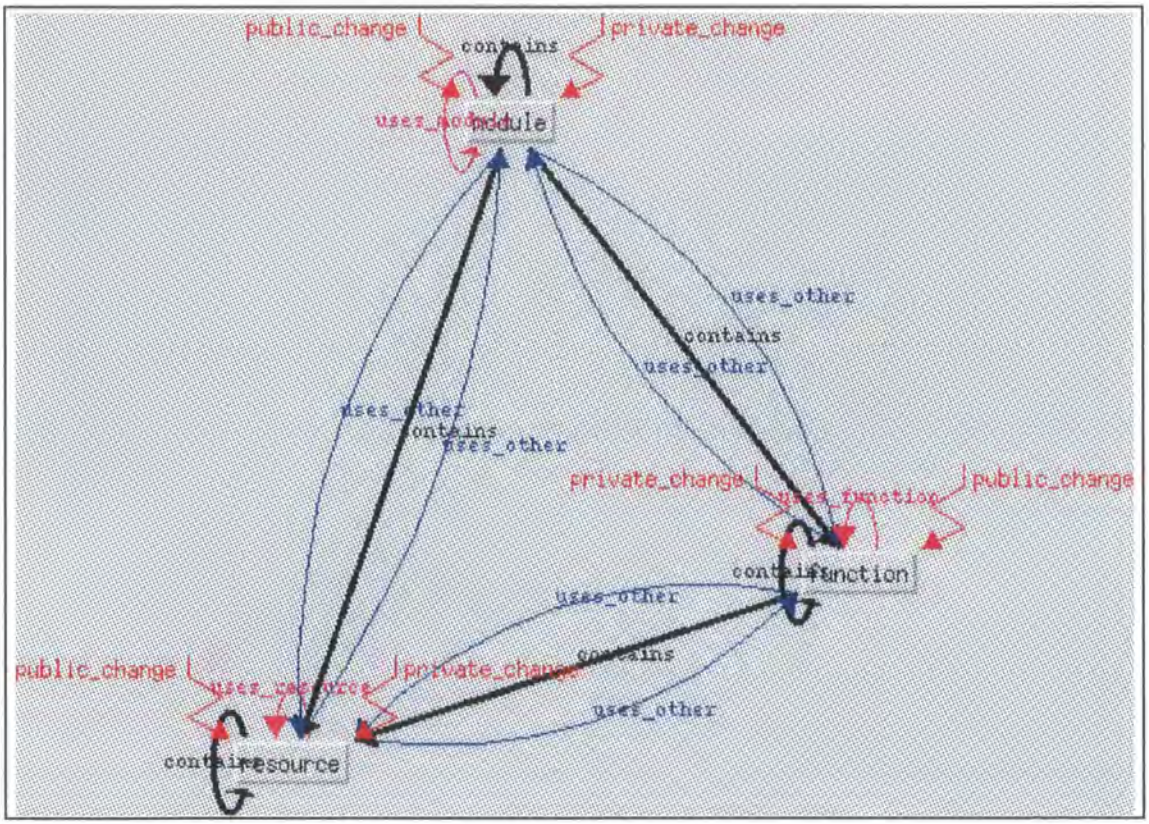


Figure 27 : A Propagation model (Graph Representation)

```
modif_type(public_change).
modif_type(private_change).
valid_modif(module, public_change).
valid_modif(module, private_change).
valid_modif(function, public_change).
valid_modif(function, private_change).
valid_modif(resource, public_change).
valid_modif(resource, private_change).

/* impact(function,public-change,*-is_used-by,*) -> private-change (automatic) */
propagation_rule(module,public_change,uses-module,module, private_change,yes).
propagation_rule(module,public_change,uses-other,function, private_change,yes).
propagation_rule(module,public_change,uses_other,resource, private_change,yes).

/* impact(function,public-change, *-is_used-by, *) -> private-change (automatic) */
propagation_rule(function,public_change,uses_function,function, private_change,yes).
propagation_rule(function,public_change,uses_other,module, private_change,yes).
propagation_rule(function,public_change,uses_other,resource, private_change,yes).

/* impact(resource,public-change, *-is_used-by, *) -> private-change (automatic) */
propagation_rule(resource,public_change,uses_resource,resource, private_change,yes).
propagation_rule(resource,public_change,uses_other,module, private_change,yes).
propagation_rule(resource,public_change,uses_other,function, private_change,yes).

/* impact(*,private-change, self, *) -> public-change (potential) */
propagation_rule(module, private_change, reflexive, module, public_change, no).
propagation_rule(function, private_change, reflexive, function, public_change, no).
propagation_rule(resource, private_change, reflexive, resource, public_change, no).

/* impact(*,public-change, self, *) -> private-change (automatic) */
propagation_rule(module, public_change, reflexive, module, private_change, yes).
propagation_rule(function, public_change, reflexive, function, private_change, yes).
propagation_rule(resource, public_change, reflexive, resource, private_change, yes).

/* impact(*,private-change, contains, *) -> public-change (potential) */
propagation_rule(module, private_change, is_contained_in, module, public_change, no).
propagation_rule(module, private_change, is_contained_in, function, public_change, no).
propagation_rule(module, private_change, is_contained_in, resource, public_change, no).
propagation_rule(function, private_change, is_contained_in, function, public_change, no).
propagation_rule(function, private_change, is_contained_in, resource, public_change, no).
propagation_rule(resource, private_change, is_contained_in, resource, public_change, no).

/* impact(*,public-change, is-contained-in, *) -> private-change (automatic) */
propagation_rule(module, public_change, contains, module, private_change, yes).
propagation_rule(function, public_change, contains, module, private_change, yes).
propagation_rule(resource, public_change, contains, module, private_change, yes).
propagation_rule(function, public_change, contains, function, private_change, yes).
propagation_rule(resource, public_change, contains, function, private_change, yes).
propagation_rule(resource, public_change, contains, resource, private_change, yes).

/* create_object => impact(*,created, self, *) */
/* Defined automatically in the engine : creation */
/* impact(*,created, self, *) -> public-change (automatic) */
propagation_rule(module, creation, reflexive, module, public_change, yes).
propagation_rule(function, creation, reflexive, function, public_change, yes).
propagation_rule(resource, creation, reflexive, resource, public_change, yes).

/* create(uses-*,*,*) -> (private-change,none) (automatique) */
/* Link creation not yet defined !!!!! */

/* create(contains,*,*) -> (private-change,none) (automatique) */
/* Link creation not yet defined !!!!! */

/* impact(*,deleted, contains, *) -> father-deleted (automatic) */
propagation_rule(module, suppression, is_contained_in, module, father_deleted, yes).
propagation_rule(module, suppression, is_contained_in, function, father_deleted, yes).
propagation_rule(module, suppression, is_contained_in, resource, father_deleted, yes).
propagation_rule(function, suppression, is_contained_in, function, father_deleted, yes).
propagation_rule(function, suppression, is_contained_in, resource, father_deleted, yes).
propagation_rule(resource, suppression, is_contained_in, resource, father_deleted, yes).
```

Figure 28 : A Propagation model (Textual Representation)

5.3.1. Description of the Dependency Model

object_type(type).

It declares a type of object.

```
For instance: object_type(function).
```

link_type(type).

It declares a type of link.

```
For instance: link_type(calls).
```

valid_link(obj1_type, lk_type, obj2_type).

It declares a valid link of type "lk_type", which must been defined with:

```
link_type("lk_type").
```

between two objects, the first object (source of the link) of type "obj1_type" and the second (destination of the link) of type "obj2_type".

```
For instance: valid_link(function, calls, function).
```

5.3.2. Description of the Propagation Model

modif_type(type).

It declares a type of modification.

```
For instance: modif_type(public_change).
```

valid_modif(obj_type,mod_type).

It declares a valid modification of type "mod_type" on object of type "obj_type".

```
For instance: valid_modif(function, public_change).
```

propagation_rule(trigger_obj_type, trigger_mod_type, lk_type, imp_obj_type, imp_mod_type, auto).

It declares a propagation rule for which a modification of type "trigger_mod_type" on an object of type "trigger_obj_type" linked to another object of type "imp_obj_type" by a link of type "lk_type" will be propagated on the second object (impacted) by applying a modification of type "imp_mod_type" on it. The potentiality level of the modification on the impacted object will depend on the potentiality level of the trigger modification and on the automaticity "auto" (Normally if auto==yes then the same potentiality level for both else the second will be higher of one).

```
For instance: propagation_rule(function, public_change,
              calls, function, private_change, yes).
```

5.4. Dependencies Generation

5.4.1. Parser

```
edge(0, 'division', 0, 1).
edge(0, 'division', 1, 2).
edge(0, 'division', 10, 11).
edge(0, 'division', 11, 6).
edge(0, 'division', 2, 3).
edge(0, 'division', 3, 4).
edge(0, 'division', 4, 5).
edge(0, 'division', 4, 6).
edge(0, 'division', 5, 4).
edge(0, 'division', 6, 7).
edge(0, 'division', 6, end).
edge(0, 'division', 7, 8).
edge(0, 'division', 8, 9).
edge(0, 'division', 9, 10).
edge(0, 'division', 9, 6).
expression(0, 'division', 1, expr(assign, expr(def, 'r', ['@pointer']), expr(ref, 'x', []))).
expression(0, 'division', 10, expr(assign, expr(def, 'r', ['@pointer']), expr(connect, expr(ref,
'r', ['@pointer']), expr(ref, 'w', [])))).
expression(0, 'division', 11, expr(assign, expr(def, 'q', ['@pointer']), expr(ref, 'q',
['@pointer'])))).
expression(0, 'division', 2, expr(def, 'q', ['@pointer'])).
expression(0, 'division', 3, expr(assign, expr(def, 'w', []), expr(ref, 'y', []))).
expression(0, 'division', 4, expr(connect, expr(ref, 'w', []), expr(ref, 'x', []))).
expression(0, 'division', 5, expr(def_and_ref, 'w', [])).
expression(0, 'division', 6, expr(connect, expr(ref, 'w', []), expr(ref, 'y', []))).
expression(0, 'division', 7, expr(assign, expr(def, 'q', ['@pointer']), expr(ref, 'q',
['@pointer'])))).
expression(0, 'division', 8, expr(assign, expr(def, 'w', []), expr(ref, 'w', []))).
expression(0, 'division', 9, expr(connect, expr(ref, 'w', []), expr(ref, 'r', ['@pointer'])))).
file('Hardware.c', 0).
object(0, '@external', sc([], '@'), void, 'division', ['@fun']).
object(0, 'division', sc([1], '@'), int, 'w', []).
object(0, 'division', sc([], '@'), int, 'q', ['@pointer']).
object(0, 'division', sc([], '@'), int, 'r', ['@pointer']).
object(0, 'division', sc([], '@'), int, 'x', []).
object(0, 'division', sc([], '@'), int, 'y', []).
parameter(0, 'division', ['x', 'y', 'q', 'r']).
statement(0, 'division', [1,1], 5, [expr]).
statement(0, 'division', [1,2,1], 10, [expr]).
statement(0, 'division', [1,2,1], 11, [expr]).
statement(0, 'division', [1,2], 7, [expr]).
statement(0, 'division', [1,2], 8, [expr]).
statement(0, 'division', [1,2], 9, [sele, if]).
statement(0, 'division', [1], 1, [expr]).
statement(0, 'division', [1], 2, [expr]).
statement(0, 'division', [1], 3, [expr]).
statement(0, 'division', [1], 4, [iter, while]).
statement(0, 'division', [1], 6, [iter, while]).
```

Figure 29 : Output of the C-parser for the program

"Hardware integer division algorithm" (*Figure 10*)

Figure 29 shows the output of the C-parser for the Hardware integer division algorithm, the representation used could also have been used for programming languages other than C. The semantic of the output of the C source code parser (Perplex) is explained in *[BUNTER_93]*. The purpose of the parser is to create an output of information about a system or a part of it (i.e. source code, documentation, design,...). Normally the parser extracts all the information that it can from the input and another tool has to select from this source the information required.

5.4.2. Inference

The dependencies are represented by objects and links. As Prolog is used to propagate the modifications through the dependencies, "Prolog facts" are used to represent these objects and links. An identifier is given to each object of the dependencies and information is added to each object with another "Prolog fact": id_object which will collect all the information not needed by the propagation engine. For instance, *Figure 30* shows the dependencies generated from the program of the Hardware integer division algorithm and after inference from *Output of the C-parser for the program "Hardware integer division algorithm" (Figure 10).*

```
id_object([0,0],['HardwareDivision.c',none,none,none,none]).
id_object([1,0],['@external',sc([], @),void,division,['@fun']]).
id_object([2,0],[division,sc([1], @),int,w,[]]).
id_object([3,0],[division,sc([], @),int,q,['@pointer']]).
id_object([4,0],[division,sc([], @),int,r,['@pointer']]).
id_object([5,0],[division,sc([], @),int,x,[]]).
id_object([6,0],[division,sc([], @),int,y,[]]).
link([0,0],contains,[1,0]).
link([1,0],contains,[2,0]).
link([1,0],contains,[3,0]).
link([1,0],contains,[4,0]).
link([1,0],contains,[5,0]).
link([1,0],contains,[6,0]).
link([2,0],uses(resource, resource),[2,0]).
link([2,0],uses(resource, resource),[6,0]).
link([3,0],uses(resource, resource),[3,0]).
link([4,0],uses(resource, resource),[2,0]).
link([4,0],uses(resource, resource),[4,0]).
link([4,0],uses(resource, resource),[5,0]).
object([0,0],module).
object([1,0],function).
object([2,0],resource).
object([3,0],resource).
object([4,0],resource).
object([5,0],resource).
object([6,0],resource).
```

Figure 30 : Dependencies for the C-code of *Figure 10*

(Hardware integer division algorithm)

5.4.3. Description of the Dependencies

`object(obj_id, obj_type).`

It declares an object of identifier "obj_id" (which must be unique in the dependencies) and of type "obj_type".

