A program slicing method for a wide spectrum language

Joan Carrancà i Vilanova

A Program Slicing Method for a Wide Spectrum Language

Joan Carrancà i Vilanova

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

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

September 1992

Supported by a grant from Fundació Crèdit Andorrà

Acknowledgements

I should like to thank the Fundació Crèdit Andorrà for their financial backing, and particularly Josep M^a Pèrez-Gramunt for his kind work on my behalf. I also wish to express my gratitude to my supervisor, Keith H. Bennett, for his patient help and advice; to my friends and companions at Shincliffe Hall for keeping me sane against the odds; to my parents for their unfailing encouragement; and most of all to Sílvia for her support from afar, largely in the form of delicious bars of nougat.

Abstract

This thesis describes the implementation of a program slicer for WSL —a Wide Spectrum Language— which is a language that allows different levels of abstraction to coexist in the same program. WSL contains constructs not found in conventional languages, e.g. action systems (which model a segment of code with GOTOs and labels) and non deterministic constructs. Program slicing is a method for restricting a program to a specified behaviour of interest. Usually this behaviour of interest is expressed in terms of a variable or a set of variables. The method used in the thesis to slice a program is different from the classical ones in that slices do not need to be computed from an output statement, and in that slices are computed on a wide spectrum language closer to a functional language, instead of being computed on a more conventional, procedural language.

A slicer for a subset of WSL has been designed and implemented based on the data flow analysis techniques for while-programs of Bergeretti and Carré [10]. It has been necessary to modify the algorithm to permit incremental slicing. Modifications of their algorithm were also needed to accomodate the specific WSL constructs mentioned above. The implementation has been developed using a rapid prototyping approach. The prototype has provided new ideas and enhancements for a more comprehensive slicer which could be implemented in the future. The slicer has assisted the maintainer using ReForm —a reverse engineering project developed at Durham University— in understanding and debugging a program by decomposing it. At the end of this thesis results showing how slicing has helped the maintainer are presented. Conclusions on the method used, the validity of the tool, and its engineering are also summarized.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Historical review

To understand why slicing, a method for decomposing a program, is important to computer science, we have to go back to the late 1950s. At that time the cost of software was negligible compared with that of hardware, so writing new programs with the same requirements as already existing programs was a normal practice, partly due to incompatible computer architectures and the use of assembler language. Since then hardware has standardised and programs can often run on different computers without major modifications, mainly due to the use of high-level languages and the existence of common operating systems such as DOS or UNIX. As a result the cost of software has been increasing because new and more complex applications have been developed for more powerful computers. The cost of hardware has been decreasing owing to the large introduction of computers which enables them to be mass-produced based on large scale integration. Meanwhile the funds dedicated to the modification or adaption of software (software maintenance) have grown because it is believed that is more expensive to write a new program than to modify or adapt an old one that is already working. The result of all these practices is that in the 1990s less than

15% of the overall hardware-software cost is spent on hardware, and of the remaining 85% more than 60% is dedicated to the maintenance of software [11] [59]. There is an excellent article by Brooks [13] that explains what has happened in the last twenty years from his experienced personal point of view.

1.2 Definitions

Due to the large expenditure made by industry and commerce on software maintenance, improvements made are likely to be cost-effective. The improvements can come from two sources:

- improving the maintainability of new or existing software by applying modern software engineering techniques to it.
- making software maintenance easier by applying better methods, techniques and tools.

We have introduced three concepts that require definitions, namely *maintainability, software engineering* and *software maintenance.*

One definition of maintainability by Martin and McClure [61] is:

> *The ease with which a software system can be corrected when errors or deficiencies occur, and can be expanded or contracted to satisfy new requirements.*

Software engineering was conceived as a means of overcoming the problems involved in building large programs. The application of engineering techniques to the development and maintenance of software systems has improved documentation, reliability and completion time, and has decreased costs. A definition of Software Engineering by Boehm [11] that clarifies the term is:

The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate and maintain them.

This definition introduces the concept of maintenance of software systems because as Lehman [56] pointed out in his first law of program evolution:

A program that is used in a real-world environment necessarily must change or become less and less useful in that environment (the law of continuing change).

The ANSI/IEEE [5] definition of software maintenance is:

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.

There is diversity between the tasks performed in maintenance and the following types of maintenance are usually recognised [36] [90].

Corrective in order to fix a fault in a program, informally known as bug-fixing.

Adaptive due to a change in the environment, e.g. operating system, hardware.

Perfective due to a change in the requirements of the software, usually enhancement of already existing functions but also the incorporation of new ones.

Preventive in order to anticipate problems, redesign for better understanding, performance or maintainability, e.g. rewrite a whole piece of code that is error-prone.

There are several key terms used in software maintenance; a taxonomy by Chikofsky and Cross [20] is as follows.

Forward Engineering is a new term for an old activity. It is the traditional development from high level abstractions to the executable implementation of a system. The adjective 'forward' is used to distinguish the term from reverse engineering.

Reverse Engineering is the process of analyzing a subject system to identify its components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction. It is worth noting that reverse engineering is a process of examination (and possibly recording the results of this examination), not a process of change. Two of the activities usually performed in reverse engineering are *redocumentation* and *design recovery*.

Redocumentation is the creation or revision of a semantically equivalent representation within the same abstraction level. These representations can be, for instance, data flow, data structure or control flow views of the system.

Design Recovery is the identification of meaningful higher level abstractions beyond those obtained directly by examining the system itself.

Inverse Engineering although not mentioned in [20] is a term that is widely used and for some authors equivalent to reverse engineering. If we want to differentiate between them, then we could say that reverse engineering extracts design information and inverse engineering tries to extract functional specifications from the system. Often it is difficult to say whether an activity belongs to one or the other, because the two are closely related.

Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behaviour.

Reengineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. Software reengineering usually includes some form of reverse engineering followed by some form of forward engineering or restructuring. Some authors use the term reconstruction to mean reengineering.

All of the above are definitions of terms widely used in software maintenance. See Figure 1.1 for the relationships between them. Some of these terms describe methods currently used to perform software maintenance.

1.3 ReForm and WSL

This thesis describes the implementation of an incremental static data flow analysis tool in a reverse engineering environment called ReForm for the purpose of extracting slices from programs written in a Wide Spectrum Language (WSL). The ReForm project is concerned with the reengineering of old programs written in low-level languages, usually without the application of software engineering techniques. The aim is to express these programs in terms of a formal specification language such as Z [31]. To accomplish this, the program is first translated to WSL and it then undergoes transformations which have been formally mathematically proven to be correct. These transformations are based on a formal system developed by Ward [95]. By means of this theory it is possible to prove that two versions of a program are equivalent. Wide spectrum languages allow different levels of abstraction to coexist in the same program, so some parts of the program can be represented by low-level machine-oriented constructs, some parts can be expressed as high-level abstractions (e.g. predicate calculus) and some other parts can be expressed in a level of abstraction somewhere in between. The use of WSL allows great flexibility in dealing with the transformations performed on a program and allows the system to cope with programs written in any language, provided a translator is built to translate that language into WSL.

The typical translation process begins with an assembler source code program that has to be inverse engineered. This code is translated to WSL so transformations can be applied to it. When the maintainer is satisfied with the WSL code, the WSL-to-Z translator is invoked and Z-specifications produced. See Figure 1.2 (extracted from [105]) for a graphical view of this translation process.

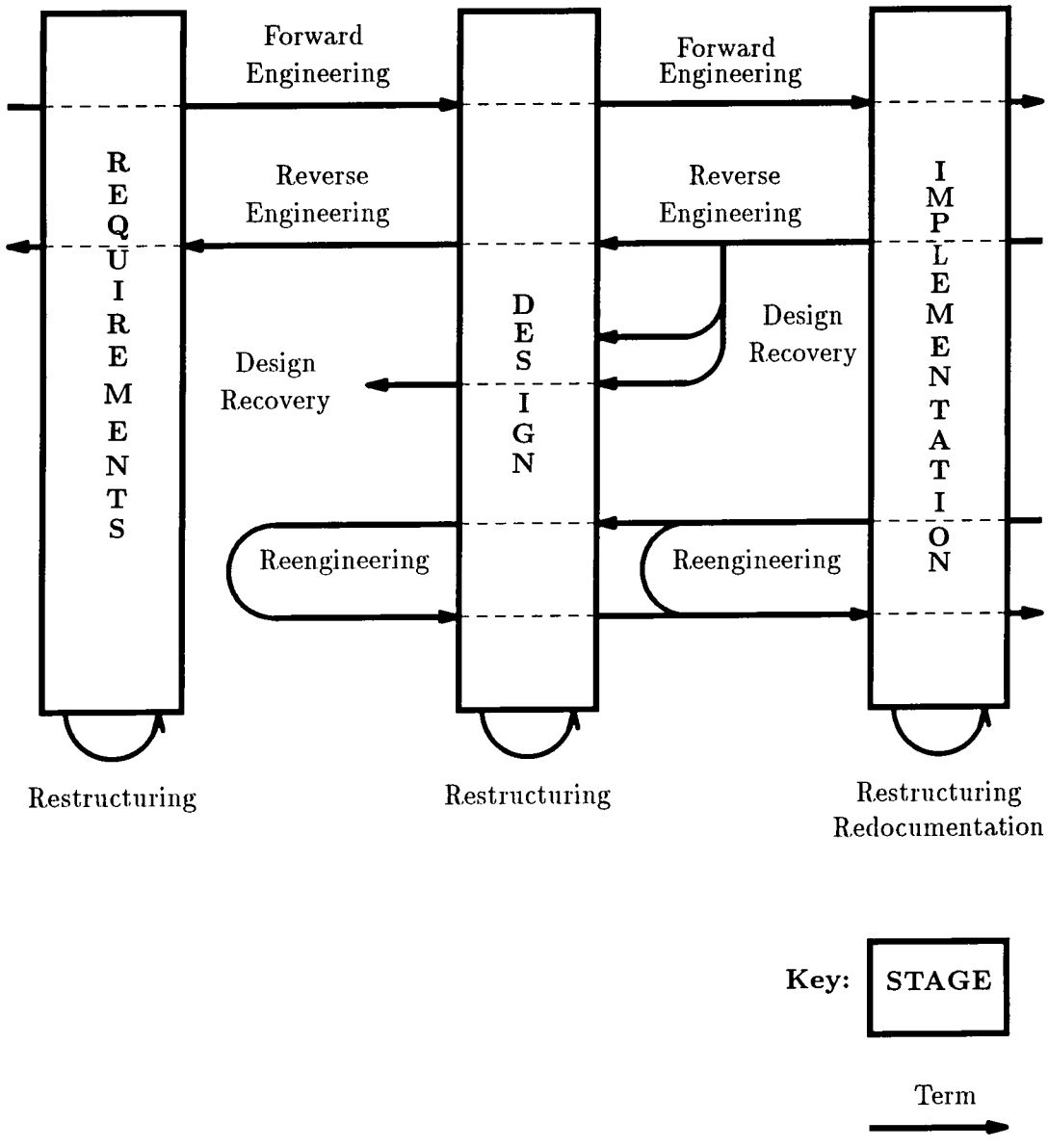

Figure 1.1: Relationships between terms in software maintenance

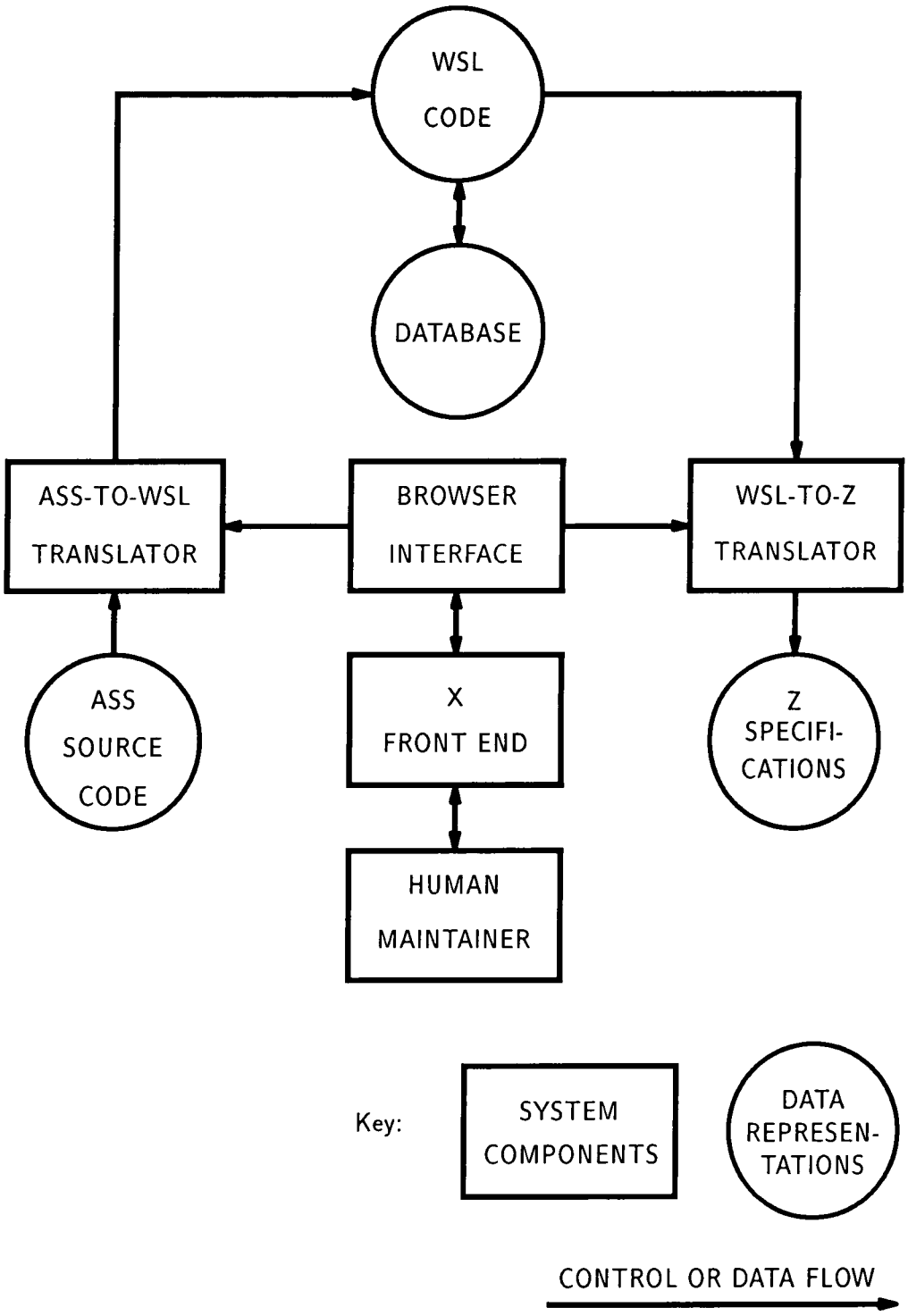

Figure 1.2: Translation process in the ReForm tool

1.4 Slicer

Program slicing [103] is a technique for restricting the behaviour of a program to some specified set of interest. A slice *S(v,n)* of program *P* on variable *v*, or set of variables, at statement *n* yields the portions of the program that contributed to the value of *v* just before statement *n* is executed. *S(v,n)* is called a *slicing criterion.* Slices can be computed automatically on source programs by analyzing data and control flow. A program slice has the added advantage of being an executable program.

Data flow information consists of the relationships between variables, basically assign and reference usages of a variable. Control flow information consists of determining the order in which statements will be performed during the execution of a program. The static analyser described in this thesis to compute the data and control flow information is of the *incremental* type because the program is continually modified by the transformation system, so a fast method of recomputing global data and control flow information is needed. With the incremental analyser we can do this without recalculating the information that has already been calculated for other parts of the program.

1.5 Problems with ReForm

ReForm is concerned with reengineering large programs, but this presents some problems:

- The programs can be very large (thousands or tens of thousands of lines).
- The control flow of a program is difficult to follow if it has some GOTOs or very small routines.
- It is difficult to deduce what the program will do if the operations are performed on memory locations or registers.

These problems are usually solved in ReForm with the application of transformations that restructure the code and convert memory locations to variables, but some of the transformations are not automatic and human interaction is needed in order to choose which transformation has to be applied.

A slicer is useful for ReForm because it assists the maintainer in the comprehension and understanding of a program by decomposing it and showing only the statements relevant to the computation of the value of a variable or a set of variables. We suggest that it may provide some assistance, perhaps aided by a modulariser, in recognising abstract data types and hence helping to cross levels of abstraction in WSL.

Slicers have been used also in:

- understanding the program while doing debugging [102]
- excluding dead code from the program [32]

1.6 Outline of chapters

This chapter has provided an introduction to the thesis and a presentation of the objectives that are intended to achieve. The next two chapters are literature surveys. Chapter 2 surveys three topics, namely software engineering, software maintenance and reverse engineering that put slicing into perspective. Chapter 3 surveys techniques for static analysis, data flow analysis and slicing. The following chapter provides a more precise definition of the problem that we want to solve, i.e. comprehension of large and complex WSL programs. Chapter 5 and 6 explain which method of solution has been chosen (i.e. static analysis and slicing, using techniques devised by Bergeretti and Carré for information-flow and data-flow analysis of while-programs) , why it has been chosen, and the implementation of the solution. In the last two chapters the results and conclusions of applying slicing techniques for the maintenance of programs written in WSL are presented. An appendix with the WSL syntax is included.

Chapter 2

Literature Survey

The purpose of this chapter is to present the context in which program slicing can contribute to software engineering and software maintenance. A survey of current state of research in three topics, namely software engineering, software maintenance and reverse engineering is presented. Models, environments, tools and metrics are described for the first two sections: software engineering and software maintenance. The last section on reverse engineering presents three current projects in this very active area.

2.1 Software Engineering

According to Buxton, there are three main aspects to computing [17]:

Computer science is the face turned to mathematics, from which we seek laws of behaviour for programs.

Computer architecture is the face turned to electronics, from which we build our computers.

Software engineering is the face turned towards the users, whose applications we implement.

In the early days of computing the third aspect did not exist as such and the general division in computing was that between hardware and software, building computers and writing programs for them. In the late 1960s people in computing were concerned about the 'software crisis' because software was late, over budget and unreliable [17]. The NATO Science Committee arranged the first conference on Software Engineering in 1968 to discuss all these problems and try to find a solution by means of engineering techniques.

The reason for using engineering techniques was that programs were becoming bigger and more complex and the usual techniques oriented to the design, development, debugging and maintenance of a small program by a single person were no longer applicable. A change of approach was needed and as pointed out by Sommerville [89]:

> *Software Engineering is concerned with software systems which are built by teams rather than individual programmers, uses engineering principles in the development of these systems, and is made up of both technical and non-technical aspects.*

2.1.1 Software Process Models

In the engineering sciences processes and process models have been used for a long time because they allow standardisation and a way to measure progress in a project. When engineering techniques were introduced in software development, the definition and use of processes and models started.

Before giving a definition of what a *software process* is, the generic notion of *process* by Osterweil [68] is presented:

> *...[A process is] a systematic approach to the creation of a product or the accomplishment of some task.*

The set of instructions that has to be followed to accomplish a task is a *process description* (or a *process model*):

> ... *while a process is a vehicle for doing a job, a process description is a specification of how the job is to be done* [68].

Process descriptions help in improving the implementation of a process because they provide a way to reason about the underlying process:

> ... *processes are hard to comprehend and reason about, while process descriptions as static objects, are far easier to comprehend* [68].

From all these generic definitions is possible to deduce the following definitions

Software engineering process: a systematic approach to the construction of software.

Software engineering process model: the specification of how software is to be constructed.

The first software engineering process model was suggested by Royce [84]. His model was mainly concerned with development activities, and later on operation and maintenance activities were incorporated and further refinements made by other authors. This model is known as the *waterfall* model and it is very useful for project management as it differentiates separate stages in the development process with explicit deliverables. It is still arguably the most popular model today. The following stages are executed linearly in the waterfall model, and most of them are recognised in other models.

Requirements analysis and definition The services that the system has to provide are discussed with the users. Once these are established, they are defined in a way that both users and developers can understand.

System and Software design Using the requirements definition of the preceding stage, a conceptual solution is envisaged. System design consists of deciding which of the requirements will be solved with hardware and which ones with software. Software design consists of deciding which modules, language, functions and data structure will be needed.

Implementation and unit testing The software design is realized by writing the program units in some executable language. Units are tested or verified to accomplish with their specification.

System testing The system is tested as a whole once all the units have been integrated.

Operation and maintenance The system is installed and used. The process of maintenance consists of correcting errors, improving the implementation or enhancing the services the system provides.

In the original waterfall model, the last stage, maintenance, did not exist, and in its later refinements, maintenance was seen as a post-delivery activity that did not need too much attention and relatively unqualified staff were allocated to it. The waterfall model is too simplistic because is very unlikely that once one stage is completed it will not be modified. Flaws and errors are discovered in all the stages, to solve them, modifications are made in the appropriate stage, but these modifications can affect previous stages (i.e. system requirements, design) that should be modified as well.

To overcome these problems, other alternative models have been proposed, and a list follows.

Boehm's model [11]: a refinement of the waterfall model, because modifications have to be done in all the stages. Hence a process of iteration in the development activities is needed, which was not considered in the original waterfall model. The maintenance stage is recognised, considered crucial, and added to the life cycle. Boehm also added risk analysis to his model.

'V' model [67]: it has basically the same phases as the waterfall model, but they are organised in a 'V' shape, relating the outputs produced by each of the stages (see Figure 2.1). Three of the main features of the 'V' model, which are vital for a good management practice, are:

- It focusses on the outputs not the activities occurring in the phase, thus providing a more objective way to assess the results of a stage.

- A testing/validation procedure is performed in each stage.
- The maintenance stage is seen as equivalent to the development cycle, performing all the activities of the previous stages: analysis, design, implementation and testing.

To give a more detailed description of what is involved in each stage of the life cycle, the outputs produced by the phases of this model are presented in Table 2.1.

Exploratory programming: a working system is developed as quickly as possible for the user to examine, it is then *modified* according to new specifications. This is useful when the requirements are not known in full, or when they may change when the user sees the system.

Prototyping: largely as above, but once the requirements are set, a *new*, more efficient and reliable system is implemented with the same specifications.

Formal transformations: a formal specification of the system is developed and transformed by correctness-preserving transformations. The ReForm project uses the same technique but applied in the reverse order to reverse engineer programs.

System assembly from reusable components: systems are built from already existing components, so the development consists only of assembling them.

2.1.2 Software Engineering Environments

Boehm [12] showed that, for a particular experiment he did, the use of a software engineering environment reduced development effort by a figure in the range from 28% to 41%. Several environments have been created with the aim of providing a collection of tools to the software engineer. Although these environments should not be dependent on any model and thus provide more flexibility to the software engineer, this is not usually the case. A definition of software engineering environment is [89]:

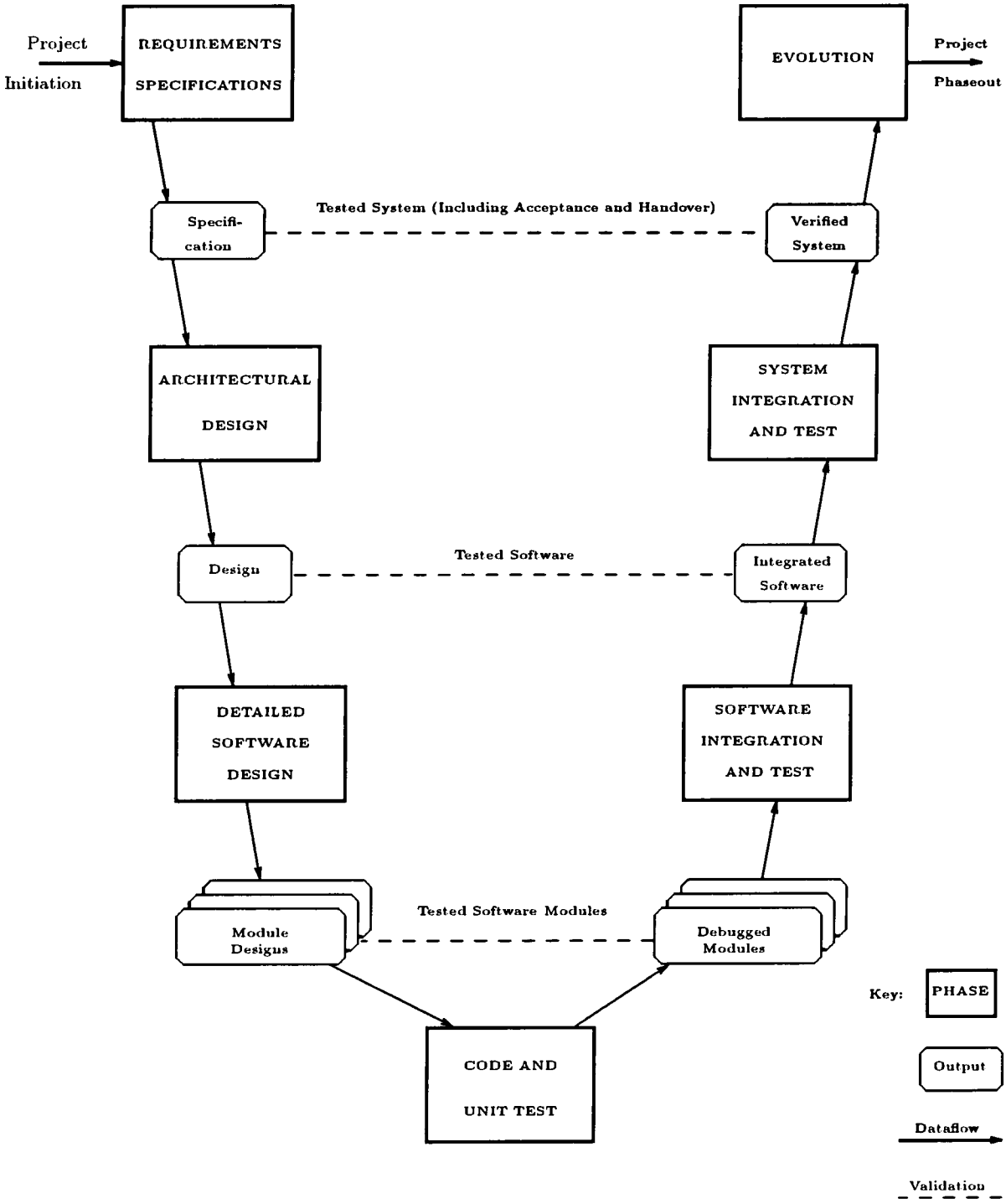

Figure 2.1: Phases and outputs of the 'V' life cycle

- **Initiation**
  - a verified and/or validated system architecture, founded on a design study with basic hardware/software allocations and an approved concept of operation, including the allocation of tasks to people and machines
  - a top-level project plan with milestones, resources, responsibilities, schedules and major activities
  - an outline quality plan with full detail for the requirement specification.
- **Requirements specification**
  - a complete, validated specification of the requirements, both functional and non-functional, which the product must satisfy
  - a detailed project plan
  - a complete quality plan.
- **Architectural design**
  - a complete, verified specification of the overall architecture, control structure and data structure for the product
  - draft user manuals and training plans
  - test plans for integration testing.
- **Detailed software design**
  - a complete, verified specification of the control structure, data structures, interface relations, sizing, key algorithms and assumptions for each software component in the system
  - test plans for each module.
- **Code and unit testing**
  - a complete, verified set of software components
  - a complete set of unit test results
  - complete documentation at the unit level
  - user manuals and training plans.
- **Software integration and test**
  - a fully functioning, validated operational system with program and data conversion, installation and training completed.

  This phase includes acceptance testing and handover to the customer.
- **Evolution and Maintenance**
  - a fully functioning, validated update of the system.

  This sub-goal is repeated for each update, all of which follow the complete development sequence of the above steps.
- **Phaseout**
  - a clean transfer, to its successors (if any), of the functions performed by the product.

Table 2.1: Detailed description of outputs from the phases of the 'V' life cycle.

... all of the automated facilities that the software engineer has available to assist with the task of software development.

To be a proper definition for a software engineering environment, this definition lacks 'and maintenance' in the end, because this is what should differentiate software engineering environments from software development environments. The facilities mentioned in the definition consist typically of tools like diagram editors, data dictionaries, documentation support systems, electronic mail, etc.

The concept of *integration* is used throughout to describe software engineering environments, so we shall define integration as 'The combination of different parts into a whole, with the aim to get something of better qualities than the ones provided by the parts separately'. These qualities can be: ease of use, efficiency, standardisation, etc. The ANSI/IEEE [5] definition of integration is:

The process of combining software elements into an overall system.

This definition is simplistic and does not explain how the integration is achieved. Brown and McDermid [14] have defined the key aspects of integration:

Interface integration Each tool has the same set of constructs at the interface, where there is common functionality.

Process integration The environment supports a single coherent software-development methodology. Tools work together with a common understanding of the software-development life cycle, and each development stage supports a single view of the system's structure.

Tool integration Tools share data via a common data format, which can be defined for a particular purpose.

Team integration The environment fosters group work by ensuring effective communication and information dissemination and by keeping users from corrupting each other's work.

Management integration The environment helps managers control actual development. Management information is derived from the technical information software engineers produce, rather than from fictitious reports.

As Brown and McDermid explain, none of these different aspects alone captures the complete notion of integration, and a proper software engineering environment should provide all these integration levels to meet its basic requirements.

There are four major types of software engineering environments: CASE environments, language independent programming environments, language specific environments and IPSEs.

CASE Environments

CASE—Computer Aided Software Engineering—tools have been available for a long time, e.g. compilers, debuggers, link-editors, etc, but now they usually include diagram editors, report generators, and some other graphic tools. When these tools are integrated into a common environment they are called CASE workbenches. Most of these workbenches are oriented to the analysis and design of the software process and they use graphical notations. The drawback of the first generation of CASE workbenches was that they lacked

- support for the maintenance stage of the software process.
- support for configuration management. Configuration management tools allow different versions of program units to coexist and permit the retrieval of a specific version of the system which might not be the current one.
- tighter integration with the implementation stage, i.e. links with code generators, test data suites, compilers, etc.
- some standards for information interchange between different CASE tools.

For a survey on CASE tools see [47].

Language-independent programming environments

These environments are usually file and character oriented, and they use the operating system to integrate the tools. This integration is not very tight but the tools are simpler to create. A classical example is the UNIX Programmer's Workbench [49]. An unusual language-independent programming environment is PECAN [78] [79]; it uses a new approach called 'Graphical Program Development'. Programs are developed by means of graphics that model the different constructions found in other languages, such as conditional and iterative statements and sequences of statements. The constructions can then be translated to a programming language for implementation purposes.

Language-specific programming environments

These environments are usually aimed at the exploratory programming and prototyping software process models. They are tightly integrated and have windowed interfaces. Examples include environments for LISP such as CEDAR [92] and Interlisp [93] and environments for Smalltalk [37].

IPSEs

The only environments that deserve to be called software engineering environments are the so-called IPSEs—Integrated Project Support Environments. The key words in this term are *integrated* and *project*.

They are called *integrated* because all the tools have to interface with a common database system and sometimes with a common front end, to form part of the environment. Although IPSEs should provide all the integration levels mentioned above, most of them have only tool integration, and is only in this context that the word *integrated* is commonly used.

The term uses the word *project* instead of *development* because IPSEs are designed to sup-

port all the stages of the software process: from initial feasibility studies to operation and maintenance.

Some of the tools found in IPSEs are the same as the ones found in CASE workbenches and in programming environments, but usually with added advantages. The advantages of using IPSEs instead of CASE tools stem from the use of a database system, an object management system and a common user interface. The database system allows a tighter integration of tools than the simpler file system provided by an operating system. The object management system moves a step further, because it allows objects to be named, to exist in different versions and to be related to other objects. The common user interface facilitates the task of the user in obviating the need to learn a different interface for each of the tools used. Examples of IPSEs include [4] [24]. Surveys on software engineering environments include [21] [101].

ICASE vs IPSEs

CASE tools have evolved since their first generation appeared, and now some of these environments are called ICASE —the 'I' standing for integrated. As a result of this evolution the distinctions between CASE tools and IPSEs are becoming less apparent. To help differentiate between them, Brown and McDermid [14] recognise these three differences:

- IPSEs are intended to support multiple methods and be open, while CASE tools support single methods and are not readily extensible.
- IPSEs are aimed at group work, while CASE tools are designed primarily for individual support.
- IPSEs have been used primarily in scientific and engineering applications, while CASE tools have been developed in more mainstream data processing applications.

2.1.3 Metrics - Maintainability

In all of the engineering sciences, metrics are developed and tested to measure all sorts of aspects. In software engineering the same has happened, but the measures are difficult to validate and standardise due to the intrinsic nature of software. Early software metrics were proposed by Halstead [39] and McCabe [63]. More recently, new experiments have been conducted to improve these old metrics or to create new ones. For instance, Felician and Zalateu [26] have proposed a correction of the length metric by Halstead for long programs in Pascal; Ramamoorthy and Melton [76] have proposed an hybrid metric from Halstead's and McCabe's metrics; and Mehndiratta and Grover [64] have compared the applicability of several metrics to different languages and have proposed a new set of metrics. The current trend in metrics seems to be to associate measures with the nodes of the abstract syntax tree, examples include the books by Ejiogu [25] and Fenton [27]. For an introduction to and a survey on the theory of software measurement, see Fenton [27] and Ince [48].

The metrics we are interested in are *maintainability metrics* that try to predict how easy a program can be maintained [41]. One of the classical measures for maintainability is to provide a value that reflects how well structured a software system is. Rombach [82] conducted an experiment to determine the influence of software structure on maintainability. His conclusions are that, in general, the use of more structured languages has a positive effect on maintainability.

In a recent article by Schneidewind [88], a method for testing the validity of software metrics is described. His 'comprehensive metrics validation methodology' consists of six validity criteria: associativity, consistency, discriminative power, tracking, predictability and repeatability. This approach is interesting because is the first time that a system for validating metrics is proposed and illustrated with case studies.

2.2 Software Maintenance

Although, as mentioned before, software maintenance often represents more than 50% of the total cost of software, management has been traditionally unconcerned by this problem. This tendency is changing according to the conclusions of the survey by Lientz and Swanson [59]:

- Maintenance and enhancement consume much of the total resources of systems and programming groups.
- Maintenance and enhancement tend to be viewed by management as at least somewhat more important than new application software development.
- In maintenance and enhancement, problems of a management orientation tend to be more significant than those of a technical orientation.
- User demands for enhancements and extension constitute the most important management problem area.

The cost of software maintenance

Improvements in the software maintenance process have been traditionally measured in terms of the ratio between the funds dedicated to maintenance and development. This measure has been used because, as explained by Foster [30]:

- it can be captured as an *instant* measure in a single survey, without the complications of recording data over a long period before a measurement can be called complete.
- the ratio has been reported by many authorities, and in several surveys. An organisation that measures it only once thus has an immediate sources of comparative data available.

This ratio is believed to be increasing over time, see [11] [12] for the prediction of its increase in the S-curve. It is intuitively appealing to think that improvements in the maintenance process should, other things being equal, reduce or at least control the value of this ratio.

The factors that determine the value of the development/maintenance ratio are [30]:

- the relative demands for software functionality delivered via the development or maintenance processes.
- the productivity of the maintenance process expressed in terms of functionality delivered per unit cost.
- the productivity of the development process, expressed in the same terms.
- the lifetime of the average program.

These factors explain why the maintenance costs compared to the development ones are always increasing:

- More software functionality is being delivered via the maintenance process. Surveys by Ben Arfa et al [8] and by Lientz and Swanson [59] [60] always show that at least 50% of the effort in maintenance is dedicated to perfective activities.
- The productivity of the development process is higher than the one of the maintenance process because new software technologies affect developers sooner than it does maintainers, who have to wait until the software developed with these new technologies is to be maintained to reap the benefits of them.
- If more functionality is delivered by the maintenance process, the lifetime of the average software system is increased and hence total maintenance costs will increase without a corresponding rise in development costs.

Foster [30] argues that the common *intuitive* view of "Improvements should reduce the spending in software maintenance" has to be revised and that the lifetime of a program could be a more adequate measure of the improvements in software maintenance.

Even if the factors mentioned above can explain the increase in software maintenance costs, it is not clear that the *apocalyptic prophecies* of Boehm [11] and others will become true. Recent surveys [8] [65] show that software maintenance costs may actually be decreasing.

Effort distribution

According to two surveys by Lientz and Swanson [59] [60], the distribution of the effort dedicated to the different tasks performed in maintenance (see Section 1.2) is as follows.

- Perfective 50%–60%
- Adaptive 18%–25%
- Corrective 17%–21%
- Preventive 4%–5%

2.2.1 Models for Software Maintenance

The usefulness of a model has been already been presented in Section 2.1.1. A model for the software maintenance process will be even more useful because maintenance seems to be more affected by management issues than development. Several models for maintenance have been created, but not all of them contemplate the four types of maintenance. A description of five models (alphabetically sorted) that take into account all types of maintenance activities follows.

Boehm model

According to Boehm [11] maintenance can be decomposed into three phases.

Understanding Good documentation and traceability between requirements and code are needed, with well structured and well formatted code.

Modification Software and hardware and data structures should be easy to expand and should minimise the side effects of changes; easy-to-update documentation is needed.

Revalidation Software structures should facilitate selective retesting, and aids for making retesting more thorough and efficient are needed.

This is not a detailed decomposition but is generally accepted as a first step requiring further refinement. Boehm does not explain how to proceed if the requirements needed are not present.

Chapin Model

In the Chapin model [19] phases are named steps and a more detailed view is presented. This model is similar to the development cycle in sharing the middle steps of analysis, design, implementation and test.

Understand existing system Personnel review any existing documentation and access relevant materials and personnel who may possess relevant knowledge.

Define the objectives for the modifications The maintainer seeks to clarify the aspirations of the user in requesting the change to the program.

Analyse the requirements The consequences of exploring alternate paths in satisfying the maintenance request are considered and evaluated with an accompanying cost-benefit analysis.

Specify modifications to be made A summary of the analysis results from the previous step produces a specification for the proposed modification.

Design modifications

Program modifications

Code and compile

Debug and test The testing aims to prove that the appropriate change has been correctly implemented.

Revalidate This attempts to confirm the stability of the system, i.e. no ripple effects are observed.

Train users prior to release of new software As soon as the specification step is completed the users are trained to use the modified system to gain familiarity prior to its release.

Convert from previous version of software and release

Document

Quality assurance review

The last two steps are performed concurrently with the other steps and provide the basis for inspections, walkthroughs and technical and management reviews. Some iteration is expected in the middle steps as a result of testing and revalidation.

Martin and McClure model

This model [61] is based on Boehm's model, but much more detail is provided for each stage.