`For instance: object([1,0], function).`

`link(obj_source_id, lk_type, obj_dest_id).`

It declares a link of type "lk_type" from the object of identifier "obj_source_id" to the object of identifier "obj_dest_id".
For instance: link([1,0], calls, [1,0]).

5.5. Propagation Engine

The propagation engine is used to propagate modifications along dependency links. It expects as input a dependency model, propagation model, a model of the system based on the dependency model and a set of modifications that the user (maintainer) will want to apply on the system. The propagation engine uses this information to do the propagation and gives the set of impacted objects with information about the types of impacts on the system that it has computed.

The propagation engine is currently written in Prolog as this language is particularly suitable for prototyping. The propagation can be controlled step by step in order to understand the effects of a particular modification. The user (maintainer) may create modification on the system and obtain the effects of this modification on the rest of the system

The propagation engine is able to deal with modification of the model of the system (modification of the structure of the system) during the propagation. For instance in order to "create" a call between two objects "function" as the modification of the system.

The approach that has been taken in the implementation (*[BARROS_94]*) allows the interactive application of typed modifications to a graph of objects and links, and to propagate these modifications through the graph, following previously defined propagation rules, in order to exhibit impacts of the proposed modifications. These impacts are themselves modifications, which can then be recursively propagated in order to obtain the complete set of impacts of the proposed modifications.The *Figure 31* shows how the modifications are propagated through links with the control a a propagation rule (The "Typed modification#1" on an object of type "Typed object#1" is propagated through a link of type "Typed link#1" in "Typed modification#2" on objects of type "Typed object#2").

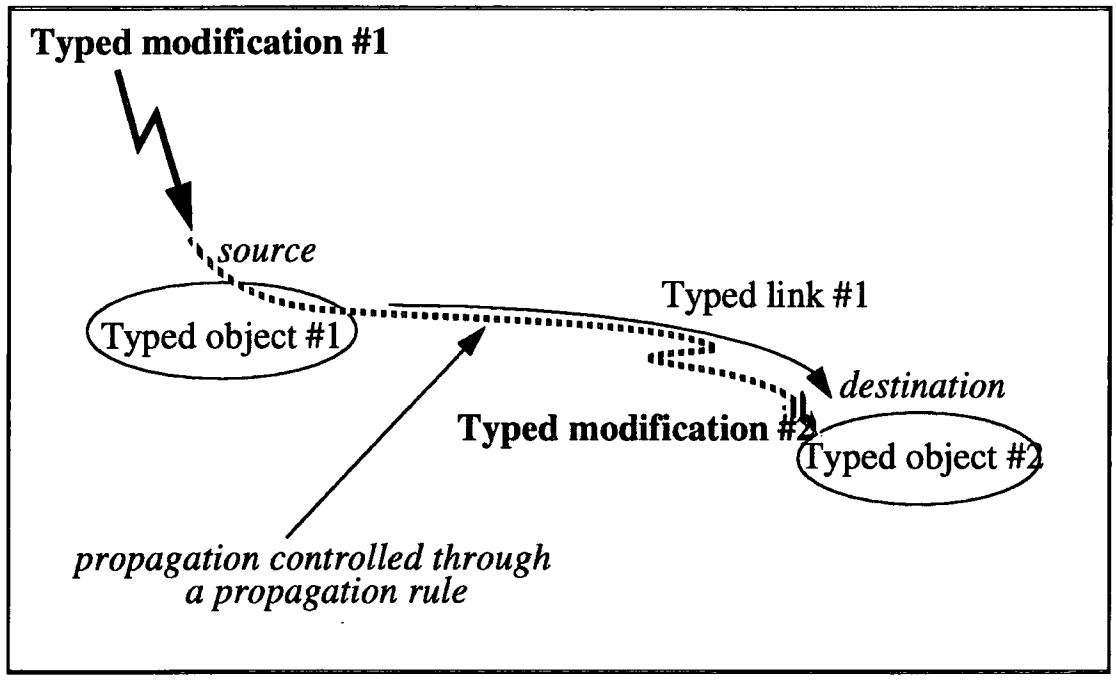


Figure 31 : How the propagation rule works?

The propagation engine has a very simple iterative algorithm displayed in *Figure 32*.

As presented in *Figure 32*, the propagation process can be resumed in two main loops:

- the first one, which runs over all the new (and so not yet propagated) modifications found by the engine at one step, aims at propagating them,
- the other loop runs over all the steps needed to achieve the propagation.

While the main loop over the propagation step is not finished, the engine is in an unstable state and cannot be stopped. But the steps are recorded, to allow the user to display later on the way the propagation was performed from modification to modification, at each step.

Within the one step propagation loop, all the modifications defined at the previous step are propagated, using the process displayed in *Figure 33*.

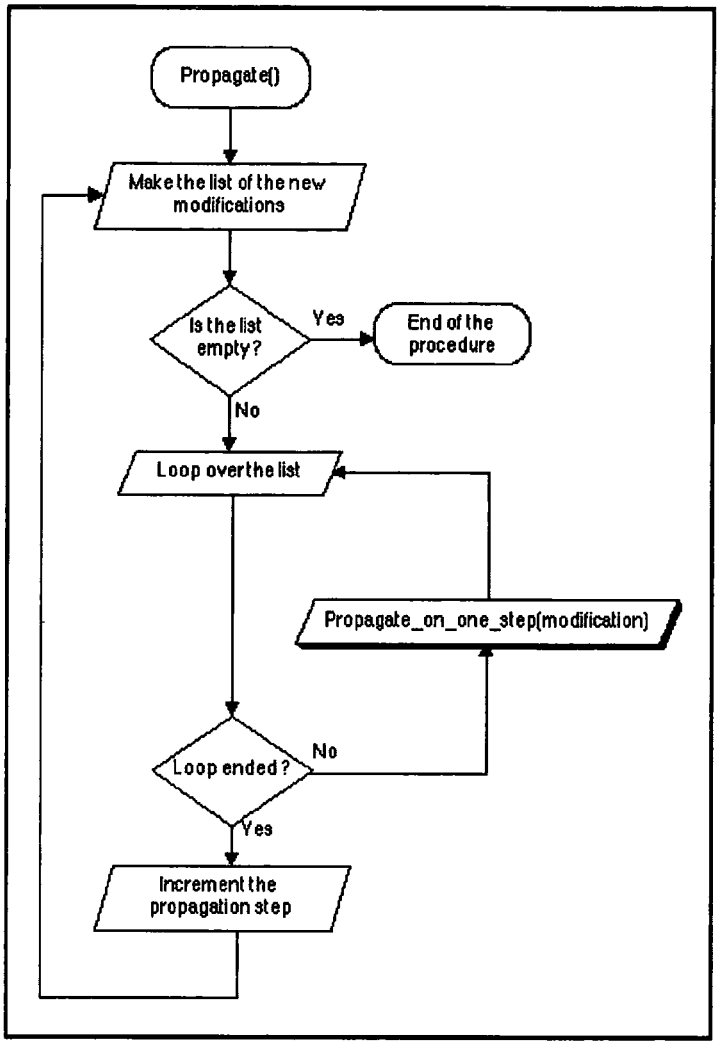


Figure 32 : Propagation algorithm, level 1.

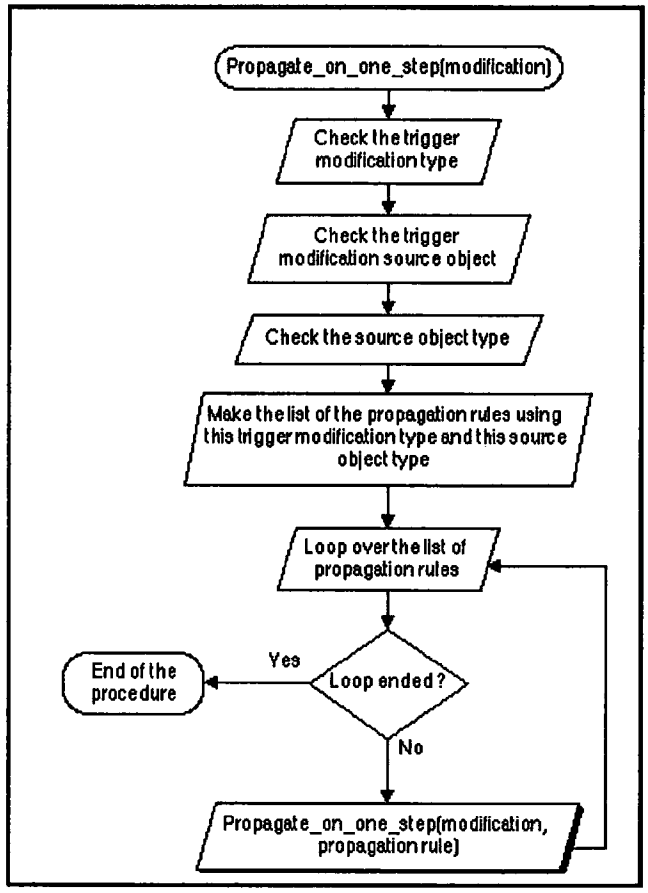


Figure 33 : Propagation algorithm, level 2.

To perform the propagation of a modification, as defined above by a couple [object identifier, modification type], the engine first gets the list of propagation rules modelling the propagation of this modification type on the type of the object.

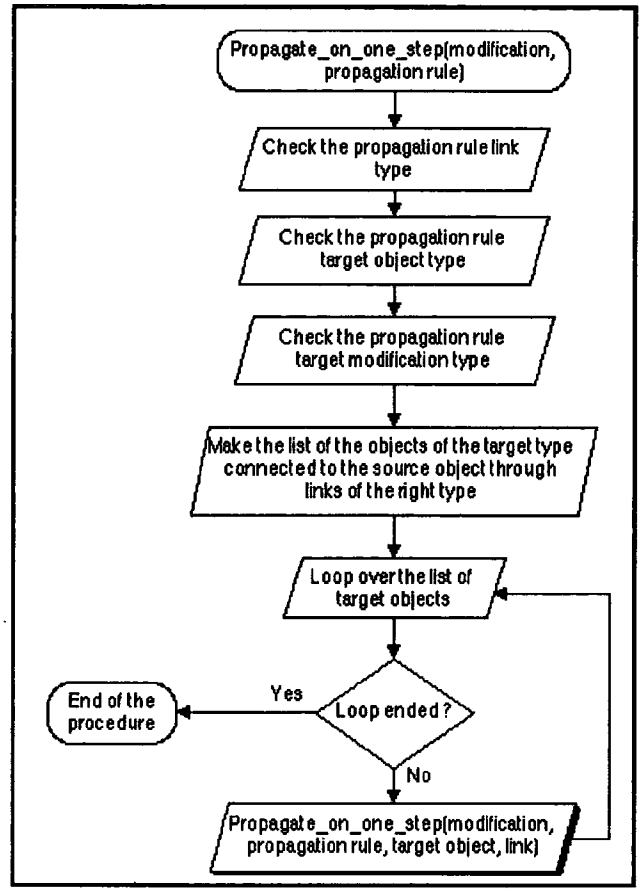


Figure 34 : Propagation algorithm, level 3.

Then, the process loops over all the selected propagation rules to check the impacts of the trigger modification on the neighbourhood of the modified object, as described in *Figure 34*.

Within this procedure, the propagation engine selects all the objects connected to the modified one with the right link type (given by the propagation rule) and impacts them.

During this, the new modifications are defined and the impacts from the trigger modification to the new target ones are stored, to be later displayed to the user if he wants to understand why and how a side effect modification occurs, and from which initial modification it depends. This mechanism is displayed on *Figure 35*.

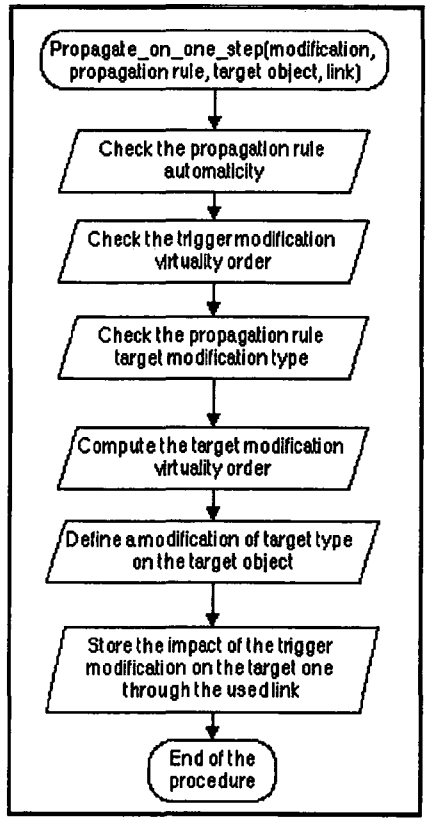


Figure 35 : Definition of a new modification

After the definition of the new modification (*Figure 35*), all the previous loops are executed until no more impacts appear.