- **Understanding**
  - *Top-down comprehension*
    * Become familiar with the overall program purpose and the overall flow of control.
    * Identify the basic program structures as well as the processing components.
    * If the program is part of a larger system, then delineate its role.

* Identify what each component does and how this is implemented in the code.

- *Improvement of documentation*

  As understanding of the program is gained, document it in a high level fashion. This makes future maintenance easier.

- *Participation in program development*

  The maintainer-to-be should take part in the development of the program.

- **Modification**

  - *Design the change and debug*

    If the change is an error then this is rectified by changing the program logic. If the change is an enhancement then new logic is developed and incorporated into the program.

    The design of the new logic is top-down:

    * Review entire program at general level by studying modules, their interface and the database.
    * Then isolate the modules and the data structures which are to be changed and those modules and data structures which are to be affected by the change.
    * Detailed study of module and data structures, design change, specifying new logic and changes (if any) to existing logic.

  - *Alter code*

    Changes should be implemented as simply as possible, exercising caution and preserving coding style.

  - *Minimise side effects*

    * Search all modules which share global variables or routines with the changed module.
    * When multiple changes are envisaged the changes should be grouped by module. The sequence of changes should follow a top-down approach, changing the main driver first, then its direct descendants and so on.

* Change one module at a time, determining potential ripple effects, before changing the next module in the sequence.

- **Revalidation**

  Revalidation is necessary to ensure that the modifications carried out to the program have not adversely affected the program. Revalidation is achieved by carrying out testing, each type of testing having its own particular goal.

  - *System testing*

    Does the program work as before ?

  - *Regression testing*

    Have the changes affected how the rest of the program works ?

  - *Change testing*

    Have the changes been designed and implemented correctly ?

Patkau model

Five basic maintenance tasks are identified in a high level manner in the Patkau model [72].

- Identification and specification of the maintenance requirements.
- Diagnose and change location.
- Design of the modification.
- Implementation of the modification.
- Validation of the new system.

The originality of this model is that the first three tasks differ according to the type of maintenance that has to be performed. The types of maintenance recognised are corrective, perfective (termed 'enhancement' by Patkau), adaptive and preventive (actually termed

'perfective' by Patkau, but preventive according to the definitions used in this thesis). The last two tasks are equivalent to the last two phases of the Martin and McClure model.

- **Corrective**
  - Identify repeatable error symptoms and specify the correct operation of the system, a test system and test data are needed.
  - Locate the part of the system responsible for the error.
  - Design the desired properties of the system, after deciding what they should be. Determine the side effects of the changes in these properties.
- **Perfective**
  - Identify new or altered requirements and specification of the operation of the enhanced system.
  - Locate the existing elements affected by the enhancements.
  - Design is split into the following sub-tasks.
    * Assess how new requirements could be met by modifying existing components.
    * Decide what new components are required.
    * Develop the specifications of the new components and/or revise the specification of existing components.
    * Examine the side effects of the addition of new components and/or the revised specifications.
    * Design new components and/or re-design existing components.
- **Adaptive**
  - Identify the type of change in the processing or data environment, describe the change and revise all specifications to reflect the change.
  - Locate all software elements affected by the change. When there is a change in the data environment locate the parts of the system which use or set the data being changed. Use a data dictionary to store the system inputs and outputs, where they are used and their properties.

- Design can be accomplished by employing techniques used for corrections or enhancements in the processing environment.

- **Preventive**
  - Identify a deficiency in the performance, quality, standards, maintainability, specify the change in performance or quality standards.
  - Locate the sources of the deficiencies.
  - Design entails some re-design of a portion of the software such that it still satisfies the original requirements, but the new software either:
    * Uses less resources.
    * Is better structured.
    * Is more maintainable.

Yau model

This model [106] [107] represents information about the development and maintenance of software systems, emphasising relationships between different phases of the software life cycle, and provides the basis for automated tools to assist maintenance personnel in making changes to existing software systems. The following phases are recognised.

- *Determining the maintenance objectives*

  Common objectives are:
  - Correction of program errors.
  - Addition of new capabilities.
  - Deletion of obsolete features.
  - Optimisation.

- *Understanding the program*

  The ease of understanding is affected by

- Complexity which measures the effort required to understand the program.
- Documentation.
- Self-descriptiveness that measures how clear is to read, understand and use the program.

- *Generating maintenance proposals*

  The proposed alterations to the system are affected by the extensibility of the program. Extensibility measures how well the program supports extensions of critical functions.

- *Accounting for ripple effect*

  This is affected by the stability of the program which Yau defines as 'The resistance to the amplification of changes in the program'.

- *Testing*

  This is affected by the testability of the program which is defined as 'The effort required to adequately test the program according to some well defined testing criterion'. If the testing of the program is not succesful then the maintenance process is performed iteratively.

Evaluation of the models

The model by Boehm is not really a model because it needs further refinement, but it provides a general view of the three main stages usually accepted in maintenance.

The Chapin model is more a sequence of steps than a proper model. His idea was to enhance the development model with activities to be performed before and after the development steps. This view was shared by Glass and Noiseux [36] but it looks simplistic nowadays.

Martin and McClure refined Boehm's model and provided a lot of detail on what has to be done to perform good maintenance and how to do it. This is a very complete model and is probably one of the best 'traditional' maintenance models.

Patkau's model differentiates between types of maintenance. Although other models take this into account, Patkau makes the differences explicit. Patkau uses a non-standard terminology to classify types of maintenance, this has been mentioned before while introducing his model. This model is described with great detail and is arguably (along with the Martin and McClure model) one of the most used in maintenance.

Yau model was proposed after a series of stability measures and reflects the importance given by the authors to the ripple effect (where most of the model is based).

2.2.2 Laws of Software Evolution

A more theoretical approach to software maintenance is done by Lehman and Belady [56] [57]. Their theory of *program evolution dynamics* is based on the study of system change, i.e. how software systems evolve during their life. After observing a number of software systems, they derived these empirical laws:

Law of continuing change A program that is used in a real-world environment necessarily must change or become less and less useful in that environment.

Law of increasing complexity As an evolving program changes, its structure becomes more complex unless active efforts are made to avoid this phenomenon.

Law of large program evolution Program evolution is a self-regulating process and measurement of system attributes such as size, time between releases, number of reported errors, etc, reveals statistically significant trends and invariances.

Law of organizational stability Over the lifetime of a program, the rate of development of that program is approximately constant and independent of the resources devoted to system development.

Law of conservation of familiarity Over the lifetime of a system, the incremental system change in each release is approximately constant.

An explanation of what the laws mean follows. The first law says, basically, that maintenance is unavoidable. The second implies that the structure of a program is degraded when maintaining and needs restructuring. The third is a controversial law, it proposes that the dynamic of a program is established during the early development phases, and it is difficult to change it during the maintenance. The fourth says that large development teams do not increase productivity because of communication overhead. The fifth law deals with configuration management. If a large number of changes are made to one release, a new release follows quickly to repair faults in the changes. This is a self-regulating process.

2.2.3 Program Comprehension

As seen before the first stages of the maintenance cycle are crucial, and the very first one is understanding the system as it is. This can be done by careful examination of all the documentation available or by asking personnel who may have worked on the development of the system. Sometimes neither of them is accessible or trustworthy and more *direct* methods have to be used.

These direct methods imply the use of source code as the basis for the comprehension of the software system. A description of three methods, from low level to high level, that use the code to gain knowledge about a software system follows:

Code reading The crudest method, effective for small programs, but difficult to use when the size and complexity of the program grow. The readability of a program is affected by the design method (i.e. top-down, data structured, data flow, object oriented, abstract data types) and the style (i.e. indentation, comments, meaningful identifiers) in which is written.

Program analysis This is automated code reading; it has an advantage over human code reading when maintaining a large system and/or when the maintainers are not the developers. The techniques used are divided between static and dynamic analysis. Static analysis is the analysis of a program without its execution, and dynamic analysis

is the analysis of a program while it is being executed. A detailed list of different tools, including slicing, for program analysis is presented in Section 2.2.4. For a survey on Program comprehension see [9].

Inverse or Reverse Engineering In Section 1.2 definitions for these terms have been provided. Tools that belong to this category are high-level aids because they try to recover the original design from the source code. Although this is impossible because information is lost in the process of translation from design to code, these methods do their best to present the maintainer with a more abstract view of the system. A study of three methods for reverse engineering, namely ReForm, REDO and MACS is presented in Section 2.3.

2.2.4 Tools for Maintenance

To implement the models for software maintenance or to help in program comprehension several tools have been produced. These tools can come as separate entities or can be integrated in an environment. A brief description of tools useful for maintenance follows.

pretty-printers to display a code listing in an improved form. Provides indentation, highlights comments and the structure of blocks in the program. For an example see [85].

version comparators to display the differences between two different versions of the same program.

diagram generators to display several kinds of information, like call-graph, data dependency, flow-chart, etc.

language converters to perform translations between languages of the same relative level of abstraction. For an example see [66].

restructurers or code-to-code transformators to enhance or provide structure to a program, e.g. removing GOTOs, transforming nested IFs into CASE statements.

testing utilities to automate or facilitate the testing task, typical tools include tracers, test data generators, which provide large number of test inputs, and simulators, which imitate the actions of another program or more commonly a hardware device not available or one that could be damaged by faulty software.

configuration management utilities to help in the management of system change. A configuration database is used to record information about change requests and system changes.

version controllers to draw up an identification scheme for different versions of a system, and ensure that this scheme is used when creating new system versions. For an example see SCCS [81].

debuggers or dynamic analysers to analyse a program while it is being executed with some data given by the maintainer and to present him/her with the results.

static analysers to analyse a program without executing it; the information is always potential information as opposed to actual information. This type of analyser is used in some tools like cross-reference generators, ripple effect analysers, data flow and control flow analysers. Some analysers are called *incremental* analysers because they do not have to re-analyse the entire system in response to a change, they use information already stored.

ripple effect analysers to reflect how the modification of the value of a variable can affect the values of other variables. These tools are commonly used in regression testing.

data flow analysers to reflect how the variables in a program are related to each other, basically assign and reference usages of the variable. Such analysis techniques are well known in compiler optimization.

slicers to decompose a program according to one variable or a set of variables. They use information produced by the data flow analyser. The statements that form the slice are those ones relevant to the computation of the (final) values of the variables.

See Chapter 3 for a more detailed presentation of static analysers, data flow analysis and slicing.

2.2.5 Software Maintenance Environments

Some software engineering environments have already been presented in Section 2.1.2. Some of them already incorporate support for maintenance. The environments presented here are more software maintenance oriented, though all of them only implement program comprehension (the first stage of software maintenance) and they do not bother about management or version control. Software maintenance environments are very limited and they do not implement any particular model. They are closer to 'enhanced tools' than to 'true environments' because they usually are closed systems, not supporting the incorporation of other tools.

VIFOR

VIFOR [74]—Visual Interactive FORtran— is a language-specific programming environment. In VIFOR programs are represented in two forms: as a text (source code) or as a graph consisting of icons and lines between icons. The data model of VIFOR contains four different classes of entities and three relations.

The entity classes are:

- *modules* i.e. files of source code.
- *declarations* which are divided into:
  - *processes* i.e. main program, subroutines and functions.
  - *commons* i.e. global data elements.

The relations are:

- The *belong to* relation that specifies whether any entities are parts of another entity. In particular declarations belong to modules.
- The *call* relation interconnects processes. The processes and their call relations constitute a *call-graph*.
- The *reference* relation interconnects processes and commons. They define which processes have access to which commons.

The relations stored in the VIFOR database help the programmer to understand the code and to follow the ripple effects of the modifications.

The same authors that developed VIFOR have created VIPEG [75]—Visual Interactive Programming Environment Generator— which provides a framework for creating environments which work in other languages with functionalities similar to VIFOR.

Surgeon's Assistant

This tool [33] was constructed to validate decomposition slicing as a maintenance technique; see Section 3.3.3 for an explanation of the decomposition technique. Surgeon's Assistant works with C language code and its interface uses the Sunview windows environment.

In a typical session the user loads a program and selects decomposition variables. The tool then slices the program and the maintainer is presented with a window containing the original program with independent statements (which are needed only for the computation of the slice) in normal video and dependent statements (which are necessary to the computation of the complement of the slice) in reverse video. The maintainer can modify the program, but only the statements which are independent, dependent statements are not modifiable. Once the editing is finished, a file containing the modified slice is saved, compiled and tested. When the modified code is accepted, surgeon's assistant merges it back to the complement, verifying that any added control flow does not control any dependent statements. this means that changes can be done without affecting the complement and hence avoiding regression

testing.

Surgeon's Assistant was also implemented as an evaluation test for a new software maintenance process model. In this model the revalidation phase of the classical models is not needed because of the use of decomposition slices. Changes are tested in the decomposition slice and cannot ripple out into the complement, making regression testing and revalidation of the whole program unnecessary. There is still some testing performed, but this is done on the decomposition slice where no side effect changes are allowed.

SPADE

SPADE —Southampton Program Analysis and Development Environment— is based around a functional description language —FDL. Program analysis and verification tools can be applied to a program written in FDL. Translators to FDL have been developed, from Pascal and from M6800 assembly code.

2.2.6 Metrics for Maintenance

Some metrics for software maintenance have been proposed by Leach [55] and by Yau and Collofello [106]. Kafura and Reddy [51] performed a study of different complexity metrics on a medium-sized software system, and their conclusions were:

- The metrics were able to determine improper integration of enhancements to the system.
- The metrics agreed with the subjective evaluations of the system by people familiar with it.
- The growth in system complexity agreed with the character of the tasks.

- The metrics demonstrated their usefulness by revealing poorly structured components of the design of a new version.

Schneidewind [87] describes how two of the criteria —discriminative power and tracking— from his metric validation method can be used in maintenance to

- establish quality control objectives.
- prioritize software components and allocate resources to maintain them.

Essentially, what is proposed is that these two criteria ought to be applied to validate any metrics used during the maintenance of software. The results should be equivalent as when the validation was applied to the metrics during the development of software.

2.3 Reverse Engineering

In Section 1.2 a definition of reverse engineering was given and the importance of reverse engineering in program comprehension has been outlined in Section 2.2.3. Reverse engineering tools help the maintainer to understand the system that is going to be maintained in the following ways:

- by extracting design documentation, useful when migrating systems between environments.
- by bringing old systems into a more modern structured method of maintenance.

Three reverse engineering projects are described in the next sections. ReForm is an inverse engineering project because the aim is to extract specification requirements while MACS and REDO are 'traditional' reverse engineering projects. MACS assists the maintainer using expert system technology, whereas REDO provides an environment where to integrate several maintenance tools.

2.3.1 ReForm

ReForm and WSL were introduced in Section 1.3. The theoretical basis for ReForm is a theory of program refinement and equivalence by Martin Ward [95] [96]. This theory has been used to create a set of mathematical program transformations which can derive a specification from a segment of code or can transform a segment of code into a logically equivalent form. A tool named Maintainer's Assistant [98] [105] was created to implement these transformations.

WSL

WSL is the internal language used in the Maintainer's Assistant, and this new language was created because:

- a language with very simple semantics was needed to simplify equivalence proofs and no current language was designed with such semantics in mind.
- a language that could express both low-level operations and high-level specifications was needed. Programming languages cannot express non executable specifications and specification languages would require important changes to express low-level operations.
- a common language allows systems that have modules written in different languages to be expressed in only one way. If programs written in a new language are to be maintained, only a new translator has to be built.

For a brief description of the syntax of WSL see Appendix A.

Browser interface

The tools provided to the maintainer using the Maintainer's Assistant are invoked from the *Browser Interface* which maintains the coherence between what the program is becoming because of the transformations applied to it, and what appears on the screen which is a pretty-printed Pascal-like version of the program.

The maintainer can manipulate the source code in three ways:

- by directly modifying the source in WSL form using *editing commands*.
- by selecting a particular transformation from the *library of transformations*.
- by requesting the *knowledge-base system* to search for a sequence of transformations that will achieve a given effect.

These three methods are described below. See Figure 2.2 for a graphical view of the architecture of the Maintainer's Assistant.

Structure editor

This is a syntax-based editor and is the only way the maintainer can directly manipulate the source code in WSL. Because this can change the effects of the program, the editor records these editing actions for future reference.

Transformation library

The maintainer selects a section of code or a single statement and sends a select transformation command which is received by the library that tests the applicability conditions of the transformation to these particular section of code. If the conditions hold, the transformation is performed by sending a sequence of editing commands to the structure editor.

Knowledge base

This will work (is not implemented yet) in the following way: the maintainer will send a request for changing a section of code and the knowledge base will select an appropriate set of transformations to apply to the code, then it will send select transformation commands to the transformation library, which will in turn send editing commands to the editor.

2.3.2 REDO

This project [94] is concerned with the development of techniques and tools for the improved maintenance of software. The aim of REDO —REngineering, validation and DOcumentation of systems— is to provide a coherent integrated toolkit for software maintenance on a single representation of an application. The REDO model is formed by three activities: triggering, assessment and decision. The triggering activity determines when an assessment of an already existing software system has to be undertaken. The assessment decides what is the best thing to do with the software. If the decision is to reconstruct the system, REDO provides the method, which consists of bringing the software system to one or more standard representations:

- original source code in the system database.
- its translation to a common intermediate language called UNIFORM.
- its transformation to a graph representation.
- the representation of the documentation.

Trigger activity

This can be anything, common examples include:

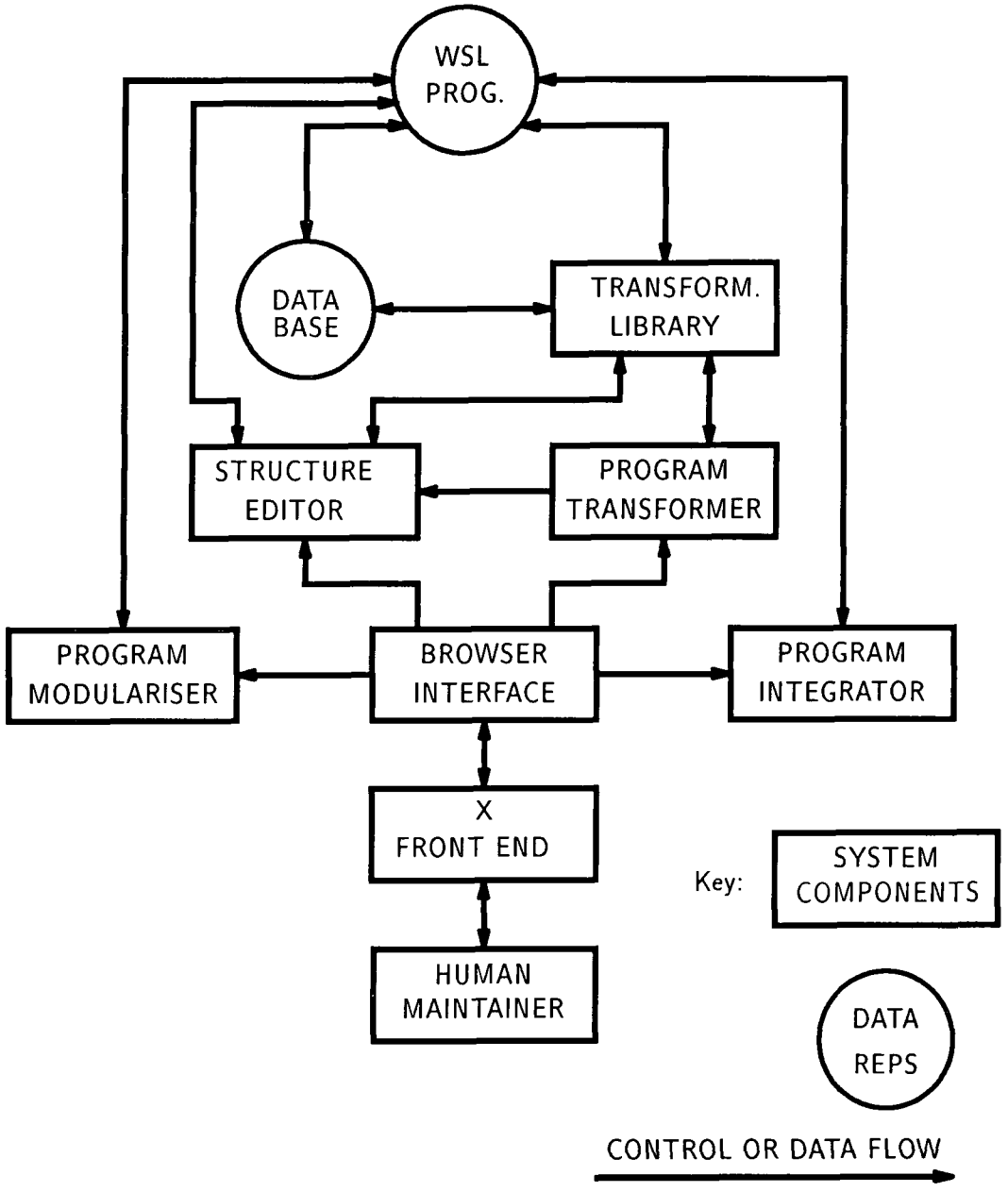

Figure 2.2: Architecture of the Maintainer's Assistant

- request for a change so large that makes it infeasible on the actual system.
- management wants some statistics about the system.
- reimplementation of the system in another computer.

Assessment activity

The subactivities are:

- identification of the software system.
- decision of the assessment criteria.
- assessment based on the above criteria.

Decision activity

Decisions taken on the software include:

- replace with vendor package.
- rebuild from scratch.
- discard software.
- reconstruct using REDO, making it more maintainable.

2.3.3 MACS

The aim of the MACS —Maintenance Assistance Capability for Software— project [23] [35] is to provide an expert system toolset to assist the maintainer, but it is not an expert

system as such. MACS, as in REDO, has reengineering capabilities: recover design specifications, create functional specifications of the software, reimplement application using modern software engineering techniques. MACS uses a graphical representation of the structure of both the code and data in the system, called dimensional design. The MACS architecture comprises four layers:

Object layer where all the objects are stored, is implemented using Eclipse, an OODBS (object oriented data base system).

Toolbox layer containing several tools:

CMS configuration management system

ABR abstraction recovery support

DSG design state graph

RW reasoning world representation

Context layer that holds knowledge about different kinds of maintenance, about maintenance tools, and about the context in which the maintenance is being done.

Domain and method layer based on the context layer, tools can identify a method layer best suited to the application domain and language used.

2.3.4 Evaluation of MACS and REDO

It is very difficult to make any criticism of these research projects because they are still prototypes and there is no practical experience. MACS looks more like a CASE tool for maintenance because it has a graphical environment (based around the design state graph) and a toolbox approach. While it is more difficult to incorporate external tools in MACS, REDO is more flexible and adaptable to external tools. Although it is not fully populated with internal tools yet, REDO should be able to work using external tools. The main contribution of REDO at this stage is that provides a well defined method; the one of MACS

is the design state graph that allows the representation of a wide range of entities during maintenance.

2.4 Summary

In this chapter the context in which slicing takes place has been reviewed. Firstly software engineering was surveyed; which techniques and models have been created and which environments and metrics are in use. The importance of software maintenance was revealed and the same topics surveyed. Finally the assistance that reverse engineering can provide in a maintenance environment was highlighted and three projects presented. A particular interest was focused on WSL and ReForm.

The rationale for slicing has now been established and attention is turned to the detailed technical approaches used in slicing. Slicing is a technique that needs some dataflow information to decompose a program. This information is calculated by a data flow analyser which is a tool widely used in static analysis. Static analysis is a technique that provides very useful information for the maintainer as it reveals the underlying structure of the code and the use of variables by the program. In almost all the environments and methods of maintenance and reverse engineering static analysers are used. The focus of research in this thesis is a slicer for WSL, the need for this language in the ReForm tool has been explained in Section 2.3.1. One of the low-level constructions in the syntax of WSL is the *action system,* which is used to model sections of code with GOTO's and labels. As a result the original data flow analysis algorithm in which this thesis is based, had to be enhanced to include this construction. This is one of the major achievements in the thesis.

Chapter 3

Slicing

In this chapter a survey of the current trends in two very active areas is presented. The topics covered are *data flow analysis* and *slicing*. A brief introduction to *static analysis* is presented first.

3.1 Static Analysis

Static analysers were introduced in Section 2.2.4 as useful tools for maintenance. They have been used as part of the verification process of a program, complementing the syntax-checking functions of compilers; for an example see Rosen [83].

The faults and anomalies a static analyser can check include [89]:

- Unreachable code
- Unconditional branches into loops
- Undeclared variables

- Parameter type mismatches
- Parameter number mismatches
- Uncalled functions and procedures
- Variables used before initialization
- Non-usage of function results
- Possible array bound violations
- Misuse of pointers

Although the anomalies listed above can be detected in recent compilers, it was unusual for a FORTRAN or C compiler to check on these anomalies. Static analysers for FORTRAN are DAVE [70] and FACES [77] while the 'standard' static analyser for C is LINT [22] [80].

Wilde et al [104] use the concept of *program entities* and *program dependencies* to define a dependency graph that helps understanding the relationships between software. Static analysers are used in software maintenance to analyse program dependencies between program entities. Program entities are divided into:

Program modules: procedures, functions, complete programs, etc.

Data objects: variables, data types, files, data structures, etc.

These program dependencies have been classified by Wilde as follows:

Definition dependencies where one program entity is used to define another. *Type dependencies*] where one data type is used to define another belong to these category.

Calling dependencies where one program module calls another. This is typically a procedure or function call.

Functional dependencies between program modules and data objects created or updated by the module.

Data Flow dependencies between data objects where the value held by one object may be used to calculate or set the value of another.

The focus in this chapter will be on data flow dependencies and their application to software maintenance and reverse engineering.

3.2 Data Flow Analysis

According to Aho et al [2], a definition of data flow analysis is:

> *Given a control flow structure, data flow analysis is the process of collecting information about the flow of data throughout the corresponding code segment.*

Data flow analysis is the gathering of information on uses and definitions of variables and the transmission of this information to where it can be of use.

Data flow analysis was originally conceived as a method for performing optimisations in compilers, and its evolution was confined to the construction of optimising compilers for a long time; see Aho et al [2] or Fischer and LeBlanc [29]. Some of the analysis techniques developed in compiler writing have been used as the foundation for data flow analysis methods in software maintenance. One of the first systematic studies on data flow analysis was done by Allen and Cocke [3]; a survey on this topic can be found in Kennedy [53]. There is currently increasing interest in extending the data flow methods to enable them to detect a much wider class of errors; see Osterweil [69].

There are three (at least) orthogonal classifications of data flow analysis methods:

- According to the use they make of the control flow graph, methods can be classified into *iterative analysis* and *interval analysis* (also termed *elimination analysis*).
- Another classification divides analysis methods into *incremental* and *exhaustive* depending on what they do after a modification of the program.
- Finally they can be divided into *interprocedural* and *intraprocedural* (also termed *global*) whether they use information of other procedures in the analysis of the current one.

Iterative analysis

This method consists of traversing the nodes in the control flow graph of a program, propagating the data flow information as the nodes are visited. This procedure *iterates* until the data flow information identified with each node does not change. One of the first iterative methods was devised by Hecht and Ullman [40]. More recent methods include Pollock and Soffa [73] and Keables et al [52].

Interval analysis

This analysis takes place in two steps: the *elimination phase* and the *propagation phase*; and defines *intervals* as subgraphs of the control flow graph of a program. The elimination phase consists of combining these intervals (and their data flow information), a series of increasingly simpler flow graphs results in the collection of all of the data flow relations for the program. The propagation phase propagates the information back to the initial intervals. There is a subclassification of methods according to which type of intervals are used to perform the analysis. One of the methods that uses Allen-Cocke intervals [3] is Ryder [86]. Tarjan intervals [91] are used in Burke [15].

High-level analysis

A different approach to data flow analysis is presented in Rosen [83]; in his method the control flow graph is not used, instead the syntax parse tree of the program is traversed

to collect the data flow information. Another method that uses a similar technique is by Bergeretti and Carré [10].

Exhaustive

If there is a change in a part of a program, all the data flow information for the whole program has to be recalculated from scratch again. This has been the traditional approach because of the compiler culture, where all the program is usually recompiled regardless of the magnitude of the changes. Recently, compilers teamed with syntax-directed editors, are implementing an incremental method of updating the data flow information.

Incremental

Incremental methods appeared as a criticism to the exhaustive method: why does all the information have to be recalculated if the change is confined to a part of the program? In the incremental methods, when a segment of the program is changed, the data flow information is updated with the information of the change. Not all the different types of changes are supported in most of the algorithms. The structural changes, involving change of control flow, are the most difficult to implement. Most of the methods cited above are incremental methods to some extent. For a survey on incremental algorithms see Burke and Ryder [16].

Intraprocedural

The data flow information is calculated for only one procedure or function at a time. When a call to a procedure appears, it is assumed that the procedure can modify and/or use any (global) variable. This results in a very conservative approach.

Interprocedural

The analysis takes into account the data flow information of the procedures. For each procedure *summary information* is calculated and exported. This information usually consists of which variables may be modified, used, and preserved. The summary information is then used at the point of call of a procedure, and the usual intraprocedural analysis can be used to

calculate the data flow information of the program. The classical interprocedural algorithm is by Barth [7].

3.3 Slicing

A definition of slice has already been given in Section 1.4. Slicing is a source to source transformation of a program [6], and is a useful technique for restricting the behaviour of a program to some specified set of interest. This set of interest is usually a variable or a collection of variables. Slices can be computed automatically on source programs by analyzing data and control flow.

3.3.1 Motivation

Weiser [102] conducted an experiment to determine if slicing was useful in debugging. The results obtained were evidence that programmers use slices when debugging, and they do not necessarily look at the programs in a textual or modular way:

> *... debugging programmers, working backwards from the variables and the statement of a bug's appearance, use that variable and statement as a slicing criterion to construct mentally the corresponding slice.*

3.3.2 Original Studies

Weiser [103] formally defined slice, presented the first algorithm (which was corrected by Leung and Reghbati [58]), and provided some experience in slicing. According to him the advantages of slices are:

1. they can be found *automatically*
2. slices are generally *smaller* than the program from which they originated
3. they execute *independently* of one another
4. each *reproduces exactly* a projection of the original program's behaviour.

The disadvantges can be summarised as:

1. they can be expensive to find
2. a program may have no significant slices
3. their total independence may cause additional complexity in each slice that could be cleaned up if simple dependencies could be represented.

Weiser's program slices are taken with respect to a program point and an arbitrary variable (both forming the *slicing criterion*).

3.3.3 Decompositional Program Slicing

Gallagher and Lyle [34] have extended the notion of a program slice to a *decomposition slice*. This slice is independent of line numbers (i.e. the slicing criterion is reduced to a variable). The decomposition slice of a variable is the union of a collection of program slices. The program slices selected have as a program point an output statement or the last statement. Decomposition slices have been defined by Gallagher [32] as:

> *Let* $Output(P, v)$ *be the set of statements in program* P *that output variable* v, *let* `last` *be the last statement of* P, *and let* $N = Output(P, v) \cup \{\texttt{last}\}$. *The statements in* $\bigcup_{n \in N} S(v, n)$ *form the* decomposition slice *on* v, *denoted* $S(v)$.

This new type of slice has been implemented in a tool named Surgeon's Assistant (see Section 2.2.5) which also reflects a new software maintenance model with no regression testing involved.

3.3.4 Slicing in C

The standard techniques of Weiser [103] work only in a limited subset of the possible constructions found in modern languages. Jiang et al [50] have enhanced the original algorithm to deal with:

- the presence of array and pointer variables (the original algorithm did not produce correct results for these constructs).
- goto, break and continue statements (these statements have effects on the behaviour of the slice).

Although the algorithm they present is for use in C constructs, it should be easily adaptable to other languages with similar constructs.

3.3.5 Slices and Module Cohesion

Ott and Thuss [71] demonstrate a relationship between the slices of a module and the notions of module cohesion. Cohesion levels and their relationship with slices are as follows.

Low cohesion (coincidental and temporal) corresponds to non-intersecting slices.

Control cohesion (logical and procedural) corresponds to slices having only common control structures.

Data cohesion (communicational) corresponds to data flow relations between slices.

High cohesion (sequential and functional) corresponds to a slice being properly contained in another.

Low cohesion occurs when the slices are independent (i.e. they do not share any statement). Control cohesion happens when they share the structure of the program (e.g. same `while` and `if` constructs). Data cohesion occurs when two slices share assignments or uses of variables. Finally high cohesion is present when the statements that form one slice are a subset of the statements of another slice.

3.3.6 Program Dependence Graphs

Ferrante et al [28] have devised an intermediate program representation called the *program dependence graph* that makes explicit the data and control dependences for each operation in a program. Usually only data dependences were used to represent the dependences in a program. With the incorporation of control dependences (extracted from the control flow graph) to the program dependence graph, better optimizations and static analysis of a program can be performed. Horwitz et al [42] have enhanced the program dependence graph, and created the *system dependence graph*, which incorporates information about procedures and procedure calls. They present an interprocedural slicing algorithm that works with the system dependence graph. This algorithm has a restriction: a slice has to be taken with respect to a variable that is defined or used at the program point where the slice is to be taken. This restricts the original definition of slicing criterion given by Weiser where the variable was arbitrary.

3.3.7 Program Dependence Relations

Bergeretti and Carré [10] have introduced a relation-based information-flow (effectively data and control) analysis of while-programs (i.e. GOTO-less programs). One of their relations provides *partial statements* which are equivalent to slices. This method will be presented

in great detail in Section 5.2. Gopal [38] has created another set of program dependence relations; rather than analysing the static behaviour of a program, the new relations model the dynamic behaviour of the program.

3.3.8 Dynamic Slicing

Gopal [38] explains the difference between dynamic and static slices:

> *Using dynamic analysis techniques is possible to define a dynamic slice that contains only those statements that* actually *affect the value of a variable at the specified program location. A static slice includes all statements that* may *affect the value of a variable at the specified location.*

Dynamic slicing was originally proposed by Korel and Laski [54] for producing more precise slices by using the program runtime information. The new dynamic dependence relations are defined with respect to a specific program execution, the actual values of the variables being taken into account for calculating the statements that will form the slice. This slice does not have to be necessarily an executable subset of the program (unlike it happened with the static dependence relations).

3.3.9 Chunks

A *chunk* is a new concept described by Samuel Hsieh [43] as:

> *A chunk intuitively corresponds to the range of impact of a program change, and is defined as the union of those program slices impacted by the change. Given a chunk with respect to a program change, a maintenance programmer can consider the impact without examining the entire program.*

As noticed by Hsieh, once a maintainer has found an error (perhaps aided with a slicer) and a change is made to correct it, a new question arises: "Can the change cause new troubles, possibly in other parts of the program ?" This question is usually answered with the use of a ripple effect analyser, but the construction of a chunk can be useful too. A chunk is a small subset of a program (like a slice), so it provides a better focus in which to analyse the change than the whole program.

Chapter 4

Definition of Problem

The main problems found while performing maintenance with ReForm have been described in Section 1.5. In this chapter a more detailed definition of the problem that the static analyser and the slicer will solve is presented.

4.1 The Problems

Programs that are maintained or transformed by ReForm are usually very large (the order of magnitude is thousands of lines) and complex (control flow difficult to follow); this is mainly due to two reasons:

- Although programs are written in WSL, the initial level of abstraction used is very low because the programs have been translated from assembler (although in the future this could be COBOL, C, etc). This low level of abstraction results in
  - large number of instructions for all but the most simple operations
  - many labels and GOTOs due to lack of iterative constructions

- large number of very small routines that implement simple operations.

- The ReForm user needs to reverse engineer *real* programs and not toy-like programs only used in demonstrations. The emphasis is focused on commercial and business applications that are currently being used. This means that programs are typically very badly structured, huge, non modular, heavily altered, etc.

The maintainer is faced by a rather large and complex program written in WSL and there are no tools available to facilitate the understanding of the *source code*. It is important to notice the use of the term *source code*, because even if some documentation is available, it is usually not complete or cannot be trusted. See Section 2.2.3 for a discussion on this topic. Other maintenance or reverse engineering projects have overcome, at least partially, these problems by using a static analyser or tools based on static analysis; see Section 2.2.5.

4.2 The Solution

In the ReForm project it was decided to follow a similar approach and build a static analysis tool that will provide the foundation for a cross-referencer, modulariser, call-graph displayer and slicer. Although this thesis only describes the work performed to implement the static analyser and the slicer, the other tools could be easily implemented. These tools will assist the maintainer in:

Decomposing a program The modulariser breaks the program in a horizontal fashion separating logical functions. The slicer breaks the program in a vertical fashion separating the statements involved in the computation of the value of a variable or set of variables.

Debugging a program The slicer can show the part of the WSL program relevant to one or a set of variables. The cross-referencer can show only the statements that use or assign a value to a variable. The call-graph displayer shows the possible paths the program can follow.

Excluding dead code from a program Static analysers can detect parts of a program that will never be executed given any combination of values for the variables.

4.3 Design Decisions

It was decided that the static analysis tool should be implemented within the Maintainer's Assistant and not as a stand-alone tool, though it should not interfere with the already existing tools. The Maintainer's Assistant will become, after the incorporation of the static analyser, an integrated tool that will help the user to swap *interactively* between analysis and reverse engineering. The concept of interaction is a crucial one; the analyser will perform its functions interacting with the other tools of the Maintainer's Assistant, this means that after the program is modified by the application of transformations, the maintainer will not have to run a 'batch' procedure to update the data flow information; this information will be updated by the analyser as needed. The design of the Maintainer's Assistant did not commit the design of the slicer to any particular method of data flow analysis. The modular design of the ReForm tool assures that the slicer (and other analysis tools) can be interfaced easily with Xma (X-windows front-end of the Maintainer's Assistant) in the future.

If C or another programming language had been used as the internal language to ReForm, it could have been possible to use a commercially available static analysis tool. The use of WSL implied that a new implementation was needed for reasons explained below. Although ideally all the constructs in WSL should be analysed, it was decided that only the *low-level* ones will be dealt with.

4.4 The WSL Language

The mathematical foundations of WSL are first order logic and set theory. The denotational semantics of the kernel statements are described in terms of mathematical objects called

"state transformations". The details on how to interpret statements as state transformations are bypassed in this thesis but can be found in [97], which is the definitive source for the syntax and semantics of WSL. Most of the following information has been extracted from there.

4.4.1 Kernel Language

The syntax (and a non-formal semantic definition) of the kernel language is as follows:

- Two primitive statements:
  - **Atomic specification**, written $x/y.\boldsymbol{Q}$
  - **Guard statement**, written $[\boldsymbol{P}]$

  where $\boldsymbol{P}$ and $\boldsymbol{Q}$ are formulae of first order logic and x and y are sequences of variables.