5.6. Interface

A fundamental part of the interface has been completed, "the graph Widget" which will graphically show views on the dependencies of a system as well as the state of the propagation on it (colours, triggers (graphical representations of the modifications on objects), etc.).

The interface for Impact Analysis has to display several types of information concerning the system and to allow the user to select the types of information that he needs. The system is represented as a graph which corresponds to the dependencies between the different parts of the system. The parts of the system (Objects) depend (or are linked) to other parts by Links. The objects may have different attributes to qualify the information they contain or their status, also the user must have the possibility to select the information on the

node that he wants to have. Each object must have a pop-up menu attached to it to call a specific method or general method of the object (For instance: open, add a modification on the object,...). The object may have attached to them modifications represented as triggers for their graphical representation. The status of these modifications may be visualized by colours, shapes or labels on the triggers.

The system usually cannot be visualized completely at the beginning as it could be enormous, so the interface has to allow the user to select which part of the system has to be visualized or not (by query or just by non-aggregation of parts of the graph dependencies).

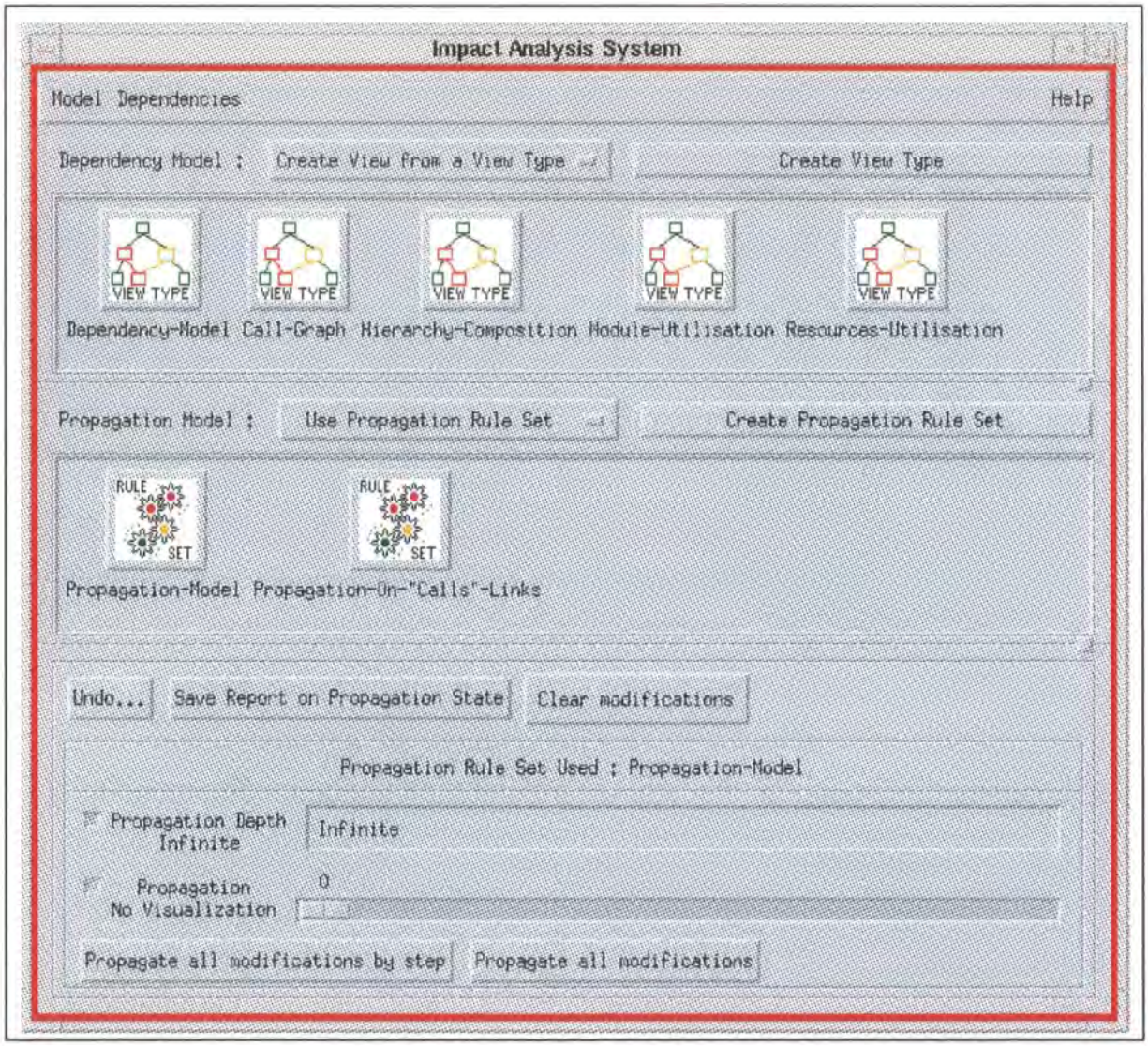


Figure 36 : Main Window of the IAS First version

The *Figure 36* shows the main window of the IAS (Impact Analysis System). This window shows the different parts of the impact analysis: The management of dependency models (The *Figure 37* shows the editing window for the dependency model) and propagation models (first and second horizontal parts of this window) and the management of the propagation itself in the third part of the window.

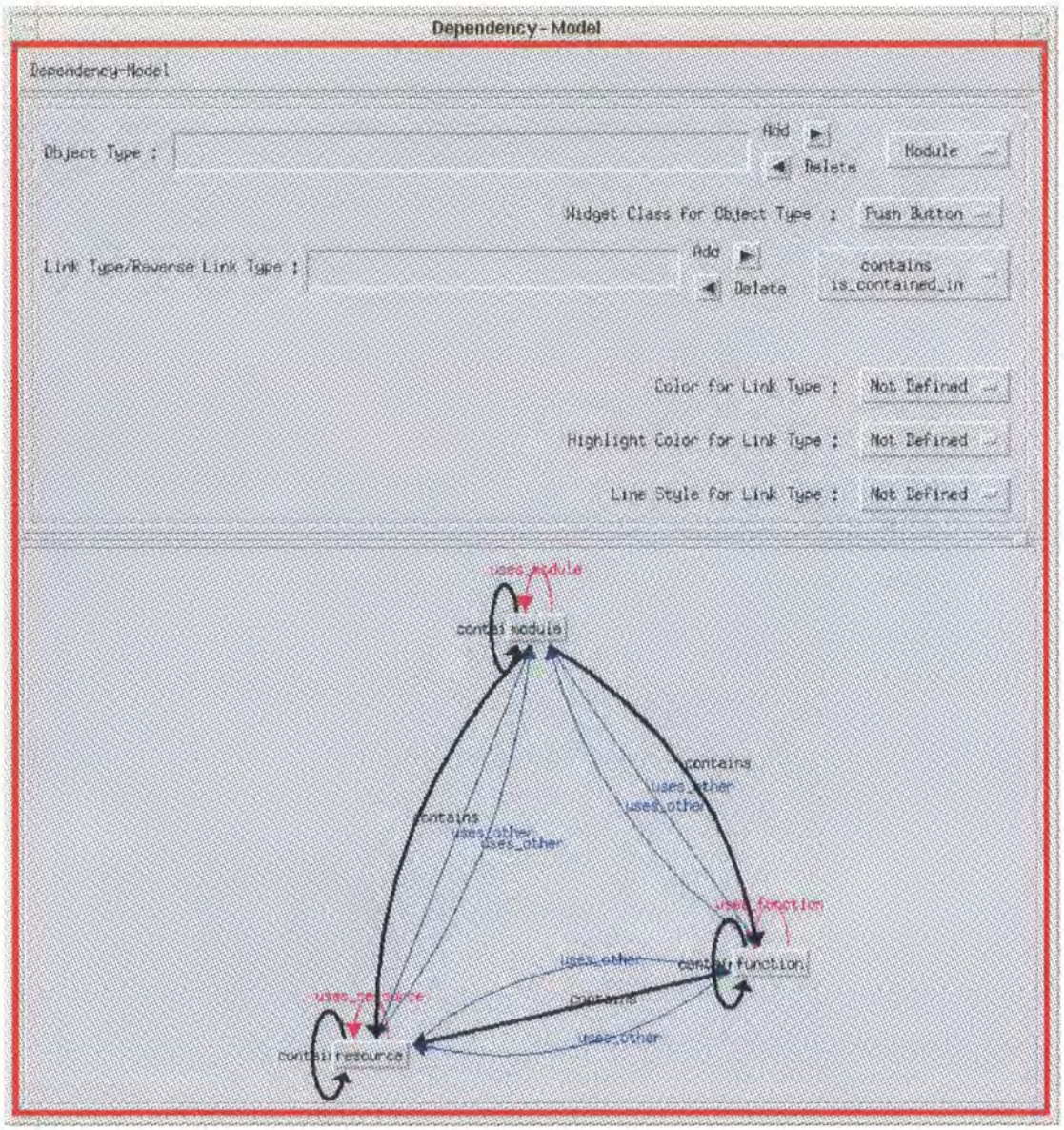


Figure 37 : The edition window for the Dependency model of the IAS.

5.7. Summary

The Impact Analysis System has been partly implemented (Dependencies Generation, Propagation Engine, Widget graph), the remaining tasks (Interface for Impact Analysis (Navigation & Display as well as editing the dependencies and/or the modifications on the dependencies), Communication between Propagation Engine and Interface) will have to be performed to obtain a complete tool for Impact Analysis. However each part of this system could be used independently each other and also for other purpose than impact analysis, the dependencies generation are useful for understanding the system and the Widget graph could be used for many applications that require the display of dependencies (network, documentation structure, etc.).

6. Case Studies

6.1. Simple Example

The system on which we will show graphically the impact analysis will be the program of *Figure 10* (*Hardware integer division algorithm*). The initial modification will be a "private_change" on the variable "resource: y of type int" (On a call to the function "division", the internal value of the formal parameter is changed). However this visualization of the propagation of such a modification has been done manually. Without a tool to propagate automatically, because the communication between the graphical tool and the propagation engine has not been yet completely realized. The model used to represent the system is very important because the granularity of the result depends on the granularity of the model. We will not know what are the functions affected by the modification of another one if we do not model the functions but just the modules and their dependencies between them. So the dependency model and the propagation model of the system have to be in coherence with the goal that the maintainer needs to reach for the impact analysis.

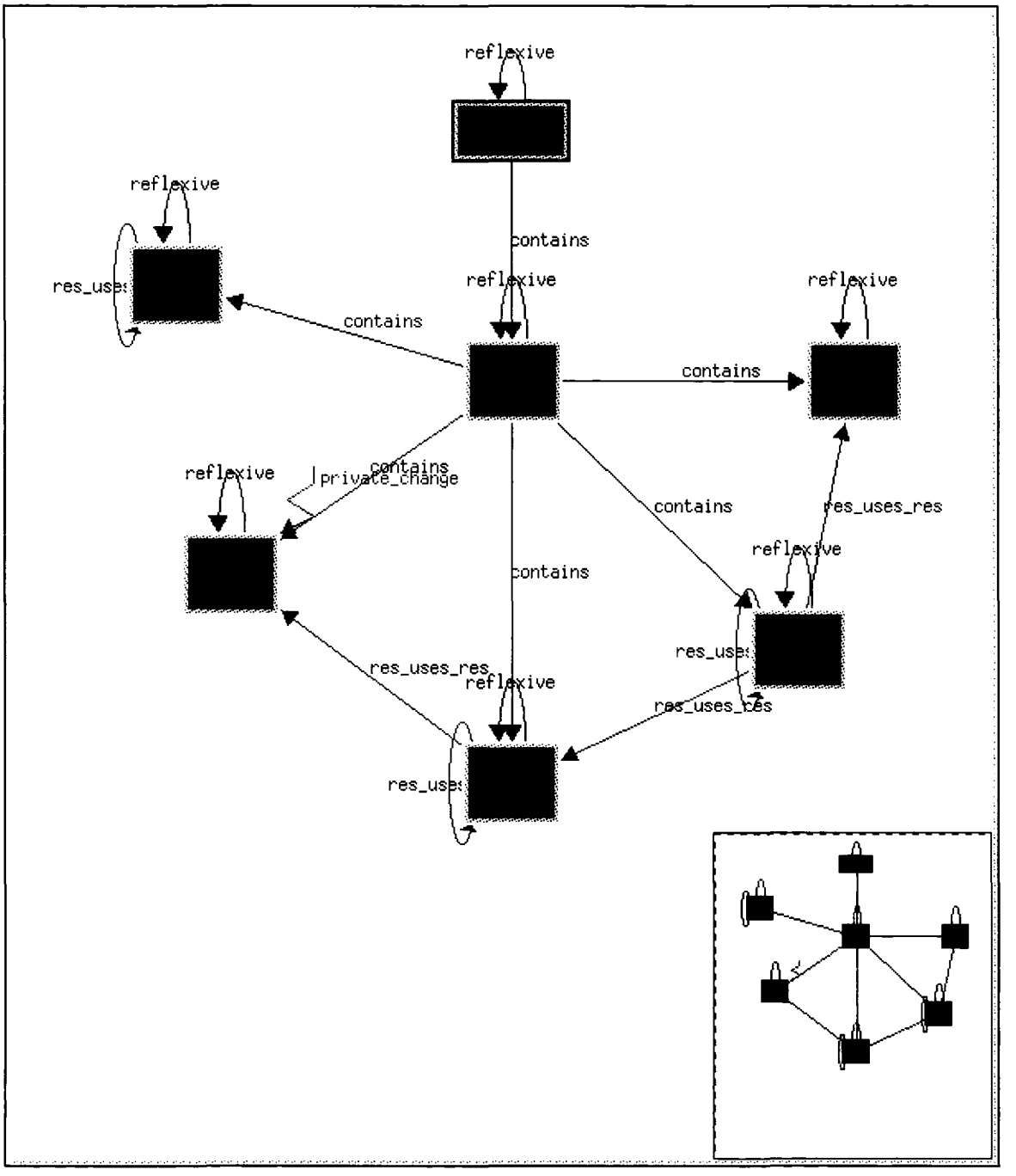


Figure 38 : First step of the propagation

Figure 38 shows the dependencies for the *Hardware integer division algorithm* and a user modification on the system ("private_change" on the variable "y").