  The effect of the atomic specification statement is to add the variables in x to the *state space*, assign new values to them such that $\boldsymbol{Q}$ is satisfied, remove the variables in y from the *state* and terminate. A state is a collection of variables with values assigned to them. The collection of all possible states is called the state space. The guard statement always terminates; it enforces $\boldsymbol{P}$ to be true at this point in the program, and it has the effect of restricting previous nondeterminism to those cases which leave $\boldsymbol{P}$ true at this point. The guard statement cannot be implemented directly because it can force determinism on previous non deterministic constructs.

- A set of **statement variables**, which are symbols used to represent recursive calls of recursive statements.

- Three compounds:
  - **Sequential composition**: $(\boldsymbol{S}_1; \boldsymbol{S}_2)$

    First $\boldsymbol{S}_1$ is executed and then $\boldsymbol{S}_2$.

- **Choice:** $(S_1 \sqcap S_2)$

  One of the statements S_1 or S_2 is chosen for execution.

- **Recursive procedure:** $(\mu X.S_1)$

  Within the body S_1, occurrences of the statement variable X represent recursive calls to the procedure.

4.4.2 The First Level Language

The kernel language is very compact and has a sound mathematical foundation, allowing transformations of the code be proved, but it is not very useful for writing programs or for use as the target language for a translator. Extensions to the language are built in a series of *language levels*, with each level defined in terms of the previous one. The first-level language has constructs usually found in other languages (e.g. assignment, deterministic choice, deterministic iteration, procedure call) and some other constructs which are not new, but rarely found in programming languages (e.g. nondeterministic choice, nondeterministic iteration, action system). High levels are concerned with specification issues, but those remain to be implemented in ReForm. A complete definition of the syntax of WSL is given in Appendix A.

The slicer will eventually work upon all the constructs in the first level (and perhaps will be extended to other levels), but the implementation is being carried out in incremental stages because this allows the assessment of the prototype, and because the algorithms utilised to analyse some of the constructs can be written in terms of other constructs' algorithms (in the same way that one language level can be implemented in terms of the previous one). In the last two stages an interprocedural analyser will be needed, and the information that it will provide will be used in the construction of a library of procedures which is being designed for the Maintainer's Assistant.

The stages are divided as follows (the figures in square brackets refer to the index of the Table in Appendix A):

1. Basic statements:

   `Abort [30] , Assert [33] , Comment [36] , Skip [46] , Assignment [8,34]`

   `Deterministic Choice [37] , Nondeterministic Choice [38] , While [49]`

2. Loop statements:

   `For loop [42] , Nondeterministic do loop [39]`

   `Multiple-level exit loops [40,41]`

   All these can be expressed in terms of while loops, but the last one is more difficult to implement.

3. `Action systems [31,35]`

   These will be discussed below.

4. Enhanced facilities to deal with variables:

   `Arrays [29,32] , Local variables [47]`

5. WSL procedures and functions:

   `Block with local procedure [48] , Procedure definitions [50,51,52]`

   `Calls [45,63,87]`

6. `External procedures, functions and conditions: [43,44,64,88]`

   These are effectively calls to procedures which are not written in WSL or belong to another module.

In practice, constructs of the first three groups (with the exception of the multiple-level exit loops) are supported by the analyser. The main problems of WSL for static analysis compared with 'more conventional' programming languages are: action systems and non determinism.

Action Systems

An *action system* is a set of parameterless mutually recursive procedures [97]. They are used to model programs written using labels and jumps (GOTOs). The main difference with 'actions' found in other languages is explained by Ward [97]:

```
Actions (A)
   A ≡ x:=3; call B; x:=x+7;
   B ≡ x:=x+5;
end;
{x = 15}
```

```
Actions (A)
   A ≡ x:=3; call B; x:=x+7;
   B ≡ x:=x+5; call Z;
end;
{x = 8}
```

Figure 4.1: Examples of Action Systems

> *... if the end of the body of an action is reached, then control is passed to the action which called it (or the statement following the action system) rather than "falling through" to the next label. The exception to this is a special action called the terminating action, usually denoted* ***Z****, which when called results in the immediate termination of the whole action system.*

The behaviour of an action is very similar to the one of a procedure, except when the action ***Z*** is included in the action system. The small example of Figure 4.1 should clarify this.

There is a characteristic that classifies action systems: an action is *regular* if every execution of the action leads to an action call. An action system is regular if every action in the system is regular. Any algorithm defined by a flowchart, or a program which contains labels and GOTOs but no procedure calls in non-terminal positions, can be expressed as a regular action system [97]. The first example in Figure 4.1 is non-regular because it has an action which is not regular (B) and hence makes the whole system non-regular; the second example is regular.

Non determinism

There are two non deterministic constructs: the nondeterministic choice and the nondeterministic iteration. The second can be rewritten in terms of a deterministic iteration and a nondeterministic choice. The nondeterministic choice is equivalent to the "guarded command" of Dijkstra; it is like a multiple (deterministic) choice but it aborts if there is no

condition that evaluates true.

Chapter 5

Method of Solution

In this chapter the choice of Bergeretti and Carré's static analysis techniques is explained and justified, and the method itself is presented. The key issues addressed in this thesis that are novel are:

- The non-deterministic constructs in WSL and the action systems.
- The engineering need for an interactive integrated tool.

5.1 The Choice

This method was chosen because it is not compiler oriented. Most of the traditional methods for static analysis were designed for optimising compilers and their analysis is based in call-graphs and basic blocks; see for example Aho et al [2] or Fischer and LeBlanc [29].

Bergeretti and Carré [10] have devised a method to perform static analysis on a syntax-tree representation of a program, which is a representation closer to the source language than the one provided by basic blocks which is more object language oriented.

Another key advantage of Bergeretti and Carré's method is that once the data flow information has been calculated for a program, slices can be obtained in *linear* time. This is not the case in other methods (e.g. the original (Weiser) and derived (Gallagher) decomposition methods) where data flow information has to be recalculated if the slice has to be built for another variable or set of variables.

5.2 The Method

The data flow analysis method used collects syntactic information, which implies a "conservative" approach. This means that some data dependencies will be indicated even if the program semantics do not allow them. This also means that the word *'may'* appears throughout the definitions to describe the dependencies between variables and statements. A formal definition of the method extracted from Bergeretti and Carré [10] is presented below.

5.2.1 Notation

Basic sets

- $\boldsymbol{V}$ set of all variables in a program.
- $\boldsymbol{E}$ set of all instances of an expression in a program (this is usually associated with statement numbers or labels).
- $\boldsymbol{\Gamma(e)}$ set of all variables that appear in e for each $e \in E$.
- $\boldsymbol{S}$ statement (or compound statement).
- $\boldsymbol{D_S}$ set of variables which S *may define*: $v \in D_S$ if S contains at least one assignment to v.

- $\boldsymbol{P_S}$ set of variables which S *may preserve*: $v \in P_S$ if S contains at least one path from entry to exit which is definition clear for v (i.e. which has no assignments to v).

Relations

The analysis of a program is performed by constructing three binary relations:

$$\lambda : V \rightarrow E \, , \; \mu : E \rightarrow V \, , \; \rho : V \rightarrow V$$

The informal meaning of these relations is

- $\boldsymbol{v \lambda_S e}$ the value of v on entry to S *may be used* in evaluating e
- $\boldsymbol{e \mu_S v}$ a value in e *may be used* in obtaining the value of v on exit from S
- $\boldsymbol{v \rho_S v'}$ the value of v on entry to S *may be used* in obtaining the value of v' on exit from S

This last relation looks very much as a combination of the first two, and in fact it can be expressed as:

$$\rho_S = \lambda_S \mu_S \cup \Pi_S \text{ where } \Pi_S = \{(v, v) \in V \times V \mid v \in P_S\}$$

which informally means that:

1. the value of v on entry to S *may be used* in obtaining e which in turn *may be used* in obtaining the value of v' on exit from S, or
2. $v = v'$ and S *may preserve* v.

Another useful definition is the equality relation on V:

$$\iota = \{(v, w) \in V \times V \mid v = w\}$$

Operations

Finally the elemental operations on matrices and sets will be denoted as follows:

product of two matrices, represented with no space between them

$\times$ cartesian product of two sets to form a matrix

$*$ transitive closure as in ρ_S^*.

$\cup$ union of sets or sum of matrices

$\cap$ intersection of sets

5.2.2 Definitions

Once all the notation has been established, it is possible to give the definitions for the relations λ_S, μ_S, and ρ_S for each type of statement. The building of these definitions can be found in Section 2.2 of the article by Bergeretti and Carré [10]. A summary of these definitions is presented in Table 5.1, which has been copied from Table 1 of [10].

5.3 Examples

To clarify how the method works and to understand the result of analysing different sections of a program, an example extracted from [10] is presented. The extended Euclidean algorithm to calculate the greatest common denominator (x) of two integers (m and n) and their multipliers (y and z) is presented in Figure 5.1.

Three sets of information-flow relations referring to the implementation of these algorithm are displayed in Figures 5.2–5.4. Figure 5.2 contains the relations for the body of the while-statement (i.e. statements 8–19), the relations for the while-statement (i.e. statements 7–19)

Empty Statements. For an empty (or "`skip`") statement S,

$$D_S = \phi \quad \text{(T1)} \qquad P_S = V \quad \text{(T2)}$$
$$\lambda_S = \phi \quad \text{(T3)} \qquad \mu_S = \phi \quad \text{(T4)}$$
$$\rho_S = \iota \quad \text{(T5)}$$

Assignment Statements. For an assignment statement S, which assigns a value to v and whose expression part is e: v:=e,

$$D_S = \{v\} \quad \text{(T6)} \qquad P_S = V - \{v\} \quad \text{(T7)}$$
$$\lambda_S = \Gamma(e) \times \{e\} \quad \text{(T8)} \qquad \mu_S = \{(e, v)\} \quad \text{(T9)}$$
$$\rho_S = (\Gamma(e) \times \{v\}) \cup (\iota - \{(v, v)\}) \quad \text{(T10)}$$

Sequences of Statements. For a sequence S of two statements: $(A; B)$,

$$D_S = D_A \cup D_B \quad \text{(T11)} \qquad P_S = P_A \cap P_B \quad \text{(T12)}$$
$$\lambda_S = \lambda_A \cup \rho_A \lambda_B \quad \text{(T13)} \qquad \mu_S = \mu_A \rho_B \cup \mu_B \quad \text{(T14)}$$
$$\rho_S = \rho_A \rho_B \quad \text{(T15)}$$

Conditional Statements. For a statement S of the form: `if` e `then` A `else` B,

$$D_S = D_A \cup D_B \quad \text{(T16)} \qquad P_S = P_A \cup P_B \quad \text{(T17)}$$
$$\lambda_S = (\Gamma(e) \times \{e\}) \cup \lambda_A \cup \lambda_B \quad \text{(T18)} \qquad \mu_S = (\{e\} \times (D_A \cup D_B)) \cup \mu_A \cup \mu_B \quad \text{(T19)}$$
$$\rho_S = (\Gamma(e) \times (D_A \cup D_B)) \cup \rho_A \cup \rho_B \quad \text{(T20)}$$

For a statement S of the form: `if` e `then` A,

$$D_S = D_A \quad \text{(T21)} \qquad P_S = V \quad \text{(T22)}$$
$$\lambda_S = (\Gamma(e) \times \{e\}) \cup \lambda_A \quad \text{(T23)} \qquad \mu_S = (\{e\} \times D_A) \cup \mu_A \quad \text{(T24)}$$
$$\rho_S = (\Gamma(e) \times D_A) \cup \rho_A \cup \iota \quad \text{(T25)}$$

Repetitive statements. For a statement S of the form: `while` e `do` A,

$$D_S = D_A \quad \text{(T26)} \qquad P_S = V \quad \text{(T27)}$$
$$\lambda_S = \rho_A^*((\Gamma(e) \times \{e\}) \cup \lambda_A) \quad \text{(T28)} \qquad \mu_S = (\{e\} \times D_A) \cup \mu_A \rho_A^*((\Gamma(e) \times D_A) \cup \iota) \quad \text{(T29)}$$
$$\rho_S = \rho_A^*((\Gamma(e) \times D_A) \cup \iota) \quad \text{(T30)}$$

Table 5.1: Definitions of the Information-Flow Relations

Expression Number	
	procedure GCD(m, n:integer; **var** x, y, z:integer);
	var a1, a2, b1, b2, c, d, q, r:integer; $\{m \geq 0, n > 0\}$
	begin {Greatest Common Divisor x of m and n, Extended Euclid's Algorithm}
1–4	a1:=0; a2:=1; b1:=1; b2:=0;
5, 6	c:=m; d:=n;
7	**while** (d<>0) **do**
	begin $\{d = a1 * m + b1 * n,\ c = a2 * m + b2 * n,\ \gcd(c, d) = \gcd(m, n)\}$
8, 9	q:=(c **div** d); r:=(c **mod** d);
10, 11	a2:=(a2-(q*a1)); b2:=(b2-(q*b1));
12, 13	c:=d; d:=r;
14–16	r:=a1; a1:=a2; a2:=r;
17–19	r:=b1; b1:=b2; b2:=r
	end;
20–22	x:=c; y:=a2; z:=b2
	$\{x = \gcd(m, n) = y * m + z * n\}$
	end

Figure 5.1: Extended Euclidean algorithm

appear in Figure 5.3, and finally the relations for the whole procedure (i.e. statements 1–22) are given in Figure 5.4. Each relation is represented by its boolean matrix, with empty rows and columns removed.

The results shown in Figures 5.2–5.4 have been corrected from the original paper by Bergeretti and Carré; there was one mistake in each of the three matrices for the μ relation (this μ is unconnected with the recursive function in WSL), and the corrections were made in positions $(19, r)$, $(8, r)$, and $(11, r)$, respectively.

5.4 Slicing

Bergeretti and Carré define the new concept of "Partial Statements", which is very similar to the concept of program slice. Given any program variable v and for any statement S let E_S^v be the set of expressions

$$E_S^v = \{e \in E \mid e \mu_S v\}$$

which informally means that E_S^v is the set of expressions in S whose values *may be used* in obtaining the value of v on exit from S. Now let S^v be the program derived from S by replacing every statement within S which does not contain some member of E_S^v by an empty statement. The program S^v is equivalent to S in the sense that, for any set of input values, the values of v on exit from S and S^v are identical. The set of expressions S^v is described as the partial statement of S associated with v. Again, an example will clarify this concept; Figure 5.5 shows the slices (or partial statements) of the program in Figure 5.1 for the variables x, y, and z. The slices can be easily obtained from the matrix μ_S in Figure 5.4 by looking at the columns x, y, and z; the statements relevant to the slice for a variable have a '1' in the column of that variable.

$$\lambda_A = \begin{array}{c} \\ a1 \\ a2 \\ b1 \\ b2 \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccccccccccc} 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

$$\mu_A = \begin{array}{c} \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \\ 16 \\ 17 \\ 18 \\ 19 \end{array} \begin{array}{c} \begin{array}{cccccccc} a1 & a2 & b1 & b2 & c & d & q & r \end{array} \\ \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$\rho_A = \begin{array}{c} \\ a1 \\ a2 \\ b1 \\ b2 \\ c \\ d \\ m \\ n \\ q \\ r \\ x \\ y \\ z \end{array} \begin{array}{c} \begin{array}{cccccccccccc} a1 & a2 & b1 & b2 & c & d & m & n & q & r & x & y & z \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

Figure 5.2: Information-flow relations for the body of the while-statement

$$\lambda_S = \begin{array}{c} \\ a1 \\ a2 \\ b1 \\ b2 \\ c \\ d \end{array} \begin{array}{c} \begin{array}{ccccccccccccc} 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

$$\mu_S = \begin{array}{c} \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \\ 16 \\ 17 \\ 18 \\ 19 \end{array} \begin{array}{c} \begin{array}{cccccccc} a1 & a2 & b1 & b2 & c & d & q & r \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

$$\rho_S = \begin{array}{c} \\ a1 \\ a2 \\ b1 \\ b2 \\ c \\ d \\ m \\ n \\ q \\ r \\ x \\ y \\ z \end{array} \begin{array}{c} \begin{array}{ccccccccccccc} a1 & a2 & b1 & b2 & c & d & m & n & q & r & x & y & z \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

Figure 5.3: Information-flow relations for the while-statement

$$\lambda_S = \begin{array}{c} \\ m \\ n \end{array} \begin{array}{c} \begin{array}{cccccccccccccccccc} 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 \end{array} \\ \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

$$\mu_S = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \\ 16 \\ 17 \\ 18 \\ 19 \\ 20 \\ 21 \\ 22 \end{array} \begin{array}{c} \begin{array}{cccccccccccc} a1 & a2 & b1 & b2 & c & d & q & r & x & y & z \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

$$\rho_S = \begin{array}{c} \\ m \\ n \\ q \\ r \end{array} \begin{array}{c} \begin{array}{ccccccccccccc} a1 & a2 & b1 & b2 & c & d & m & n & q & r & x & y & z \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Figure 5.4: Information-flow relations for the complete algorithm

Expression Number

```
          begin
 1, 2     a1:=0; a2:=1;
 5, 6     c:=m; d:=n;
  7       while (d<>0) do
          begin
  8          q:=(c div d);
  9          r:=(c mod d);
 10          a2:=(a2-(q*a1));
12, 13       c:=d; d:=r;
14–16        r:=a1; a1:=a2; a2:=r;
          end;
 21       y:=a2
          end
```

Slice on y

Expression Number

```
          begin
 3, 4     b1:=1; b2:=0;
 5, 6     c:=m; d:=n;
  7       while (d<>0) do
          begin
  8          q:=(c div d);
  9          r:=(c mod d);
 11          b2:=(b2-(q*b1));
12, 13       c:=d; d:=r;
17–19        r:=b1; b1:=b2; b2:=r
          end;
 22       z:=b2
          end
```

Slice on z

Expression Number

```
          begin
 5, 6     c:=m; d:=n;
  7       while (d<>0) do
          begin
  9          r:=(c mod d);
12, 13       c:=d; d:=r;
          end;
 20       x:=c
          end
```

Slice on x

The slice on x is equivalent to a simplified algorithm to calculate the gcd only, without the multipliers.

Figure 5.5: Slices of the extended Euclidean algorithm of Figure 5.1

$$\mu_S = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \\ 16 \\ 17 \\ 18 \\ 19 \\ 20 \\ 21 \\ 22 \end{array} \begin{array}{c} \begin{array}{cccccccccccc} a1 & a2 & b1 & b2 & c & d & q & r & x & y & z \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

Figure 5.6: μ relation for the modified algorithm

5.5 Ineffective Statements

Relation μ can also be used to detect "ineffective statements", which are statements that do not contribute to the final values of the exported variables. The exported variables are the ones which are *alive* (i.e. their value is used outside the procedure, usually result parameters) at the end of a procedure. Following with the example in Figure 5.1, the exported variables are x, y, and z, and there are no ineffective statements. However if statement number 17 is modified to `r:=a1` (it was originally `r:=b1`) then the relation μ for the complete algorithm would become that shown in Figure 5.6.

Statements 3, 11, and 18 have their rows empty for the exported variables x, y, and z so they would be ineffective statements. The elimination of ineffective statements is equivalent to excluding dead code.

5.6 Modifications

Table 5.1 had to be modified to adapt the different syntax and semantics of WSL, as explained in Section 4.4, and to include other features of the language such as action systems and non-deterministic constructs.