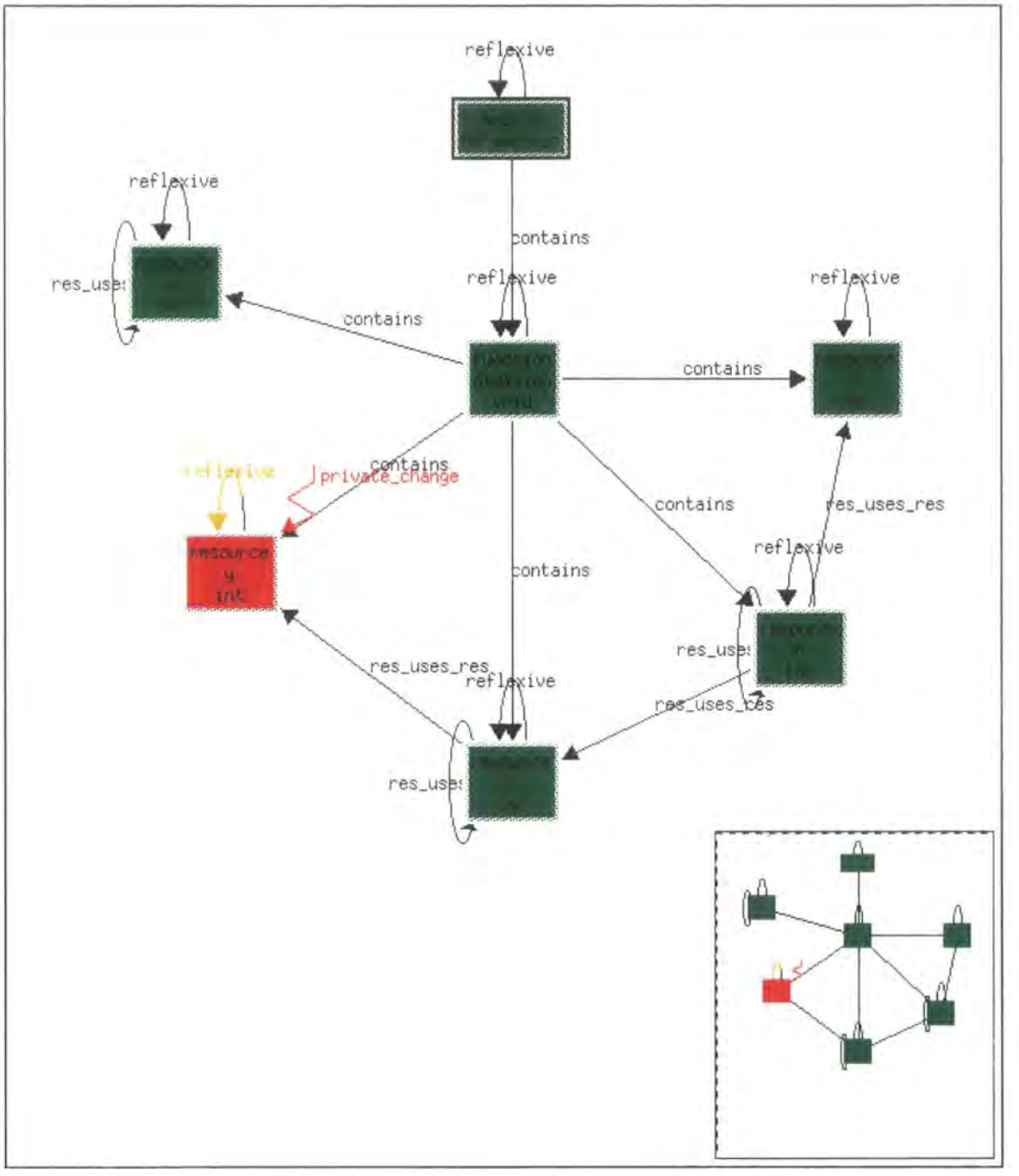


Figure 39 : Second step of the propagation

The modification "private_change" on the object "y" is propagated through the link "reflexive" on the same object:

```
propagation_rule(resource,private_change,reflexive,resource,
                 public_change, no)
```

which means that a modification of type "private_change" on object of type

"resource" like "y" is propagated through a link of type "reflexive" to an object of type "resource" with a modification of type "public_change" and this modification is potential compared to the previous one (As the previous one was User Defined (Impact Level=0) this one will be Propagated (Impact Level=0+1=1)).

The propagation through the link can be graphically animated as shown in *Figure 39* as a two coloured line on which the colour (colour which represents the level of impact) of the impacted path moves as the propagation takes place.

Figure 40 shows the propagation through the links ("contains" and "res_uses_res") of the impact "public_change" on the object "resource: y of type int":

```
propagation_rule(resource,public_change,contains,
                 function,private_change, yes)
```

and

```
propagation_rule(resource,public_change,
                 res_uses_res,resource, private_change, yes).
```

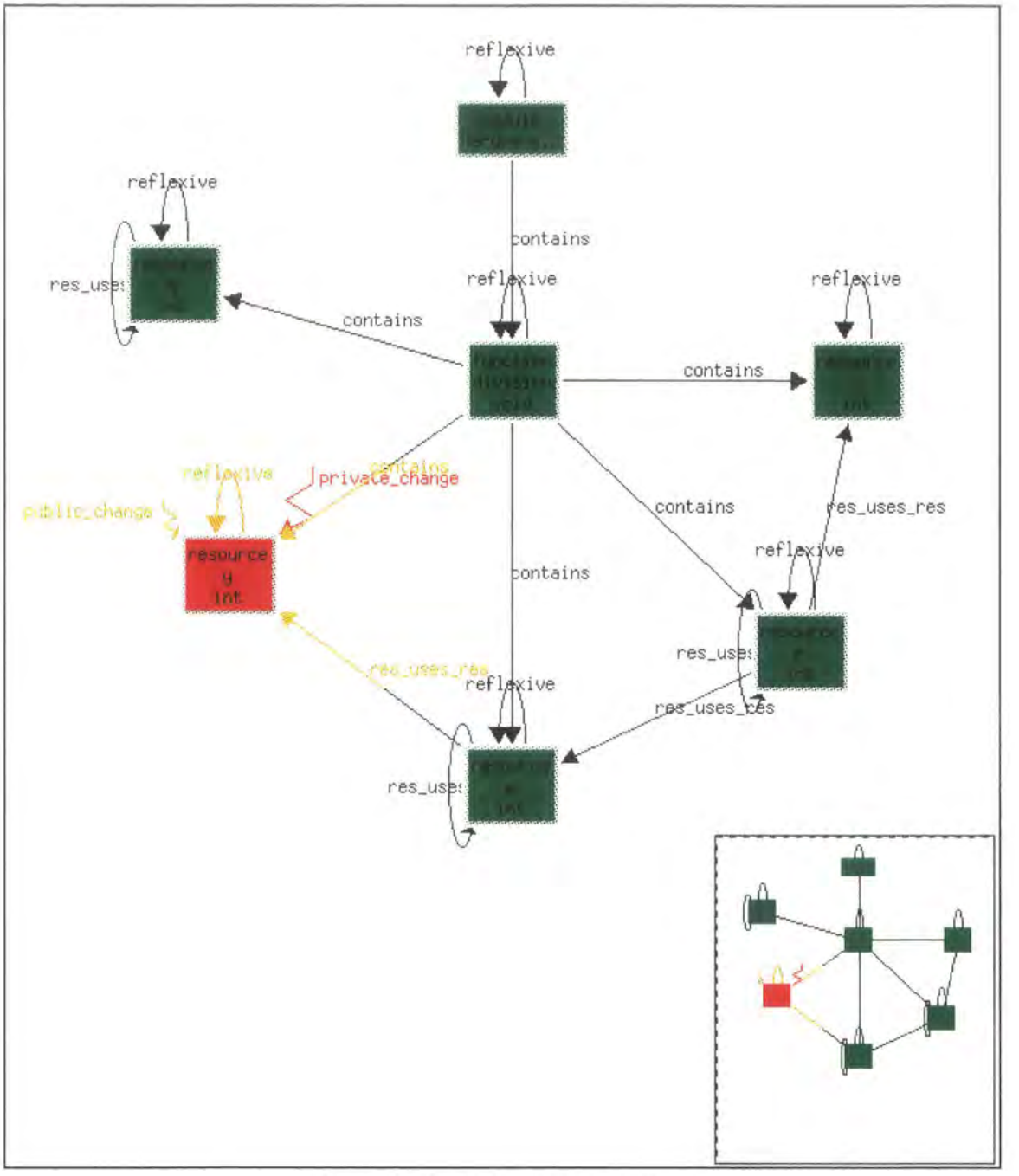


Figure 40 : Third step of the propagation

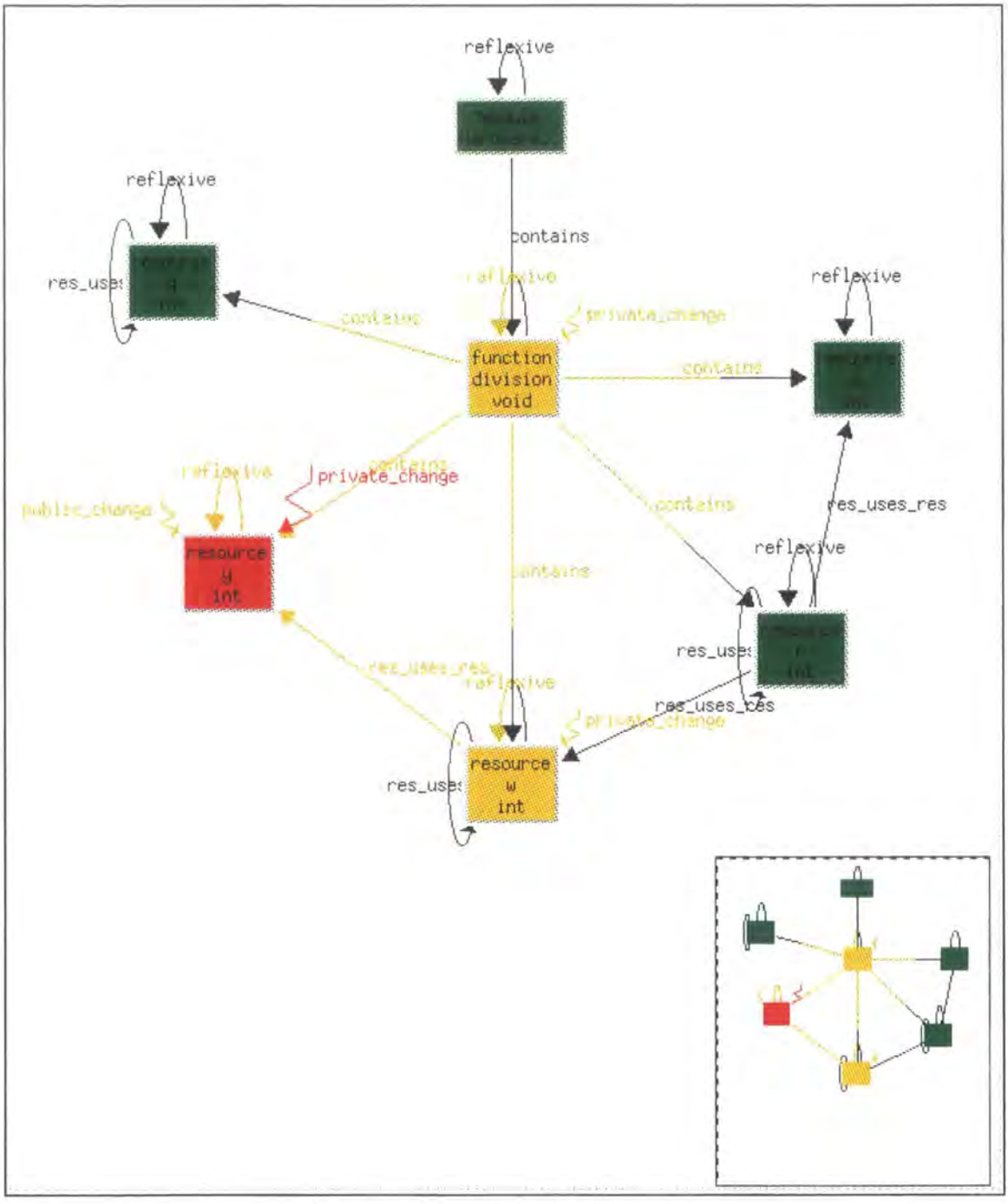


Figure 41 : Fourth step of the propagation

Figure 41 shows the two impacts on "function: division" and on "resource: w of type int" which are themselves propagated through the links "contains" and "reflexive" to others objects (*Figure 42*).