The method used by Bergeretti and Carré to compute the information-flow relations of a program is based on a postorder traversal of the syntax tree; see [1] for several algorithms on tree traversals. The information needed to obtain the information-flow relations for a node can be found in the node itself and in its most direct descendants (i.e. sons). Once the traversal of the tree has finished the only data flow information saved is the information-flow relations for the root node. It is a global method: although information has not to be recalculated if a slice is needed for a different variable and the same statement, if the slice is needed for a different position, information has to be computed from scratch. This way of working suggests that the analyser is used in a 'batch' manner. Programs are only analysed once it is not likely they will be modified.

5.6.1 Interactive

It was essential to modify this method because a more interactive use of the analyser is required. Three major differences with the original method were designed:

1. Type of tree traversal used to analyse the program. In the new method a preorder traversal of the program tree is performed to check whether the data flow information for a node has to be computed or not. If this is the case, the information for the descendants is recursively calculated first (like in a postorder traversal) and then the information for the node itself is computed and stored in the database.

2. The information obtained for each node (and not only for the root) is kept in the database.

3. The original method assumes that all the variables are known in any node of the tree (this is needed to compute ι in the assignment, if and while statements), but in the new method, no assumption is made and the set of variables is computed by the algorithm itself.

These modifications to the original algorithm provide more flexibility and allow a much quicker recalculation of the data flow information if this is needed, because they imply an incremental update of the information, avoiding a global computation each time the program is modified.

For instance, if a slice is wanted for a node of the tree which is a descendant of a node for which the data flow information has already been computed, then no extra computation is needed; it is just a matter of reading the database, extracting the information from the μ relation and displaying the slice. The reciprocal case is also possible: if a slice is wanted for a node of the tree which is an ascendant of a node for which the data flow information has already been computed, then computation is needed only on the part of the tree where database information is not existent (or is no longer valid due to modifications).

An example based on Figure 5.7 will clarify this last point: Assume that node 5 has been modified, and the data flow information (dfi) has to be calculated for node 9. The current validity of the data flow information is stored in the boolean variable *flag*. On an exhaustive postorder traversal of the tree, all the nodes will be visited (and updated!) in the order from 1 to 9 because there is no information kept in the nodes. On the new method's traversal, the nodes will be visited as follows:

9 test flag nil

5 test flag nil

3 test flag ok

4 test flag ok

5 update dfi set flag:= ok

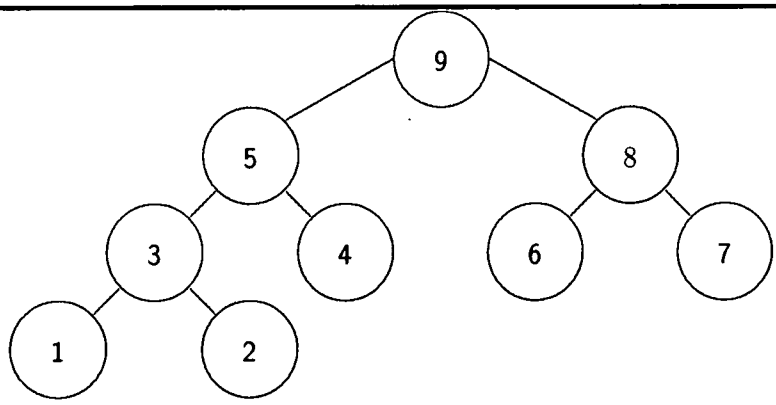

Figure 5.7: Types of traversal

8 test flag ok

9 update dfi set flag:=ok

5.6.2 New constructs

The method used to analyse action systems is as follows:

1. For each action: collect data flow information, assuming that an embedded call statement is equivalent to a skip statement.
2. While there has been a change on the global data flow information do steps 3–5
3. For each action do steps 4–5
4. Substitute the information of any call statement with the information of the action called.
5. Recalculate the new information for the action.

It is an iterative method that converges to a fixed point which is a solution of the data flow analysis problem.

Non determinism was treated as determinism from the syntactical point of view. In both deterministic and non deterministic choice statements, all paths are potentially executable, and that is the only thing relevant for the static analysis which is not concerned with the actual path executed. The non deterministic iteration can be rewritten as a non deterministic choice statement enclosed in a deterministic iteration, so no special treatment is required.

Chapter 6

Solution

6.1 Prototyping

The advantages of using a prototyping model for the development of software have been demonstrated in the article by Carey [18] and in the book by Maude and Willis [62]. The use of prototyping causes a very quick development, and hence the system can be implemented rapidly. Once the implementation is finished, an assessment whether the prototype reflects what was expected can be made. Changes and minor modifications can be done very quickly and the system retested again in a very short period of time (the choice of language is also responsible for this). Hence a prototype model was used for this implementation.

6.2 Architecture

The overall architecture of the Maintainer's Assistant has already been shown in Figure 2.2. This architecture will be enhanced with the incorporation of the data flow analyser, slicer

and other static tools, as shown in Figure 6.1. At the present moment the interface with the maintainer is not very user friendly, and all the commands to the tool have to be typed in the LISP interpreter. In the future the user interface will be through the already existing browser and the X front-end; some experiments have shown that this is a simple matter, and the relevant screen design has been incorporated into ReForm.

6.3 LISP

Although the author did not have any previous experience in LISP, it was chosen as the implementation language because:

- All the code for the Maintainer's Assistant is written in LISP (except the user interface which is written in C because of X-Windows). This provides a library of code which can be reused with no modification and without the problems (e.g. external calls, parameter passing conventions, return of values, etc.) inherent when more than one programming language is used.
- LISP is a functional language with facilities to perform operations on symbols; there is no need for 'strings'. This is very useful because objects like variables and positions of statements have to be treated during the analysis.
- It is an interpreted language which results in benefits for a rapid prototyping (i.e. programming driven by testing). Debugging facilities are built into the interpreter, so there is no need for special compilations and external debuggers.
- It is possible to compile Common Lisp (which is the LISP used in ReForm) to give C source code, which can be in turn compiled producing an executable that runs faster than the interpreted language. This is convenient once the development stage is finished and when a more efficient code is important.

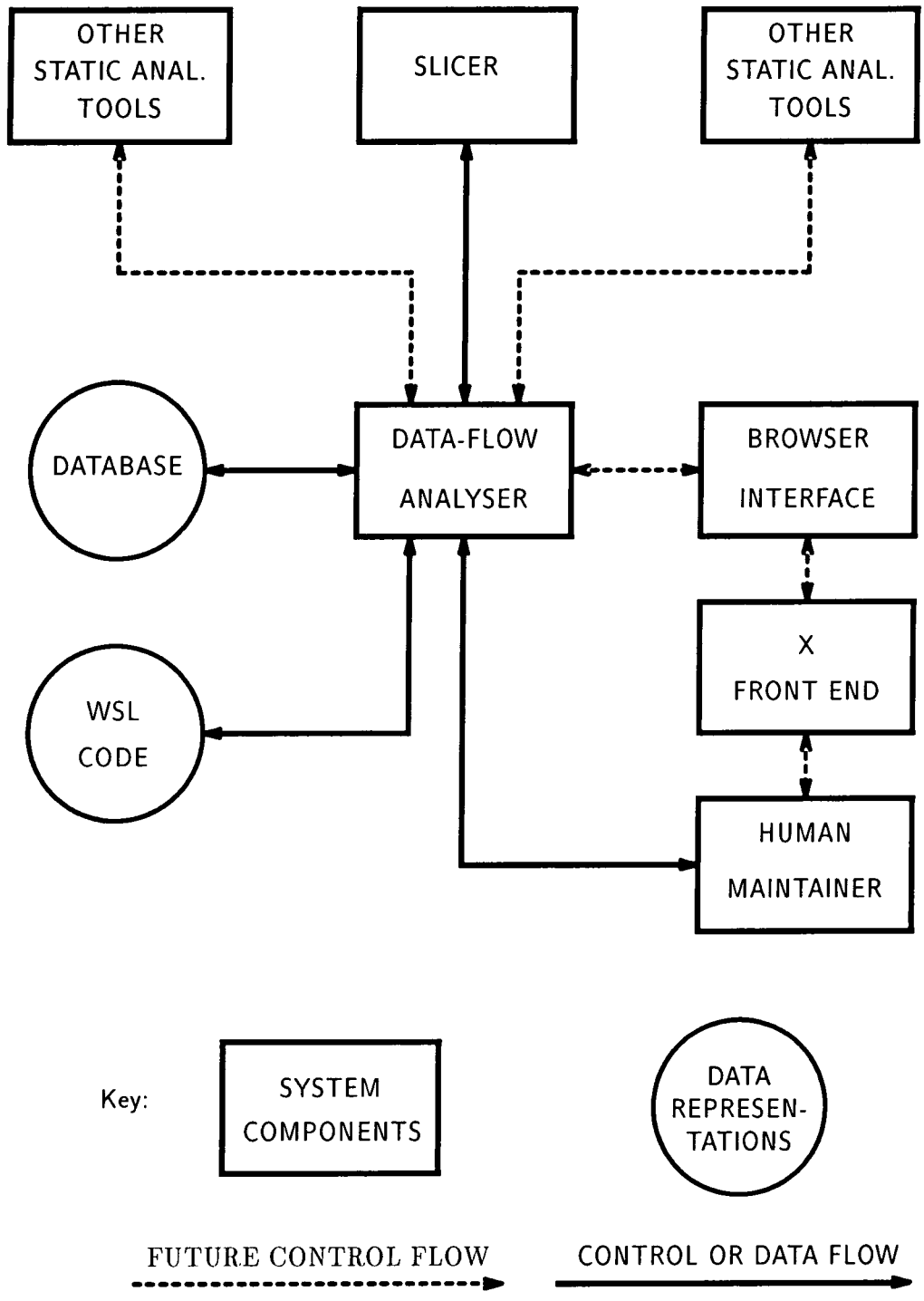

Figure 6.1: Architecture of the data flow tool in the Maintainer's Assistant

6.4 Data Structures

The database used was the one already present in the Maintainer's Assistant (see Figure 2.2), which holds various types of information needed to perform the transformations. Some of this information was also used in the data flow analyser and new information was added too.

Two types of information specific to static analysis are saved in the database for each node of the syntax tree:

- A flag to specify whether the present information is correct or has to be updated.
- The actual data flow information; this consists of:

  n name of this action

  c actions (and positions of calls) called by this one

  v all the variables at this position

  a assigned variables in this position

  g Γ used variables, only for conditions

  l λ relation

  m μ relation, used to compute slices

  r ρ relation

Positions are usually associated with statement numbers or labels, but to be coherent with the syntax tree representation used in WSL, the position of statement in a node n will be represented as the path from the node acting as reference system and the node n itself. When a node is being analysed its own position is nil, but if this same node is being referenced by one of its ascendants, its position is represented by a list containing the path from the ascendant to the node. An example of the different values of positions for the same node is shown in Figure 6.2.

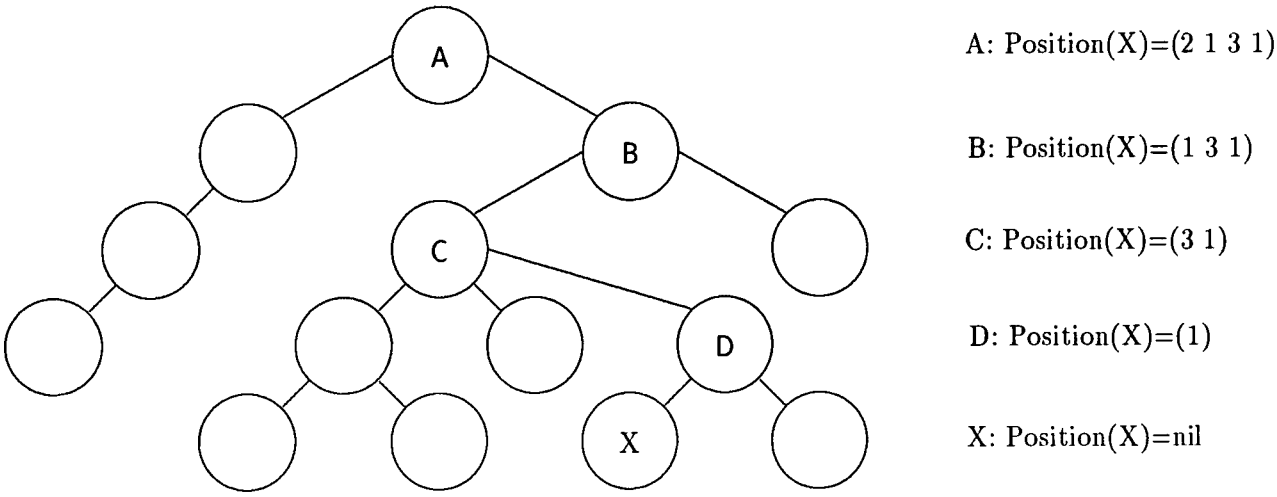

Figure 6.2: Positions of node X with respect to nodes A–D

6.5 Matrices

Matrices are the very basic data structure of the algorithm, so an abstract data type was created for them. The physical implementation of matrices, containing data structures and operations, is held in a separate file. The operations needed are *identity* and *cartesian product* to create matrices and *sum*,*product* and*transitive closure* to combine matrices. Although the transitive closure could have been implemented using the method of Warshall [100] or the improved method of Warren [99], a more straightforward implementation in terms of product and sum was used.

The LISP representation of a matrix is a list of dotted lists, for example:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{ccc} x & y & z \end{array} \\ \left(\begin{array}{ccc} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{array} \right) \end{array} \quad \text{is represented as } ((2.y)\,(3.x)\,(5.z))$$

This has some advantages:

- The actual dimensions of the matrix are not needed neither to save the data nor to perform the operations on it.
- It is a much more compact representation if the matrices are very sparse, and this is the case in almost all programs (except very small programs of few lines or so).
- It simplifies the implementation of some matrix operations, for example the sum can be implemented as a merge of lists. In some operations this also means a speed-up.

The reason why these matrices are very sparse is because of the data they represent. If all the elements of column x in the matrix above were set to '1', this would mean that all the statements (from 1 to 6) are relevant to the computation of the final value of x, which is unlikely in a normal program.

6.6 Implementation details

The already existing database provided two simple functions that returned the assigned and used variables of a statement. Although these functions were not used for all the statements,

they proved very useful for calculating the used variables of an expression without having to go into great detail analysing the expression.

The lack of experience in LISP proved very hard sometimes, particularly with the functions `equalp, sort, setq`. To cut a long story short:

- `equalp` is the function to use to compare structures; such comparison does not work with `eq` or `equal`
- `sort` destroys its argument
- `setq` does not copy lists, though it copies simple variables

The combination of the last two proved particularly annoying, as shown in the examples of Figure 6.3 which (for the first two cases) shows the pitfalls of misunderstanding the above points.

The construct action system has been implemented in full for non-regular action systems, and implemented for non-recursive regular action systems.

An abstract data type implementing matrices and their operations was created on a separate file. Once the matrix operations were reliable this file was compiled. This meant that the analysis was somewhat faster and that each modification of the main file containing the analyser did not imply a recompilation.

```
>(setq x '(f i s c h e r))    ; set the value of x
(F I S C H E R)

>(setq y x)                  ; copy the value of x (Wrong!)
(F I S C H E R)

>(sort x #'string<)          ; sort x (Wrong!)
(C E F H I R S)

>x                           ; print value of x (!!!)
(F H I R S)

>y                           ; print value of y (!!!)
(F H I R S)
```

```
>(setq x '(f i s c h e r))    ; set the value of x
(F I S C H E R)

>(setq y x)                  ; copy the value of x (Wrong!)
(F I S C H E R)

>(setq x (sort x #'string<)) ; sort x (OK!)
(C E F H I R S)

>x                           ; print value of x (OK!)
(C E F H I R S)

>y                           ; print value of y (!!!)
(F H I R S)
```

```
>(setq x '(f i s c h e r))    ; set the value of x
(F I S C H E R)

>(setq y (copy-tree x))      ; copy the value of x (OK!)
(F I S C H E R)

>(setq x (sort x #'string<)) ; sort x (OK!)
(C E F H I R S)

>x                           ; print value of x (OK!)
(C E F H I R S)

>y                           ; print value of y (OK!)
(F I S C H E R)
```

Figure 6.3: Examples in LISP

Chapter 7

Results - Test cases

Testing the algorithm has presented a challenge. This was achieved by testing small examples whose results can be checked by hand. Two larger examples were tested by comparing the actual results with results given in published papers. A selection of small examples together with the two larger examples are shown in this chapter.

The example provided by Bergeretti and Carré's paper is an algorithm to calculate the great common divisor (and the two multipliers) of two integers, it has already been presented in Section 5.3. The example discussed in Gallagher's dissertation [32] is the unix utility wc (actually it is a simplified version, but this does not affect the validity of the results in any way).

The analyser commands that appear in these listings are:

- **(nwp prog)** the WSL program prog becomes the current program, the information about the last program loaded is lost.
- **(prpr)** a pretty-printed format of the WSL program is displayed.
- **(dfa_update pos)** updates the data flow information of the program at position pos;

```
>(prpr)

((ASSIGN (X Y)))

>(dfa_slice nil '(x))

((ASSIGN (X Y)))

>(dfa_slice nil '(y))

NIL
```

Figure 7.1: Simple assign instruction

usually this position is nil (i.e. the whole program).

- **(dfa_slice pos list_vars)** a WSL pretty-printed format of the slice at position pos for the variables list_vars is displayed.

The programs shown on the text are displayed in a Pascal-like syntax for the benefit of readers unfamiliar with WSL.

7.1 Small

In Figure 7.1 the simplest program in WSL is presented (actually a program consisting of one skip statement is even simpler, but it is of no use). This simple program consists of only one assignment: `x:=y` The slice for x is obviously the same assignment; it has to be noted that the slice for y is not the same assignment because no value is assigned to y.

7.2 Sequential vs Parallel

```
>(nwp'((assign (x y)) (assign (z x))))
NIL

>(dfa_slice nil '(z))

((ASSIGN (X Y)) (ASSIGN (Z X)))

>(nwp'((assign (x y) (z x))))
NIL

>(dfa_slice nil '(z))

((ASSIGN (Z X)))

>
```

Figure 7.2: Differences between parallel and sequential assignments in WSL

This second example (Figure 7.2) illustrates the differences between parallel and sequential assignments.

The first part of the figure shows the program

```
x:=y;
z:=x;
```

We then slice on z giving the result

```
x:=y;
z:=x;
```

This is as expected because of the dependency of the x on the second assignment over the first assignment, showing the data flow dependency.

In the second part of the figure we now show there is no dependency. We start with a parallel assignment

```
>(prpr)

((ASSIGN (SUM 0) (PROD 1))
 (FOR I 1 5 2 (ASSIGN (SUM (+ SUM I))) (ASSIGN (PROD (* PROD I)))))

>(dfa_slice nil '(sum))

((ASSIGN (SUM 0)) (FOR I 1 5 2 (ASSIGN (SUM (+ SUM I)))))

>(dfa_slice nil '(prod))

((ASSIGN (PROD 1)) (FOR I 1 5 2 (ASSIGN (PROD (* PROD I)))))
```

Figure 7.3: For loop

```
< x:=y; z:=x; >
```

and then slice on z giving

```
z:=x;
```

There is no data flow dependency now because both assignments are executed at the same time. The slice shows clearly the differences between programs.

7.3 For Loop

Figure 7.3 shows a very simple for loop that calculates the sum and product of the first three odd numbers. We first show the program:

```
< sum:=0; prod:=1 >
for i:=1 to 5 step 2 do
```

```
    sum:=sum+i;
    prod:=prod*i;
  od
```

Then the slices are presented, The slice for sum is

```
  sum:=0;
  for i:=1 to 5 step 2 do
    sum:=sum+i;
  od
```

The slice for prod is

```
  prod:=1;
  for i:=1 to 5 step 2 do
    prod:=prod*i;
  od
```

7.4 GCD

The algorithm already presented in Section 5.3 is presented in Figure 7.4 in WSL format. The slices for this algorithm are given in Figure 7.4. Comparing the two sets of results provided a way of debugging the slicer and also showed the mistakes highlighted in Section 5.3.