The colours of the objects corresponds to the level of impact of the modifi-

cations which are applied to the objects. For instance the red colour means that a real modification has been done on an object, green no impact and orange is the first level of potential impact after red. An orange impact means that this modification has to be checked by the maintainer in order to validate or invalidate the fact that it is real or not.

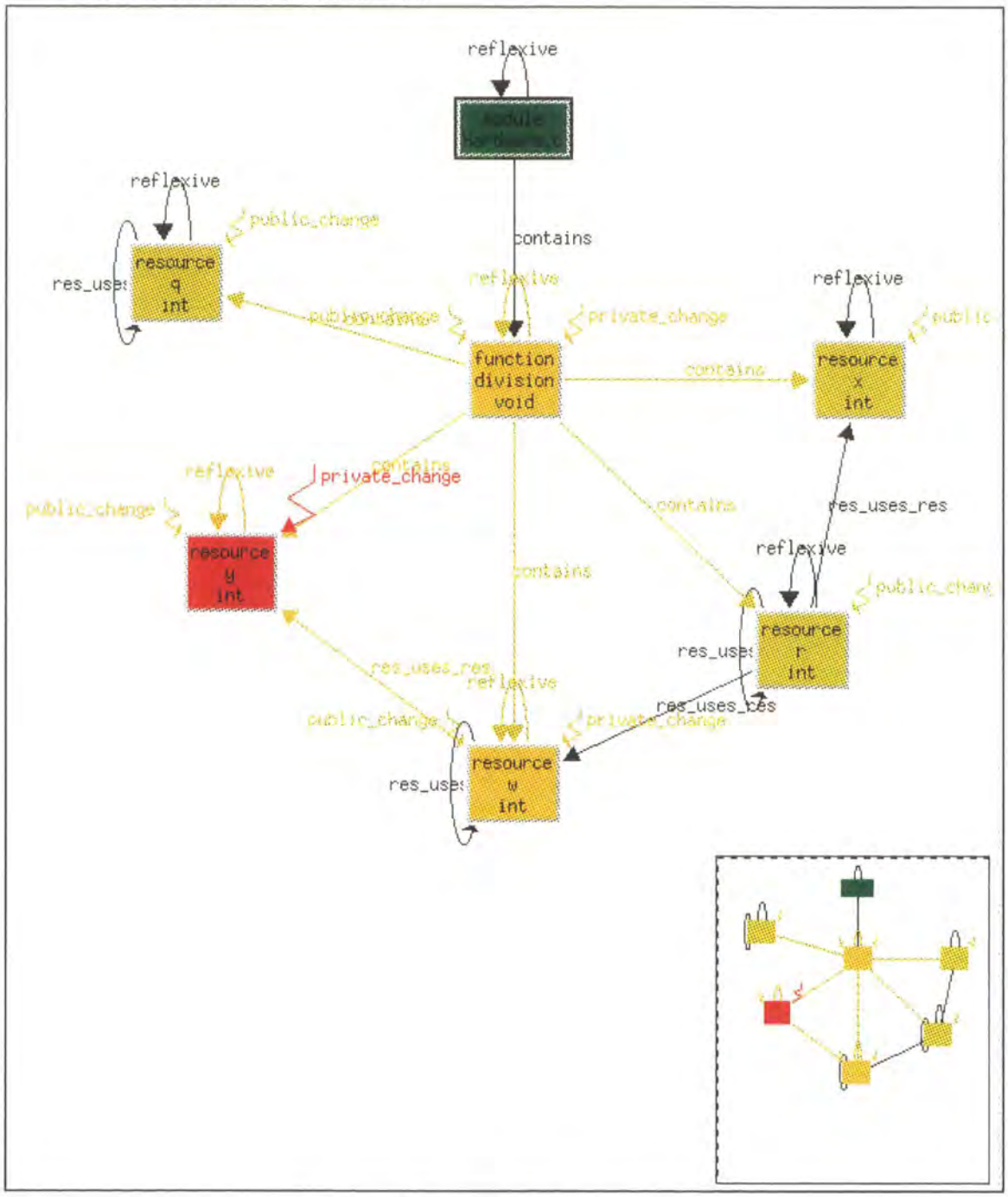


Figure 42 : Fifth step of the propagation

The next steps will not be described as the description is always the same as previously.