7.5 Word Counter

The algorithm shown in Figure 7.5 has been extracted from Gallagher's thesis and has been used as a test for the slicer. Its slices are shown in Figure 7.6. Assignments of the type

```
>(prpr)

((ASSIGN (A1 0) (A2 1) (B1 1) (B2 0) (C M) (D N))
 (WHILE (<> D 0)
   (ASSIGN (Q (DIV C D)))
   (ASSIGN (R (MOD C D)))
   (ASSIGN (A2 (- A2 (* Q A1))))
   (ASSIGN (B2 (- B2 (* Q B1))))
   (ASSIGN (C D)) (ASSIGN (D R))
   (ASSIGN (R A1)) (ASSIGN (A1 A2)) (ASSIGN (A2 R))
   (ASSIGN (R B1)) (ASSIGN (B1 B2)) (ASSIGN (B2 R)))
 (ASSIGN (X C)) (ASSIGN (Y A2)) (ASSIGN (Z B2)))

>(dfa_slice nil '(x))

((ASSIGN (C M) (D N))
 (WHILE (<> D 0) (ASSIGN (R (MOD C D))) (ASSIGN (C D)) (ASSIGN (D R)))
 (ASSIGN (X C)))

>(dfa_slice nil '(y))

((ASSIGN (A1 0) (A2 1) (C M) (D N))
 (WHILE (<> D 0) (ASSIGN (Q (DIV C D))) (ASSIGN (R (MOD C D)))
        (ASSIGN (A2 (- A2 (* Q A1)))) (ASSIGN (C D)) (ASSIGN (D R))
        (ASSIGN (R A1)) (ASSIGN (A1 A2)) (ASSIGN (A2 R)))
 (ASSIGN (Y A2)))

>(dfa_slice nil '(z))

((ASSIGN (B1 1) (B2 0) (C M) (D N))
 (WHILE (<> D 0) (ASSIGN (Q (DIV C D))) (ASSIGN (R (MOD C D)))
        (ASSIGN (B2 (- B2 (* Q B1)))) (ASSIGN (C D)) (ASSIGN (D R))
        (ASSIGN (R B1)) (ASSIGN (B1 B2)) (ASSIGN (B2 R)))
 (ASSIGN (Z B2)))

>
```

Figure 7.4: Algorithm for GCD and its slices

```
((ASSIGN (YES 1) (NO 0)) (ASSIGN (INWORD NO) (NL 0) (NW 0) (NC 0))
 (ASSIGN (C GETCHAR))
 (WHILE (<> C EOF) (ASSIGN (NC (+ NC 1)))
        (COND ((= C "n") (ASSIGN (NL (+ NL 1)))))
        (COND
          ((OR (OR (= C " ") (= C "n")) (= C "t"))
           (ASSIGN (INWORD NO)))
          ((ELSE)
           (COND
             ((= INWORD NO) (ASSIGN (INWORD YES))
              (ASSIGN (NW (+ NW 1)))))))
        (ASSIGN (C GETCHAR)))
 (ASSIGN (NL NL) (NW NW) (NC NC)))
```

Figure 7.5: Unix utility wc (word counter) in WSL

```
x:=x
```

are used to represent the print statement

7.6 Action Systems

The Figures 7.7–7.8 show an example on slicing action systems and elimination of dead code. It also doubles as an example on non determinism. The program is

```
< x:=0; y:=0; z:=0; >
Actions: (A B)
  A: < x:=1; z:=1; > call C
  B: < x:=2; y:=z; z:=x; >
  C: < x:=3; z:=y; >
end
```

The program starts by resetting the values of x, y and z; and then the action system is

```
>(dfa_slice nil '(nl))

((ASSIGN (NL 0)) (ASSIGN (C GETCHAR))
 (WHILE (<> C EOF) (COND ((= C "n") (ASSIGN (NL (+ NL 1)))))
        (ASSIGN (C GETCHAR)))
 (ASSIGN (NL NL)))

>(dfa_slice nil '(nw))

((ASSIGN (YES 1) (NO 0)) (ASSIGN (INWORD NO) (NW 0))
 (ASSIGN (C GETCHAR))
 (WHILE (<> C EOF)
        (COND
          ((OR (OR (= C " ") (= C "n")) (= C "t"))
           (ASSIGN (INWORD NO)))
          ((ELSE)
           (COND
             ((= INWORD NO) (ASSIGN (INWORD YES))
              (ASSIGN (NW (+ NW 1)))))))
        (ASSIGN (C GETCHAR)))
 (ASSIGN (NW NW)))

>(dfa_slice nil '(nc))

((ASSIGN (NC 0)) (ASSIGN (C GETCHAR))
 (WHILE (<> C EOF) (ASSIGN (NC (+ NC 1))) (ASSIGN (C GETCHAR)))
 (ASSIGN (NC NC)))

>(dfa_slice nil '(inword))

((ASSIGN (YES 1) (NO 0)) (ASSIGN (INWORD NO)) (ASSIGN (C GETCHAR))
 (WHILE (<> C EOF)
        (COND
          ((OR (OR (= C " ") (= C "n")) (= C "t"))
           (ASSIGN (INWORD NO)))
          ((ELSE) (COND ((= INWORD NO) (ASSIGN (INWORD YES))))))
        (ASSIGN (C GETCHAR))))

>(dfa_slice nil '(c))

((ASSIGN (C GETCHAR)) (WHILE (<> C EOF) (ASSIGN (C GETCHAR))))
```

Figure 7.6: Slices of the algorithm in Figure 7.5

```
>(prpr)

((ASSIGN (X 0) (Y 0) (Z 0))
 (ACTIONS (A B) (A (ASSIGN (X 1) (Z 1)) (CALL C 0))
          (B (ASSIGN (X 2) (Y Z) (Z X)) (CALL Z 0))
          (C (ASSIGN (X 3) (Z Y)) (CALL Z 0))))

>(dfa_slice nil '(x))

((ACTIONS (A B) (A (CALL C 0)) (B (ASSIGN (X 2)) (CALL Z 0))
                (C (ASSIGN (X 3)) (CALL Z 0))))

>(dfa_slice nil '(y))

((ASSIGN (Y 0) (Z 0))
 (ACTIONS (A B) (A (CALL C 0)) (B (ASSIGN (Y Z)) (CALL Z 0))
                (C (CALL Z 0))))

>(dfa_slice nil '(z))

((ASSIGN (X 0) (Y 0))
 (ACTIONS (A B) (A (CALL C 0)) (B (ASSIGN (Z X)) (CALL Z 0))
                (C (ASSIGN (Z Y)) (CALL Z 0))))
>
```

Figure 7.7: Action Systems of WSL

```
>(dfa_slice nil '(x y))

((ASSIGN (Y 0) (Z 0))
 (ACTIONS (A B) (A (CALL C 0)) (B (ASSIGN (X 2) (Y Z)) (CALL Z 0))
          (C (ASSIGN (X 3)) (CALL Z 0))))

>(dfa_slice nil '(x z))

((ASSIGN (X 0) (Y 0))
 (ACTIONS (A B) (A (CALL C 0)) (B (ASSIGN (X 2) (Z X)) (CALL Z 0))
          (C (ASSIGN (X 3) (Z Y)) (CALL Z 0))))

>(dfa_slice nil '(y z))

((ASSIGN (X 0) (Y 0) (Z 0))
 (ACTIONS (A B) (A (CALL C 0)) (B (ASSIGN (Y Z) (Z X)) (CALL Z 0))
          (C (ASSIGN (Z Y)) (CALL Z 0))))

>(dfa_slice nil '(x y z))

((ASSIGN (X 0) (Y 0) (Z 0))
 (ACTIONS (A B) (A (CALL C 0)) (B (ASSIGN (X 2) (Y Z) (Z X)) (CALL Z 0))
          (C (ASSIGN (X 3) (Z Y)) (CALL Z 0))))

>
```

Figure 7.8: Action Systems of WSL (cont).

executed. This action system is non deterministic because it can execute either action A or B as the first action. Action A calls C which in turn calls Z and finishes the execution of the action system. Action B eventually calls Z and also finishes execution of the action system.

The slice on x is

```
Actions: (A B)
  A: call C
  B: x:=2;
  C: x:=3;
end
```

This shows that the assignment to x in the action A does not contribute to the final value of x. The assignments to x on actions B and C will be the last ones to be executed so they have to be included on the slice.

The slice on y is

```
< y:=0; z:=0; >
Actions: (A B)
  B: y:=z;
end
```

The assignment z:=0 appears in the slice because if the first action chosen to execute is action B, we will need the value of z to calculate y; on the other hand if action A is chosen, then the value of y before the action system is needed because neither A nor C have any assignment to y.

The slice on z is

```
< x:=0; y:=0; >
```

```
Actions: (A B)
  A: call C
  B: z:=x;
  C: z:=y;
end
```

The initial assignments to x or y are needed to compute z, depending on whether the chosen action is A or B, respectively.

The slice on x and y is

```
< y:=0; z:=0; >
Actions: (A B)
  A: call C
  B: < x:=2; y:=z; >
  C: x:=3;
end
```

This slice results from the merging of the two slices.

The slice on x and z is

```
< x:=0; y:=0; >
Actions: (A B)
  A: call C
  B: < x:=2; z:=x; >
  C: < x:=3; z:=y; >
end
```

This slice results from the merging of the two slices.

The slice on y and z is

```
< x:=0; y:=0; z:=0; >
Actions: (A B)
  A: call C
  B: < y:=z; z:=x; >
  C: < z:=y; >
end
```

This slice results from the merging of the two slices too.

The slice on all the variables

```
< x:=0; y:=0; z:=0; >
Actions: (A B)
  A: call C
  B: < x:=2; y:=z; z:=x; >
  C: < x:=3; z:=y; >
end
```

Finally the slice taken for all the variables should be the entire program, unless there is some 'inefficient statements' or dead code. In this program the assignments in action A are inefficient statements because there is always a call to C which executes some assignments to the same variables.

7.7 Conclusions

The result of testing the tool on these and other examples have provided confidence that it behaves as expected. Examples that include different types statatements have been presented (e.g. while, for, assignment, if, action systems ...). The incremental part of the slicer has not been shown because it would yield the same result, and the great strength of the incremental

methods lies in speed and amount of recalculation, both of which have not been measured empirically yet.

Chapter 8

Conclusions

The slicer has been tested on some small programs and has worked successfully, as shown in Chapter 7. This means that there is no experience with real (i.e. large with complex control flow) programs yet. Some demonstrations have been given to other people involved in the ReForm project and the suggestions, comments and technical discussions have been most useful to solve problems, enhance facilities or modify other technical aspects.

It is considered that it will be a simple activity to add the slicer to the ReForm system. The interface with the ReForm database has been explored and works satisfactorily. It remains to add suitable 'buttons' to the Xma user interface to invoke the slicer. The slicer features an incremental algorithm which provides the necessary performance to the Xma user.

8.1 Method

There were several methods to choose from and Bergeretti and Carré was selected for the reasons outlined in Section 5.1. Due to the use of a prototyping development method, had the analysis method not worked adequately, it could have been possible to change it for another

one. The prototype showed that the first two stages in the development (see Section 4.4.2) were reasonably easy to implement (this was expected owing to the characteristics of the method).

Action systems took most of the implementation time, and in fact there are some special cases (i.e. recursion for regular action systems) which have to be implemented yet. It was found that action systems are not a construct well suited for a parse tree analyser because it expects to be able to calculate the information for the current node from its sons and the node itself only. If this is not the case, and information has to be gathered from other places in the tree, the general pattern of traversal doesn't work and a different algorithm has to be used. When a method to perform interprocedural analysis for constructs like procedures is added for the next stage, action systems will be reconsidered because actions are recursive parameterless procedures and a solution which accommodates both actions and procedures will be required.

8.2 Future Directions

The implementation of the static analyser has not finished yet, and there is wide scope for further enhancements:

- Analyse the rest of the low-level WSL constructs according to the stages described in Section 4.4. An important investigation has to be carried out in order to decide how to perform the interprocedural analysis of WSL. The techniques suggested at the end of Bergeretti and Carré's paper [10] may prove useful.
- Devise a method for analysing high-level WSL constructs when they have been defined.
- Perform tests on large programs to measure the efficiency of the implementation. If necessary fine-tune some sections and create a compiled version. It may be necessary to remove the data flow information on the leaves and on the nodes of the lower levels of the parsing tree if more memory is needed. Of course this will imply that slicing

on some statements will take a longer time than usual because of the recalculation for these nodes.

- Implement the rest of the tools i.e. cross-referencer, modulariser, call-graph displayer, using the same data flow information already stored in the database.

- Integrate the slicer with the more user-friendly browser interface and use the pretty-printer to display the slice. This should be very simple because the protocol used in other parts of the Maintainer's Assistant to communicate with the X front-end can be used with almost no modifications on the code of the slicer.

Appendix A

WSL Syntax

In the following tables the WSL syntax is presented, they include all the constructions which are been used by the Maintainer's Assistant. The data flow analyser and slicer can only work, at the present moment, with a reduced subset of all the different statements.

The meaning of the entries is as follows:

Number This is the number of the type number that is passed to the pretty-printer as a more efficient alternative to passing the actual type of the object. This both reduces the amount of information which needs to be passed, and also speeds up the process of finding the form of the pretty-printed version. It is also used as an index in Section 4.4 to reference the entries of the Table.

Name This is the name of the item.

Generic Type This is the "parent" type of the given type. For example, "Skip" is a type of statement and "Number" is a type of expression.

Leading Token This is either "yes" or "no" if and only if the type of the item is the first part of the printed form. For example, an "Assign" statement begins with the word

"Assign", but an assignment does not begin with the word "Assignment" (or any other word).

Minimum Size This is the least number of components that the type can have. Examples are an assignment which must have at least two (in fact only two) components and a "For" loop which must have at least five components, whereas a list of variables can have any number.

Component Types This holds the types of components of the given type (if there are any). For example, the components of an assignment are a variable and and expression. If there is an unlimited number of components for a given item, any additional components must have the same type as the last component. For example, a "For" loop must have a (loop) variable, three expressions (for the initial, final and step values of the loop) and it can have any number of statements in it. There can be more than one item of the last type if the entry "Component Types" finishes with "...". For example, a "Call" statement can only have a single name and a single number as its components, whereas a "For" loop can have any number of statements in it.

Num	Name	Generic Type	Leading Token	Min Size	Component Types
1	Thing	—	No	0	
2	A_List	Thing	No	0	Thing ...
3	Symbol	Thing	No	0	
4	Name	Thing	No	0	
5	Statement	Thing	Yes	0	...
6	Expression	Thing	Yes	0	Expression ...
7	Condition	Thing	Yes	0	Condition ...
8	Assignment	Thing	No	2	Assd_Var Expression
9	Guarded	Thing	No	2	Condition Statement ...
10	Action	Thing	No	2	Name Statement ...
11	Definition	Thing	Yes	0	Name Variables Variables Statement ...
12	$Statement$	Statement	Yes	0	
13	$Expn$	Expression	Yes	0	—
14	Var	Expression	Yes	0	—
15	$Condition$	Condition	Yes	0	—
16	$Name$	Name	No	0	
17	Statements	A_List	No	1	Statement ...
18	Expressions	A_List	No	0	Expression ...
19	Variables	A_List	No	0	Variable ...
20	Assd_Vars	A_List	No	0	Assd_Var ...
21	Assignments	A_List	No	1	Assignment ...
22	Guardeds	A_List	No	1	Guarded ...
23	Names	A_List	No	1	Name ...
24	!L	Expression	Yes	1	A_List
25	Number	Expression	No	0	—
26	String	Expression	No	0	—
27	Variable	Expression	No	0	—
28	Assd_Var	Variable	No	0	—
29	Aref	Variable	Yes	2	Variable Expression
30	Abort	Statement	Yes	0	

Num	Name	Generic Type	Leading Token	Min Size	Component Types
31	Actions	Statement	Yes	2	Names Action ...
32	Array	Statement	Yes	2	Assd_Var Expression
33	Assert	Statement	Yes	1	Condition
34	Assign	Statement	Yes	1	Assignment ...
35	Call	Statement	Yes	2	Name Number
36	Comment	Statement	Yes	1	String
37	Cond	Statement	Yes	1	Guarded ...
38	D_If	Statement	Yes	1	Guarded ...
39	D_Do	Statement	Yes	1	Guarded ...
40	Exit	Statement	Yes	1	Number
41	Floop	Statement	Yes	1	Statement ...
42	For	Statement	Yes	5	Assd_Var Expr. Expr. Expr. Statement ...
43	!Xp	Statement	Yes	2	Name Expressions
44	!P	Statement	Yes	3	Name Expressions Assd_Vars
45	Proc_Call	Statement	Yes	3	Name Expressions Variables
46	Skip	Statement	Yes	0	
47	Var	Statement	Yes	2	Assignments Statement ...
48	Where	Statement	Yes	2	Statements Definition ...
49	While	Statement	Yes	2	Condition Statement ...
50	Proc	Definition	Yes	4	Name Variables Variables Statement ...
51	Funct	Definition	Yes	3	Name Variables Expression
52	B_Funct	Definition	Yes	3	Name Variables Condition
53	+	Expression	Yes	2	Expression ...
54	-	Expression	Yes	2	Expression
55	*	Expression	Yes	2	Expression ...
56	/	Expression	Yes	2	Expression
57	**	Expression	Yes	2	Expression
58	Min	Expression	Yes	2	Expression ...
59	Max	Expression	Yes	2	Expression ...
60	Div	Expression	Yes	2	Expression

Num	Name	Generic Type	Leading Token	Min Size	Component Types
61	Mod	Expression	Yes	2	Expression
62	If	Expression	Yes	3	Condition Expression
63	Funct_Call	Expression	Yes	2	Name Expressions
64	!F	Expression	Yes	2	Name Expressions
65	Gen_Expr	Expression	Yes	3	Assignments Statements Expression
66	Int	Expression	Yes	1	Expression
67	Frac	Expression	Yes	1	Expression
68	Abs	Expression	Yes	1	Expression
69	Sgn	Expression	Yes	1	Expression
70	True	Condition	No	0	—
71	False	Condition	No	0	—
72	Else	Condition	Yes	0	—
73	=	Condition	Yes	2	Expression
74	<>	Condition	Yes	2	Expression
75	<	Condition	Yes	2	Expression
76	>	Condition	Yes	2	Expression
77	<=	Condition	Yes	2	Expression
78	>=	Condition	Yes	2	Expression
79	==	Condition	Yes	2	Expression
80	Even?	Condition	Yes	1	Expression
81	Odd?	Condition	Yes	1	Expression
82	True?	Condition	Yes	1	Expression
83	False?	Condition	Yes	1	Expression
84	And	Condition	Yes	1	Condition ...
85	Or	Condition	Yes	1	Condition ...
86	Not	Condition	Yes	1	Condition
87	B_Funct_Call	Condition	Yes	2	Name Expressions
88	!C	Condition	Yes	2	Name Expressions
89	Gen_Cond	Condition	Yes	3	Assignments Statements Condition
90	Empty	Expression	Yes	0	—

Num	Name	Generic Type	Leading Token	Min Size	Component Types
91	Cons	Expression	Yes	2	Expression
92	Append	Expression	Yes	2	Expression
93	Intersection	Expression	Yes	2	Expression ...
94	Union	Expression	Yes	2	Expression ...
95	Set_Diff	Expression	Yes	2	Expression
96	List	Expression	Yes	1	Expression ...
97	Hd	Expression	Yes	1	Expression
98	Tl	Expression	Yes	1	Expression
99	Length	Expression	Yes	1	Expression
100	Reverse	Expression	Yes	1	Expression
101	Empty?	Condition	Yes	1	Expression
102	Non_Empty?	Condition	Yes	1	Expression
103	Member?	Condition	Yes	2	Expression
104	Some_Member?	Condition	Yes	2	Expression
105	Any_Member?	Condition	Yes	2	Expression
106	Subset?	Condition	Yes	2	Expression
107	Same?	Condition	Yes	2	Expression
108	Push	Statement	Yes	2	Expression Assd_Var
109	Pop	Expression	Yes	1	Assd_Var
110	A_Size	Expression	Yes	1	Variable
111	[_S+_]	Expression	Yes	2	Expression ...
112	Spec	Statement	Yes	3	Assd_Vars Assd_Vars Condition
113	Assn_Spec	Statement	Yes	2	Assd_Vars Condition
114	Old	Variable	Yes	1	Variable
115	%N	Expression	Yes	0	—
116	%Z	Expression	Yes	0	—
117	%Q	Expression	Yes	0	—
118	%R	Expression	Yes	0	—
119	Map	Expression	Yes	4	Name Name Variable Expression
120	Reduce	Expression	Yes	4	Name Name Variable Expression
121	Set	Expression	Yes	2	Expression Condition
122	For_All	Condition	Yes	2	Variable Condition
123	Exists	Condition	Yes	2	Variables Condition

References

[1] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. **Data structures and algorithms**, chapter 3, pp. 75–106. Addison-Wesley, Reading, Massachusetts, 1983.

[2] Aho, A. V., Sethi, R., and Ullman, J. D. **Compilers: Principles, Techniques, and Tools**, chapter 10, pp. 585–722. Addison-Wesley, Reading, Massachusetts, 1986.

[3] Allen, F. E. and Cocke, J. **A program data flow analysis procedure.** *Communications of the ACM*, vol. 19, num. 3, pp. 137–147, March 1976.

[4] Ambras, J. and O'Day, V. **MicroScope: a knowledge-based programming environment.** *IEEE Software*, vol. 5, num. 3, pp. 50–58, March 1988.

[5] ANSI/IEEE. **IEEE Standard Glossary of Software Engineering Terminology.** ANSI/IEEE Standard 729, 1983.

[6] Arsac, J. J. **Syntactic source to source transformations and program manipulation.** *Communications of the ACM*, vol. 22, num. 1, pp. 43–53, January 1979.

[7] Barth, J. M. **A practical interprocedural data flow analysis algorithm.** *Communications of the ACM*, vol. 21, num. 9, pp. 724–736, September 1978.

[8] Ben Arfa, L., Mili, A., and Sekhri, L. **An empirical study of software maintenance.** In *Proceedings Conference on Software Maintenance 1991* [46], pp. 52–58.

[9] Bennett, K. H., Cornelius, B. J., Munro, M., and Robson, D. J. **Approaches to program comprehension**. *Journal of Systems and Software*, vol. 14, num. 1, pp. 79–84, 1991.

[10] Bergeretti, J.-F. and Carré, B. A. **Information-flow and data-flow analysis of while-programs**. *ACM Transactions on Programming Languages and Systems*, vol. 7, num. 1, pp. 37–61, January 1985.

[11] Boehm, B. W. **Software Engineering**. *IEEE Transactions on Computers*, vol. 25, num. 12, pp. 1226–1241, December 1976.

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

[13] Brooks, F. P. **No silver bullet: Essence and accidents of Software Engineering**. *Computer*, vol. 20, num. 4, pp. 10–19, 1987.

[14] Brown, A. W. and McDermid, J. A. **Learning from IPSE's mistakes**. *IEEE Software*, vol. 9, num. 3, pp. 23–28, March 1992.

[15] Burke, M. G. **An interval-based approach to exhaustive and incremental interprocedural data-flow analysis**. *ACM Transactions on Programming Languages and Systems*, vol. 12, num. 3, pp. 341–395, July 1990.

[16] Burke, M. G. and Ryder, B. G. **A critical analysis of incremental iterative data flow analysis algorithms**. *IEEE Transactions on Software Engineering*, vol. 16, num. 7, pp. 723–728, July 1990.

[17] Buxton, J. N. **Software engineering — 20 years on and 20 years back**. *Journal of Systems and Software*, vol. 13, num. 2, pp. 153–155, 1990.

[18] Carey, J. M. **Prototyping: alternative systems development methodology**. *Information & Software Technology*, vol. 32, num. 3, 1990.

[19] Chapin, N. **Software maintenance life cycle**. In *Proceedings Conference on Software Maintenance 1988* [44], pp. 6–13.

[20] Chikofsky, E. J. and Cross, J. H. **Reverse engineering and Design recovery: a Taxonomy**. *IEEE Software*, vol. 7, num. 1, pp. 13–18, 1990.

[21] Dart, S. A., Ellison, R. J., Feiler, P. H., and Habermann, A. N. **Software development environments**. *IEEE Computer*, vol. 20, num. 11, pp. 18–28, 1987.

[22] Darwin, I. F. **Checking C programs with LINT**. O'Reilly & Associates, Sebastopol, California, 1988.

[23] Desclaux, C. and Ribault, M. **MACS: A K.A.D.M.E.** In *Proceedings Conference on Software Maintenance 1991* [46], pp. 2–12.

[24] Dowson, M. **Integrated project support with ISTAR**. *IEEE Software*, vol. 4, num. 6, pp. 6–15, 1987.

[25] Ejiogu, L. O. **Software Engineering with Formal Metrics**. McGraw-Hill, Maidenhead, Berkshire, UK, 1991.

[26] Felician, L. and Zalateu, G. **Validating Halstead's theory for Pascal programs**. *IEEE Transactions on Software Engineering*, vol. 15, num. 12, pp. 1630–1632, December 1989.

[27] Fenton, N. E. **Software Metrics — A Rigorous Approach**. Chapman & Hall, London, 1991.

[28] Ferrante, J., Ottenstein, K., and Warren, J. **The program dependence graph and its use in optimization**. *ACM Transactions on Programming Languages and Systems*, vol. 9, num. 3, pp. 319–349, March 1987.

[29] Fischer, C. N. and LeBlanc, Jr., R. J. **Crafting A Compiler**, chapter 16, pp. 609–680. Benjamin/Cummings, Menlo Park, California, 1988.

[30] Foster, J. **Program lifetime: a vital statistic for maintenance**. In *Proceedings Conference on Software Maintenance 1991* [46], pp. 98–103.

[31] Gadd, R. J. **ReForm — from assembler to Z using formal transformations**. In **Proceedings of the Fourth Software Maintenance Workshop**, Durham, UK, September 1990.

[32] Gallagher, K. B. **Using program slicing in software maintenance**. PhD thesis, Univ. Maryland, Baltimore, December 1989.

[33] Gallagher, K. B. **Surgeon's Assistant limits side effects**. *IEEE Software*, vol. 7, num. 5, pp. 64, May 1990.

[34] Gallagher, K. B. and Lyle, J. R. **Using program slicing in software maintenance**. *IEEE Transactions on Software Engineering*, vol. 17, num. 8, pp. 751–761, August 1991.

[35] Georges, M. **MACS: Maintenance Assistance Capability for Software**. In **Proceedings of the Fourth Software Maintenance Workshop**, Durham, UK, September 1990.

[36] Glass, R. and Noiseux, R. **Software Maintenance Guidebook**. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[37] Goldberg, A. **Smalltalk–80 — The Interactive Programming Environment**. Addison-Wesley, Reading, Massachusetts, 1984.

[38] Gopal, R. **Dynamic program slicing based on dependence relations**. In *Proceedings Conference on Software Maintenance 1991* [46], pp. 191–200.

[39] Halstead, M. H. **Elements of Software Science**. Elsevier North-Holland, New York, 1979.

[40] Hecht, M. S. and Ullman, J. D. **A simple algorithm for global data flow analysis problems**. *SIAM Journal of Computing*, vol. 4, num. 4, pp. 519–532, December 1975.

[41] Henry, S. and Wake, S. **Predicting maintainability with software quality metrics**. *Software Maintenance: Research & Practice*, vol. 3, num. 3, pp. 129–143, 1991.

[42] Horwitz, S., Reps, T., and Binkley, D. **Interprocedural slicing using dependence graphs**. *ACM Transactions on Programming Languages and Systems*, vol. 12, num. 1, pp. 26–60, January 1990.

[43] Hsieh, C. S. **Slice, chunk, and dataflow anomaly as Datalog rules.** *Journal of Systems and Software*, vol. 16, pp. 197–203, 1991.

[44] IEEE. **Proceedings Conference on Software Maintenance 1988**, Phoenix, Arizona, October 24–27.

[45] IEEE. **Proceedings Conference on Software Maintenance 1989**, Miami, Florida, October 16–19.

[46] IEEE. **Proceedings Conference on Software Maintenance 1991**, Sorrento, Italy, October 15–17.

[47] IEEE. **Special number dedicated to Software Engineering Environments.** *IEEE Software*, vol. 5, num. 2, March 1988.

[48] Ince, D. **Software metrics: introduction.** *Information & Software Technology*, vol. 32, num. 4, pp. 297–303, 1990.

[49] Ivie, E. L. **The Programmers Workbench — a machine for software development.** *Communications of the ACM*, vol. 20, num. 10, pp. 746–753, 1977.

[50] Jiang, J., Zhou, X., and Robson, D. J. **Program slicing for C —The problems in implementation.** In *Proceedings Conference on Software Maintenance 1991* [46], pp. 82–190.

[51] Kafura, D. and Reddy, G. R. **The use of software complexity metrics in software maintenance.** *IEEE Transactions on Software Engineering*, vol. 13, num. 3, pp. 335–343, March 1987.

[52] Keables, J., Roberson, K., and von Maryhauser, A. **Data flow analysis and its application to software maintenance.** In *Proceedings Conference on Software Maintenance 1988* [44], pp. 335–347.

[53] Kennedy, K. **A survey of data flow analysis techniques.** In Muchnick, S. and Jones, N., editors, **Program Flow Analysis: Theory and Applications**, pp. 5–54. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[54] Korel, B. and Laski, J. **Dynamic program slicing**. *Information Processing Letters*, vol. 29, num. 10, pp. 155–163, October 1988.

[55] Leach. **Software metrics and software maintenance**. *Journal of Software Maintenance*, vol. 2, num. 2, June 1990.

[56] Lehman, M. M. **Programs, life cycles, and laws of software evolution**. *Proceedings IEEE*, vol. 68, num. 9, pp. 1060–1076, 1980.

[57] Lehman, M. M. and Belady, L. **Program Evolution. Processes of software Change**. Academic Press, London, UK, 1985.

[58] Leung, H. K. N. and Reghbati, H. K. **Comments on 'Program Slicing'**. *IEEE Transactions on Software Engineering*, vol. 13, num. 12, pp. 1370–1371, December 1987. This is a correction to the article by Weiser, see [103].

[59] Lientz, B. and Swanson, E. B. **Software Maintenance Management**. Addison-Wesley, Reading, Massachusetts, 1980.

[60] Lientz, B., Swanson, E. B., and Tompkins, G. E. **Characteristics of application software maintenance**. *Communications of the ACM*, vol. 21, num. 6, pp. 466–471, June 1978.

[61] Martin, J. and McClure, C. **Software Maintenance: The Problem and its Solutions**. Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[62] Maude, T. and Willis, G. **Rapid prototyping: the management of software risk**. Pitman, London, 1991.

[63] McCabe, T. J. **A complexity measure**. *IEEE Transactions on Software Engineering*, vol. 2, num. 4, pp. 308–320, December 1976.

[64] Mehndiratta, B. and Grover, P. S. **Software metrics — an experiment analysis**. *ACM SIGPLAN Notices*, vol. 25, num. 2, pp. 35–41, 1990.

[65] Moreton, R. **Analysis and results from a maintenance survey**. In **Proceedings of the Second Software Maintenance Workshop**, Durham, UK, September 1988.

[66] Moynihan, V. D. and Wallis, P. J. L. **The design and implementation of a high-level language converter**. *Software: Practice & Experience*, vol. 21, num. 4, pp. 391–400, April 1991.

[67] National Computing Centre, editor. **The STARTS Guide**, volume 1. NCC Publications, Oxford Road, Manchester M1 7ED, UK, second edition, 1987.

[68] Osterweil, L. **Software processes are software too**. In **Proceedings of the Ninth International Conference on Software Engineering**, pp. 2–13, Monterey, California, March 30 – April 2 1987.

[69] Osterweil, L. J. **Using data flow tools in software engineering**. In Muchnick, S. and Jones, N., editors, **Program Flow Analysis: Theory and Applications**, pp. 237–263. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[70] Osterweil, L. J. and Fosdick, L. D. **DAVE — a validation, error detection and documentation system for FORTRAN programs**. *Software: Practice & Experience*, vol. 6, num. 3, pp. 473–486, 1976.

[71] Ott, L. and Thuss, J. **The relationship between slices and module cohesion**. In **International Conference on Software Engineering**, May 1989.

[72] Patkau, B. H. **A foundation for software maintenance**. Master's thesis, Univ. of Toronto, Canada, December 1983.

[73] Pollock, L. L. and Soffa, M. L. **An incremental version of iterative data flow analysis**. *IEEE Transactions on Software Engineering*, vol. 15, num. 12, pp. 1537–1549, December 1989.

[74] Rajlich, V., Damaskinos, N., Linos, P., and Khorshid, W. **VIFOR: A tool for software maintenance**. *Software: Practice & Experience*, vol. 20, num. 1, pp. 67–77, January 1990.

[75] Rajlich, V. and Khorshid, W. **VIPEG: A generator of environments for software maintenance**. In **Proceedings Fourteenth Annual International Computer**

Software & Applications Conference, pp. 471–475. IEEE Computer Society Press, October 29 – November 2 1990.

[76] Ramamoorthy, B. and Melton, A. **A sysnthesis of software science measures and the cyclomatic number.** *IEEE Transactions on Software Engineering*, vol. 14, num. 8, pp. 1116–1121, August 1988.

[77] Ramamoorthy, C. V. and Ho, S. F. **Testing large software with automated software evaluation systems.** *IEEE Transactions on Software Engineering*, vol. 1, num. 1, pp. 46–58, 1975.

[78] Reiss, S. P. **Graphical program development with PECAN Program Development Systems.** *ACM SIGPLAN Notices*, vol. 19, num. 5, pp. 30–41, 1984.

[79] Reiss, S. P. **PECAN: Program Development Systems that support multiple views.** *IEEE Transactions on Software Engineering*, vol. 11, num. 3, pp. 276–285, 1985.

[80] Ritchie, D. M., Johnson, S. C., Lesk, M. E., and Kernighan, B. W. **The C programming language.** *Bell Systems Technical Journal*, vol. 57, num. 6, pp. 1991–2020, 1978.

[81] Rochkind, M. J. **The Source Code Control System.** *IEEE Transactions on Software Engineering*, vol. 1, num. 4, pp. 255–265, 1975.

[82] Rombach, H. D. **A controlled experiment on the impact of software structure on maintainability.** *IEEE Transactions on Software Engineering*, vol. 13, num. 3, pp. 344–354, March 1987.

[83] Rosen, B. K. **High level data flow analysis.** *Communications of the ACM*, vol. 20, num. 10, pp. 712–724, October 1977.

[84] Royce, W. W. **Managing the development of large software systems: concepts and techniques.** In **Proceedings WESTCON**, pp. 113–121. IEEE Computer Society Press, 1970.

[85] Rubin, L. F. **Syntax-directed pretty printing.** *IEEE Transactions on Software Engineering*, vol. 9, num. 2, pp. 119–127, 1983.

[86] Ryder, B. G. and Paull, M. C. **Incremental data flow analysis algorithms.** *ACM Transactions on Programming Languages and Systems*, vol. 10, num. 1, pp. 1–50, 1988.

[87] Schneidewind, N. F. **Setting maintenance quality objectives and prioritizing maintenance work by using quality metrics.** In *Proceedings Conference on Software Maintenance 1991* [46], pp. 240–249.

[88] Schneidewind, N. F. **Validating software metrics: producing quality discriminators.** In **Proceedings of International Symposium on Software Reliability Engineering**, pp. 225–232, May 17–18 1991.

[89] Sommerville, I. **Software Engineering.** Addison-Wesley, Reading, Massachusetts, third edition, 1989.

[90] Swanson, E. F. **The dimensions of maintenance.** In **Proceedings of the Second International Conference on Software Engineering**, pp. 492–497, October 1976.

[91] Tarjan, R. E. **Testing flow graph reducibility.** *Journal of Computer System Science*, vol. 9, num. 3, pp. 355–365, December 1974.

[92] Teitelman, W. **A tour through Cedar.** *IEEE Transactions on Software Engineering*, vol. 11, num. 3, pp. 285–302, 1985.

[93] Teitelman, W. and Masinter, L. **The Interlisp programming environment.** *IEEE Computer*, vol. 14, num. 4, pp. 25–34, 1981.

[94] van Zuylen, H., editor. **The REDO Handbook.** John Wiley & sons, Chichester, UK, 1992.

[95] Ward, M. **Proving Program Refinements and Transformations.** PhD thesis, Oxford University, 1988.

[96] Ward, M. **Transforming a program into a specification.** Computer Science Technical Report 88-1, Durham University, January 1988.

[97] Ward, M. **The syntax and semantics of the wide spectrum language WSL.** Computer science technical report, Durham University, December 1991.

[98] Ward, M., Munro, M., and Calliss, F. W. **The Maintainer's Assistant.** In *Proceedings Conference on Software Maintenance 1989* [45], pp. 307–315.

[99] Warren Jr, H. S. **A modification of Warshall's algorithm for the transitive closure of binary relations.** *Communications of the ACM*, vol. 18, num. 4, pp. 218–220, April 1975.

[100] Warshall, S. **A theorem on boolean matrices.** *Journal of the ACM*, vol. 9, num. 1, pp. 11–12, January 1962.

[101] Wasserman, A. I. **Software engineering environments.** *Advances In Computers*, vol. 22, 1983.

[102] Weiser, M. **Programmers use slices when debugging.** *Communications of the ACM*, vol. 25, num. 7, pp. 446–452, July 1982.

[103] Weiser, M. **Program Slicing.** *IEEE Transactions on Software Engineering*, vol. 10, num. 4, pp. 352–357, July 1984. See [58] for a correction.

[104] Wilde, N., Huitt, R., and Huitt, S. **Dependency analysis tools — reusable components for software maintenance.** In *Proceedings Conference on Software Maintenance 1989* [45].

[105] Yang, H. **The supporting environment for a reverse engineering system — The Maintainer's Assistant.** In *Proceedings Conference on Software Maintenance 1991* [46], pp. 13–22.

[106] Yau, S. S. and Collofello, J. S. **Some stability measures for software maintenance.** *IEEE Transactions on Software Engineering*, vol. 6, num. 6, pp. 545–552, November 1980.

[107] Yau, S. S., Nicholl, R. A., Tsai, J. J.-P., and Liu, S. S. **An integrated life-cycle model for software maintenance.** *IEEE Transactions on Software Engineering*, vol. 14, num. 8, pp. 1128–1144, August 1988.