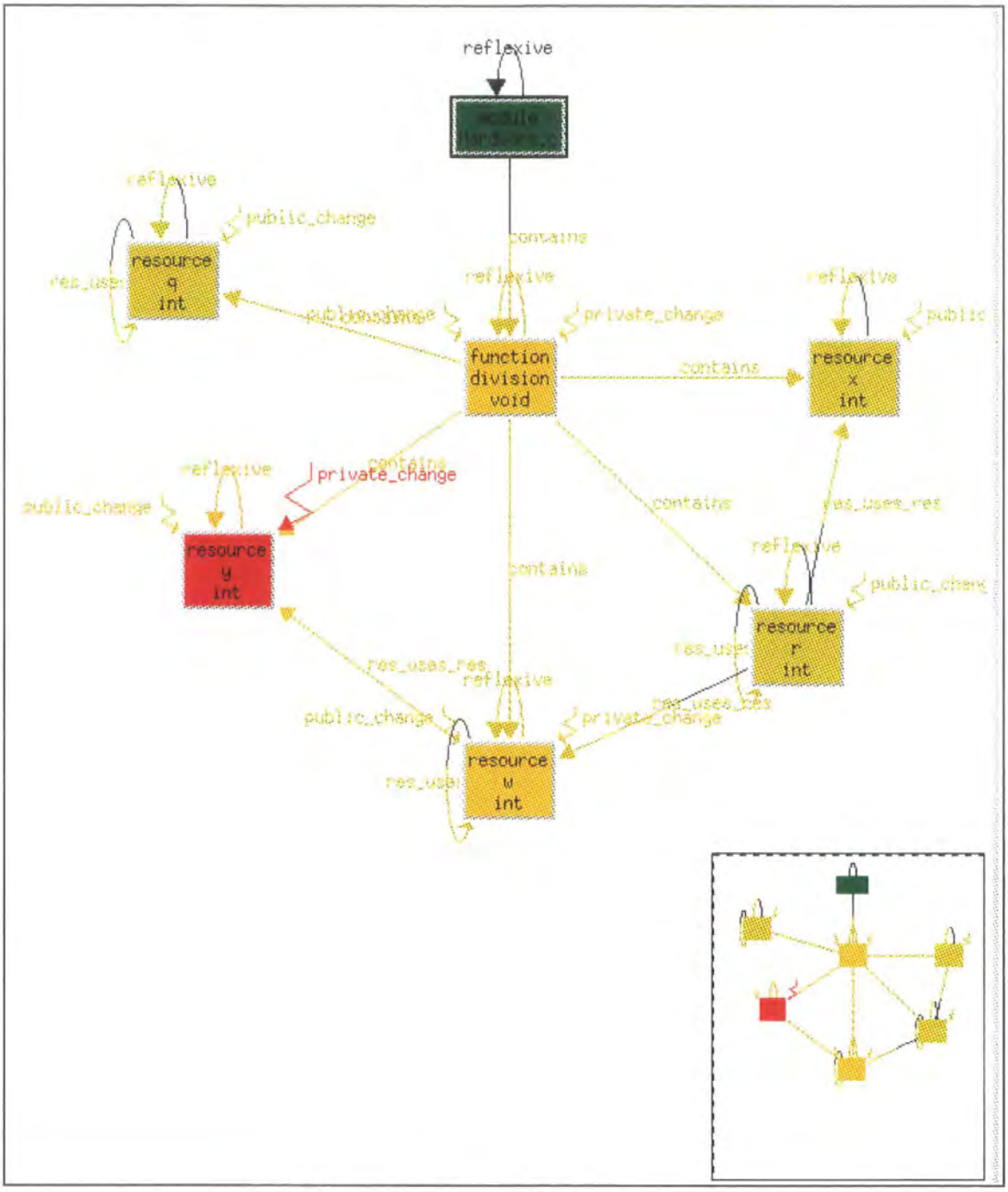


Figure 43 : Sixth step of the propagation

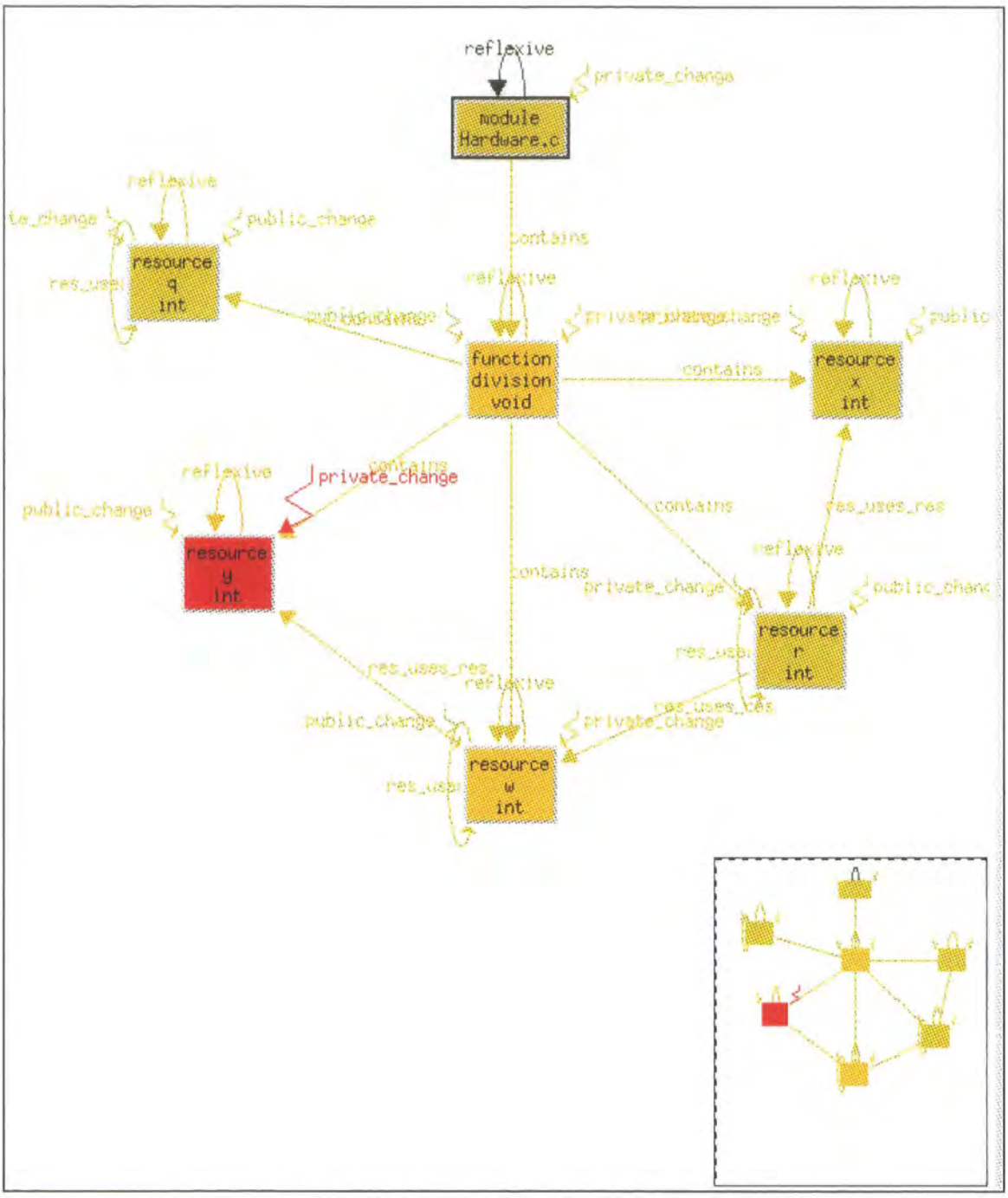


Figure 44 : Seventh step of the propagation

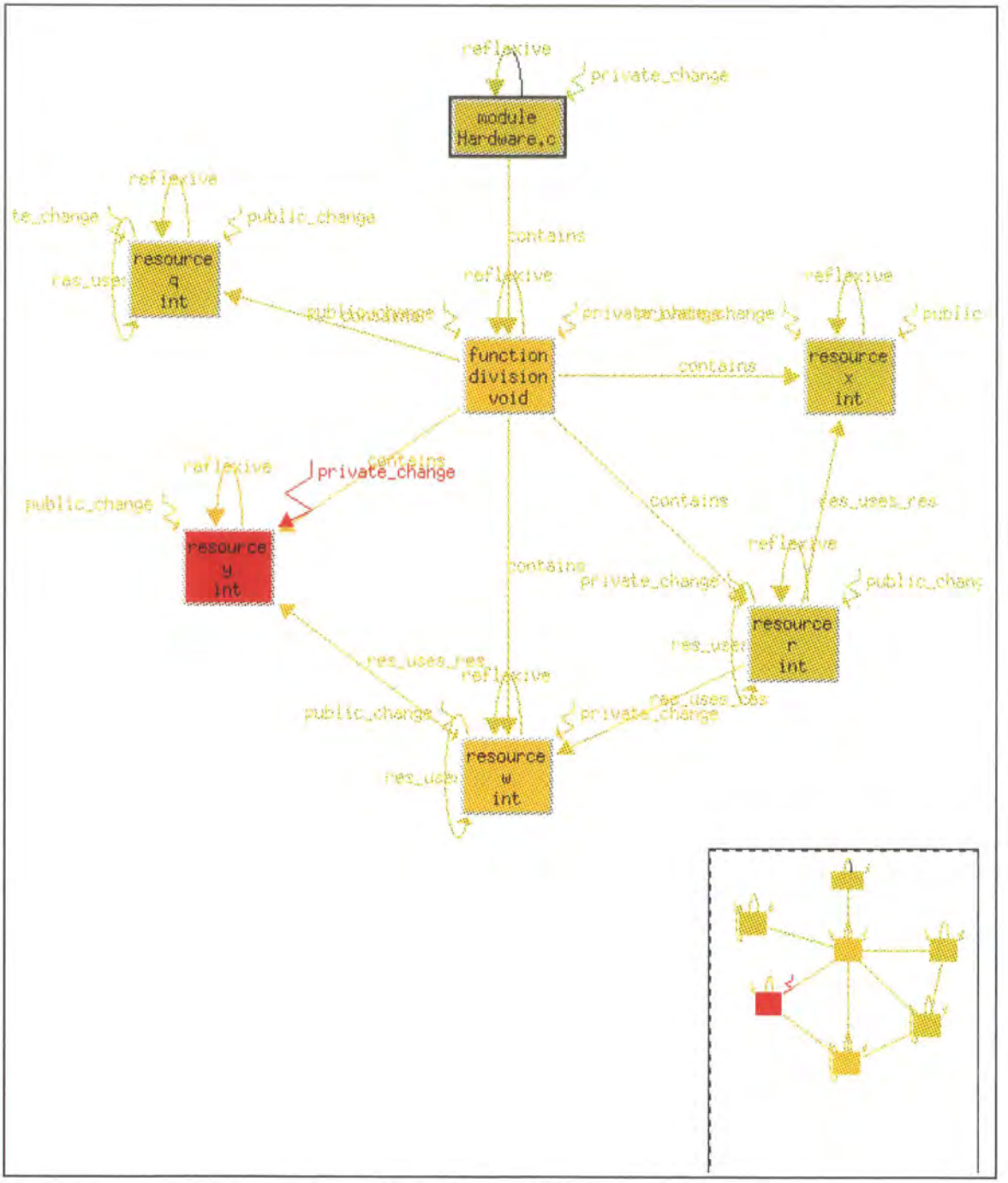


Figure 45 : Eighth step of the propagation

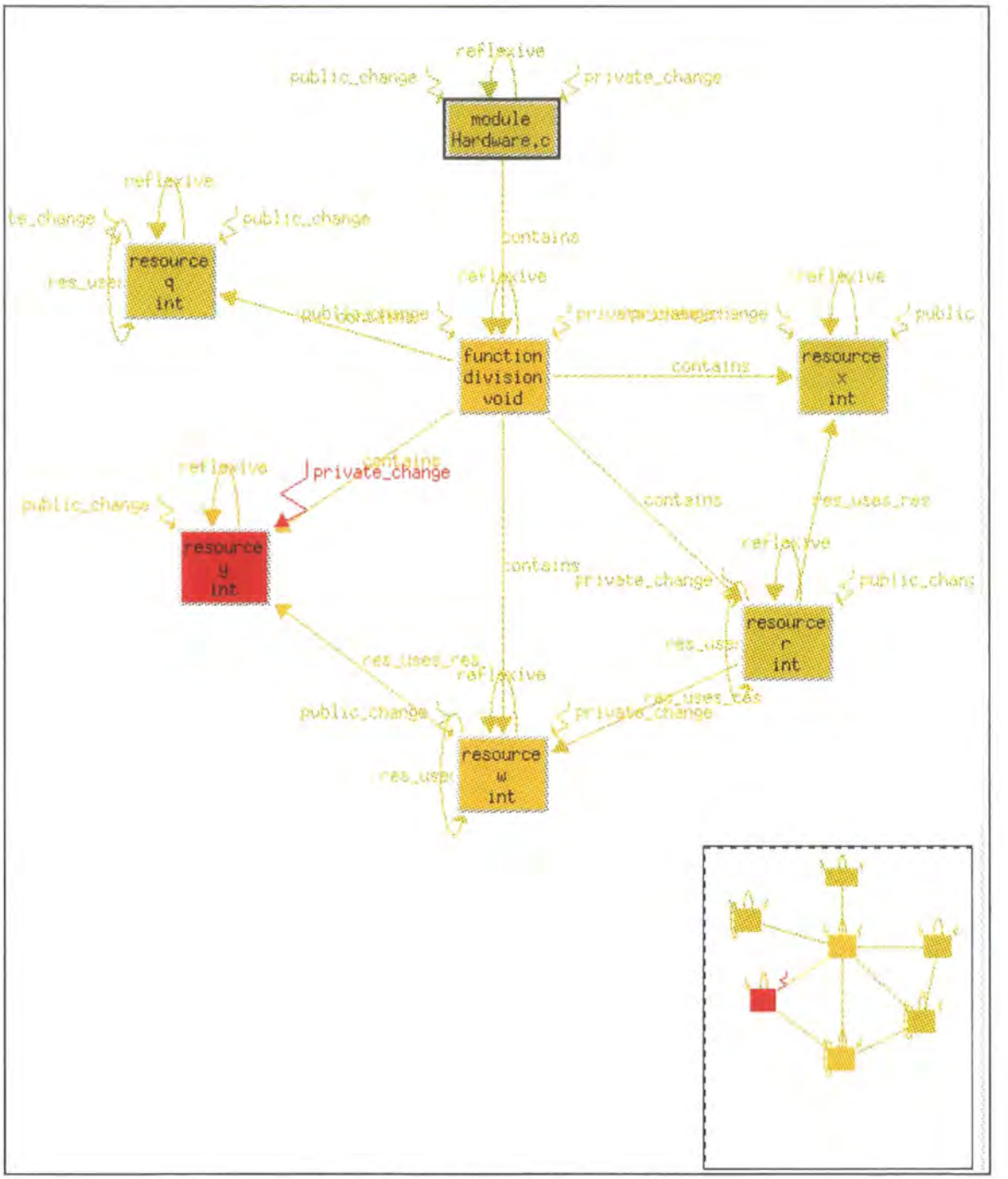


Figure 46 : Ninth step of the propagation

On the last step of the propagation, it can be seen (*Figure 46*) that all the system is potentially impacted by the initial modification. However the potentiality level of the different impacts (triggers) indicates which parts of the system seem to be really affected by the initial modification.

6.2. Larger Example

6.2.1. The Graph Tool System

The Graph Tool System is a X/Motif Library (for the Widget XmGraph) and several programs which test or use this library. The Graph Tool System will be considered here like the library and the test program: "gtest".

It is composed of 5 ".c" files and 4 ".h" files: gtest.c, libXmGraph.a (XmGraph.c, XmArc.c, XmGraphUtil.c, XmGraphPrint.c) and XmGraphP.h, XmGraph.h, XmArcP.h, XmArc.h. The extension "P.h" is for a private usage of the private components of the library as the extension ".h" is for end user usage. The length of the ".c" and ".h" files is: 26000 lines and 700 Ko of C source code.

6.2.2. Generation of Dependencies

A sub-set of dependencies for "gtest" are is shown in *Figure 47*. For the system these are 4420 objects and 3323 links.

```
id_object([0,0],['XmArc.c',none,none,none,none]).
id_object([0,1],['XmGraph.c',none,none,none,none]).
id_object([0,2],['XmGraphPrint.c',none,none,none,none]).
id_object([0,3],['XmGraphUtil.c',none,none,none,none]).
id_object([0,4],['gtest.c',none,none,none,none]).
id_object([1,0],['@external',sc([], @),'Boolean','ArcVisibleInGraph',['@fun']]).
id_object([1,1],['@external',sc([], @),'Boolean','ExistAnStructuralLinkBetween',['@fun']]).
id_object([1,2],['@external',sc([], @),hrtime_t,gethrtime,['@fun']]).
...
...
id_object([995,0],['@external',sc([], extern),void,'XmAddTabGroup',['@fun']]).
id_object([997,0],['@external',sc([], extern),void,'XmChangeColor',['@fun']]).
id_object([998,0],['@external',sc([], extern),void,'XmCvtStringToUnitType',['@fun']]).
id_object([999,0],['@external',sc([], extern),void,'XmFontListEntryFree',['@fun']]).
object([0,0],module).
object([0,1],module).
object([0,2],module).
object([0,3],module).
object([0,4],module).
object([1,0],function).
object([1,1],function).
object([1,2],function).
object([1,3],function).
object([1,4],function).
object([10,0],function).
link([0,0],contains,[1,0]).
link([0,0],contains,[10,0]).
link([0,0],contains,[11,0]).
link([0,0],contains,[12,0]).
link([0,0],contains,[13,0]).
...
...
link([98,4],contains,[1946,4]).
link([98,4],fun_uses_fun,[1111,0]).
link([98,4],fun_uses_fun,[612,0]).
link([98,4],fun_uses_fun,[637,0]).
link([99,4],contains,[1947,4]).
link([99,4],contains,[1948,4]).
link([99,4],contains,[1949,4]).
link([99,4],fun_uses_fun,[45,3]).
```

Figure 47 : Sub-set of Dependencies for "gtest".

6.2.3. Structure Visualisation of the Graph Tool

A part of the dependency graph for "gtest" is shown in *Figure 48.*

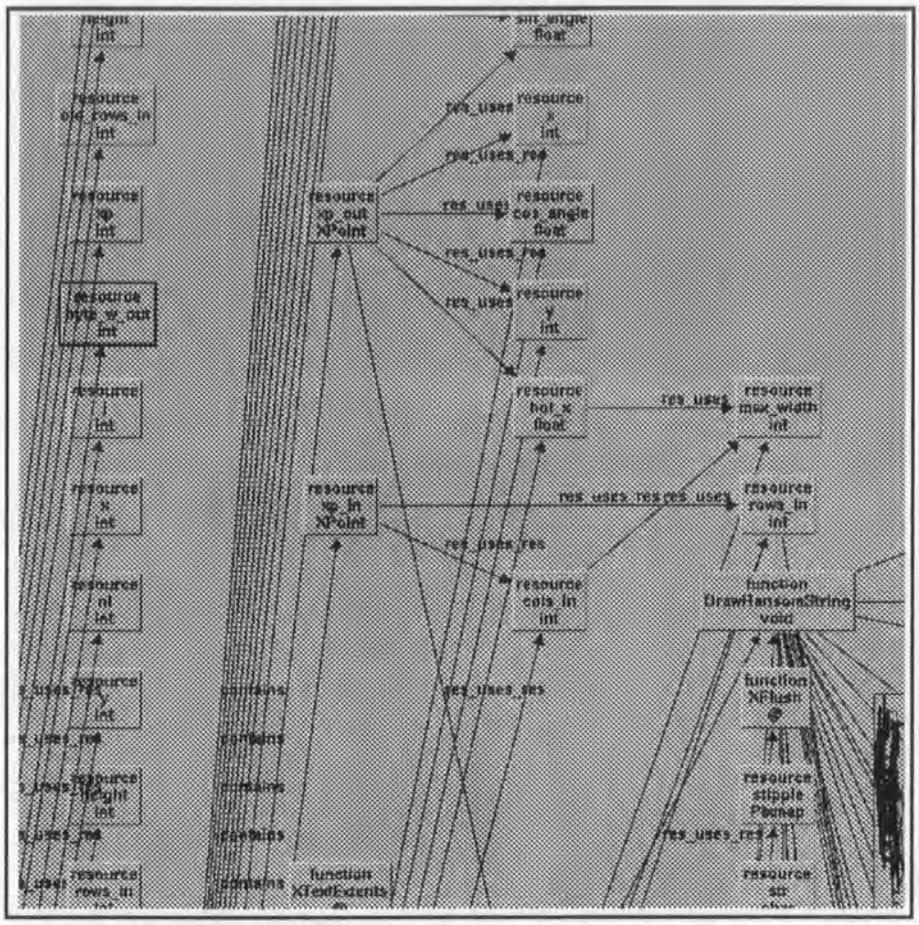


Figure 48 : Sub-graph of the Dependencies for "gtest".

The graph in *Figure 48* is very large because every code object (function, variable) has been represented as a node in the graph. In order to reduce the size of the graph but still represent these dependencies, we need to structure the system into modules and even in functional levels. So each level of the structure could be a graph which can be obtained by opening the node in the upper level of the structure.

6.3. Summary

As the model of the propagation will be finer the description of the impacts will be also more precise. However, the maintainer will have to validate the impacts on the system by his knowledge of the system and by looking at the

system (at the position of the impact). This feedback that the maintainer will have to give to the propagation engine is necessary as the information (Dependencies) given to this tool would have been generally produced by automatic dependencies generation. This can lead to an incomplete representation of the dependencies of the system because of the complexity of such a task. The drawback of visualisation of large systems is that the graph becomes very large and thus needs to be structured in some way.

7. Conclusion

7.1. Summary

The results of this master are the dependencies generation for C code and a tool to visualize, as a graph, the dependencies and the impacts on them of some modifications. Some studies (Graph theory, Software Maintenance Tools and Interaction paradigms) have been achieved in order to obtain these results.

7.2. Achievement of the Research

This research has achieved the different proposed goals:

- *How to model the system on which the impact analysis has to be performed ?*

  The system can be modelled as a set of objects and links as shown in *5.4. Dependencies Generation* and *5.4.2. Inference* for a particular granularity of the system.

- *How to propagate the information that an object has been modified through the system and to determine the possible impacts of this change ?*

  The system is supposed to be modelled in terms of objects and links And a propagation model is modelled according to the dependency model used to represent the system. The propagation engine (*5.5. Propagation Engine*) together with the initial set of modifications that the maintainer want to apply on the system will give as result the set of impacted objects of the system.

- *How to visualize the system and the impacts of a set of changes on it ?*

  The graph tool developed during this master allows the maintainer to see the system as a graph, representing the dependencies between the different entities of the system, and as the impact analysis is performed to see the impacts (triggers on the objects of the system) on the system.

As criteria for success (*1.3. Criteria for Success*):

- Description of a model for *Software Maintenance* process focuses on *Impact Analysis*.

  Figure 5 and *Figure 5* show a *Software Maintenance* process which include and focus on the *Impact Analysis* task.

- An *Impact Analysis System* that will visually represent the connectivity between objects and show the different impacts and their importances:

  The graph tool which has been developed allows the *Impact Analysis* system which will be done to do these two requirements.

- Evaluation of the *Impact Analysis System*:

  An evaluation of the *Impact Analysis System* has been done for the graph tool and the propagation engine with the models (Dependency and Propagation) in *6. Case Studies* even if the evaluation has been done without the complete and integrated *Impact Analysis* system.

7.3. Further Research

Several areas of this research would have to be followed:

- The dependency generation is of great interest because it is needed for testing purpose for the impact analysis on real system.
- The graph tool needs still a lot of improvements concerning the automatic placement of nodes and links in the graph (for instance the aesthetics could be chosen by the user).
- The graph tool could be improved as a navigation and browser tool (which has been already partially done with the integration with mosaic (The graphical World Wide Web browser for Unix/X/Windows platform)).
- The impact analysis tool together with the navigation and browser tool will have to be completed as an extension of the graph tool.
- The graph tool need to address the issue of displaying large graphs using some structuring of the subject system.

8. Glossary

Adaptive maintenance
It is the maintenance which is required because of changes in the environment of the software system.

Arcs incident to and from a node
See *Arcs incident to and from a node [CARRE_91].*

Arcs incident to and from a node *[CARRE_91]*
If an arc A has a node n*i* as its initial end-point, we say that the arc is incident from n_i; whereas if an arc A has node n_j as its terminal end-point we say that arc A is incident to n_j. The number of arcs incident from a node n_i is called the out-degree of n_i and it is denoted by $\rho^+(n_i)$; while the number of arcs incident to n_i is called the in-degree of ni and is denoted $\rho^-(n_i)$.

Centre of Software Maintenance

Change modelling
See *Change modelling [ARNOLD_93].*

Change modelling *[ARNOLD_93]*
Using objects and relationships to characterize a change.

Component
See *Component [QUEILLE_93].*

Component *[QUEILLE_93]*
A software system comprises the following elements (list not extensive):
* source code,
* generation and installation procedures, with associated generation and installation tools,
* data files which may be required to execute properly the software,
* usage documentation and operational procedures, with associated tools,
* development documentation (requirement specifications, design specifications,...), if this documentation is maintained, and/or maintenance documentation, with associated tools,
* test cases, with associated tools,
The word "component" will be used to designate any of these elements at any different granularity levels.

Corrective maintenance

It is the correction of previously undiscovered system errors.

Decomposition [ARNOLD_93]

A representation of one document that features new, parsed information not explicitly parsed in the original document, is a decomposition.
Example:
Non-decomposed: "Bob saw the..."
Decomposed: {(Bob subject) (saw verb) (the article-for-object)...}

Dependency model

It is a representation in terms of types of objects/links, types of link that are allowed between two types of objects that can allow the representation of a software system by instantiation of this model on the system to analyse.

Development life-cycle *[IEEE_83]*

It is the period of time that begins with the decision to develop a software product and ends when the product is delivered. The development cycle typically includes a requirement phase, design phase, implementation/testing phase and integration/testing phase.

Digraph

is a directed graph, all the arcs (edges) between nodes are directed: they have a source and a destination node.

Exploratory programming

See *Exploratory programming [SOMMERVILLE_92].*

Exploratory programming *[SOMMERVILLE_92]*

This approach involves developing a working system, as quickly as possible, and then modifying that system until it performs in an adequate way. This approach is usually used in artificial intelligence (AI) systems development where users cannot formulate a detailed requirements specification and where adequacy rather than correctness is the aim of the system designers.

Formal transformation

See *Formal transformation [SOMMERVILLE_92].*

Formal transformation *[SOMMERVILLE_92]*

This approach involves developing a formal specification of the software system and transforming this specification using correctness-preserving transformations, to a program.

Impact Analysis
See *Impact Analysis [WILDE_94].*

Impact Analysis *[WILDE_94]*
The task of assessing the effects of making a set of changes to a software system.

Initial and terminal end-points of an arc
See *Initial and terminal end-points of an arc [CARRE_91].*

Initial and terminal end-points of an arc *[CARRE_91]*
For an arc (n_i,n_j), the node n_i is the initial end-point and the node n_j is the terminal end-point.

Matra Marconi Space France
French-English company specialized in Aerospace domain. This company has supported my work and my MSc in Durham university.

Partial graphs
See *Partial graphs [CARRE_91].*

Partial graphs [CARRE_91]
If we remove from a graph G=(N,A) a subset of its arcs, we are left with a graph of the form: $(H = (N, A'))$, $where\ (A' \subseteq A)$
which is called a partial graph of N.

Paths and cycles
See *Paths and cycles [CARRE_91].*

Paths and cycles *[CARRE_91]*
A path is a finite sequence of arcs of the form:

$$\mu = \left(n_{i_0}, n_{i_1}\right), \left(n_{i_1}, n_{i_2}\right), \ldots, \left(n_{i_{r-1}}, n_{i_r}\right)$$

i.e. a finite sequence of arcs in which the terminal node of each arc coincides with the initial node of the following arc; the number of arc 'r' of arcs in the sequence is called the order of the path. The initial end-point of the first arc and the terminal end-point of the last arc of a path are called respectively the initial and terminal end-point of the path. A path whose end-points are distinct is said open, whereas a path whose end-points are

coincide is called a closed path, or cycle. A path is elementary if it does not traverse any node more than once, i.e. if all the initial end-points (or all the terminal end-points) of its arcs are distinct. It is evident that a path is completely determined by the sequence of nodes $n_{i_0}, n_{i_1}, \cdots, n_{i_r}$ which it visits; we shall sometimes find it convenient to specify a path by listing this node sequence rather than the arc sequence.

Planar

A drawing (of a graph) is planar if no two arcs intersect.

Planar representation

A planar representation is a data structure representing the combinatorial adjacencies between the faces of a planar drawing.

Perfective maintenance

It means changes which improve the system in some way without changing its functionality.

Preventive maintenance

It includes the activities designed to make the code, design and documentation easier to understand and to work with, such as restructuring or documentation up-dates. This type of maintenance usually improves the maintainability of the system.

Propagation model

It is a representation in terms of types of modifications allowed on objects/ links and propagation rules of modifications on the *Dependency model.* It depends on the dependency model on which it is based.

Propagation rule

It is a representation of modification on the state of a system from a set of changes on the system to another set of changes on the same system.

Prototyping

See *Prototyping [SOMMERVILLE_92].*

Prototyping *[SOMMERVILLE_92]*

This approach is similar to *Exploratory programming* in that the first phase of development involves developing a program for user experiment. However, the objective of the development is to establish the system requirements. This is followed by a re-implementation of the software to produce a production-quality system.

Software Engineering

See *Software Engineering [BOEHM_76],Software Engineering [IEEE_90].*

Software Engineering *[BOEHM_76]*

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

Software Engineering *[IEEE_90]*

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

Software life-cycle *[IEEE_83]*

It is the period of time that starts when a software product is conceived and ends when the product is no longer available for use. The software life-cycle typically includes the development life-cycle and the operation and maintenance phase.

Software Maintenance

See *Software Maintenance [IEEE_90].*

Software Maintenance *[IEEE_90]*

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

Sub-graphs

See *Sub-graphs [CARRE_91].*

Sub-graphs [CARRE_91]

If we remove from a graph G=(N,A) a subset of its nodes, together with all the arcs incident to or from those nodes, we are left with a graph of the form:

$$(H = (N', A')), where\ (N' \subseteq N)\ ,A' = A \cap (N' \times N')$$

which is called a sub-graph of N. We may describe H more precisely, as the sub-graph of G generated by N'.

System assembly from reusable components

See *System assembly from reusable components [SOMMERVILLE_92].*

System assembly from reusable components *[SOMMERVILLE_92]*
This technique assumes that systems are mostly made up of components which already exist. The system development process becomes one of assembly rather than creation.

Traceability
See Traceability [ARNOLD_93].

Traceability *[ARNOLD_93]*
Finding the objects and relationships affected by a change.

Ripple Effects
See *Ripple Effects [YAU_78].*

Ripple Effects *[YAU_78]*
They are the phenomena by which changes to one program area have tendencies to be felt in other program areas.

University Of Durham
The University Of Durham is the third oldest in England and was founded in 1832. It consists of eleven Colleges and two Societies, each of which controls its own undergraduate admissions and accommodations. Postgraduate admissions and accommodations are controlled centrally through the Postgraduate Admission Office. This Office will also assist in finding appropriate accommodation.

9. Bibliography

[ARNOLD_82]
"The dimensions of healthy maintenance"
R.S. ARNOLD and D.A. PARKER
Proceedings of the 6th International Conference Software Engineering
pp 10-27
1982

[ARNOLD_93]
"Software Impact Analysis"
R.S. ARNOLD
Conference on Software maintenance (A One Day Seminar)
1993

[BATINI_84]
"What is a good diagram ? A pragmatic approach"
C. BATINI, L. FURLANI and E. NARDELLI
Proceedings 4th International Conference on Entity Relationship Approach
Chicago, IL
1985

[BARROS_94]
"Specification of a Propagation Engine for Impact Analysis"
S. BARROS/ MATRA MARCONI SPACE
AMES Deliverable D3.4.2, version 1.0,
1994.

[BATTISTA_88]
"Hierarchies and Planarity Theory"
G. Di BATTISTA and E. NARDELI
IEEE Transactions on Systems, Man and Cybernetics, 18 (6)
pp 1035-1046
November/ December 1988

[BATTISTA_93]
"Algorithms for Drawing Graphs: An Annotated Bibliography"
G. DI BATTISTA, P. EADES, R. TAMASSIA and I.G. TOLLIS
1993

[BENNETT_91]
"Software maintenance"
K. BENNETT, B. CORNELIUS, M. MUNRO, D. ROBSON
Software Engineer's Reference Book
Chapter 20
J.A. McDermid, Oxford: Butterworth-Heinemann
1991

[BIRJANDI_89]
"Code Analysis and Maintenability"
A. BIRJANDI
IEEE Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences, II()
pp 477-478
1989

[BLOESCH_93]
"Aesthetic Layout of Generalized Trees"
A. BLOESCH
Software-Practice and Experience, 23(8)
pp 817-827
August 1993

[BOEHM_75]
"The high cost of software"
B.W. BOEHM
Practical Strategies for Developing Large Software Systems (Horowitz E., ed.)
Reading MA: Addison-Wesley
1975

[BOEHM_76]
"Software Engineering"
B.W. BOEHM
IEEE Transactions on Computers, 25(12)
pp 1226-1241
1976

[BOEHM_88]
"A Spiral Model of Software Development and Enhancement"
B.W. BOEHM
Computer, IEEE Computer Society
pp 61-72
May 1988

[BOLLOBAS_78]
"Extremal Graph Theory"
B. BOLLOBÁS
1978

[BUNTER_93]
"PERPLEX: An Extensible Tool Architecture for C Source Code"
T.A. BUNTER
Computer Science Technical Report
University of Durham
June 93

[BURNS_88]
"A Graphical Entity-Relationship Database Browser"
L.M. BURNS, J.L. ARCHIBALD and A. MALHOTRA
IEEE Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences, II ()
pp 694-704
1988

[CALLISS_88]
"Dynamic Data Flow Analysis of C Programs"
F.W. CALLISS and B.J. CORNELIUS
IEEE Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences, II()
pp 518-523
1988

[CALLISS_89]
"Two Module Factoring Techniques"
F. W. CALLISS and B. J. CORNELIUS
Software Maintenance: Research and Practice, I()
pp 81-89
1989

[CALLISS_90]
"Potpourri Module Detection"
F.W. CALLISS, B.J. CORNELIUS
Proceedings Conference on Software Maintenance (San Diego), IEEE Computer Society Press
pp 46-51
1990

[CANFORA_93]

"Extracting Abstract Data Types from C Programs: A Case Study"
G. CANFORA, A. CIMITILE, M. MUNRO, C.J. TAYLOR
1993

[CARPANO_80]
"Automatic Display of Hierarchized Graphs for Computed-Aided Decision Analysis"
M.J. CARPANO
IEEE Transactions on Systems, Man, and Cybernetics, SMC-10(11)
pp 705-715
November 1980

[CARRE_91]
"Graph Theory"
B. CARRE
Software Engineer's Reference Book
Chapter 4
J.A. McDermid, Oxford: Butterworth-Heinemann
1991

[COLBROOK_89]
"The Retrospective Introduction of Abstraction into Software"
A COLBROOK and C. SMYTH
Conference on Software Maintenance, Miami, IEEE
pp 166-173
1989

[COOPER_88]
"Interprocedural Side-Effect Analysis in Linear Time"
K. D. COOPER and K. KENNED
Proceedings of the SIGPLAN'88
Page 57-66
1988

[CORBI_88]
"Code Analysis and Maintenability"
T. A. CORBI
IEEE Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences, II()
pp 490-491
1988

[CORDY_90]

"TuringTool: A User Interface to Aid in the Software Maintenance Task"
J. R. CORDY, N. L. ELIOT and M. G. ROBERTSON
IEEE Transactions on Software Engineering, 16(3)
pp 294-301
March 1990

[DANNENBERG_90]
"A Structure for Efficient Update, Incremental Redisplay, and Undo in Graphical Editors"
R. B. DANNENBERG
Software Practice and Experience, 20(2)
pp 110-132
February 1990

[DEO_74]
"Graph theory with applications to Engineering and Computer Science"
N. DEO
Prentice-Hall series in Automatic Computation
1974

[DING_90]
"A Framework for the Automated Drawing of Data Structure Diagrams"
C. DING and P. MATETI
IEEE Transactions on Software Engineering, 16(5)
pp 543-557
May 1990

[DUNLAVEY_93]
"Differential Evaluation: a Cache-based Technique for Incremental Update of Graphical Displays of Structures"
M. R. DUNLAVEY
Software- Practice and Experience, 23(8)
pp 871-893
August 1993

[GALLAGHER_91]
"Using Program Slicing in Software Maintenance"
K. B. GALLAGHER and J. R. LYLE
IEEE Transactions on Software Maintenance
1991

[GANSNER_88]
"DAG, A Program that Draws Directed Graphs"
E.R. GANSNER, S.C. NORTH and K.P. VO

Software Practice and Experience, 18(11)
pp 1047-1062
November 1988

[GAREY_79]
"Computers and Intractability: A guide to the theory of NP-Completeness"
M.R. GAREY,D. S. JOHNSON
1979

[GOPAL_89]
"Using Automatic Program Decomposition Techniques in Software Maintenance Tools"
R. GOPAL and S.R. SCHACH
IEEE
pp 132-141
1989

[GOLUMBIC_80]
"Algorithmic Graph Theory and Perfect Graphs"
M.C. GOLUMBIC
1980

[HARJANI_92]
"Maintenance in a software factory- Towards an integrated maintenance support environment"
D.R. HARJANI, J.P. QUEILLE, J.F. VOIDROT
Proceedings of the ESF Seminar, Berlin, Germany
pp 1-12
1992

[HENNINGER_89]
"A Knowledge-based Design Environment for Graphical NetWork Editors"
S. HENNINGER, A. IGNATOWSKI, C. RATHKE, D. REDMILES
IEEE Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences, II()
pp 881-891
1989

[HILL_93]
"History-Enriched Source Code"
W.C. HILL and J.D. HOLLAN
ACM Annual Symposium on User Interface Software and Technology Conference (UIST'93)

1993

[HOFFMAN_88]
"Trace Specification: Methodology and Models"
D. HOFFMAN and R. SNODGRAS
IEEE Transactions On Software Engineering, 14(9)
pp 1243-1252
September 1988

[HTML_ANALYZER_94]
Tool to analyse and extract links from HTML documents. The "HTML analyzer" has been developed by James E. Pitkow (pitkow@cc.gatech.edu). Development of this software was funded by the NASA Earth Observing System Project under NASA contract NAS5-32392.

[HUGH_90]
"Algorithmic Graph Theory"
J. A. McHugh
Englewood Cliffs: Prentice Hall
1990

[IEEE_83]
"Software Engineering Standards"
ANSI/IEEE Std729,
1983

[IEEE_90]
"An American National Standard and IEEE Standard Glossary of Software Engineering Terminology"
IEEE Standards Board and ANSI Standards Institute
ANSI/IEEE Std610.12,
1990

[JABLONOWSKI_89]
"GMB: A Tool for Manipulating and Animating Graph Data Structures"
D. JABLONOWSKI and V. A. GUARNA, JR.
Software Practice and Experience,19(3)
pp 283-301
March 1989

[JIANG_91]
"Program Slicing For C - The Problems In Implementation"
J. JIANG, X. ZHOU, D. J. ROBSON

IEEE
pp 182-190
1991

[KELLER_93]
"Abstraction Refinement: A Model of Software Evolution"
B. J. KELLER and R. E. NANCE
Software Maintenance: Research and Practice, 5()
pp 123-145
1993

[KRAMER_89]
"Workstation Environments for Graphical Programming"
B. KRAMER
IEEE Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences, II()
pp 848-849
1989

[LEUNG_89]
"Insights into Regression Testing"
H. K. N. LEUNG and L.WHITE
IEEE
pp 60-69
1989

[LIENTZ_80]
"Software Maintenance Management"
B.P. LIENTZ and E,B, SWANSON
Addison-Wesley
Reading MA,
1980

[LYLE_89]
"A Program Decomposition Scheme with Applications to Software Modification and Testing"
J.R. LYLE and K.B. GALLAGHER
IEEE Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences, II()
pp 479-485
1989

[MADHAVJI_92]
"Environment Evolution: The Prism Model of Changes"
N. H. MADHAVJI

IEEE Transactions on Software Engineering, 18(5)
pp 380-392
May 1992

[MOEN_90]
"Drawing Dynamic Trees"
S. SMOEN
IEEE Software,
pp 21-28
1990

[MORET_91]
"Algorithms from P to NP"
B.M.E. MORET, H.D. SHAPIRO
Design & Efficiency, I()
1991

[MOSER_90]
"Data Dependency Graphs for Ada Programs"
L. E. MOSER, Member IEEE
IEEE Transactions on Software Engineering, 16(5)
pp 498-509
May 1990

[NARAYANASWAMY_88]
"Static Analysis: An Aid to Program Maintenance and Development"
K. NARAYANASWAMY
IEEE Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences, II()
pp 492-499
1988

[OZAWA_80]
"A graph-planarization algorithm and its application to random graphs"
T. OZAWA and H. TAKAHASHI
Lectures Notes in Computer Science (108)
Edited by G. GOOS and J. HARTMANIS
Graph Theory and Algorithms
Sendai, Japan, October 1980, Proceedings
Edited by N. SAITO and T. NISHIZEKI
Springler-Verlag
October 1980

[PAULISCH_88]

"EDGE: An Extendible Graph Editor"
F.N. PAULISCH and W. F. TICHY
Software Practice and Experience, 20(S1)
pp 63-88
June 1988

[PFLEEGER_87]
"Software Engineering: the production of quality software"
S.L. PFLEEGER
Macmillan Publishing
1987

[PFLEEGER_90]
"A Framework for Software Maintenance Metrics"
S.L. PFLEEGER and S. A. BOHNER
IEEE Conference on Software Maintenance
pp 320-327
1990

[PRESSMAN_85]
"Software Engineering: A Practitioner's Approach"
R.S. PRESSMAN.
McGraw Hill, NY, USA
1985

[QUEILLE_93]
"Impact Analysis - Position Paper"
J.P. QUEILLE
Matra Marconi Space France
September 1993

[RAJLICH_90]
"VIFOR: A Tool for Software Maintenance"
V. RAJLICH, N. DAMASKINOS, P. LINOS and W. KHORSHID
Software Practice and Experience, 20(1)
pp 67-77
January 1990

[RAMAMOORTHY_90]
"The Evolution Support Environment System"
C.V. RAMAMOORTHY
IEEE Transactions On Software Engineering, 16(11)
pp 1225-1234

November 1990

[REINGOLD_81]
"Tidier Drawings of Trees"
E. M. REINGOLD and J. S. TILFORD
IEEE Transactions on Software Engineering, SE-7(2)
pp 223-228
March 1981

[REGGIANI_88]
"A Proposed Method for Representing Hierarchies"
M. G. REGGIANI and F. E. MARCHETTI
IEEE Transactions on Systems, Man and Cybernetics, 18(1)
pp 2-8
January/February 1988

[ROYCE_70]
"Managing the development of large software systems"
W.W. ROYCE
Proceedings WESTCON San Francisco CA
1970

[SIMON_91]
"Requirements for a Software Maintenance Support Environment"
A. SIMON
MSc by thesis, University of Durham
1991

[SNEED_89]
"The Myth of "Top-Down" Software Development and its Consequences for Software Maintenance"
H. M. SNEED
Conference on Software Maintenance, Miami, IEEE
pp 22-29
1989

[SOMMERVILLE_92]
"Software Engineering" (Fourth Edition)
I. SOMMERVILLE
Addison-Wesley
1992

[SUGIYAMA_81]

"Methods for Visual Understanding of Hierarchical System Structures"
K. SUGIYAMA, S. TAGAWA and M. TODA
IEEE Transactions on Systems, Man and Cybernetics, SMC-11(2)
pp 109-125
February 1981

[TAMASSIA_88]
"Automatic Graph Drawing and Readability of Diagrams"
R. TAMASSIA, G. Di BATTISTA and C. BATINI
IEEE Transactions on Systems, Man and Cybernetics, 18(1)
pp 61-79
January/February 1988

[TURVER_89]
"Software Maintenance: Generating Front Ends for Cross Reference Tools"
R. J. TURVER
M.Sc Thesis, University of Durham, Dept. Computer Science
1989

[TURVER_93a]
"A Decision Based Model of the Software Maintenance Process"
R. J. TURVER and M. MUNRO
University of Durham, Dept. Computer Science
1993

[TURVER_93b]
"An Early Impact Analysis Technique for Software Maintenance"
R. J. TURVER and M. MUNRO
Software Maintenance: Research and Practice, Vol. 6, 35-52
1994

[TURVER_93c]
"Early Detection of Ripple Propagation In Evolving Software Systems"
R. J. TURVER
University of Durham, Dept. Computer Science
1993

[WARD_89]
"The Maintainer's Assistant"
M. WARD, F.W. CALLISS and M. MUNRO
Conference on Software Maintenance, Miami, IEEE
pp 307-315

1989

[WETHERELL_79]
"Tidy Drawings of Trees"
C. WETHERELL and A. SHANNON
IEEE Transactions on Software Engineering, SE5(5)
pp 514-520
September 1979

[WILDE_87]
"Dependency Analysis: An Aid for Software Maintenance"
N. WILDE and B. NEJMEH
SERC TR-13-F
Software Engineering Research Center,
University of Florida/Purdue University
September 1987

[WILDE_89]
"Dependency Analysis Tools: Reusable Components for Software Maintenance"
N. WILDE, R. HUITT and S. HUITT
Conference on Software Maintenance, Miami, IEEE
pp 126-131
1989

[WILDE_92]
"Locating User Functionality in Old Code"
N. WILDE, J.A. GOMEZ, T. GUST and D. STRASBURG
ICSM-92(International Conference for Software Maintenance).
1992

[WILDE_94]
"The Impact Analysis Task in Software Maintenance: A Model and a Case Study"
N. WILDE, J.P. QUEILLE, J.F. VOIDROT, M. MUNRO.
ICSM 94 (International Conference for Software Maintenance).
1994

[XANTHAKIS_93]
"Une algèbre du flot des données pour l'analyse statique d'un programme"
S. XANTHAKIS & C. SKOURLAS
Internal report, OPL, Département Recherche, UNICITE, 10 Rue Alfred Kasler, 14000 CAEN, FRANCE.

1993

[YAU_78]
"Ripple Effect Analysis of Software Maintenance"
S.S. YAU, J. S. COLLOFELLO and T. MACGREGOR
Proceedings IEEE COMPSAC
pp 60-65
1978

[YAU_80]
"Some Stability Measures for Software Maintenance"
S.S. YAU, and J. S. COLLOFELLO
IEEE Transactions on Software Engineering, SE-6(6)
pp 545-552
November 1980

[YAU_85]
"Design Stability Measures for Software Maintenance"
S.S. YAU, and J. S. COLLOFELLO
IEEE Transactions on Software Engineering, SE-11(9)
pp 849-856
September 1985

Bibliography